Comparison of different SQL implementations

The goal of this page was 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 handled various SQL (and related) features. If possible, the tables also stated how the implementations *should* do things, according to the SQL standard.

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

Unfortunately, I don't have the time and motivation to keep this page up-to-date any longer.

Contents:

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:2008. 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 `5CD2-02-Foundation-2006-01.pdf`. No books cover SQL:2008 yet. Regarding the previous standard, SQL:2003, the only book covering the subject is in German which I was never any good at. Therefore, I also use the following book as reference: Jim Melton and Alan Simon: *SQL:1999—Understanding Relational Language Components* (ISBN 1-55860-456-1).
PostgreSQL	PostgreSQL 8.4.1 on CentOS Linux. DOCUMENTATION
DB2	DB2 Express-C v. 9.1 on Fedora Linux. Note that there are differences between various DB2 flavors; this page is about DB2 for "LUW"

	(Linux/Unix/Windows). DOCUMENTATION
MS SQL Server	MS SQL Server 2005 on Windows XP. Microsoft's SQL implementation is sometimes named *Transact-SQL*, or *TSQL*. In this document, I'll generally write *MSSQL* as a short-hand for Microsoft's SQL Server product. DOCUMENTATION
MySQL	MySQL Database Server 5.0.18 on Fedora Linux (i.e. MySQL AB's "classic" DBMS product—not MaxDB). DOCUMENTATION
Oracle	Oracle Database 11*g* Release 2 on Red Hat Enterprise Linux. DOCUMENTATION
Informix	Informix Dynamic Server Workgroup Edition v. 11.50 on Red Hat Enterprise Linux. DOCUMENTATION

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 (for MySQL, there is a dedicated documentation page about this). 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:2008 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.
Informix	Conforms to at least SQL-92.

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*.

Join type/feature	PostgreSQL	DB2	MSSQL	MySQL	Oracle	Informix
Natural joins (only tested: `NATURAL LEFT JOIN`)	✔			✔	✔	
`USING`-clause	✔			✔	✔	
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.

Data definition language (DDL)

Copying structure

Objective: An existing table, *t1* needs to be copied to a new table, *t2*, *without* copying data. I.e., only the structure/definition of the table is copied.

Standard	Optional feature T171 defines *LIKE clause in table definition*:
	`CREATE TABLE t2 ( LIKE t1 )`
	The DBMS may support an extension of this (feature T173) which allows for more table properties to be copied:
	`CREATE TABLE t2 ( LIKE t1 INCLUDING IDENTITY INCLUDING DEFAULTS INCLUDING GENERATED )`
	If `INCLUDING DEFAULTS` is not specified, column defaults will not be part of *t2*; likewise with `IDENTITY` and `GENERATED` properties.
	Triggers, CHECK constraints, and other 'non-trivial' table features are not copied to the new table.

PostgreSQL	Complies with the core of the feature (T171). The extended T173 feature is only partially supported, and extended with a few non-standard options: • The `INCLUDING IDENTITY` and `INCLUDING GENERATED` options are not supported • `INCLUDING CONSTRAINTS` and `INCLUDING INDEXES` options are added

	PostgreSQL does not allow you to copy the structure of a view, using `CREATE TABLE ... (LIKE ...)`. For that, you may use another construct: `CREATE TABLE copytable AS SELECT * FROM viewname WHERE false` DOCUMENTATION
DB2	Behaves as if *inspired* by the standard. I.e., DB2 conforms to the standard, except: • the `LIKE ...` clause is stated *outside* any parenthesis • the extended `INCLUDING GENERATED` option is not supported • DB2 defaults to copy IDENTITY, DEFAULTS, and GENERATED properties, unless `EXCLUDING IDENTITY` and/or `EXCLUDING DEFAULTS` is specified. Example: `CREATE TABLE t2 LIKE t1 INCLUDING DEFAULTS` DB2 allows you to copy the structure of a view into a table. DOCUMENTATION
MSSQL	Does not support the standard. Instead, MSSQL has a special `SELECT ... INTO ... FROM ...` construct which can be combined with an impossible WHERE-clause to copy structure only: `SELECT * INTO t2 FROM t1 WHERE 1<>1` The source (*t1*) may be a view, as well as a table. `SELECT ... INTO` copies `NOT NULL` column attributes, but nothing else. DOCUMENTATION
MySQL	Complies with the core of the feature (T171), but not with the extended features (T173). MySQL does not allow you to copy the structure of a view into a table. DOCUMENTATION
Oracle	Does not support the standard. Oracle lets you copy a table structure using a special `CREATE TABLE ... AS` construct, combined with an impossible `WHERE`-clause: `CREATE TABLE t2 AS SELECT * FROM t1 WHERE 1<>1` DOCUMENTATION
Informix	On my TODO.

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:
	`DECLARE cursorname CURSOR FOR`
	`  SELECT ... FROM ... WHERE ...`
	`  ORDER BY column_name1,column_name2,...`
	The DBMS may additionally allow `ORDER BY` outside cursor definitions (optional feature IDs F850, F851, F852, F855).(Since SQL:2008)
	The standard doesn't specify how NULLs should be ordered in comparison with non-NULL values, except that any two NULLs are to be considered equally ordered, and that NULLs should sort either above or below all non-NULL values. However, the DBMS may optionally (as part of feature ID T611, "Elementary OLAP operations") allow the user to specify whether NULLs should sort first or last:
	`... ORDER BY ... NULLS FIRST`
	or
	`... ORDER BY ... NULLS LAST`

PostgreSQL	As well as in cursor definitions, it allows `ORDER BY` in other contexts. By default, NULLs are considered **higher** than any non-NULL value; however,(since version 8.3) this sorting behaviour may be changed by adding `NULLS FIRST` or `NULLS LAST` to the `ORDER BY` expression. DOCUMENTATION
DB2	As well as in cursor definitions, it allows `ORDER BY` in other contexts. NULLs are considered **higher** than any non-NULL value. DOCUMENTATION
MSSQL	As well as in cursor definitions, it allows `ORDER BY` in other contexts. NULLs are considered **lower** than any non-NULL value. DOCUMENTATION
MySQL	As well as in cursor definitions, it allows `ORDER BY` in other contexts. 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	As well as in cursor definitions, it allows `ORDER BY` in other contexts.

	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
Informix	As well as in cursor definitions, it allows `ORDER BY` in other contexts. NULLs are considered **lower** than any non-NULL 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.

By the way, *Use The Index, Luke!* has a page about this subject.

<table>
<tr>
<td>Standard</td>
<td>

The SQL standard provides three ways of performing a 'simple limit':

- Using **`FETCH FIRST`**:(since SQL:2008)

  Non-core feature IDs F856, F857, F858, and F859 describe using `SELECT ... FROM ... WHERE ... ORDER BY ... FETCH FIRST n ROWS ONLY`

  You may write `ROW` instead of `ROWS`.

- Using a **Window function**:(since SQL:2003)

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

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

</td>
</tr>
</table>

	- Using a **cursor**: 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	Supports all standards-based approaches.
	
	In old PostgreSQL versions (versions 8.3 and older), a special PostgreSQL (and MySQL) specific method was used:
	
	`SELECT columns`
	`FROM tablename`
	`ORDER BY key ASC`
	`LIMIT n`
	
	Note that `LIMIT` changes the semantics of `SELECT...FOR UPDATE`.
	
	Documentation:
	- FETCH FIRST/LIMIT
	- Window functions

DB2	Supports all standards-based approaches. Documentation: - OLAP functions - FETCH FIRST (general page about the SELECT statement; use your browser's search function to locate `FETCH FIRST`)

MSSQL	Supports the `ROW_NUMBER()` $^{\text{(since MSSQL 2005)}}$ and cursor standards-based approaches; doesn't support `FETCH FIRST`.
	
	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`
	The `TOP` construct is still available in MSSQL 2008, and it's handy for casual SQL work.
	
	Documentation

MySQL	Doesn't support the standard. Alternative solution:

	`SELECT columns`
	`FROM tablename`
	`ORDER BY key ASC`
	`LIMIT n`
	
	DOCUMENTATION

Oracle	Supports `ROW_NUMBER`; doesn't support `FETCH FIRST`.
	
	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`
	
	DOCUMENTATION
	
	A reader of this page told me that using the Oracle-specific `ROWNUM` 'magic' column yields better performance than using the ROW_NUMBER function. You may want to experiment with this. *Ask Tom* has an article on ROWNUM.

Informix	Supports neither ROW_NUMBER(), nor FETCH FIRST.
	
	Alternative solution (which is illegal in plain sub-queries):
	`SELECT FIRST n columns`
	`FROM tablename`
	`ORDER BY key ASC`
	
	DOCUMENTATION

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 |
|        6 | Vivian      |  77 |
+----------+-------------+-----+
```

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

```
+----------+-------------+-----+
|PERSON_ID | PERSON_NAME | AGE |
+----------+-------------+-----+
|        7 | Hilda       |  12 |
|        8 | Bill        |  12 |
|        4 | Joe         |  23 |
|        2 | Veronica    |  23 |
+----------+-------------+-----+
```

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 was 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`:

  ```
  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:

<table>
<tr>
<td></td>
<td>

If the DBMS doesn't support the elementary OLAP features, then the top-n solution may be obtained in an alternative way which is 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):

```
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.

</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>

Supports the fast standard SQL variant.

In version 8.3 and older, PostgreSQL only supported the slow standard SQL query variant. In practice, a PostgreSQL-only method was used instead, in order to obtain acceptable query performance:

```
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.)

DOCUMENTATION

</td>
</tr>
<tr>
<td>DB2</td>
<td>

Supports the fast standard SQL variant.

DOCUMENTATION

</td>
</tr>
<tr>
<td>MSSQL</td>
<td>Supports the fast standard SQL variant.</td>
</tr>
</table>

<table>
<tr><td></td><td>
MSSQL 2000 supported the slow standard SQL variant. In practice, a MSSQL-only expression had to be used instead, in order to obtain acceptable query performance:
<pre>
SELECT TOP 3 WITH TIES *
FROM person
ORDER BY age ASC
</pre>
(Change ASC to DESC in the position marked like this in order to get a <i>top-3 oldest</i> query instead.)
</td></tr>
<tr><td>MySQL</td><td>
Supports the slow standard SQL solution. In practice, this MySQL-specific solution should be used instead, in order to obtain acceptable query performance:
<pre>
SELECT *
FROM person
WHERE age <= COALESCE(-- note: no space between
"COALESCE" and opening parenthesis
  (
    SELECT age
    FROM person
    ORDER BY age ASC
    LIMIT 1 OFFSET 2   -- 2=n-1
  ),
  (
    SELECT MAX(age)
    FROM person
)
)
</pre>
(Change <= to >= and ASC to DESC and MAX to MIN in the positions marked like this in order to get a <i>top-3 oldest</i> query instead.)

The offset-value <i>2</i> is the result of <i>n-1</i> (remember: <i>n</i> 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 <i>n</i>.
</td></tr>
<tr><td>Oracle</td><td>
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:
<pre>
SELECT * FROM (
  SELECT
    RANK() OVER (ORDER BY age ASC) AS ranking,
    person_id,
    person_name,
    age
  FROM person
)
WHERE ranking <= 3
</pre>
(Change ASC to DESC in the position marked like this in order to get a <i>top-3 oldest</i> query instead.)

DOCUMENTATION
</td></tr>
<tr><td>Informix</td><td>On my TODO.</td></tr>
</table>

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**.

By the way, *Use the Index, Luke!* has a page about this.

Standard	The SQL standard provides three ways of performing 'limit with offset':
	- Using **`OFFSET`** and **`FETCH FIRST`**:(since SQL:2008)
	`SELECT...`
	`FROM ...`
	`WHERE ...`
	`ORDER BY ...`
	`OFFSET skip ROWS`
	`FETCH FIRST n ROWS ONLY`
	You may write `ROW` instead of `ROWS`.
	- Using a **window function**:(since SQL:2003)
	Non-core Feature ID T611 specifies *window functions*, one of which is `ROW_NUMBER() OVER`:
	`SELECT * FROM (`
	`  SELECT`
	`    ROW_NUMBER() OVER (ORDER BY key ASC) AS rownum,`
	`    columns`
	`  FROM tablename`
	`) AS foo`
	`WHERE rownum > skip AND rownum <= (n+skip)`
	- Using a **cursor**:
	You may use a *cursor* (core feature ID E121), if the programming environment permits it. This involves:
	○ `DECLARE cursor-name CURSOR FOR ...`
	○ `OPEN cursor-name`
	○ `FETCH RELATIVE number-of-rows-to-skip ...`
	○ `CLOSE cursor-name`

PostgreSQL	Supports all the standards-based approaches.
	In version 8.3 and older, cursors should be used, or a special construct:
	`SELECT columns`
	`FROM tablename`
	`ORDER BY key ASC`
	`LIMIT n OFFSET skip`

<table>
<tr><td></td><td>Documentation:
<ul>
<li>OFFSET...FETCH/LIMIT...OFFSET</li>
<li>Window functions</li>
</ul></td></tr>
<tr><td>DB2</td><td>Supports the <i>window function</i> based approach.

Regarding cursors: DB2 for Linux/Unix/Windows doesn't support <code>FETCH RELATIVE</code> (which is strange, because DB2 for the mainframe seems to support it). Instead, see if the DB2 driver for your programming environment supports <code>SQLFetchScroll()</code>.

Documentation: OLAP functions, the FETCH statement.</td></tr>
<tr><td>MSSQL</td><td>Supports the <i>window function</i> and cursor based approaches.

MSSQL 2000 didn't support <code>ROW_NUMBER()</code>; instead, a MSSQL-specific syntax had to be used:
<pre>
SELECT * FROM (
  SELECT TOP n * FROM (
    SELECT TOP z columns   -- (z=n+skip)
    FROM tablename
    ORDER BY key ASC
) AS FOO ORDER BY key DESC -- ('FOO' may be anything)
) AS BAR ORDER BY key ASC   -- ('BAR' may be anything)
</pre>
Documentation</td></tr>
<tr><td>MySQL</td><td>Doesn't support the standard approaches. Alternative solution:
<pre>
SELECT columns
FROM tablename
ORDER BY key ASC
LIMIT n OFFSET skip
</pre>
In older versions of MySQL, the LIMIT-syntax is less clear:
<pre>
... LIMIT [skip,] n
</pre>
(i.e. the <b><i>skip</i></b> argument is optional).
The old syntax is still supported by later MySQL versions (the old syntax is widely used).

Documentation</td></tr>
<tr><td>Oracle</td><td>Supports <code>ROW_NUMBER()</code>. I'm unsure if Oracle's cursor support is standards-compliant.

As Oracle doesn't accept <code>AS</code> 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 <code>ROWNUM</code> is a reserved word in Oracle, with special meaning. The Oracle code becomes:
<pre>
SELECT * FROM (
  SELECT
    ROW_NUMBER() OVER (ORDER BY key ASC) AS rn,
</pre></td></tr>
</table>

	`    columns`
	`  FROM tablename`
	`)`
	`WHERE rn > skip AND rn <= (n+skip)`
	
	DOCUMENTATION
	
	A reader of this page told me that using the Oracle-specific `ROWNUM` 'magic' column yields better performance than using the ROW_NUMBER function. You may want to experiment with this. *Ask Tom* has an article on ROWNUM.

Informix	Supports neither `OFFSET`...`FETCH FIRST` nor `ROW_NUMBER`. Supports cursors.
	
	An alternative to using cursors is to us an Informix-specific construct:
	`SELECT SKIP skip FIRST n *`
	`FROM tablename`
	
	DOCUMENTATION: SKIP and FIRST

Note:

FETCH FIRST/LIMIT/TOP 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;
 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
         6 | Vivian      |  77
```

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;
 person_id | person_name | age
-----------+-------------+-----
         7 | Hilda       |  12
         8 | Bill        |  12
         4 | Joe         |  23
         2 | Veronica    |  23
         9 | Marianne    |  27
         3 | Michael     |  27
         1 | Ben         |  50
        10 | Michelle    |  50
         5 | Irene       |  77
         6 | Vivian      |  77
```

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 legacy syntax in the example:

```
SELECT * FROM person ORDER BY age ASC LIMIT 5;
 person_id | person_name | age
-----------+-------------+-----
         7 | Hilda       |  12
         8 | Bill        |  12
         4 | Joe         |  23
         2 | Veronica    |  23
         3 | Michael     |  27

SELECT * FROM person ORDER BY age ASC LIMIT 5 OFFSET 5;
 person_id | person_name | age
-----------+-------------+-----
         3 | Michael     |  27
         1 | Ben         |  50
        10 | Michelle    |  50
         5 | Irene       |  77
         6 | Vivian      |  77
```

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 let the application cache the complete result set (e.g. in a *session* if your web application environment provides for sessions).

The INSERT statement

Inserting several rows at a time

<table>
<tr><td>Standard</td><td>

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 may be read as a shorthand for

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

</td></tr>
<tr><td>PostgreSQL</td><td>

Supported.(since version 8.2)

</td></tr>
<tr><td>DB2</td><td>

Supported.

</td></tr>
<tr><td>MSSQL</td><td>

Supported.(since version 2008)

</td></tr>
<tr><td>MySQL</td><td>

Supported.

</td></tr>
<tr><td>Oracle</td><td>

An Oracle-specific kludge:

```
INSERT INTO tablename
  SELECT 0,'foo' FROM DUAL
    UNION ALL
  SELECT 1,'bar' FROM DUAL
    UNION ALL
  SELECT 2,'baz' FROM DUAL
```

Alternatively:

```
INSERT ALL
  INTO tablename VALUES(0,'foo')
  INTO tablename VALUES(1,'bar')
  INTO tablename VALUES(2,'baz')
SELECT null FROM dual
```

</td></tr>
<tr><td>Informix</td><td>

On my TODO.

</td></tr>
</table>

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.

	(Take care if you use TINYINT(1) and 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. Side note: MySQL has a `BIT` type which may be interesting for people with enormous amounts of boolean-type data. DOCUMENTATION
Oracle	Doesn't support the BOOLEAN type. Judging from various JDBC documentation and a discussion at *Ask Tom*, 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).
Informix	On my TODO.

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:

```
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:

```
SELECT * FROM chartest; -- should show two rows
CHARVAL1   CHARVAL2   VARCHARVAL
========== ========== ==============================
aaa        aaa        aaa
aaaaaa     aaa        aaa


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


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	Stores CHARs in space padded form, but violates the standard by (conceptually) truncating trailing white-space before performing most functions, operators, and comparisons (like the `CHARACTER_LENGTH`-function and the concatenation(`||`) operator). DOCUMENTATION

DB2	Follows the standard. DOCUMENTATION
MSSQL	Generally follows standard, but (conceptually) truncates trailing white-space before performing some functions (at least before `LEN()`). DOCUMENTATION
MySQL	Breaks the standard by silently inserting the string, truncated to specified column CHAR-length. (It's actually not completely silent, as it issues warnings if values were truncated: If you manually check for warnings, you will know that something bad happened, but not which of the rows are now invalid.) 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`, `CHAR_LENGTH`, etc. DOCUMENTATION
Oracle	Follows the standard, with a minor exception: Oracle doesn't remove trailing spaces which exceed the specified CHAR length, but raises an exception. DOCUMENTATION
Informix	On my TODO.

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 F411): TIMESTAMP WITH TIME ZONE which also stores the time zone. Examples of TIMESTAMP literals: • `TIMESTAMP '2003-07-29 13:19:30'` • `TIMESTAMP '2003-07-29 13:19:30.5'` 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:
	In some cases, `TIMESTAMP '2003-08-23 01:02:03 +02:00'` is interpreted as a TIMESTAMP WITH**OUT** 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:
	`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

DB2	DB2 has the TIMESTAMP data type, but not the extended TIMESTAMP WITH TIME ZONE type.
	
	Performs good sanity checks on inserted timestamp values; e.g. this will work:
	`INSERT INTO tablename (columnname)`
	`VALUES ('2003-02-28 00:05:00')`
	while this will *fail*:
	`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:
	`INSERT INTO tablename (columnname)`
	`VALUES ('2003-02-28 00:05:00')`
	while this will *fail*:
	`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 '2003-02-2**9** 00:05:00' and '2003-01-3**2** 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

Informix	On my TODO.

SQL functions

CHARACTER_LENGTH

Standard	`CHARACTER_LENGTH(argument)`
	If the optional feature T061 is implemented, the function may be augmented with an indication of *string unit*:
	`CHARACTER_LENGTH(argument USING string-unit)`
	string-unit may be `UTF8`, `UTF16`, `UTF32`.
	
	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	Has a CHARACTER_LENGTH function, but it's non-compliant because it requires indication of *string unit*, and db2's string units are different from the standard's. Provides the `LENGTH` function for those who don't want to think about string units, but the LENGTH function may return wrong results in UTF-8 databases. 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(foo))` if you want the length without trailing spaces. DOCUMENTATION: CHARACTER_LENGTH and LENGTH
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
Informix	On my TODO.

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: `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* (The standard 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:
	`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:
	`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
	`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
	`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.
	DOCUMENTATION

DB2	Provides (since version 9) the `SUBSTRING` function, but requires you to indicate string unit by appending "`USING unit`".
	The unit identifier may be `CODEUNITS16`, `CODEUNITS32`, or `OCTETS`. `CODEUNITS16`/`CODEUNITS32` seem non-standard. The standard's `CHARACTERS` unit isn't supported by DB2.
	Example:
	`SELECT SUBSTRING(somecolumn FROM 3 USING OCTETS) FROM sometable`
	`SELECT SUBSTRING(somecolumn FROM 3 FOR 2 USING OCTETS) FROM sometable`
	For old DB2 versions, use the non-standard `SUBSTR` function.
	DB2 doesn't provide any built-in regular expression facilities at all (but you may manually add PCRE capabilities).
	DOCUMENTATION: SUBSTRING and SUBSTR

MSSQL	MSSQL has a `SUBSTRING` function, but its syntax differs from that of the standard. The syntax is: `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. MSSQL has no regular expression functionality. Documentation
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. Documentation
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. Documentation: SUBSTR and REGEXP_SUBSTR.
Informix	On my TODO.

Note: If you find yourself using SUBSTRING in a WHERE-expression, then consider if LIKE could be used instead: The use of LIKE will typically make your DBMS try to use an index, whereas it will typically not try to do so in connection with functions.

REPLACE

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

PostgreSQL	Follows *de facto* standard. DOCUMENTATION
DB2	Follows *de facto* standard. DOCUMENTATION
MSSQL	Follows *de facto* standard with the exception that MSSQL by default works case **in**sensitively. DOCUMENTATION
MySQL	Follows *de facto* standard. MySQL even works case **sensitively**.[1] Note that the `REPLACE`-function is different from MySQL's non-standard `REPLACE` **`INTO`** expression. DOCUMENTATION
Oracle	Follows *de facto* standard. DOCUMENTATION
Informix	On my TODO.

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 **in**sensitively in plain WHERE-clauses involving string comparisons.

TRIM

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. DOCUMENTATION
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). DOCUMENTATION
MySQL	Follows the standard. DOCUMENTATION
Oracle	Follows the standard.

Informix	On my TODO.

Concatenation

Standard	Core feature ID E021-07: Concatenating two strings is done with the `||` operator: `string1 || string2` If at least one 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. DOCUMENTATION
DB2	Follows the standard, partly. Does not automatically cast concatenated values into compatible types. DOCUMENTATION
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. DOCUMENTATION
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. DOCUMENTATION
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. DOCUMENTATION

Informix	Follows the standard. Automatically casts numeric data into character data, if needed. If an operand is NULL then the result is NULL. DOCUMENTATION

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 the standard-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. DOCUMENTATION
DB2	Follows the non-optional parts of the UNIQUE-constraint. Doesn't implement the optional NULLs allowed feature. DOCUMENTATION (see the *unique-constraint* section of the page).
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. DOCUMENTATION
MySQL	Follows the standard, including the optional NULLs allowed feature.

Oracle	Follows the standard—with a twist regarding multiple-column UNIQUE-constraints: 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 DOCUMENTATION
Informix	On my TODO.

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.

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.

<table>
<tr>
<td></td>
<td>

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.

</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>

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

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 will 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.

If you want semantics like the standard's `GENERATED ALWAYS AS IDENTITY`, then `SERIAL` will not do it; instead you need to:

1. Create a sequence for the table (assuming that the table is called *footab*, having a an integer column called *id*):
   ```
   CREATE SEQUENCE footab_id_seq
   ```
2. Add the PL/pgSQL language to the database, in case it doesn't already exist (extra additions don't hurt):
   ```
   CREATE LANGUAGE plpgsql
   ```
3. Create a function to be called by a trigger when *footab* is changed:
   ```
   CREATE OR REPLACE FUNCTION protect_footab_id() RETURNS TRIGGER AS $$
   BEGIN
     IF tg_op = 'INSERT' THEN
       IF new.id IS NOT NULL THEN
         RAISE EXCEPTION 'setting ID manually not allowed (%)', new.id;
       END IF;
       new.id = NEXTVAL('footab_id_seq');
     ELSE
       IF new.id IS DISTINCT FROM old.id THEN
         RAISE EXCEPTION 'changing ID is not allowed (% to %)', old.id,
   new.id;
       END IF;
     END IF;
   ```

</td>
</tr>
</table>

	`    RETURN NEW;`
	`  END;`
	`  $$ LANGUAGE PLPGSQL`
	4. Create the trigger, calling the above function:
	`CREATE TRIGGER protect_footab_id`
	`  BEFORE INSERT OR UPDATE ON footab`
	`  FOR EACH ROW EXECUTE PROCEDURE protect_footab_id()`
	
	Another option is to add the `WITH OIDS` clause when creating a table. Object identifiers (OIDs) will then be added to a special *oid* column which is hidden by default, i.e. isn't included in `SELECT * FROM ...` result sets). The *oid* column can be revealed by explicitly adding it to the `SELECT`-list, and it can be referred to in `WHERE` clauses. OIDs cannot be assigned by the user, so the semantics of OIDs resemble the standard's `GENERATED` **`ALWAYS`** `AS IDENTITY` attribute.
	
	DOCUMENTATION: The SERIAL and OIDs 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 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 the value, unless the user has first run `SET IDENTITY_INSERT tablename ON` MSSQL refuses to update values in IDENTITY columns.
	
	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:
	
	`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

	<largest value in column>+<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:
	`CREATE TABLE tablename (`
	`  columnname INTEGER AUTO_INCREMENT PRIMARY KEY`
	`  ...`
	`)`
	Standard 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:
	`CREATE TABLE mytable (`
	`  mytable_id INTEGER PRIMARY KEY,`
	`  ... -- (other columns)`
	`);`
	`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`.

Informix	On my TODO.

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

Bulk operations

TRUNCATE TABLE

Often, it's useful to be able to remove all rows from a large table in a quick way. And often, `DELETE` isn't as quick as you'd like it to be. So several DBMSes implement a `TRUNCATE` operation. Typically, truncating means that deletion isn't associated with triggers which may exist for the table, and typically, truncating involves little (if any) transaction log activity.

Standard	The SQL standard defines the `TRUNCATE TABLE` *`tablename`* statement (optional feature ID F200, new in SQL:2008) as: Delete all rows of a base table without causing any triggered action. Unfortunately, the standard doesn't specify 1. whether `TRUNCATE TABLE` should be allowed in a transaction involving other statements, or not 2. whether `TRUNCATE TABLE` should imply an immediate `COMMIT`, or not
PostgreSQL	Follows the standard. In PostgreSQL, `TRUNCATE TABLE` is *allowed in a transaction* involving other operations, and `TRUNCATE TABLE` does *not* imply an immediate `COMMIT` operation.

	See the documentation for variations and restrictions. Most importantly, you need to have be owner of the table to be truncated (or work as a superuser); alternatively, you need to have TRUNCATE privilege on the table. Note also the nice—but potentially dangerous—`CASCADE` modifier which may be useful for emptying related tables. DOCUMENTATION

DB2	Almost follows the standard.(since version 9.7)
	DB2 requires that the `IMMEDIATE` keyword be added the the ordinary `TRUNCATE TABLE` statement, e.g.:
	`TRUNCATE TABLE someschema.sometable IMMEDIATE`
	`TRUNCATE TABLE` must be the first statement in a transaction. A transaction starting with `TRUNCATE TABLE` may include other statements, but if the transaction is rolled back, the `TRUNCATE TABLE` operation is not undone.
	DB2s `TRUNCATE TABLE` operation has a number of optional arguments, see the documentation for more on this; especially, the `REUSE STORAGE` argument may be important for ad-hoc DBA tasks.
	
	In DB2 versions < 9.7, you may ~~ab~~use the `IMPORT` statement. Unfortunately, you need to know which operating system the command is executed from for this to work:
	
	• On unix-like systems:
	`IMPORT FROM /dev/null OF DEL REPLACE INTO tablename`
	• On Windows:
	`IMPORT FROM NUL OF DEL REPLACE INTO tablename`
	
	`IMPORT` cannot be ~~ab~~used in all contexts. E.g., when working with dynamic SQL (from Java/.NET/PHP/...—not using the `db2` command line processor), you need to wrap the `IMPORT` command in a call to `ADMIN_CMD`, e.g.:
	`CALL ADMIN_CMD('IMPORT FROM /dev/null OF DEL REPLACE INTO tablename')`
	
	`IMPORT` seems to be *allowed in a transaction* involving other operations, however it implies an immediate `COMMIT` operation.
	
	The `ALTER TABLE` command may also be ~~ab~~used to quickly empty a table, but it requires more privileges, and may cause trouble with rollforward recovery.
	
	DOCUMENTATION:
	
	• `TRUNCATE TABLE`
	• `IMPORT`
	• `IMPORT` through `ADMIN_CMD`

MSSQL	Follows the standard.

	In MSSQL, TRUNCATE TABLE is *allowed in a transaction* involving other operations, and TRUNCATE TABLE does *not* imply an immediate COMMIT operation. You need to have at least ALTER-permission on the table to be truncated. Documentation
MySQL	MySQL has a TRUNCATE TABLE statement, but it doesn't always follow the standard. Note that in some cases, MySQL's truncate command is really the equivalent of an unrestricted DELETE command (i.e.: potentially slow and trigger-invoking). Its behaviour depends on which *storage engine* the table is managed by. When using InnoDB (transaction safe) tables, TRUNCATE TABLE is *allowed in a transaction* involving other operations, however TRUNCATE TABLE *implies* an immediate COMMIT operation. Documentation
Oracle	Follows the standard. Note, that the TRUNCATE TABLE *implicitly* commits the current transaction. You may find that TRUNCATE TABLE isn't nearly as quick as expected; in this case, consider using TRUNCATE TABLE *tablename* **REUSE STORAGE** instead. Needed privileges—Quoting from the documentation: *...the table or cluster must be in your schema or you must have DROP ANY TABLE system privilege.* Documentation
Informix	On my TODO.

Command line procedures / metadata

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 GUIs 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: `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. Documentation

DB2	Run:
	`db2 -t`
	(The `-t` argument tells the command line processor to a semicolon as statement terminator instead of the default (newline). This allows for multi-line SQL statements.)
	
	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:
	`. /home/db2inst1/sqllib/db2profile`
	
	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 a 'utility' called `db2batch` which some might find at bit nicer to work with.
	
	Documentation

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 sqlsmd: It has command history, so you may press arrow-up for previous commands in the current sqlsmd 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, except it doesn't seem to support Kerberos, so you need to log into the database using a database-account (not a Windows-account). Documentation
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 has very limited built-in command history handling. The bad command history handling is especially bad on Linux/unix; fortunately, this can be fixed using rlwrap. Documentation A unique feature of Oracle is that a web-based administration interface is provided, as a supplement to the local administration software. The URL to the interface is typically • in version 11: `https://hostname:1158/em/` • in version 12: `https://hostname:5500/em/`
Informix	Informix' command line utility is called `dbaccess`. If run without arguments, it starts a menu system. If you simply want to shoot SQL statements off to the database, another form is more convenient, at least on unix: `echo 'SELECT foo FROM bar' | dbaccess databasename` Documentation

Getting a list of databases

Standard	Not specified, as far as I know. (By the way: The SQL standard doesn't have the concept of a *database* as a container of schemas; instead, the standard specifies that schemas are contained in a *catalog*.)
PostgreSQL	Using SQL: `SELECT datname FROM pg_catalog.pg_database` When working in the `psql` command line interface: `\l` or `\l+`

	Alternative (when working from the terminal, not in `psql`): `psql --list` DOCUMENTATION: The psql tool, the pg_database catalog.
DB2	Offers the `LIST DATABASE DIRECTORY` command, but only when working in the `db2` command line processor (i.e. not when working from `db2batch`); this command's output is human readable, but sub-optimal as machine readable format. DOCUMENTATION
MSSQL	`EXEC SP_HELPDB` DOCUMENTATION
MySQL	`SHOW DATABASES` DOCUMENTATION
Oracle	In Oracle, there is a one-to-one relationship between *databases* and *instances* (unless you work with a clustered Oracle system). You can get a list of instances; the way to do it depends on the operating system which Oracle is running on: • On unix-like systems: Look in the `/etc/oratab` file. • On Windows: Start Windows' *Services* management console and look for services with names starting with `OracleService`**xxxx**. Each `xxxx` is the name (AKA *SID*) of an instance. Documentation: ORATAB
Informix	Connect to the *sysmaster* database (all users are allowed to do this) and run: `SELECT name FROM sysmaster:sysdatabases` DOCUMENTATION

Getting a list of schemas

Standard	`SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA`
PostgreSQL	In the command line interface: `\dn` or `\dn+` (for more details). Using SQL: Follows the standard. DOCUMENTATION: • The `psql` tool • The schemata INFORMATION_SCHEMA view
DB2	`SELECT schemaname FROM syscat.schemata` DOCUMENTATION

MSSQL	Follows the standard. Documentation
MySQL	MySQL doesn't support schemas.
Oracle	Oracle has a peculiar approach to schemas: A schema exists for each and every user. And there cannot be a schema without a corresponding user. Consequently, a way to get a list of schemas in Oracle is to query the `ALL_USERS` dictionary view: `SELECT username FROM all_users` Documentation
Informix	Informix' concept of schemas is closely related to user names, so—somewhat surprisingly—the query is: `SELECT UNIQUE owner FROM systables WHERE tabid>99` Documentation

Getting a 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. 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`

	These commands are only available in the `db2` command line processor (i.e. not from—e.g.— `db2batch`). DOCUMENTATION
MSSQL	Follows that standard. Sometimes, the `SP_TABLES` system stored procedure is easier to use. DOCUMENTATION: - The INFORMATION_SCHEMA.TABLES view - `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 'INFORMATION_*DATABASE*' 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 in the current schema:
	`SELECT * FROM tab`
	Use of the *tab* dictionary view is officially deprecated, though. The following query takes longer to write, but is more future proof:
	`SELECT owner||'.'||table_name FROM all_all_tables`
	(Remember that in Oracle, there is a one-to-one relationship between 'owners' and schemas.)
	
	DOCUMENTATION

Informix	Doesn't provide the standard `INFORMATION_SCHEMA` out of the box. A few of the standard's INFORMATION_SCHEMA views may be added by running a special script, though.
	
	Informix offers a set of *system catalogs* instead. To get a list of tables:
	`SELECT tabname FROM systables WHERE tabid > 99`
	
	The above query will include views and other objects; if you want base tables only:
	`SELECT tabname FROM systables WHERE tabid > 99 AND tabtype='T'`
	
	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`*`'`

<table>
<tr>
<td></td>
<td>

```
  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_SCHEMA.CONSTRAINT_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_SCHEMA.CONSTRAINT_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 remark`".

See also: Warning about potential case sensitivity problems above.

</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>

Follows the standard, except for some gotchas mentioned above.

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

```
\d tablename
```

</td>
</tr>
<tr>
<td>DB2</td>
<td>

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
```

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>

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=UCASE('schemaname') AND
  tc.tabname=UCASE('tablename')
ORDER BY const_name,col_name
```

DOCUMENTATION:

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

</td>
</tr>
<tr>
<td>MSSQL</td>
<td>

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:2008'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`

</td>
</tr>
<tr>
<td>MySQL</td>
<td>

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*'.

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>

- MySQL's INFORMATION_SCHEMA doesn't have all SQL:2008'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 `TABLE_CATALOG` is `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:

```
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'
```

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

```
DESCRIBE tablename
```

DOCUMENTATION:

- The `INFORMATION_SCHEMA`
- DESCRIBE

</td>
</tr>
<tr>
<td>Oracle</td>
<td>

Doesn't provide the standard INFORMATION_SCHEMA. Offers *data dictionary views* instead.

To get (very) basic information:

```
DESCRIBE tablename
```

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

```
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,
```

</td>
</tr>
</table>

```
  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.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;
```

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

```
COLUMN index_name  FORMAT a11;
COLUMN type        FORMAT a8;
COLUMN uniness     FORMAT a9;
COLUMN column_name FORMAT a20;
      SELECT index_name,
             index_type type,
             uniqueness uniness,
             column_name
        FROM user_indexes ui
NATURAL JOIN user_ind_columns uic
       WHERE dropped='NO'
         AND table_name=upper('tablename')
    ORDER BY index_name,column_name
```

DOCUMENTATION:

- DESCRIBE sqlplus command
- COLUMN sqlplus command
- Static Data Dictionary Views
- USER_CONSTRAINTS data dictionary view
- USER_CONS_COLUMNS data dictionary view
- USER_INDEXES data dictionary view
- USER_IND_COLUMNS data dictionary view

Informix

Doesn't provide the standard `INFORMATION_SCHEMA` out of the box. If a special script is run, an INFORMATION_SCHEMA may be added which allows for using the most basic standards-based table description query.

In practice, an Informix-only query is used. The following query provides very basic table information, excluding constraints:

```
SELECT
       colname,
       coltype,
       CASE
         WHEN (coltype-256)<0 THEN 'YES'
         ELSE                      'NO'
```

	`       END        AS nullable`
	`FROM   systables  AS a`
	`  JOIN syscolumns AS b   ON a.tabid = b.tabid`
	`WHERE tabname='`***`tablename`***`'`
	
	Notice that the table name is in lower case. The *colname* values are numeric codes which need to be looked up in order to provide meaning.
	
	DOCUMENTATION

Manually 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).

Standard	Not standardized.
PostgreSQL	`ANALYZE `*`tablename`* If the *tablename* parameter is left out, then statistics are gathered for all tables in the current database. DOCUMENTATION
DB2	`RUNSTATS ON TABLE `*`schema-name.table-name`*` AND INDEXES ALL` (many variations/options available) The `RUNSTATS` command needs to be invoked in a special way if you aren't using the `db2` command line processor, namely through the `ADMIN_CMD` procedure. DOCUMENTATION: `RUNSTATS` and `RUNSTATS` wrapped in `ADMIN_CMD`.
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 `CREATE STATISTICS` step is not needed for indexed columns. Thus, this step may be skipped if you are satisfied with keeping statistics on indexed columns only.) 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 `*`tablename`* 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.
	`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:
	`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

Informix	On my TODO.

Getting a query explanation

Standard	Not standardized.
PostgreSQL	`EXPLAIN <query>` 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:
	`db2expln -database databasename -stmtfile query.sql -terminator ';' -terminal`
	In the above example, the query has been saved to a file called "query.sql".
	In some situations, you may want to use the `dynexpln` utility instead of `db2expln`. And in yet other situations, the `db2exfmt` tool is a better choice. 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 `sqllib/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:
	`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	`EXPLAIN PLAN FOR <query>`
	After the query has run, do the following to get the plan explanation:
	`SELECT plan_table_output FROM table(dbms_xplan.display())`
	DOCUMENTATION

Informix	On my TODO.

Turning on query timing

Standard	Not standardized.
PostgreSQL	`\timing` DOCUMENTATION
DB2	Run the query in the "`db2batch`" command line processor; `db2batch` prints the elapsed time of each query. DOCUMENTATION
MSSQL	`SET STATISTICS TIME ON` DOCUMENTATION
MySQL	MySQL's command line interface prints query times by default.
Oracle	`SET TIMING ON` DOCUMENTATION
Informix	On my TODO.

JDBC

JDBC driver jar file name, and general documentation

PostgreSQL	The PostgreSQL JDBC Driver: postgresql-***postgresqlversion-jdbcbuild#***.jdbc4.jar DOCUMENTATION
DB2	IBM Data Server Driver for JDBC: db2jcc.jar (included in default DB2 client software installations; may also be downloaded separately, after registration) DOCUMENTATION
MSSQL	Microsoft's driver: sqljdbc.jar Alternative: The open source JTDS driver: jtds-***version***.jar DOCUMENTATION: • Microsoft's driver • The jTDS driver
MySQL	The MySQL Connector/J driver: mysql-connector-java-***version***-bin.jar DOCUMENTATION
Oracle	Oracle's JDBC drivers: ojdbc5.jar (for Java 5), ojdbc6.jar (for Java 6) DOCUMENTATION
Informix	IBM's Informix JDBC driver: ifxjdbc.jar (download requires registration and filling out annoying questionnaires, and an installer which only works with some JREs has to be run to unpack the driver) DOCUMENTATION

JDBC driver class name

PostgreSQL	org.postgresql.Driver DOCUMENTATION
DB2	com.ibm.db2.jcc.DB2Driver
MSSQL	Microsoft's driver: com.microsoft.sqlserver.jdbc.SQLServerDriver jTDS' driver: net.sourceforge.jtds.jdbc.Driver
MySQL	com.mysql.jdbc.Driver DOCUMENTATION
Oracle	oracle.jdbc.driver.OracleDriver
Informix	com.informix.jdbc.IfxDriver

JDBC connection URL

PostgreSQL	jdbc:postgresql://***hostname***/***DBname*** DOCUMENTATION
DB2	jdbc:db2://***hostname***:***50000***/***DBname*** or (if the database is on the local host): jdbc:db2:***DBname*** DOCUMENTATION
MSSQL	jTDS' driver: jdbc:jtds:sqlserver://<server>[:<port>][/<database>][;<property>=<value>[;...]] If you need to connect to a *named instance*, see add the instance name like this: jdbc:jtds:sqlserver://<server>[:<port>][/<database>]**;instance=INSTANCE_NAME**[;<property>=<value>[;...]]
MySQL	jdbc:mysql://[***host***][,***failoverhost***][:***port***]/[***database***]?user=***username***&password=***password*** DOCUMENTATION
Oracle	jdbc:oracle:thin:@*hostname*:*1521*:*instancename*
Informix	jdbc:informix-sqli://*hostname*:*9088*/*DBname*:INFORMIXSERVER=*instancename* Use port 1526 instead of 9088 if the Informix version is <11.

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 dummy-table
```

Standard	On my TODO.
PostgreSQL	No need for dummy-table. In addition, the `VALUES` keyword may be used to produce a simple result set, without introducing a `FROM` clause, e.g. `VALUES(1+1)` (Note the missing SELECT and FROM keywords). DOCUMENTATION
DB2	Dummy-table: `SYSIBM.SYSDUMMY1`.

	In addition, the `VALUES` keyword may be used to produce a simple result set, without introducing a `FROM` clause, e.g. `VALUES(1+1)` (Note the missing SELECT and FROM keywords). Documentation
MSSQL	No need for dummy-table.
MySQL	No need for dummy-table, although MySQL allows you to refer to a `DUAL` dummy-table (for Oracle compatibility).
Oracle	Dummy-table: `DUAL`.
Informix	Informix requires that you include a `FROM` specification. In recent versions of Informix(since version 11.10), a dummy table has been included: `sysmaster:sysdual`. For older Informix versions, the tradition is to use code like: `SELECT ... FROM systables WHERE tabid=1` This code makes use of the fact that the *systables* table is guaranteed to contain a row where *tabid* equals 1. Documentation: • The sysdual table • The systables table

Obtaining DBMS version

Standard	`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: `SELECT VERSION()` Documentation
DB2	`SELECT service_level FROM SYSIBMADM.ENV_INST_INFO` —or run the special `db2level` program. Documentation: `SYSIBMADM.ENV_INST_INFO` and `db2level`

MSSQL	MSSQL's implementation of the IMPLEMENTATION_SCHEMA doesn't seem to include the SQL_IMPLEMENTATION_INFO view. In stead, you may use
	`SELECT SERVERPROPERTY('ProductVersion')`
	(just the version), or
	`SELECT @@VERSION`
	(verbose, harder to parse).

	DOCUMENTATION: SERVERPROPERTY, @@VERSION
MySQL	MySQL's `INFORMATION_SCHEMA` doesn't include the `SQL_IMPLEMENTATION_INFO` view. Work-around: `SELECT VERSION()` DOCUMENTATION

Oracle	`SELECT banner FROM v$version`
	
	v$version will not reveal patch set updates. For details which include exact patch level, use the *opatch* command-line tool (which might exist as /u01/app/oracle/product/12.1.0/dbhome_1/OPatch/opatch, or similar):
	`opatch lsinventory`
	
	Documentation: V$VERSION

Informix	Using SQL: `SELECT dbinfo('version','full') FROM systables WHERE tabid=1` From the command line: `onstat -` DOCUMENTATION: • The dbinfo function • The onstat utility

Standard TCP/IP port

Product	Port#	Notes	Documentation
Standard	Not specified		
PostgreSQL	5432	For security reasons, PostgreSQL doesn't listen to non-local TCP interfaces by default.	Documentation
DB2	50000		
MSSQL	1433	By default, MSSQL Express Edition doesn't listen for TCP connections.	
MySQL	3306		
Oracle	1521		Documentation
Informix	9088 (unencrypted)	Informix versions prior to version 11, the default port was 1526.	

Diagnostic log

Each DBMS has different ways to record diagnostic information (event logs).

Standard	Not covered by the standard.
PostgreSQL	By default, PostgreSQL logs to stderr, meaning that it's highly installation specific where the dianostic information is put; on this author's system, the default ends up in `/var/lib/pgsql/pgstartup.log`. The default can be set to something more reasonable (such as `syslog` on unix, `eventlog` on Windows) by adjusting the `log_destination` configuration parameter. DOCUMENTATION
DB2	On unix systems, DB2s diagnostic log file is called `db2diag.log` and lives in the `sqllib/db2dump` sub-directory of the instance user's home directory. I.e., a typical full path is: `/home/db2inst1/sqllib/db2dump/db2diag.log` If the file is renamed or deleted, DB2 will create a new `db2diag.log` without having to be restarted. DOCUMENTATION
MSSQL	On my TODO.
MySQL	On my TODO.
Oracle	A diagnostic directory contains a file called `alert_`***`INSTANCE`***`.log`. The diagnostic directory is determined by the following query: `SELECT value FROM v$parameter WHERE name='background_dump_dest'` Examples of the diagnostic directory: • On an Oracle 11gR2 running on Linux: `/usr/local/oracle/diag/rdbms/`***`INSTANCE`***`/`***`INSTANCE`***`/trace` • On an Oracle 9.2 running on Windows: `D:\oracle\admin\`***`INSTANCE`***`\bdump` DOCUMENTATION
Informix	The path of the diagnostic log is defined by the MSGPATH configuration parameter. On a Linux installation, using default options: `/opt/IBM/informix/tmp/online.log` DOCUMENTATION