

Structured Query Language (SQL)

Mariano Beiró

Dpto. de Computación - Facultad de Ingeniería (UBA)

12 de abril de 2022

Temas

1 Introducción

2 Definición de Datos en SQL

3 Manipulación de Datos en SQL → EXPLAIN (el gestor muestra la query plan)

- **SELECT...FROM...WHERE**

- **JOIN**

- Operaciones de conjuntos

- Ordenamiento y paginación

- **GROUP BY...HAVING**

- Consultas anidadas

- **ABM**

- **DROP's**

4 Funciones y estructuras auxiliares

5 Bibliografía

1 Introducción

2 Definición de Datos en SQL

3 Manipulación de Datos en SQL

- `SELECT...FROM...WHERE`
- `JOIN`
- Operaciones de conjuntos
- Ordenamiento y paginación
- `GROUP BY...HAVING`
- Consultas anidadas
- ABM
- `DROP's`

4 Funciones y estructuras auxiliares

5 Bibliografía

Introducción

- Recordemos que los **lenguajes** son las herramientas a través de las cuales interactuamos con los modelos de datos.
- En el contexto de las bases de datos los lenguajes se dividen en varios tipos:



- Los **lenguajes de definición de datos** nos permiten expresar la estructura y las restricciones de nuestro modelo de datos.
- Los **lenguajes de manipulación de datos** nos permiten ingresar, modificar, eliminar y consultar datos en nuestro modelo.
- Los **lenguajes de control de datos** manejan cuestiones vinculadas con los permisos de acceso a los datos.

Características

SQL como DML

- El lenguaje **SQL (Structured Query Language)** es hoy en día el estándar para la operación de bases de datos relacionales.
- Es tanto un lenguaje de definición de datos (DDL) como de manipulación de datos (DML).
- Como lenguaje de manipulación de datos, SQL:
 - Es no procedural.
 - Está basado en el cálculo relacional de tuplas.
- Para ponernos de acuerdo...

Modelo relacional	SQL
Relación	Tabla
Tupla	Fila
Atributo	Columna

Ejemplo

- Supongamos que queremos obtener los datos de las películas que ganaron algún Oscar:

PELÍCULAS

nombre_película	año	nombre_director	cant_oscars
Kill Bill	2003	Quentin Tarantino	0
Django Unchained	2012	Quentin Tarantino	2
Star Wars III	2005	George Lucas	0
Coco	2017	Lee Unkrich	2

```
SELECT *
FROM Peliculas
WHERE cant_oscars > 0;
```



nombre_película	año	nombre_director	cant_oscars
Django Unchained	2012	Quentin Tarantino	2
Coco	2017	Lee Unkrich	2

Historia

- La versión original de SQL se denominaba SEQUEL (*Structured English Query Language*) y fue desarrollada por IBM Research (1974).
- Fue implementado en el System R de IBM (1977). El primer producto comercial basado en SEQUEL fue Oracle (1979).

- 1986 • Estándar ANSI-SQL.
- 1987 • Estándar ISO-SQL.
- 1989 • Una pequeña revisión introduce la definición de restricciones de integridad.
- 1992 • Sale la segunda versión del estándar. Aparecen numerosos tipos de datos: DATE, TIME, VARCHAR. Se introduce soporte para transacciones, esquemas, el NATURAL JOIN y la consulta de información de catálogo (INFORMATION_SCHEMA).
- 1999 • El SQL::1999 (SQL3) introduce numerosas mejoras: tipos de dato booleanos, la clausura transitiva (WITH RECURSIVE), capacidades analíticas OLAP (CUBE, ROLLUP) y soporte para control de acceso basado en roles (RBAC). Las versiones comienzan a dividirse en varias partes.
- 2003 • SQL::2003. Algunas novedades son: el CREATE TABLE AS SELECT, los MERGE de tablas, los tipos de dato XML y el mapeo SQL/XML.
- 2006 • SQL::2006. Crece la integración con XML, permitiendo la creación y manipulación de documentos XML desde SQL.
- 2008 • El estándar SQL::2008 introduce soporte para *pattern matching* y triggers de INSTEAD OF.
- 2011 • (SQL::2011) Soporte para bases de datos temporales. Claves primarias y foráneas con validez temporal. Predicados temporales.
- 2016 • (SQL::2016) Última y octava versión del estándar. Nuevas funcionalidades para JSON y *pattern matching* de filas.

Estructura del estándar

- El estándar ISO SQL tiene actualmente 9 partes: [SQL]
 - ISO/IEC 9075-1: Framework (SQL/Framework)
 - **ISO/IEC 9075-2: Foundation (SQL/Foundation)**
 - ISO/IEC 9075-3: Call-Level Interface (SQL/CLI)
 - ISO/IEC 9075-4: Persistent Stored Modules (SQL/PSM)
 - ISO/IEC 9075-9: Management of External Data (SQL/MED)
 - ISO/IEC 9075-10: Object Language Bindings (SQL/OLB)
 - **ISO/IEC 9075-11: Information and Definition Schemas (SQL/Schemata)**
 - ISO/IEC 9075-13: SQL Routing and Types using Java (SQL/JRT)
 - ISO/IEC 9075-14: XML Related Specifications (SQL/XML)
- Se llama **Core SQL** a los requerimientos incluidos en las partes 2 y 11. Un SGBD que respeta el *Core SQL* se dice que cumple con la *conformance* mínima al estándar.
- El estándar SQL es abierto pero *no es gratuito*. En <http://modern-sql.com/standard> pueden encontrar *drafts* muy cercanos a la publicación de algunas versiones.

Características

SQL como gramática libre de contexto

[SQL PART1 6; SQLGRAM]

- SQL es una **gramática libre de contexto** (*context-free grammar, CFG*). Ésto implica que su sintaxis puede ser descrita a través de reglas de producción.
- Una de las notaciones más conocidas para CFG's es la **notación de Backus-Naur** (*Backus-Naur form, BNF*). Esta es la notación adoptada en el estándar.
- El sitio <https://jakewheat.github.io/sql-overview/sql-2011-foundation-grammar.html> recopila **la gramática** de la Parte 2 del estándar.

SQL como gramática libre de contexto

Especificación: Ejemplo

```

<query specification> ::=
    SELECT [ <set quantifier> ] <select list> <table expression>
           distinct          *          from where having ...

<set quantifier> ::=
    DISTINCT
    | ALL

<select list> ::=
    <asterisk>
    | <select sublist> [ { <comma> <select sublist> }... ]

<table expression> ::=
    <from clause>
    [ <where clause> ]
    [ <group by clause> ]
    [ <having clause> ]
    [ <window clause> ]
  
```

Recursos utilizados en esta clase

- Stack Exchange Data Explorer

<https://data.stackexchange.com/stackoverflow/query/new>

- Corre sobre un servidor SQL Server



question
title

SQL Server String Concatenation with Null

Ask Question

question
score

64

I am creating a computed column across fields of which some are potentially null.

question body

The problem is that if any of those fields is null, the entire computed column will be null. I understand from the Microsoft documentation that this is expected and can be turned off via the setting SET CONCAT_NULL_YIELDS_NULL. However, there I don't want to change this default behavior because I don't know its implications on other parts of SQL Server.

Is there a way for me to just check if a column is null and only append its contents within the computed column formula if its not null?

[sql-server](#) [null](#) [string-concatenation](#) [calculated-columns](#)

question tags

user display name

share improve this question

add a comment

asked May 26 '10 at 21:05



Alex

30.9k • 65 • 223 • 317

active

oldest

votes

asked 7 years, 10 months ago

viewed 102,669 times

question view count

active 9 months ago

BLOG

Quantum Computing Site Launches with the Help of Strangeworks

questions

posts

answers

9 Answers

answer
score

110

You can use `ISNULL(...)`

question body

```
SET @Concatenated = ISNULL(@Column1, '') + ISNULL(@Column2, '')
```

If the value of the column/expresson is indeed NULL, then the second value specified (here: empty string) will be used instead.

share improve this answer

user

display name

answered May 26 '10 at 21:07



marc_s

532k • 116 • 1034 • 1194

comment

comment
score

13

"Coalesce" is the ANSI-standard function name, but ISNULL is easier to spell. – Philip Kelley May 26 '10 at 21:08

user display name

1

And ISNULL seems to be a tad faster on SQL Server, too - so if you want to use it in a function that concatenates strings into a computed column, you might forgo the ANSI standard and opt for speed (see Adam Machanic: [sqlblog.com/blogs/adam_machanic/archive/2006/07/12/...](http://sqlblog.com/blogs/adam_machanic/archive/2006/07/12/)) – marc_s May 26 '10 at 21:15

Looking for a job?

Senior 3D Engineer

Envelope New York, NY

\$100K - \$150K REMOTE

[sql](#) [javascript](#)

Java Developer

Wallethub Washington, DC

REMOTE

[java](#) [spring](#)

Java Developer - Best practices oriented

Almundo Retiro, Argentina

RELOCATION

[mongodb](#) [java](#)

DevOps Engineer - Remote

Peak Games No office location

REMOTE

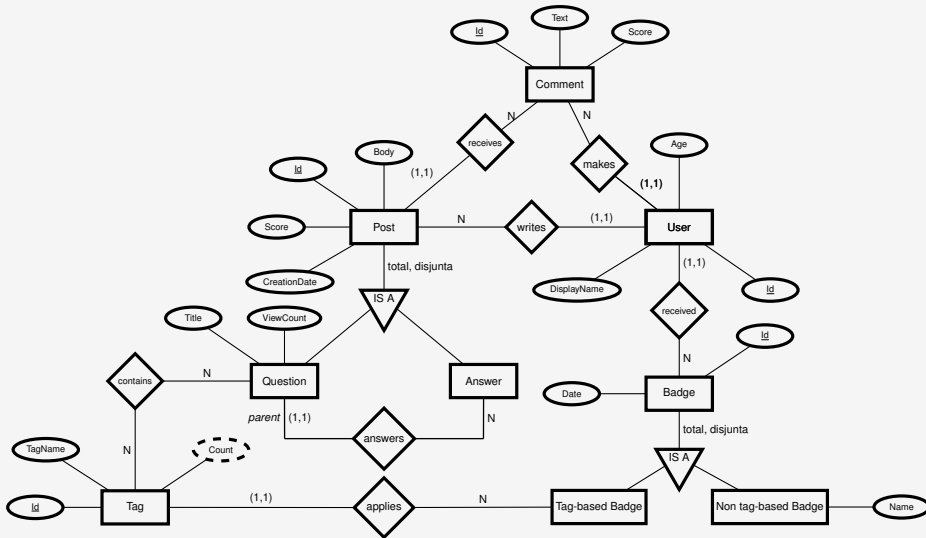
[linux](#) [amazon-web-services](#)

Linked

How can I avoid my selected row being a

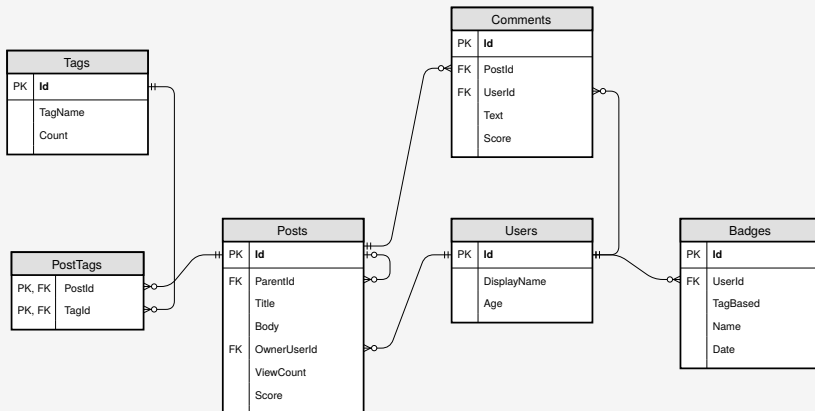
Stack Exchange Data Explorer

Modelo conceptual simplificado



Stack Exchange Data Explorer

Diagrama de tablas simplificado



Esta es una visualización comunmente utilizada del modelo lógico relacional. No estudiaremos estos diagramas en el curso, pero aquí utilizamos uno para ilustrar la estructura de la base de datos de Stack Exchange. Encontrarán un diagrama más completo en <https://meta.stackexchange.com/questions/250396/database-diagram-of-stack-exchange-model>.

También, en el panel de la derecha del Data Explorer se muestra la descripción completa de cada tabla.

1 Introducción

2 Definición de Datos en SQL

3 Manipulación de Datos en SQL

- `SELECT...FROM...WHERE`
- `JOIN`
- Operaciones de conjuntos
- Ordenamiento y paginación
- `GROUP BY...HAVING`
- Consultas anidadas
- ABM
- `DROP's`

4 Funciones y estructuras auxiliares

5 Bibliografía

CREATE SCHEMA

Creación de una base de datos

- El comando **CREATE SCHEMA** nos permite crear un nuevo esquema de base de datos dentro de nuestro SGBD.
- Su sintaxis es:

```
CREATE SCHEMA nombre_esquema [ AUTHORIZATION AuthId ];
```

- Ejemplo:

```
CREATE SCHEMA empresa [ AUTHORIZATION mbeiro ];
```

- La opción **AUTHORIZATION** identifica quién será el **dueño** del esquema.
- En un entorno SQL pueden haber varios esquemas de bases de datos. Los esquemas se agrupan en colecciones denominadas **catálogos**.
- Todo catálogo contiene un esquema llamado **INFORMATION_SCHEMA**, que describe a todos los demás esquemas contenidos en él.

Tipos de variables en SQL

■ Tipos numéricos estándar:

[ELM16 6.1.3]

- **INTEGER**: Tipo entero. Abreviado **INT**.
- **SMALLINT**: Tipo entero pequeño.
- **FLOAT(n)**: Tipo numérico aproximado. n indica la precisión en bits.
- **DOUBLE PRECISION**: Tipo numérico aproximado de alta precisión.

→ En Postgres, double precision f-p, IEEE 754 ($n=53$, $e=11$)

- **NUMERIC(i, j)**: Tipo numérico exacto. Permite especificar la precisión (i) y la escala (j) en dígitos.

precisión
dígitos totales

escala
dígitos decimales

- **Strings**: Se delimitan con comillas simples (').

- **CHARACTER(n)**: De longitud fija. Abreviado **CHAR(n)**. → default, $n=1$
- **CHARACTER VARYING(n)**: De longitud variable. Abrev. **VARCHAR(n)**.

■ Fecha y hora:

- **DATE**: Precisión de días. Se ingresa como string con formato YYYY-MM-DD. → (ISO 8601)
- **TIME(i)**: Precisión de hasta microsegundos. Se ingresa como string con formato HH:MM:SS.[0-9]ⁱ (ISO 8601). Tantos dígitos decimales como i .
- **TIMESTAMP(i)**: Combina un **DATE** y un **TIME(i)**.

Tipos de variables en SQL

- Booleanos (*opcional*):

- **BOOLEAN**: **TRUE**, **FALSE** o **UNKNOWN**. Se emplea *lógica de tres valores*.

- Otros tipos:

- **CLOB**: (**Character Large Object**) Para documentos de texto de gran extensión.
- **BLOB**: (**Binary Large Object**) Para archivos binarios de gran extensión.

- Tipos definidos por el usuario:

```
CREATE DOMAIN NOMBRE_DOMINIO AS TIPO_BASIC0;
```

- Ejemplo:

```
CREATE DOMAIN CODIGO_PAIS AS CHAR(2);
```

- Facilita la realización de cambios futuros en el diseño.

CREATE TABLE

Creación de una tabla

- El comando **CREATE TABLE** nos permite definir la estructura de una tabla:

```
CREATE TABLE Persona (  
  dni_persona INT PRIMARY KEY,  
  nombre_persona VARCHAR(255),  
  fecha_nacimiento DATE);
```

```
CREATE TABLE HijoDe (  
  dni_hijo INT,  
  dni_padre INT,  
  PRIMARY KEY (dni_hijo, dni_padre),  
  FOREIGN KEY (dni_hijo) REFERENCES Persona(dni_persona),  
  FOREIGN KEY (dni_padre) REFERENCES Persona(dni_persona));
```

CREATE TABLE

Creación de una tabla

Creación de una tabla: estructura general

```

CREATE TABLE  $T_1$  (
 $A_1$  type1 [NOT NULL] [CHECK condition1] [PRIMARY KEY],
 $A_2$  type2 [NOT NULL] [CHECK condition2] [PRIMARY KEY],
...,
 $A_n$  typen [NOT NULL] [CHECK conditionn] [PRIMARY KEY],
[PRIMARY KEY ( $A_{p_1}, A_{p_2}, \dots, A_{p_k}$ )]
{UNIQUE ( $A_{u_1}, A_{u_2}, \dots, A_{u'_k}$ )} ...
{FOREIGN KEY ( $A_{h_1}, A_{h_2}, \dots, A_{h'_k}$ ) REFERENCES  $T_2(B_{f_1}, B_{f_2}, \dots, B_{f'_k})$ 
[ON DELETE SET NULL | RESTRICT | CASCADE | SET DEFAULT]
[ON UPDATE SET NULL | RESTRICT | CASCADE | SET DEFAULT] }...);

```

- Las columnas también pueden ser configuradas con valores por defecto (**DEFAULT**) o autoincrementales (**AUTO_INCREMENT**).

Restricciones de dominio

[ELM16 6.2.1]

- Además de definir el tipo de una columna, es posible:
 - Restringir la posibilidad de que tome un valor nulo (**NOT NULL**).

```
fecha_nac DATE NOT NULL,
```

- Restringir aún más el conjunto de valores posibles, a través de un “chequeo en forma dinámica”:

```
CUIT_tipo INT CHECK (CUIT_tipo=20) OR (CUIT_tipo=23) OR...
```

Restricciones de unicidad

[ELM16 6.2.2]

- La clave primaria se indica con **PRIMARY KEY**. Si está compuesta de una única columna, puede indicarse a continuación del tipo.
- Con la palabra clave **UNIQUE** se indica que una columna o conjunto de columnas no puede estar repetido en dos filas distintas.
 - Es una manera de identificar *claves candidatas*.
- **Atención!** SQL no obliga a definir una clave primaria, pero siempre deberíamos hacerlo.

Restricciones de integridad

Integridad de entidad

- La clave primaria de una tabla nunca debería ser **NULL**, aunque algunos SGBD's lo permiten.

Integridad referencial

- Las claves foráneas se especifican con **FOREIGN KEY...REFERENCES**.

SQL vs. modelo relacional

Diferencias

[ELM16 6.3.4]

- En el modelo relacional una relación es un conjunto cuyos elementos son las tuplas.
- Por lo tanto, una tupla no puede estar repetida en una relación.
- En SQL se permite que una fila esté repetida muchas veces en una tabla.
- Este concepto se conoce como **multiset** o **bag of tuples**.

1 Introducción

2 Definición de Datos en SQL

3 Manipulación de Datos en SQL

- **SELECT...FROM...WHERE**
- **JOIN**
- Operaciones de conjuntos
- Ordenamiento y paginación
- **GROUP BY...HAVING**
- Consultas anidadas
- **ABM**
- **DROP's**

4 Funciones y estructuras auxiliares

5 Bibliografía

SELECT...FROM...WHERE

Esquema básico de consulta

- El esquema básico de una consulta en SQL es:

```
SELECT  $A_1, A_2, \dots, A_n$   
FROM  $T_1, T_2, \dots, T_m$   
[ WHERE condition ];
```

- En donde A_1, A_2, \dots, A_n es una lista de nombres de columnas, T_1, T_2, \dots, T_m es una lista de nombres de tablas, y *condition* es una condición.
- Es el análogo a la siguiente expresión del álgebra relacional:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_{condition}(T_1 \times T_2 \times \dots \times T_m))$$

- Con la diferencia de que la proyección en SQL no elimina filas repetidas.

SELECT...FROM...WHERE**WHERE: Condiciones****[ELM16 6.3.1]**

- Las condiciones atómicas admitidas dentro de la cláusula **WHERE** son:
 - $A_i \odot A_j$
 - $A_i \odot c$, con $c \in \text{dom}(A_i)$
 - A_i [NOT] LIKE p , en donde A_i es un *string* y p es un *patrón*
 - (A_i, A_{i+1}, \dots) [NOT] IN m , en donde m es un *set* o un *multiset*
 - A_i [NOT] BETWEEN a AND b , con $a, b \in \text{dom}(A_i)$
 - A_i IS [NOT] NULL
 - EXISTS t , en donde t es una tabla
 - $A_i \odot$ [ANY|ALL] t , en donde t es una tabla
- En donde \odot debe ser un operador de comparación:
 - $=, <>$
 - $>, >=, <, <=$ (para columnas cuyos dominios están ordenados)
- Varias condiciones atómicas pueden unirse a través de operadores lógicos para formar una condición más compleja. Los operadores lógicos permitidos son:
 - **AND, OR, NOT**

SELECT...FROM...WHERE**FROM: Alias y ambigüedad****[ELM16 6.3.2]**

- En la cláusula **FROM** es posible indicar un **alias** para las tablas:

```
...FROM Persona p...  
...FROM Persona AS p...
```

- Cuando se selecciona una columna, si la misma es ambigua **se deberá** indicar el nombre de la tabla ó su alias.

```
SELECT business.id  
FROM business...  
  
SELECT b.id  
FROM business b...
```

- Si una tabla se utiliza dos veces en la cláusula **FROM**, será indispensable indicar alias para distinguir sus columnas. También es posible renombrar las columnas.

```
..FROM Persona AS p1(dni1, nombre1), Personas AS p2(dni2, nombre2)..
```

SELECT...FROM...WHERE**SELECT: Redenominación y operaciones**

- También es posible cambiar el nombre de las columnas en el resultado:

```
SELECT p1.nombre AS NPadre, p2.nombre AS NHijo...
```

- Y realizar operaciones entre las columnas en el resultado:

```
SELECT Producto.precio * 0.90 AS precioDescontado...
```

- Las operaciones permitidas son:
 - +, -, *, / (columnas numéricas)
 - || (concatenación de *strings*)
 - +, - (sumar a una fecha, hora o timestamp un intervalo)
 - LN, EXP, POWER, LOG, SQRT, FLOOR, CEIL, ABS, .. (no son Core-SQL)

SELECT...FROM...WHERE**SELECT:** Funciones de agregación

- Por último, podemos aplicar una **función de agregación** a cada una de las columnas del resultado. Las más habituales son:
 - **SUM(A)**
 - Suma los valores de la columna *A* de todas las filas
 - **COUNT([DISTINCT] A | *)**
 - **COUNT(A)** cuenta la cantidad de filas con valor no nulo de *A*.
 - **COUNT(DISTINCT A)** cuenta la cantidad de valores distintos de *A*, sin contar el valor nulo.
 - **COUNT(*)** cuenta la cantidad de filas en el resultado.
 - **AVG(A)**
 - Calcula el promedio de los valores de *A*, descartando los valores nulos.
 - **MAX(A)**
 - Sólo para dominios ordenados.
 - **MIN(A)**
 - Sólo para dominios ordenados.
- En este caso, el resultado colapsa a una única fila.

SELECT...FROM...WHERE

Ejemplos

Cuenta la cantidad de usuarios existentes en la base de datos.

```
SELECT COUNT(*)  
FROM Users;
```

Cuenta la cantidad de *posts* que son preguntas.

```
SELECT COUNT(*)  
FROM Posts p  
WHERE p.ParentId IS NULL;
```

SELECT...FROM...WHERE

Omisión de la selección y de la proyección. Duplicados

- La condición de selección puede omitirse si no se escribe la cláusula **WHERE**.
- La proyección sobre un subconjunto de columnas puede omitirse escribiendo **SELECT ***.
- La palabra clave **DISTINCT** después de la cláusula **SELECT** elimina los duplicados en el resultado.

SELECT...FROM...WHERE**WHERE:** Pattern matching

[ELM16 6.3.5]

- La cláusula **WHERE** también permite condiciones de reconocimiento de patrones para columnas que son *strings*.

```
...WHERE attrib LIKE pattern;
```

- Se acepta como patrón una secuencia de caracteres delimitada por comillas ('), combinada con los siguientes caracteres especiales en su interior:
 - `_` (representa un caracter arbitrario)
 - `%` (representa cero o más caracteres arbitrarios)
 - Si se necesita un `_` ó un `%` literal en el patrón, se debe *escapear*

SELECT...FROM...WHERE

Ejemplos

Liste los nombres de *badges* que otorga StackOverflow y que no están basados en *tags*.

```
SELECT DISTINCT Name
FROM Badges
WHERE TagBased=0;
```




Liste los *tags* que utilizó el usuario 'Jon Skeet'.

```
SELECT DISTINCT t.TagName
FROM Users u, Posts p, PostTags pt, Tags t
WHERE pt.PostId=p.Id AND pt.TagId=t.Id
AND p.OwnerUserId=u.Id AND u.DisplayName='Jon Skeet'
```

- Junta theta (\bowtie)

- Junta natural (*)

- Al igual que en el álgebra relacional, los nombres de las columnas de junta deben coincidir en ambas tablas.

- Junta externa (, , )

35 / 71

JOIN

Ejemplo

Utilizando la junta theta, encuentre los posts que utilizan conjuntamente los *tags* 'relational' y 'entity-relationship', indicando el id del post, el título y la cantidad de vistas.

```
SELECT DISTINCT p.Id, p.Title, p.ViewCount
FROM (((Tags t1 INNER JOIN PostTags pt1 ON t1.Id=TagId)
      INNER JOIN Posts p ON pt1.PostId=p.Id)
      INNER JOIN PostTags pt2 ON p.Id=pt2.PostId)
      INNER JOIN Tags t2 ON pt2.TagId=t2.Id
WHERE t1.TagName='relational'
AND t2.TagName='entity-relationship';
```

Operaciones de conjuntos

■ SQL incorpora las 3 operaciones de conjuntos:

■ Unión (\cup)

```
...R UNION [ALL] S...
```

■ Intersección (\cap)

```
...R INTERSECT [ALL] S...
```

■ Diferencia ($-$)

```
...R EXCEPT [ALL] S...
```

■ Tener en cuenta que:

- Las tablas R y S pueden provenir a su vez de una subconsulta.
- R y S deben ser **union compatibles**.
- Si no se agrega la palabra clave **ALL**, el resultado será un *set* en vez de un *multiset*, y entonces no habrá filas repetidas.

Ordenamiento y paginación

- Extendemos el esquema básico con la cláusula **ORDER BY**:

```
SELECT A1, A2, ..., An
FROM T1, T2, ..., Tm
[ WHERE condition ]
[ ORDER BY Ak1 [ ASC | DESC ], Ak2 [ ASC | DESC ], ...];
```

- Las **columnas de ordenamiento** A_{k_1}, A_{k_2}, \dots deben pertenecer a dominios ordenados, y deben estar incluidas dentro de las columnas de la proyección en la cláusula **SELECT**.
- Si no se especifica, cada columna es ordenada en forma ascendente.
- La **paginación** es la posibilidad de escoger un rango $[t_{inicio}, t_{fin}]$ del listado de filas del resultado.
 - La forma estándar es: **[OFFSET..ROWS] FETCH FIRST..ROWS ONLY**.
 - Algunos SGBD's implementan otras cláusulas como **LIMIT**.
 - <https://www.jooq.org/doc/3.10/manual/sql-building/sql-statements/select-statement/limit-clause/>

Ordenamiento y paginación

Ejemplo

Ejemplo

De entre las preguntas hechas desde el 01/01/2017 en adelante, muestre las 10 que recibieron más visitas, indicando su título, la fecha en que se hicieron y la cantidad de visitas que recibieron.

```
SELECT Title, CreationDate, ViewCount
FROM Posts
WHERE CreationDate >= '2017-01-01'
AND ParentId IS NULL
ORDER BY ViewCount DESC
OFFSET 0 ROWS FETCH FIRST 10 ROWS ONLY;
```

Agregación

- Analicemos la siguiente tabla Campeones(nombre_tenista, nombre_torneo, premio) indicando los torneos ganados por distintos tenistas en 2016:

CAMPEONES

nombre_tenista	nombre_torneo	premio
Novak Djokovic	Abierto de Australia	8.000.000
Rafael Nadal	Abierto de Barcelona	1.500.000
Novak Djokovic	Abierto de Madrid	2.500.000
Novak Djokovic	Roland Garros	5.000.000
Andy Murray	Abierto de China	2.500.000
Andy Murray	Master de Shangai	4.000.000
Juan Martín del Potro	Abierto de Estocolmo	300.000
Andy Murray	Master BNP Paribas	2.000.000
Andy Murray	ATP Tour Finals de Londres	4.000.000

- Nos gustaría resumir en una tabla la cantidad de títulos ganados por cada tenista y su premio total anual.

Agregación

- Para ello necesitamos *agrupar* los datos de cada tenista...

CAMPEONES

nombre_tenista	nombre_torneo	premio
Novak Djokovic	Abierto de Australia	8.000.000
Novak Djokovic	Abierto de Madrid	2.500.000
Novak Djokovic	Roland Garros	5.000.000
Rafael Nadal	Abierto de Barcelona	1.500.000
Juan Martín del Potro	Abierto de Estocolmo	300.000
Andy Murray	Abierto de China	2.500.000
Andy Murray	Master de Shangai	4.000.000
Andy Murray	Master BNP Paribas	2.000.000
Andy Murray	ATP Tour Finals de Londres	4.000.000

- ...y luego dejar una única tupla por cada uno, que muestre su nombre pero resuma los torneos mostrando su cantidad, y los premios mostrando el total.

Agregación

- El resultado sería:

nombre_tenista	nombre_torneo	premio
Novak Djokovic	3	15.500.000
Rafael Nadal	1	1.500.000
Juan Martín del Potro	1	300.000
Andy Murray	4	12.500.000

- La **agregación** colapsa las tuplas que coinciden en una serie de atributos (en este caso, *nombre_tenista*), en una única tupla que las representa a todas.
- Diremos que “*agrupamos la relación Campeones por tenista, y resumimos los torneos indicando su cantidad, y los premios indicando su suma*”.
- En SQL, esto puede hacerse con la cláusula **GROUP BY**:

```
SELECT nombre_tenista, COUNT(nombre_torneo), SUM(premio)
FROM Campeones
GROUP BY nombre_tenista;
```

GROUP BY...HAVING

Esquema de consulta con agregación

[ELM16 7.1.8]

- La cláusula **GROUP BY** implementa la operación de agregación. El esquema de una consulta con agregación es:

```
SELECT  $A_{k_1}, A_{k_2}, \dots, f_1(B_1), f_2(B_2), \dots, f_p(B_p)$   
FROM  $T_1, T_2, \dots, T_m$   
[ WHERE  $condition_1$  ]  
GROUP BY  $A_1, A_2, \dots, A_n$   
[ HAVING  $condition_2$  ]  
[ ORDER BY  $A_{k_1}$  [ ASC | DESC ],  $A_{k_2}$  [ ASC | DESC ], ...];
```

- A_1, A_2, \dots, A_n son las *columnas de agrupamiento*, y algunas de ellas participan de la selección final. B_1, B_2, \dots, B_p no son columnas de agrupamiento, pero participan de la selección final a través de las *funciones de agregación* anteriormente mencionadas.

Agregación en álgebra relacional extendida [ELM16 8.4.2]

- Podemos definir un operador de agregación como extensión del álgebra relacional básica:
- Definimos la **agregación**

$$A_1, A_2, \dots, A_n \mathfrak{S}_{f_1(B_1), f_2(B_2), \dots, f_p(B_p)}(R)$$

como un operador que primero proyecta R en el conjunto (A_1, \dots, A_n) , y luego por cada tupla t de dicha proyección devuelve una tupla t' cuyos primeros n valores corresponden a t y los p restantes a funciones de agregación $f_i(B_i)$, aplicadas al *listado* de valores que toma B_i en aquellas tuplas $r \in R$ que fueron proyectadas a t .

- Por último indicamos cuáles serán las posibles funciones de agregación, como: **SUM**, **AVG**, **COUNT**, **MAX**, **MIN**...
- Observación: Si en una agregación no se especifican atributos de agregación ($n = 0$), el resultado tendrá una única tupla.

GROUP BY...HAVING

Cláusula HAVING

- La cláusula **HAVING** es opcional, y nos permite seleccionar sólo algunos de los *grupos* del resultado.
- *condition₂* es por lo tanto una condición que involucra *funciones de agregación sobre las columnas que no son de agrupamiento en el **GROUP BY***.

GROUP BY...HAVING

Ejemplo

Ejemplo

Muestre los nombres de los 10 usuarios cuyas respuestas a preguntas taggeadas con 'c#' acumulan mayor puntaje.

```
SELECT u.DisplayName, SUM(resp.Score)
FROM Users u, Posts resp, Posts preg, PostTags pt, Tags t
WHERE u.Id = resp.OwnerUserId
AND pt.PostId = preg.Id
AND pt.TagId=t.Id
AND t.TagName='c#'
AND resp.ParentId = preg.Id
GROUP BY u.Id, u.DisplayName
ORDER BY SUM(resp.Score) DESC
OFFSET 0 ROWS FETCH FIRST 10 ROWS ONLY;
```

GROUP BY...HAVING

Ejemplo

Ejemplo

Liste los tags cuyo primer uso ocurrió después del 01/01/2018.

```
SELECT t.TagName
FROM Tags t, PostTags pt, Posts p
WHERE t.Id = pt.TagId
AND pt.PostId = p.Id
GROUP BY t.TagName
HAVING MIN(p.CreationDate)>='2018-01-01';
```

Consultas anidadas

Subqueries en la cláusula **WHERE**

[ELM16 7.1.2 7.1.3]

- Es posible incluir una subconsulta dentro de la cláusula **WHERE**.
 - Recordemos: el resultado de una consulta es siempre una tabla.
 - **Tip:** ¡Y cuando el resultado sólo contiene una fila con una única columna, puede ser pensado y utilizado como un valor constante!

Subquery son no correlacionadas (más eficientes)

```
SELECT ... FROM ...
WHERE A IN (SELECT X FROM ...); — Debe devolver una única columna
```

```
SELECT ... FROM ...
WHERE A = (SELECT X FROM ...); — Debe devolver sólo 1 fila!
```

```
SELECT ... FROM ... — (feature opcional)
WHERE (A, B) IN (SELECT X, Y FROM ...); — Debe devolver 2 columnas
```

```
SELECT ... FROM ...
WHERE A < [ SOME | ALL ] (SELECT X FROM ...);
```

¿tenemos A todos o a algunos?

```
SELECT ... FROM ... → Correlacionado — Devuelve FALSE/TRUE según
WHERE [ NOT ] EXISTS (SELECT ... FROM ...); — la tabla esté vacía o no
```

→ Correlacionado — Devuelve FALSE/TRUE según la tabla esté vacía o no

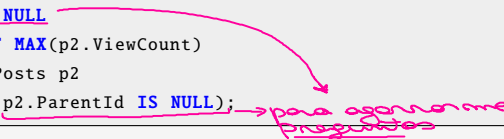
consultas correlacionadas → dependiendo del gestor puede no ser eficiente

Consultas anidadas

Subqueries en la cláusula **WHERE**

Encuentre el Id, el título y la cantidad de vistas de la/s pregunta/s que haya/n tenido la mayor cantidad de vistas.

```
SELECT preg.Id, preg.Title, preg.ViewCount
FROM Posts preg
WHERE preg.ParentId IS NULL
AND ViewCount = (SELECT MAX(p2.ViewCount)
                  FROM Posts p2
                  WHERE p2.ParentId IS NULL);
```



- Cuando la consulta interna hace referencia a columnas de las tablas de la consulta externa, se dice que las mismas están correlacionadas.
- ¡El costo de una subconsulta correlacionada es mucho más alto!

Consultas anidadas

Cuantificadores: **ALL** y **SOME**

→ ej "python"
Encuentre para cada tag la/s pregunta/s que haya/n tenido la mayor cantidad de vistas, indicando el nombre del tag, el título de la/s pregunta/s y la cantidad de vistas. Sólo muestre aquellos tags para los que la cantidad de vistas es de al menos 2 millones.

```
SELECT t.TagName, preg.Title, preg.ViewCount
FROM Tags t, PostTags pt, Posts preg
WHERE t.Id = pt.TagId
AND pt.PostId = preg.Id
AND preg.ParentId IS NULL → filtro las preguntas
AND preg.ViewCount > 20000000
AND ViewCount >= ALL (SELECT ViewCount
                        FROM PostTags pt2, Posts p
                        WHERE pt2.TagId = t.Id
                        AND pt2.PostId = p.Id);
```

no correlacionada

Bonus Track

Ejercicio

Encuentre para cada *tag* el/los usuarios cuyas respuestas a preguntas con ese *tag* acumulan mayor puntaje, indicando el nombre del *tag*, el nombre del usuario, y su puntaje. Sólo muestre aquellos *tags* para los que el usuario con puntaje máximo tiene un puntaje de al menos 30000.

Bonus Track: World Cup 2010 Dataset

hacer para el parcial

Ejercicio 1

Encuentre el nombre del jugador que llevó la camiseta número 14 en la Selección Argentina.

Ejercicio 2

Liste los nombres y la cantidad de goles convertidos por los jugadores que convirtieron 4 o más goles en el Mundial.

(Nota: No cuente los goles en series de penales ni los goles en contra (sólo cuentan los score_types 1,2, 3 ó 4).

Ejercicio 3

Encuentre el nombre del jugador de mayor edad que participó del Mundial.

(Puede ejecutar estas consultas directamente desde la solapa SQL del RelaX. Tenga en cuenta que no podrá utilizar subconsultas y que los alias en la cláusula FROM requieren de AS (ej., FROM Player AS p)).

Inserciones

[ELM16 6.4.1]

- Las inserciones se realizan con el comando **INSERT INTO**. Dada una tabla T con columnas A_1, A_2, \dots, A_n , se admiten las siguientes posibilidades:

- Insertar un listado de n -filas:

```
INSERT INTO  $T$   
VALUES ( $a_{11}, a_{12}, \dots, a_{1n}$ ), ( $a_{21}, a_{22}, \dots, a_{2n}$ ),  
..., ( $a_{p1}, a_{p2}, \dots, a_{pn}$ );
```

- El orden de carga de los valores debe ser el mismo que el de las columnas en la tabla.

- Insertar un listado de k -filas, con $k < n$

```
INSERT INTO  $T(A_{i_1}, A_{i_2}, \dots, A_{i_k})$   
VALUES ( $a_{1i_1}, a_{1i_2}, \dots, a_{1i_k}$ ), ( $a_{2i_1}, a_{2i_2}, \dots, a_{2i_k}$ ),  
..., ( $a_{pi_1}, a_{pi_2}, \dots, a_{pi_k}$ );
```

Inserciones

- Insertar el resultado de una consulta:

```
INSERT INTO T( $A_{i_1}, A_{i_2}, \dots, A_{i_k}$ )
SELECT ...
FROM ...;
```

- La consulta debe devolver una tabla con k columnas, y las columnas deben ser unión compatibles.

- En cualquiera de los casos, si...

- Se asigna a una columna un valor fuera de su dominio
- Se omitió una columna que no podía ser **NULL**
- Se puso en **NULL** una columna que no podía ser **NULL**
- La clave primaria asignada ya existe en la tabla
- Una clave foránea hace referencia a una clave no existente

- ... **X** no se inserta. → Se inserta todo o si no puede vuelve al estado original

Eliminaciones → *los foráneos dependen de su configuración*

[ELM16 6.4.2]

- La sintaxis para las eliminaciones es:

```
DELETE FROM T  
WHERE condition;
```

- *condition* puede ser una combinación de condiciones atómicas.
- Si no se especifican condiciones, se eliminan todas las filas.
- Para cada fila *t* que se intente eliminar y que es referenciada por una clave foránea desde otra tabla:
 - Si dicha clave foránea se **configuró** en **ON DELETE CASCADE**
 - ✓ Se eliminan todas las filas que referencian a ésta, y luego se elimina *t*.
 - Si en cambio se **configuró** en **ON DELETE SET NULL**
 - ✓ Se ponen en **NULL** todas las claves foráneas de las filas que referencian a ésta, y luego se elimina *t*.
 - Si se **configuró** en **ON DELETE RESTRICT**
 - ✗ No se elimina *t*.

Modificaciones

[ELM16 6.4.3]

- Las modificaciones se realizan con el comando **UPDATE**.

```
UPDATE T
SET  $A_1 = c_1, A_2 = c_2, \dots, A_k = c_k$ 
WHERE condition;
```

- condition* puede ser una combinación de condiciones atómicas.
- Un único **UPDATE** puede modificar muchas filas.
- Para cada fila *t* que cumpla la condición, si...
 - Se modifica una columna asignándole un valor fuera de su dominio
 - Se pasó a **NULL** una columna que no podía tomar ese valor
 - Se asignó a la clave primaria un valor que ya existe en la tabla
 - Se modificó una clave foránea para hacer referencia a una clave no existente
- ... **X** entonces *t* no se actualiza. *id est*

Modificaciones

- Atención! Si en la fila t se modifica una clave primaria que es referenciada por una clave foránea desde otra tabla:
 - Si dicha ^{idém} clave foránea se configuró en **ON UPDATE CASCADE**
 - ✓ Se modifican todas las filas que referencian a ésta en forma acorde, y luego se modifica t .
 - Si en cambio se configuró en **ON UPDATE SET NULL**
 - ✓ Se ponen en **NULL** todas las claves foráneas de las filas que referencian a ésta, y luego se modifica t .
 - Si se configuró en **ON UPDATE RESTRICT**
 - ✗ No se modifica t .

DROP SCHEMA Y DROP TABLE

- Una tabla se elimina con **DROP TABLE**.

```
DROP TABLE T [ RESTRICT | CASCADE ];
```

→ borra la restricción de la persona (le borro a quien tiene mi ref, esa ref)

- Un esquema se elimina con **DROP SCHEMA**.

```
DROP SCHEMA S [ RESTRICT | CASCADE ];
```

1 Introducción

2 Definición de Datos en SQL

3 Manipulación de Datos en SQL

- **SELECT...FROM...WHERE**
- **JOIN**
- Operaciones de conjuntos
- Ordenamiento y paginación
- **GROUP BY...HAVING**
- Consultas anidadas
- ABM
- **DROP's**

4 Funciones y estructuras auxiliares

5 Bibliografía

Manejo de strings

[SQLDR Strings; SQLCOMP Functions and operators]

- El estándar SQL cuenta con varias funciones para manipular strings. Entre ellas:
 - **SUBSTRING**(string FROM start FOR length): Selecciona un substring desde la posición *start* y de largo *length*.
 - **UPPER**(string)/**LOWER**(string): Convierte el string a mayúsculas/minúsculas.
 - **CHAR_LENGTH**(string): Devuelve la longitud del string.

Conversión de tipos

- **CAST**(attr AS TYPE) permite realizar conversiones entre tipos. *costeo*
- **EXTRACT**(campo FROM attr) permite extraer información de una columna de fecha/hora (*feature* opcional).

```
SELECT (EXTRACT(DAY FROM fecha)) AS dia, COUNT(nro_factura)
FROM Facturas f;
GROUP BY dia;
```

- Valores posibles para el campo: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, ...
- **COALESCE**(expr1, expr2, ..., exprN) devuelve para cada tupla la primera expresión no nula de izquierda a derecha.

```
—Reemplaza los valores nulos en la columna domicilio
—por el string '<desconocido>'.
SELECT apellido, nombre, COALESCE(domicilio, '<desconocido>')
FROM Clientes;
```

Conversión de tipos

Ejercicio

A partir de las columnas `dia_venc`, `mes_venc` y `año_venc` obtenga la fecha de vencimiento de cada factura, con tipo de dato **DATE**. Complete para ello el siguiente código.

```
SELECT nro_factura ,  
       CAST(  
           ....  
       AS DATE) AS fecha_venc  
FROM Facturas f;
```

Nota: Suponga que las mismas representan una fecha válida, y que todos los años tienen 4 dígitos.

Conversión de tipos

Respuesta

```

SELECT nro_factura,
       CAST(
         CAST(año_venc AS CHAR) || '-' ||
         SUBSTRING(( '0' || CAST(mes_venc AS VARCHAR)
                     FROM (2-CAST(mes_venc<10 AS INTEGER)) FOR 2
                   ) || '-' ||
         SUBSTRING(( '0' || CAST(dia_venc AS VARCHAR)
                     FROM (2-CAST(dia_venc<10 AS INTEGER)) FOR 2
                   ) AS DATE
       ) AS fecha_venc
FROM Facturas f;

```

Estructura CASE WHEN..THEN..ELSE..END

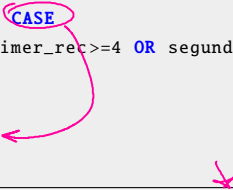
- La estructura **CASE** nos permite agregar cierta lógica de la programación estructurada a una sentencia SQL.
- Permite seleccionar entre distintos valores posibles de salida en función de distintas condiciones.

padrón	apellido	nombre	primera_op	primer_rec	segundo_rec
102111	Bertrán	Verónica	9		
103553	Salamanca	Ernesto	2	7	
104617	Guzmán	Claudia			8
105928	Sanz	Rubén		2	

```

SELECT padrón, apellido, nombre, CASE
    WHEN primera_op >= 4 OR primer_rec >= 4 OR segundo_rec >= 4
    THEN 'APROBO_PARCIAL'
    ELSE 'DESAPROBO_PARCIAL'
END AS situacion_parcial
FROM Notas_Parcial;

```



padrón	apellido	nombre	situacion_parcial
102111	Bertrán	Verónica	APROBO_PARCIAL
103553	Salamanca	Ernesto	APROBO_PARCIAL
104617	Guzmán	Claudia	APROBO_PARCIAL
105928	Sanz	Rubén	DESAPROBO_PARCIAL

Estructura CASE WHEN..THEN..ELSE..END

Solución alternativa usando CASE

```
SELECT nro_factura,
       CAST(
         CAST(año_venc AS CHAR) || '—' ||
         CASE WHEN mes_venc>9 THEN '' ELSE '0' END ||
         CAST(mes_venc AS VARCHAR) || '—' ||
         CASE WHEN dia_venc>9 THEN '' ELSE '0' END ||
         CAST(dia_venc AS VARCHAR)
       AS DATE
       ) AS fecha_venc
FROM Facturas f;
```

Nota: La comparación por mayor es una extensión, y puede no ser soportada por algunos SGBD's. Se puede reemplazar por `LENGTH(CAST(mes_venc AS VARCHAR))=2`.

Cláusula **WITH** (opcional según estándar)

- La cláusula **WITH** permite construir una tabla auxiliar temporal previa a una consulta. No es Core-SQL.

```
WITH T[(A1, A2, ..., An)]
AS (<subquery>)
<query>;
```

WITH 1: Pseudocódigo

```
T = subquery(R1, R2, ...);
return query(T, ...);
```

Listar los nombres de badges que tiene el usuario Jon Skeet.

```
WITH Jon
AS (SELECT Id
      FROM Users
      WHERE DisplayName = 'Jon_Skeet')
SELECT DISTINCT Name
FROM Badges b, Jon j
WHERE b.UserId = j.Id;
```

→ Funciona como un inner join

→ Se puede meter la tabla así pero no se recomienda, es menos legible usar el WITH

Cláusula `WITH` (opcional según estándar)

→ Práctica para parciales
Sem 5 min 5:22 (teórico)

Ejercicio 1

En StackOverflow existen usuarios que contestan un gran número de preguntas y usuarios que contestan muy pocas. Nos interesa construir un histograma de la cantidad de consultas respondidas por usuario, que indique *para cada número entero i entre 0 y MAX la cantidad de usuarios que respondieron una cantidad de consultas igual a i* , en donde MAX es la cantidad de consultas respondidas por el/los usuario/s que respondió/eron más consultas.

Ayuda: Construya primero una tabla auxiliar que indique para cada usuario la cantidad de preguntas distintas que el mismo respondió.

Ejercicio 2

Modifique la consulta anterior para que calcule el histograma con *bins* de la forma $[0,10)$, $[10,100)$, $[100,1000)$, ...

- 1 Introducción
- 2 Definición de Datos en SQL
- 3 Manipulación de Datos en SQL
 - `SELECT...FROM...WHERE`
 - `JOIN`
 - Operaciones de conjuntos
 - Ordenamiento y paginación
 - `GROUP BY...HAVING`
 - Consultas anidadas
 - ABM
 - `DROP's`
- 4 Funciones y estructuras auxiliares
- 5 Bibliografía

Bibliografía

[ELM16] Fundamentals of Database Systems, 7th Edition.

R. Elmasri, S. Navathe, 2016.

Capítulo 6, Capítulo 7

[SILB19] Database System Concepts, 7th Edition.

A. Silberschatz, H. Korth, S. Sudarshan, 2019.

Capítulo 3, Capítulo 4

[CONN15] Database Systems, a Practical Approach to Design, Implementation and Management, 6th Edition.

T. Connolly, C. Begg, 2015.

Capítulo 6, Capítulo 7, Capítulo 8

[GM09] Database Systems, The Complete Book, 2nd Edition.

H. García-Molina, J. Ullman, J. Widom, 2009.

Capítulo 6, Capítulo 7

Bibliografía

Bibliografía relativa al estándar

[SQL] ISO/IEC 9075:2011 Standard

Estándar ISO, 2011

Versión *draft* en <http://www.wisecorp.com/sql20nn.zip> a la que se puede acceder también desde <http://modern-sql.com/standard>.

[SQLGRAM] SQL::2011 Foundation Grammar

Gramática de la Parte 2 del estándar

<https://jakewheat.github.io/sql-overview/sql-2011-foundation-grammar.html>.

Bibliografía

Sitios comparativos DBMS's vs. estándar

[SQLDR] SQL Dialects Reference

Wikibooks

`https:`

`//en.wikibooks.org/wiki/SQL_Dialects_Reference.`

[SQLCOMP] Comparison of different SQL implementations

T. Arvin

`http://troels.arvin.dk/db/rdbms/.`