Comparison of different SQL implementations

The goal of this page — which is a work in progress — is to gather information relevant for people who are porting SQL from one product to another and/or are interested in possibilities and limits of 'cross-product' SQL.

The following tables compare how different DBMS products handle various SQL (and related) features. If possible, the tables also state how the implementations should do things, according to the SQL standard.

I will only write about subjects that I've worked with personally, or subjects which I anticipate to find use for in the near future. Subjects on which there are no significant implementation variances are not covered. Beta-versions of software are not examined.

I'm sorry about the colors. They are a result of wanting to mark each DBMS differently and at the same time wanting to be relatively nice to printers.

If you have corrections or suggestions, please contact me; even notifications about spelling errors are welcome.

Contents:

- Legend, definitions, and notes
- Features
  - Views
  - Join types/features
- The SELECT statement
  - Ordering result sets
  - Limiting result sets (RANK() / ROW_NUMBER() / LIMIT / TOP / FETCH FIRST)
    - Simple limit
    - Top-n (quota-queries)
    - Limit—with offset, including note about the importance of sorting on unique values
- The INSERT statement
  - Inserting several rows at a time
- Data types
  - BOOLEAN
  - CHAR
  - Date and time types
    - TIMESTAMP
- Functions and operators
  - CHARACTER_LENGTH
  - SUBSTRING
  - REPLACE
  - TRIM
  - LOCALTIMESTAMP
  - Concatenation
- Constraint handling
  - The UNIQUE constraint
- Mixture of type and operations
  - Automatic key generation (IDENTITY/serial/AUTO_INCREMENT)
- Command line operations
  - Starting the command line interface
  - Getting a list of tables
  - Getting a table description
  - Telling the DBMS to collect statistics
  - Getting a query explanation
  - Turning on query timing
- Other topics
  - Dummy-table use
  - Obtaining DBMS version
- Related work
- Acknowledgments
- TODOs
Legend, definitions, and notes

The following SQL standard and implementations have been examined, if not otherwise stated:

Standard  The latest official version of SQL is SQL:2003.

I don’t have access to the official ISO standard text, but Whitemarsh Information Systems Corporation provides a rather final draft as a zip-archive, containing several files. Most important to this page is the file 5WD-02-Foundation-2003-09.pdf.

SQL:2003 is very new, and the only book currently covering the subject is in German which I was never any good at. Therefore, I also use the following book as reference:


PostgreSQL
PostgreSQL 8.0.0 on Fedora Core Linux.

DB2
DB2 Universal Database Personal Edition v. 8.1FP7 (AKA v. 8.2) on Fedora Core Linux. Note that there are differences between various DB2 UDB flavors; this page is about DB2 for “LUW” (Linux/Unix/Windows).

MS SQL Server
MS SQL Server 2005 on Windows XP. Microsoft’s SQL implementation is sometimes named Transact-SQL. In this document, I’ll generally write MSSQL as a short-hand for Microsoft’s SQL Server product.

MySQL
MySQL Database Server 5.0.18 on Fedora Core Linux (i.e. MySQL AB’s “classic” DBMS product—not MaxDB).

Oracle
Oracle Database 10g Release 1 Standard Edition on Fedora Core release 1 (Linux). The tables should hold for version 9i, as well (Oracle 10g contains remarkably few improvements/changes in Oracle’s SQL standard compliance).

The products are running with their default settings. This is important for MySQL and MSSQL: Their interpretation of SQL may be changed rather drastically by adjusting certain configuration options, potentially increasing the level of standard compliance. However, such non-default configuration options are not of great value for people writing SQL applications because the developer often cannot rely on non-default configuration settings.

Features

Views

Standard  Views are part of the standard, and they may be updated, as long as it ‘makes sense’.

SQL:2003 has a rather complicated set of rules governing when a view is updatable, basically saying that a view is updatable, as long as the update-operation translates into an unambiguous change.

SQL-92 was more restrictive, specifying that updatable views cannot be derived from more than one base table.

PostgreSQL Has views. Breaks that standard by not allowing updates to views; offers the non-standard ‘rules’-system as a work-around.

DB2 Conforms to at least SQL-92.
MSSQL Conforms to at least SQL-92.
MySQL Conforms to at least SQL-92.
Oracle Conforms to at least SQL-92.

Peter Gulutzan has written an article about the implementation of views in three major products.

Join types and features

All the DBMSes support basic INNER JOINs, but vary in their support for other join types.

In the following feature chart, a ✅ means yes; an empty table cell means no.

<table>
<thead>
<tr>
<th>Join type/feature</th>
<th>PostgreSQL</th>
<th>DB2</th>
<th>MSSQL</th>
<th>MySQL</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural joins (only tested: NATURAL LEFT JOIN) USING-clause</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
</tbody>
</table>
FULL joins\(^1\) (tested: `SELECT...FULL JOIN...ON...=...`) ✔ ✔ ✔ ✔ ✔ ✔

Explicit CROSS JOIN (cartesian product) ✔ ✔ ✔ ✔ ✔ ✔

Remarks:

1. Note that FULL joins may be emulated with a union of a left and a right join.

The SELECT statement

Ordering result sets

Standard

The SQL-standard states that relations are unordered, but result sets may be ordered when returned to the user through a cursor:

```sql
DECLARE cursornname CURSOR FOR
    SELECT ... FROM ...
    ORDER BY column_name1, column_name2, ...
```

As such, the standard doesn't allow ORDER BY anywhere else than in cursor declarations. Special exceptions exist, such as the ORDER BY part of window functions (including `ROW_NUMBER()` OVER... and `RANK()` OVER...).

The standard doesn't specify how NULLs should be ordered in comparison with non-NULL values, except that NULLs are to be considered equal in the ordering, and that NULLs should sort either above or below all non-NULL values.

PostgreSQL

Allows ORDER BY in contexts other than cursor definitions. NULLs are considered **higher** than any non-NULL value.

**DOCUMENTATION**

DB2

Allows ORDER BY in contexts other than cursor definitions. NULLs are considered **higher** than any non-NULL value.

**DOCUMENTATION**

MSSQL

Allows ORDER BY in contexts other than cursor definitions. NULLs are considered **lower** than any non-NULL value.

**DOCUMENTATION**

MySQL

Allows ORDER BY in contexts other than cursor definitions.

NULLs are considered **lower** than any non-NULL value, except if a – (minus) character is added before the column name and ASC is changed to DESC, or DESC to ASC; this minus-before-column-name feature seems undocumented.

**DOCUMENTATION**

Oracle

Allows ORDER BY in contexts other than cursor definitions.

By default, NULLs are considered **higher** than any non-NULL value; however, this sorting behaviour may be changed by adding **NULLS FIRST** or **NULLS LAST** to the ORDER BY expression.

Beware of Oracle's strange treatment of empty strings and NULLs as the same 'value'.

**DOCUMENTATION**

Limiting result sets

Simple limit

Objective: Want to only get \(n\) rows in the result set. Usually only makes sense in connection with an ORDER BY expression.

Note: This is **not** the same as a top-\(n\) query — see next section.

Note also: Some of the queries below may not be legal in all situations, such as in views or sub-queries.

Standard

Non-core Feature ID T611 specifies window functions, of which one is `ROW_NUMBER()` OVER:

```sql
SELECT * FROM ( ...
```
SELECT
  ROW_NUMBER() OVER (ORDER BY key ASC) AS rownumber,
  columns
FROM tablename
) AS foo
WHERE rownumber <= n

If your application is stateful (in contrast to web applications which normally have to be seen as stateless), then you might look at cursors (core feature ID E121) instead. This involves:

- DECLARE cursor-name CURSOR FOR ...
- OPEN cursor-name
- FETCH ...
- CLOSE cursor-name

POSTGRESQL Doesn't support ROW_NUMBER(). Supports cursors (in all contexts, not only in embedded, dynamic SQL).

Alternative to using ROW_NUMBER():

SELECT columns
FROM tablename
ORDER BY key ASC
LIMIT n

Note that LIMIT changes the semantics of SELECT...FOR UPDATE.

DB2 Supports both standards-based approaches.

MSSQL Supports both standards-based approaches.

MSSQL 2000 didn't support ROW_NUMBER(). Instead, a MSSQL 2000-specific syntax was needed:

SELECT TOP n columns
FROM tablename
ORDER BY key ASC

MySQL Doesn't support the standard. Alternative solution:

SELECT columns
FROM tablename
ORDER BY key ASC
LIMIT n

Oracle Supports ROW_NUMBER. Seems to have non-compliant cursor facilities.

As Oracle doesn't allow AS for subquery naming (and doesn't need a subquery-name at all in this case), the standard SQL code above needs to be rewritten slightly:

SELECT * FROM (  
SELECT
  ROW_NUMBER() OVER (ORDER BY key ASC) AS rownumber,
  columns
FROM tablename
)  
WHERE rownumber <= n

Top-n query

Objective: Like the simple limit-query above, but include rows with tie conditions. Thus, the query may return more than n rows.

Some call this a quota-query.

The following examples are based on this table:
SELECT * FROM person ORDER BY age ASC;
+----------+-------------+-----+
|PERSON_ID | PERSON_NAME | AGE |
+----------+-------------+-----+
|    7     | Hilda       | 12  |
|    8     | Bill        | 12  |
|    4     | Joe         | 23  |
|    2     | Veronica    | 23  |
|    3     | Michael     | 27  |
|    9     | Marianne    | 27  |
|    1     | Ben         | 50  |
|   10     | Michelle    | 50  |
|    5     | Irene       | 77  |
|    9     | Marianne    | 77  |
|    1     | Ben         | 77  |
+----------+-------------+-----+

Now, we only want the three \(n=3\) youngest persons displayed, i.e. a result set like this:

<table>
<thead>
<tr>
<th>PERSON_ID</th>
<th>PERSON_NAME</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hilda</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Bill</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Joe</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Veronica</td>
<td>23</td>
</tr>
</tbody>
</table>
+----------+-------------+-----+

Standard

With standard SQL, there are two principal ways to obtain the wanted data:

- **The fast variant:**

  One of the major additions in SQL:2003 is the addition of non-core (i.e. optional) OLAP (online analytic processing) features. If the DBMS supports elementary OLAP (feature ID F611), then the top-\(n\) query may be formulated using a *window function*, such as `RANK()` over:

  ```sql
  SELECT * FROM (SELECT 
                  RANK() OVER (ORDER BY age ASC) AS ranking, 
                  person_id, 
                  person_name, 
                  age 
              FROM person 
           ) AS foo 
WHERE ranking <= 3
```

(Change `ASC` to `DESC` in the position marked *like this* in order to get a top-3 oldest query instead.)

- **The slow variant:**

  If the DBMS doesn't support the elementary OLAP features, then the top-\(n\) solution may be obtained in an alternative way which so slow that it's not a real option in most situations:

  Correlated subquery method, mentioned in the book *Practical Issues in Database Management* (chapter 9: *Quota Queries*) by Fabian Pascal (who, again, quotes Date for the solution):

  ```sql
  SELECT * FROM person AS px
WHERE ( 
      SELECT COUNT(*)
      FROM person AS py 
      WHERE py.age < px.age 
    ) < 3;
```

The query may make more sense if the objective is re-phrased as "Find all persons (px) such that the number of younger, other persons (py) is less than 3".

(Change `<` to `>` in the position marked *like this* in order to get a top-3 oldest query instead.)

In the article *Going To Extremes* by Joe Celko, there is a description of yet another principle for performing quota queries, using *scalar subqueries*. Scalar subqueries are more tedious to write but might yield better performance on your system.
Comparison of different SQL implementations

PostgreSQL Supports the slow standard SQL query variant. In practice, a PostgreSQL-only method should be used, in order to obtain acceptable query performance:

```sql
SELECT *
FROM person
WHERE (age <= (
    SELECT age FROM person
    ORDER BY age ASC
    LIMIT 1 OFFSET 2       -- 2=n-1
) ) IS NOT FALSE
```

(Change <= to >= and ASC to DESC in the positions marked like this in order to get a top-3 oldest query instead.)

**DB2** Supports the fast standard SQL variant.

**MSSQL** Supports the fast standard SQL variant.

MSSQL 2000 supported the slow standard SQL variant. In practice, a MSSQL-only expression had to be used, in order to obtain acceptable query performance:

```sql
SELECT TOP 3 WITH TIES *
FROM person
ORDER BY age ASC
```

(Change ASC to DESC in the position marked like this in order to get a top-3 oldest query instead.)

**MySQL** Supports the slow standard SQL solution. In practice, this MySQL-specific solution should be used, in order to obtain acceptable query performance:

```sql
SELECT *
FROM person
WHERE age <= COALESCE(
    (SELECT age
     FROM person
     ORDER BY age ASC
     LIMIT 1 OFFSET 2    -- 2=n-1
),
    (SELECT MAX(age)
     FROM person)
)
```

(Change <= to >= and ASC to DESC and MAX to MIN in the positions marked like this in order to get a top-3 oldest query instead.)

The offset-value 2 is the result of \( n-1 \) (remember: \( n \) is 3 in these examples).

The second argument to the **COALESCE** call makes the query work in cases where the cardinality of the table is lower than \( n \).

**Oracle** Supports the fast standard SQL variant. However, as Oracle doesn't like "AS ..." after subqueries (and doesn't require naming of subqueries), the query has to be paraphrased slightly:

```sql
SELECT * FROM (SELECT
    RANK() OVER (ORDER BY age ASC) AS ranking,
    person_id,
    person_name,
    age
FROM person
)
WHERE ranking <= 3
```

(Change ASC to DESC in the position marked like this in order to get a top-3 oldest query instead.)
Limit—with offset

Objective: Want to only get \( n \) rows in the result set, and we want the first \( skip \) rows in the result set discarded. Usually only makes sense in connection with an ORDER BY expression.

In the recipes below, basic ordering is ASCending, i.e. lowest-first queries. If you want the opposite, then change ASC->DESC and DESC->ASC at the places emphasized like this.

Standard Non-core Feature ID T611 specifies \textit{window functions}, one of which is \texttt{ROW_NUMBER()} \texttt{OVER}:

\begin{verbatim}
SELECT * FROM ( 
    SELECT ROW_NUMBER() OVER (ORDER BY key \texttt{ASC}) AS rownum, 
    columns 
    FROM tablename 
) AS foo 
WHERE rownum > \texttt{skip} AND rownum <= \texttt{(n+skip)}
\end{verbatim}

Alternatively, you may use a \texttt{cursor} (core feature ID E121), if the programming environment permits it. This involves:

\begin{itemize}
    \item \texttt{DECLARE cursor-name CURSOR FOR ...}
    \item \texttt{OPEN cursor-name}
    \item \texttt{FETCH RELATIVE number-of-rows-to-skip ...}
    \item \texttt{CLOSE cursor-name}
\end{itemize}

\texttt{POSTGRESQL} Doesn't support \texttt{ROW_NUMBER()}. Supports cursors.

Alternative to \texttt{ROW_NUMBER()}:

\begin{verbatim}
SELECT columns 
FROM tablename 
ORDER BY key \texttt{ASC} 
\texttt{LIMIT n OFFSET skip}
\end{verbatim}

\texttt{DB2} Supports both standard approaches.

\texttt{MSSQL} Supports both standard approaches.

\texttt{MSSQL 2000} didn't support \texttt{ROW_NUMBER()}; instead, a MSSQL-specific syntax had to be used:

\begin{verbatim}
SELECT * FROM ( 
    SELECT TOP \( n \) * FROM ( 
        SELECT TOP \( z \) columns -- (z=n+skip) 
        FROM tablename 
        ORDER BY key \texttt{ASC} 
    ) AS FOO ORDER BY key \texttt{DESC} -- (\texttt{FOO} may be anything) 
    ) AS BAR ORDER BY key \texttt{ASC} -- (\texttt{BAR} may be anything)
\end{verbatim}

\texttt{MySQL} Doesn't support the standard approaches. Alternative solution:

\begin{verbatim}
SELECT columns 
FROM tablename 
ORDER BY key \texttt{ASC} 
\texttt{LIMIT n OFFSET skip}
\end{verbatim}

In older versions of MySQL, the LIMIT-syntax is less clear:

\begin{verbatim}
... \texttt{LIMIT \( \lfloor skip \rfloor, n \)}
\end{verbatim}

(i.e. the \texttt{skip} argument is optional).

The old syntax is still supported by later MySQL versions, as the old syntax is widely used.
Oracle Supports **ROW_NUMBER()**. Oracle's cursor support doesn't look standards-compliant.

As Oracle doesn't accept `AS` for subquery naming (and doesn't require naming of subqueries in this case), the standard SQL solution has to be re-written slightly. An other reason for the re-write is that `ROWNUM` is a reserved word in Oracle, with special meaning. The Oracle code becomes:

```
SELECT * FROM (SELECT ROW_NUMBER() OVER (ORDER BY key ASC) AS rn, columns FROM tablename ) WHERE rn > skip AND rn <= (n+skip)
```

**DOCUMENTATION**

Note:

LIMIT/TOP/FIRST queries with offset are often used in a result presentation context: To retrieve only—say—30 rows at a time so that the end-user isn't overwhelmed by the complete result set, but instead is offered a paginated result presentation. In this case, be careful not to (only) sort on a non-unique column.

Consider the following example (where PostgreSQL is used):

```
SELECT * FROM person ORDER BY age ASC;
```

<table>
<thead>
<tr>
<th>person_id</th>
<th>person_name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hilda</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Bill</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Joe</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Veronica</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>Michael</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>Marianne</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>Ben</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Michelle</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Irene</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>Vivian</td>
<td>77</td>
</tr>
</tbody>
</table>

When ordering is performed on the non-unique age-value, ties may occur and it's not guaranteed that the DBMS will fetch the rows in the same order every time.

Instead of the above listing, the DBMS is allowed to return the following display order where Michael and Marianne are displayed in the opposite order compared to above:

```
SELECT * FROM person ORDER BY age ASC;
```

<table>
<thead>
<tr>
<th>person_id</th>
<th>person_name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hilda</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Bill</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Joe</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Veronica</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>Michael</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>Marianne</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>Ben</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Michelle</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Irene</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>Vivian</td>
<td>77</td>
</tr>
</tbody>
</table>

Now, suppose the end-user wants the results displayed five rows at a time. The result set is fetched in two queries where the DBMS happens to sort differently, as above. We will use PostgreSQL's syntax in the example:

```
SELECT * FROM person ORDER BY age ASC LIMIT 5;
```

<table>
<thead>
<tr>
<th>person_id</th>
<th>person_name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hilda</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Bill</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Joe</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Veronica</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>Michael</td>
<td>27</td>
</tr>
</tbody>
</table>

```
SELECT * FROM person ORDER BY age ASC LIMIT 5 OFFSET 5;
```

<table>
<thead>
<tr>
<th>person_id</th>
<th>person_name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Michael</td>
<td>27</td>
</tr>
<tr>
<td>1</td>
<td>Ben</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Michelle</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Irene</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>Vivian</td>
<td>77</td>
</tr>
</tbody>
</table>

Notice that Marianne was not displayed in any of the two split result set presentations.

The problem could be avoided if the result set ordering had been done in a deterministic way, i.e. where the unique person_id value was considered in case of a tie:
SELECT * FROM person ORDER BY age ASC, person_id ASC ...

This is safer than to pray for the DBMS to behave in a predictable way when handling non-unique values.

Note: If the table is updated between parts of the result set pagination, then the user might still get an inconsistent presentation. If you want to guard against this, too, then you should see if use of an insensitive cursor is an option in your application. Use of cursors to paginate result sets usually require that your application is stateful, which is not the case in many web-application settings. Alternatively, you could copy the result set to a table and use the copied table for pagination; with this solution, you might accumulate a vast amount of result-set generated tables if you don't remember to clean up. If you end up playing with such ideas, then it's probably a good idea to start considering if the entire result set may be cached by the application (e.g. in a session if your web application environment provides for sessions).

The INSERT statement

Inserting several rows at a time

Standard An optional SQL feature is row value constructors (feature ID F641). One handy use of row value constructors is when inserting several rows at a time, such as:

```
INSERT INTO tablename
VALUES (0,'foo') , (1,'bar') , (2,'baz');
```

— which can be seen as a shorthand for

```
INSERT INTO tablename VALUES (0,'foo');
```
```
INSERT INTO tablename VALUES (1,'bar');
```
```
INSERT INTO tablename VALUES (2,'baz');
```

PostgreSQL Not supported.
DB2 Supported.
MSSQL Not supported.
MySQL Supported.
Oracle Not supported.

Data types

The BOOLEAN type

Standard The BOOLEAN type is optional (has feature ID T031), which is a bit surprising for such a basic type. However, it seems that endless discussions of how NULL is to be interpreted for a boolean value is holding BOOLEAN from becoming a core type.

The standard says that a BOOLEAN may be one of the following literals:

- TRUE
- FALSE
- UNKNOWN or NULL (unless prohibited by a NOT NULL constraint)

The DBMS may interpret NULL as equivalent to UNKNOWN. It is unclear from the specification if the DBMS must support UNKNOWN, NULL or both as boolean literals. In this author's opinion, you should forget about the UNKNOWN literal in order to simplify the situation and let the normal SQL three-way logic apply.

It's defined that TRUE > FALSE (true larger than false).

PostgreSQL Follows the standard.

Accepts NULL as a boolean literal; doesn't accept UNKNOWN as a boolean literal.

DOCUMENTATION

DB2 Doesn't support the BOOLEAN type.

Judging from various JDBC-documentation, it seems that IBM recommends a CHAR(1) field constrained to values '0' and '1' (and perhaps NULL) as the way to store boolean values.

MSSQL Doesn't support the BOOLEAN type.

Possible alternative type: the BIT type which may have 0 or 1 (or NULL) as value. If you insert an integer value other than these into a field of type BIT, then the inserted value will silently be converted to 1.
Rudy Limeback has some notes about oddities with the MSSQL BIT type.

**DOCUMENTATION**

**MySQL**

Offers a non-conforming BOOLEAN type. MySQL's BOOLEAN is one of many aliases to its TINYINT(1) type.

(Never use TINYINT(1) as the column type if you use JDBC with MySQL and expect to get non-boolean values from it.)

MySQL accepts the literals TRUE and FALSE as aliases to 1 and 0, respectively. However, you may also assign a value of — e.g. — 9 to a column of type BOOLEAN (which is non-conforming).

If you use JDBC with MySQL, then BOOLEAN is the preferred type for booleans: MySQL's JDBC-driver implicitly converts between Java's boolean and MySQL's pseudo-BOOLEAN type.

**DOCUMENTATION**

**Oracle**

Doesn't support the BOOLEAN type.

Judging from various JDBC documentation, it seems that Oracle recommends NUMBER(1) as the way to store boolean values; it's probably wise to constrain such columns to values 0 and 1 (and perhaps NULL).

Warning to JDBC users:
According to the JDBC standard, getBoolean() must convert a SQL-value of NULL to the false Java value. To check if the database-value was really NULL, use wasNull().

**The CHAR type**

For the following section, I have used this test-SQL to try to illuminate differences (unfortunately, even standard SQL as simple as this has to be adjusted for some products):

**Test steps:**

```sql
CREATE TABLE chartest (
    charval1 CHAR(10) NOT NULL,
    charval2 CHAR(10) NOT NULL,
    varcharval VARCHAR(30) NOT NULL
);
INSERT INTO chartest VALUES ('aaa','aaa','aaa');
INSERT INTO chartest
    VALUES ('aaaaaa      ','aaa','aaa'); -- should truncate to 'aaaaaa    '
INSERT INTO chartest
    VALUES ('aaaaaaaaaaaa','aaa','aaa'); -- should raise error
SELECT * FROM chartest; -- should show two rows
DELETE FROM chartest WHERE charval1='aaaaaa';
SELECT * FROM chartest; -- should show one row
SELECT * FROM chartest WHERE charval1=varcharval;
SELECT charval1 || 'X' AS res FROM chartest;
SELECT CHAR_LENGTH(charval1 || charval2) AS res FROM chartest;
SELECT CHAR_LENGTH(charval1) + CHAR_LENGTH(charval2)
    AS res FROM chartest;
```

**Expected results, after CREATE and INSERTS:**

<table>
<thead>
<tr>
<th>CHARVAL1</th>
<th>CHARVAL2</th>
<th>VARCHARVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaa</td>
<td>aaa</td>
<td>aaa</td>
</tr>
<tr>
<td>aaaaaaa</td>
<td>aaa</td>
<td>aaa</td>
</tr>
</tbody>
</table>

DELETE FROM chartest WHERE charval1='aaaaaa';

SELECT * FROM chartest; -- should show one row
CHARVAL1 | CHARVAL2 | VARCHARVAL |
----------|----------|------------|
aaa      | aaa      | aaa        |

SELECT * FROM chartest WHERE charval1=varcharval;
CHARVAL1 | CHARVAL2 | VARCHARVAL |
----------|----------|------------|
aaa      | aaa      | aaa        |
Comparison of different SQL implementations

SELECT charval1 || 'X' FROM chartest AS res;
res
==========
aaa       X

SELECT CHAR_LENGTH(charval1 || charval2) AS res FROM chartest;
res
==========
20

SELECT character_length(charval1) + character_length(charval2)
AS res
FROM chartest;
res
==========
20

Actual results.

Standard

- Return with an exception state if the inserted string is too long, unless the characters exceeding the limit are all spaces.
- Pad CHAR columns with spaces if the inserted string is shorter than the specified CHAR-length.
- Pad with trailing spaces as needed when casting or comparing to other string-like values (e.g. VARCHARs).

PostgreSQL Generally follows standard, but (conceptually) truncates trailing white-space before performing some functions (like the CHARACTER_LENGTH-function).

PostgreSQL's handling of the CHAR-type has changed in every recent major release. Consider using its VARCHAR type instead, unless you know for sure that your CHAR values will always be of a certain length and without trailing spaces; VARCHARs shouldn't have worse performance than CHAR in PostgreSQL.

DB2 Follows the standard.

MSSQL Generally follows standard, but (conceptually) truncates trailing white-space before performing some functions (at least before LEN()).

MySQL Violates the standard by effectively truncating all trailing spaces.

The documentation states that MySQL truncates trailing spaces when CHAR values are retrieved. That may be true, but it seems that truncation even happens before the CHAR values are used as input in functions like CONCAT, CHARACTER_LENGTH, etc.

Oracle Follows the standard, with a minor exception: Oracle doesn't remove trailing spaces which exceed the specified CHAR length, but raises an exception.

Date and time

The TIMESTAMP type

Standard Part of the Core requirements, feature ID F051-03.
Stores year, month, day, hour, minute, second (with fractional seconds; default is 6 fractional digits).
Extension to Core SQL (feature ID 411): TIMESTAMP WITH TIME ZONE which also stores the time zone.

Examples of TIMESTAMP literals:
Examples of TIMESTAMP WITH TIME ZONE literals:

- TIMESTAMP '2003-07-29 13:19:30+02:00'
- TIMESTAMP '2003-07-29 13:19:30.5+02:00'

It's strange that TIMESTAMP WITH TIME ZONE literals are not represented as, e.g.,
TIMESTAMP WITH TIME ZONE '2003-07-29 13:19:30+01:00', but according to Melton & Simon's book, they aren't.

PostgreSQL Follows that standard with one exception:
TIMESTAMP '2003-08-23 01:02:03 +02:00' is interpreted as a TIMESTAMP WITHOUT TIME ZONE (discarding the '+02:00' part)—not as a TIMESTAMP WITH TIME ZONE value. The standard may be illogical regarding this, but a standard is a standard...

Performs good sanity checks on inserted timestamp values; e.g. this will work:

```sql
INSERT INTO tablename (columnname)
VALUES (TIMESTAMP '2003-02-28 00:05:00')
```

while this will **fail**:

```sql
INSERT INTO tablename (columnname)
VALUES (TIMESTAMP '2003-02-29 00:05:00')
```

**DOCUMENTATION**

DB2 has the TIMESTAMP data type, but not the extended TIMESTAMP WITH TIME ZONE type. DB2 accepts TIMESTAMP literals like '2003-07-23 00:00:00', however it doesn't accept the typed

```sql
TIMESTAMP '2003-07-23 00:00:00' variant.
```

Performs good sanity checks on inserted timestamp values; e.g. this will work:

```sql
INSERT INTO tablename (columnname)
VALUES ('2003-02-28 00:05:00')
```

while this will **fail**:

```sql
INSERT INTO tablename (columnname)
VALUES ('2003-02-29 00:05:00')
```

**DOCUMENTATION**

MSSQL Note that MSSQL's choice of words related to date and time is confusing: In MSSQL's vocabulary, **datetime** is a concrete data type, whereas in the SQL standard, datetime is a general term covering the DATE, TIME and TIMESTAMP types.

MSSQL has a strange pseudo-type called TIMESTAMP, but has deprecated it; don't use it in new code.

The closest match to the SQL standard's TIMESTAMP type is **DATETIME**. This type stores the combination of date and time. It has a maximum of three fractional digits for seconds.

Performs good sanity checks on inserted timestamp values; e.g. this will work:

```sql
INSERT INTO tablename (columnname)
VALUES ('2003-02-28 00:05:00')
```

while this will **fail**:

```sql
INSERT INTO tablename (columnname)
VALUES ('2003-02-29 00:05:00')
```

**DOCUMENTATION**

MySQL No matter what date/time data type chosen in MySQL, storage of fractional seconds and time zones are not supported (the **TIME** type accepts time literals with fractional seconds, but discards the fractional part when storing the value). You will have to invent your own systems for such information.

Note also, that MySQL's choice of words related to date and time is confusing: In MySQL's vocabulary, **datetime** is a concrete data type, whereas in the SQL standard, datetime is a general term covering the DATE, TIME and TIMESTAMP types.

MySQL has a type called TIMESTAMP, but it is quite different from the standard TIMESTAMP: It's a 'magic' data type with side effects in that it's automatically updated to the current date and time if some criteria are fulfilled.

MySQL has a type called **DATETIME**. Like MySQL's TIMESTAMP type, it stores a combination of date and time without fractional seconds. There are no side effects associated with the DATETIME type—which makes it the closest match to the SQL standard's TIMESTAMP type.
By default, MySQL's sanity checks with regard to dates and time are (deliberately) poor. For example, MySQL accepts DATETIME values of '2003-02-29 00:05:00' and '2003-01-32 00:00:00'. Such values yield warnings (which you must check for if you want to be warned), but result in a value of zero being stored.

**DOCUMENTATION**

Oracle

Follows the standard. Oracle has both the TIMESTAMP and the extended TIMESTAMP WITH TIME ZONE types.

A special gotcha applies, though: *Oracle forbids columns of type TIMESTAMP WITH TIME ZONE as part of a unique key*: this includes primary and foreign keys. Timestamps without time zone (and Oracle's special TIMESTAMP WITH LOCAL TIME ZONE) are accepted.

Performs good sanity checks on inserted timestamp values; e.g. this will work:

```
INSERT INTO tablename (columnname)
VALUES (TIMESTAMP'2003-02-28 00:05:00')
```

while this will *fail*:

```
INSERT INTO tablename (columnname)
VALUES (TIMESTAMP'2003-02-29 00:05:00')
```

**DOCUMENTATION**

See also this link to an article timestamp handling in the 'Big Three' (DB2, MSSQL, Oracle).

**SQL functions**

**CHARACTER_LENGTH**

**Standard**

CHARACTER_LENGTH(*argument*); returns NUMERIC. Returns NULL if the input is NULL. 

Alias: CHAR_LENGTH.  
The argument may be of type CHAR or VARCHAR.  
Part of the Core SQL requirements (feature ID E021-04).  
Related function: OCTET_LENGTH.

**PostgreSQL**

Follows the standard, providing CHARACTER_LENGTH (and CHAR_LENGTH).

Note that PostgreSQL removes trailing (not leading) space from from CHAR values before counting. Note also that the behaviour of CHARACTER_LENGTH with regard to CHAR values has changed between versions 7.4 and 8.0 of PostgreSQL.

**DOCUMENTATION**

**DB2**

Doesn't have CHARACTER_LENGTH. Provides the LENGTH function instead.

Note that CHAR values are space-padded (like the standard says they should be), so the length of 'HEY ' is 5. Consider using LENGTH(RTRIM(foo)) if you want the length without trailing spaces.

**DOCUMENTATION**

**MSSQL**

Doesn't have CHARACTER_LENGTH. Provides the LEN and DATALENGTH functions instead (the latter is especially valid for 'special' data types like the TEXT type).  
Note that MSSQL's LEN-function removes trailing (not leading) spaces from CHAR values before counting; MSSQL's DATALENGTH doesn't discard spaces.

**DOCUMENTATION:** LEN and DATALENGTH

**MySQL**

Provides CHARACTER_LENGTH.  
Aliases: CHAR_LENGTH, LENGTH.  
Note that MySQL removes trailing (not leading) spaces from CHAR values before counting.

**DOCUMENTATION**

**Oracle**

Doesn't have CHARACTER_LENGTH. Provides the LENGTH function instead.

Behaves in strange ways if the input is the empty string or NULL, because of Oracles non-standard NULL handling (it considers NULL and the empty string identical 'values').

Note that CHAR values are space-padded (like the standard says they should be), so the length of 'HEY ' is 5. Consider using LENGTH(TRIM(TRAILING FROM foo)) if you want the length without leading/trailing spaces.

**DOCUMENTATION**
SUBSTRING

Standard The standard defines two variants of the SUBSTRING function:

1. To comply with Core SQL (Feature E021-06), the DBMS must support an 'ordinary' SUBSTRING function which extracts characters from a string:
   
   ```sql
   SUBSTRING(input FROM start-position [FOR length])
   ```

   Strings start at position 1. The `start-position` argument is a numeric value, as is the optional `length`-argument. If no `length` parameter is indicated, `length` becomes `infinite` (SQL:2003 specifies an extra optional argument—`USING x`—that has to do with Universal Character Sets, e.g. Unicode. `x` may be one of OCTETS or CHARACTERS.)

   The result is NULL if any of the arguments is NULL.

   Some cases of out-of-range values for `start-position` and `length` are allowed. Examples:
   
   - `SUBSTRING('12345' FROM 6)` yields the empty string.
   - A `start-position` less than 1 effectively sets `start-position` to 1 and reduces the value of `length` by `1+abs(start-position)`.
     I.e., if `start-position` is -3 and `length` is 6, then the `length` value becomes 2.

     Another way to put it is that when `start-position` is negative, a bunch of arbitrary/blank characters are prepended to the input-value. `bunch=1-start-position`.

     For an exact definition: see item three in the "General Rules" part of section 6.29 in the standard.

2. The DBMS may optionally offer a regular expression variant (Feature T581) of SUBSTRING:
   
   ```sql
   SUBSTRING(input SIMILAR pattern ESCAPE escape-char)
   ```

   Pattern deserves some explanation. It's a string which needs to consist of three parts: A part matching `before` the wanted sub-string, the wanted substring, and a part matching `after` the wanted substring. The parts must be separated by a combination of the indicated `escape-char` (escape-character) and a double-quote ("). Example:

   ```sql
   SUBSTRING('abc' SIMILAR 'a"b"c' ESCAPE '"')
   ```

   should yield `b`

   The pattern description rules in SQL don't completely resemble POSIX regular expressions, as far as I can see.

PostgreSQL PostgreSQL provides three SUBSTRING flavors:

- Ordinary SUBSTRING: As the standard's ordinary SUBSTRING variant.
- POSIX regular expression SUBSTRING: Syntax is
  
  ```sql
  SUBSTRING(input FROM pattern-string)
  ```

  Pattern rules are of the POSIX variant. Returns NULL when pattern doesn't match.
- Sort-of SQL-style regular expression SUBSTRING: Syntax is
  
  ```sql
  SUBSTRING(input FROM pattern-string FOR escape-char)
  ```

  Pattern-rules are supposed to match the SQL-standard's rules, although my tests sometimes suggest otherwise (hasn't been reported as bugs, because I'm not completely sure how SQL's regex-rules are supposed to be expressed). Returns NULL when pattern doesn't match.

DB2 Doesn't provide the standard SUBSTRING function. Provides

```sql
SUBSTR(input, start-pos[, length])
``` instead (i.e. `length` is optional).

A `start-pos` of 1 is the first character in the input string.

DB2 is less permissive than the standard: Out-of-range values are not permitted, and NULL cannot be the value for `start-pos` or `length`.

DB2 doesn't seem to provide regular expression facilities for queries.

MSSQL MSSQL has a SUBSTRING function, but its syntax differs from that of the standard. The syntax is:

```sql
SUBSTRING(input, start, length)
```

where `start` is an integer specifying the beginning of the string, and `length` is a non-negative integer indicating how many characters to return.
**Comparison of different SQL implementations**

**MySQL**
MySQL supports the standard's ordinary SUBSTRING function, with some twists (see below). No regular expression based substring extraction is supported.

MySQL breaks the standard when negative values are used as either start-position or length:
- According to the standard, `SUBSTRING('abc' FROM -2 FOR 4)` should yield 'a'; in MySQL, the result is 'bc'.
- According to the standard, `SUBSTRING('abc' FROM 2 FOR -4)` should yield an error; MySQL returns an empty string.

**Oracle**
Oracle doesn't provide the standard SUBSTRING function. Provides `SUBSTR(input, start-pos[, length])` instead (i.e. `length` is optional).
Oracle provides a number of SUBSTR-variants (SUBSTRB, SUBSTRC, SUBSTR2, SUBSTR4, same syntax as for SUBSTR), mainly for handling various kinds of non-latin-only string-types.
Oracle doesn’t have support for string-extraction with the special SQL-style regular expressions. Instead, it has the `REGEXP_SUBSTR` function which offers string extraction, using POSIX-style regular expression pattern matching.

**REPLACE**

By REPLACE is meant a string-function which searches a source string (haystack) for occurrences of a string to be replaced (needle) and replaces it with a new string (replacement).

**Standard**
Not mentioned. May be obtained through a combination of other functions (have a look at the OVERLAY, POSITION and CHARACTER_LENGTH functions).

A *de facto* standard seems to have emerged with regard to REPLACE:

```sql
REPLACE (haystack:string, needle:string, replacement:string)
```

which means 'replace needle with replacement in the string haystack'. Replacement is done case-sensitively unless otherwise stated.

The REPLACE function may be handy for correcting spelling errors (and other situations):

```sql
UPDATE tablename
SET fullname=REPLACE(fullname,'Jeo ','Joe ')
```

**PostgreSQL**
Follows *de facto* standard.

**DB2**
Follows *de facto* standard.

**MSSQL**
Follows *de facto* standard with the exception that MSSQL by default works case insensitively.

**MySQL**
Follows *de facto* standard.
MySQL even works case sensitively.

Note that the `REPLACE`-function is different from MySQL's non-standard `REPLACE INTO` expression.

**Oracle**
Follows *de facto* standard.

**TRIM**

**Standard**
Core SQL feature ID E021-09: `TRIM(where characters FROM string_to_be_trimmed)`

*where* may be one of LEADING, TRAILING or BOTH—or omitted which implies BOTH.

Note 1:
In this author's opinion, it's confusing that most (if not all) string-related functions in MySQL work case sensitively, while MySQL's default behaviour is to work case insensitively in plain WHERE-clauses involving string comparisons.
characters indicates what character(s) to remove from the head and/or tail of the string. It may be omitted which implies the value '' (space character).

In other words, the shortest form is TRIM(string_to_be_trimmed) which in effect means TRIM(BOTH ' ' FROM string_to_be_trimmed).

Trimming NULL returns NULL.

PostgreSQL Follows the standard.

DB2 Doesn't support the standard TRIM function.

Provides
LTRIM(string_to_be_trimmed)
and
RTRIM(string_to_be_trimmed)

DOCUMENTATION: LTRIM and RTRIM.

MSSQL Doesn't support the standard TRIM function.

Provides
LTRIM(string_to_be_trimmed)
and
RTRIM(string_to_be_trimmed)

DOCUMENTATION: LTRIM and RTRIM.

MySQL Follows the standard.

DOCUMENTATION

Oracle Follows the standard with two exceptions:

- Oracle doesn't allow you to trim multiple characters. I.e., TRIM('**' FROM foo) is illegal in Oracle.
- Due to Oracle's non-standard NULL-handling, you may get strange results of trimming NULL or the empty string.

DOCUMENTATION

LOCALTIMESTAMP

It's often important to get the value of current date and time. Below are the functions used to do that in the different implementations.

Standard The current timestamp (without time zone) is retrieved with the LOCALTIMESTAMP function which may be used as:

```
SELECT LOCALTIMESTAMP ...
```
or
```
SELECT LOCALTIMESTAMP(precision) ...
```

Note that "SELECT LOCALTIMESTAMP() ..." is illegal: If you don't care about the precision, then you must not use any parenthesis.

If the DBMS supports the non-core time zone features (feature ID F411), then it must also provide the functions CURRENT_TIMESTAMP and CURRENT_TIMESTAMP(precision) which return a value of type TIMESTAMP WITH TIME ZONE. If it doesn't support time zones, then the DBMS must not provide a CURRENT_TIMESTAMP function.

PostgreSQL Follows the standard.

DOCUMENTATION

DB2 Doesn't have the LOCALTIMESTAMP function.

Instead, it provides a special, magic value ('special register' in IBM language), CURRENT_TIMESTAMP (alias to 'CURRENT_TIMESTAMP') which may be used as though it were a function without arguments. However, since
DB2 doesn't provide TIMESTAMP WITH TIME ZONE support, the availability of CURRENT_TIMESTAMP could be said to be against the standard—at least confusing.

**MSSQL**

Doesn't have the LOCALTIMESTAMP function.

Instead, it has CURRENT_TIMESTAMP which—however—doesn't return a value of TIMESTAMP WITH TIME ZONE, but rather a value of MSSQL's DATETIME type (which doesn't contain time zone information).

**MySQL**

Follows the standard.

**Oracle**

Follows the standard.

### Concatenation

**Standard**

Core feature ID E021-07:

Concatenating two strings is done with the `||` operator:

```
string1 || string2
```

If at least operand is NULL, then the result is NULL.

It's unclear to me if the DBMS is allowed to try to automatically cast the operands to concatenation-compatible types.

**PostgreSQL**

Follows the standard.

Automatically casts the concatenated values into types compatible with concatenation. If an operand is NULL then the result is NULL.

**DB2**

Follows the standard, partly.

Does not automatically cast concatenated values into compatible types. Throws exception if an operand is NULL.

**MSSQL**

Breaks the standard by using the `+` operator instead of `||`.

Does not automatically cast operands to compatible types. If an operand is NULL, then the result is NULL.

**MySQL**

Badly breaks the standard by redefining `||` to mean **OR**.

Offers instead a function, `CONCAT(string, string)`, which accepts two or more arguments.

Automatically casts values into types which can be concatenated. If an operand is NULL, then the result is NULL.

**Oracle**

Follows the standard, partly.

Automatically casts values into types which can be concatenated.

As Oracle interprets NULL as the empty string, it doesn't return NULL if an operand is NULL.

### Constraint handling

**The UNIQUE constraint**

**Standard**

As the constraint name indicates, a (set of) column(s) with a UNIQUE constraint may only contain unique (combinations of) values.
A column—or a set of columns—which is subject to a UNIQUE constraint must also be subject to a not NULL constraint, unless the DBMS implements an optional "NULLs allowed" feature (Feature ID 591). The optional feature adds some additional characteristics to the UNIQUE constraint:

1. Columns involved in a UNIQUE constraint may also have NOT NULL constraints, but they do not have to.
2. If columns with UNIQUE constraints do not also have NOT NULL constraints, then the columns may contain any number of NULL-'values'. (Logical consequence of the fact that NULL<>NULL.)

In SQL:2003 parlance, the constraint is satisfied, if

there are no two rows in [the relation] such that the value of each column in one row is non-null and is not distinct from the value of the corresponding column in the other row

PostgreSQL Follows the standard, including the optional NULLs allowed feature.

DB2 Follows the non-optional parts of the UNIQUE-constraint. Doesn't implement the optional NULLs allowed feature.

MSSQL Follows the standard—with a twist:

MSSQL offers the NULLs allowed feature, but allows at most one instance of a NULL-'value', if NULLs are allowed; i.e. breaks characteristic 2 in the above description of the standard.

MySQL Follows the standard, including the optional NULLs allowed feature.

Oracle Follows the standard—with a twist:

The optional NULLs allowed feature is implemented: If the UNIQUE-constraint is imposed on a single column, then the column may contain any number of NULLs (as expected from characteristic 2 in the above description of the standard). However, if the UNIQUE-constraint is specified for multiple columns, then Oracle sees the constraint as violated if any two rows

- contain at least one NULL in a column affected by the constraint
- identical, non-NULL values in the rest of the columns affected by the constraint

MySQL Follows the standard, including the optional NULLs allowed feature.

MSSQL Follows the standard—with a twist:

MSSQL offers the NULLs allowed feature, but allows at most one instance of a NULL-'value', if NULLs are allowed; i.e. breaks characteristic 2 in the above description of the standard.

MySQL Follows the standard, including the optional NULLs allowed feature.

Oracle Follows the standard—with a twist:

The optional NULLs allowed feature is implemented: If the UNIQUE-constraint is imposed on a single column, then the column may contain any number of NULLs (as expected from characteristic 2 in the above description of the standard). However, if the UNIQUE-constraint is specified for multiple columns, then Oracle sees the constraint as violated if any two rows

- contain at least one NULL in a column affected by the constraint
- identical, non-NULL values in the rest of the columns affected by the constraint

Mixture of type and operations

Automatic key generation

It's sometimes handy to have the DBMS handle generation of keys. The DBMSes offer various means for this. Note, however, that some database authorities warn against—at least some variants of—auto-generated keys; this is a classic database discourse.
Comparison of different SQL implementations

Standard

The standard specifies a column attribute of:

GENERATED ... AS IDENTITY (non-core feature ID T174+T175).

When creating a table, an IDENTITY clause may be declared for certain types of columns (INTEGER being one):

CREATE TABLE tablename (  
tablename_id INTEGER GENERATED ALWAYS AS IDENTITY  
...  
)

or

CREATE TABLE tablename (  
tablename_id INTEGER GENERATED BY DEFAULT AS IDENTITY  
...  
)

The column with the IDENTITY attribute will be given values in increasing order, possibly with 'holes' (...3,4,7,...).

A base table may at most contain one column with the IDENTITY attribute. NOT NULL is implied for an IDENTITY column. Normally, a column declared with IDENTITY will also be declared PRIMARY KEY, but it's not implied.

The examples differ in their 'ALWAYS' vs. 'BY DEFAULT' clauses:

- When ALWAYS is specified, the user cannot specify a value for the column which means that the DBMS can guarantee successful insertion of a unique value on each table insert.
- When BY DEFAULT is specified, the user may manually specify what value to put in the identity field of a row. The flip side is that the DBMS cannot guarantee that this will work.

The standard specifies several extended options which may be declared for a generated IDENTITY column.

PostgreSQL

PostgreSQL doesn't support the standard's IDENTITY attribute.

By default, PostgreSQL 'secretly' assigns an Object ID to every row in a base table and that value may be cast to an integer and used as identity, but it's very non-standard, and therefore not a good idea as an identity source in most situations.

PostgreSQL's best offering for a column with auto-generated values is to declare a column of 'type' SERIAL:

CREATE TABLE tablename (  
tablename_id SERIAL,  
...  
)

'SERIAL' is a short-hand for creating a sequence and using that sequence to create unique integers for a column. If the table is dropped, PostgreSQL seems to remember to drop the sequence which was created as a side-effect of using the SERIAL type.

As a user may manually insert or update a value in a column created as SERIAL, this comes closest to the standard's GENERATED BY DEFAULT AS IDENTITY variant. It's probably possible to use a trigger to protect a SERIAL-column, such that it will get semantics matching the standard's GENERATED ALWAYS AS IDENTITY attribute.

DOCUMENTATION: OID and the SERIAL types.

DB2

Follows standard, albeit with some restrictions on how identity columns may (not) be added to an existing table, etc.

DOCUMENTATION: CREATE TABLE syntax and description of identity columns.

MSSQL

MSSQL offers IDENTITY as a column property, but with a different syntax (not as intuitive and with less options) than the standard's specification. An example of creating a table with an IDENTITY column:

CREATE TABLE tablename (  
tablename_id INT IDENTITY PRIMARY KEY,  
...  
)
With MSSQL's IDENTITY attribute, the user cannot manually insert/change the value, unless the user has first run
SET IDENTITY_INSERT * ON
I.e., MSSQL's IDENTITY type is closest to the standard's GENERATED ... ALWAYS AS IDENTITY variant.

**Documentation**: The IDENTITY property and SET IDENTITY_INSERT.

**MySQL**
MySQL doesn't support the standard's IDENTITY attribute.

As an alternative, an integer column may be assigned the non-standard AUTO_INCREMENT attribute:

```sql
CREATE TABLE tablename (  
    tablename_id INTEGER AUTO_INCREMENT PRIMARY KEY,
    ...
)
```

Columns with the AUTO_INCREMENT attribute will—under certain conditions—automatically be assigned a value of \(<\text{largest value in column}> + <\text{at least 1} >\). Look in MySQL's documentation for the (rather extensive) details.

A table can have at most one column with the AUTO_INCREMENT attribute; that column must be indexed (it doesn't have to be a primary key, as in the example SQL above) and cannot have a DEFAULT value attribute.

It's probably not too far fetched to think of MySQL's AUTO_INCREMENT feature as this equivalence:

**MySQL**:
```sql
CREATE TABLE tablename (  
    columnname INTEGER AUTO_INCREMENT PRIMARY KEY
    ...
)
```

**Standard SQL**:
```sql
CREATE TABLE tablename (  
    columnname INTEGER DEFAULT some_func() PRIMARY KEY
    ...
)
```

where `some_func()` is a function which finds 1 plus the currently largest value of `columnname`.

The nice thing about this approach is that the automatic value insertion should never fail, even though some of the column's values might have been manually set—i.e. the combined advantages of the standard's ALWAYS and BY DEFAULT variants.

The drawback is that it might result in more house-keeping: The system may need extra table locks when performing row updates/insertions to protect against ghost updates in concurrent transactions—thus slowing down the system in case of many concurrent updates/insertions.

**Documentation**
Oracle

Oracle doesn't support the standard's IDENTITY attribute.

If you want an auto-incrementing column in Oracle, then create a sequence and use that sequence in a trigger associated to the table. Example: For the table `mytable`, you want the `mytable_id` column to be of integer type, with an auto-incrementing values:

```sql
CREATE TABLE mytable (  
  mytable_id INTEGER PRIMARY KEY,  
  ...  
) ;

CREATE SEQUENCE mytable_seq ;

CREATE TRIGGER mytable_seq_trigger  
BEFORE INSERT ON mytable FOR EACH ROW  
BEGIN  
  IF (:new.mytable_id IS NULL) THEN  
    SELECT mytable_seq.nextval INTO :new.mytable_id  
    FROM DUAL ;  
  END IF ;  
END ;
/
```

This will create an auto-incrementing column resembling the `GENERATED BY DEFAULT` variant from the standard. If an column resembling the `GENERATED ALWAYS` variant is needed, then the trigger should be extended to raise an exception if the user tries to insert a non-NULL value, and a trigger preventing UPDATEs of the relevant column should be added.

Note: If 'nice', incrementing values aren't important, you may use Oracle's SYS_GUID function as the default for a column; that way, universally unique identifiers will be assigned if you don't indicate a value for the column in new rows.

**DOCUMENTATION:** `CREATE TRIGGER`, `CREATE SEQUENCE`, and `SYS_GUID`.

Note: IBM has a page comparing IDENTITY columns and sequences.

**Command line procedures**

The following are not necessarily SQL operations, but rather a description of how different operations are performed in the command line interface provided by each product.

The shape of the command line interfaces in the commercial products is depressing. Vendors, please do something about it: Not all database developers like to use slow GUls for technical stuff. And sometimes, DBMS work is performed over slow Internet lines which makes a decent command line interface vital.

Fortunately, a tool like HenPlus exists. It can be a pain to install, but once working, it's nice to work with.

**Starting the command line interface**

- **Standard**: Not defined.
- **PostgreSQL Run:**
  ```bash
  psql  
  which should be in the PATH in any sensible installation.  
  PostgreSQL's command line interface is very user friendly. It has command history (press arrow-up for previous commands) and a fairly well-working command completion feature.
  ```
- **DB2 Run:**
  ```bash
  db2 -t  
  The `db2` binary may not be in your PATH or may be missing vital environment variables (which is one of the stupid parts of DB2's installation procedure: It doesn't offer to set up a proper global DB2 environment for the users on the server) and you may have to include the `db2profile` file (situated in the `sqllib` directory in the home directory of the special DB2 instance user) into your shell.
  
  E.g. on my Linux system, I've added the following line to my `.bash_profile` in order to get a shell with proper DB2 environment when logging in:
  ```
The 'utility' doesn't seem to have anything resembling useful command history or command completion. Fortunately, queries may be sent to the `db2` 'utility' in a non-interactive way like this:

```
db2 "SELECT a_column FROM a_table"
```

This allows you to make use of your shell's command history handling.

DB2 also has 'utilities' called `db2sql92` and `db2batch` which some might find a bit nicer to work with, although they lack support for some useful non-SQL db2 commands like `LIST TABLES`.

### MSSQL

The command line interface is started by running `sqlcmd`

`sqlcmd` is not nice to work with. It's bad at formatting result sets. It doesn't have command line completion. You have to say `go` after your commands. A positive thing about `sqlcmd`: It has command history, so you may press arrow-up for previous commands in the current `sqlcmd` session.

In MSSQL 2000, the command line interface was started by running `osql`.

An alternative to `osql`—apart from HenPlus, mentioned above—is `SQSH` which should work on any modern open source operating system.

### MySQL

Run:

```
mysql
```

If you need help on the optional command line options, see the man page.

On platforms like Linux and FreeBSD (which have decent readline-capabilities), MySQL's command line interface is simply great; not much else to say. MySQL's command line interface is said to be rather poor on Windows, though.

### Oracle

Run:

```
sqlplus
```

`sqlplus` lacks command completion and command history for more than the last command.

### Get list of tables

**Standard**

Part 11 of the SQL standard specifies the INFORMATION_SCHEMA schema which must be part of all database catalogues. The schema may be used like this:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE='BASE TABLE'
```

or (often more relevant):

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE='BASE TABLE'
  AND TABLE_SCHEMA='SCHEMA-NAME'
```

See a warning about potential case sensitivity problems below.

**PostgreSQL** Follows the standard, except for some gotchas mentioned below; also, PostgreSQL's INFORMATION_SCHEMA views don't contain all SQL-2003's columns (an example: PostgreSQL's INFORMATION_SCHEMA.COLUMNS view does not contain the IS_IDENTITY column).

In command-line context, it's easier to use the following non-SQL command instead of querying the INFORMATION_SCHEMA:

```
\dt
```

Documentation: The `TABLES INFORMATION_SCHEMA VIEW`, the `PSQL TOOL`.

**DB2**

Doesn't provide the standard `INFORMATION_SCHEMA`. Instead, DB2 offers the `SYSCAT` schema (catalog) which is somewhat compatible.

Offers what is probably a shorthand to some system catalog query:
LIST TABLES
or - if you want to see tables in another schema:
LIST TABLES FOR SCHEMA foo

**MSSQL**
Follows that standard.
Sometimes, the `SP_TABLES` system stored procedure is easier to use.

**Documentation**:
- Information Schema Views
- `sp_tables`

**MySQL**
Follows the standard, except that MySQL doesn't support schemas, so one might say that MySQL's `INFORMATION_SCHEMA` is really an 'INFORMATIONATABASE' or 'INFORMATION_CATALOGUE'.

In command-line context, it's easier to use the following non-standard SQL:
SHOW TABLES

**Documentation**:
- The `INFORMATION_SCHEMA`
- `SHOW TABLES`

**Oracle**
 Doesn't provide the standard `INFORMATION_SCHEMA`. Provides a data dictionary system instead.

The quickest way to get a usable list of 'normal' tables:

```
SELECT * FROM tab
```
(Use of the `tab` dictionary view is officially deprecated, though.)

**Documentation**

**Warning about a general case sensitivity gotcha**

Note that there may be case sensitivity issues involved when using meta-data views like those in the `INFORMATION_SCHEMA`. Generally, the standard states that the name of an identifier (such as table names) are implicitly converted to uppercase, unless double-quotes are used when referring to the identifier. The same goes for identifiers used in queries: A query like `SELECT foo FROM tablename` is implicitly converted to `SELECT FOO FROM TABLENAME`.

If you create your table as

```
CREATE TABLE testtab (id INTEGER PRIMARY KEY)
```
then a query like

```
SELECT * FROM testtab
```
should work fine, and

```
SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME='TESTTAB'
```
should work, while the following query will probably fail:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME='testtab'
```

**Warning about INFORMATION_SCHEMA gotchas in PostgreSQL**

Warning: PostgreSQL's case-conversion rules for unquoted identifiers (such as table names) are non-standard: PostgreSQL converts the identifiers to lower case, instead of converting to upper case. This means that you may try altering the case of identifier names used for queries in the `INFORMATION_SCHEMA` if you experience unexpected, empty metadata queries.

Note also that due to PostgreSQL's handling of constraint names, the `INFORMATION_SCHEMA` cannot safely be used to deduce referential constraints; for this, you have to use PostgreSQL's `pg_catalog` system-schema.

**Getting a table description**

**Standard**
Part 11 of the SQL standard specifies the `INFORMATION_SCHEMA` schema which must be part of all database catalogues. The schema may be used like this:

```
SELECT column_name, data_type, column_default, is_nullable
FROM
    information_schema.tables AS t
JOIN
    information_schema.columns AS c ON
    t.table_catalog=c.table_catalog AND
    t.table_schema=c.table_schema AND
    t.table_name=c.table_name
```
WHERE
t.table_name='TABLE-NAME'

—or like this (more verbose):

SELECT
    column_name,
data_type,
    character_maximum_length,
    numeric_precision,
    column_default,
is_nullable
FROM
    information_schema.tables as t
JOIN
    information_schema.columns AS c
    ON t.table_catalog=c.table_catalog AND
t.table_schema=c.table_schema AND
t.table_name=c.table_name
WHERE
    c.table_schema='TABLE-SCHEMA'
    AND
    c.table_name='TABLE-NAME'

To get information about constraints, involved columns and (possibly) referenced columns, a query like this may be used:

SELECT
    tc.CONSTRAINT_NAME,
    CONSTRAINT_TYPE,
    ccu.COLUMN_NAME,
    rccu.COLUMN_NAME,
    rccu.TABLE_CATALOG,
    rccu.TABLE_SCHEMA,
    rccu.TABLE_NAME,
    CHECK_CLAUSE
FROM
    INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
LEFT JOIN
    INFORMATION_SCHEMACONSTRAINT_COLUMN_USAGE ccu
    ON tc.CONSTRAINT_CATALOG=ccu.CONSTRAINT_CATALOG AND
    tc.CONSTRAINT_SCHEMA=ccu.CONSTRAINT_SCHEMA AND
    tc.CONSTRAINT_NAME=ccu.CONSTRAINT_NAME AND
    tc.TABLE_CATALOG=ccu.TABLE_CATALOG AND
    tc.TABLE_SCHEMA=ccu.TABLE_SCHEMA AND
    tc.TABLE_NAME=ccu.TABLE_NAME
LEFT JOIN
    INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS rc
    ON rc.CONSTRAINT_CATALOG=ccu.CONSTRAINT_CATALOG AND
    rc.CONSTRAINT_SCHEMA=ccu.CONSTRAINT_SCHEMA AND
    rc.CONSTRAINT_NAME=ccu.CONSTRAINT_NAME
LEFT JOIN
    INFORMATION_SCHEMACONSTRAINT_COLUMN_USAGE rccu
    ON rc.UNIQUE_CONSTRAINT_CATALOG=rccu.CONSTRAINT_CATALOG AND
    rc.UNIQUE_CONSTRAINT_SCHEMA=rccu.CONSTRAINT_SCHEMA AND
    rc.UNIQUE_CONSTRAINT_NAME=rccu.CONSTRAINT_NAME
LEFT JOIN
    INFORMATION_SCHEMA.CHECK_CONSTRAINTS cc
    ON tc.CONSTRAINT_CATALOG=cc.CONSTRAINT_CATALOG AND
    tc.CONSTRAINT_SCHEMA=cc.CONSTRAINT_SCHEMA AND
    tc.CONSTRAINT_NAME=cc.CONSTRAINT_NAME
WHERE
    tc.TABLE_CATALOG='CATALOG-NAME' AND -- see remark
tc.TABLE_SCHEMA='SCHEMA-NAME' AND -- see remark
tc.TABLE_NAME='TABLE-NAME'
ORDER BY tc.CONSTRAINT_NAME

If you don't care about potential namespace conflicts, you may leave out the lines commented with
See also: Warning about potential case sensitivity problems above.

PostgreSQL Follows the standard, except for some gotchas mentioned above.

In command-line context it's easier to use this non-SQL command:
```
d tablename
```

DB2 Doesn't provide the standard INFORMATION_SCHEMA.

To obtain (very) basic information about a table:
```
DESCRIBE TABLE tablename
DESCRIBE INDEXES FOR TABLE tablename SHOW DETAIL
```

To get information about constraints, including involved/referred columns, a query like the following may be used, although the db2 'utility' isn't good at adjusting column widths in output (i.e. the output is not easy to read):
```
SELECT
tc.constname as const_name,
type as const_type,
kcu.colname as col_name,
r.reftabschema as ref_tabschema,
r.reftabname as ref_tabname,
kcu_r.colname as ref_colname
FROM
  syscat.tabconst tc
JOIN
  syscat.keycoluse kcu ON
tc.constname=kcu.constname
LEFT JOIN
  syscat.references r ON
  type='F' AND
  tc.constname=r.constname
LEFT JOIN
  syscat.keycoluse kcu_r ON
  r.constname=kcu_r.constname
WHERE
  tc.tabschema=CURRENT_SCHEMA AND
  tc.tabname=UCASE('tablename')
ORDER BY const_name,col_name
```

DOCUMENTATION:

- The DESCRIBE command in the "db2" command line processor
- SYSCAT views

MSSQL Follows the standard, except that

- MSSQL uses non-standard names for some standard datatypes, i.e. VARCHAR instead of the standard's CHARACTER VARYING
- MSSQL's INFORMATION_SCHEMA doesn't have all SQL:2003's columns (an example: MSSQL's INFORMATION_SCHEMA.COLUMNS view does not contain the IS.IDENTITY column)

Often, the SP_HELP tablename system stored procedure is easier to use.

DOCUMENTATION:

- Information Schema Views
- sp_help

MySQL Follows the standard, except that

- MySQL doesn't support schemas, so one might say that MySQL's INFORMATION_SCHEMA is really an 'INFORMATION_DATABASE' or 'INFORMATION_CATALOGUE'.
- MySQL's INFORMATION_SCHEMA doesn't have all SQL:2003's columns (an example: MySQL's INFORMATION_SCHEMA.COLUMNS view does not contain the IS.IDENTITY column).
- As MySQL's namespaces don't match the SQL standard fully, the standard queries mentioned above will
not work. The reason is that in MySQL, the value of \texttt{TABLE_CATALOG} is \texttt{NULL} for all tables and columns. To obtain the wanted information, you need to remove the table_catalog join-conditions. I.e., the first (and simplest) of the above queries must be re-written to:

\begin{verbatim}
SELECT column_name, data_type, column_default, is_nullable
FROM
  information_schema.tables AS t
JOIN
  information_schema.columns AS c ON
  t.`table_schema` = c.`table_schema` AND
  t.`table_name` = c.`table_name`
WHERE
  t.`table_name` = 'TABLE-NAME'
\end{verbatim}

In command-line context it's easier to use this non-SQL command:

\texttt{DESCRIBE tablename}

\textbf{DOCUMENTATION:}

- The \texttt{INFORMATION_SCHEMA}
- \texttt{DESCRIBE}

\textbf{Oracle}

Oracle doesn't provide the standard \texttt{INFORMATION_SCHEMA}. Offers \textit{data dictionary views} instead.

To get (very) basic information:

\texttt{DESCRIBE tablename}

To get information on constraints, including foreign (referred) table/column information, a query like this may be used (adjust \texttt{tablename} in one of the last lines):

\begin{verbatim}
COLUMN consname FORMAT a11;
COLUMN colname FORMAT a10;
COLUMN type FORMAT a11;
COLUMN cond FORMAT a20;
COLUMN ref_tabname FORMAT a11;
COLUMN ref_colname FORMAT a11;
SELECT
  uc.constraint_name consname,
  ucc.column_name colname,
  CASE
    WHEN uc.constraint_type = 'C' THEN 'CHECK'
    WHEN uc.constraint_type = 'P' THEN 'PRIMARY KEY'
    WHEN uc.constraint_type = 'R' THEN 'REFERENTIAL'
    WHEN uc.constraint_type = 'U' THEN 'UNIQUE'
    ELSE uc.constraint_type
  END as type,
  uc.search_condition cond,
  ucc_r.table_name ref_tabname,
  ucc_r.column_name ref_colname
FROM
  user_constraints uc
JOIN
  user_cons_columns ucc ON
    uc.constraint_name = ucc.constraint_name AND
    uc.owner = ucc.owner
LEFT JOIN
  user_constraints uc_r ON
    uc_r.constraint_name = uc_r.constraint_name AND
    uc_r.owner = uc_r.owner
LEFT JOIN
  user_cons_columns ucc_r ON
    uc_r.constraint_name = ucc_r.constraint_name AND
    uc_r.owner = ucc_r.owner
WHERE
  uc.TABLE_NAME = UPPER('tablename')
ORDER BY consname, colname
\end{verbatim}

To get information on indexes on a table, a query like this may be used (adjust \texttt{tablename} in one of the last lines):

\begin{verbatim}
SELECT
  index_name,
  table_name,
  index_type
FROM
  information_schema.indices
WHERE
  table_name = 'TABLE-NAME'
\end{verbatim}

To get information on constraints, including foreign (referred) table/column information, a query like this may be used (adjust \texttt{tablename} in one of the last lines):

\begin{verbatim}
SELECT
  constraint_name,
  table_name,
  constraint_type
FROM
  information_schema.constraints
WHERE
  table_name = 'TABLE-NAME'
\end{verbatim}

To get information on indexes on a table, a query like this may be used (adjust \texttt{tablename} in one of the last lines):

\begin{verbatim}
SELECT
  index_name,
  table_name,
  index_type
FROM
  information_schema.indices
WHERE
  table_name = 'TABLE-NAME'
\end{verbatim}
Telling the DBMS to collect statistics

In most DBMSes, it's possible to enable automatic statistics gathering, but sometimes, it's nice to be able to manually tell the DBMS to gather statistics for a table (or a number of tables).

<table>
<thead>
<tr>
<th>Standard</th>
<th>Not standardized.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>ANALYZE tablename</td>
</tr>
</tbody>
</table>

If the `tablename` parameter is left out, then statistics are gathered for all tables in the current database.

**DB2**

```
RUNSTATS ON TABLE schema-name.table-name AND INDEXES ALL
```

The `RUNSTATS` command has various options, see the **DOCUMENTATION**.

**MSSQL**

First, you have to add statistics to the table:

```
CREATE STATISTICS stats_name ON table_name
    (column_name_1, column_name_2, column_name_3, ...)
```

The statistics may then be updated when needed:

```
UPDATE STATISTICS table_name
```

Having to explicitly mention tables and columns can be tedious, and in many cases, the `sp_createstats` and `sp_updatestats` stored procedures are easier to use.

**DOCUMENTATION**: `CREATE STATISTICS`, `UPDATE STATISTICS`, `sp_createstats`, `sp_updatestats`

**MySQL**

```
ANALYZE TABLE table name
```

**DOCUMENTATION**
Oracle

Oracle offers to estimate (quick) or compute (thorough) statistics for a database object. The quick way to do this is to use the deprecated `ANALYZE` command which can be used in various ways, e.g.

```sql
ANALYZE TABLE tablename ESTIMATE STATISTICS;
ANALYZE TABLE tablename ESTIMATE STATISTICS FOR ALL INDEXES;
```

(It's unclear to me if both are needed to gain the relevant statistics.)

—Or:

```sql
ANALYZE TABLE tablename COMPUTE STATISTICS;
ANALYZE TABLE tablename COMPUTE STATISTICS FOR ALL INDEXES;
```

If you want to stay away from deprecated features (although I doubt that Oracle will remove `ANALYZE...STATISTICS...` any time soon), you need to use the `DBMS_STATS` package.

**DOCUMENTATION**

**Getting a query explanation**

<table>
<thead>
<tr>
<th>Standard</th>
<th>Not standardized.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td><code>EXPLAIN &lt;query&gt;</code></td>
</tr>
</tbody>
</table>

**DOCUMENTATION**

DB2

The easiest way to get a query explanation is to save the query in a file (without a terminating semicolon), and then run a special command-line utility:

```bash
db2exp1n -database databasename -stmtfile query.sql -terminal
```

In the above example, the query has been saved to a file called "query.sql".

In some versions of DB2, you need to use the `dynexp1n` utility instead of `db2exp1n`. A visual explanation tool also exists.

If you prefer to get the explanation through SQL:

1. Set up needed `explain tables` using `EXPLAIN.DDL` which should exist in `sqlib/misc` of your DB2 instance user's home directory.
2. Optionally: Clean up old plan explanations: `DELETE FROM EXPLAIN_INSTANCE`  
3. Generate the explanation: `EXPLAIN PLAN FOR <SQL-statement>`
4. Display plan:
   ```sql
   SELECT O.Operator_ID, S2.Target_ID, O.Operator_Type, S.Object_Name, CAST(O.Total_Cost AS INTEGER) Cost
   FROM EXPLAIN_OPERATOR O
   LEFT OUTER JOIN EXPLAIN_STREAM S2
   ON O.Operator_ID=S2.Source_ID
   LEFT OUTER JOIN EXPLAIN_STREAM S
   ON O.Operator_ID = S.Target_ID
   AND O.Explain_Time = S.Explain_Time
   AND S.Object_Name IS NOT NULL
   ORDER BY O.Explain_Time ASC, Operator_ID ASC
   ```

   (Adapted from recipe in *SQL Tuning*.)

**DOCUMENTATION**

MSSQL

MSSQL can be put in a query explanation mode where queries are not actually executed, but a query explanation is returned instead:

```
SET SHOWPLAN_TEXT ON
```

The query explanation mode is turned off by running

```
SET SHOWPLAN_TEXT OFF
```

**DOCUMENTATION**

MySQL

`EXPLAIN <query>`

**DOCUMENTATION**

Oracle

After having set up a `plan table`, running

```
DELETE FROM plan_table WHERE statement_id='explanationX';
EXPLAIN PLAN
SET STATEMENT_ID = 'explanationX'
FOR <query>
```
Comparison of different SQL implementations

will place an explanation into your PLAN_TABLE. Substitute explanationX with a suitable name for the explanation (and make sure you don't delete other users' explanation plans in the DELETE-line above).

The plan explanation may now be viewed by a query like this:

```
SELECT LPAD(' ',2*(LEVEL-1))||operation operation,
       options,
       object_name,
       object_type otype,
       cardinality,
       cost
FROM plan_table
START WITH id = 0 AND statement_id = 'explanationX'
CONNECT BY PRIOR id = parent_id
       AND statement_id = 'explanationX'
```

A bit of documentation reading can probably not be avoided:

**Turning on query timing**

<table>
<thead>
<tr>
<th>Standard</th>
<th>Not standardized.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>(\text{\texttt{\textbackslash timing}})</td>
</tr>
</tbody>
</table>

**DB2**

Run the query in the "\texttt{db2batch}" command line processor; \texttt{db2batch} prints the elapsed time of each query.

**MSSQL**

`SET STATISTICS TIME ON`

**MySQL**

MySQL's command line interface prints query times by default.

**Oracle**

`SET TIMING ON`

**Other topics**

**Dummy table use**

Some DBMSes let you perform a query like this:

```
SELECT 1+1
answering
2
```

With other DBMSes, you need to insert a dummy-table expression to obtain the same result:

```
SELECT 1+1 FROM \textit{dummy-table}
```

<table>
<thead>
<tr>
<th>Standard</th>
<th>On my TODO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>No need for dummy-table.</td>
</tr>
<tr>
<td>DB2</td>
<td>\textit{Dummy-table: SYSIBM.SYSDUMMY1}.</td>
</tr>
</tbody>
</table>

In addition, the \texttt{VALUES} keyword may be used to produce a simple result set, without introducing a \texttt{FROM} clause, e.g.

```
VALUES(1+1)
```

(Note the missing SELECT and FROM keywords).

| MSSQL | No need for dummy-table. |
| MySQL | No need for dummy-table, although MySQL allows you to refer to a \texttt{DUAL} dummy-table (for Oracle compatibility). |
| Oracle | \textit{Dummy-table: DUAL}. |
Obtaining DBMS version

Standard

```sql
SELECT CHARACTER_VALUE
FROM INFORMATION_SCHEMA.SQL_IMPLEMENTATION_INFO
WHERE IMPLEMENTATION_INFO_NAME='DBMS VERSION'
```

PostgreSQL

Follows the standard. An alternative, non-standard function may be used:

```sql
SELECT VERSION()
```

**DOCUMENTATION**

DB2

Run the special `db2level` program.

**DOCUMENTATION**

MSSQL

MSSQL's implementation of the IMPLEMENTATION_SCHEMA doesn't seem to include the SQL_IMPLEMENTATION_INFO view. Instead, you may use

```sql
SELECT SERVERPROPERTY('ProductVersion')
```

(just the version), or

```sql
SELECT @@VERSION
```

(verbosely harder to parse).

**DOCUMENTATION:** `SERVERPROPERTY`, `@@VERSION`

MySQL

MySQL's `INFORMATION_SCHEMA` doesn't include the `SQL_IMPLEMENTATION_INFO` view.

Work-around:

```sql
SELECT VERSION()
```

**DOCUMENTATION**

Oracle

```sql
SELECT banner FROM v$version
```

**DOCUMENTATION**

Related work

- Mimer Information Technology AB (makers of the Mimer SQL DBMS) has an interesting feature comparison chart, displaying what SQL:1999 features are implemented in different commercial products. May be biased because it's created by a DBMS vendor.
  Mimer also has lists of reserved words.


- MySQL AB has a [feature comparison machine](http://www.mysql.com/doc/redo); possibly somewhat biased in favor of MySQL.

- Chris Fehily's [SQL: Visual QuickStart Guide](http://www.oracle.com/technetwork/indexes/technetworkray-indexes-137726.html) teaches SQL by first describing the standards-based (SQL:2003) approach, and then how to adjust to the real world, using MS Access, MSSQL, Oracle, MySQL, PostgreSQL, and DB2. (Full disclosure note: I was technical editor on second edition of the book.)

- Peter Gulutzan (who works for MySQL AB) has written several articles related to the subject. He has also written two related books:
  - [SQL-99 Complete, Really](http://www.mysql.com/doc/redo/) (co-authored with Trudy Pelzer) is said to be good.
  - [SQL Performance Tuning](http://www.mysql.com/doc/redo/) (also co-authored with Trudy Pelzer), mentions quite a few cross-product SQL issues (primarily related to performance, of course).

- Some DBMS evaluations performed at the Astrogrid Virtual Observatory (focus on spatial functionality):
  - Various [comparisons of MySQL, PostgreSQL and DB2](http://www.mysql.com/doc/redo/).
  - Comparison of [availability and names of mathematical functions](http://www.mysql.com/doc/redo/) in major DBMS products.
  - Autumn '03: Comparison of DB2, MySQL, and Postgres, comparing ease of use, scalability and performance of two types of spatial joins.
  - Autumn '02: Comparison of PostgreSQL, MySQL, Oracle, SQL Server and DB2.

- Uday Parmar: [Open Source Database Feature Comparison Matrix](http://www.mysql.com/doc/redo/).
  Note: Created by employees of a database vendor.

- SQLite's survey of NULL-handling: [NULL Handling in SQLite Versus Other Database Engines](http://www.mysql.com/doc/redo/).

- Bowman/Emerson/Darnovsky's [The Practical SQL Handbook—Using SQL Variants](http://www.mysql.com/doc/redo/) is OK for this subject, although it's already a bit out-dated and (worse) doesn't include any guidance on working with open source DBMSs.
Comparison of different SQL implementations

- Kevin E. Kline's *SQL in a Nutshell* is a good reference.
- The Analysis and Solutions Company:
  - *Building Truly Portable Database Applications in PHP* includes advice on DBMS differences (some of the presentation is PHP-specific, as the title indicates).
  - *Database Portability: Date and Timestamp Columns*.
- IBM has an online book about writing portable SQL with DB2: *SQL Reference for Cross-Platform Development*.
- Jutta Horstmann *OSDBmigration*.
- See also my DBMS links.

**Acknowledgments**

The following people have provided comments, suggestions and/or fixes, resulting in content changes on this page:

- Ian Barwick
- Chester Kustarz
- Bruno Wolff III
- Carsten Pedersen
- Jürgen Auer
- Edi Stocker
- Tzvetan Tzankov
- Jess Robinson
- Gordon P. Hemsley
- Philip Nelson
- Andreas Plesner Jacobsen
- Clive Page
- Holger Jakobs
- Dennis Björklund
- Chris Fehily
- Alf-Ivar Holm
- Joseph Fuda
- J M Sykes
- Greg Sabino Mullane
- Jari Aalto

(In chronological order.)

To Troels' home page.

Version control information: $LastChangedDate: 2006-03-12 00:02:06 +0100 (Sun, 12 Mar 2006) $.

This work is licensed under a Creative Commons License which allows anyone to use the work in any way, as long as I'm attributed.