

SEMINARIO DE ACTUALIZACIÓN



Bases de Datos II





PROGRAMA DE ESTUDIO 2024

Asignatura	
Seminario de actualización	
Docente	Hs. Semanales
Jorge Insfran	2
Extensión de la Asignatura	Carrera
Anual	Tecnicatura Superior en Análisis de Sistemas

1. Objetivos Generales

- Profundizar en los principios básicos de la tecnología de bases de datos.
- Familiarizarse con el uso de sistemas de gestión de bases de datos relacionales
- Conocer los conceptos relacionados las bases no relacionales, o NoSQL
- Conocer los principios básicos de la Seguridad Informática

2. Objetivos específicos

- Ampliar el concepto de bases de datos y sus características principales.
- Conocer el concepto de sistema de gestión de bases de datos, sus componentes y funciones.
- Conocer el concepto de independencia de datos en bases de datos y los mecanismos asociados.
- Conocer los mecanismos y estrategias para el control de la integridad (calidad) y la seguridad (privacidad) en bases de datos.
- Ser capaz de realizar consultas complejas en SQL
- Conocer los tipos de modelos de datos que manejan las bases de datos NoSQL y situaciones en las que es conveniente su uso
- Manejar conceptos básicos de la Seguridad Informática

3. Programa didáctico

Unidad 1: Filtros TOP y OFFSET-FETCH. Predicados y Operadores. Trabajando con Fecha y Hora.

Unidad 2: Programando. Lotes. Cursores. Tablas temporales. SQL Dinámico. Rutinas. Manejo de errores

Unidad 3: Consultas sobre múltiples tablas. Subconsultas. Expresiones de Tabla. Tablas derivadas, Expresiones Comunes de Tabla (CTE), Vistas, Funciones de Tabla. JOINS. CROSS, INNER y OUTER. Operaciones de Conjuntos

Unidad 4: Consultas complejas. Funciones de ventana. Pivoteo. Despivoteo. Trabajo con conjuntos de agrupamiento

Unidad 5: Seguridad de la Información. Cifrado, Vectores de ataque. Criptología

Unidad 6: Bases de datos NoSQL

Unidad 7: Tablas cronológicas

4. Programa Analítico

Semana	Temas a Desarrollar
1	Repaso SQL
2	Filtros TOP y OFFSET-FETCH. Predicados y Operadores
3	Predicados y Operadores
4	Trabajando con Fecha y Hora
5	Programando. Lotes.
6	Tablas temporales. Cursores
7	SQL Dinámico Rutinas.
8	Manejo de errores



9	Trabajo Práctico
10	Examen Parcial
11	Subconsultas. Expresiones de Tabla. Tablas derivadas, CTE, Vistas, Funciones de Tabla.
12	JOINS. Operaciones de Conjunto
13	Consultas complejas
14	Consultas complejas
15	Consultas complejas
16	Pivoteo y Despivoteo / Trabajo con conjuntos de agrupamiento
17	Recuperatorio
Receso	
18	Seguridad de la información
19	Seguridad de la información
20	Seguridad de la información
21	Seguridad de la información
22	Seguridad de la información
23	Seguridad de la información
24	Seguridad de la información
25	Bases NoSQL
26	Bases NoSQL
27	Bases NoSQL
28	Bases NoSQL
29	Tablas cronológicas
30	Tablas cronológicas
31	Trabajo Práctico
32	Examen Parcial
33	Seguridad de la Información
34	Recuperatorio

5. Bibliografía de Lectura Obligatoria

Título	Autor	Editorial
T-SQL Fundamentals, Fourth Edition	Itzik Ben-Gan	Microsoft Press
T-SQL Querying	Itzik Ben-Gan, Dejan Sarka, Adam Machanic, Kevin Farlee	Microsoft Press
Beginning Microsoft SQL Server® 2008 Programming	Robert Vieira	Wiley Publishing Inc.
SQL & NoSQL Databases	Andreas Meier, Michael Kaufmann	Springer

6. Bibliografía General

Título	Autor	Editorial
Introducing Microsoft SQL Server 2019	Kellyn Gorman, Allan Hirt, Dave Noderer, James Rowland-Jones, Arun Sirpal, Dustin Ryan, y Buck Woody	Packt Publishing Ltd.
Database System Concepts, Seventh Edition	Silberschatz, Abraham, Korth, Henry F., Sudarshan, S.	McGraw-Hill Education
SQL and Relational Theory - How to Write Accurate SQL Code	C. J. Date	O'Reilly Media, Inc.
Microsoft SQL Server 2012 Transact-SQL DML Reference	SQL Server Books Online	Microsoft
Transact-SQL Data Definition Language (DDL) Reference	SQL Server Books Online	Microsoft



Programming Microsoft® SQL Server® 2012	Leonard Lobel, Andrew Brust	Microsoft Press
Sistemas de Bases de Datos	R. Elmasri y S.B.Navathe	Addison-Wesley Iberoamericana, 1997
Database Systems: The Complete Book	Hector Garcia-Molina Jeffrey D. Ullman Jennifer Widom	Prentice Hall
SQL Server 2012 Tutorials: Writing Transact-SQL Statements	SQL Server Books Online	Microsoft
Securing SQL Server - Protecting Your Database from Attackers	Denny Cherry	Syngress
NoSQL Distilled	Pramod J. Sadalage, Martin Fowler	Addison-Wesley
Introducción a las bases de datos NoSQL usando MongoDB	Antonio Sarasa	Editorial UOC
SQL Server 2022 Revealed	Bob Ward	Apress
7. Estrategias de Enseñanza		
La asignatura se desarrollará a través de clases teóricas, así como prácticas en las aulas informáticas. Se valorará muy positivamente la participación de los alumnos. Además, cuando el profesor lo determine y fomentando así el pleno uso de las nuevas tecnologías, se hará hincapié en el trabajo colaborativo de los alumnos/as con el fin de fomentar el espíritu crítico, reflexión y la creatividad.		
Se utilizará la plataforma como elemento indispensable de comunicación y seguimiento de las clases por parte de los alumnos.		
Habrá exposición oral por parte del docente. Se utilizará el laboratorio con el software SQL Server 2016, Se realizarán trabajos prácticos.		
8. Criterios y Pautas de Evaluación		
El alumno para poder aprobar la cursada deberá:		
<ul style="list-style-type: none">- Aprobar un examen parcial por cuatrimestre, cada uno tendrá un recuperatorio.- Trabajos prácticos de entrega obligatoria para evaluar el avance en los contenidos.- Trabajo práctico final, obligatorio.		





REPASO SQL

SQL

El modelo relacional fue propuesto por E. F. Codd en un artículo ya famoso "A relational model of data for large shared data banks"¹. Desde ese momento se instituyeron varios proyectos de investigación con el propósito de construir sistemas de gestión de bases de datos relacionales.

Entre estos proyectos podemos mencionar:

- Sistema R del IBM San José Research Laboratory.
- Ingres de la Universidad de California en Berkeley.
- Query-by-example del IBM T. J. Watson Research Center.
- PRTV (Peterlee Relational Test Vehicle) del IBM Scientific Center en Peterlee, Inglaterra.

El lenguaje SQL se introdujo como lenguaje de consulta del Sistema R, y fue desarrollado por Raymond F. Boyce y Donald D. Chamberlin. Posteriormente, varios sistemas comerciales lo adoptaron como lenguaje para sus bases de datos.

El nombre **SQL** está formado por las iniciales de **Structured Query Language** (Lenguaje de consultas estructuradas).

Si bien el SQL está definido como un estándar, las bases de datos relacionales tienen sus extensiones propietarias y variaciones.

A continuación, se enumera las versiones del estándar a lo largo de los años, y sus principales características:

Año	Nombre	Alias	Comentarios
1986	SQL-86	SQL-87	Primera publicación hecha por ANSI. Confirmada por ISO en 1987.
1989	SQL-89	FIPS127-1	Revisión menor, el agregado más importante fueron las restricciones de integridad. Adoptado como FIPS 127-1.
1992	SQL-92	SQL2, FIPS 127-2	Revisión mayor (ISO 9075)
1999	SQL:1999	SQL3	Se agregaron expresiones regulares, consultas recursivas (para relaciones jerárquicas), triggers y algunas características orientadas a objetos.
2003	SQL:2003	SQL 2003	Introduce algunas características de XML, cambios en las funciones, estandarización del objeto sequence y de las columnas auto numéricas.
2006	SQL:2006	SQL 2006	ISO/IEC 9075-14:2006 Define las maneras en las cuales el SQL se puede utilizar juntamente con XML. Define maneras de importar y guardar datos XML en una base de datos SQL, manipulándolos dentro de la base de datos y publicando el XML y los datos SQL convencionales en forma XML. Además, proporciona facilidades que permiten a las aplicaciones integrar dentro de su código SQL el uso de XQuery, lenguaje de consulta XML publicado por el W3C (World Wide Web Consortium) para acceso concurrente a datos ordinarios SQL y documentos XML.

¹ "A relational model of data for large shared data banks" - Comm. ACM 13,6 - June 1970



2008	SQL:2008	SQL 2008	Permite el uso de la cláusula ORDER BY fuera de las definiciones de los cursor. Incluye los disparadores del tipo INSTEAD OF. Añade la sentencia TRUNCATE, OFFSET-FETCH
2011	SQL:2011		Datos cronológicos (PERIOD FOR). Mejoras en las funciones de ventana y de la cláusula FETCH.
2016	SQL:2016		Permite búsqueda de patrones, funciones de tabla polimórficas y compatibilidad con los ficheros JSON.
2019	SQL:2019		Arrays Multidimensionales (SQL/MDA)
2023	SQL:2023		Se agrega el tipo JSON (SQL/Foundation); Se agrega parte 16, Property Graph Queries (SQL/PGQ)

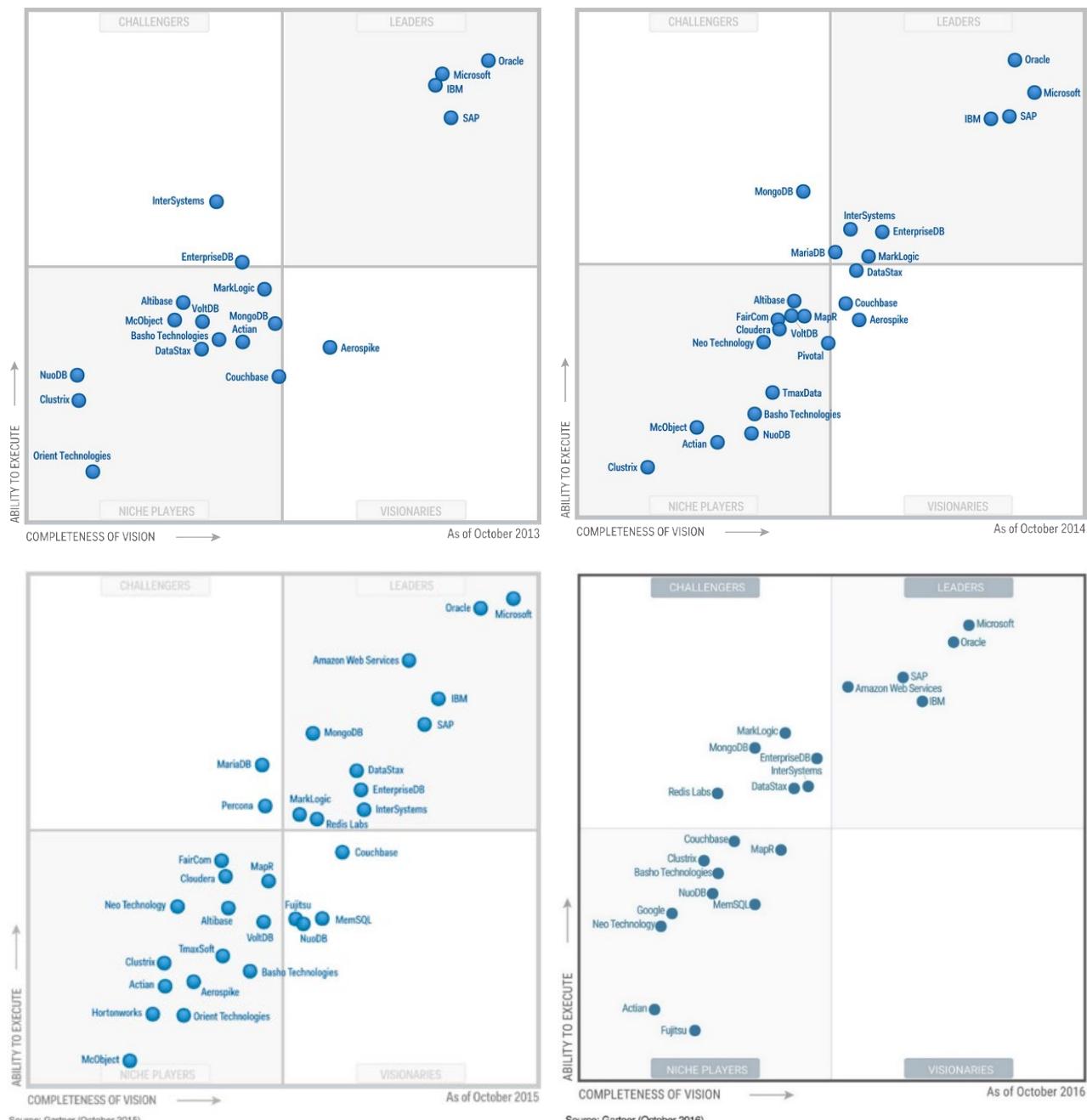
Desde SQL:1999, el estándar se divide en varias partes numeradas del 1 al 16 (a partir de SQL:2023). Algunos de ellos nunca fueron publicados (5-8, 12), otros nunca llegaron a ser populares. Dejando a un lado la metaparte (parte 1), sólo la parte 2 (el lenguaje SQL), la parte 11 (esquema de información) y la parte 14 (XML) se implementaron ampliamente. Las partes 15 (MDA) y 16 acaban de publicarse, por lo que es demasiado pronto para ver si se generalizarán o no.²

- Parte 1 – Marco: Una descripción general aproximada y algunas definiciones de términos de uso común.
- Parte 2 – Fundación: Define la mayor parte del lenguaje SQL (otras partes lo amplían, por ejemplo, para la funcionalidad XML).
- Parte 3: Interfaz a nivel de llamada (SQL/CLI) Describe las API de C y COBOL para acceder a bases de datos SQL.
- Parte 4: Módulos almacenados persistentes (SQL/PSM) Define un lenguaje utilizado para la programación del lado del servidor (“procedimientos almacenados”).
- Parte 5: Enlaces del idioma anfitrión (SQL/Enlaces) Fusionado en la parte 2 con SQL:2003 (aviso de retiro).
- Parte 6: Soporte de transacciones globales (SQL/Transacción) Nunca liberado (?)
- Parte 7: Temporal (SQL/Temporal) Nunca liberado. Finalmente se agregó soporte temporal a SQL:2011 parte 2.
- Parte 8: Soporte extendido de objetos Nunca liberado. Contenido absorbido en otras partes (aviso).
- Parte 9 - Gestión de datos externos (SQL/MED) Define mecanismos para acceder a datos almacenados fuera de la base de datos.
- Parte 10: Enlaces de lenguaje de objetos (SQL/OLB) Define cómo incrustar sentencias SQL en programas Java. Esto no es JDBC, que trata las sentencias SQL como cadenas (“SQL dinámico”).
- Parte 11 - Esquemas de información y definición (SQL/Schemata) Define INFORMACIÓN_SCHEMA y DEFINITION_SCHEMA, que se trataron en la parte 2 anterior a SQL:2003.
- Parte 12: Replicación (SQL/Replicación) Nunca liberado.
- Parte 13: Rutinas y tipos que utilizan el lenguaje de programación Java (SQL/JRT) Define cómo ejecutar Java dentro de la base de datos.
- Parte 14: Especificaciones relacionadas con XML (SQL/XML) Define el tipo de datos XML y los métodos para trabajar en documentos XML. Apareció con SQL:2003.
- Parte 15: Arreglos multidimensionales (SQL/MDA) Apareció por primera vez en 2019. Ver ISO.
- Parte 16: Consulta de gráfico de propiedades (SQL/PGQ) Apareció por primera vez en 2023. Incorpora partes del nuevo estándar GQL en SQL.

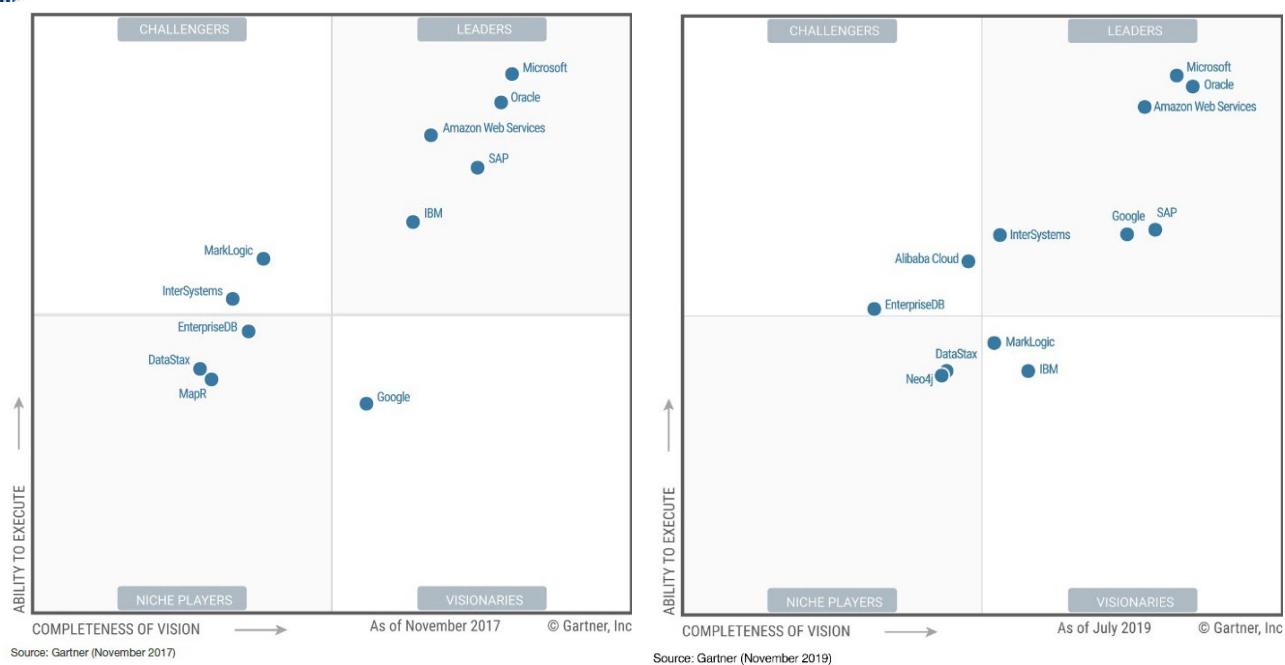
² <https://modern-sql.com/standard/parts>



Según Gartner³ (octubre 2013/2019), las compañías líderes son Microsoft, Oracle, IBM, SAP y Amazon.



³ <https://www.gartner.com/reviews/market/cloud-database-management-systems>



En la última edición de su reporte, en 2023, Gartner analizó sistemas de gestión de bases de datos en la **nube**, ya que la tendencia del mercado es clara.



Source: Gartner

Por este marcado y sostenido liderazgo y por la posibilidad de contar con versiones gratuitas para hacer las prácticas, es que en la materia utilizaremos MS SQL Server como Sistema de Gestión de Bases de Datos.



HISTORIA DE MICROSOFT SQL SERVER

Es el sistema de gestión de base de datos de Microsoft.

Microsoft comenzó el desarrollo de SQL Server en conjunto con Sybase Corporation para ser utilizado en la plataforma OS/2 de IBM. Con la salida de su sistema operativo de red (Windows NT Advanced Server) Microsoft decidió continuar con el desarrollo para su propia plataforma por su cuenta. El producto resultante fue el Microsoft SQL Server 4.21

Versión	Año	Nombre Release	Nombre Clave
1.0 (OS/2)	1989	SQL Server 1.0 (16 bit)	Ashton-Tate /Microsoft SQL Server
1.1 (OS/2)	1991	SQL Server 1.1 (16 bit)	-
4.21 (WinNT)	1993	SQL Server 4.21	SQLNT
6.0	1995	SQL Server 6.0	SQL95
6.5	1996	SQL Server 6.5	Hydra
7.0	1998	SQL Server 7.0	Sphinx
-	1999	SQL Server 7.0 OLAP Tools	Palato mania
8.0	2000	SQL Server 2000	Shiloh
8.0	2003	SQL Server 2000 64-bit Edition	Liberty
9.0	2005	SQL Server 2005	Yukon
10.0	2008	SQL Server 2008	Katmai
10.25	2010	Azure SQL DB	Cloud Database or CloudDB
10.50	2010	SQL Server 2008 R2	Kilimanjaro (aka KJ)
11.0	2012	SQL Server 2012	Denali
12.0	2014	SQL Server 2014	SQL14
13.0	2016	SQL Server 2016	SQL16
14.0	2017	SQL Server 2017	Helsinki
15.0	2019	SQL Server 2019	Seattle
16.0	2022	SQL Server 2022	Dallas

Las versiones 7.0 y 2000 fueron modificaciones y extensiones al código desarrollado en conjunto con Sybase. Para la versión 2005 el código fue reescrito completamente.

Desde el lanzamiento de la versión Microsoft SQL Server 2000 se han realizado avances en la performance, las IDE⁴ de cliente, y se han agregado paquetes y sistemas complementarios incluyendo:

- ETL⁵
- Reporting Server
- OLAP⁶ y Data Mining Server (Analysis Services)
- Tecnologías de mensajería y comunicación (Service Broker, Notification Services)

⁴ Integrated Development Environment: Ambiente de desarrollo interactivo o Entorno de desarrollo integrado

⁵ Extract Transform Load

⁶ On Line Analytical Processing



SQL SERVER 2005

Lanzado en octubre de 2005. Incluyó soporte nativo para XML⁷, con XQuery como lenguaje de consultas para datos en XML, que pueden ser embebidos dentro del T-SQL.

SQL SERVER 2008

Lanzado en agosto de 2008. Incluyó la tecnología SQL Server Always On, con el objetivo de hacer que la administración de los datos sea auto optimizante, auto organizada y auto mantenida para lograr un tiempo de caídas de casi cero.

Incluyó además soporte para datos estructurados y semiestructurados, incluyendo formatos para medios digitales como imágenes, audio, video y otros datos multimedia (que anteriormente se almacenaban como objetos genéricos (BLOBS)).

Se incorpora además la funcionalidad Full-text Search que simplifica la administración y mejora la performance.

Con SQL Reporting Services se incorpora la facilidad de obtener reportes y gráficos de manera nativa.

La versión de SQL Server Management Studio incluye IntelliSense para consultas SQL, que permite completar código de manera inteligente y chequeo de sintaxis.

SQL SERVER 2008 R2

Lanzado en abril de 2010. Incluye Master Data Services, que permite una administración centralizada de entidades de datos maestras y su jerarquía.

Se incluyen nuevos servicios orientados al análisis de datos como Power Pivot, StreamInsight, Report Builder 3.0, un agregado de Reporting Services para Sharepoint, funcionalidades de capa de datos para Visual Studio que permite empaquetar bases de datos como parte de una aplicación.

SQL SERVER 2012

Lanzado en marzo de 2012. Incluye mejoras orientadas a la alta disponibilidad como AlwaysOn SQL Server Failover Cluster Instances y Availability Groups, Contained Databases que simplifica la tarea de mover una base de datos entre instancias.

Mejoras en cuanto a la programación, almacenamiento de ubicaciones, metadata, objetos de secuencia y mejoras de performance y seguridad

SQL SERVER 2014

Lanzado en abril de 2014. Incluye mejoras de performance que permite utilizar la velocidad de discos SSD como intermedio entre la memoria RAM y los discos mecánicos, mejoras en los servicios de alta disponibilidad

SQL SERVER 2016

Fue liberada el 1 de junio de 2016. La versión RTM es 13.0.1601.5. SQL Server 2016 es soportado solo en procesadores x64. No existe versión de 32 bits.

SQL SERVER 2017

Microsoft liberó SQL Server 2017 el 2 de octubre de 2017 y es la primera versión con soporte para Linux.
<https://www.microsoft.com/es-es/sql-server/sql-server-2017>

⁷ Extensible Markup Language



SQL SERVER 2019

Esta versión se liberó al público en noviembre de 2019. Incluye además de mejoras sobre el motor, la capacidad de manejar clústeres de Big Data

SQL SERVER 2022

Incluye innovaciones en integración de datos y bases de datos habilitadas para Azure. Características como la alta disponibilidad bidireccional por primera vez y la recuperación ante desastres entre Azure SQL Managed Instance y SQL Server, la integración de Azure Synapse Link con SQL para ETL, informes y análisis casi en tiempo real sobre sus datos operativos, y nueva inteligencia de consultas integrada de próxima generación con optimización de planes sensible a parámetros.

EDICIONES PRINCIPALES

Enterprise: Edición completa, con performance para aplicaciones de misión crítica y con requerimientos de inteligencia de negocios. Provee los más altos niveles de servicio y performance.

Standard: Administración de datos central y capacidad para inteligencia de negocios para aplicaciones que no sean de misión crítica con el mínimo de recursos de IT

Developer: Versión con todas las características orientada a desarrolladores, que permite desarrollar, probar y demostrar aplicaciones basadas en software SQL Server.

Web: Versión especial para proveedores de hosting.

Express: Nivel básico y gratuita. Ideal para el aprendizaje y construcción de aplicaciones de escritorio o pequeños servidores. Permite hasta 10 GB de almacenamiento.

FUNCIONES DEL SERVIDOR

Característica	Descripción
Motor de base de datos de SQL Server	Motor de base de datos de SQL Server incluye el motor de base de datos, el servicio principal para almacenar, procesar y proteger datos, replicación, búsqueda de texto completo, herramientas para administrar datos relacionales y XML, integración de análisis de bases de datos e integración de PolyBase para acceso a fuentes de datos heterogéneas y servicios de aprendizaje automático para ejecutar scripts de Python y R con datos relacionales.
Analysis Services	Analysis Services incluye las herramientas para crear y administrar aplicaciones de procesamiento analítico en línea (OLAP) y minería de datos.
Reporting Services	Reporting Services incluye componentes de servidor y cliente para crear, administrar e implementar informes tabulares, matriciales, gráficos y de formato libre. Reporting Services también es una plataforma extensible que puede utilizar para desarrollar aplicaciones de informes.
Servicios de integración	Los servicios de integración son un conjunto de herramientas gráficas y objetos programables para mover, copiar y transformar datos. También incluye el componente de Servicios de calidad de datos (DQS) para Servicios de integración.
Master Data Services	Master Data Services (MDS) es la solución de SQL Server para la gestión de datos maestros. MDS se puede configurar para administrar cualquier dominio (productos, clientes, cuentas) e incluye jerarquías, seguridad granular, transacciones, control de versiones de datos y reglas comerciales, así como un complemento para Excel que se puede usar para administrar datos.

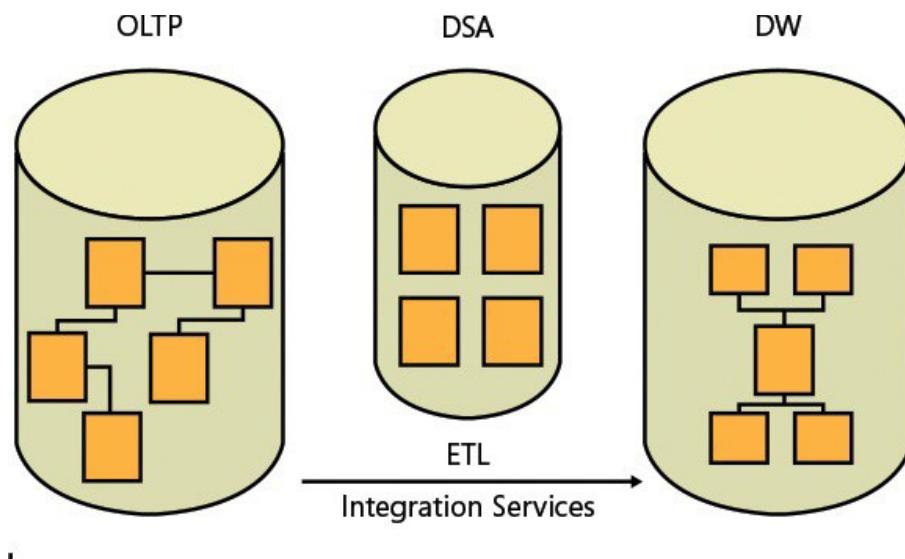


Servicios de aprendizaje automático (en la base de datos)	Los servicios de aprendizaje automático (en la base de datos) admiten soluciones de aprendizaje automático distribuidas y escalables que utilizan fuentes de datos empresariales. En SQL Server 2016, se admitía el lenguaje R. SQL Server 2022 (16.x) es compatible con R y Python.
Virtualización de datos con PolyBase	Consulta diferentes tipos de datos en diferentes tipos de fuentes de datos desde SQL Server.
Servicios conectados de Azure	SQL Server 2022 (16.x) amplía las funciones y los servicios conectados de Azure, incluidos Azure Synapse Link, las políticas de acceso de Microsoft Purview, la extensión de Azure para SQL Server, la facturación de pago por uso y la función de enlace para Instancia administrada de SQL..

TIPOS DE SISTEMAS DE BASES DE DATOS

Hay dos tipos de sistemas que utilizan SQL Server como base de datos y T-SQL para administrar y manipular los datos:

- OLTP (*Online transactional processing*)
- DWs (*Data Warehouses*)



SQL Server
T-SQL

- DSA: *Data-staging area*
- ETL: *Extract, Transform and Load*

En la materia vamos a trabajar sobre bases de datos **OLTP**.

ARQUITECTURA DE SQL SERVER

Para contar con SQL server hay distintas alternativas de plataforma, y dentro de cada una de las alternativas veremos cómo se organizan los distintos componentes.

ALTERNATIVAS ABC

Proviene de los nombres en inglés de cada alternativa. A de **Appliance** (aparato), B de **Box** (caja), C de **Cloud** (nube).



CAJA (Box)

Es la alternativa tradicional. Se aloja habitualmente en las instalaciones del cliente. El cliente es responsable de todo, obtener el hardware, instalar el software, aplicar parches, montar la estrategia para lograr alta disponibilidad, recupero ante desastres, seguridad y todo.

El cliente puede instalar múltiples instancias del producto en el mismo servidor y puede realizar consultas que interactúen con múltiples bases de datos.

El lenguaje de consulta utilizado es el T-SQL.

APARATO (Appliance)

Es una solución llave en mano, con hardware y software preconfigurado. El aparato se aloja habitualmente en las instalaciones del cliente. Estos productos son configurados en conjunto por Microsoft y fabricantes de hardware, como Dell o HP.

Hay aparatos especializados para Data Warehouse, y para manejo de Big Data.

NUBE (Cloud)

Se proveen recursos a demanda disponibles en la nube. Se puede utilizar una nube pública o una nube privada. La nube privada es similar a la caja, pero por el concepto de nube, sirve para contabilizar el uso de los recursos.

Para la nube pública, Microsoft provee dos alternativas:

- IaaS (**Infrastructure as a Service**)
- PaaS (**Platform as a Service**)

Con la alternativa **IaaS**, se suministra una máquina virtual que reside en la infraestructura cloud de Microsoft. Hay distintas variantes de máquinas virtuales preconfiguradas, que pueden traer instalada alguna versión de SQL Server. El hardware es mantenido por Microsoft, pero el cliente es responsable del mantenimiento y aplicación de parches del software.

Con **PaaS**, Microsoft suministra la base de datos en la nube como un servicio. Está alojada en datacenters de Microsoft. La instalación y mantenimiento del hardware y el software, la alta disponibilidad, el recupero ante desastres, son responsabilidad de Microsoft.

Microsoft cuenta con distintas ofertas de PaaS. Para OLTP ofrece el servicio Azure SQL Database, o SQL Database. También cuenta con una versión para Data Warehouse, llamada Microsoft Azure SQL Data Warehouse.

Microsoft ofrece además otros servicios de datos en la nube, como Data Lake para servicios relacionados con Big Data, Azure DocumentDB para servicios de documentos NoSQL., y otros.

AZURE SQL DATABASE

Azure SQL Database es un motor de base de datos **totalmente administrado** como **Plataforma como Servicio (PaaS)** proporcionado por **Microsoft Azure**. Se encarga de la mayoría de las funciones de administración de bases de datos, lo que permite a los desarrolladores centrarse en construir aplicaciones en lugar de preocuparse por el mantenimiento de la infraestructura. Aquí tienes algunos puntos clave sobre **Azure SQL Database**:

1. **Base de Datos en la Nube Administrada:** Azure SQL Database es un servicio de base de datos relacional en la nube **flexible, escalable y seguro**. Ofrece características como escalado automático, copias de seguridad y mejoras de seguridad, lo que lo convierte en una excelente opción para aplicaciones modernas.



2. **Niveles de Servicio:** Azure SQL Database ofrece diferentes niveles de servicio para satisfacer diversas necesidades:

- **General Purpose (Propósito General):** Adecuado para la mayoría de las cargas de trabajo.
- **Business Critical (Crítico para el Negocio):** Diseñado para aplicaciones de alto rendimiento.
- **Hyperscale (Hiperscala):** Ideal para aplicaciones a gran escala con escalado automático.

3. **Funcionalidades:**

- **Elastic Pools (Grupos Elásticos):** Permite administrar y asignar recursos eficientemente entre múltiples bases de datos.
- **Intelligent Query Processing (Procesamiento de Consultas Inteligente):** Mejora el rendimiento de las consultas.
- **Serverless Compute (Cómputo sin Servidor):** Escala automáticamente los recursos de cómputo según la demanda.
- **Seguridad y Gobernanza:** Incluye funciones de seguridad integradas y certificaciones de cumplimiento.

4. **Comparación con SQL Server:**

- Azure SQL Database es un servicio nativo de la nube, mientras que **SQL Server** puede ejecutarse en las instalaciones o en máquinas virtuales en Azure.
- **Azure SQL Managed Instance** es otra opción que proporciona una instancia de SQL Server totalmente administrada en Azure.
- **SQL Server en Máquinas Virtuales de Azure** te permite ejecutar SQL Server en una VM en Azure (en este caso sería IaaS)

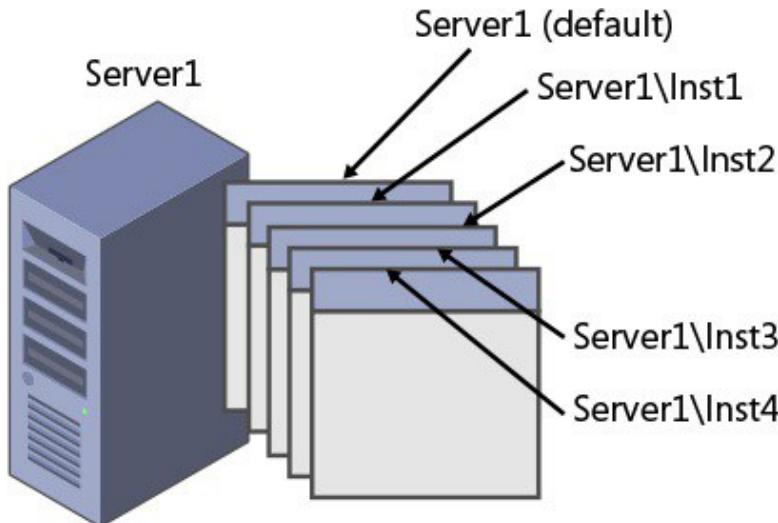
En resumen, Azure SQL Database simplifica la administración de bases de datos, garantiza la escalabilidad y ofrece sólidas características de seguridad. Ya sea que estés construyendo una pequeña aplicación o un sistema a gran escala, Azure SQL Database puede satisfacer tus requisitos.

En <https://learn.microsoft.com/en-us/azure/azure-sql/database/free-offer?view=azuresql> pueden conocer la alternativa para utilizar una base de datos de Azure SQL gratis (en el artículo pueden ver las limitaciones que aplican).



INSTANCIAS DE SQL SERVER

Dentro de un mismo servidor se pueden instalar múltiples instancias de SQL Server. Cada instancia es completamente independiente de las otras en cuanto a seguridad, la información que manejan y en definitiva en todos los aspectos.

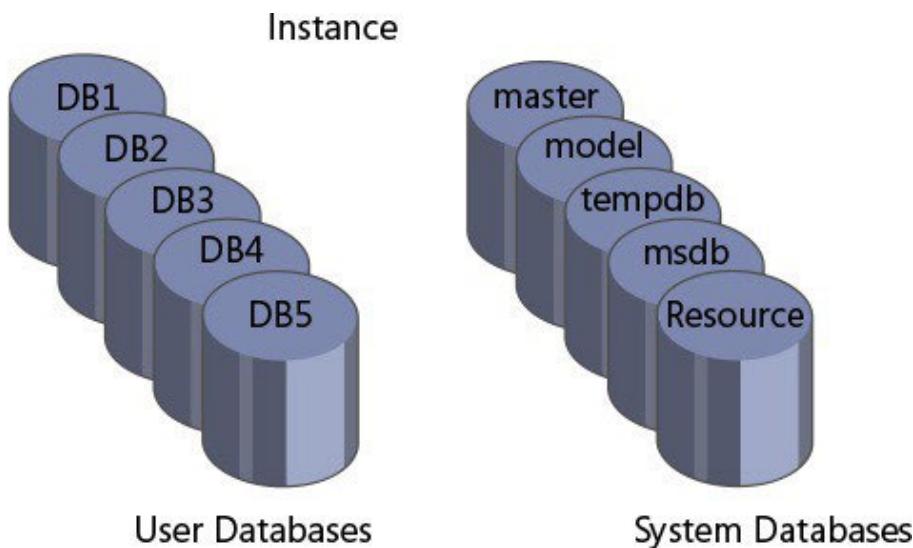


A nivel lógico dos instancias instaladas en el mismo equipo son tan independientes entre sí como si estuvieran instaladas en distintos equipos físicos.

En mismo equipo puede instalarse una instancia default, que es accedida a través del nombre del equipo, y múltiples instancias nominadas, que son accedidas a través del nombre del equipo y el nombre de la instancia separados por una barra invertida (\), por ejemplo, MIPC\SQLEXPRESS (el equipo llamado MIPC y la instancia SQLEXPRESS).

BASES DE DATOS

Una base de datos puede verse como un contenedor de objetos, como tablas, vistas, stored procedures y otros. Cada instancia de SQL Server puede contener múltiples bases de datos.



Al instalar una instancia, se crean varias bases de datos de sistema, que contienen información propia del sistema y se utilizan de manera interna para la administración de la base de datos.



Una vez que está instalada la instancia de la base de datos se pueden crear múltiples bases de datos de usuario.

Las bases de datos del sistema son:

- **master**: contiene metadata de la instancia, configuración del servidor, información sobre las bases de datos de la instancia e información de inicialización.
- **Resource**: es una base oculta y de sólo lectura que contiene la definición de los objetos de sistema.
- **model**: es usada como template para las nuevas bases de datos que se crean en la instancia.
- **tempdb**: es la base donde SQL Server almacena la información temporal, como tablas de trabajo, espacio de ordenamiento, versionado de filas, etc.
- **msdb**: es utilizada por el servicio SQL Server Agent. Este agente es encargado de la automatización de procesos

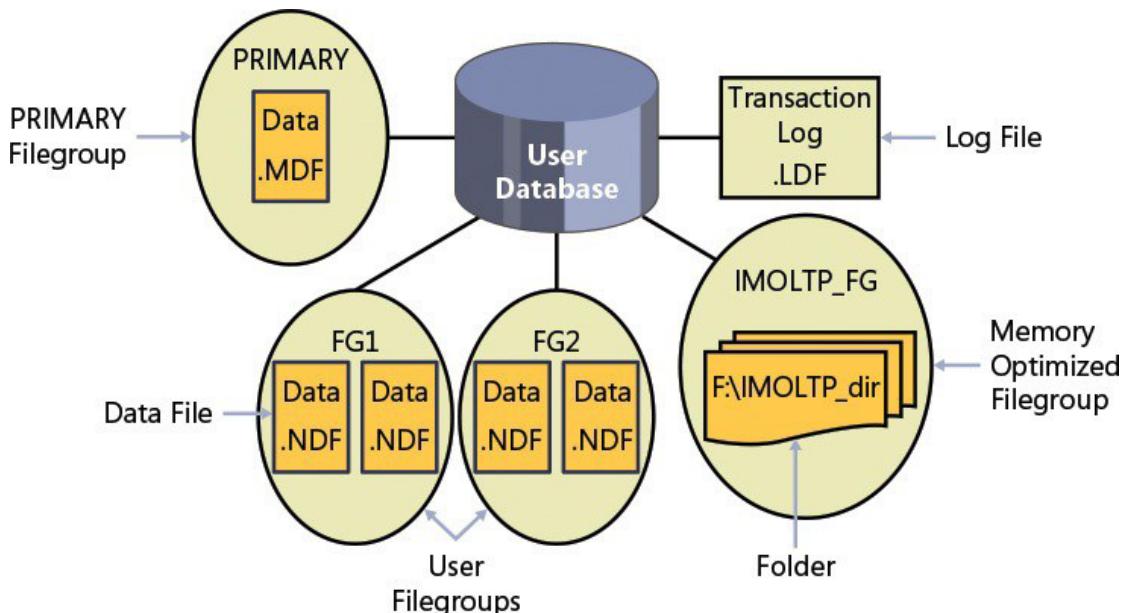
En una instancia pueden crearse tantas bases de datos como sean necesarias.

Para poder ejecutar comandos T-SQL contra la base de datos es necesario que una aplicación cliente se conecte a la instancia y especificar el contexto o base de datos necesaria.

En términos de seguridad, para poder conectarse a una instancia, administrador de la base de datos (DBA) debe crear un **login** para el usuario. Veremos este tema más en profundidad cuando veamos los aspectos de seguridad.

Esto es la parte lógica de la base de datos. Como usuarios, es probable que hasta este nivel nos interese conocer. Pero hay un aspecto físico en cuanto a la base de datos, que necesitamos conocer sobre todo si estamos en el rol de administrador de la base de datos.

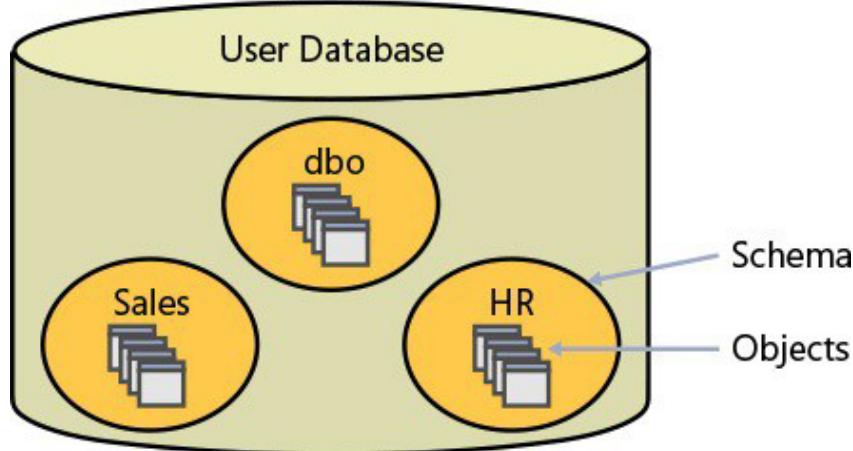
La base de datos está formada por *archivos de datos*, *archivos de log de transacciones*, y opcionalmente por archivos de control (*checkpoint*) que contienen los datos optimizados en memoria (In-Memory OLTP)





ESQUEMAS Y OBJETOS

Al decir que una base de datos contenía objetos, en realidad simplificamos un poco las cosas. En realidad, una base de datos contiene esquemas y son los esquemas los que contienen objetos.



Se pueden controlar los permisos de acceso a nivel esquema, de modo de poder separar los diferentes permisos de acceso a objetos de manera organizacional.

El esquema es también un espacio de nombres, ya que es usado como prefijo de los nombres de objeto. Por ejemplo, dentro del esquema del departamento de ventas (**Sales**) podemos tener una tabla para almacenar las órdenes (**Orders**) a la que accederemos como **Sales.Orders**.

Si se omite el nombre del esquema al hacer referencia a un objeto, SQL Server buscará primero en el esquema por default del usuario y si no encuentra el objeto lo buscará en el esquema dbo.

CATEGORÍAS DE LAS INSTRUCCIONES SQL

El lenguaje SQL puede dividirse en cuatro partes:

- **DML (Database Manipulation Language** o Lenguaje de manipulación de datos): SELECT, INSERT, UPDATE, DELETE, TRUNCATE
- **DDL (Database Definition Language** o Lenguaje de definición de datos): CREATE, ALTER, DROP
- **DCL (Database Control Language** o Lenguaje de control de datos): GRANT, DENY, REVOKE
- **TCL (Transaction Control Language** o Lenguaje de control de transacciones): COMMIT, ROLLBACK, SAVEPOINT

RESTRICCIONES (**CONSTRAINTS**)

Las restricciones ayudan a mantener la integridad de los datos. Son reglas que los datos deben cumplir. Se definen en el modelo de datos y el sistema de gestión de base de datos se encarga de que se cumplan.

El primer tipo de restricción, aunque no es una restricción en sí, es el uso de tipos de datos. Si un atributo está definido como fecha, sólo aceptará como valores fechas que sean válidas.

Junto con el tipo de datos se define para cada campo si el mismo puede tomar valores indefinidos (NULL).

Para cumplir la regla de integridad de entidad, se define para cada tabla una clave primaria, que es un identificador único de cada registro (tupla), y por lo tanto no admiten valores indefinidos (NULL).

Las claves candidatas son otro tipo de restricción que también colaboran con la integridad de datos.



Las claves foráneas permiten cumplir con la regla de integridad referencial. Una clave foránea es un atributo, o conjunto de atributos, que referencian a una clave candidata de otra tabla (o podría ser de la misma tabla). Por ejemplo, en una tabla de Empleados, podemos tener una clave foránea definida en el atributo iddepartamento, que referencia a la clave primaria iddepartamento de la tabla Departamentos. Esto quiere decir que los únicos valores que puede tomar Empleados.iddepartamento son los que aparecen en Departamentos.iddepartamento.

Para cumplir la integridad referencial existen distintas estrategias que pueden definirse, para ajustarse a la realidad del dominio de datos.

Por defecto, las claves foráneas **restringen** las operaciones. Al intentar eliminar un registro cuya clave primaria tiene registros asociados en otra tabla como clave foránea, el RDBMS evitará la eliminación, ya que de permitirla los registros asociados violarían la integridad referencial. En estos casos, se deberían primero eliminar o modificar los registros asociados.

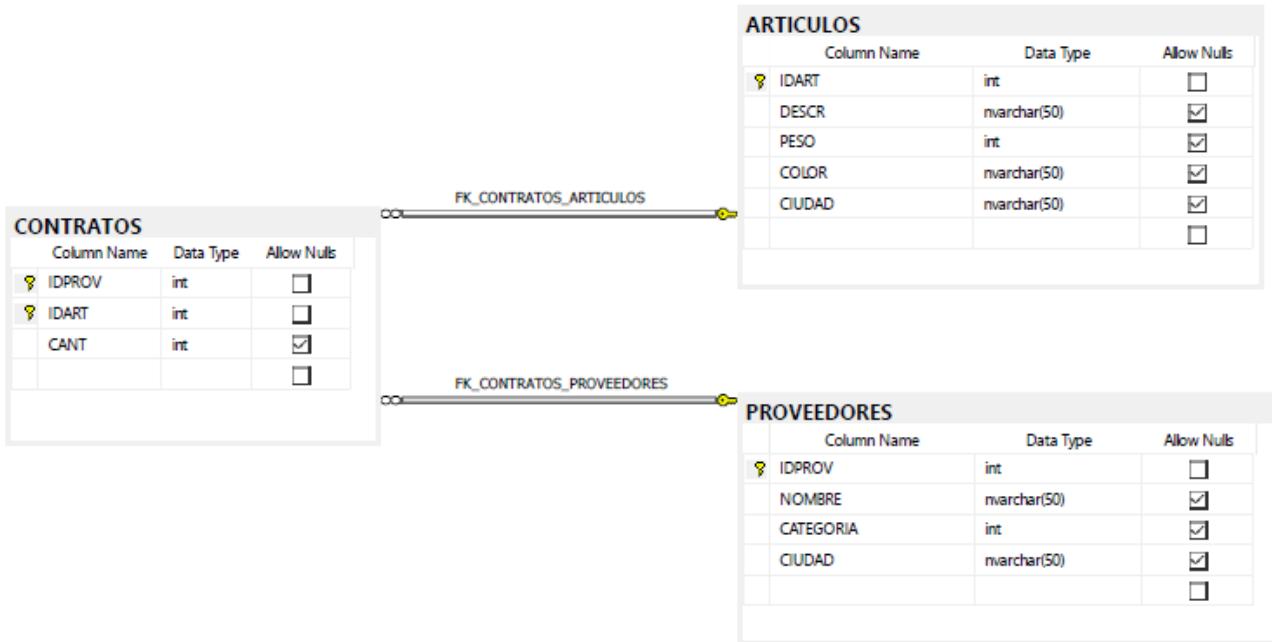
En SQL Server podemos implementar tres alternativas:

- **Cascada:** Al eliminar un registro, el RDBMS elimina también los registros relacionados
- **Default:** Al eliminar un registro, el RDBMS cambia el valor de las claves foráneas colocando el valor por default
- **Null:** Al eliminar un registro, el RDBMS cambia el valor de las claves foráneas colocando como valor NULL.



CREANDO TABLAS

Vamos a crear como ejemplo la tabla proveedores de la práctica 1:



```
USE practica1;
DROP TABLE IF EXISTS dbo.PROVEEDORES;
CREATE TABLE dbo.PROVEEDORES(
    IDPROV int NOT NULL,
    NOMBRE nvarchar(50) NULL,
    CATEGORIA int NULL,
    CIUDAD nvarchar(50) NULL
);
```

Mediante **USE** definimos sobre qué base de datos vamos a trabajar. Es importante usarlo para evitar confusiones y errores.

Con **DROP IF EXISTS** verificamos si en la base de datos ya existe una tabla PROVEEDORES. En caso de que exista la eliminamos, para poder crear la tabla sin problemas. Este formato de **DROP** se incorporó en la versión 2016. La manera de hacer esto mismo en versiones anteriores es:

```
IF OBJECT_ID('dbo.PROVEEDORES', 'U') IS NOT NULL DROP TABLE dbo.PROVEEDORES;
```

Este formato también funciona en la versión 2016.

En próximas unidades veremos en detalle el uso de funciones y de scripts.

Noten que para cada atributo se define un tipo de datos y si puede tener datos en NULL. Esto depende del modelo lógico que estemos implementando. En este caso el único atributo que no acepta NULL es IDPROV que es la clave primaria.

Recordemos que una clave primaria es un atributo o conjunto de atributos que definen únicamente un registro, por lo tanto, no puede aceptar un valor NULL en ninguno de los atributos que formen parte de la clave.

La clave primaria puede definirse en el momento de creación de la tabla, o podemos definirla luego mediante el comando **ALTER TABLE**.



```
ALTER TABLE dbo.PROVEEDORES  
ADD CONSTRAINT PK_PROVEEDORES  
PRIMARY KEY(IDPROV);
```

En algunos modelos lógicos puede darse el caso de que alguno de los campos que no sea la clave primaria tampoco permita valores repetidos. Para eso tenemos la restricción **UNIQUE**

Supongamos que no puede repetirse el nombre de los proveedores:

```
ALTER TABLE dbo.PROVEEDORES  
ADD CONSTRAINT UNQ_PROVEEDORES_NOMBRE  
UNIQUE(NOMBRE);
```

CLAVES FORÁNEAS

Las claves foráneas sirven para implementar la integridad referencial, es decir, cuando un atributo o conjunto de atributos es clave primaria o candidata en otra tabla, sólo puede tomar valores válidos de instancias de la tabla o ser NULL (en caso de que sean atributos en los que se permita NULL). Tener en cuenta que la tabla referenciada puede ser la misma tabla en la que se está definiendo (como en los casos de relaciones unarias).

Primero creamos la tabla CONTRATOS:

```
DROP TABLE IF EXISTS dbo.CONTRATOS;  
  
CREATE TABLE dbo.CONTRATOS(  
    IDPROV int NOT NULL,  
    IDART int NOT NULL,  
    CANT int NULL,  
    CONSTRAINT PK_CONTRATOS PRIMARY KEY (IDPROV, IDART)  
)
```

Y luego agregamos la clave foránea a la tabla PROVEEDORES:

```
ALTER TABLE dbo.CONTRATOS ADD CONSTRAINT FK_CONTRATOS_PROVEEDORES FOREIGN KEY(IDPROV)  
REFERENCES dbo.PROVEEDORES (IDPROV)
```

RESTRICCIÓN CHECK

Las restricciones **CHECK** permiten definir predicados que una fila debe cumplir la ser ingresado en una tabla o al ser modificada.

Por ejemplo, podríamos definir un **CHECK** para asegurarnos que las cantidades de la tabla CONTRATOS no puedan ser negativas:

```
ALTER TABLE dbo.CONTRATOS  
ADD CONSTRAINT CHK_CONTRATOS_CANT  
CHECK(CANT >= 0);
```

Si alguien intenta insertar o modificar una fila de la tabla CONTRATOS y poner un importe negativo en CANT la operación será rechazada por el SQL Server (la operación es rechazada cuando el predicado da falso, y es aceptada cuando da verdadero o desconocido).

Tanto al agregar restricciones **CHECK** o **FOREIGN KEY**, se puede especificar la opción WITH NOCHECK que le dice al SQL Server que no aplique el control de la restricción sobre los datos ya existentes en la tabla. Pero hacer esto es considerado una mala práctica, ya que no podemos asegurar la consistencia de los datos.

Tanto **CHECK** como **FOREIGN KEY** pueden ser deshabilitados y luego se pueden volver a habilitar.



```
ALTER TABLE dbo.CONTRATOS NOCHECK CONSTRAINT ALL; --Deshabilitar CHECK y FOREIGN KEY  
ALTER TABLE dbo.CONTRATOS WITH CHECK CHECK CONSTRAINT ALL; --Habilitar
```

RESTRICCIÓN DEFAULT

Se asocia a un atributo. Consta de una expresión que es utilizada como valor por defecto (default) cuando no es especificado un valor explícito para el atributo al insertar una nueva fila.

Por ejemplo, podemos definir para el campo *orderts* de la tabla *Orders* que asuma como valor por defecto la fecha y hora actual del sistema.

```
ALTER TABLE dbo.Orders  
ADD CONSTRAINT DFT_Orders_orderts  
DEFAULT(SYSDATETIME()) FOR orderts;
```

NULL Y LA LÓGICA DE TRES VALORES

El T-SQL utiliza una lógica de tres valores (verdadero, falso y desconocido) con las siguientes tablas de verdad

OR	Verdadero	Falso	Desconocido
Verdadero	Verdadero	Verdadero	Verdadero
Falso	Verdadero	Falso	Desconocido
Desconocido	Verdadero	Desconocido	Desconocido

AND	Verdadero	Falso	Desconocido
Verdadero	Verdadero	Falso	Desconocido
Falso	Falso	Falso	Desconocido
Desconocido	Desconocido	Desconocido	Desconocido

NOT	
Verdadero	Falso
Falso	Verdadero
Desconocido	Desconocido

ORDEN LÓGICO DE EJECUCIÓN

SQL Server procesa los distintos comandos SQL en un orden lógico determinado:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
 - Expresiones
 - DISTINCT
- ORDER BY
 - TOP/OFFSET-FETCH

Conocer y entender el orden lógico nos ayudará a evitar errores lógicos e incluso de sintaxis.



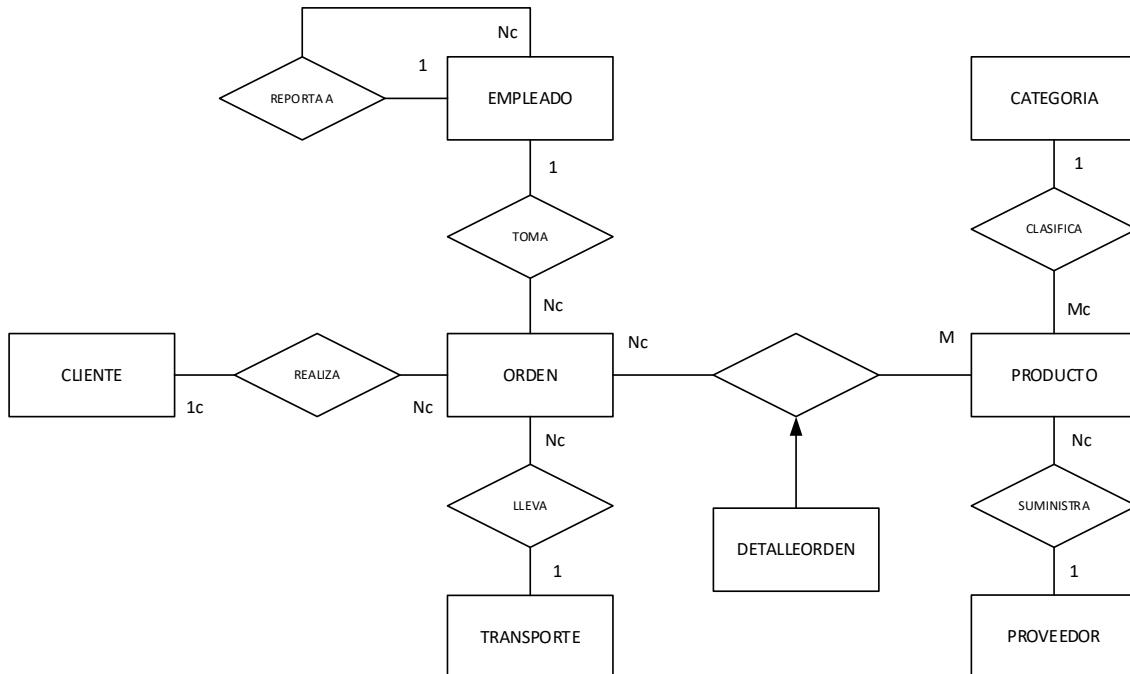
BASE TSQLV6ES

En este apunte utilizaremos como ejemplo la base TSQLV6ES. Esta base es derivada de la base Northwind⁸, que es una base de datos de ejemplo creada por Microsoft para sus primeras versiones de bases de datos.

Esta base, a diferencia de la base Northwind, hace uso de los esquemas de SQL Server. Utiliza los esquemas Ventas, RH y Producción

A continuación, se incluye un DER y un MER con la estructura de la base de datos.

DER



DICCIONARIO DE DATOS

CATEGORIA	=	@ID + Nombre + Descripcion
CLIENTE	=	@ID + NombreEmpresa + NombreContacto + CargoContacto + Direccion + Ciudad + Region + CodigoPostal + Pais + Telefono + Fax
DETALLEORDEN	=	ORDEN-ref-Nc + ORDEN-ref-Mc + PrecioUnitario + Cantidad + Descuento
EMPLEADO	=	@ID + Apellido + Nombre + Puesto + Saludo + Nacimiento + FechaAlta + Direccion + Ciudad + Region + CodigoPostal + Pais + Telefono
ORDEN	=	@ID + FechaOrden + FechaRequerida + FechaDespacho + Flete + NombreEnvio + DireccionEnvio + CiudadEnvio + RegionEnvio + CodigoPostalEnvio + PaisEnvio
PRODUCTO	=	@ID + Nombre + PrecioUnitario + Discontinuado
PROVEEDOR	=	@ID + NombreEmpresa + NombreContacto + CargoContacto + Direccion + Ciudad + Region + CodigoPostal + Pais + Telefono + Fax
TRANSPORTE	=	@ID + NombreEmpresa + Telefono

⁸ <https://github.com/microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs>

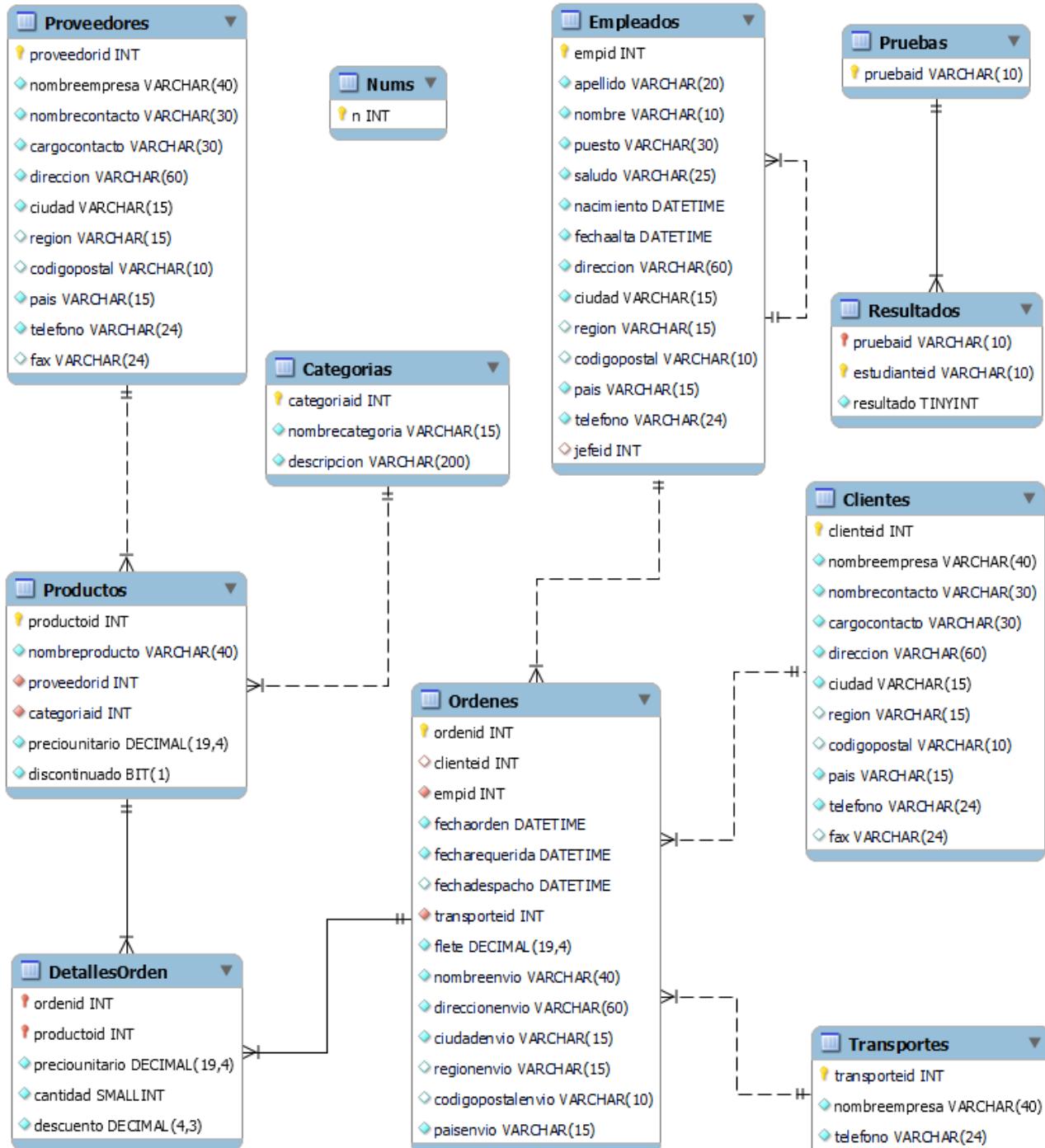
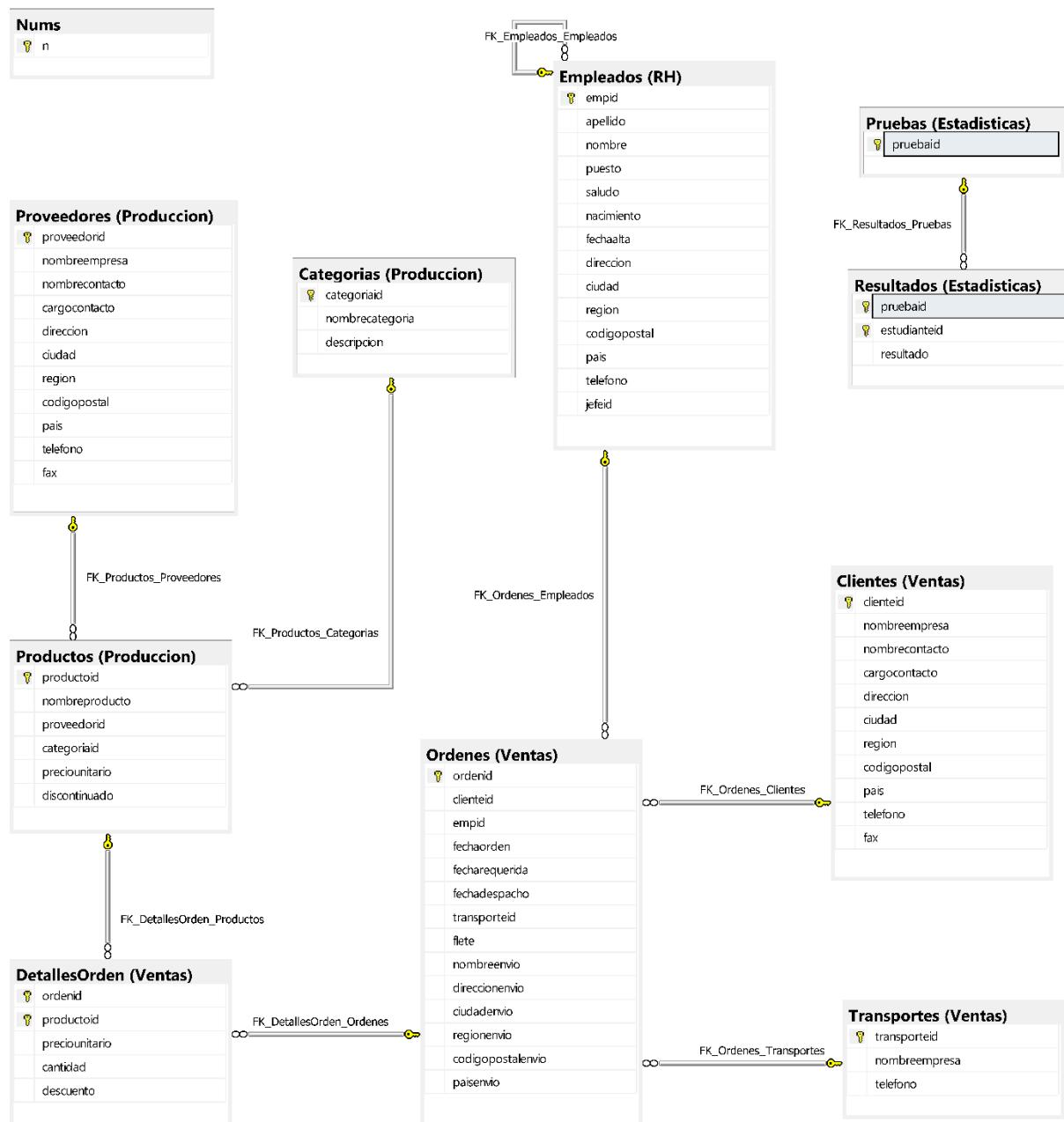




DIAGRAMA DE BASE DE DATOS





UNIDAD 1

FILTROS TOP Y OFFSET-FETCH

Los resultados de las consultas se pueden filtrar mediante predicados lógicos utilizando WHERE o HAVING según corresponda. Existen además otros filtros en SQL que permiten filtrar filas en basados en su número y ordenamiento.

El filtro TOP está incorporado en SQL Server desde la versión **7.0**. El filtro OFFSET-FETCH se incorporó en la versión **2012**.

FILTRO TOP

Es una función propietaria de T-SQL, esto significa que no es parte del estándar SQL y puede no estar implementada en otros motores.

Permite limitar el número o porcentaje de filas que devuelve como resultado una consulta. Está basado en dos elementos básicos, uno es el número o porcentaje de filas a devolver, y el otro es el ordenamiento. Por ejemplo, para devolver de la tabla Ordenes las cinco más recientes, se deberá especificar *TOP (5)* en el *SELECT* y *fechaorden DESC* en el *ORDER BY*, como puede verse en el siguiente ejemplo:

```
USE TSQLV6ES;
SELECT TOP (5) ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
ORDER BY fechaorden DESC;
```

ordenid	fechaorden	clienteid	empid
11077	2022-05-06	65	1
11076	2022-05-06	9	4
11075	2022-05-06	68	8
11074	2022-05-06	73	7
11073	2022-05-05	58	2

Recuerden que *ORDER BY* es evaluada luego del *SELECT* en el orden lógico, que incluye la opción *DISTINCT*. Esto mismo ocurre con *TOP*, que está basado en *ORDER BY* para poder realizar el filtrado. Esto significa que, si en el *SELECT* se especifica usar *DISTINCT*, el filtro *TOP* es evaluado luego de que los duplicados han sido removidos.

Es importante notar que, si se utiliza *TOP*, la cláusula *ORDER BY* sirve para un doble propósito en la consulta. Por un lado, define el orden de presentación de las filas en el resultado. Por otro lado, define junto con *TOP* las filas a filtrar. En el ejemplo que detallamos previamente, las cinco órdenes más recientes son mostradas en orden descendente de fecha.

La cláusula *TOP* puede ser utilizada con la opción *PERCENT*, en cuyo caso SQL Server calcula el número de filas a mostrar basado en un porcentaje del número de filas del resultado, pero redondeado hacia arriba. Por ejemplo, la siguiente consulta devuelve el uno por ciento de órdenes más recientes.



```
USE TSQLV6ES;
SELECT TOP (1) PERCENT ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
ORDER BY fechaorden DESC;
```

ordenid	fechaorden	clienteid	empid
11074	2022-05-06	73	7
11075	2022-05-06	68	8
11076	2022-05-06	9	4
11077	2022-05-06	65	1
11070	2022-05-05	44	2
11071	2022-05-05	46	1
11072	2022-05-05	20	4
11073	2022-05-05	58	2
11067	2022-05-04	17	1

Esto devuelve nueve filas porque la tabla Ordenes tiene 830 filas, y el uno por ciento de 830, redondeado hacia arriba es 9.

Pero comparemos los resultados de ambas consultas

TOP (5)				TOP (1) PERCENT			
ordenid	fechaorden	clienteid	empid	ordenid	fechaorden	clienteid	empid
11077	2022-05-06	65	1	11074	2022-05-06	73	7
11076	2022-05-06	9	4	11075	2022-05-06	68	8
11075	2022-05-06	68	8	11076	2022-05-06	9	4
11074	2022-05-06	73	7	11077	2022-05-06	65	1
11073	2022-05-05	58	2	11070	2022-05-05	44	2
Muestra los primeros 5				11071	2022-05-05	46	1
				11072	2022-05-05	20	4
				11073	2022-05-05	58	2
				11067	2022-05-04	17	1
Muestra los primeros 9, pero los primeros 5, ¿no deberían coincidir con el TOP(5)?							

TIEBREAKER

Cuando en una cláusula *ORDER BY*, los atributos utilizados para el ordenamiento no son claves primarias o claves candidatas, se dice que el resultado obtenido es no determinístico, ya que ante igualdad en el campo por el que se está ordenado, el orden en que se muestran las filas con el mismo no está determinado, e incluso se pueden obtener ordenamientos distintos en distintos momentos, o por distintas personas.

En algunos casos este comportamiento puede aceptado por las reglas del negocio, pero en otros casos, puede ser inaceptable.

Estos casos se ven más afectados aún al utilizar filtros de cantidad de filas, como *TOP*, ya que las filas obtenidas pueden ser distintas en distintas ocasiones, cuando por lo general sería lógico obtener el mismo resultado.

Para evitar estas anomalías, cuando las mismas sean inaceptables se debe utilizar la técnica del *TIEBREAK*, que en definitiva no es otra cosa que agregar más criterios al ordenamiento, hasta lograr que el resultado de este sea único (formando o agregando una clave primaria o candidata)

En los ejemplos anterior, podríamos agregar el campo *ordenid*:



```
USE TSQLV6ES;
SELECT TOP (5) ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
ORDER BY fechaorden DESC, ordenid DESC;
```

ordenid	fechaorden	clienteid	empid
11077	2022-05-06	65	1
11076	2022-05-06	9	4
11075	2022-05-06	68	8
11074	2022-05-06	73	7
11073	2022-05-05	58	2

```
USE TSQLV6ES;
SELECT TOP (1) PERCENT ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
ORDER BY fechaorden DESC, ordenid DESC;
```

ordenid	fechaorden	clienteid	empid
11077	2022-05-06	65	1
11076	2022-05-06	9	4
11075	2022-05-06	68	8
11074	2022-05-06	73	7
11073	2022-05-05	58	2
11072	2022-05-05	20	4
11071	2022-05-05	46	1
11070	2022-05-05	44	2
11069	2022-05-04	80	1

Podemos verificar que ahora sí, las primeras cinco filas de ambos resultados coinciden, ya que ambas consultas son determinísticas,

EMPATES (TIES)

Observando con atención el resultado de la segunda consulta, podemos ver que hay tres filas que no salen en el primer resultado, pero que tienen exactamente el mismo valor de fecha.

De acuerdo con las reglas de negocio que apliquen a la consulta, esto podría estar bien y no generar mayores problemas, pero hay casos en los que esto sería inaceptable.

Recordemos como ejemplo lo que pasó en los Juegos Olímpicos de Rio 2016, en la competencia de los 100 metros estilo Mariposa.⁹

⁹ <https://www.olympic.org/rio-2016/swimming/100m-butterfly-men>



Atleta	Tiempo
Joseph Schooling Singapur	50.39 s
Michael Phelps Estados Unidos	51.14 s
Chad le Clos Sudáfrica	51.14 s
László Cseh Hungria	51.14 s



Si se realiza una consulta con TOP (3) para determinar el podio, cometéramos seguramente una injusticia, ya que uno de los participantes que obtuvo el segundo mejor tiempo quedaría fuera del resultado.

Para los que no lo saben o no lo recuerdan, el podio de esta prueba quedó conformado por una medalla de oro y tres de plata, así que nadie quedó fuera (como pueden observar en la foto).

SQL Server nos permite también obtener las filas necesarias para este tipo de casos, permitiendo que el TOP devuelva además los casos que generen empates.

La opción es *WITH TIES* y podemos verla en el siguiente ejemplo:

```
USE TSQLV6ES;
SELECT TOP (5) WITH TIES ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
ORDER BY fechaorden DESC;
```

ordenid	fechaorden	clienteid	empid
11077	2022-05-06	65	1
11076	2022-05-06	9	4
11075	2022-05-06	68	8
11074	2022-05-06	73	7
11073	2022-05-05	58	2
11072	2022-05-05	20	4
11071	2022-05-05	46	1
11070	2022-05-05	44	2

En este caso, a pesar de que utilizamos TOP (5), la consulta devuelve los primeros cinco, pero al detectar que hay más casos con el mismo valor de fechaorden, tres en este caso, los agrega como parte del resultado.

FILTRO OFFSET-FETCH

El filtro *TOP* es muy práctico, pero tiene un par de desventajas, no es parte del estándar SQL, y no tiene la capacidad de realizar saltos.

En el estándar de SQL se define un filtro llamado *OFFSET-FETCH* que tiene la capacidad de realizar saltos, lo que lo hace muy práctico para realizar paginados ad-hoc. Se incorporó al estándar en la versión SQL:2008.

El soporte para *OFFSET-FETCH* fue incluido en SQL Server en la versión 2012.

En SQL Server el filtro *OFFSET-FETCH* es considerado parte de la cláusula *ORDER BY*. Mediante *OFFSET* se indica cuantas filas saltar, y mediante *FETCH* se puede indicar cuántas filas filtrar luego de las filas saltadas. Por ejemplo:



```
USE TSQLV6ES;
SELECT ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
ORDER BY fechaorden, ordenid
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

En definitiva, se saltan 50 filas y se devuelven las siguientes 25.

Las consultas que utilizan *OFFSET-FETCH* deben tener la cláusula *ORDER BY*. Además, la cláusula *FETCH* no puede ser utilizada si antes no se utilizó la cláusula *OFFSET*, si no se quieren saltar filas se deberá indicar *OFFSET 0 ROWS*. En cambio, sí se puede utilizar *OFFSET* sin *FETCH*, en este caso se saltan las filas y se devuelven todas las filas restantes.

La idea de este filtro es que pueda ser escrito de tal manera que pueda leerse y entenderse de manera intuitiva (en inglés). Por lo tanto, se puede usar de manera indistinta *ROW* y *ROWS*, dependiendo de los números utilizados. De modo similar se puede usar de manera indistinta *NEXT* o *FIRST*.

OFFSET-FETCH permite entonces el salteo que *TOP* no permite, pero no soporta opciones como *PERCENT* o *WITH TIES*, curiosamente estas opciones están incluidas en el estándar de SQL, pero T-SQL no las ha implementado.

Pero como *OFFSET-FETCH* es parte del estándar, se recomienda su uso como primera opción, y usar *TOP* sólo cuando se necesite alguna de sus opciones específicas.

```
<result offset clause> ::=  
  OFFSET <offset row count> { ROW | ROWS }  
  
<fetch first clause> ::=  
  FETCH { FIRST | NEXT } [ <fetch first quantity> ] { ROW | ROWS } { ONLY | WITH TIES }  
  
<fetch first quantity> ::=  
  <fetch first row count>  
  | <fetch first percentage>  
  
<offset row count> ::=  
  <simple value specification>  
  
<fetch first row count> ::=  
  <simple value specification>  
  
<fetch first percentage> ::=  
  <simple value specification> PERCENT
```

PREDICADOS Y OPERADORES

T-SQL utiliza predicados lógicos en instrucciones como *WHERE*, *HAVING*, restricciones *CHECK*, y otras. Los predicados son expresiones lógicas cuyo resultado puede ser Verdadero, Falso o Desconocido (NULL).

Se pueden combinar predicados con operadores lógicos Y (*AND*) y O (*OR*), y se pueden incluir otros operadores, como los de comparación.

En T-SQL podemos utilizar predicados como *IN*, *BETWEEN* y *LIKE*.

IN permite verificar si un valor o una expresión escalar es igual a al menos uno de los elementos de un conjunto. Por ejemplo, la siguiente consulta permite obtener las órdenes cuyo ID sea igual a 10248, 10249 o 10250.

```
USE TSQLV6ES;
SELECT ordenid, empid, fechaorden
FROM Ventas.Ordenes
WHERE ordenid IN(10248, 10249, 10250);
```

BETWEEN permite verificar si un valor o una expresión escalar está dentro de un rango, considerando los límites como parte del rango, es decir, los incluye. Por ejemplo, la siguiente consulta devuelve las órdenes cuyo ID esté entre 10300 y 10310, ambos inclusive.

```
USE TSQLV6ES;
SELECT ordenid, empid, fechaorden
FROM Ventas.Ordenes
WHERE ordenid BETWEEN 10300 AND 10310;
```



LIKE permite verificar si una cadena de caracteres cumple con un determinado patrón o expresión. Por ejemplo, la siguiente consulta devuelve los empleados cuyo apellido comience con la letra D.

```
USE TSQLV6ES;
SELECT empid, nombre, apellido
FROM RH.Empleados
WHERE apellido LIKE N'D%';
```

La **N** que antecede a la expresión se utiliza para denotar que la cadena a comparar es de tipo Unicode (*NCHAR* o *NVARCHAR*).

T-SQL soporta los siguientes operadores de comparación:

=, >, <, >=, <=, !=, !=, !<

Los últimos tres no son estándar, y como existen alternativas que sí están en el estándar, no se recomienda utilizarlos.

T-SQL soporta los cuatro operadores aritméticos básicos:

+, -, *, /

Además, soporta el operador módulo (%) que devuelve el resto de la división entera.

Por ejemplo, la siguiente consulta devuelve el valor neto, luego de realizar operaciones aritméticas sobre los campos cantidad, precio unitario y descuento.

```
USE TSQLV6ES;
SELECT ordenid, productoid, cantidad, preciounitario, descuento,
cantidad * preciounitario * (1 - descuento) AS valorneto
FROM Ventas.DetallesOrden;
```

PRECEDENCIA DE TIPOS DE DATOS

Al realizar una operación entre dos expresiones escalares, el tipo de datos del resultado lo determina el que tiene la precedencia de datos más alta. Si los dos operadores tienen el mismo tipo de datos, el resultado de la expresión tendrá el mismo tipo de datos.

De acuerdo con la documentación¹⁰, SQL Server maneja la siguiente precedencia:

1. user-defined data types (la más alta)
2. **sql_variant**
3. **xml**
4. **datetimeoffset**
5. **datetime2**
6. **datetime**
7. **smalldatetime**
8. **date**
9. **time**
10. **float**
11. **real**
12. **decimal**
13. **money**

¹⁰ <https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-type-precedence-transact-sql>



14. **smallmoney**
15. **bigint**
16. **int**
17. **smallint**
18. **tinyint**
19. **bit**
20. **ntext**
21. **text**
22. **image**
23. **timestamp**
24. **uniqueidentifier**
25. **nvarchar** (incluyendo **nvarchar(max)**)
26. **nchar**
27. **varchar** (incluyendo **varchar(max)**)
28. **char**
29. **varbinary** (incluyendo **varbinary(max)**)
30. **binary** (la más baja)

PRECEDENCIA DE OPERADORES

Cuando en una expresión aparecen múltiples operadores, SQL Server aplica la siguiente precedencia:

1. () (Paréntesis)
2. * (Multiplicación), / (División), % (Módulo)
3. + (Positivo), – (Negativo), + (Suma), + (Concatenación), – (Resta)
4. =, >, <, >=, <=, <>, !=, !=, !< (Operadores de comparación)
5. NOT
6. AND
7. BETWEEN, IN, LIKE, OR
8. = (Asignación)

EXPRESIONES CASE

Una expresión *CASE* es una expresión escalar que devuelve un valor basado en lógica condicional. Se basa en el estándar SQL.

Dado que *CASE* es una expresión escalar, se permite siempre que se permiten expresiones escalares, como en las cláusulas *SELECT*, *WHERE*, *HAVING* y *ORDER BY* y en las restricciones *CHECK*.

Hay dos formas de expresiones *CASE*: simple y de expresión. Se utiliza la forma simple para comparar un valor o una expresión escalar con una lista de valores posibles y devolver un valor para la primera coincidencia. Si ningún valor en la lista es igual al valor probado, la expresión *CASE* devuelve el valor que aparece en la cláusula *ELSE* (si existe). Si la expresión *CASE* no tiene una cláusula *ELSE*, devuelve *NULL*.

La forma simple de *CASE* tiene un único valor de prueba o expresión justo después de la palabra clave *CASE* que se compara con una lista de valores posibles en las cláusulas *WHEN*. La forma de expresión de *CASE* es más flexible en el sentido de que puede especificar predicados en las cláusulas *WHEN* en lugar de restringirse a usar comparaciones de igualdad. La expresión *CASE* buscada devuelve el valor en la cláusula *THEN* que está asociada con el primer predicado *WHEN* que se evalúa como *TRUE*. Si ninguno de los predicados *WHEN* se evalúa como *TRUE*, la expresión *CASE* devuelve el valor que aparece en la cláusula *ELSE* (o *NULL* si no existe una cláusula *ELSE*).

Cada expresión *CASE* simple se puede convertir en el formato *CASE* de expresión, pero lo contrario no es cierto.



T-SQL soporta algunas funciones que puede considerar como abreviaturas de la expresión CASE: *ISNULL*, *COALESCE*, *IIF* y *CHOOSE*. Tenga en cuenta que, de los cuatro, sólo *COALESCE* es estándar.

La función *ISNULL* acepta dos argumentos como entrada y devuelve el primero que no es *NULL* o *NULL* si ambos son *NULL*. Por ejemplo, *ISNULL* (col1, '') devuelve el valor col1 si no es *NULL* y una cadena vacía si es *NULL*. La función *COALESCE* es similar, sólo que soporta dos o más argumentos y devuelve el primero que no es *NULL* o *NULL* si todos son *NULL*.

Las funciones *IIF* y *CHOOSE* no estánndar se agregaron a T-SQL para permitir migraciones más fáciles de Microsoft Access. La función *IIF* (<expresión_lógica>, <expr1>, <expr2>) devuelve expr1 si la expresión lógica es *TRUE* y devuelve expr2 de lo contrario. Por ejemplo, la expresión *IIF* (col1 <> 0, col2 / col1, NULL) devuelve el resultado de col2 / col1 si col1 no es cero; De lo contrario, devuelve un *NULL*. La función *CHOOSE* (<índice>, <expr1>, <expr2>, ..., <exprn>) devuelve la expresión de la lista en la ubicación especificada por el índice. Por ejemplo, la expresión *CHOOSE* (3, col1, col2, col3) devuelve el valor de col3. Por supuesto, las expresiones reales que utilizan la función *CHOOSE* tienden a ser más dinámicas, por ejemplo, depender de la entrada del usuario.

NULLS

El tratamiento de *NULL* y *DESCONOCIDO* en SQL puede ser confuso porque intuitivamente las personas están más acostumbradas a pensar en términos de lógica de dos valores (VERDADERO y FALSO). Para añadir a la confusión, los diferentes elementos de lenguaje en SQL tratan *NULLs* y *DESCONOCIDO* de manera inconsistente.

Comencemos con la lógica de tres valores. Una expresión lógica que implica sólo valores no *NULL* se evalúa como *TRUE* o *FALSE*. Cuando la expresión lógica implica un valor faltante, se evalúa como *DESCONOCIDO*. Por ejemplo, considere el predicado salario > 0.

Cuando el salario es igual a 1.000, la expresión se evalúa como *TRUE*. Cuando el salario es igual a -1000, la expresión se evalúa como *FALSO*. Cuando el salario es *NULL*, la expresión se evalúa como *DESCONOCIDO*.

SQL trata *TRUE* y *FALSO* de una manera intuitiva y probablemente esperada. Por ejemplo, si el predicado salario > 0 aparece en un filtro de consulta (como en una cláusula WHERE o HAVING), se devuelven filas o grupos para los que la expresión se evalúa como *TRUE*, mientras que los que se evalúan como *FALSO* se descartan. Del mismo modo, si el predicado salario > 0 aparece en una restricción CHECK de una tabla, se aceptan instrucciones INSERT o UPDATE para las que la expresión se evalúa como *TRUE* para todas las filas, mientras que aquellas para las que la expresión se evalúa *FALSE* para cualquier fila se rechazan.

SQL tiene diferentes tratamientos para *DESCONOCIDO* en diferentes elementos de lenguaje (y para algunas personas, no necesariamente los tratamientos esperados). El tratamiento que SQL tiene para los filtros de consulta es "aceptar *TRUE*", lo que significa que tanto *FALSO* como *DESCONOCIDO* se descartan. A la inversa, la definición del tratamiento que SQL tiene para las restricciones CHECK es "rechazar *FALSE*", lo que significa que tanto *TRUE* como *DESCONOCIDO* son aceptados. Si SQL usara una lógica predicada de dos valores, no habría habido una diferencia entre las definiciones "aceptar VERDADERO" y "rechazar FALSO".

Pero con lógica de tres valores, "aceptar *TRUE*" rechaza *DESCONOCIDO*, mientras que "rechazar *FALSE*" lo acepta. Con el predicado salario > 0 del ejemplo anterior, un salario *NULO* haría que la expresión se evaluara en *DESCONOCIDO*. Si este predicado aparece en la cláusula WHERE de una consulta, se descartará una fila con un salario *NULL*. Si este predicado aparece en una restricción CHECK en una tabla, se aceptará una fila con un salario *NULL*.

Uno de los aspectos difíciles del valor lógico *DESCONOCIDO* es que cuando se lo niega, se obtiene *DESCONOCIDO*. Por ejemplo, dado el predicado NOT (salario > 0), cuando el salario es *NULL*, salario > 0 se evalúa como *DESCONOCIDO*, y NO *DESCONOCIDO* sigue siendo *DESCONOCIDO*.



Lo que algunas personas encuentran sorprendente es que una expresión que compara dos NULL (NULL = NULL) se evalúa como DESCONOCIDO. El razonamiento para esto desde la perspectiva de SQL es que un NULL representa un valor faltante, y realmente no se puede saber si un valor faltante es igual a otro. Por lo tanto, SQL proporciona los predicados IS NULL y IS NOT NULL, que debe utilizar en lugar de = NULL y <> NULL.

SQL también trata NULLs de manera inconsistente en diferentes elementos del lenguaje para fines de comparación y clasificación. Algunos elementos tratan a dos NULL como iguales entre sí, y otros los tratan como diferentes.

Por ejemplo, para fines de agrupación y clasificación, dos NULL se consideran iguales. Es decir, la cláusula GROUP BY organiza todos los NULL en un grupo como los valores presentes y la cláusula ORDER BY ordena todos los NULL juntos. El SQL estándar deja a la implementación del producto determinar si los NULL clasifican antes de los valores definidos o después de ellos, pero debe ser consistente dentro de la implementación. T-SQL ordena NULL antes de los valores definidos.

Como se mencionó anteriormente, los filtros de consulta "aceptan TRUE". Una expresión que compara dos NULL produce DESCONOCIDO; por lo tanto, dicha fila es filtrada.

Con el fin de aplicar una restricción UNIQUE, SQL estándar trata NULL como diferentes entre sí (permitiendo varios NULL). Por el contrario, en T-SQL, una restricción UNIQUE considera dos NULL como igual (permitiendo sólo un NULL).

La complejidad en el manejo de NULL a menudo resulta en errores lógicos. Por lo tanto, debe pensar en ellos en cada consulta que escriba. Si el tratamiento por defecto no es lo buscado, se debe intervenir explícitamente; De lo contrario, sólo asegúrese de que el comportamiento predeterminado es, de hecho, lo que desea.

En SQL Server 2022 se incorporó una novedad para el tratamiento de los NULL mediante un predicado que compara la igualdad de dos expresiones y garantiza un resultado verdadero o falso, incluso si uno o ambos operandos son NULL.

IS [NOT] DISTINCT FROM es un predicado que se usa en la condición de búsqueda de las cláusulas WHERE y HAVING, en las condiciones de JOIN de las cláusulas FROM y en otras construcciones que requieren un valor booleano.

Su sintaxis es:

expresión IS [NOT] DISTINCT FROM expresión

La comparación de un valor *NULL* con cualquier otro valor, incluido otro *NULL*, generará un resultado desconocido. *IS [NOT] DISTINCT FROM* siempre devolverá true o false, ya que tratará los valores *NULL* como valores conocidos cuando se usen como operador de comparación.

En la tabla de ejemplo siguiente se usan valores A y B para ilustrar el comportamiento de *IS [NOT] DISTINCT FROM*:

A	B	A = B	A IS NOT DISTINCT FROM B
0	0	Verdadero	Verdadero
0	1	Falso	Falso
0	NULL	Desconocido	Falso
NULL	NULL	Desconocido	Verdadero



FUNCIONES GREATEST Y LEAST

En SQL Server 2022 se incorporaron estas funciones, que permiten obtener el máximo y el mínimo de una lista de expresiones de una misma fila, como funcionan *MAX* y *MIN* para una misma expresión en diferentes filas.

Estas funciones son simples y fáciles de usar como podremos ver en el siguiente ejemplo. La consulta devuelve las órdenes realizadas por cliente 8, y utiliza *GREATEST* y *LEAST* para calcular, para cada orden, la más temprana y la más tardía entre las fechas requeridas y despachadas.

```
USE TSQLV6ES;
SELECT ordenid, fecharequerida, fechadespacho,
GREATEST(fecharequerida, fechadespacho) AS fechatardia,
LEAST(fecharequerida, fechadespacho) AS fechatemprana
FROM Ventas.Ordenes
WHERE clienteid = 8;
```

ordenid	fecharequerida	fechadespacho	fechatardia	fechatemprana
10326	2020-11-07	2020-10-14	2020-11-07	2020-10-14
10801	2022-01-26	2021-12-31	2022-01-26	2021-12-31
10970	2022-04-07	2022-04-24	2022-04-24	2022-04-07

Estas funciones no se limitan a valores numéricos, utilizan la precedencia de los tipos de datos, y devuelven el tipo de mayor precedencia. Soportan entre 1 y 254 parámetros, los valores NULL son ignorados, pero si todos los parámetros son NULL, devolverá NULL.

OPERACIONES SIMULTÁNEAS

SQL soporta un concepto llamado operaciones "simultáneas", lo que significa que todas las expresiones que aparecen en la misma fase de procesamiento de consultas lógicas se evalúan lógicamente en el mismo momento. La razón de esto es que todas las expresiones que aparecen en la misma fase lógica son tratadas como un conjunto, y en un conjunto no tienen orden a sus elementos.

Este concepto explica por qué, por ejemplo, no podemos hacer referencia a los alias de columna asignados en la cláusula *SELECT* dentro de la misma cláusula *SELECT*. Consideren la siguiente consulta:

```
USE TSQLV6ES;
SELECT
ordenid,
YEAR(fechaorden) AS anioorden,
anioorden + 1 AS aniosiguiente
FROM Ventas.Ordenes;
```

La referencia al alias de columna *anioorden* en la tercera expresión de la lista *SELECT* no es válida, aunque la expresión de referencia aparece después de aquella en la que se asigna el alias. La razón es que, lógicamente, no hay un orden de evaluación de las expresiones en la cláusula *SELECT*: es un conjunto de expresiones. Conceptualmente, todas las expresiones se evalúan en el mismo momento. Por lo tanto, esta consulta genera el siguiente error:

**Msg 207, Level 16, State 1, Line 4
Invalid column name 'anioorden'.**

Aquí hay otro ejemplo de las ramificaciones de las operaciones de todo a la vez: supongamos que tenemos una tabla llamada T1 con dos columnas de enteros llamadas *col1* y *col2*, y deseamos devolver todas las filas para las que *col2/col1* es mayor que 2. Porque podría haber en la tabla en las que *col1* es cero, debemos asegurarnos de que la división no se realice en esos casos; de lo contrario, la consulta fallaría debido a un error de división por cero. Así que escribimos una consulta usando el siguiente formato:



```
SELECT col1, col2  
FROM dbo.T1  
WHERE col1 <> 0 AND col2/col1 > 2;
```

Sería lógico asumir que SQL Server evalúa las expresiones de izquierda a derecha, y que si la expresión `col1 <> 0` se evalúa como FALSO, SQL Server provocará un cortocircuito, es decir, que no se molestará en evaluar la expresión `col2/col1 > 2` porque en este punto se sabe que toda la expresión es FALSA. Entonces, podríamos pensar que esta consulta nunca debería producir un error de división por cero.

SQL Server admite cortocircuitos, pero debido al concepto de operaciones todo a la vez, es libre de procesar las expresiones en la cláusula `WHERE` en cualquier orden. SQL Server suele tomar decisiones como esta en función de las estimaciones de costos. Podemos ver que si SQL Server decide procesar primero la expresión `col2/col1 > 2`, esta consulta podría fallar debido a un error de división por cero.

Tenemos varias formas de evitar una falla aquí. Por ejemplo, se garantiza el orden en que se evalúan las cláusulas `WHEN` de una expresión `CASE`. Por lo tanto, podríamos reescribir la consulta de la siguiente manera:

```
SELECT col1, col2  
FROM dbo.T1  
WHERE  
CASE  
WHEN col1 = 0 THEN 'no' -- o 'si' si la fila debe ser devuelta  
WHEN col2/col1 > 2 THEN 'si'  
ELSE 'no'  
END = 'si';
```

En las filas donde `col1` es igual a cero, la primera cláusula `WHEN` se evalúa como VERDADERO y la expresión `CASE` devuelve la cadena 'no'. (Reemplace 'no' con 'si' si desea devolver la fila cuando `col1` es igual a cero). Sólo si la primera expresión `CASE` no se evalúa como VERDADERO, lo que significa que `col1` no es 0, la segunda cláusula `WHEN` verifica si la expresión `col2/col1 > 2` se evalúa como VERDADERO. Si es así, la expresión `CASE` devuelve la cadena 'si'. En todos los demás casos, la expresión `CASE` devuelve la cadena 'no'. El predicado en la cláusula `WHERE` devuelve VERDADERO solo cuando el resultado de la expresión `CASE` es igual a la cadena 'si'. Esto significa que aquí nunca habrá un intento de dividir por cero.

Esta solución resultó ser bastante complicada. En este caso particular, podemos usar una solución matemática que evita la división por completo:

```
SELECT col1, col2  
FROM dbo.T1  
WHERE (col1 > 0 AND col2 > 2*col1) OR (col1 < 0 AND col2 < 2*col1);
```

Se incluyó este ejemplo para explicar el concepto único e importante de las operaciones todo a la vez y para profundizar en el hecho de que SQL Server garantiza el orden de procesamiento de las cláusulas `WHEN` en una expresión `CASE`.

TRABAJANDO CON CADENAS DE CARACTERES

TIPOS DE DATOS

SQL Server admite dos tipos de tipos de datos de caracteres: regular y Unicode. Los tipos de datos regulares incluyen `CHAR` y `VARCHAR`, y los tipos de datos Unicode incluyen `NCHAR` y `NVARCHAR`.

Los caracteres normales utilizan 1 byte de almacenamiento para cada carácter, mientras que los datos Unicode requieren 2 bytes por carácter, y en casos en los que se necesita un par sustituto, se requieren 4 bytes.



Si elige un tipo de carácter normal para una columna, está restringido a un solo idioma además de inglés. El soporte lingüístico de la columna está determinado por el cotejo de la columna, que describiré en breve. Con tipos de datos Unicode, se admiten varios idiomas. Por lo tanto, si almacena datos de caracteres en varios idiomas, asegúrese de que utiliza tipos de caracteres Unicode y no caracteres normales.

Los dos tipos de tipos de datos de caracteres también difieren en la forma en que se expresan los literales. Al expresar un carácter normal literal, simplemente utilice comillas simples: 'Esta es una cadena de caracteres normal literal'. Cuando se expresa un carácter literal Unicode, es necesario especificar el carácter N (para National) como prefijo: N'Esto es una cadena de caracteres Unicode literal'.

Cualquier tipo de datos sin el elemento VAR (CHAR, NCHAR) en su nombre tiene una longitud fija, lo que significa que SQL Server conserva el espacio en la fila en función del tamaño definido de la columna y no en el número real de caracteres en la cadena de caracteres. Por ejemplo, cuando una columna se define como CHAR (25), SQL Server conserva espacio para 25 caracteres en la fila independientemente de la longitud de la cadena de caracteres almacenada. Debido a que no se requiere expansión de la fila cuando se expanden las cadenas, los tipos de datos de longitud fija son más adecuados para los sistemas enfocados en actualización. Pero como el consumo de almacenamiento no es óptimo con cadenas de longitud fija, son más costosas al leer datos.

Un tipo de datos con el elemento VAR (VARCHAR, NVARCHAR) en su nombre tiene una longitud variable, lo que significa que SQL Server utiliza tanto espacio de almacenamiento en la fila como se requiere para almacenar los caracteres que aparecen en la cadena de caracteres y dos bytes adicionales para datos de offset. Por ejemplo, cuando una columna se define como VARCHAR (25), el número máximo de caracteres admitidos es 25, pero en la práctica, el número real de caracteres en la cadena determina la cantidad de almacenamiento. Dado que el consumo de almacenamiento para estos tipos de datos es menor que el de los tipos de longitud fija, las operaciones de lectura son más rápidas. Sin embargo, las actualizaciones pueden resultar en expansión de filas, lo que podría resultar en movimiento de datos fuera de la página actual. Por lo tanto, las actualizaciones de datos que tienen tipos de datos de longitud variable son menos eficientes que las actualizaciones de datos que tienen tipos de datos de longitud fija.

También se pueden definir tipos de datos de longitud variable con el especificador MAX en lugar de un número máximo de caracteres. Cuando la columna se define con el especificador MAX, cualquier valor con un tamaño hasta un cierto umbral (8.000 bytes por defecto) se puede almacenar dentro de la fila (siempre que pueda caber en la fila). Cualquier valor con un tamaño por encima del umbral se almacena de manera externa a la fila como un objeto grande (LOB).

COTEJO

Es una propiedad de cadenas de caracteres que encapsula varios aspectos: soporte de idioma, ordenamiento, sensibilidad de mayúsculas y minúsculas, sensibilidad de acento y más. Para obtener el conjunto de cotejos compatibles y sus descripciones, puede consultar la función de tabla fn_helpcollations de la siguiente manera:

```
SELECT name, description  
FROM sys.fn_helpcollations();
```

Por ejemplo, la lista siguiente explica el cotejo Latin1_General_CI_AS:

- Latin1_General utiliza la página de códigos 1252. (Esto admite caracteres en inglés y alemán, así como caracteres utilizados por la mayoría de los países de Europa Occidental.)
- Clasificación de diccionarios: La clasificación y comparación de datos de caracteres se basan en el orden del diccionario (A y a < B y b).

Se utiliza el orden de diccionario porque es el valor predeterminado cuando no se define explícitamente ningún otro orden. Más específicamente, el elemento BIN no aparece explícitamente en el nombre del



cotejo. Si aparecía el elemento BIN, significaría que la clasificación y comparación de los datos de los caracteres se basaba en la representación binaria de los caracteres ($A < B < a < b$).

- CI Los datos no distinguen entre mayúsculas y minúsculas ($a = A$).
- AS Los datos son sensibles al acento ($\grave{a} <> \acute{a}$).

En una implementación local de SQL Server, el cotejo se puede definir en cuatro niveles diferentes: instancia, base de datos, columna y expresión. El nivel efectivo más bajo es el que debe ser usado. En Azure SQL Database, el cotejo se puede definir en los niveles de base de datos, columna y expresión. Hay algunos aspectos especializados del cotejo en las bases de datos contenidas. (Para obtener más información, consulte <https://msdn.microsoft.com/en-GB/library/ff929080.aspx>)

El cotejo de la instancia se elige al momento de instalación. Determina el cotejo de todas las bases de datos del sistema y se utiliza como valor predeterminado para las bases de datos de usuario.

Cuando crea una base de datos de usuario, puede especificar un cotejo para la base de datos mediante la cláusula **COLLATE**. Si no lo hace, el cotejo de la instancia se asume de forma predeterminada.

El cotejo de la base de datos determina el cotejo de los metadatos de los objetos en la base de datos y se utiliza como predeterminada para las columnas de la tabla de usuario. Quiero enfatizar la importancia del hecho de que el cotejo de base de datos determina el cotejo de los metadatos, incluidos los nombres de objeto y columna. Por ejemplo, si el cotejo de la base de datos no distingue entre mayúsculas y minúsculas, no puede crear dos tablas denominadas T1 y t1 dentro del mismo esquema, pero si la intercalación de la base de datos distingue entre mayúsculas y minúsculas, puede hacerlo. Tenga en cuenta, sin embargo, que los aspectos de intercalación de los identificadores de variables y parámetros se determinan por la instancia y no por el cotejo de la base de datos, independientemente de la base de datos a la que esté conectado al declararlos. Por ejemplo, si su instancia tiene un cotejo no sensible a mayúsculas y minúsculas y su base de datos tiene un cotejo con distinción entre mayúsculas y minúsculas, no podrá definir dos variables o parámetros denominados @p y @P en el mismo ámbito. Tal intento resultará en un error diciendo que el nombre de la variable ya ha sido declarado.

Puede especificar explícitamente un cotejo para una columna como parte de su definición mediante la cláusula **COLLATE**. Si no lo hace, la intercalación de base de datos se asume de forma predeterminada.

Por ejemplo, en un entorno case-insensitive, la siguiente consulta hace una comparación case-insensitive:

```
USE TSQLV6ES;
SELECT empid, nombre, apellido
FROM RH.Empleados
WHERE apellido = N'davis';
```

Si queremos que el filtrado sea case-sensitive, podemos convertir la expresión con **COLLATE**:

```
USE TSQLV6ES;
SELECT empid, nombre, apellido
FROM RH.Empleados
WHERE apellido COLLATE Latin1_General_CS_AS = N'davis';
```

CONCATENACIÓN DE CADENAS (OPERADOR + FUNCIÓN CONCAT Y FUNCIÓN CONCAT_WS)

T-SQL proporciona el operador de signo más (+) y las funciones CONCAT y CONCAT_WS para concatenar cadenas.

SQL estándar dicta que una concatenación con un NULL debe producir un valor NULL. Este es el comportamiento predeterminado de T-SQL.



Para tratar un NULL como una cadena vacía -o con mayor precisión, para sustituir un NULL por una cadena vacía- puede utilizar la función **COALESCE**. Esta función acepta una lista de valores de entrada y devuelve la primera que no es NULL.

T-SQL soporta una función llamada **CONCAT** que acepta una lista de entradas para la concatenación y sustituye automáticamente NULLs con cadenas vacías. Por ejemplo, la expresión **CONCAT ('a', NULL, 'b')** devuelve la cadena 'ab'.

T-SQL también soporta una función llamada **CONCAT_WS**, que acepta como primer parámetro un separador, por eso el agregado de _WS en el nombre (With Separator), y el resto de los parámetros se concatenan utilizando el separador indicado en el primer parámetro. Con respecto a los valores NULL, se comporta igual que **CONCAT**.

```
USE TSQLV6ES;
SELECT clienteid, pais, region, ciudad,
CONCAT_WS(N',', pais, region, ciudad) AS ubicacion
FROM Ventas.Clientes;
```



FUNCIONES DE CADENA

FUNCIÓN ASCII

Devuelve el valor ASCII¹¹ del carácter que se le pase como parámetro. Si se le pasa una cadena, devuelve el valor ASCII del primer carácter.

ASCII(cadena)

```
SELECT ASCII('A')
```

```
-----
```

```
65
```

FUNCIÓN CHAR

Convierte el entero que recibe como parámetro en el carácter correspondiente a su valor ASCII.

CHAR(entero)

```
SELECT CHAR(65)
```

```
----
```

```
A
```

FUNCIÓN CHARINDEX

Devuelve la posición de la primera ocurrencia de una subcadena dentro de una cadena. La sintaxis es:

CHARINDEX(subcadena, cadena[, pos_desde])

Si no se especifica pos_desde busca desde el primer carácter. Si no encuentra la subcadena devuelve 0.

```
SELECT CHARINDEX(' ', 'Instituto Lujan Buen Viaje');
```

```
-----
```

```
10
```

FUNCIÓN CONCAT

Esta función devuelve una cadena resultante de la concatenación, o unión, de dos o más valores de cadena de un extremo a otro.

CONCAT(cadena1, cadena2 [, cadena])

```
SELECT CONCAT('Instituto', 'Lujan')
```

```
-----
```

```
InstitutoLujan
```

¹¹ ASCII (acrónimo inglés de American Standard Code for Information Interchange —Código Estándar estadounidense para el Intercambio de Información—), es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno



FUNCIÓN CONCAT_WS

Esta función devuelve una cadena resultante de la concatenación, unión, de dos o más valores de cadena de un extremo a otro. Es similar a la función CONCAT, pero tiene un primer parámetro que permite indicar una expresión para utilizar como separador.

CONCAT_WS(separador, cadena1, cadena2 [, cadena])

```
SELECT CONCAT_WS(' - ', 'Instituto', 'Lujan')
```

```
-----  
Instituto - Lujan
```

FUNCIÓN DIFFERENCE

Esta función devuelve un valor entero que mide la diferencia entre los valores [SOUNDEX\(\)](#) de dos expresiones de caracteres diferentes.

Difference compara dos valores SOUNDEX diferentes y devuelve un valor entero. Este valor mide el grado en que coinciden los valores de SOUNDEX, en una escala de 0 a 4. Un valor de 0 indica una similitud débil o nula entre los valores de SOUNDEX; 4 indica valores SOUNDEX muy similares, o incluso idénticos.

Difference y SOUNDEX tienen sensibilidad de clasificación (collation).

Difference (cadena1 , cadena2)

```
SELECT SOUNDEX('Green'), SOUNDEX('Greene'), DIFFERENCE('Green', 'Greene');
```

```
-----  
G650 G650 4
```

FUNCIÓN FORMAT

Devuelve una cadena formateada con el formato y la referencia cultural opcional especificados en SQL Server 2016. Use la función FORMAT para aplicar formato específico de la configuración regional de los valores de fecha/hora y de número como cadenas. Para las conversiones de tipos de datos generales, use CAST o CONVERT.

Más información en <http://go.microsoft.com/fwlink/?LinkId=211776>

FORMAT(entrada , formato_cadena, referencia_cultural)

```
SELECT FORMAT(1759, '000000000');
```

```
-----  
000001759
```



FUNCIONES LEFT Y RIGHT

Estas funciones devuelven la cantidad de caracteres solicitadas desde la izquierda o la derecha de la cadena. La sintaxis es:

LEFT(cadena, n), RIGHT(cadena, n)

```
SELECT LEFT('Instituto', 5)
```

```
-----  
Insti
```

```
SELECT RIGHT('Instituto Lujan', 5)
```

```
-----  
Lujan
```

FUNCIONES LEN Y DATALENGTH

Ambas funciones devuelven la longitud de las cadenas, pero la diferencia está en que LEN devuelve en todas las oportunidades la cantidad de caracteres que forman la cadena, pero DATALENGTH devuelve la cantidad de bytes. En caracteres regulares ambos números coinciden, pero cuando trabajamos con caracteres Unicode (que se almacenan en doble byte) los resultados varían.

LEN (cadena), DATALENGTH (cadena)

```
SELECT LEN(N'abcde');  
SELECT LEN('abcde');
```

```
-----  
5
```

```
SELECT DATALENGTH('abcde');
```

```
-----  
5
```

```
SELECT DATALENGTH(N'abcde');
```

```
-----  
10
```

Otra diferencia entre ambas funciones es que LEN omite los blancos al final de la cadena.

FUNCIÓN LOWER

Devuelve la cadena de entrada toda en minúsculas.

LOWER(cadena)

```
SELECT LOWER('Instituto Buen Viaje');
```

```
-----  
instituto buen viaje
```



FUNCIÓN LTRIM

Devuelve la cadena de entrada sin los espacios en blanco delanteros.

LTRIM(cadena)

```
SELECT LTRIM(' abc ');
-----
abc
```

En SQL Server 2022 se agregó la opción de eliminar otros caracteres que se establecen en el segundo parámetro de manera opcional.

LTRIM(cadena [,caracteres])

```
SELECT LTRIM('123abc.' , '123.');
-----
abc.
```

FUNCIÓN NCHAR

Devuelve el carácter Unicode con el código entero especificado, según lo definido por el estándar Unicode. Es similar a CHAR pero con Unicode.

Cuando la intercalación de la base de datos no contiene el indicador de carácter suplementario (SC), este es un número entero positivo de 0 a 65535 (de 0 a 0xFFFF). Si se especifica un valor fuera de este rango, se devuelve NULL¹².

Cuando la intercalación de la base de datos admite el indicador SC, este es un número entero positivo de 0 a 1114111 (de 0 a 0x10FFFF). Si se especifica un valor fuera de este rango, se devuelve NULL.

Devuelve:

nchar (1) cuando la intercalación de la base de datos predeterminada no admite caracteres suplementarios.

nvarchar (2) cuando la intercalación de la base de datos predeterminada admite caracteres suplementarios.

Si el parámetro entero se encuentra en el rango 0 - 0xFFFF, solo se devuelve un carácter. Para valores más altos, NCHAR devuelve el par sustituto correspondiente. No cree un par sustituto utilizando NCHAR (<sustituto alto>) + NCHAR (\ <sustituto bajo>). En su lugar, utilice una intercalación de base de datos que admita caracteres suplementarios y luego especifique el punto de código Unicode para el par suplente. El siguiente ejemplo demuestra tanto el método de estilo antiguo para construir un par sustituto como el método preferido para especificar el punto de código Unicode.

NCHAR (entero)

¹² Para más información: <https://docs.microsoft.com/en-us/sql/relational-databases/collations/collation-and-unicode-support>



```
CREATE DATABASE test COLLATE Finnish_Swedish_100_CS_AS_SC;
DECLARE @d NVARCHAR(10) = N'𣇱';

-- Metodo estilo antiguo.
SELECT NCHAR(0xD84C) + NCHAR(0xDD7F);

-- Metodo preferido.
SELECT NCHAR(143743);

-- Metodo preferido alternativo.
SELECT NCHAR(UNICODE(@d));

-----
𣇱
```

FUNCIÓN PATINDEX

Devuelve la posición de la primera ocurrencia de un patrón dentro de una cadena. La sintaxis es:

PATINDEX(patron, cadena)

Si no encuentra el patrón devuelve 0.

```
SELECT PATINDEX('%[0-9]%', 'abcd123efgh');

-----
5
```

FUNCIÓN QUOTENAME

Devuelve una cadena Unicode con los delimitadores agregados para que la cadena de entrada sea un identificador delimitado de SQL Server válido.

QUOTENAME (cadena, delimitador)

Cadena, es una cadena de datos de caracteres Unicode. Está limitado a 128 caracteres. Las entradas de más de 128 caracteres devuelven NULL.

Delimitador, es una cadena de un carácter para usar como delimitador. Puede ser una comilla simple ('), un corchete izquierdo o derecho ([]), un signo de comillas dobles (""), un paréntesis izquierdo o derecho (()), un signo mayor o menor que (> <), una llave izquierda o derecha ({}) o una comilla invertida (`). Devuelve NULL si se proporciona un carácter inaceptable. Si no se especifica el carácter de comillas, se utilizan corchetes.

```
SELECT QUOTENAME('abc[]def');

-----
[abc[]]def
```

FUNCIÓN REPLACE

Reemplaza todas las ocurrencias de una subcadena por otra. La sintaxis es:

REPLACE(cadena, subcadena1, subcadena2)

```
SELECT REPLACE('1-a 2-b', '-', ':');

-----
1:a 2:b
```



FUNCIÓN REPLICATE

Replica una cadena el número de veces solicitado. La sintaxis es:

REPLICATE(cadena, n)

```
SELECT REPLICATE('abc', 3);  
-----  
abcabcabc
```

FUNCIÓN REVERSE

Devuelve una cadena en orden invertido.

REVERSE (cadena)

```
SELECT REVERSE(1234) AS Invertido ;  
Invertido  
-----  
4321
```

FUNCIÓN RTRIM

Devuelve la cadena de entrada sin los espacios en blanco traseros.

RTRIM (cadena)

```
SELECT RTRIM(' abc ');  
-----  
abc
```

En SQL Server 2022 se agregó la opción de eliminar otros caracteres que se establecen en el segundo parámetro de manera opcional.

RTRIM (cadena [,caracteres])

```
SELECT RTRIM('.123abc.', 'abc.');
```

.123

FUNCIÓN SOUNDEX

Devuelve un código de cuatro caracteres (SOUNDEX¹³) para evaluar la similitud de dos cadenas.

SOUNDEX (cadena)

SOUNDEX convierte una cadena alfanumérica en un código de cuatro caracteres que se basa en cómo suena la cadena cuando se habla en inglés. El primer carácter del código es el primer carácter de *cadena*, convertido a mayúsculas. Los caracteres del segundo al cuarto del código son números que representan las letras de la expresión. Las letras A, E, I, O, U, H, W e Y se ignoran a menos que sean la primera letra de la cadena. Se agregan ceros al final si es necesario para producir un código de cuatro caracteres.

¹³ Para más información sobre SOUNDEX <https://www.archives.gov/research/census/soundex>



Los códigos SOUNDEX de diferentes cadenas se pueden comparar para ver qué tan similares suenan las cadenas cuando se pronuncian. La función DIFFERENCE realiza un SOUNDEX en dos cadenas y devuelve un número entero que representa qué tan similares son los códigos SOUNDEX para esas cadenas.

SOUNDEX es sensible a la intercalación. Las funciones de cadena se pueden anidar.

```
SELECT SOUNDEX ('Smith'), SOUNDEX ('Smythe');  
-----  
S530  S530
```

FUNCIÓN SPACE

Devuelve una cadena formada por tantos espacios como especifica el parámetro.

SPACE (entero)

```
SELECT SPACE(5)  
-----
```

FUNCIÓN STR

Devuelve datos de cadena convertidos a partir de datos numéricos. Los datos de cadena están justificados a la derecha, con una longitud y precisión decimal especificadas en los parámetros.

STR (expresión_numérica [, Longitud [, decimales]])

Expresión_numérica, es una expresión numérica aproximada (float) con punto decimal

Longitud, es la longitud total, incluye punto decimal, signo, dígitos y espacios. El valor por defecto es 10

Decimal, es la cantidad de números a la derecha del punto decimal. Debe ser igual o menor a 16.

```
SELECT STR(123.45, 6, 1);  
-----  
123.5
```

FUNCIÓN STRING_AGG

Concatena los valores de las expresiones de cadena y coloca valores de separador entre ellos. El separador no se agrega al final de la cadena.

*STRING_AGG (expresión , separador) [<clausula_orden>]
<clausula_orden> ::= WITHIN GROUP (ORDER BY <expresión_orden> [ASC | DESC])*

```
SELECT STRING_AGG(nombre, ',') WITHIN GROUP ( ORDER BY nombre ASC) AS csv  
FROM RH.Empleados  
  
CSV  
-----  
Don,Judy,Maria,Patricia,Paul,Russell,Sara,Sven,Yael
```



FUNCIÓN STRING_ESCAPE

Escapa caracteres especiales en textos y devuelve texto con caracteres de escape. STRING_ESCAPE es una función determinista, introducida en SQL Server 2016.

STRING_ESCAPE(texto , tipo)

```
SELECT STRING_ESCAPE('\' /  
\\ " ', 'json') AS textoEscapado;
```

```
SELECT STRING_ESCAPE('\' → /' → '\" → ', 'json') AS textoEscapado;
```

textoEscapado

```
\\t\\n\\\\\\t\"t
```

FUNCIÓN STRING_SPLIT

Incorporada en SQL Server 2016. Es una función de tabla (como resultado devuelve una tabla), que separa los valores de una cadena, según el separador especificado, y devuelve una fila por cada valor.

En SQL Server 2022 se agregó la opción que admite un tercer argumento opcional que deshabilita o habilita la columna de salida ordinal. Si se omite o tiene valor NULL o cero, se deshabilita la columna ordinal. La columna ordinal contiene los valores de índice, comenzando por 1, de la posición de cada cadena en la cadena de entrada.

SELECT value FROM STRING_SPLIT(cadena, separador [,enable_ordinal]);

```
SELECT CAST(value AS INT) AS misvalores  
FROM STRING_SPLIT('10248,10249,10250', ',') AS S;  
  
misvalores  
-----  
10248  
10249  
10250  
  
SELECT value, ordinal  
FROM STRING_SPLIT('10248,10249,10250', ',', 1) AS S;  
  
value          ordinal  
-----  
10248          1  
10249          2  
10250          3
```

FUNCIÓN STUFF

Remueve una subcadena de una cadena e inserta una nueva subcadena en su lugar.

STUFF(cadena, pos, longitud_a_borrar, cadena_a_insertar)

```
SELECT STUFF('xyz', 2, 1, 'abc');  
-----  
xabcz
```



FUNCIÓN SUBSTRING

Extrae una subcadena de una cadena. La sintaxis es:

SUBSTRING(cadena, comienzo, Longitud)

```
SELECT SUBSTRING('abcdefg', 2, 3);  
----  
bcd
```

FUNCIÓN TRANSLATE

Devuelve la cadena proporcionada como primer argumento después de que algunos caracteres especificados en el segundo argumento se traduzcan a un conjunto de caracteres de destino especificado en el tercer argumento.

TRANSLATE (cadena, caracteres, traducciones)

```
SELECT TRANSLATE('2*[3+4]/{7-2}', '[]{}', '()());  
-----  
2*(3+4)/(7-2)
```

FUNCIÓN TRIM

Elimina el carácter de espacio char (32) u otros caracteres especificados del principio y final de una cadena. Se incorporó en SQL Server 2017.

TRIM ([caracteres FROM] cadena)

```
SELECT TRIM(' abc ')  
----  
abc  
  
SELECT TRIM( '.,! ' FROM '      #      prueba      .') AS Resultado;  
Resultado  
-----  
#      prueba
```

En SQL Server 2022 se modificó la sintaxis agregando las opciones LEADING, TRAILING y BOTH.

TRIM ([LEADING | TRAILING | BOTH] [caracteres FROM] cadena)

```
SELECT TRIM(LEADING '.,! ' FROM '      .#      prueba      .') AS Resultado;  
Resultado  
-----  
#      prueba      .  
SELECT TRIM(TRAILING '.,! ' FROM '      .#      prueba      .') AS Resultado;  
Resultado  
-----  
.#      prueba
```



FUNCIÓN UNICODE

Devuelve el valor entero, tal como lo define el estándar Unicode, para el primer carácter de la expresión de entrada.

UNICODE (cadena)

```
SELECT UNICODE('N'A')
```

```
-----  
65
```

FUNCIÓN UPPER

Devuelve la cadena de entrada toda en mayúsculas.

UPPER(cadena)

```
SELECT UPPER('Instituto Buen Viaje');
```

```
-----  
INSTITUTO BUEN VIAJE
```

PREDICADO LIKE

El predicado LIKE se utiliza para verificar cuando una cadena de texto coincide con un determinado patrón. Los patrones se forman utilizando caracteres comodines.

% (porcentaje): representa una cadena de cualquier tamaño, incluyendo la cadena vacía.

_ (guion bajo): representa un único carácter

[lista de caracteres]: le puede definir una lista de caracteres entre corchetes, que representan a un único carácter.

[carácter_desde-carácter_hasta]: representa un único carácter, comprendido en el rango establecido entre corchetes.

[^carácter_desde-carácter_hasta]: representa un único carácter, que no esté en el rango establecido. Es la negación de la expresión anterior.

Carácter de escape: si en alguna expresión se quiere utilizar un carácter que puede ser utilizado como comodín (_, %, [,]), se lo antecede con el carácter de escape, para indicar que no está funcionando como comodín, si no como un carácter normal. Se lo declara luego de la expresión, por ejemplo, col1 LIKE '%!_%' ESCAPE '!'.

Los comodines % y _ se pueden escapar también colocándolos entre corchetes, por ejemplo, col1 LIKE '%[_]%'.



TRABAJANDO CON DATOS DE FECHA Y HORA

Trabajar con datos de fecha y hora en SQL Server no es trivial. Hay muchos desafíos en esta área, como expresar literales de una manera neutral al lenguaje y trabajar con la fecha y la hora por separado. Veremos primero los tipos de datos de fecha y hora compatibles con SQL Server, luego veremos la forma recomendada de trabajar con esos tipos. Y finalmente las funciones de fecha y tiempo.

TIPOS DE DATOS DE FECHA Y HORA

T-SQL admite seis tipos de datos de fecha y hora: dos tipos legados llamados DATETIME y SMALLDATETIME y cuatro adicionales posteriores (desde SQL Server 2008) denominados DATE, TIME, DATETIME2 y DATETIMEOFFSET. Los tipos legados DATETIME y SMALLDATETIME incluyen componentes de fecha y hora que son inseparables. Los dos tipos difieren en sus requisitos de almacenamiento, su rango de fechas compatible y su precisión. Los tipos de datos DATE y TIME proporcionan una separación entre los componentes de fecha y hora si es necesario. El tipo de datos DATETIME2 tiene un rango de fechas más grande y una mejor precisión que los tipos legados. El tipo de datos DATETIMEOFFSET es similar a DATETIME2, pero también incluye el desplazamiento de UTC.

Tipo de Datos	Almac. (bytes)	Rango de Fechas	Precisión	Formato de entrada recomendado y ejemplo
DATETIME	8	1 de enero de 1753 hasta 31 de diciembre de 9999	3 1/3 milisegundos	'AAAA-MM-DD hh:mm:ss.nnn' '20090212 12:30:15.123'
SMALLDATETIME	4	1 de enero de 1900 hasta 6 de junio de 2079	1 minuto	'AAAA-MM-DD hh:mm' '20090212 12:30'
DATE	3	1 de enero de 0001 hasta 31 de diciembre de 9999	1 día	'AAAA-MM-DD' '2009-02-12'
TIME	3 a 5	N/D	100 nanosegundos	'hh:mm:ss.nnnnnnnn' '12:30:15.1234567'
DATETIME2	6 a 8	1 de enero de 0001 hasta 31 de diciembre de 9999	100 nanosegundos	'AAAA-MM-DD hh:mm:ss.nnnnnnnn' '2009-02-12 12:30:15.1234567'
DATETIMEOFFSET	8 a 10	1 de enero de 0001 hasta 31 de diciembre de 9999	100 nanosegundos	'AAAA-MM-DD hh:mm:ss.nnnnnnnn [+/-]hh:mm' '2009-02-12 12:30:15.1234567 +02:00'

Los requisitos de almacenamiento para los tres últimos tipos de datos en la Tabla (TIME, DATETIME2 y DATETIMEOFFSET) dependen de la precisión que elija. Especifique una precisión de fracción de segundo como un entero en el rango de 0 a 7. Por ejemplo, TIME (0) significa precisión de 0 segundos fraccionarios, es decir, precisión de un segundo. TIME (3) significa precisión de un milisegundo y TIME (7) significa precisión de 100 nanosegundos. Si no especifica una precisión de segundo fraccionario, SQL Server asume 7 de forma predeterminada. Cuando se convierte un valor en un tipo de datos con una precisión inferior, se redondea al valor expresable más cercano en la precisión de destino.

LITERALES

Cuando necesite especificar un literal (constante) de un tipo de datos de fecha y hora, debe considerar varias cosas. En primer lugar, aunque podría sonar un poco extraño, T-SQL no proporciona los medios para expresar una fecha y hora literal; En su lugar, puede especificar un literal de un tipo diferente que se puede convertir, explícita o implícitamente, en un tipo de datos de fecha y hora. Es una práctica recomendada utilizar cadenas de caracteres para expresar valores de fecha y hora, como se muestra en el ejemplo siguiente:



```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE fechaorden = '20220212';
```

SQL Server reconoce el literal '20220212' como un literal de cadena de caracteres y no como una fecha y hora literal, pero debido a que la expresión implica operandos de dos tipos diferentes, un operando necesita convertirse implícitamente al tipo del otro. Normalmente, la conversión implícita entre tipos se basa en lo que se llama precedencia de tipo de datos. SQL Server define la precedencia entre los tipos de datos y suele convertir implícitamente el operando que tiene una prioridad de tipo de datos inferior a la que tiene mayor precedencia. En este ejemplo, el literal de cadena de caracteres se convierte al tipo de datos de la columna (DATETIME) porque las cadenas de caracteres se consideran inferiores en términos de precedencia de tipo de datos con respecto a los tipos de datos de fecha y hora. Para obtener la descripción completa de la precedencia de tipo de datos, vea "Precedencia de tipo de datos" en los Libros en pantalla de SQL Server.

El punto que estoy tratando de hacer es que, en el ejemplo anterior, la conversión implícita tiene lugar detrás de las escenas. La conversión se puede hacer de manera explícita, como podemos ver en el siguiente ejemplo:

```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE fechaorden = CAST('20220212' AS DATE);
```

Esta consulta es equivalente a la anterior, con la diferencia que la primera hace la conversión de manera implícita y esta lo hace de manera explícita.

Tenga en cuenta que algunos formatos de cadena de caracteres de literales de fecha y hora son dependientes del idioma, lo que significa que, al convertirlos a un tipo de datos de fecha y hora, SQL Server podría interpretar el valor de forma diferente en función de la configuración de idioma en vigor en la sesión. Cada inicio de sesión definido por el administrador de la base de datos tiene un lenguaje predeterminado asociado a él y, a menos que se cambie explícitamente, ese idioma se convierte en el idioma efectivo en la sesión. Puede sobrescribir el idioma predeterminado en su sesión mediante el comando SET LANGUAGE, pero generalmente no se recomienda porque algunos aspectos del código pueden basarse en el idioma predeterminado del usuario.

El lenguaje efectivo en la sesión establece varios ajustes relacionados con el idioma detrás de escena. Entre ellos se encuentra uno llamado DATEFORMAT, que determina cómo SQL Server interpreta los literales que ingresa cuando se convierten de un tipo de cadena de caracteres a un tipo de fecha y hora. El ajuste DATEFORMAT se expresa como una combinación de los caracteres d, m, yy. Por ejemplo, la configuración de idioma us_english establece el DATEFORMAT en mdy, mientras que el idioma inglés británico establece el DATEFORMAT en dmy. Puede anular la configuración DATEFORMAT en su sesión mediante el comando SET DATEFORMAT, pero como se mencionó antes, generalmente no se recomienda cambiar la configuración relacionada con el idioma.

Consideremos, por ejemplo, el literal '02/12/2022'. SQL Server puede interpretar la fecha como 12 de febrero de 2022 o 2 de diciembre de 2022 cuando convierta este literal a uno de los siguientes tipos: DATETIME, SMALLDATETIME, DATE, DATETIME2 o DATETIMEOFFSET. El parámetro LANGUAGE / DATEFORMAT efectivo es el factor determinante. Para demostrar diferentes interpretaciones de la misma cadena de caracteres literal, ejecute el siguiente código:



```
SET LANGUAGE British;
SELECT CAST('02/12/2022' AS DATE);
SET LANGUAGE us_english;
SELECT CAST('02/12/2022' AS DATE);
```

Changed language setting to British.

2022-12-02

Changed language setting to us_english.

2022-02-12

Tenga en cuenta que el ajuste LANGUAGE / DATEFORMAT sólo afecta a la forma en que se interpretan los valores introducidos. Estos ajustes no tienen impacto en el formato utilizado en la salida para propósitos de presentación. El formato de salida se determina por la interfaz de base de datos utilizada por la herramienta de cliente (como ODBC) y no por la configuración LANGUAGE / DATEFORMAT. Por ejemplo, OLE DB y ODBC presentan valores de fecha en el formato 'AAAA-MM-DD'.

Debido a que el código que escribe puede ser utilizado por usuarios internacionales con diferentes configuraciones de idioma para sus inicios de sesión, la comprensión de que algunos formatos de literales dependen del idioma es crucial. Es muy recomendable que exprese sus literales de una manera neutral al lenguaje. Los formatos de lenguaje neutro siempre son interpretados por SQL Server de la misma manera y no se ven afectados por la configuración relacionada con el idioma.

En la siguiente tabla podemos ver los literales neutrales para cada tipo:

Tipo de Dato	Precisión	Formato recomendado de entrada y ejemplo
DATETIME	'AAAMMDD hh:mm:ss.nnn' 'AAAA-MM-DDThh:mm:ss.nnn' 'AAAAMMDD'	'20220212 12:30:15.123' '2022-02-12T12:30:15.123' '20220212'
SMALLDATETIME	'AAAAMMDD hh:mm' 'AAAA-MM-DDThh:mm' 'AAAAMMDD'	'20220212 12:30' '2022-02-12T12:30' '20220212'
DATE	'AAAAMMDD' 'AAAA-MM-DD'	'20220212' '2022-02-12'
DATETIME2	'AAAMMDD hh:mm:ss.nnnnnnnn' 'AAAA-MM-DD hh:mm:ss.nnnnnnnn' 'AAAA-MM-DDThh:mm:ss.nnnnnnnn' 'AAAAMMDD' 'AAAA-MM-DD'	'20220212 12:30:15.1234567' '2022-02-12 12:30:15.1234567' '2022-02-12T12:30:15.1234567' '20220212' '2022-02-12'
DATETIMEOFFSET	'AAAMMDD hh:mm:ss.nnnnnnnn [+ -]hh:mm' 'AAAA-MM-DD hh:mm:ss.nnnnnnnn [+ -]hh:mm' 'AAAAMMDD' 'AAAA-MM-DD'	'20220212 12:30:15.1234567 +02:00' '2022-02-12 12:30:15.1234567 +02:00' '20220212' '2022-02-12'
TIME	'hh:mm:ss.nnnnnnnn'	'12:30:15.1234567'

Los tipos que incluyen fecha y hora, cuando no se establece la hora en el literal, SQL Server asume como hora la medianoche. Si no se establece la diferencia con UTC, se asume 00:00.

Los literales 'AAAA-MM-DD' y 'AAAA-MM-DD hh:mm...' son dependientes del lenguaje cuando son convertidos a DATETIME y SMALLDATETIME, pero son neutrales cuando son convertidos a DATE, DATETIME2 y DATETIMEOFFSET.

Por lo tanto, es recomendable usar la forma 'AAAAMMDD', ya que siempre es neutral.



Si a pesar de las recomendaciones, se quiere usar un formato dependiente del idioma para expresar literales, hay dos opciones disponibles. Una es usar la función CONVERT para convertir explícitamente el literal de cadena de caracteres al tipo de datos solicitado y, en el tercer argumento, especificar un número que represente el estilo que usó ([ver anexo](#)). Por ejemplo, si desea especificar el literal '02/12/2022' con el formato MM/DD/AAAA, utilice el número de estilo 101:

```
SELECT CONVERT(DATE, '02/12/2022', 101);
```

```
-----  
2022-02-12
```

El literal se interpreta como 12 de febrero 2022, independientemente de la configuración de idioma que está en vigente.

Si desea utilizar el formato DD/MM/AAAA, utilice el número de estilo 103:

```
SELECT CONVERT(DATE, '02/12/2022', 103);
```

```
-----  
2022-12-02
```

Esta vez, el literal se interpreta como el 2 de diciembre de 2022.

Otra opción es utilizar la función PARSE. Al usar esta función, puede analizar un valor como un tipo solicitado e indicar la cultura. Por ejemplo, lo siguiente es el equivalente de usar CONVERT con estilo 101 (inglés de EE. UU.):

```
SELECT PARSE('02/12/2022' AS DATE USING 'en-US');
```

```
-----  
2022-02-12
```

Y el siguiente es equivalente a usar CONVERT con el estilo 103 (inglés británico):

```
SELECT PARSE('02/12/2022' AS DATE USING 'en-GB');
```

```
-----  
2022-12-02
```

TRABAJANDO CON FECHA Y HORA POR SEPARADO

Si necesitamos trabajar sólo con fechas o sólo horas, se recomienda utilizar los tipos de datos DATE y TIME, respectivamente. Seguir este consejo puede convertirse en un desafío si necesita restringirse a usar sólo los tipos heredados DATETIME y SMALLDATETIME por razones tales como la compatibilidad con sistemas antiguos. El reto es que los tipos heredados contienen los componentes de fecha y hora. La mejor práctica en tal caso dice que cuando se quiere trabajar sólo con fechas, se almacena la fecha con un valor de medianoche en la parte de tiempo. Cuando desea trabajar sólo con tiempos, almacena la hora con la fecha base 1 de enero de 1900.

Para demostrar el trabajo con fecha y hora por separado, utilizaré una tabla llamada Ventas.Ordenes2, que tiene una columna llamada fechaorden de un tipo de datos DATETIME. Ejecute el código siguiente para crear la tabla Ventas.Ordenes2 copiando datos de la tabla Ventas.Ordenes y casteando la columna fechaorden de origen, que es de un tipo DATE, a DATETIME:



```
USE TSQLV6ES;
DROP TABLE IF EXISTS Ventas.Ordenes2;
SELECT ordenid, clienteid, empid, CAST(fechaorden AS DATETIME) AS fechaorden
INTO Ventas.Ordenes2
FROM Ventas.Ordenes;
```

Como se mencionó, la columna fechaorden en la tabla Ventas.Ordenes2 es de un tipo de datos DATETIME, pero como sólo el componente fecha es realmente relevante, todos los valores contienen la medianoche como hora. Cuando necesita filtrar sólo pedidos de una fecha determinada, no tiene que utilizar un filtro de rango. En su lugar, puede utilizar el operador de igualdad de la siguiente manera:

```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes2
WHERE fechaorden = '20220212';
```

Cuando SQL Server convierte un literal de cadena de caracteres que sólo tiene una fecha a DATETIME, asume por default de hora la medianoche. Dado que todos los valores de la columna de fecha de pedido contienen la medianoche en el componente de tiempo, se devolverán todos los pedidos realizados en la fecha solicitada.

Si la parte de horas tuviera un valor distinto a la medianoche, la consulta se puede hacer filtrando por rangos:

```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes2
WHERE fechaorden >= '20220212'
AND fechaorden < '20220213';
```

Si desea trabajar sólo con horas utilizando los tipos legados, puede almacenar todos los valores con la fecha base del 1 de enero de 1900. Cuando SQL Server convierte un literal de cadena de caracteres que contiene sólo un componente de tiempo a DATETIME o SMALLDATETIME, SQL Server Supone que la fecha es la fecha base. Por ejemplo, ejecute el código siguiente:

```
SELECT CAST('12:30:15.123' AS DATETIME);
```

```
-----  
1900-01-01 12:30:15.123
```

FILTRANDO RANGOS DE FECHAS

Cuando se necesita filtrar un rango de fechas, como un año entero o un mes completo, parece natural utilizar funciones como YEAR y MONTH. Por ejemplo, la siguiente consulta devuelve todos los pedidos realizados en el año 2021:

```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE YEAR(fechaorden) = 2021;
```

Sin embargo, debemos tener en cuenta que, en la mayoría de los casos, cuando se aplica la manipulación en la columna filtrada, SQL Server no puede utilizar un índice de manera eficiente. Esto es probablemente difícil de entender sin conocimientos previos sobre los índices y el rendimiento, que aún no hemos visto. Por ahora, basta con tener presente este punto general: Para tener el potencial de usar un índice de manera eficiente, no debe manipular la columna filtrada. Para lograr esto, puede corregir el predicado de filtro de la última consulta se la siguiente manera:



```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE fechaorden >= '20210101' AND fechaorden < '20220101';
```

Del mismo modo, si queremos filtrar las órdenes emitidas en un mes en particular:

```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE YEAR(fechaorden) = 2022 AND MONTH(fechaorden) = 2;
```

Puede escribirse con rangos de la siguiente manera:

```
USE TSQLV6ES;
SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE fechaorden >= '20220201' AND fechaorden < '20220301';
```

FUNCIONES DE FECHA Y HORA

OBTENER LA FECHA Y HORA ACTUAL

Las siguientes funciones niládicas (sin parámetros) devuelven los valores actuales de fecha y hora en el sistema donde reside la instancia de SQL Server: GETDATE, CURRENT_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME y SYSDATETIMEOFFSET.

Función	Tipo Devuelto	Descripción
GETDATE	DATETIME	Fecha y hora actual
CURRENT_TIMESTAMP	DATETIME	Igual a GETDATE pero cumple SQL ANSI
GETUTCDATE	DATETIME	Fecha y hora actual en UTC
SYSDATETIME	DATETIME2	Fecha y hora actual
SYSUTCDATETIME	DATETIME2	Fecha y hora actual en UTC
SYSDATETIMEOFFSET	DATETIMEOFFSET	Fecha y hora actual incluyendo zona

Tenga en cuenta que debe especificar paréntesis vacíos con todas las funciones que se deben especificar sin parámetros, excepto la función estándar CURRENT_TIMESTAMP. Además, debido a que CURRENT_TIMESTAMP y GETDATE devuelven lo mismo, pero sólo el primero es estándar, se recomienda que utilice el anterior.

El siguiente código muestra el uso de las funciones actuales de fecha y hora:

```
SELECT
GETDATE() AS [GETDATE],
CURRENT_TIMESTAMP AS [CURRENT_TIMESTAMP],
GETUTCDATE() AS [GETUTCDATE],
SYSDATETIME() AS [SYSDATETIME],
SYSUTCDATETIME() AS [SYSUTCDATETIME],
SYSDATETIMEOFFSET() AS [SYSDATETIMEOFFSET];
```

Como probablemente habrán notado, ninguna de las funciones devuelve sólo la fecha del sistema actual o sólo la hora actual del sistema. Sin embargo, se pueden obtener fácilmente mediante la conversión de CURRENT_TIMESTAMP o SYSDATETIME a DATE o TIME de esta manera:

```
SELECT
CAST(SYSDATETIME() AS DATE) AS [current_date],
CAST(SYSDATETIME() AS TIME) AS [current_time];
```



FUNCIONES CAST, CONVERT Y PARSE Y SUS CONTRAPARTES TRY_

Las funciones CAST, CONVERT y PARSE se utilizan para convertir un valor de entrada en algún tipo de destino. Si la conversión tiene éxito, las funciones devuelven el valor convertido, de lo contrario, provocan la falla de la consulta. Las tres funciones tienen contrapartidas llamadas TRY_CAST, TRY_CONVERT y TRY_PARSE, respectivamente. Cada versión con el prefijo TRY_ acepta la misma entrada que su contraparte y aplica la misma conversión. La diferencia es que, si la entrada no es convertible al tipo de destino, la función devuelve un NULL en lugar de fallar la consulta. Las funciones TRY_ se incorporaron en SQL Server 2012.

CAST(valor AS tipo de datos)

TRY_CAST(valor AS tipo de datos)

CONVERT (tipo de datos, valor [, estilo de número])

TRY_CONVERT (tipo de datos, valor [, estilo de número])

PARSE (valor AS tipo de datos [USING cultura])

TRY_PARSE (valor AS tipo de datos [USING cultura])

Las tres funciones de base convierten el valor de entrada al tipo de dato de destino especificado. En algunos casos, CONVERT tiene un tercer argumento con el que puede especificar el estilo de la conversión.

Por ejemplo, cuando se convierte de una cadena de caracteres a uno de los tipos de datos de fecha y hora (o al revés), el número de estilo indica el formato de la cadena. Por ejemplo, el estilo 101 indica 'MM/DD/AAAA', y el estilo 103 indica 'DD/MM/AAAA'. Puede encontrar la lista completa de números de estilo y sus significados en los Libros en pantalla de SQL Server en "CAST y CONVERT". De forma similar, cuando corresponda, la función PARSE admite la indicación de una cultura (por ejemplo, 'en-US' Y 'en-GB' para el inglés británico).

Como se mencionó anteriormente, cuando se está convirtiendo de una cadena de caracteres a uno de los tipos de datos de fecha y hora, algunos de los formatos de cadena son dependientes del idioma. Se recomienda utilizar uno de los formatos de idioma neutro o utilizar la función CONVERT y especificando explícitamente el número de estilo. De esta manera, su código se interpreta de la misma manera independientemente del idioma del login que lo ejecuta.

Tenga en cuenta que CAST es estándar y CONVERT y PARSE no lo son, por lo que a menos que necesite utilizar el número de estilo o cultura, se recomienda utilizar la función CAST.

FUNCIÓN DATEPART

Devuelve un entero que representa la unidad deseada de la fecha

DATEPART(unid, fecha_val)

Los valores válidos para el argumento *unid* incluyen year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond, nanosecond, TZoffset, e ISO_WEEK. Como se mencionó, puede utilizar abreviaturas para las partes de fecha y hora, como yy en lugar de year, mm en lugar de month, dd en lugar de day y así sucesivamente.

```
SELECT DATEPART(month, '20220212');
```

2



FUNCIONES YEAR, MONTH Y DAY

Las funciones YEAR, MONTH y DAY son abreviaturas para la función DATEPART que devuelve la representación entera de las partes del año, mes y día de un valor de fecha y hora de entrada.

YEAR(fecha_val)

MONTH(fecha_val)

DAY(fecha_val)

```
SELECT  
DAY('20220212') AS eldia,  
MONTH('20220212') AS elmes,  
YEAR('20220212') AS elanio;
```

eldia	elmes	elianio
12	2	2022

FUNCIÓN DATENAME

La función DATENAME devuelve una cadena de caracteres que representa un valor en una unidad determinada de un valor de fecha y hora.

DATENAME(unid, fecha_val)

Esta función es similar a DATEPART y, de hecho, tiene las mismas opciones para la entrada de unid. Sin embargo, cuando es relevante, devuelve el nombre de la parte solicitada en lugar del número.

```
SELECT DATENAME(month, '20220212');
```

February

Recordemos que DATEPART devolvió el entero 2 para esta entrada. DATENAME devuelve el nombre del mes, que depende del idioma. Si el idioma de la sesión es uno de los idiomas en inglés (como el inglés de los Estados Unidos o inglés británico), se recupera el valor 'February'. Si el idioma de la sesión es el italiano, se recupera el valor 'febbraio'. Si se solicita una parte que no tiene nombre y sólo un valor numérico (como año), la función DATENAME devuelve su valor numérico como una cadena de caracteres.

```
SELECT DATENAME(year, '20220212');
```

2022



FUNCIONES FROMPARTS

Las funciones FROMPARTS aceptan entradas enteras que representan partes de un valor de fecha y hora y construyen un valor del tipo solicitado desde esas partes.

DATEFROMPARTS (year, month, day)

DATETIME2FROMPARTS (year, month, day, hour, minute, seconds, fractions, precision)

DATETIMEFROMPARTS (year, month, day, hour, minute, seconds, milliseconds)

DATETIMEOFFSETFROMPARTS (year, month, day, hour, minute, seconds, fractions, hour_offset, minute_offset, precision)

SMALLDATETIMEFROMPARTS (year, month, day, hour, minute)

TIMEFROMPARTS (hour, minute, seconds, fractions, precision)

Estas funciones facilitan a los programadores la construcción de valores de fecha y hora de los diferentes componentes y también simplifican la migración de código de otros entornos que ya soportan funciones similares.

SELECT

```
DATEFROMPARTS(2022, 02, 12),
DATETIME2FROMPARTS(2022, 02, 12, 13, 30, 5, 1, 7),
DATETIMEFROMPARTS(2022, 02, 12, 13, 30, 5, 997),
DATETIMEOFFSETFROMPARTS(2022, 02, 12, 13, 30, 5, 1, -8, 0, 7),
SMALLDATETIMEFROMPARTS(2022, 02, 12, 13, 30),
TIMEFROMPARTS(13, 30, 5, 1, 7);
```

```
-----  
2022-02-12 2022-02-12 13:30:05.0000001 2022-02-12 13:30:05.997 2022-02-12 13:30:05.0000001 -08:00 2022-02-12 13:30:00 13:30:05.0000001
```

FUNCIÓN SWITCHOFFSET

Ajusta una fecha y hora de tipo DATETIMEOFFSET a una zona horaria particular. La sintaxis es:

SWITCHOFFSET(valor_datetimeoffset, UTC_offset)

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '-05:00');
```

```
-----  
2021-05-08 07:36:20.4133924 -05:00
```

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '+00:00');
```

```
-----  
2021-05-08 12:36:20.4133924 +00:00
```

FUNCIÓN TODATETIMEOFFSET

Para setear la zona horaria a una fecha y hora. La sintaxis es:

TODATETIMEOFFSET(valor_fecha_y_hora_Local, UTC_offset)

Esta función se diferencia de SWITCHOFFSET en que su primera entrada es un valor local de fecha y hora sin un componente de desplazamiento. Esta función simplemente fusiona el valor de fecha y hora de entrada con el desplazamiento especificado para crear un nuevo valor de tiempo de salida de fecha.



Por lo general, utilizará esta función al migrar datos no compatibles con offset para datos con detección de offset. Imagine que tiene una tabla que contiene valores de fecha y hora locales en un atributo llamado dt de un tipo de datos DATETIME2 o DATETIME y que mantiene el desplazamiento en un atributo denominado offset.

A continuación, decide combinar los dos a un atributo que reconoce offset denominado dto. Cambia la tabla y agrega el nuevo atributo. A continuación, actualiza el resultado de la expresión TODATETIMEOFFSET(dt, theoffset). A continuación, puede eliminar los dos atributos existentes dt y theoffset.

FUNCTION AT TIME ZONE

Acepta un valor de fecha y hora de entrada y lo convierte en un valor de hora de salida que corresponde a la zona horaria objetivo especificada. Esta función se introdujo en SQL Server 2016. La sintaxis es la siguiente:

fecha_val AT TIME ZONE zona_horaria

La entrada fecha_val puede ser de los siguientes tipos de datos: DATETIME, SMALLDATETIME, DATETIME2 y DATETIMEOFFSET. La entrada zona_horaria puede ser cualquiera de los nombres de zona horaria de Windows admitidos, tal como aparecen en la columna de nombre en la vista sys.time_zone_info.

Utilice la consulta siguiente para ver las zonas horarias disponibles, su desplazamiento actual de UTC y si es actualmente hora de verano (DST):

```
SELECT name, current_utc_offset, is_currently_dst
FROM sys.time_zone_info;
```

Respecto a fecha_val: cuando se utiliza cualquiera de los tres tipos no-datetimeoffset (DATETIME, SMALLDATETIME y DATETIME2), la función AT TIME ZONE asume que el valor de entrada ya está en la zona horaria objetivo. Como resultado, se comporta de forma similar a la función TODATETIMEOFFSET, excepto que el desplazamiento no es necesariamente fijo. Depende de si se aplica DST. Tome como ejemplo la zona horaria de la hora estándar del Pacífico. Cuando no es DST, la diferencia de UTC es -08: 00; Cuando es DST, el desplazamiento es -07: 00. El código siguiente demuestra el uso de esta función con las entradas no-datetimeoffset:

```
SELECT
CAST('20220212 12:00:00.000000' AS DATETIME2)
AT TIME ZONE 'Pacific Standard Time' AS val1,
CAST('20220812 12:00:00.000000' AS DATETIME2)
AT TIME ZONE 'Pacific Standard Time' AS val2;

val1                         val2
-----
2022-02-12 12:00:00.000000 -08:00 2022-08-12 12:00:00.000000 -07:00
```

El primer valor ocurre cuando DST no se aplica; Por lo tanto, el desplazamiento -08:00 se asume. El segundo valor ocurre durante el horario de verano; Por lo tanto, se supone offset -07:00. Aquí no hay ambigüedad.

Hay dos casos difíciles: al cambiar a DST y al cambiar de DST. Por ejemplo, en Pacific Standard Time, al cambiar a DST el reloj se avanza por una hora, por lo que hay una hora que no existe. Si especifica un tiempo no existente durante esa hora, se supondrá el desplazamiento antes del cambio (-08: 00). Al cambiar de DST, el reloj retrocede por una hora, así que hay una hora que se repite. Si se especifica una hora durante esa hora, comenzando en la parte inferior de la hora, se supone que no es DST (es decir, se utiliza el desplazamiento -08: 00).

Cuando la entrada fecha_val es un valor datetimeoffset, la función AT TIME ZONE se comporta de forma más similar a la función SWITCHOFFSET. De nuevo, sin embargo, el desplazamiento de destino no es necesariamente fijo. T-SQL utiliza las reglas de conversión de zona horaria de Windows para aplicar la conversión. El código siguiente demuestra el uso de la función con entradas datetimeoffset:



```
SELECT  
CAST('20220212 12:00:00.000000 -05:00' AS DATETIMEOFFSET)  
AT TIME ZONE 'Pacific Standard Time' AS val1,  
CAST('20220812 12:00:00.000000 -04:00' AS DATETIMEOFFSET)  
AT TIME ZONE 'Pacific Standard Time' AS val2;
```

Los valores de entrada reflejan la zona horaria Hora estándar del este. Ambos tienen el mediodía en el componente de hora. El primer valor ocurre cuando DST no se aplica (offset es -05:00), y el segundo ocurre cuando DST se aplica (es decir, el desplazamiento es -04:00). Ambos valores se convierten a la zona horaria Hora estándar del Pacífico (la compensación -08:00 cuando DST no se aplica y -07:00 cuando lo hace). En ambos casos, el tiempo necesita retroceder en tres horas a las 9:00 AM. Obtendrá la salida siguiente:

val1	val2
2022-02-12 09:00:00.000000 -08:00	2022-08-12 09:00:00.000000 -07:00

FUNCIÓN DATEADD

Permite agregar una cantidad específica de unidades (especificadas en el primer parámetro) a una fecha y hora

DATEADD(unid, n, fecha_val)

Los valores válidos para *unid* incluyen year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond, y nanosecond. También puede especificar *unid* en forma abreviada, como yy en lugar de year. Consulte los Libros en pantalla de SQL Server para obtener más información.

El tipo devuelto para un parámetro de fecha y hora es del mismo tipo que el tipo del parámetro de entrada. Si a esta función se le da una cadena literal como entrada, la salida es DATETIME.

```
SELECT DATEADD(year, 1, '20220212');  
-----  
2023-02-12 00:00:00.000
```

FUNCIONES DATEDIFF Y DATEDIFF_BIG

Devuelven la diferencia entre dos valores de fecha y hora en términos de una unidad de fecha especificada. El primero devuelve un valor tipificado como INT (un entero de 4 bytes), y el segundo devuelve un valor escrito como BIGINT (un entero de 8 bytes). La función DATEDIFF_BIG se introdujo en SQL Server 2016.

DATEDIFF(unid, fecha_val1, fecha_val2)

DATEDIFF_BIG(unid, fecha_val1, fecha_val2)

```
SELECT DATEDIFF(day, '20210212', '20220212');  
-----  
365  
  
SELECT DATEDIFF_BIG(millisecond, '00010101', '20220212');  
-----  
6359083200000
```

Si necesitamos calcular el comienzo del día que corresponde a un valor de fecha y hora de entrada, simplemente podemos castear el valor de entrada al tipo DATE y luego mostrar el resultado al tipo de destino. Pero con un uso un poco más sofisticado de las funciones DATEADD y DATEDIFF, podemos calcular el comienzo o el final de diferentes unidades (día, mes, trimestre, año) que corresponden al valor de entrada.



```
SELECT  
DATEADD(  
day,  
DATEDIFF(day, '19000101', SYSDATETIME()), '19000101');
```

Esto se logra utilizando primero la función DATEDIFF para calcular la diferencia en términos de días enteros entre una fecha de anclaje a medianoche ('19000101' en este caso) y la fecha y hora actuales (llamar a esa diferencia diff). A continuación, la función DATEADD se utiliza para añadir días diff al ancla. Y así se obtiene la fecha actual del sistema a la medianoche.

A partir de SQL Server 2012 podemos utilizar DATEFROMPARTS. Componiendo la fecha con el año, mes y día.

```
SELECT DATEFROMPARTS(YEAR(SYSDATETIME()), MONTH(SYSDATETIME()), DAY(SYSDATETIME()));
```

Y desde SQL Server 2022 podemos utilizar la función DATE_BUCKET.

FUNCIÓN DATE_BUCKET

DATE_BUCKET es una función agregada en SQL Server 2022 que colapsa una fecha/hora a un intervalo fijo, eliminando la necesidad de redondear valores de fecha y hora, extraer partes de fecha, realizar conversiones salvajes hacia y desde otros tipos o realizar cálculos elaborados y poco intuitivos de DATEADD/DATEDIFF utilizando fechas mágicas del pasado.

La salida es un tipo de fecha/hora (basado en la entrada), pero en un intervalo gobernado por el datepart y el bucket_width.

DATE_BUCKET (datepart, bucket_width, fecha_entrada [, origen])

```
SELECT DATE_BUCKET(DAY,1,SYSDATETIME());  
-----  
2023-03-30 00:00:00.0000000
```

FUNCIÓN ISDATE

Acepta una cadena de caracteres como entrada y devuelve 1 si es convertible en un tipo de datos de fecha y hora y 0 si no lo es.

ISDATE(string)

```
SELECT ISDATE('20220212');  
-----  
1  
  
SELECT ISDATE('20220230');  
-----  
0
```



FUNCIÓN EOMONTH

La función EOMONTH acepta un valor de fecha y hora de entrada y devuelve la fecha respectiva del fin de mes como un valor de tipo DATE. La función también admite un segundo argumento opcional que indica cuántos meses añadir (o restar, si es negativo).

EOMONTH(entrada [, meses_a_agregar])

```
SELECT EOMONTH(SYSDATETIME());
```

```
-----  
2021-05-31
```

FUNCIÓN DATETRUNC

La función DATETRUNC trunca, o realiza el equivalente a la función floor, a la unidad especificada. Esta función fue incorporada en SQL Server 2022. En el ejemplo, al pasarle month como unidad, nos devuelve el primer día del mes correspondiente a la fecha.

DATETRUNC(unid, fecha_val)

```
SELECT DATETRUNC(month, '20220212');
```

```
-----  
2022-02-01 00:00:00.000000
```

FUNCIÓN GENERATE_SERIES

La función GENERATE_SERIES es una función de tabla que devuelve una secuencia de números en un rango solicitado. Esta función fue incorporada en SQL Server 2022. Se especifican el valor de inicio, el valor de finalización y opcionalmente el incremento (o decremento si utiliza un valor negativo). Como resultado se obtiene una tabla con una columna llamada value.

GENERATE_SERIES(valor_inicio, valor_final [,incremento])

```
SELECT value  
FROM GENERATE_SERIES( 1, 5 ) AS N;  
  
value  
-----  
1  
2  
3  
4  
5
```

Si bien no es una función de fechas, puede utilizarse para obtener una secuencia de fechas. Supongamos que necesitamos una secuencia con todos los días del año 2022, podemos obtenerlo con la siguiente expresión:

```
SELECT DATEADD(day, value, '20220101') AS dt  
FROM GENERATE_SERIES( 0, DATEDIFF(day, '20220101', '20221231') ) AS N;
```

La serie se genera desde 0 hasta el resultado de la diferencia en días de la fecha desde y la fecha hasta que necesitamos, en este caso el primer y el último día de 2022. Esto nos genera una secuencia entre 0 y 364, que utilizamos con la función DATEADD agregándolos como días a la fecha de inicio que nos interesa.



ANEXO ESTILOS DE FECHA Y HORA

Sin Siglo (yy) ⁽¹⁾	Con Siglo (yyyy)	Standard	Formato
-	0 or 100 ^(1,2)	Default for datetime and smalldatetime	mon dd yyyy hh:miAM (or PM) 1 = mm/dd/yy 101 = mm/dd/yyyy
1	101	U.S.	101 = mm/dd/yyyy
2	102	ANSI	2 = yy.mm.dd 102 = yyyy.mm.dd
3	103	British/French	3 = dd/mm/yy 103 = dd/mm/yyyy
4	104	German	4 = dd.mm.yy 104 = dd.mm.yyyy
5	105	Italian	5 = dd-mm-yy 105 = dd-mm-yyyy
6	106 ⁽¹⁾	-	6 = dd mon yy 106 = dd mon yyyy
7	107 ⁽¹⁾	-	7 = Mon dd, yy 107 = Mon dd, yyyy
8	108	-	hh:mi:ss
-	9 or 109 ^(1,2)	Default + milliseconds	mon dd yyyy hh:mi:ss:mmmAM (or PM)
10	110	USA	10 = mm-dd-yy 110 = mm-dd-yyyy
11	111	JAPAN	11 = yy/mm/dd 111 = yyyy/mm/dd
12	112	ISO	12 = yymmdd 112 = yyyymmdd
-	13 or 113 ^(1,2)	Europe default + milliseconds	dd mon yyyy hh:mi:ss:mmm(24h)
14	114	-	hh:mi:ss:mmm(24h)
-	20 or 120 ⁽²⁾	ODBC canonical	yyyy-mm-dd hh:mi:ss(24h)
-	21 or 121 ⁽²⁾	ODBC canonical (with milliseconds) default for time, date, datetime2, and datetimeoffset	yyyy-mm-dd hh:mi:ss.mmm(24h)
-	126 ⁽⁴⁾	ISO8601	yyyy-mm-ddThh:mi:ss.mmm (no spaces) Note: For a milliseconds (mmm) value of 0, the millisecond decimal fraction value will not display. For example, the value '2012-11-07T18:26:20.000 displays as '2012-11-07T18:26:20'.
-	127 ^(6,7)	ISO8601 with time zone Z.	yyyy-mm-ddThh:mi:ss.mmmZ (no spaces) Note: For a milliseconds (mmm) value of 0, the millisecond decimal value will not display. For example, the value '2012-11-07T18:26:20.000 will display as '2012-11-07T18:26:20'.
-	130 ^(1,2)	Hijri ⁽⁵⁾	dd mon yyyy hh:mi:ss:mmMM In this style, mon represents a multi-token Hijri unicode representation of the full month name. This value does not render correctly on a default US installation of SSMS.
-	131 ⁽²⁾	Hijri ⁽⁵⁾	dd/mm/yyyy hh:mi:ss:mmMMAM

¹ Estos valores de estilo devuelven resultados no deterministas. Incluye todos los estilos (yy) (sin siglo) y un subconjunto de estilos (yyyy) (con siglo).

² Los valores predeterminados (0 o 100, 9 o 109, 13 o 113, 20 o 120, y 21 o 121) siempre devuelven el siglo (yyyy).

³ Entrada cuando se convierte a **datetime**; salida cuando se convierte a cadenas de caracteres.

⁴ Diseñado para uso XML. Para la conversión de **datetime** o **smalldatetime** a cadenas de caracteres, consulte la tabla anterior para el formato de salida.

⁵ Hijri es un sistema de calendario con varias variaciones. SQL Server utiliza el algoritmo kuwaití.

⁶ Solo se admite cuando se convierte de cadenas de caracteres a **datetime** o **smalldatetime**. Al convertir cadenas de caracteres que representan solo la fecha o solo los componentes de tiempo para los tipos de datos **datetime** o **smalldatetime**, el componente de tiempo no especificado se establece en 00:00:00.000, y el componente de fecha no especificado se establece en 1900-01-01.



Use el indicador de zona horaria **Z** opcional para facilitar la asignación de valores de fecha y hora **XML** que tienen información de zona horaria a valores de fecha y hora de SQL Server que no tienen zona horaria. Z indica la zona horaria UTC-0. El desplazamiento HH:MM, en la dirección + o -, indica otras zonas horarias. Por ejemplo: 2006-12-12T23:45:12-08:00.

From	To	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	date	time	datetimeoffset	datetime2	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml	CLR UDT	hierarchyid
binary		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
varbinary		●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
char		■	■		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
varchar		■	■	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
nchar		■	■	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
nvarchar		■	■	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
datetime		■	■	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
smalldatetime		■	■	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
date		■	■	●	●	●	●	●	●		✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
time		■	■	●	●	●	●	●	●	●	✗		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
datetimeoffset		■	■	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
datetime2		■	■	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
decimal		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
numeric		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
float		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
real		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
bigint		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
int(INT4)		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
smallint(INT2)		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
tinyint(INT1)		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
money		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
smallmoney		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
bit		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
timestamp		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
uniqueidentifier		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
image		●	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●	✗	✗	✗	✗	✗	✗	✗	✗			
ntext		✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		
text		✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		
sql_variant		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
xml		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	○		
CLR UDT		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
hierarchyid		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		

■ Explicit conversion
● Implicit conversion
✗ Conversion not allowed
◆ Requires explicit CAST to prevent the loss of precision or scale that might occur in an implicit conversion.
○ Implicit conversions between xml data types are supported only if the source or target is untyped xml. Otherwise, the conversion must be explicit.



UNIDAD 2

PROGRAMANDO

En esta unidad veremos los objetos programables para familiarizarnos con las capacidades de T-SQL en esta área y con los conceptos que implica. Veremos variables, lotes, elementos de control de flujo, cursores, tablas temporales, rutinas como funciones definidas por el usuario, procedimientos almacenados y triggers.

VARIABLES

Las variables se utilizan para almacenar valores temporalmente, para darles un uso posterior en el mismo lote en donde son declaradas (veremos más adelante la definición de lotes, pero por ahora es importante saber que un lote es una o más instrucciones T-SQL enviadas al SQL Server para ser ejecutadas como una unidad).

Se utiliza *DECLARE* para declarar una o más variables, y se utiliza *SET* para asignar valor a una variable. Por ejemplo, en el siguiente código declaramos una variable llamada *@i* del tipo *INT* y le asignamos el valor 10:

```
DECLARE @i AS INT;
```

```
SET @i = 10;
```

También se puede declarar la variable e inicializarla en la misma sentencia, como aquí:

```
DECLARE @i AS INT = 10;
```

Cuando asignamos valor a una variable escalar, este debe ser el resultado de una expresión escalar. La expresión puede ser una subconsulta escalar. Por ejemplo, en el siguiente código declaramos una variable llamada *@nombreemp* y le asignamos el resultado de una consulta escalar que devuelve el nombre completo del empleado con ID igual a 3:

```
USE TSQLV6ES;
```

```
DECLARE @nombreemp AS NVARCHAR(61);
```

```
SET @nombreemp = (SELECT nombre + N' ' + apellido  
                  FROM RH.Empleados  
                 WHERE empid = 3);
```

```
SELECT @nombreemp AS nombreemp;
```

Este código devuelve la siguiente salida:

```
nombreemp
```

```
-----  
Judy Lew
```

La sentencia *SET* puede operar solamente sobre una variable a la vez, así que, si necesitamos asignar valores a múltiples variables, necesitamos múltiples sentencias *SET*. Esto puede implicar mucho trabajo adicional cuando necesitamos extraer múltiples atributos de una misma fila. Por ejemplo, en el siguiente código utilizaremos dos *SET* para extraer el nombre y el apellido respectivamente, del empleado con ID igual a 3:



```
USE TSQLV6ES;
DECLARE @nombre AS NVARCHAR(20), @apellido AS NVARCHAR(40);

SET @nombre = (SELECT nombre
               FROM RH.Empleados
               WHERE empid = 3);
SET @apellido = (SELECT apellido
                  FROM RH.Empleados
                  WHERE empid = 3);

SELECT @nombre AS nombre, @apellido AS apellido;
```

Este código devuelve la siguiente salida:

nombre	apellido
Judy	Lew

T-SQL también soporta el uso de *SELECT* como sentencia de asignación (que no es estándar), que puede utilizarse para consultar y asignar múltiples valores obtenidos de la misma fila a múltiples variables utilizando una sola sentencia. Como en el siguiente ejemplo:

```
USE TSQLV6ES;
DECLARE @nombre AS NVARCHAR(20), @apellido AS NVARCHAR(40);

SELECT @nombre = nombre, @apellido = apellido
FROM RH.Empleados
WHERE empid = 3;

SELECT @nombre AS nombre, @apellido AS apellido;
```

Cuando la consulta tiene como resultado una sola fila, el comportamiento del *SELECT* es predecible.

Sin embargo, notemos que, si la consulta devuelve más de una fila, la sintaxis es correcta y el código no falla. La asignación tiene lugar para cada fila resultante, y con cada fila accedida, los valores de la fila sobrescriben el valor existente de las variables. Cuando la asignación finaliza, los valores en las variables serán los de la última fila que se haya evaluado. Por ejemplo, la siguiente asignación tiene dos filas como resultado:

```
USE TSQLV6ES;
DECLARE @nombreemp AS NVARCHAR(61);

SELECT @nombreemp = nombre + N' ' + apellido
FROM RH.Empleados
WHERE jefeid = 2;

SELECT @nombreemp AS nombreemp;
```

La información del empleado que termina en la variable luego de la asignación con *SELECT* depende del orden en el que SQL Server acceda a las filas – y no podemos controlar ese orden –

Cuando corrí este código, yo obtuve la siguiente salida:

nombreemp
Sven Mortensen

La sentencia *SET* es más segura que la asignación con *SELECT* porque requiere el uso de una consulta escalar para obtener datos de una tabla. Recordemos que una consulta escalar falla en tiempo de ejecución si devuelve más de un valor. Por ejemplo, el siguiente código falla:



```
USE TSQLV6ES;
DECLARE @nombreemp AS NVARCHAR(61);
SET @nombreemp = (SELECT nombre + N' ' + apellido
                   FROM RH.Empleados
                   WHERE jefeid = 2);

SELECT @nombreemp AS nombreemp;
```

Como no se asignó valor a la variable, permanece en *NULL*, que es el valor por default cuando las variables no son inicializadas. Este código devuelve la siguiente salida:

```
Msg 512, Level 16, State 1, Line 3
Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=,
<, <= , >, >= or when the subquery is used as an expression.
nombreemp
-----
NULL
```

LOTES (BATCHES)

Un lote está formado por una o más sentencias T-SQL enviadas por una aplicación cliente a SQL Server para su ejecución como una unidad. El lote se somete a análisis (comprobación de sintaxis), resolución / enlace (comprobación de la existencia de objetos y columnas a los que se hace referencia, comprobación de permisos) y optimización como una unidad.

No confundamos transacciones y lotes. Una transacción es una unidad atómica de trabajo. Un lote puede tener múltiples transacciones y una transacción puede enviarse en partes como múltiples lotes. Cuando una transacción se cancela o se revierte, SQL Server deshace la actividad parcial que ha tenido lugar desde el inicio de la transacción, independientemente de dónde comenzó el lote.

Las interfaces de programación de aplicaciones del cliente (API), como ADO.NET, proporcionan métodos para enviar un lote de código a SQL Server para su ejecución. Las utilidades de SQL Server como SQL Server Management Studio (SSMS), SQLCMD y OSQL, o utilidades como Azure Data Studio, proporcionan un comando de herramienta de cliente llamado GO que señala el final de un lote. Tengan en cuenta que el comando GO es un comando de la herramienta cliente y no un comando del servidor T-SQL.

UN LOTE COMO UNIDAD DE ANÁLISIS

Un lote es un conjunto de comandos que se analizan y ejecutan como una unidad. Si el análisis se realiza correctamente, SQL Server intenta ejecutar el lote. En el caso de un error de sintaxis en el lote, el lote completo no se envía a SQL Server para su ejecución. Por ejemplo, el siguiente código tiene tres lotes, el segundo de los cuales tiene un error de sintaxis (*FOM* en lugar de *FROM* en la segunda consulta):

```
-- Lote válido
PRINT 'Primer Lote';
USE TSQLV6ES;
GO
-- Lote inválido
PRINT 'Segundo Lote';
SELECT clienteid
FROM Ventas.Clientes
SELECT ordenid
FOM Ventas.Ordenes;
GO
-- Lote válido
PRINT 'Tercer lote';
SELECT empid
FROM RH.Empleados;
```



Como el segundo lote tiene un error de sintaxis, el lote entero no es enviado a SQL Server para su ejecución. El primer y el tercer lote pasan la validación de sintaxis y por lo tanto son enviados para su ejecución. Este código genera la siguiente salida, en donde se puede ver que el segundo lote no fue ejecutado:

```
Primer lote
Msg 102, Level 15, State 1, Line 10
Incorrect syntax near 'Ventas'.
Tercer lote
empid
-----
2
7
1
5
6
8
3
9
4
```

LOTES Y VARIABLES

Una variable es local al lote en el que está definida. Si se refiere a una variable que se definió en otro lote, obtendremos un error que indica que la variable no se definió. Por ejemplo, el siguiente código declara una variable e imprime su contenido en un lote, y luego intenta imprimir su contenido de otro lote:

```
DECLARE @i AS INT;
SET @i = 10;

-- Exito
PRINT @i;
GO

-- Error
PRINT @i;
```

La referencia a la variable en la primera instrucción *PRINT* es válida porque aparece en el mismo lote donde se declaró la variable, pero la segunda referencia no es válida. Por lo tanto, la primera instrucción *PRINT* devuelve el valor de la variable (10), mientras que la segunda falla. Aquí está la salida devuelta de este código:

```
10
Msg 137, Level 15, State 2, Line 9
Must declare the scalar variable "@i".
```

SENTENCIAS QUE NO SE PUEDEN COMBINAR EN EL MISMO LOTE

Las siguientes sentencias no pueden ser combinadas con otras sentencias en el mismo lote:

CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER, y CREATEVIEW. Por ejemplo, el siguiente código tiene una sentencia *DROP* seguida de un *CREATE VIEW* en el mismo lote, lo que es inválido:



```
USE TSQLV6ES;
DROP VIEW IF EXISTS Ventas.MiVista;

CREATE VIEW Ventas.MiVista AS
SELECT YEAR(fechaorden) AS anioorden, COUNT(*) AS numordenes
FROM Ventas.Ordenes
GROUP BY YEAR(fechaorden);
GO
```

Un intento por ejecutar este código genera el siguiente error:

```
Msg 111, Level 15, State 1, Line 3
'CREATE VIEW' must be the first statement in a query batch.
```

Para solucionar el problema, separen el *DROP VIEW* y el *CREATE VIEW* en diferentes lotes, agregando *GO* después del *DROP VIEW*.

UN LOTE COMO UNIDAD DE RESOLUCIÓN

Un lote es una unidad de resolución (también conocida como *enlace*). Esto significa que la verificación de la existencia de objetos y columnas ocurre a nivel de lote. Tengamos en cuenta este hecho cuando estemos diseñando límites de lotes. Cuando aplicamos cambios de esquema a un objeto e intentamos manipular los datos del objeto en el mismo lote, es posible que SQL Server aún no esté al tanto de los cambios de esquema y falle la declaración de manipulación de datos con un error de resolución. Demostraré el problema a través de un ejemplo y luego recomendaré las mejores prácticas.

Ejecute el siguiente código para crear una tabla llamada *T1* en la base de datos actual, con una columna llamada *col1*:

```
DROP TABLE IF EXISTS dbo.T1;
CREATE TABLE dbo.T1(col1 INT);
```

A continuación, intentemos agregar una columna llamada *col2* a *T1* y consultemos la nueva columna en el mismo lote:

```
ALTER TABLE dbo.T1 ADD col2 INT;
SELECT col1, col2 FROM dbo.T1;
```

Aunque el código parezca perfectamente válido, el lote falla durante la fase de resolución con el siguiente error:

```
Msg 207, Level 16, State 1, Line 5
Invalid column name 'col2'.
```

En el momento en que se resolvió la instrucción *SELECT*, *T1* tenía solo una columna, y la referencia a la columna *col2* causó el error. Una buena práctica que podemos seguir para evitar tales problemas es separar las declaraciones en lenguaje de definición de datos (DDL) y lenguaje de manipulación de datos (DML) en diferentes lotes, como se muestra en el siguiente ejemplo:

```
ALTER TABLE dbo.T1 ADD col2 INT;
GO
SELECT col1, col2 FROM dbo.T1;
```

LA OPCIÓN GO

El comando *GO* no es realmente un comando T-SQL; En realidad, es un comando utilizado por las herramientas cliente de SQL Server, como SSMS, para indicar el final de un lote. Este comando admite un



argumento que indica cuántas veces desea ejecutar el lote. Para ver cómo funciona el comando *GO* con el argumento, primero cree la tabla *T1* utilizando el siguiente código:

```
DROP TABLE IF EXISTS dbo.T1;
CREATE TABLE dbo.T1(col1 INT IDENTITY);
```

La columna *col1* obtiene sus valores automáticamente de una propiedad de identidad. Tengamos en cuenta que la demostración funciona igual de bien si utiliza una restricción predeterminada para generar valores a partir de un objeto de secuencia.

A continuación, ejecutemos el siguiente código para suprimir la salida predeterminada producida por las declaraciones DML que indica cuántas filas se vieron afectadas:

```
SET NOCOUNT ON;
```

Finalmente, ejecutemos el siguiente código para definir un lote con una instrucción *INSERT DEFAULT VALUES* y para ejecutar el lote 100 veces:

```
INSERT INTO dbo.T1 DEFAULT VALUES;
GO 100
SELECT * FROM dbo.T1;
```

La consulta devuelve 100 filas con los valores de 1 hasta 100 en la columna *col1*.

ELEMENTOS DE FLUJO

Usamos elementos de flujo para controlar el flujo de nuestro código. T-SQL proporciona formas básicas de control con elementos de flujo, incluido el *IF...ELSE* y el *WHILE*.

EL ELEMENTO DE FLUJO IF...ELSE

Usamos el *IF...ELSE* para controlar el flujo de nuestro código basado en el resultado de un predicado. Especificamos una instrucción o un bloque de instrucciones que se ejecuta si el predicado es *VERDADERO* y, opcionalmente, una instrucción o un bloque de instrucciones que se ejecuta si el predicado es *FALSO* o *DESCONOCIDO*.

Por ejemplo, el siguiente código verifica si hoy es el último día del año (en otras palabras, si el año de hoy es diferente al de mañana). Si esto es cierto, el código imprime un mensaje que dice que hoy es el último día del año; si no es cierto ("else"), el código imprime un mensaje que dice que hoy no es el último día del año:

```
IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))
PRINT 'Hoy es el último día del año.';
ELSE
PRINT 'Hoy no es el último día del año.';
```

En este ejemplo utilizamos *PRINT* para demostrar qué partes del código son ejecutadas y qué partes no, pero por supuesto se puede utilizar cualquier sentencia.

Tengamos en cuenta que T-SQL utiliza lógica de tres valores y que el bloque *ELSE* se activa cuando el predicado es *FALSO* o *DESCONOCIDO*. En los casos en los que tanto *FALSO* como *DESCONOCIDO* son posibles resultados del predicado (por ejemplo, cuando están implicados los valores nulos) y necesita un tratamiento diferente para cada caso, asegúrese de realizar una prueba explícita para los valores nulos con el predicado *IS NULL*.



Si el flujo que necesita controlar involucra más de dos casos, puede anidar elementos *IF...ELSE*. Por ejemplo, el siguiente código maneja los siguientes tres casos de manera diferente:

- Hoy es el último día del año.
- Hoy es el último día del mes, pero no el último día del año.
- Hoy no es el último día del mes.

```
IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))
    PRINT 'Hoy es el último día del año.';
ELSE
    IF MONTH(SYSDATETIME()) <> MONTH(DATEADD(day, 1, SYSDATETIME()))
        PRINT 'Hoy es el último día del mes, pero no es el último día del año.';
    ELSE
        PRINT 'Hoy no es el último día del mes.';
```

Si necesitamos ejecutar más de una sentencia en las secciones *IF* o *ELSE*, necesitamos usar un bloque de declaración. Marcamos los límites de un bloque de instrucciones con las palabras clave *BEGIN* y *END*. Por ejemplo, el siguiente código muestra cómo ejecutar un tipo de proceso si es el primer día del mes y otro tipo de proceso si no es:

```
IF DAY(SYSDATETIME()) = 1
BEGIN
    PRINT 'Hoy es el primer día del mes.';
    PRINT 'Comenzando el proceso primer-dia-de-mes.';
    /* ... acá iría el proceso ... */
    PRINT 'Se ha finalizado el proceso primer-dia-de-mes.';
END;
ELSE
BEGIN
    PRINT 'Hoy no es el primer día del mes.';
    PRINT 'Comenzando el proceso no-primer-dia-de-mes.';
    /* ... acá iría el proceso ... */
    PRINT 'Se ha finalizado el proceso no-primer-dia-de-mes.';
END;
```

WHILE

T-SQL cuenta con *WHILE*, mediante el cual podemos ejecutar código en un bucle. *WHILE* ejecuta una sentencia o un bloque de sentencias repetidas veces mientras que el predicado que se especificó se evalúe como *VERDADERO*. Cuando el predicado se evalúa como *FALSO* o como *DESCONOCIDO*, el bucle termina.

T-SQL no cuenta con un elemento de flujo que permita ejecutar un número predeterminado de veces (como es el *FOR* en otros lenguajes), pero este comportamiento puede ser emulado con un *WHILE* y una variable. Por ejemplo, en el siguiente código veremos cómo emular un bucle que se ejecute 10 veces:

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END;
```

En el código declaramos una variable entera llamada *@i* que es utilizada como contador en el bucle y que se inicializa con el valor 1. El código entra en el bucle y se ejecuta mientras que la variable sea menor o igual a 10. En cada iteración, el código imprime el valor actual de *@i* y lo incrementa en 1. El código devuelve la siguiente salida, mostrando que la iteración se ejecutó 10 veces:



1
2
3
4
5
6
7
8
9
10

Si en algún punto del cuerpo del bucle deseamos salir del bucle actual y proceder a ejecutar la instrucción que aparece después del cuerpo del bucle, usamos el comando *BREAK*. Por ejemplo, el siguiente código sale del bucle si el valor de *@i* es igual a 6:

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    IF @i = 6 BREAK;
    PRINT @i;
    SET @i = @i + 1;
END;
```

Este código produce la siguiente salida que muestra que el bucle iteró cinco veces y terminó al comienzo de la sexta iteración:

1
2
3
4
5

Por supuesto, este código no es muy notable; si deseamos que el bucle se repita solo cinco veces, simplemente debemos especificar el predicado *@i <= 5*. Aquí solo quería demostrar el uso del comando *BREAK* con un simple ejemplo.

Si en algún punto del cuerpo del bucle desea omitir el resto de la actividad en la iteración actual y evaluar nuevamente el predicado del bucle, use el comando *CONTINUE*. Por ejemplo, el siguiente código muestra cómo omitir la actividad de la sexta iteración del bucle desde el punto donde aparece la instrucción *IF* hasta el final del cuerpo del bucle:

```
DECLARE @i AS INT = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    IF @i = 6 CONTINUE;
    PRINT @i;
END;
```

La salida de este código muestra que el valor de *@i* fue impreso en todas las iteraciones, menos en la sexta:



1
2
3
4
5
7
8
9
10

Como otro ejemplo del uso de *WHILE*, el siguiente código crea una tabla llamada *dbo.Numeros* y la llena con 1.000 filas con los valores desde 1 hasta 1.000 en la columna *n*:

```
SET NOCOUNT ON;
DROP TABLE IF EXISTS dbo.Numeros;
CREATE TABLE dbo.Numeros(n INT NOT NULL PRIMARY KEY);
GO

DECLARE @i AS INT = 1;
WHILE @i <= 1000
BEGIN
    INSERT INTO dbo.Numeros(n)
    VALUES(@i);
    SET @i = @i + 1;
END;
```

TRANSACCIONES Y CONCURRENCIA

Veremos a continuación las transacciones y sus propiedades y veremos cómo Microsoft SQL Server maneja a los usuarios que intentan acceder a los mismos datos al mismo tiempo. Explicaremos cómo SQL Server usa bloqueos para aislar datos inconsistentes, cómo podemos solucionar situaciones de bloqueo y cómo podemos controlar el nivel de coherencia cuando consultamos datos con niveles de aislamiento. También veremos interbloqueos y formas de mitigar su ocurrencia.

Nos enfocaremos en los aspectos de concurrencia de la representación de datos tradicional en tablas basadas en disco.

TRANSACCIONES

Una transacción es una unidad de trabajo que puede incluir múltiples actividades que consultan y modifican datos y que también pueden cambiar la definición de datos.

Podemos definir los límites de las transacciones de forma explícita o implícita. El comienzo de una transacción se define explícitamente con una instrucción **BEGIN TRAN** (o **BEGIN TRANSACTION**). El final de una transacción se define explícitamente con una declaración **COMMIT TRAN** si deseamos confirmarla y con una declaración **ROLLBACK TRAN** (o **ROLLBACK TRANSACTION**) si deseamos deshacer sus cambios. Aquí hay un ejemplo de cómo marcar los límites de una transacción con dos instrucciones *INSERT*:

```
BEGIN TRAN;
INSERT INTO dbo.T1(keycol, col1, col2) VALUES(4, 101, 'C');
INSERT INTO dbo.T2(keycol, col1, col2) VALUES(4, 201, 'X');
COMMIT TRAN;
```

Si no marcamos explícitamente los límites de una transacción, de forma predeterminada, SQL Server trata cada instrucción individual como una transacción; en otras palabras, de forma predeterminada, SQL Server inicia automáticamente una transacción antes de que comience cada sentencia y confirma la transacción al final de cada sentencia. Este modo se conoce como modo de confirmación automática (auto-commit).



Podemos cambiar la forma en que SQL Server maneja las transacciones implícitas para que no se confirme automáticamente activando una opción de sesión llamada **IMPLICIT_TRANSACTIONS**. Esta opción está desactivada de forma predeterminada. Cuando esta opción está activada, no tenemos que especificar la declaración **BEGIN TRAN** para marcar el comienzo de una transacción, pero debemos marcar el final de la transacción con una declaración **COMMIT TRAN** o **ROLLBACK TRAN**.

Después de que una transacción se confirma o revierte, a menos que abramos otra transacción explícita, la siguiente declaración ejecutada implícitamente inicia otra transacción.

Las transacciones tienen cuatro propiedades: **Atomicidad**, **Consistencia**, **aislamiento (Isolation)** y **Durabilidad**, abreviadas con el acrónimo **ACID**:

- **Atomicidad:** Una transacción es una unidad atómica de trabajo. O se realizan todos los cambios en la transacción o no se realiza ninguno. Si el sistema falla antes de que se complete una transacción (antes de que la instrucción de confirmación se registre en el registro de transacciones), al reiniciar, SQL Server deshace los cambios realizados. Además, si se encuentran errores durante la transacción y el error se considera lo suficientemente grave, como que el grupo de archivos de destino está lleno cuando intenta insertar datos, SQL Server revierte automáticamente la transacción. Algunos errores, como las infracciones de clave principal y los tiempos de espera de vencimiento de bloqueo, no se consideran lo suficientemente graves como para justificar una reversión automática de la transacción. Si deseamos que todos los errores cancelen la ejecución y provoquen la reversión de cualquier transacción abierta, podemos habilitar una opción de sesión llamada **XACT_ABORT**. Podemos usar el código de manejo de errores para capturar dichos errores y aplicar algún curso de acción (por ejemplo, registrar el error y revertir la transacción).
- **Consistencia:** el término consistencia se refiere al estado de los datos a los que el sistema de administración de bases de datos relacionales (RDBMS) le da acceso a medida que las transacciones simultáneas los modifican y consultan. Como probablemente podemos imaginar, la consistencia es un término subjetivo, que depende de las necesidades de cada aplicación. La sección "Niveles de aislamiento" más adelante explica el nivel de coherencia que proporciona SQL Server de forma predeterminada y cómo podemos controlarlo si el comportamiento predeterminado no es adecuado para la aplicación. La consistencia también se refiere al hecho de que la base de datos debe adherirse a todas las reglas de integridad que se han definido dentro de ella mediante restricciones (como claves primarias, restricciones únicas y claves foráneas). La transacción hace que la base de datos pase de un estado coherente a otro.
- **Aislamiento (Isolation):** el aislamiento garantiza que las transacciones solo accedan a datos consistentes. Cada uno controla lo que significa la consistencia para sus transacciones a través de un mecanismo llamado niveles de aislamiento. Con las tablas basadas en disco, SQL Server admite dos modelos diferentes para manejar el aislamiento: uno basado únicamente en el bloqueo y otro basado en una combinación de bloqueo y control de versiones de filas. Para simplificar, nos referiremos a este último sólo como control de versiones de filas. El modelo basado en el bloqueo es el predeterminado en instalaciones en caja. En este modelo, los lectores requieren bloqueos compartidos. Si el estado actual de los datos es inconsistente, los lectores se bloquean hasta que el estado de los datos sea consistente. El modelo basado en el control de versiones de filas es el predeterminado en Azure SQL Database. En este modelo, los lectores no toman bloqueos compartidos y no necesitan esperar. Si el estado actual de los datos es inconsistente, el lector obtiene un estado consistente anterior. La sección "Niveles de aislamiento" más adelante proporciona más detalles sobre ambas formas de manejar el aislamiento.
- **Durabilidad:** La propiedad de durabilidad significa que una vez que el motor de la base de datos reconoce una instrucción de confirmación, se garantiza que los cambios de la transacción serán duraderos, o en otras palabras, persistentes en la base de datos. Se reconoce una confirmación devolviendo el control a la aplicación y ejecutando la siguiente línea de código. El mecanismo que garantiza la propiedad de



durabilidad en SQL Server depende de la arquitectura de recuperación que esté en vigor. Existe la arquitectura más tradicional utilizada antes de SQL Server 2019. Hay una arquitectura más nueva que está disponible en SQL Server 2019 o posterior si habilitamos una función llamada Recuperación acelerada de datos (**ADR**) para admitir un proceso de recuperación más rápido. La arquitectura tradicional es más simple e involucra menos componentes. Los cambios de datos siempre se escriben en el registro de transacciones de la base de datos en el disco antes de que se escriban en la parte de datos de la base de datos en el disco. Una vez que la instrucción de confirmación se asienta en el registro de transacciones en el disco, la transacción se considera duradera incluso si el cambio aún no se ha realizado en la porción de datos en el disco. Cuando el sistema se inicia, ya sea normalmente o después de una falla del sistema, SQL Server ejecuta un proceso de recuperación en cada base de datos que implica analizar el registro, luego aplicar una fase de rehacer y luego aplicar una fase de deshacer. La fase de rehacer implica avanzar (reproducir) todos los cambios de cualquier transacción cuya instrucción de confirmación se escriba en el registro, pero cuyos cambios aún no hayan llegado a la parte de datos. La fase de deshacer implica revertir (deshacer) los cambios de cualquier transacción cuya instrucción de confirmación no se asentó en el registro. Si habilita ADR, la arquitectura de recuperación es más sofisticada, involucrando componentes adicionales y un proceso más sofisticado. Puede encontrar detalles sobre ADR en la documentación del producto aquí: <https://learn.microsoft.com/en-us/sql/relational-databases/accelerated-databaserecovery-concepts>. Ya sea que se base en la arquitectura de recuperación tradicional o en la más nueva, obtendremos la misma garantía de durabilidad de la transacción tras el reconocimiento de su instrucción de confirmación, sólo que con una mecánica diferente detrás de bambalinas.

Por ejemplo, las siguientes líneas de código definen una transacción que registra información sobre una nueva orden en la base *TSQLV6ES*:

```
USE TSQLV6ES;
-- Comienzo una nueva transaccion
BEGIN TRAN;
-- Declaro una variable
DECLARE @nuevoordenid AS INT;
-- Inserto una nueva orden en la tabla Ventas.Ordenes
INSERT INTO Ventas.Ordenes
(clienteid, empid, fechaorden, fecharequerida, fechadespacho,
transporteid, flete, nombreenvio, direccionenvio, ciudadenvio,
codigopostalenvio, paisenvio)
VALUES
(85, 5, '20220212', '20220301', '20220216',
3, 32.38, N'Ship to 85-B', N'6789 rue de l''Abbaye', N'Reims',
N'10345', N'France');
-- Guardo el nuevo ID de orden en una variable
SET @nuevoordenid = SCOPE_IDENTITY();
-- Devuelvo el nuevo ID de orden
SELECT @nuevoordenid AS nuevoordenid;
-- Inserto lineas de orden para la nueva orden en Ventas.DetallesOrden
INSERT INTO Ventas.DetallesOrden
(ordenid, productoid, preciouunitario, cantidad, descuento)
VALUES(@nuevoordenid, 11, 14.00, 12, 0.000),
(@nuevoordenid, 42, 9.80, 10, 0.000),
(@nuevoordenid, 72, 34.80, 5, 0.000);
-- Confirmo la transaccion
COMMIT TRAN
```

El código de la transacción inserta una fila con la información del encabezado del pedido en la tabla *Ventas.Ordenes* y algunas filas con la información de las líneas del pedido en la tabla *Ventas.DetallesOrden*. SQL Server produce automáticamente el nuevo ID de pedido porque la columna *ordenid* tiene una propiedad de identidad (**IDENTITY**). Inmediatamente después de que el código inserta la nueva fila en la tabla *Ventas.Ordenes*,



almacena el ID de pedido recién generado en una variable local y luego usa esa variable local al insertar filas en la tabla *Ventas.Ordenes*. Para fines de prueba, agregamos una instrucción *SELECT* que devuelve el ID de pedido del pedido recién generado. Aquí está el resultado de la declaración *SELECT* después de que se ejecuta el código:

```
nuevoordenid  
-----  
11078
```

Tengan en cuenta que este ejemplo no tiene manejo de errores y no hace ninguna provisión para ROLLBACK en caso de error. Para manejar errores, puede incluir en la transacción una construcción TRY/CATCH (Lo veremos más adelante en Manejo de Errores)

Cuando hayamos terminado, ejecutamos el siguiente código para la limpieza:

```
USE TSQLV6ES;  
DELETE FROM Ventas.DetallesOrden  
WHERE ordenid > 11077;  
DELETE FROM Ventas.Ordenes  
WHERE ordenid > 11077;
```

TRABAS Y BLOQUEO

De forma predeterminada, un producto de caja de SQL Server utiliza un modelo de bloqueo puro para aplicar la propiedad de aislamiento de las transacciones. Las siguientes secciones brindan detalles sobre el bloqueo y explican cómo solucionar situaciones de bloqueo causadas por solicitudes de bloqueo en conflicto.

Como se mencionó, Azure SQL Database usa el modelo de control de versiones de filas de forma predeterminada. Si van a probar el código de esta sección en Azure SQL Database, deben desactivar la propiedad de la base de datos **READ_COMMITTED_SNAPSHOT** para cambiar al modelo de bloqueo como predeterminado. Use el siguiente código para lograr esto:

```
ALTER DATABASE TSQLV6ES SET READ_COMMITTED_SNAPSHOT OFF;
```

Si estamos conectados a la base de datos TSQLV6ES, también podemos usar la palabra clave CURRENT en lugar del nombre de la base de datos. Además, de forma predeterminada, las conexiones a Azure SQL Database se agotan con bastante rapidez. Entonces, si una demostración que está ejecutando no funciona como se esperaba, podría ser que una conexión involucrada en esa demostración se agotó.

TRABAS (LOCKS)

Las trabas (locks) son recursos de control obtenidos por una transacción para proteger los recursos de datos, evitando el acceso conflictivo o incompatible por parte de otras transacciones. Primero cubriremos los modos de bloqueo importantes admitidos por SQL Server y su compatibilidad, y luego describiremos los recursos bloqueables.

MODOS DE BLOQUEO Y COMPATIBILIDAD

A medida que comenzamos a aprender sobre transacciones y simultaneidad, primero debemos familiarizarnos con dos modos de bloqueo principales: exclusivo y compartido.

Cuando intentamos modificar datos, la transacción solicita un bloqueo exclusivo en el recurso de datos, independientemente de su nivel de aislamiento. Si se otorga, el bloqueo exclusivo se mantiene hasta el final de la transacción. Para transacciones de una sola sentencia, esto significa que el bloqueo se mantiene hasta que se completa la sentencia. Para transacciones de múltiples instrucciones, esto significa que el bloqueo se mantiene hasta que se completan todas las instrucciones y la transacción finaliza mediante un comando **COMMIT TRAN** o **ROLLBACK TRAN**.



Los bloqueos exclusivos se denominan "exclusivos" porque no podemos obtener un bloqueo exclusivo en un recurso si otra transacción tiene un modo de bloqueo en el recurso, y no se puede obtener ningún modo de bloqueo en un recurso si otra transacción tiene un bloqueo exclusivo en el recurso. Esta es la forma en que se comportan las modificaciones de forma predeterminada, y este comportamiento predeterminado no se puede cambiar, ni en términos del modo de bloqueo requerido para modificar un recurso de datos (exclusivo) ni en términos de la duración del bloqueo (hasta el final de la transacción). En términos prácticos, esto significa que, si una transacción modifica filas, hasta que se complete la transacción, otra transacción no puede modificar las mismas filas. Sin embargo, si otra transacción puede leer las mismas filas o no, depende de su nivel de aislamiento.

En cuanto a la lectura de datos, los valores predeterminados son diferentes para un producto de caja de SQL Server y Azure SQL Database. En SQL Server, el nivel de aislamiento predeterminado se denomina **READ COMMITTED**. En este aislamiento, cuando intentamos leer datos, la transacción solicita de forma predeterminada un bloqueo compartido en el recurso de datos y libera el bloqueo tan pronto como se realiza la sentencia de lectura con ese recurso. Este modo de bloqueo se denomina "compartido" porque varias transacciones pueden mantener bloqueos compartidos en el mismo recurso de datos simultáneamente. Aunque no podemos cambiar el modo de bloqueo y la duración requerida cuando estamos modificando datos, podemos controlar la forma en que se maneja el bloqueo cuando estamos leyendo datos cambiando su nivel de aislamiento.

En Azure SQL Database, el nivel de aislamiento predeterminado se denomina **READ COMMITTED SNAPSHOT**. En lugar de confiar sólo en el bloqueo, este nivel de aislamiento se basa en una combinación de bloqueo y control de versiones de filas. Bajo este nivel de aislamiento, los lectores no requieren bloqueos compartidos y, por lo tanto, nunca esperan; confían en la tecnología de control de versiones de filas para proporcionar el aislamiento esperado. En términos prácticos, esto significa que bajo el nivel de aislamiento **READ COMMITTED**, si una transacción modifica filas, hasta que la transacción se complete, otra transacción no podrá leer las mismas filas. Este enfoque del control de concurrencia se conoce como enfoque de concurrencia pesimista. Bajo el nivel de aislamiento **READ COMMITTED SNAPSHOT**, si una transacción modifica filas, otra transacción que intente leer los datos obtendrá el último estado confirmado de las filas que estaba disponible cuando se inició la instrucción. Este enfoque del control de concurrencia se conoce como enfoque de concurrencia optimista.

Esta interacción de bloqueo entre transacciones se conoce como compatibilidad de bloqueo. En la Tabla 1 podemos ver la compatibilidad de bloqueo de bloqueos exclusivos y compartidos (cuando trabajamos con un nivel de aislamiento que genera estos bloqueos). Las columnas representan los modos de bloqueo otorgados y las filas representan los modos de bloqueo solicitados.

TABLA 1

Modo solicitado	Concedido Exclusivo (X)	Concedido Compartido (S)
Exclusivo	No	No
Compartido	No	Sí

Un "No" en la intersección significa que los bloqueos son incompatibles y se deniega el modo solicitado; el solicitante debe esperar. Un "Sí" en la intersección significa que los bloqueos son compatibles y se acepta el modo solicitado.

Lo siguiente resume la interacción de bloqueo entre transacciones en términos simples: los datos que fueron modificados por una transacción no pueden ser modificados ni leídos (al menos de forma predeterminada en un producto de caja de SQL Server) por otra transacción hasta que finalice la primera transacción. Y mientras una transacción lee los datos, otra no puede modificarlos (al menos de forma predeterminada en un producto de caja de SQL Server).



TIPOS DE RECURSOS BLOQUEABLES

SQL Server puede bloquear diferentes tipos de recursos. Estos incluyen filas, páginas, objetos (por ejemplo, tablas), bases de datos y otros. Las filas residen dentro de las páginas, y las páginas son los bloques de datos físicos que contienen datos de tablas o índices. Primero debemos familiarizarnos con estos tipos de recursos y, en una etapa más avanzada, es posible que tengamos que familiarizarnos con otros tipos de recursos bloqueables, como extensiones, unidades de asignación y heaps o árboles B.

Para obtener un bloqueo en un determinado tipo de recurso, la transacción primero debe obtener bloqueos de intención del mismo modo en niveles más altos de granularidad. Por ejemplo, para obtener un bloqueo exclusivo en una fila, la transacción primero debe adquirir bloqueos exclusivos de intención en la tabla y la página donde reside la fila. De manera similar, para obtener un bloqueo compartido en un cierto nivel de granularidad, la transacción primero debe adquirir bloqueos compartidos de intención en niveles más altos de granularidad. El propósito de los bloqueos de intención es detectar de manera eficiente solicitudes de bloqueo incompatibles en niveles más altos de granularidad y evitar que se concedan. Por ejemplo, si una transacción mantiene un bloqueo en una fila y otra solicita un modo de bloqueo incompatible en toda la página o tabla donde reside esa fila, es fácil para SQL Server identificar el conflicto debido a los bloqueos de intención que adquirió la primera transacción, en la página y la tabla. Los bloqueos de intención no interfieren con las solicitudes de bloqueos en niveles más bajos de granularidad. Por ejemplo, un intento de bloqueo en una página no evita que otras transacciones adquieran modos de bloqueo incompatibles en filas dentro de la página. La Tabla 2 amplía la tabla de compatibilidad de bloqueos que se muestra en la Tabla 1, agregando bloqueos exclusivos de intención y compartidos de intención.

TABLA 2

Modo solicitado	Concedido Exclusivo (X)	Concedido Compartido (S)	Concedido Intención Exclusivo (IX)	Concedido Intención Compartida (IS)
Exclusivo	No	No	No	No
Compartido	No	Sí	No	Sí
Intención Exclusivo	No	No	Sí	Sí
Intención Compartido	No	Sí	Sí	Sí

SQL Server determina dinámicamente qué tipos de recursos bloquear. Naturalmente, para una simultaneidad ideal, es mejor bloquear sólo lo que debe bloquearse, es decir, sólo las filas afectadas. Sin embargo, los bloqueos requieren recursos de memoria y sobrecarga de administración interna. Por lo tanto, SQL Server considera tanto la concurrencia como los recursos del sistema cuando elige qué tipos de recursos bloquear. Cuando SQL Server estima que una transacción interactuará con una pequeña cantidad de filas, tiende a usar bloqueos de fila. Con un mayor número de filas, SQL Server tiende a utilizar bloqueos de página.

SQL Server puede adquirir primero bloqueos detallados (como bloqueos de fila o de página) y, en determinadas circunstancias, intentar escalar los bloqueos detallados a un bloqueo de tabla para preservar la memoria. La escalada de bloqueos se activa cuando una sola declaración adquiere al menos 5000 bloqueos contra el mismo objeto. La verificación de escalada de bloqueos ocurre primero cuando una transacción contiene 2500 bloqueos y luego por cada 1250 bloqueos nuevos.

Podemos establecer una opción de tabla denominada **LOCK_ESCALATION** mediante la instrucción **ALTER TABLE** para controlar el comportamiento de la escalada de bloqueo. Podemos deshabilitar el escalamiento de bloqueos si lo deseamos, o podemos determinar si el escalamiento se lleva a cabo a nivel de tabla (predeterminado) o a nivel de partición. (Una tabla se puede organizar físicamente en varias unidades más pequeñas llamadas particiones).



SOLUCIÓN DE PROBLEMAS DE BLOQUEO

Cuando una transacción mantiene un bloqueo en un recurso de datos y otra transacción solicita un bloqueo incompatible en el mismo recurso, la solicitud se bloquea y el solicitante ingresa en un estado de espera. De forma predeterminada, la solicitud bloqueada sigue esperando hasta que el bloqueador libera el bloqueo que interfiere. Más adelante en esta sección, veremos cómo podemos definir un tiempo de caducidad de bloqueo en la sesión si deseamos restringir la cantidad de tiempo que espera una solicitud bloqueada antes de que se agote.

El bloqueo es normal en un sistema siempre que las solicitudes se satisfagan dentro de un período de tiempo razonable. Sin embargo, si algunas solicitudes terminan esperando demasiado, es posible que debamos solucionar la situación de bloqueo y ver si podemos hacer algo para evitar latencias tan largas. Por ejemplo, las transacciones de ejecución prolongada dan como resultado que los bloqueos se mantengan durante períodos prolongados. Podemos intentar acortar dichas transacciones, moviendo actividades que se supone que no son parte de la unidad de trabajo fuera de la transacción. Un error en la aplicación puede resultar en una transacción que permanece abierta en ciertas circunstancias. Si identificamos un error de este tipo, podemos corregirlo y asegurarnos de que la transacción se cierre en todas las circunstancias.

El siguiente ejemplo demuestra una situación de bloqueo y cómo solucionarla. Supongamos que se estamos ejecutando bajo del nivel de aislamiento **READ COMMITTED**. Abrimos tres ventanas de consulta separadas en SQL Server Management Studio. (Para este ejemplo, nos referiremos a ellos como *Conexión 1*, *Conexión 2* y *Conexión 3*). Asegúrense de que en todos ellos esté conectado a la base de datos de ejemplo *TSQLV6ES*:

```
USE TSQLV6ES;
```

Ejecute el código siguiente en la *Conexión 1*, para actualizar una fila en la tabla *Produccion.Productos*, agregando 1 al precio actual de 19 para el producto 2.

```
BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;
```

Para actualizar la fila, la sesión tuvo que adquirir un bloqueo exclusivo y, si la actualización fue exitosa, SQL Server otorgó el bloqueo a la sesión. Recuerden que los bloqueos exclusivos se mantienen hasta el final de la transacción. Debido a que aún no se envió ninguna instrucción **COMMIT TRAN** o **ROLLBACK TRAN**, la transacción permanece abierta y el bloqueo aún se mantiene.

Ejecute el siguiente código en la *Conexión 2* para intentar consultar la misma fila:

```
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

La sesión necesita un bloqueo compartido para leer los datos, pero debido a que la fila está bloqueada exclusivamente por la otra sesión y un bloqueo compartido es incompatible con un bloqueo exclusivo, la sesión está bloqueada y tiene que esperar.

Suponiendo que tal situación de bloqueo ocurra en nuestro sistema y que la sesión bloqueada termine esperando durante mucho tiempo, probablemente desearemos solucionar la situación.

El resto de esta sección proporciona consultas sobre objetos de administración dinámica (objetos que brindan información dinámica sobre varios aspectos de su sistema), incluidas vistas y funciones, que debemos ejecutar desde *Conexión 3* cuando resolvamos la situación de bloqueo.



Para obtener información de bloqueo, incluidos los bloqueos que se otorgan actualmente a las sesiones y los bloqueos que las sesiones están esperando, consultamos la vista de administración dinámica (**DMV**) `sys.dm_tran_locks` en la *Conexión 3*:

```
USE TSQLV6ES;
SELECT -- use * para explorar otros atributos disponibles
request_session_id AS sid,
resource_type AS restype,
resource_database_id AS dbid,
DB_NAME(resource_database_id) AS dbname,
resource_description AS res,
resource_associated_entity_id AS resid,
request_mode AS mode,
request_status AS status
FROM sys.dm_tran_locks;
```

Cuando ejecuto este código en mi sistema (sin otra ventana de consulta abierta), obtengo el siguiente resultado:

sid	restype	dbid	dbname	res	resid	mode	status
70	DATABASE	12	TSQLV6ES		0	S	GRANT
62	DATABASE	12	TSQLV6ES		0	S	GRANT
69	DATABASE	12	TSQLV6ES		0	S	GRANT
62	PAGE	12	TSQLV6ES	1:456	72057594046251008	IX	GRANT
69	PAGE	12	TSQLV6ES	1:456	72057594046251008	IS	GRANT
62	OBJECT	12	TSQLV6ES		1029578706	IX	GRANT
69	OBJECT	12	TSQLV6ES		1029578706	IS	GRANT
62	KEY	12	TSQLV6ES	(61a06abd401c)	72057594046251008	X	GRANT
69	KEY	12	TSQLV6ES	(61a06abd401c)	72057594046251008	S	WAIT

Cada sesión se identifica mediante un ID de sesión único. Podemos determinar el ID de sesión consultando la función **@@SPID**. Si estamos trabajando con SQL Server Management Studio, encontraremos el ID de sesión entre paréntesis a la derecha del nombre de inicio de sesión en la barra de estado en la parte inferior de la ventana de consulta que tiene el foco, y también en el título de la ventana de consulta conectada. Por ejemplo, la imagen siguiente muestra una captura de pantalla de SQL Server Management Studio (SSMS), donde aparece el ID de sesión 62, 69 y 70 a la derecha del nombre de inicio de sesión **sa**.

The screenshot shows the SSMS interface with three query windows open:

- SQLQuery3.sql - loc....TSQLV6ES (sa (70))*
- SQLQuery2.sql - loc...a (69) Executing...
- SQLQuery1.sql - loc....TSQLV6ES (sa (62))*

The code in the first window is:

```
USE TSQLV6ES;
SELECT -- use * para explorar otros atributos disponibles
request_session_id AS sid,
resource_type AS restype,
resource_database_id AS dbid,
DB_NAME(resource_database_id) AS dbname,
resource_description AS res,
resource_associated_entity_id AS resid,
request_mode AS mode,
request_status AS status
FROM sys.dm_tran_locks;
```

Como pueden ver en el resultado de la consulta contra `sys.dm_tran_locks`, tres sesiones (62, 69 y 70) actualmente tienen bloqueos. Pueden ver lo siguiente:



- El tipo de recurso que está bloqueado (por ejemplo, **KEY** para una fila en un índice)
 - El ID de la base de datos en la que está bloqueada, que puede traducir al nombre de la base de datos mediante la función *DB_NAME*
 - El recurso y el ID del recurso.
 - El modo de bloqueo
 - Si se otorgó el bloqueo o si la sesión lo está esperando

Tengan en cuenta que esto es sólo un subconjunto de los atributos de la vista; Les recomiendo que exploren los otros atributos de la vista para saber qué otra información sobre bloqueos está disponible.

En el resultado de mi consulta, puede observar que la sesión 70 está esperando un bloqueo compartido en una fila en la base de datos de ejemplo *TSQLV6ES*. Observen que la sesión 62 tiene un bloqueo exclusivo en la misma fila. Podemos determinar esto observando que ambas sesiones bloquean una fila con los mismos valores de **res** y **resid**. Podemos averiguar qué tabla está involucrada moviéndonos hacia arriba en la jerarquía de bloqueo para la sesión 62 o 69 e inspeccionando los bloqueos de intención en el objeto (tabla) donde reside la fila. Podemos utilizar la función **OBJECT_NAME** para traducir el ID de objeto (1029578706, en este ejemplo) que aparece bajo el atributo **resid** en el bloqueo de objeto. Encontraremos que la tabla involucrada es *Produccion.Productos*.

La vista `sys.dm_tran_locks` nos brinda información sobre los ID de las sesiones involucradas en la cadena de bloqueo. Una cadena de bloqueo es una cadena de dos o más sesiones que están involucradas en la situación de bloqueo. Podríamos tener la sesión x bloqueando la sesión y, la sesión y bloqueando la sesión z, y así sucesivamente, de ahí el uso del término cadena. Para obtener información sobre las conexiones asociadas con esos ID de sesión, consultamos una vista llamada `sys.dm_exec_connections` y filtramos solo los ID de sesión que están involucrados:

```
USE TSQV6ES;
SELECT -- use * para explorar
session_id AS sid,
connect_time,
last_read,
last_write,
most_recent_sql_handle
FROM sys.dm_exec_connections
WHERE session_id IN(62, 69);
```

Tengan en cuenta que los ID de sesión que estaban involucrados en la cadena de bloqueo en mi sistema eran 62 y 69. Dependiendo de qué más estén haciendo en su sistema, es posible que obtengan diferentes IDs.

Cuando ejecuten las consultas que demuestro aquí en sus sistemas, asegúrense de sustituir los ID de sesión con los que encuentren involucrados en sus cadenas de bloqueo.

Esta consulta devuelve el siguiente resultado (dividido en varias partes para fines de visualización aquí):

sid	connect_time	last_read
62	2024-03-08 17:11:03.660	2024-03-08 17:22:11.083
69	2024-03-08 17:21:42.220	2024-03-08 17:22:19.487

La información que brinda esta consulta sobre las conexiones incluye

- La hora en que se conectaron.
 - La hora de su última lectura y escritura.



- Un valor binario que contiene un identificador del lote de SQL más reciente ejecutado por la conexión. Proporciona este identificador como parámetro de entrada a una función de tabla denominada `sys.dm_exec_sql_text` y la función devuelve el lote de código representado por el identificador. Podemos consultar la función de tabla pasando explícitamente el identificador binario, pero probablemente resultará más conveniente usar el operador de tabla `APPLY`, para aplicar la función de tabla a cada fila de conexión como ésta (ejecutar en *Conexión 3*):

```
USE TSQLV6ES;
SELECT session_id, text
FROM sys.dm_exec_connections
CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(62, 69);
```

Cuando ejecutamos esta consulta, obtenemos el siguiente resultado, que muestra el último lote de código invocado por cada conexión involucrada en la cadena de bloqueo:

```
session_id  text
-----
62          USE TSQLV6ES;

BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoId = 2;
69          (@1 tinyint)SELECT [productoId],[preciounitario] FROM [Produccion].[Productos] WHERE [productoId]=@1
```

La sesión bloqueada, 69, muestra la consulta que está esperando porque es lo último que se ejecutó en la sesión. En cuanto al bloqueador, en este ejemplo, podemos ver la declaración que causó el problema, pero tengan en cuenta que en otras situaciones el bloqueador podría seguir funcionando y que lo último que vemos en el código no es necesariamente la declaración que causó el problema.

A partir de SQL Server 2016, podemos usar la función `sys.dm_exec_input_buffer` en lugar de `sys.dm_exec_sql_text` para obtener el código que enviaron las últimas sesiones de interés. La función acepta un ID de sesión y un ID de solicitud (de `sys.dm_exec_requests`, que se describe en breve), o un `NULL` en lugar de un ID de solicitud si el ID no es relevante. Aquí está el código para reemplazar el último ejemplo usando la nueva función:

```
USE TSQLV6ES;
SELECT session_id, event_info
FROM sys.dm_exec_connections
CROSS APPLY sys.dm_exec_input_buffer(session_id, NULL) AS IB
WHERE session_id IN(62, 69);
```

También podemos encontrar mucha información útil sobre las sesiones involucradas en una situación de bloqueo en DMV `sys.dm_exec_sessions`. La siguiente consulta devuelve solo un pequeño subconjunto de los atributos disponibles sobre esas sesiones:

```
USE TSQLV6ES;
SELECT -- use * para explorar
session_id AS sid,
login_time,
host_name,
program_name,
login_name,
nt_user_name,
last_request_start_time,
last_request_end_time
FROM sys.dm_exec_sessions
WHERE session_id IN(62, 69);
```

Esta consulta devuelve el siguiente resultado en este ejemplo, dividido aquí en varias partes:



sid	login_time	host_name	

62	2024-03-08 17:11:03.680	NSIS02	
69	2024-03-08 17:21:42.237	NSIS02	
sid	program_name	login_name	

62	Microsoft SQL Server Management Studio - Query	sa	
69	Microsoft SQL Server Management Studio - Query	sa	
sid	nt_user_name	last_request_start_time	last_request_end_time

62	NULL	2024-03-08 17:22:11.593	2024-03-08 17:22:11.600
69	NULL	2024-03-08 17:22:19.527	2024-03-08 17:21:42.253

Esta salida contiene información como la hora de inicio de la sesión, el nombre del host, el nombre del programa, el nombre de inicio de sesión, el nombre de usuario de Windows (la columna nt_user_name que en este caso es NULL porque estoy usando la cuenta sa), la hora en que comenzó la última solicitud y la hora en que finalizó la última solicitud. Este tipo de información da una buena idea de lo que están haciendo esas sesiones.

Otro DMV que probablemente encontraremos útil para solucionar situaciones de bloqueo es `sys.dm_exec_requests`. Esta vista tiene una fila para cada solicitud activa, incluidas las solicitudes bloqueadas. De hecho, podemos aislar fácilmente las solicitudes bloqueadas porque el atributo `blocking_session_id` es mayor que cero. Por ejemplo, los siguientes filtros de consulta solo bloquearon solicitudes:

```
USE TSQLV6ES;
SELECT -- use * para explorar
session_id AS sid,
blocking_session_id,
command,
sql_handle,
database_id,
wait_type,
wait_time,
wait_resource
FROM sys.dm_exec_requests
WHERE blocking session id > 0
```

Esta consulta devuelve el siguiente resultado, dividido en varias líneas:

Podemos identificar fácilmente las sesiones que participan en la cadena de bloqueo, el recurso en disputa, cuánto tiempo está esperando la sesión bloqueada en milisegundos y más.

Alternativamente, podemos consultar un DMV llamado `sys.dm_os_waiting_tasks`, que sólo tiene tareas que están esperando actualmente. También tiene un atributo llamado `blocking_session_id`, y para solucionar problemas de bloqueo, filtraremos sólo las tareas en espera donde este atributo sea mayor que cero. Parte de la información de esta vista se superpone con la de la vista `sys.dm_exec_requests`, pero proporciona algunos atributos que son únicos con más información, como la descripción del recurso que está en conflicto.



Si necesitamos terminar el bloqueador, por ejemplo, si nos damos cuenta de que, como resultado de un error en la aplicación, la transacción permaneció abierta y nada en la aplicación puede cerrarla, podemos hacerlo usando el comando **KILL <session_id>**. (No lo hagan todavía.)

Anteriormente, mencionamos que, de forma predeterminada, la sesión no tiene establecido un tiempo de espera de bloqueo. Si deseamos restringir la cantidad de tiempo que la sesión espera un bloqueo, podemos configurar una opción de sesión llamada **LOCK_TIMEOUT**. Especificamos un valor en milisegundos, como 5000 para 5 segundos, 0 para un tiempo de espera inmediato y -1 para ningún tiempo de espera (que es el valor predeterminado). Para ver cómo funciona esta opción, primero detenemos la consulta en *Conexión 2* eligiendo Cancelar ejecución de consulta en el menú Consulta (o presionando Alt+Pausa). Tengan en cuenta que, si tenían abierta una transacción explícita, cancelar la consulta en ejecución no cancelaría la transacción automáticamente. Ejecuten el siguiente código para establecer el tiempo de espera de bloqueo en cinco segundos y vuelvan a ejecutar la consulta:

```
USE TSQLV6ES;
SET LOCK_TIMEOUT 5000;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

La consulta aún está bloqueada porque la *Conexión 1* aún no finalizó la transacción de actualización, pero si después de 5 segundos no se satisface la solicitud de bloqueo, SQL Server finaliza la consulta y obtenemos el siguiente error:

**Msg 1222, Level 16, State 51, Line 7
Lock request time out period exceeded.**

Tengan en cuenta que los tiempos de espera de bloqueo no revierten las transacciones.

Para eliminar el valor de tiempo de espera de bloqueo, volvemos a establecerlo en el valor predeterminado (sin tiempo de espera) y emitimos la consulta nuevamente. Ejecuten el siguiente código en *Conexión 2* para lograr esto:

```
USE TSQLV6ES;
SET LOCK_TIMEOUT -1;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Para finalizar la transacción de actualización en la *Conexión 1*, ejecuten el siguiente código desde la *Conexión 3*:

```
KILL 62;
```

Esta sentencia provoca una reversión de la transacción en la *Conexión 1*, lo que significa que el cambio de precio del producto 2 de 19.00 a 20.00 se deshace y se libera el bloqueo exclusivo. Vayan a *Conexión 2*. La consulta que estaba bloqueada hasta ahora puede adquirir el bloqueo, y obtiene los datos después de que se deshace el cambio, es decir, antes del cambio de precio:

productoid	preciounitario
2	19.00

Si intentan cerrar una ventana de consulta mientras una transacción aún está abierta, SSMS le pedirá que elija confirmar o revertir la transacción abierta.



NIVELES DE AISLAMIENTO

Los niveles de aislamiento determinan el nivel de consistencia que obtenemos cuando interactuamos con los datos. En el nivel de aislamiento predeterminado en un producto de caja, un lector usa bloqueos compartidos en los recursos de destino y un escritor usa bloqueos exclusivos. No podemos controlar la forma en que se comportan los escritores en términos de los bloqueos que adquieren y la duración de los bloqueos, pero podemos controlar la forma en que se comportan los lectores. Además, como resultado de controlar el comportamiento de los lectores, podemos tener una influencia implícita en el comportamiento de los escritores. Lo hacemos configurando el nivel de aislamiento, ya sea en el nivel de sesión con una opción de sesión o en el nivel de consulta con una sugerencia de tabla, que es una instrucción que especifica después del nombre de la tabla como parte de la consulta.

SQL Server admite cuatro niveles de aislamiento que se basan en el modelo de bloqueo puro:

READ UNCOMMITTED, READ COMMITTED (el valor predeterminado en un producto de caja de SQL Server), **REPEATABLE READ**, y **SERIALIZABLE**. SQL Server también admite dos niveles de aislamiento que se basan en una combinación de bloqueo y control de versiones de filas: **SNAPSHOT** y **READ COMMITTED SNAPSHOT** (el valor predeterminado en Azure SQL Database). **SNAPSHOT** y **READ COMMITTED SNAPSHOT** son, en cierto sentido, las contrapartes de versiones de fila de **READ COMMITTED** y **SERIALIZABLE**, respectivamente.

Algunos textos se refieren a **READ COMMITTED** y **READ COMMITTED SNAPSHOT** como un nivel de aislamiento con dos tratamientos semánticos diferentes.

Podemos establecer el nivel de aislamiento de toda la sesión con el siguiente comando:

```
SET TRANSACTION ISOLATION LEVEL <isolation name>;
```

Podemos usar una sugerencia de tabla para establecer el nivel de aislamiento de una consulta:

```
SELECT ... FROM <table> WITH (<isolationname>);
```

*No puede establecer explícitamente el nombre del nivel de aislamiento **READ COMMITTED SNAPSHOT** como una opción de sesión o consulta. Para usar este nivel de aislamiento, necesitamos habilitar un indicador de base de datos.*

Con la opción de sesión, especificamos un espacio entre las palabras en caso de que el nombre del nivel de aislamiento esté formado por más de una palabra, como **REPEATABLE READ**. Con la sugerencia de consulta, no especifica un espacio entre las palabras, por ejemplo, **WITH (REPEATABLEREAD)**.

Además, algunos nombres de nivel de aislamiento que se usan como sugerencias de tabla tienen sinónimos. Por ejemplo, **NOLOCK** es el equivalente de especificar **READUNCOMMITTED** y **HOLDLOCK** es el equivalente de especificar **SERIALIZABLE**.

Al cambiar el nivel de aislamiento, afectamos tanto la concurrencia de los usuarios de la base de datos como la consistencia que obtenemos de los datos.

Con los primeros cuatro niveles de aislamiento, cuanto mayor sea el nivel de aislamiento, más estrictos son los bloqueos que solicitan los lectores y mayor es su duración; por lo tanto, cuanto mayor sea el nivel de aislamiento, mayor será la consistencia y menor la concurrencia.

Con los dos niveles de aislamiento basados en el control de versiones de filas, SQL Server puede almacenar versiones confirmadas anteriores de filas en un almacén de versiones. Los lectores no solicitan bloqueos



compartidos; en cambio, si la versión actual de las filas no es la que se supone que deben ver, SQL Server les proporciona una versión anterior.

Las siguientes secciones describen cada uno de los seis niveles de aislamiento admitidos y demuestran su comportamiento.

EL NIVEL DE AISLAMIENTO READ UNCOMMITTED

READ UNCOMMITTED es el nivel de aislamiento más bajo disponible. En este nivel de aislamiento, un lector no solicita un bloqueo compartido. Un lector que no solicita un bloqueo compartido nunca puede estar en conflicto con un escritor que tiene un bloqueo exclusivo. Esto significa que el lector puede leer los cambios no confirmados (también conocidos como lecturas sucias). También significa que el lector no interferirá con un escritor que solicite un bloqueo exclusivo. En otras palabras, un escritor puede cambiar los datos mientras un lector que se ejecuta bajo el nivel de aislamiento **READ UNCOMMITTED** lee datos.

Para ver cómo funciona una lectura no confirmada (lectura sucia), abra dos ventanas de consulta. (Nos referiremos a ellas como *Conexión 1* y *Conexión 2*). Asegúrense de que en todas las conexiones el contexto de su base de datos sea el de la base de datos de muestra *TSQLV6ES*. Para evitar confusiones, asegúrense de que ésta sea la única actividad en la instancia.

Ejecutamos el siguiente código en la *Conexión 1* para abrir una transacción, actualizamos el precio unitario del producto 2 agregando 1,00 a su precio actual (19,00) y luego consultamos la fila del producto:

```
USE TSQLV6ES;
BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;

SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Tengan en cuenta que la transacción permanece abierta, lo que significa que la fila del producto está bloqueada exclusivamente por la *Conexión 1*. El código en la *Conexión 1* devuelve el siguiente resultado que muestra el nuevo precio del producto:

```
productoid  preciounitario
-----  -----
2          20.00
```

En la *Conexión 2*, ejecutamos el siguiente código para establecer el nivel de aislamiento en **READ UNCOMMITTED** y consultamos la fila del producto 2:

```
USE TSQLV6ES;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Debido a que la consulta no solicitó un bloqueo compartido, no estaba en conflicto con la otra transacción. Esta consulta devolvió el estado de la fila después del cambio, aunque el cambio no se confirmó:

```
productoid  preciounitario
-----  -----
2          20.00
```



Tengan en cuenta que la Conexión 1 podría aplicar más cambios a la fila más adelante en la transacción o incluso retroceder en algún momento. Por ejemplo, ejecuten el siguiente código en la Conexión 1 para revertir la transacción:

```
ROLLBACK TRAN;
```

Esta reversión deshace la actualización del producto 2, cambiando su precio nuevamente a 19.00. El valor 20,00 que obtuvo el lector nunca se confirmó. Ese es un ejemplo de una **lectura sucia**.

EL NIVEL DE AISLAMIENTO READ COMMITTED

Si deseamos evitar que los lectores lean cambios no confirmados, debemos usar un nivel de aislamiento más fuerte. El nivel de aislamiento más bajo que evita lecturas sucias es **READ COMMITTED**, que también es el nivel de aislamiento predeterminado en SQL Server (en el producto de caja). Como su nombre lo indica, este nivel de aislamiento permite a los lectores leer solo los cambios confirmados. Evita lecturas no confirmadas al requerir que un lector obtenga un bloqueo compartido. Esto significa que, si un escritor tiene un bloqueo exclusivo, la solicitud de bloqueo compartido del lector entrará en conflicto con el escritor y tendrá que esperar. Tan pronto como el escritor confirma la transacción, el lector puede obtener su bloqueo compartido, pero lo que lee son necesariamente sólo cambios confirmados.

El siguiente ejemplo demuestra que, en este nivel de aislamiento, un lector sólo puede leer los cambios confirmados.

Ejecutemos el siguiente código en la Conexión 1 para abrir una transacción, actualizamos el precio del producto 2 y consultamos la fila para mostrar el nuevo precio:

```
USE TSQLV6ES;
BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;

SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

La consulta devuelve el siguiente resultado:

productoid	preciounitario
2	20.00

La Conexión 1 ahora bloquea la fila para el producto 2 exclusivamente.

Ejecutamos el siguiente código en la Conexión 2 para establecer el nivel de aislamiento de la sesión en **READ COMMITTED** y consultamos la fila del producto 2:

```
USE TSQLV6ES;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Tengan en cuenta que este nivel de aislamiento es el predeterminado, por lo que, a menos que hayan cambiado previamente el nivel de aislamiento de la sesión, no necesitamos establecerlo explícitamente. La declaración **SELECT** está actualmente bloqueada porque necesita un bloqueo compartido para poder leer la fila, y esta solicitud de bloqueo compartido está en conflicto con el bloqueo exclusivo que tiene el escritor en la Conexión 1.



A continuación, ejecutamos el siguiente código en la Conexión 1 para confirmar la transacción:

```
COMMIT TRAN;
```

Ahora vayan a Conexión 2 y observen que obtenemos el siguiente resultado:

productoid	preciounitario
2	20.00

A diferencia de **READ UNCOMMITTED**, en el nivel de aislamiento **READ COMMITTED**, no obtenemos lecturas sucias. En su lugar, solo podemos leer los cambios confirmados.

En cuanto a la duración de los bloqueos, en el nivel de aislamiento **READ COMMITTED**, un lector retiene el bloqueo compartido sólo hasta que termina con el recurso. No mantiene el bloqueo hasta el final de la transacción; de hecho, ni siquiera mantiene el bloqueo hasta el final de la instrucción. Esto significa que, entre dos lecturas del mismo recurso de datos en la misma transacción, no se mantiene ningún bloqueo en el recurso. Por lo tanto, otra transacción puede modificar el recurso entre esas dos lecturas y el lector puede obtener valores diferentes en cada lectura. Este fenómeno se denomina **lecturas no repetibles** o **análisis inconsistente**. Para muchas aplicaciones, este fenómeno es aceptable, pero para algunas no lo es.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza en cualquiera de las conexiones abiertas:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19.00
WHERE productoid = 2;
```

Además, asegúrense de que todas las transacciones abiertas en todas las ventanas estén cerradas.

EL NIVEL DE AISLAMIENTO REPEATABLE READ

Si deseamos asegurarnos de que nadie pueda cambiar los valores entre las lecturas que tienen lugar en la misma transacción, debemos ascender en los niveles de aislamiento a **REPEATABLE READ**. En este nivel de aislamiento, un lector no sólo necesita un bloqueo compartido para poder leer, sino que también mantiene el bloqueo hasta el final de la transacción. Esto significa que tan pronto como el lector adquiere un bloqueo compartido en un recurso de datos para leerlo, nadie puede obtener un bloqueo exclusivo para modificar ese recurso hasta que el lector finalice la transacción. De esta manera, tiene la garantía de obtener **lecturas repetibles** o **análisis consistentes**.

El siguiente ejemplo demuestra cómo obtener lecturas repetibles. Ejecutamos el siguiente código en la Conexión 1 para establecer el nivel de aislamiento de la sesión en **REPEATABLE READ**, abrimos una transacción y leemos la fila del producto 2:

```
USE TSQLV6ES;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Este código devuelve el siguiente resultado que muestra el precio actual del producto 2:

productoid	preciounitario
2	19.00



La Conexión 1 aún mantiene un bloqueo compartido en la fila del producto 2 porque en **REPEATABLE READ**, los bloqueos compartidos se mantienen hasta el final de la transacción. Ejecutamos el siguiente código desde Conexión 2 para intentar modificar la fila del producto 2:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;
```

Tengan en cuenta que el intento está bloqueado porque la solicitud del modificador de un bloqueo exclusivo está en conflicto con el bloqueo compartido otorgado por el lector. Si el lector se ejecutaba bajo el nivel de aislamiento **READ UNCOMMITTED** o **READ COMMITTED**, no mantendría el bloqueo compartido en este punto y el intento de modificar la fila sería exitoso.

De vuelta en la Conexión 1, ejecutamos el siguiente código para leer la fila del producto 2 por segunda vez y confirmamos la transacción:

```
USE TSQLV6ES;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
COMMIT TRAN;
```

Este código devuelve la siguiente salida:

productoid	preciounitario
2	19.00

Observemos que la segunda lectura obtuvo el mismo precio unitario para el producto 2 que la primera lectura. Ahora que se ha confirmado la transacción del lector y se ha liberado el bloqueo compartido, el modificador en la Conexión 2 puede obtener el bloqueo exclusivo que estaba esperando y actualizar la fila.

Otro fenómeno evitado por **REPEATABLE READ**, pero no por niveles de aislamiento más bajos se denomina **actualización perdida**. Una actualización perdida ocurre cuando dos transacciones leen un valor, hacen cálculos basados en lo que leen y luego actualizan el valor. Debido a que en los niveles de aislamiento inferiores a **REPEATABLE READ** no se mantiene ningún bloqueo en el recurso después de la lectura, ambas transacciones pueden actualizar el valor, y la última transacción que actualice el valor "gana", sobrescribiendo la actualización de la otra transacción. En **REPEATABLE READ**, ambos lados mantienen sus bloqueos compartidos después de la primera lectura, por lo que ninguno puede adquirir un bloqueo exclusivo más tarde para actualizar. La situación da como resultado un interbloqueo y se evita el conflicto de actualización.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19
WHERE productoid = 2;
```

EL NIVEL DE AISLAMIENTO SERIALIZABLE

Al ejecutarse bajo el nivel de aislamiento de **REPEATABLE READ**, los lectores mantienen los bloqueos compartidos hasta el final de la transacción. Por lo tanto, tenemos la garantía de obtener una lectura repetible de las filas que leímos la primera vez en la transacción. Sin embargo, la transacción bloquea sólo los recursos (por ejemplo, filas) que la consulta encontró la primera vez que se ejecutó, no las filas que no estaban allí cuando se ejecutó la consulta. Por lo tanto, una segunda lectura en la misma transacción también podría devolver filas nuevas. Esas



nuevas filas se denominan fantasmas, y tales lecturas se denominan **lecturas fantasmas**. Esto sucede si, entre las lecturas, otra transacción inserta nuevas filas que satisfacen el filtro de consulta del lector.

Para evitar lecturas fantasmas, debemos subir los niveles de aislamiento a **SERIALIZABLE**.

En su mayor parte, el nivel de aislamiento **SERIALIZABLE** se comporta de manera similar a **REPEATABLE READ**: es decir, requiere que un lector obtenga un bloqueo compartido para poder leer, y mantiene el bloqueo hasta el final de la transacción. Pero el nivel de aislamiento **SERIALIZABLE** agrega otra faceta: lógicamente, este nivel de aislamiento hace que un lector bloquee todo el rango de claves que califican para el filtro de la consulta. Esto significa que el lector bloquea no sólo las filas existentes que califican para el filtro de la consulta, sino también las futuras. O, más exactamente, bloquea los intentos realizados por otras transacciones para agregar filas que califiquen para el filtro de consulta del lector.

El siguiente ejemplo demuestra que el nivel de aislamiento **SERIALIZABLE** evita las lecturas fantasmas. Ejecutamos el siguiente código en la *Conexión 1* para establecer el nivel de aislamiento de la transacción en **SERIALIZABLE**, abrimos una transacción y consultamos todos los productos con la categoría 1:

```
USE TSQLV6ES;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
SELECT productoid, nombreproducto, categoriaid, preciounitario
FROM Produccion.Productos
WHERE categoriaid = 1;
```

Obtenemos el siguiente resultado, que muestra 12 productos en la categoría 1:

productoid	nombreproducto	categoriaid	preciounitario
1	Product HHYDP	1	18.00
2	Product RECZE	1	19.00
24	Product QOGNU	1	4.50
34	Product SWNJY	1	14.00
35	Product NEVTJ	1	18.00
38	Product QDOMO	1	263.50
39	Product LSOFL	1	18.00
43	Product ZZZHR	1	46.00
67	Product XLXQF	1	14.00
70	Product TOONT	1	15.00
75	Product BWRLG	1	7.75
76	Product JYGFE	1	18.00

Desde *Conexión 2*, ejecutamos el siguiente código en un intento de insertar un nuevo producto con categoría 1:

```
USE TSQLV6ES;
INSERT INTO Produccion.Productos
(nombreproducto, proveedorid, categoriaid,
preciounitario, discontinuado)
VALUES ('Product ABCDE', 1, 1, 20.00, 0);
```

En todos los niveles de aislamiento inferiores a **SERIALIZABLE**, dicho intento tendría éxito. En el nivel de aislamiento **SERIALIZABLE** se bloquea el intento.

De vuelta en la *Conexión 1*, ejecute el siguiente código para consultar los productos con la categoría 1 por segunda vez y confirmamos la transacción:



```
USE TSQLV6ES;
SELECT productoid, nombreproducto, categoriaid, preciouunitario
FROM Produccion.Productos
WHERE categoriaid = 1;
COMMIT TRAN;
```

Obtenemos el mismo resultado que antes, sin fantasmas. Ahora que la transacción del lector está confirmada y se libera el bloqueo de rango de claves compartidas, el modificador en la *Conexión 2* puede obtener el bloqueo exclusivo que estaba esperando, insertar la fila y realizar la confirmación automática. Si en este punto volvemos a ejecutar el código para consultar los productos en la categoría 1 de una nueva transacción, obtendremos 13 filas en la salida.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza:

```
USE TSQLV6ES;
DELETE FROM Produccion.Productos
WHERE productoid > 77;
```

Ejecuten el siguiente código en todas las conexiones abiertas para restablecer el nivel de aislamiento al valor predeterminado:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

NIVELES DE AISLAMIENTO BASADOS EN EL CONTROL DE VERSIONES DE FILAS

Con la tecnología de control de versiones de filas, SQL Server puede almacenar versiones anteriores de filas confirmadas en un almacén de versiones. Si la característica recuperación acelerada de la base de datos (**ADR**), que mencionamos con anterioridad al hablar de la durabilidad, no está habilitada en la base de datos, el almacén de versiones reside en la base de datos tempdb. Si **ADR** está habilitado, el almacén de versiones reside en la base de datos del usuario en cuestión. SQL Server admite dos niveles de aislamiento, denominados **SNAPSHOT** y **READ COMMITTED SNAPSHOT**, que se basan en esta tecnología de control de versiones de filas. El nivel de aislamiento **SNAPSHOT** es lógicamente similar al nivel de aislamiento **SERIALIZABLE** en términos de los tipos de problemas de coherencia que pueden o no ocurrir; el nivel de aislamiento de **READ COMMITTED SNAPSHOT** es similar al nivel de aislamiento de **READ COMMITTED**. Sin embargo, los lectores que utilizan niveles de aislamiento basados en el control de versiones de filas no adquieren bloqueos compartidos, por lo que no esperan a que los datos solicitados se bloqueen de forma exclusiva. En otras palabras, los lectores no bloquean a los escritores y los escritores no bloquean a los lectores. Los lectores aún obtienen niveles de consistencia similares a **SERIALIZABLE** y **READ COMMITTED**. SQL Server proporciona a los lectores una versión anterior de la fila si la versión actual no es la que se supone que deben ver.

Tengan en cuenta que si habilitamos cualquiera de los niveles de aislamiento basados en versiones de filas (que están habilitados en Azure SQL Database de forma predeterminada), las instrucciones **DELETE** y **UPDATE** deben copiar la versión de la fila antes del cambio en el almacén de versiones; Las declaraciones **INSERT** no necesitan escribir nada en el almacén de versiones, porque no existe una versión anterior de la fila. Pero es importante tener en cuenta que habilitar cualquiera de los niveles de aislamiento que se basan en el control de versiones de filas puede tener un impacto negativo en el rendimiento de las actualizaciones y eliminaciones. El rendimiento de los lectores suele mejorar, a veces de forma espectacular, porque no adquieren bloqueos compartidos y no necesitan esperar cuando los datos están bloqueados de forma exclusiva o su versión no es la esperada.

EL NIVEL DE AISLAMIENTO SNAPSHOT

Bajo el nivel de aislamiento **SNAPSHOT**, cuando el lector está leyendo datos, se garantiza que obtendrá la última versión confirmada de la fila que estaba disponible cuando comenzó la transacción. Esto significa que tiene la garantía de obtener lecturas confirmadas y lecturas repetibles, y también tiene la garantía de no obtener lecturas



fantasmas, al igual que en el nivel de aislamiento **SERIALIZABLE**. Pero en lugar de utilizar bloqueos compartidos, este nivel de aislamiento se basa en el control de versiones de filas.

Como se mencionó, el control de versiones de fila incurre en una penalización de rendimiento, principalmente cuando se actualizan y eliminan datos, independientemente de si la modificación se ejecuta o no desde una sesión que se ejecuta bajo uno de los niveles de aislamiento basados en el control de versiones de fila. Por este motivo, para permitir que sus transacciones funcionen con el nivel de aislamiento **SNAPSHOT** en una instancia de producto de caja de SQL Server (un comportamiento que está habilitado de forma predeterminada en Azure SQL Database), primero debemos habilitar la opción en el nivel de base de datos ejecutando el siguiente código en cualquier ventana de consulta abierta:

```
ALTER DATABASE TSQLV6ES SET ALLOW_SNAPSHOT_ISOLATION ON;
```

El siguiente ejemplo demuestra el comportamiento del nivel de aislamiento **SNAPSHOT**. Ejecutamos el siguiente código desde la *Conexión 1* para abrir una transacción, actualizamos el precio del producto 2 agregando 1,00 a su precio actual de 19,00 y consultamos la fila del producto para mostrar el nuevo precio:

```
USE TSQLV6ES;
BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;

SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Aquí, el resultado de este código muestra que el precio del producto se actualizó a 20,00:

productoid	preciounitario
2	20.00

Tengan en cuenta que incluso si la transacción en la *Conexión 1* se ejecuta bajo el nivel de aislamiento **READ COMMITTED**, SQL Server tiene que copiar la versión de la fila antes de la actualización (con el precio de 19,00) en el almacén de versiones. Esto se debe a que el nivel de aislamiento **SNAPSHOT** está habilitado en el nivel de la base de datos. Si alguien inicia una transacción utilizando el nivel de aislamiento **SNAPSHOT**, esa sesión puede solicitar la versión antes de la actualización. Por ejemplo, ejecutamos el siguiente código desde la *Conexión 2* para establecer el nivel de aislamiento en **SNAPSHOT**, abrimos una transacción y consultamos la fila del producto 2:

```
USE TSQLV6ES;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRAN;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Si la transacción estuviera bajo el nivel de aislamiento **SERIALIZABLE**, la consulta sería bloqueada. Pero debido a que se ejecuta en **SNAPSHOT**, obtiene la última versión confirmada de la fila que estaba disponible cuando comenzó la transacción. Esa versión (con el precio de 19,00) no es la versión actual (con el precio de 20,00), por lo que SQL Server extrae la versión adecuada del almacén de versiones y el código devuelve el siguiente resultado:

productoid	preciounitario
2	19.00

Volvamos a la *Conexión 1* y confirmamos la transacción que modificó la fila:



COMMIT TRAN;

En este punto, la versión actual de la fila con el precio de 20,00 es una versión confirmada. Sin embargo, si volvemos a leer los datos en la *Conexión 2*, aún deberíamos obtener la última versión confirmada de la fila que estaba disponible cuando comenzó la transacción (con un precio de 19,00). Ejecutamos el siguiente código en la *Conexión 2* para volver a leer los datos y luego confirmamos la transacción:

```
USE TSQLV6ES;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
COMMIT TRAN;
```

Como era de esperar, obtenemos el siguiente resultado con un precio de 19,00:

```
productoid  preciounitario
----- -----
2           19.00
```

Ejecutamos el siguiente código en la *Conexión 2* para abrir una nueva transacción, consultar los datos y confirmar la transacción:

```
USE TSQLV6ES;
BEGIN TRAN
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
COMMIT TRAN;
```

Esta vez, la última versión confirmada de la fila que estaba disponible cuando comenzó la transacción es la que tiene un precio de 20,00. Por lo tanto, obtenemos el siguiente resultado:

```
productoid  preciounitario
----- -----
2           20.00
```

Ahora que ninguna transacción necesita la versión de la fila con el precio de 19,00, la próxima vez que se ejecute un subproceso de limpieza, puede eliminar la versión de la fila del almacén de versiones. Como podemos imaginar, las transacciones muy largas impiden que SQL Server pueda limpiar las versiones de fila y pueden hacer que el almacén de versiones se expanda.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19.00
WHERE productoid = 2;
```

DETECCIÓN DE CONFLICTOS

El nivel de aislamiento **SNAPSHOT** evita conflictos de actualización, pero a diferencia de los niveles de aislamiento **REPEATABLE READ** y **SERIALIZABLE** que lo hacen generando un punto muerto, el nivel de aislamiento **SNAPSHOT** genera un error más específico, lo que indica que se detectó un conflicto de actualización. El nivel de aislamiento **SNAPSHOT** puede detectar conflictos de actualización examinando el almacén de versiones. Podemos averiguar si otra transacción modificó los datos entre una lectura y una escritura que tuvo lugar en la transacción. En otras palabras, **REPEATABLE READ**, **SERIALIZABLE** y **SNAPSHOT** evitan la pérdida de actualizaciones; sin embargo, sólo **SNAPSHOT** genera un error específico que indica que detectó un conflicto de actualización en lugar del error de interbloqueo más generalizado, que también podría ocurrir por otras razones. Por esta razón, la Tabla 3, en la



sección “Resumen de los niveles de aislamiento”, indica que el aislamiento **SNAPSHOT** detecta conflictos de actualización, pero **REPEATABLE READ** y **SERIALIZABLE** no.

El siguiente ejemplo muestra un escenario sin conflicto de actualización, seguido de un ejemplo de un escenario con un conflicto de actualización.

Ejecutamos el siguiente código en la *Conexión 1* para establecer el nivel de aislamiento de la transacción en **SNAPSHOT**, abrimos una transacción y leemos la fila del producto 2:

```
USE TSQLV6ES;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRAN;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Obtenemos la siguiente salida:

```
productoid  preciounitario
-----
2           19.00
```

Asumiendo que hicimos algunos cálculos basados en lo que leímos, ejecutamos el siguiente código mientras aún estamos en la *Conexión 1* para actualizar el precio del producto que consultamos con anterioridad a 20.00 y confirmamos la transacción:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 20.00
WHERE productoid = 2;
COMMIT TRAN;
```

Ninguna otra transacción modificó la fila entre su lectura, cálculo y escritura; por lo tanto, no hubo conflicto de actualización y SQL Server permitió que se realizara la actualización.

Ejecutamos el siguiente código para modificar el precio del producto 2 de nuevo a 19,00:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19.00
WHERE productoid = 2;
```

A continuación, ejecutamos de nuevo el siguiente código en la *Conexión 1* para abrir una transacción y leer la fila del producto 2:

```
USE TSQLV6ES;
BEGIN TRAN;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Obtenemos el siguiente resultado, que indica que el precio del producto es 19,00:

```
productoid  preciounitario
-----
2           19.00
```

Esta vez, ejecutamos el siguiente código en la *Conexión 2* para actualizar el precio del producto 2 a 25,00:



```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 25.00
WHERE productoid = 2;
```

Supongamos que realizamos cálculos en la *Conexión 1* según el precio de 19,00 que leímos. Según sus cálculos, intentamos actualizar el precio del producto a 20,00 en la *Conexión 1*:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 20.00
WHERE productoid = 2;
```

SQL Server detectó que esta vez otra transacción modificó los datos entre su lectura y escritura; por lo tanto, falla la transacción con el siguiente error:

```
Msg 3960, Level 16, State 2, Line 7
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot isolation to access table 'Produccion.Productos' directly or indirectly in database 'TSQLV6ES' to update, delete, or insert the row that has been modified or deleted by another transaction. Retry the transaction or change the isolation level for the update/delete statement.
```

Por supuesto, podemos usar el código de manejo de errores para volver a intentar toda la transacción cuando se detecta un conflicto de actualización.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19.00
WHERE productoid = 2;
```

Cierren todas las conexiones. Tengan en cuenta que, si no se cierran todas las conexiones, es posible que los resultados del ejemplo no coincidan con los de los ejemplos del apunte.

EL NIVEL DE AISLAMIENTO DE READ COMMITTED SNAPSHOT

El nivel de aislamiento de **READ COMMITTED SNAPSHOT** también se basa en el control de versiones de filas. Se diferencia del nivel de aislamiento **SNAPSHOT** en que, en lugar de proporcionar al lector una vista coherente de los datos a nivel de transacción, proporciona al lector una vista coherente de los datos a nivel de instrucción. El nivel de aislamiento **READ COMMITTED SNAPSHOT** tampoco detecta conflictos de actualización. Esto da como resultado un comportamiento lógico similar al nivel de aislamiento **READ COMMITTED**, excepto que los lectores no adquieren bloqueos compartidos y no esperan cuando el recurso solicitado está bloqueado de forma exclusiva. Si en **READ COMMITTED SNAPSHOT** deseamos que un lector adquiera un bloqueo compartido, debemos agregar una sugerencia de tabla llamada **READCOMMITTEDLOCK** a las instrucciones **SELECT**, como en **SELECT * FROM dbo.T1 WITH (READCOMMITTEDLOCK)**.

Para habilitar el uso del nivel de aislamiento **READ COMMITTED SNAPSHOT** en un producto de caja de SQL Server (un comportamiento que está habilitado de forma predeterminada en Azure SQL Database), debemos activar una opción de base de datos denominada **READ_COMMITTED_SNAPSHOT**. Ejecutamos el siguiente código para habilitar esta opción en la base de datos TSQLV6ES:

```
ALTER DATABASE TSQLV6ES SET READ_COMMITTED_SNAPSHOT ON;
```

Tengan en cuenta que para que este código se ejecute correctamente, necesitamos acceso exclusivo a la base de datos TSQLV6ES.



Un aspecto interesante de habilitar este indicador de base de datos es que, a diferencia del nivel de aislamiento **SNAPSHOT**, este indicador en realidad cambia el significado, o la semántica, del nivel de aislamiento **READ COMMITTED** a **READ COMMITTED SNAPSHOT**. Esto significa que cuando este indicador de base de datos está activado, a menos que cambie explícitamente el nivel de aislamiento de la sesión, **READ COMMITTED SNAPSHOT** es el valor predeterminado.

Para una demostración del uso del nivel de aislamiento **READ COMMITTED SNAPSHOT**, abrimos dos conexiones. Ejecutamos el siguiente código en la *Conexión 1* para abrir una transacción, actualizamos la fila del producto 2 y leemos la fila, dejando la transacción abierta:

```
USE TSQLV6ES;
BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;

SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Obtenemos el siguiente resultado, que indica que el precio del producto se cambió a 20,00:

productoid	preciounitario
2	20.00

En la *Conexión 2*, abrimos una transacción y leemos la fila del producto 2, dejando la transacción abierta:

```
USE TSQLV6ES;
BEGIN TRAN;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
```

Obtenemos la última versión confirmada de la fila que estaba disponible cuando comenzó la sentencia (19.00):

productoid	preciounitario
2	19.00

Ejecutamos el siguiente código en la *Conexión 1* para confirmar la transacción:

```
COMMIT TRAN;
```

Ahora ejecutamos el código en la *Conexión 2* para leer la fila del producto 2 nuevamente y confirmamos la transacción:

```
USE TSQLV6ES;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
COMMIT TRAN;
```

Si este código se ejecutara con el nivel de aislamiento **SNAPSHOT**, obtendría un precio de 19,00; sin embargo, debido a que el código se ejecuta bajo el nivel de aislamiento **READ COMMITTED SNAPSHOT**, obtenemos la última versión confirmada de la fila que estaba disponible cuando comenzó la instrucción (20.00) y no cuando comenzó la transacción (19.00):



productoid	preciounitario
-----	-----
2	20.00

Recuerden que este fenómeno se denomina **lectura no repetible** o **análisis inconsistente**.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19.00
WHERE productoid = 2;
```

Cierren todas las conexiones. Abran una nueva conexión y ejecuten el siguiente código para deshabilitar los niveles de aislamiento que se basan en el control de versiones de filas en la base de datos TSQLV6ES:

```
ALTER DATABASE TSQLV6ES SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE TSQLV6ES SET READ_COMMITTED_SNAPSHOT OFF;
```

RESUMEN DE LOS NIVELES DE AISLAMIENTO

La Tabla 3 proporciona un resumen de los problemas de coherencia lógica que pueden ocurrir o no en cada nivel de aislamiento, e indica si el nivel de aislamiento detecta conflictos de actualización y si el nivel de aislamiento utiliza el control de versiones de fila.

TABLA 3

Nivel de aislamiento	¿Permite lecturas no confirmadas?	¿Permite lecturas no repetibles?	¿Permite actualizaciones perdidas?	¿Permite lecturas fantasma?	¿Detecta conflictos de actualización?	¿Utiliza el control de versiones de filas?
READ UNCOMMITTED	Sí	Sí	Sí	Sí	No	No
READ COMMITTED	No	Sí	Sí	Sí	No	No
READ COMMITTED SNAPSHOT	No	Sí	Sí	Sí	No	Sí
REPEATABLE READ	No	No	No	Sí	No	No
SERIALIZABLE	No	No	No	No	No	No
SNAPSHOT	No	No	No	No	Sí	Sí

INTERBLOQUEOS (DEADLOCKS)

Un interbloqueo es una situación en la que dos o más sesiones se bloquean entre sí. Un ejemplo de interbloqueo de dos sesiones es cuando la sesión A bloquea la sesión B y la sesión B bloquea la sesión A. Un ejemplo de interbloqueo que implica más de dos sesiones es cuando la sesión A bloquea la sesión B, la sesión B bloquea la sesión C y la sesión C bloquea sesión A. En cualquiera de estos casos, SQL Server detecta el interbloqueo e interviene al terminar una de las transacciones. Si SQL Server no interviniere, las sesiones involucradas permanecerían bloqueadas para siempre.

A menos que se especifique lo contrario, SQL Server opta por finalizar la transacción que realizó el menor trabajo (en función de la actividad escrita en el registro de transacciones), porque revertir el trabajo de esa transacción es la opción más económica. Sin embargo, con SQL Server podemos establecer una opción de sesión llamada **DEADLOCK_PRIORITY** en uno de los 21 valores en el rango de -10 a 10. La sesión con la prioridad de interbloqueo más baja se elige como la "víctima" de interbloqueo independientemente de cuánto trabajo se realice; en caso de empate, la cantidad de trabajo se utiliza como criterio de desempate. Si se estima la misma cantidad de trabajo para todas las sesiones involucradas, el sistema elige a la víctima al azar.

El siguiente ejemplo demuestra un interbloqueo simple. Despues de presentar el ejemplo, explicaremos cómo podemos mitigar las ocurrencias de interbloqueo en el sistema.



Abrimos dos conexiones y asegúrense de estar conectados a la base de datos TSQLV6ES en ambas. Ejecutamos el siguiente código en la *Conexión 1* para abrir una nueva transacción, actualizamos una fila en la tabla *Produccion.Productos* para el producto 2 y dejamos la transacción abierta:

```
USE TSQLV6ES;
BEGIN TRAN;
UPDATE Produccion.Productos
SET preciounitario += 1.00
WHERE productoid = 2;
```

Ejecutamos el siguiente código en la *Conexión 2* para abrir una nueva transacción, actualizamos una fila en la tabla *Ventas.DetallesOrden* para el producto 2 y dejamos la transacción abierta:

```
USE TSQLV6ES;
BEGIN TRAN;
UPDATE Ventas.DetallesOrden
SET preciounitario += 1.00
WHERE productoid = 2;
```

En este punto, la transacción en la *Conexión 1* mantiene un bloqueo exclusivo en la fila del producto 2 en la tabla *Produccion.Productos* y la transacción en la *Conexión 2* ahora mantiene bloqueos en las filas del producto 2 en la tabla *Ventas.DetallesOrden*. Ambas consultas tienen éxito y aún no se ha producido ningún bloqueo.

Ejecutamos el siguiente código en la *Conexión 1* para intentar consultar las filas del producto 2 en la tabla *Ventas.DetallesOrden* y confirmamos la transacción:

```
USE TSQLV6ES;
SELECT ordenid, productoid, preciounitario
FROM Ventas.DetallesOrden
WHERE productoid = 2;
COMMIT TRAN;
```

La transacción en la *Conexión 1* necesita un bloqueo compartido para poder realizar su lectura. Debido a que la otra transacción tiene un bloqueo exclusivo en el mismo recurso, la transacción en la *Conexión 1* está bloqueada. En este punto, tenemos una situación de bloqueo, aún no un punto muerto. Por supuesto, existe la posibilidad de que la *Conexión 2* finalice la transacción, liberando todos los bloqueos y permitiendo que la transacción en la *Conexión 1* obtenga los bloqueos solicitados.

A continuación, ejecutamos el siguiente código en la *Conexión 2* para intentar consultar la fila del producto 2 en la tabla *Produccion.Productos* y confirmamos la transacción:

```
USE TSQLV6ES;
SELECT productoid, preciounitario
FROM Produccion.Productos
WHERE productoid = 2;
COMMIT TRAN;
```

Para poder realizar su lectura, la transacción en la *Conexión 2* necesita un bloqueo compartido en la fila del producto 2 en la tabla *Produccion.Productos*, por lo que esta solicitud ahora está en conflicto con el bloqueo exclusivo que mantiene la *Conexión 1* en el mismo recurso. Cada una de las sesiones bloquea a la otra: tenemos un punto muerto. SQL Server identifica el interbloqueo (normalmente en unos segundos), elige una de las sesiones involucradas como víctima del interbloqueo y finaliza su transacción con el siguiente error:

Msg 1205, Level 13, State 51, Line 7
Transaction (Process ID 51) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.



En este ejemplo, SQL Server optó por finalizar la transacción en la *Conexión 1* (que se muestra aquí como ID de proceso 51). Debido a que no establecimos una prioridad de interbloqueo y ambas transacciones realizaron una cantidad de trabajo similar, cualquiera de las transacciones podría haberse cancelado.

Los interbloqueos son costosos porque implican deshacer el trabajo que ya se ha hecho y luego, generalmente con alguna lógica de manejo de errores, rehacer el trabajo. Podemos seguir algunas prácticas para mitigar los casos de interbloqueo en el sistema.

Obviamente, cuanto más largas son las transacciones, más bloqueos se mantienen, lo que aumenta la probabilidad de puntos muertos. Debemos tratar de mantener las transacciones lo más cortas posible, eliminando actividades de la transacción que lógicamente no se supone que sean parte de la misma unidad de trabajo. Por ejemplo, ¡no usen transacciones que requieran la entrada del usuario para finalizar!

Un punto muerto típico, también llamado punto muerto de abrazo mortal, ocurre cuando las transacciones acceden a los recursos en orden inverso. En el ejemplo que acabamos de dar, la *Conexión 1* primero accedió a una fila en *Produccion.Productos* y luego accedió a una fila en *Ventas.DetallesOrden*, mientras que la *Conexión 2* primero accedió a una fila en *Ventas.DetallesOrden* y luego accedió a una fila en *Produccion.Productos*. Este tipo de interbloqueo no puede ocurrir si ambas transacciones acceden a los recursos en el mismo orden. Al intercambiar el orden en una de las transacciones, podemos evitar que ocurra este tipo de interbloqueo, suponiendo que no genere una diferencia lógica para la aplicación.

El ejemplo de interbloqueo tiene un conflicto lógico real porque ambos lados intentan acceder a las mismas filas. Sin embargo, los interbloqueos a menudo ocurren cuando no hay un conflicto lógico real, debido a la falta de una buena indexación para admitir filtros de consulta. Por ejemplo, supongamos que ambas sentencias en la transacción en la *Conexión 2* filtraron el producto 5. Ahora que las sentencias en la *Conexión 1* manejan el producto 2 y las declaraciones en la *Conexión 2* manejan el producto 5, no debería haber ningún conflicto. Sin embargo, si no tenemos índices definidos en la columna *productoid* en las tablas para admitir el filtro, SQL Server tiene que escanear (y bloquear) todas las filas de la tabla. Esto, por supuesto, puede conducir a un punto muerto. En resumen, un buen diseño de índice puede ayudar a mitigar la aparición de interbloqueos que no tienen un conflicto lógico real.

Otra opción que considerar para mitigar las ocurrencias de puntos muertos es la elección del nivel de aislamiento. Las declaraciones *SELECT* en el ejemplo necesitaban bloqueos compartidos porque se ejecutaban bajo el nivel de aislamiento **READ COMMITTED**. Si utilizamos el nivel de aislamiento **READ COMMITTED SNAPSHOT**, los lectores no necesitarán bloqueos compartidos y se pueden eliminar los interbloqueos que evolucionan debido a la participación de bloqueos compartidos.

Cuando hayan terminado, ejecuten el siguiente código para la limpieza en cualquier conexión:

```
USE TSQLV6ES;
UPDATE Produccion.Productos
SET preciounitario = 19.00
WHERE productoid = 2;
UPDATE Ventas.DetallesOrden
SET preciounitario = 19.00
WHERE productoid = 2
AND ordenid >= 10500;
UPDATE Ventas.DetallesOrden
SET preciounitario = 15.20
WHERE productoid = 2
AND ordenid < 10500;
```



CONCLUSIÓN

Esta sección presentó las transacciones y la concurrencia. Describimos qué son las transacciones y cómo las administra SQL Server. Explicamos cómo SQL Server aísla los datos a los que accede una transacción del uso inconsistente de otras transacciones y cómo solucionar situaciones de bloqueo. Describimos cómo podemos controlar el nivel de consistencia que obtenemos de los datos eligiendo un nivel de aislamiento y el impacto que tiene su elección en la concurrencia. Describimos cuatro niveles de aislamiento que no se basan en el control de versiones de filas y dos que sí lo hacen. Finalmente, cubrimos los interbloqueos y explicamos las prácticas que podemos seguir para reducir la frecuencia de su ocurrencia.



CURSORES

Una consulta que no utiliza *ORDER BY* devuelve un conjunto (o un multiset), mientras que una consulta con *ORDER BY* devuelve lo que el SQL estándar llama *cursor* –una estructura no relacional con el orden garantizado entre sus filas.

SQL y T-SQL también admiten un objeto llamado *cursor* que podemos utilizar para procesar las filas resultantes de una consulta de a una a la vez y en el orden solicitado. Esto contrasta con el uso de consultas basadas en conjuntos: consultas normales sin un cursor para las cuales manipula el conjunto o multiset y no puede confiar en el orden.

Quiero enfatizar que la elección predeterminada debe ser utilizar consultas basadas en conjuntos; solo cuando tengamos una razón convincente para hacer lo contrario, debemos considerar el uso de cursos. Esta recomendación se basa en varios factores, incluidos los siguientes:

- En primer lugar, cuando utilizamos cursos, vamos en contra del modelo relacional, que se basa en la teoría de conjuntos.
- La manipulación registro por registro realizada por el cursor tiene sobrecarga. Un cierto costo adicional está asociado con cada manipulación de registros por el cursor en comparación con la manipulación basada en conjuntos. Dada una consulta basada en un conjunto y un código de cursor que realiza un procesamiento físico similar detrás de las escenas, el código del cursor suele ser mucho más lento que el código basado en el conjunto.
- Con los cursos, escribimos soluciones imperativas; en otras palabras, somos responsables de definir cómo procesar los datos (declarar el cursor, abrirlo, recorrer los registros del cursor, cerrar el cursor y desasignar el cursor). Con las soluciones basadas en conjuntos, escribimos un código declarativo donde se enfoca principalmente en los aspectos lógicos de la solución; en otras palabras, en qué obtener en lugar de como obtenerlo. Por lo tanto, las soluciones de cursor tienden a ser más largas, menos legibles y más difíciles de mantener que las soluciones basadas en conjuntos.

Para la mayoría de las personas, no es sencillo pensar en términos relacionales inmediatamente cuando comienzan a aprender SQL. Es más intuitivo para la mayoría de las personas pensar en términos de cursos: procesar un registro a la vez en un cierto orden. Como resultado, los cursos se utilizan ampliamente y, en la mayoría de los casos, se utilizan incorrectamente; es decir, se utilizan incluso cuando existen soluciones mucho mejores basadas en conjuntos. Hagamos un esfuerzo consciente para adoptar el estado mental basado en conjuntos y para pensar verdaderamente en términos de conjuntos. Puede llevar tiempo, en algunos casos años, pero mientras trabajemos con un lenguaje basado en el modelo relacional, esa es la manera correcta de pensar.

Cada regla tiene excepciones. Un ejemplo es cuando necesitamos aplicar una determinada tarea a cada fila de alguna tabla o vista. Por ejemplo, puede que necesite ejecutar alguna tarea administrativa para cada índice o tabla en su base de datos. En tal caso, tiene sentido usar un cursor para iterar a través de los nombres de índice o tabla uno a la vez y ejecutar la tarea relevante para cada uno de ellos.

Otro ejemplo de cuándo debería considerar los cursos es cuando su solución basada en conjuntos funciona mal y agotamos esfuerzos de ajuste utilizando el enfoque basado en conjuntos. Como se mencionó, las soluciones basadas en conjuntos tienden a ser mucho más rápidas, pero en algunos casos excepcionales, la solución de cursor es más rápida. Un ejemplo de este tipo es el cálculo de agregados en ejecución utilizando el código T-SQL que es compatible con versiones legacy de SQL Server que no admiten la opción de marco en las funciones de ventana. Las soluciones relacionales para ejecutar agregados utilizando uniones o subconsultas son extremadamente lentas. Una solución iterativa, como una basada en



un cursor, suele ser la óptima. Si no hay restricciones de compatibilidad, utilizar una solución relacional con funciones de ventana es la forma óptima de calcular los totales acumulados.

Trabajar con un cursor generalmente implica los siguientes pasos:

1. Declare el cursor basado en una consulta.
2. Abra el cursor.
3. Obtenga los valores de los atributos del primer registro de cursor en variables.
4. Mientras no hayamos llegado al final del cursor (mientras que el valor de una función llamada `@@FETCH_STATUS` es 0), recorremos los registros del cursor; en cada iteración del bucle, realizamos el procesamiento necesario para la fila actual y luego obtenemos los valores de los atributos de la siguiente fila en las variables.
5. Cierre el cursor.
6. Desasignar el cursor.

El siguiente ejemplo con código de cursor calcula la cantidad total acumulada para cada cliente y mes desde la vista `Ventas.ClientesOrdenes`:

```
USE TSQLV6ES;
SET NOCOUNT ON;

DECLARE @Resultado AS TABLE (
clienteid      INT,
mesorden       DATE,
cantidad        INT,
cantidadacumulada  INT,
PRIMARY KEY(clienteid, mesorden)
);

DECLARE
@clienteid      AS INT,
@antclienteid   AS INT,
@mesorden        AS DATE,
@cantidad         AS INT,
@cantidadacumulada AS INT;

DECLARE C CURSOR FAST_FORWARD /* solo lectura, solo avanza */
FOR SELECT clienteid, MesOrden, cantidad
FROM Ventas.ClientesOrdenes
ORDER BY clienteid, MesOrden;
OPEN C;
FETCH NEXT FROM C INTO @clienteid, @mesorden, @cantidad;
SELECT @antclienteid = @clienteid, @cantidadacumulada = 0;
WHILE @@FETCH_STATUS = 0
BEGIN
    IF @clienteid <> @antclienteid
        SELECT @antclienteid = @clienteid, @cantidadacumulada = 0;
        SET @cantidadacumulada = @cantidadacumulada + @cantidad;
        INSERT INTO @Resultado VALUES(@clienteid, @mesorden, @cantidad, @cantidadacumulada);

    FETCH NEXT FROM C INTO @clienteid, @mesorden, @cantidad;
END;
CLOSE C;
DEALLOCATE C;
SELECT clienteid, CONVERT(VARCHAR(7), mesorden, 121) AS mesorden, cantidad,
cantidadacumulada
FROM @Resultado
ORDER BY clienteid, mesorden;
```



El código declara un cursor basado en una consulta que devuelve las filas de la vista de Clientes ordenados por ID de cliente y mes de pedido, y se itera a través de los registros de uno en uno. El código realiza un seguimiento de la cantidad total actual en ejecución en una variable llamada `@cantidadacumulada` que se restablece cada vez que se encuentra un nuevo cliente. Para cada fila, el código calcula el total acumulado actual agregando la cantidad del mes actual (`@cantidad`) a `@cantidadacumulada`, e inserta una fila con el ID del cliente, el mes del pedido, la cantidad del mes actual y la cantidad de ejecución en una variable de tabla llamada `@Resultado`. Cuando el código termina de procesar todos los registros del cursor, consulta la variable de la tabla para presentar los agregados en ejecución.

Aquí está la salida devuelta por este código, que se muestra en forma abreviada:

clienteid	mesorden	cantidad	cantidadacumulada
1	2021-08	38	38
1	2021-10	41	79
1	2022-01	17	96
1	2022-03	18	114
1	2022-04	60	174
2	2020-09	6	6
2	2021-08	18	24
2	2021-11	10	34
2	2022-03	29	63
3	2020-11	24	24
3	2021-04	30	54
3	2021-05	80	134
3	2021-06	83	217
3	2021-09	102	319
3	2022-01	40	359
...			
90	2021-07	5	5
90	2021-09	15	20
90	2021-10	34	54
90	2022-02	82	136
90	2022-04	12	148
91	2020-12	45	45
91	2021-07	31	76
91	2021-12	28	104
91	2022-02	20	124
91	2022-04	81	205

En otra unidad veremos que T-SQL permite el uso de funciones de ventana, que podemos utilizar para obtener una solución elegante y mucho más eficiente, sin necesidad de utilizar cursosres, como podemos ver a continuación:

```
USE TSQLV6ES;
SELECT clienteid, MesOrden, cantidad, SUM(cantidad) OVER(PARTITION BY clienteid
ORDER BY MesOrden ROWS UNBOUNDED PRECEDING) AS cantidadacumulada
FROM Ventas.ClientesOrdenes
ORDER BY clienteid, MesOrden;
```



TABLAS TEMPORALES

Cuando necesitamos almacenar temporalmente datos en tablas, en ciertos casos podemos preferir no trabajar con tablas permanentes. Supongamos que necesitamos que los datos sean visibles solo para la sesión actual, o incluso solo para el lote actual. Como ejemplo, supongamos que necesitamos almacenar datos temporales durante el procesamiento de los datos, como en el ejemplo del cursor en la sección anterior.

Otro caso en el que solemos usar tablas temporales es cuando no tenemos permisos para crear tablas permanentes en una base de datos de usuarios.

SQL Server admite tres tipos de tablas temporales con las que puede ser más conveniente trabajar que las tablas permanentes en tales casos: tablas temporales locales, tablas temporales globales y variables de tabla. Las siguientes secciones describen los tres tipos y demuestran su uso con ejemplos de código.

TABLAS TEMPORALES LOCALES

Para crear una tabla temporal local, la nombramos con único numeral como prefijo, como #T1. Los tres tipos de tablas temporales se crean en la base de datos *tempdb*.

Una tabla temporal local solo es visible para la sesión que la creó, en el nivel de creación y en todos los niveles internos de la pila de llamadas (procedimientos internos, activadores y lotes dinámicos). SQL Server destruye automáticamente una tabla temporal local cuando el nivel de creación en la pila de llamadas queda fuera del alcance. Por ejemplo, supongamos que un procedimiento almacenado llamado *Proc1* llama a un procedimiento llamado *Proc2*, que a su vez llama a un procedimiento llamado *Proc3*, que a su vez llama a un procedimiento llamado *Proc4*. *Proc2* crea una tabla temporal llamada #T1 antes de llamar a *Proc3*. La tabla #T1 es visible para *Proc2*, *Proc3* y *Proc4*, pero no para *Proc1*, y SQL Server la destruye automáticamente cuando termina *Proc2*. Si la tabla temporal se crea en un lote ad hoc en el nivel de anidación más externo de la sesión (en otras palabras, cuando el valor de la función @@NESTLEVEL es 0), también es visible para todos los lotes posteriores y se destruye por SQL Server automáticamente solo cuando la sesión de creación se desconecta.

Podría preguntarse cómo SQL Server evita conflictos de nombres cuando dos sesiones crean tablas temporales locales con el mismo nombre. SQL Server agrega internamente un sufijo al nombre de la tabla que lo hace único en *tempdb*. Como desarrollador, no debería importarte: te refieres a la tabla utilizando el nombre que proporcionaste sin el sufijo interno, y solo tu sesión tiene acceso a tu tabla.

Un escenario obvio para el cual las tablas temporales locales son útiles es cuando tiene un proceso que necesita almacenar resultados intermedios temporalmente, como durante un ciclo, y luego consultar los datos.

Otro escenario es cuando necesita acceder al resultado de un procesamiento costoso varias veces. Por ejemplo, supongamos que necesitamos unir las tablas *Ventas.Ordenes* y *Ventas.DetallesOrden*, sumar las cantidades de pedido por año de pedido y unir dos instancias de los datos agregados para comparar la cantidad total de cada año con el año anterior. Las tablas *Ordenes* y *DetallesOrden* en la base de datos de muestra son muy pequeñas, pero en situaciones reales, estas tablas pueden tener millones de filas. Una opción es usar expresiones de tabla, pero recuerde que las expresiones de tabla son virtuales. El trabajo costoso que involucra escanear todos los datos, unir las tablas *Ordenes* y *DetallesOrden*, y agregar los datos tendría que ocurrir dos veces con expresiones de tabla. En su lugar, tiene sentido hacer todo el trabajo costoso solo una vez (almacenar el resultado en una tabla temporal local) y luego unir dos instancias de la tabla temporal, especialmente porque el resultado del trabajo costoso es un conjunto pequeño con solo una fila por año de orden.

El siguiente código ilustra este escenario utilizando una tabla temporal local:



```
USE TSQLV6ES;
DROP TABLE IF EXISTS #MiTotalOrdenesPorAnio;
GO

CREATE TABLE #MiTotalOrdenesPorAnio (
ordenanio INT NOT NULL PRIMARY KEY,
cantidad INT NOT NULL
);

INSERT INTO #MiTotalOrdenesPorAnio(ordenanio, cantidad)
SELECT YEAR(0.fechaorden) AS ordenanio, SUM(Do.cantidad) AS cantidad
FROM Ventas.Ordenes AS O
INNER JOIN Ventas.DetallesOrden AS DO ON DO.ordenid = O.ordenid
GROUP BY YEAR(fechaorden);

SELECT Act.ordenanio, Act.cantidad AS cantidadanioact, Prv.cantidad AS cantidadaniopr
FROM #MiTotalOrdenesPorAnio AS Act
LEFT OUTER JOIN #MiTotalOrdenesPorAnio AS Prv ON Act.ordenanio = Prv.ordenanio + 1;
```

Este código genera la siguiente salida:

ordenanio	cantidadanioact	cantidadaniopr
2020	9581	NULL
2021	25489	9581
2022	16247	25489

Para verificar que la tabla local temporal es visible solo en la sesión que la crea, intenten acceder a la tabla desde otra sesión:

```
USE TSQLV6ES;
SELECT ordenanio, cantidad
FROM #MiTotalOrdenesPorAnio;
```

Obtendremos el siguiente error:

```
Msg 208, Level 16, State 0, Line 1
Invalid object name '#MiTotalOrdenesPorAnio'.
```

Al finalizar, vuelvan a la sesión original y eliminan la tabla temporal:

```
DROP TABLE IF EXISTS #MiTotalOrdenesPorAnio;
```

Generalmente se recomienda limpiar los recursos tan pronto como haya terminado de trabajar con ellos.

TABLAS TEMPORALES GLOBALES

Cuando creamos una tabla temporal global, es visible para todas las demás sesiones. SQL Server destruye automáticamente las tablas temporales globales cuando la sesión de creación se desconecta y no hay referencias activas a la tabla. Creamos una tabla temporal global al nombrarla con dos numerales como un prefijo, como ##T1.

Las tablas temporales globales son útiles cuando desea compartir datos temporales con todos. No se requieren permisos especiales, y todos tienen acceso total a DDL y DML. Por supuesto, el hecho de que todos tengan acceso completo significa que cualquiera puede cambiar o incluso eliminar la tabla, así que considere las alternativas con cuidado.

Por ejemplo, el siguiente código crea una tabla temporal global llamada ##Globales con columnas llamadas id y val:



```
CREATE TABLE ##Globales (
id sysname NOT NULL PRIMARY KEY,
val SQL_VARIANT NOT NULL
);
```

La tabla en este ejemplo está diseñada para imitar variables globales, que no son compatibles con T-SQL. La columna id es de un tipo de datos sysname (el tipo que SQL Server utiliza internamente para representar identificadores), y la columna val es de un tipo de datos SQL_VARIANT (un tipo genérico que puede almacenar dentro de él un valor de casi cualquier tipo de datos).

Cualquiera puede insertar filas en la tabla. Por ejemplo, ejecute el siguiente código para insertar una fila que represente una variable llamada *i* e inicialícela con el valor entero 10:

```
INSERT INTO ##Globales(id, val) VALUES(N'i', CAST(10 AS INT));
```

Cualquiera puede modificar y consultar datos de la tabla. Por ejemplo, correr el siguiente código desde cualquier sesión para consultar el valor actual de la variable *i*:

```
SELECT val FROM ##Globales WHERE id = N'i';
```

Este código devuelve la siguiente salida:

```
val
-----
10
```

Tengan en cuenta que tan pronto como la sesión que creó la tabla temporal global se desconecta y no hay referencias activas a la tabla, SQL Server destruye la tabla automáticamente.

Si desea que se cree una tabla temporal global cada vez que se inicie SQL Server y no quiere que SQL Server intente destruirlo automáticamente, debe crear la tabla a partir de un procedimiento almacenado que esté marcado como un procedimiento de inicio. (Para más detalles, puede ver “sp_procoption” en los libros en línea de SQL Server en la siguiente dirección: [sp_procoption \(Transact-SQL\) - SQL Server | Microsoft Docs](#)).

Corra el siguiente código desde cualquier sesión para destruir explícitamente la tabla temporal global:

```
DROP TABLE IF EXISTS ##Globales;
```

VARIABLES DE TABLA

Las variables de tabla son similares a las tablas temporales locales en algunos aspectos y diferentes en otros. Declaramos las variables de tabla de manera muy similar a como declaramos otras variables, utilizando la instrucción *DECLARE*.

Al igual que con las tablas temporales locales, las variables de tabla tienen una presencia física como una tabla en la base de datos *tempdb*, contrariamente a la idea errónea de que existen solo en la memoria. Al igual que las tablas temporales locales, las variables de la tabla son visibles solo para la sesión de creación, pero como son variables, tienen un alcance más limitado: solo el lote actual. Las variables de tabla no son visibles para los lotes internos en la pila de llamadas ni para los lotes posteriores en la sesión.

Si una transacción explícita se revierte, los cambios realizados en las tablas temporales de esa transacción también se retrotraen; sin embargo, los cambios realizados en las variables de tabla por las declaraciones que se completaron en la transacción no se retrotraen. Sólo se deshacen los cambios realizados por la declaración activa que falló o que se terminó antes de completarse.



Las tablas temporales y las variables de tabla también tienen diferencias de optimización, pero esos temas están fuera del alcance de la materia. Por ahora, solo diremos que, en términos de rendimiento, generalmente tiene más sentido usar variables de tabla con pequeños volúmenes de datos (solo unas pocas filas) y usar tablas temporales locales en el resto de los casos.

Por ejemplo, el siguiente código utiliza una variable de tabla en lugar de una tabla temporal local para comparar las cantidades totales de pedidos de cada año de pedido con el año anterior:

```
USE TSQLV6ES;
DECLARE @MiTotalOrdenesPorAnio TABLE (
    anioorden INT NOT NULL PRIMARY KEY,
    cantidad   INT NOT NULL);

INSERT INTO @MiTotalOrdenesPorAnio(anioorden, cantidad)

SELECT
    YEAR(O.fechaorden) AS orderyear, SUM(DO.cantidad) AS cantidad
    FROM Ventas.Ordenes AS O
    INNER JOIN Ventas.DetallesOrden AS DO ON DO.ordenid = O.ordenid
    GROUP BY YEAR(fechaorden);

SELECT Act.anioorden, Act.cantidad AS actaniocant, Prv.cantidad AS prvaniocant
    FROM @MiTotalOrdenesPorAnio AS Act
    LEFT OUTER JOIN @MiTotalOrdenesPorAnio AS Prv
    ON Act.anioorden = Prv.anioorden + 1;
```

Este código devuelve la siguiente salida:

ordenanio	cantidadanioact	cantidadanionprv
2020	9581	NULL
2021	25489	9581
2022	16247	25489

Tengan en cuenta que en lugar de usar una variable de tabla o una tabla temporal y un auto-join aquí, esta tarea en particular se puede manejar alternativamente con la función LAG, como esto:

```
USE TSQLV6ES;
SELECT YEAR(O.fechaorden) AS anioorden, SUM(DO.cantidad) AS actaniocant,
LAG(SUM(DO.cantidad)) OVER(ORDER BY YEAR(fechaorden)) AS prvaniocant
    FROM Ventas.Ordenes AS O
    INNER JOIN Ventas.DetallesOrden AS DO ON DO.ordenid = O.ordenid
    GROUP BY YEAR(fechaorden);
```

TIPOS DE TABLA

Podemos usar un tipo de tabla para conservar una definición de tabla como un objeto en la base de datos. Más tarde, podemos reutilizarlo como en la definición variables de tabla como tipo y parámetros de entrada de procedimientos almacenados y funciones definidas por el usuario. Los tipos de tabla son necesarios para los parámetros con valores de tabla (TVPs).

Por ejemplo, el siguiente código crea un tipo de tabla llamado dbo.TotalOrdenesPorAnio en la base de datos actual:

```
USE TSQLV6ES;
DROP TYPE IF EXISTS dbo.TotalOrdenesPorAnio ;

CREATE TYPE dbo.TotalOrdenesPorAnio AS TABLE (
    anioorden INT NOT NULL PRIMARY KEY,
    cantidad   INT NOT NULL);
```



Después de crear el tipo de tabla, siempre que necesitemos declarar una variable de tabla según la definición del tipo de tabla, no necesitaremos repetir el código; en su lugar, simplemente podemos especificar dbo.TotalOrdenesPorAnio como el tipo de la variable, de esta manera:

```
DECLARE @MiTotalOrdenesPorAnio AS dbo.TotalOrdenesPorAnio;
```

Como un ejemplo más completo, el siguiente código declara una variable llamada @MiTotalOrdenesPorAnio del nuevo tipo de tabla, consulta las tablas Ordenes y DetallesOrden para calcular las cantidades totales de orden por año de pedido, almacena el resultado de la consulta en la variable de tabla y consulta la variable para presentar sus contenidos:

```
DECLARE @MiTotalOrdenesPorAnio AS dbo.TotalOrdenesPorAnio;

INSERT INTO @MiTotalOrdenesPorAnio(anioorden, cantidad)
SELECT YEAR(O.fechaorden) AS anioorden, SUM(DO.cantidad) AS cantidad
FROM Ventas.Ordenes AS O
INNER JOIN Ventas.DetallesOrden AS DO ON DO.ordenid = O.ordenid
GROUP BY YEAR(fechaorden);

SELECT anioorden, cantidad
FROM @MiTotalOrdenesPorAnio;
```

Este código devuelve la siguiente salida:

anioorden	cantidad
2020	9581
2021	25489
2022	16247

El beneficio de la función de tipo de tabla se extiende más allá de simplemente ayudarlo a acortar su código. Como mencioné, podemos usarlo como el tipo de parámetros de entrada de procedimientos y funciones almacenados, lo cual es una capacidad útil.



SQL DINÁMICO

Con SQL Server, podemos construir un lote de código T-SQL como una cadena de caracteres y luego ejecutar ese lote. Esta capacidad se llama SQL dinámico. SQL Server proporciona dos formas de ejecutar SQL dinámico: usar el comando *EXEC* (abreviatura de *EXECUTE*) y usar el procedimiento almacenado *sp_executesql*. Explicaré la diferencia entre los dos y proporcionaré ejemplos para usar cada uno.

El SQL dinámico es útil para varios propósitos, incluyendo los siguientes:

- Automatizar tareas administrativas. Por ejemplo, consultar metadatos y construir y ejecutar una instrucción *BACKUP DATABASE* para cada base de datos en la instancia.
- Mejora del rendimiento de ciertas tareas. Por ejemplo, la construcción de consultas ad hoc parametrizadas que pueden reutilizar los planes de ejecución previamente almacenados en caché (más sobre esto más adelante)
- Creación de elementos del código en función de la consulta de los datos reales. Por ejemplo, creación dinámica de una consulta *PIVOT* cuando no sabe de antemano qué elementos deberían aparecer en la cláusula *IN* del operador *PIVOT*

Tenga mucho cuidado al concatenar la entrada del usuario como parte de su código. Los piratas informáticos pueden intentar inyectar código que no pretendía ejecutar. La mejor medida que puede tomar contra la inyección de SQL es evitar concatenar las entradas del usuario como parte de su código (por ejemplo, mediante el uso de parámetros). Si concatena las entradas del usuario como parte de su código, asegúrese de inspeccionar minuciosamente la entrada y busque los intentos de inyección de SQL. Puede encontrar un artículo sobre el tema en los Libros en pantalla de SQL Server usando la siguiente URL: [SQL Injection - SQL Server | Microsoft Docs](#)

EL COMANDO EXEC

El comando *EXEC* acepta una cadena de caracteres entre paréntesis como entrada y ejecuta el lote de código dentro de la cadena de caracteres. *EXEC* admite cadenas de caracteres normales y Unicode como entrada. Este comando también se puede usar para ejecutar un procedimiento almacenado, como veremos más adelante en esta unidad.

El siguiente ejemplo almacena una cadena de caracteres con una instrucción *PRINT* en la variable *@sql* y luego usa el comando *EXEC* para invocar el lote de código almacenado dentro de la variable:

```
DECLARE @sql AS VARCHAR(100);
SET @sql = 'PRINT ''Este mensaje fue impreso por un lote de SQL dinamico.'';
EXEC(@sql);
```

Observe el uso de dos comillas simples para representar una comilla simple en una cadena dentro de una cadena.

Este código devuelve el siguiente resultado:

```
Este mensaje fue impreso por un lote de SQL dinamico.
```

EL STORED PROCEDURE SP_EXECUTESQL

El procedimiento almacenado *sp_executesql* es una herramienta alternativa al comando *EXEC* para ejecutar código SQL dinámico. Es más seguro y más flexible en el sentido de que tiene una interfaz; es decir, soporta parámetros de entrada y salida. Tenga en cuenta que, a diferencia de *EXEC*, *sp_executesql* solo admite cadenas de caracteres Unicode como lote de entrada de código.



El hecho de que pueda usar los parámetros de entrada y salida en su código de SQL dinámico puede ayudarlo a escribir un código más seguro y más eficiente. En términos de seguridad, los parámetros que aparecen en el código no pueden considerarse parte del código, solo pueden considerarse operandos en expresiones. Entonces, al usar los parámetros, puede eliminar su exposición a la inyección de SQL.

El procedimiento almacenado `sp_executesql` puede funcionar mejor que `EXEC` porque su parametrización ayuda a reutilizar los planes de ejecución en caché. Un plan de ejecución es el plan de procesamiento físico que SQL Server produce para una consulta, con el conjunto de instrucciones que describen a qué objetos acceder, en qué orden, qué índices utilizar, cómo acceder a ellos, qué algoritmos se unen, y así sucesivamente. Uno de los requisitos para reutilizar un plan previamente almacenado en caché es que la cadena de consulta sea la misma para la que se creó el plan almacenado en caché. La mejor manera de reutilizar de manera eficiente los planes de ejecución de consultas es utilizar procedimientos almacenados con parámetros. De esta manera, incluso cuando los valores de los parámetros cambian, la cadena de consulta sigue siendo la misma. Pero si decide usar código ad-hoc en lugar de procedimientos almacenados, al menos todavía puede trabajar con parámetros si usa `sp_executesql` y, por lo tanto, aumenta las posibilidades de reutilización del plan.

El procedimiento `sp_executesql` tiene dos parámetros de entrada y una sección de asignaciones. Especifica la cadena de caracteres Unicode que contiene el lote de código que desea ejecutar en el primer parámetro, que se llama `@stmt`. Proporciona una cadena de caracteres Unicode que contiene las declaraciones de los parámetros de entrada y salida en el segundo parámetro de entrada, que se llama `@params`. A continuación, especifique las asignaciones de los parámetros de entrada y salida separados por comas.

El siguiente ejemplo construye un lote de código con una consulta en la tabla `Ventas.Ordenes`. El ejemplo utiliza un parámetro de entrada llamado `@ordenid` en el filtro de la consulta:

```
USE TSQLV6ES;
DECLARE @sql AS NVARCHAR(100);

SET @sql = N'SELECT ordenid, clienteid, empid, fechaorden FROM Ventas.Ordenes
WHERE ordenid = @ordenid;';

EXEC sp_executesql @stmt = @sql,
@params = N'@ordenid AS INT', @ordenid = 10248;
```

Este código genera la siguiente salida:

ordenid	clienteid	empid	fechaorden
10248	85	5	2020-07-04

Este código asigna el valor 10248 al parámetro de entrada, pero incluso si lo ejecuta de nuevo con un valor diferente, la cadena de código sigue siendo la misma. De esta manera, aumenta las posibilidades de reutilizar un plan previamente almacenado en caché.

USANDO PIVOT CON SQL DINÁMICO

En una unidad anterior vimos cómo usar el operador `PIVOT` para pivotar datos. Mencionamos que, en una consulta estática, debemos saber de antemano qué valores especificar en la cláusula `IN` del operador `PIVOT`. A continuación, se muestra un ejemplo de una consulta estática con el operador `PIVOT`:

```
USE TSQLV6ES;
SELECT *
FROM (SELECT transporteid , YEAR(fechaorden) AS anioorden, flete
      FROM Ventas.Ordenes) AS D
PIVOT(SUM(flete) FOR anioorden IN([2020],[2021],[2022])) AS P;
```



Este ejemplo consulta la tabla *Ventas.Ordenes* y hace pivotar los datos para que devuelva los ID de transporte en las filas, los años de pedido en las columnas y el flete total en la intersección de cada año de envío y pedido. Este código devuelve el siguiente resultado:

transporteid	2020	2021	2022
3	4233,78	11413,35	4865,38
1	2297,42	8681,38	5206,53
2	3748,67	12374,04	12122,14

Con la consulta estática, debe saber de antemano qué valores (años de orden en este caso) debe especificar en la cláusula IN del operador PIVOT. Esto significa que necesita revisar el código cada año. En su lugar, puede consultar los distintos años de orden a partir de los datos, construir un lote de código de SQL dinámico basado en los años que consultó y ejecutar el lote de SQL dinámico de la siguiente manera:

```
USE TSQLV6ES;
DECLARE
@sql      AS NVARCHAR(1000),
@anioorden AS INT,
@primero   AS INT;

DECLARE C CURSOR FAST_FORWARD FOR
SELECT DISTINCT(YEAR(fechaorden)) AS anioorden
FROM Ventas.Ordenes
ORDER BY anioorden;
SET @primero = 1;
SET @sql = N'SELECT *
FROM (SELECT transporteid, YEAR(fechaorden) AS anioorden, flete FROM Ventas.Ordenes) AS D
PIVOT(SUM(flete) FOR anioorden IN(';
OPEN C;
FETCH NEXT FROM C INTO @anioorden;

WHILE @@fetch_status = 0
BEGIN
    IF @primero = 0
    SET @sql += N',' ELSE
    SET @primero = 0;
    SET @sql += QUOTENAME(@anioorden);
    FETCH NEXT FROM C INTO @anioorden;
END;
CLOSE C;
DEALLOCATE C;

SET @sql += N')) AS P;';
EXEC sp_executesql @stmt = @sql;
```



RUTINAS

Las rutinas son objetos programables que encapsulan código para calcular un resultado o ejecutar una actividad. SQL Server admite tres tipos de rutinas: funciones definidas por el usuario, procedimientos almacenados (stored procedures) y desencadenadores (triggers).

Con SQL Server, podemos elegir si deseamos desarrollar una rutina con T-SQL o con código .NET de Microsoft basado en la integración de CLR en el producto. Debido a que el enfoque de esta materia es T-SQL, los ejemplos de aquí usan T-SQL. Cuando la tarea en cuestión implica principalmente la manipulación de datos, T-SQL suele ser una mejor opción. Cuando la tarea es más sobre la lógica iterativa, la manipulación de cadenas o las operaciones intensivas en computación, el código .NET suele ser una mejor opción.

FUNCIONES DEFINIDAS POR EL USUARIO

El propósito de una función definida por el usuario (UDF) es encapsular la lógica que calcula algo, posiblemente en función de los parámetros de entrada, y devolver un resultado.

SQL Server admite UDF escalares y con valores de tabla. Las UDF escalares devuelven un solo valor, las UDF con valores de tabla devuelven una tabla. Una de las ventajas de usar UDF es que podemos incorporarlos a las consultas. Las UDF escalares pueden aparecer en cualquier parte de la consulta donde puede aparecer una expresión que devuelve un solo valor (por ejemplo, en la lista *SELECT*). Las UDF de tabla pueden aparecer en la cláusula *FROM* de una consulta. El ejemplo en esta sección es un UDF escalar.

Los UDF no pueden tener efectos secundarios. Obviamente, esto significa que las UDF no pueden aplicar ningún cambio de esquema o datos en la base de datos. Pero otras formas de causar efectos secundarios son menos obvias.

Por ejemplo, invocar la función *RAND* para devolver un valor aleatorio o la función *NEWID* para devolver un identificador único global (GUID) tiene efectos secundarios. Cada vez que invoca la función *RAND* sin especificar una semilla, SQL Server genera una semilla aleatoria que se basa en la invocación anterior de *RAND*. Por este motivo, SQL Server necesita almacenar información internamente cada vez que invoque la función *RAND*. De manera similar, cada vez que invoque la función *NEWID*, el sistema debe reservar algo de información para que se tenga en cuenta en la próxima invocación de *NEWID*. Debido a que *RAND* y *NEWID* tienen efectos secundarios, no está permitido usarlos en sus UDF.

Por ejemplo, el siguiente código crea un UDF llamado *dbo.GetEdad* que devuelve la edad de una persona con una fecha de nacimiento específica (argumento *@fechanacimiento*) en una fecha de evento específica (argumento *@fechaevento*):

```
USE TSQVL6ES;
DROP FUNCTION IF EXISTS dbo.GetEdad;
GO
CREATE FUNCTION dbo.GetEdad (
    @fechanacimiento AS DATE,
    @fechaevento AS DATE
) RETURNS INT
AS
BEGIN
    RETURN
        DATEDIFF(year, @fechanacimiento, @fechaevento) -
        CASE WHEN 100 * MONTH(@fechaevento) + DAY(@fechaevento) <
        100 * MONTH(@fechanacimiento) + DAY(@fechanacimiento)
            THEN 1
            ELSE 0
        END;
END;
```



La función calcula la edad como la diferencia, en términos de años, entre el año de nacimiento y el año del evento, menos 1 año en los casos en que el mes y el día del evento son más pequeños que el mes y el día del nacimiento. La expresión $100 * \text{mes} + \text{día}$ es simplemente un truco para concatenar el mes y el día. Por ejemplo, para el duodécimo día del mes de febrero, la expresión produce el número entero 212.

Tenga en cuenta que una función puede tener más de una cláusula *RETURN* en su cuerpo. Puede tener código con elementos de flujo, cálculos y más. Pero la función debe tener una cláusula *RETURN* que devuelve un valor.

Para demostrar el uso de un UDF en una consulta, el siguiente código consulta la tabla *RH.Empleados* e invoca la función *GetEdad* en la lista *SELECT* para calcular la edad de cada empleado hoy:

```
USE TSQVLV6ES;
SELECT empid, nombre, apellido, nacimiento, dbo.GetEdad(nacimiento, SYSDATETIME()) AS edad
FROM RH.Empleados;
```

Por ejemplo, si ejecutara esta consulta el 8 de marzo de 2024, obtendría la siguiente salida:

empid	nombre	apellido	nacimiento	edad
1	Sara	Davis	1968-12-08	55
2	Don	Funk	1972-02-19	52
3	Judy	Lew	1983-08-30	40
4	Yael	Peled	1957-09-19	66
5	Sven	Mortensen	1975-03-04	49
6	Paul	Suurs	1983-07-02	40
7	Russell	King	1980-05-29	43
8	Maria	Cameron	1978-01-09	46
9	Patricia	Doyle	1986-01-27	38

Tengan en cuenta que, si ejecutan la consulta en sus sistemas, los valores que obtengan en la columna de edad dependerán de la fecha en que ejecuten la consulta.

STORED PROCEDURES

Los procedimientos almacenados son rutinas que encapsulan código. Pueden tener parámetros de entrada y salida, pueden devolver conjuntos de resultados de consultas y se les permite tener efectos secundarios. No solo puede modificar datos a través de procedimientos almacenados, también puede aplicar cambios de esquema a través de ellos.

En comparación con el uso de código ad-hoc, el uso de procedimientos almacenados le brinda muchos beneficios:

- Los procedimientos almacenados encapsulan la lógica. Si necesita cambiar la implementación de un procedimiento almacenado, aplique el cambio utilizando el comando *ALTER PROC*, y todos los usuarios del procedimiento usarán la nueva versión desde ese momento.
- Los procedimientos almacenados le dan un mejor control de seguridad. Puede otorgar a un usuario permisos para ejecutar el procedimiento sin otorgarle permisos directos para que realice las actividades subyacentes. Por ejemplo, suponga que desea permitir que ciertos usuarios eliminan un cliente de la base de datos, pero no desea otorgarles permisos directos para eliminar filas de la tabla *Clients*. Desea asegurarse de que las solicitudes para eliminar un cliente se validen (por ejemplo, verificando si el cliente tiene pedidos abiertos o deudas abiertas) y es posible que también desee auditar las solicitudes. Al no otorgar permisos directos para eliminar filas de la tabla *Clients*, sino otorgar permisos para ejecutar un procedimiento que maneje la tarea, se asegura de que siempre se realicen todas las validaciones y auditorías necesarias. Además, los procedimientos almacenados con parámetros pueden ayudar a



prevenir la inyección de SQL, especialmente cuando reemplazan el SQL ad-hoc enviado desde la aplicación cliente.

- Puede incorporar todo el código de manejo de errores dentro de un procedimiento, tomando medidas correctivas en silencio cuando sea relevante. Veremos el manejo de errores más adelante en esta unidad.
- Los procedimientos almacenados dan beneficios de rendimiento. Anteriormente hablamos sobre la reutilización de planes de ejecución previamente almacenados en caché. Las consultas en el procedimiento almacenado generalmente están parametrizadas y, por lo tanto, tienen una alta probabilidad de reutilizar los planes previamente almacenados en caché. Otro beneficio de rendimiento de usar procedimientos almacenados es una reducción en el tráfico de red. La aplicación cliente debe enviar solo el nombre del procedimiento y sus argumentos a SQL Server. El servidor procesa todo el código del procedimiento y devuelve solo la salida a la persona que llama. Ningún tráfico de ida y vuelta está asociado con pasos intermedios del procedimiento.

Como un ejemplo simple, el siguiente código crea un procedimiento almacenado llamado `Ventas.GetOrdenesdeCliente`. El procedimiento acepta una ID de cliente (`@clienteid`) y un rango de fechas (`@desdefecha` y `@hastafecha`) como entrada. El procedimiento devuelve filas de la tabla `Ventas.Ordenes` que representan los pedidos realizados por el cliente solicitado en el rango de fechas solicitado como un conjunto de resultados, y el número de filas afectadas como un parámetro de salida (`@numfilas`):

```
USE TSQLV6ES;
DROP PROC IF EXISTS Ventas.GetOrdenesdeCliente;
GO

CREATE PROC Ventas.GetOrdenesdeCliente
@clienteid      AS INT,
@desdefecha    AS DATETIME = '19000101',
@hastafecha    AS DATETIME = '99991231',
@numfilas       AS INT OUTPUT
AS
SET NOCOUNT ON;

SELECT ordenid, clienteid, empid, fechaorden
FROM Ventas.Ordenes
WHERE clienteid = @clienteid
AND fechaorden >= @desdefecha AND fechaorden < @hastafecha;

SET @numfilas = @@rowcount;
GO
```

Al ejecutar el procedimiento, si no especificamos un valor en el parámetro `@desdefecha`, el procedimiento usará el valor predeterminado 19000101, y si no especificamos un valor en el parámetro `@hastafecha`, el procedimiento usará el valor predeterminado 99991231. Observe el uso de la palabra clave `OUTPUT` para indicar que el parámetro `@numfilas` es un parámetro de salida. El comando `SET NOCOUNT ON` se usa para suprimir los mensajes que indican cuántas filas se vieron afectadas por las declaraciones DML, como la instrucción `SELECT` dentro del procedimiento.

Este es un ejemplo de cómo ejecutar el procedimiento, solicitar información sobre los pedidos realizados por el cliente con el ID de 1 en el año 2021. El código obtiene el valor del parámetro de salida `@numfilas` en la variable local `@rc` y lo devuelve para mostrar cuántas filas fueron afectadas por la consulta:



```
USE TSQLV6ES;
DECLARE @rc AS INT;

EXEC Ventas.GetOrdenesdeCliente @clienteid      = 1,
@desdefecha = '20210101',
@hastafecha = '20220101',
@numfilas    = @rc OUTPUT;

SELECT @rc AS numfilas;
```

El código devuelve la siguiente salida, mostrando tres órdenes:

ordenid	clienteid	empid	fechaorden
10643	1	6	2021-08-25
10692	1	4	2021-10-03
10702	1	4	2021-10-13

numfilas
3

Vuelva a ejecutar el código y proporcione un ID de cliente que no existe en la tabla Ordenes (por ejemplo, ID de cliente 100). Obtendrá la siguiente salida que indica que hay cero pedidos calificados:

ordenid	clienteid	empid	fechaorden

numfilas
0

DISPARADORES (TRIGGERS)

Un trigger es un tipo especial de procedimiento almacenado, uno que no se puede ejecutar explícitamente. En su lugar, se adjunta a un evento. Cada vez que se produce el evento, el trigger se dispara y el código del trigger se ejecuta. SQL Server admite la asociación de desencadenantes con dos tipos de eventos: eventos de manipulación de datos (triggers DML) como *INSERT* y eventos de definición de datos (triggers DDL) como *CREATE TABLE*.

Puede usar los triggers para muchos propósitos, incluida la auditoría, hacer cumplir las reglas de integridad que no se pueden aplicar con restricciones y hacer cumplir políticas.

Un trigger se considera parte de la transacción que incluye el evento que provocó que se disparara el trigger. La emisión de un comando *ROLLBACK TRAN* dentro del código del trigger provoca una reversión de todos los cambios que tuvieron lugar en el activador, y también de todos los cambios que tuvieron lugar en la transacción asociada con el activador.

Los triggers en SQL Server se activan por instrucción y no por fila modificada.

TRIGGERS DML

SQL Server admite dos tipos de triggers DML: *after* y *instead of*. Un trigger *after* se dispara después del evento al que está asociado y se puede definir solo en tablas permanentes. Un trigger *instead of* se activa en lugar del evento al que está asociado y se puede definir en tablas y vistas permanentes.

En el código del trigger, podemos acceder a las pseudo tablas llamadas *inserted* y *deleted* que contienen las filas que se vieron afectadas por la modificación que provocó el disparo del trigger. La tabla *inserted* contiene la nueva imagen de las filas afectadas en el caso de las acciones *INSERT* y *UPDATE*. La tabla *deleted* contiene la imagen antigua de las filas afectadas en el caso de las acciones *DELETE* y *UPDATE*. Recuerde que las acciones *INSERT*,



UPDATE y *DELETE* pueden invocarse mediante las instrucciones *INSERT*, *UPDATE* y *DELETE*, así como con la instrucción *MERGE*. En el caso de los triggers *instead of*, las tablas *inserted* y *deleted* contienen las filas que supuestamente se verían afectadas por la modificación que provocó el disparo del trigger.

El siguiente ejemplo simple de un trigger *after* audita los datos que se insertan a una tabla. Ejecute el siguiente código para crear una tabla llamada *dbo.T1* en la base de datos actual, y otra tabla llamada *dbo.T1_Audit* que contiene información de auditoría para inserciones en *T1*:

```
DROP TABLE IF EXISTS dbo.T1_Audit, dbo.T1;

CREATE TABLE dbo.T1
(
keycol INT      NOT NULL PRIMARY KEY,
datacol VARCHAR(10) NOT NULL
);

CREATE TABLE dbo.T1_Audit (
audit_lsn     INT      NOT NULL IDENTITY PRIMARY KEY,
dt      DATETIME2(3) NOT NULL DEFAULT(SYSDATETIME()),
login_name sysname   NOT NULL DEFAULT(ORIGINAL_LOGIN()),
keycol INT      NOT NULL,
datacol      VARCHAR(10)  NOT NULL
);
```

En la tabla de auditoría, la columna *audit_lsn* tiene una propiedad de identidad y representa un número de serie del registro de auditoría. La columna *dt* representa la fecha y la hora de la inserción, utilizando la expresión predeterminada *SYSDATETIME()*. La columna *login_name* representa el nombre del inicio de sesión que realizó la inserción, utilizando la expresión predeterminada *ORIGINAL_LOGIN()*.

A continuación, ejecute el siguiente código para crear el trigger *AFTER INSERT* *trg_T1_insert_audit* en la tabla *T1* para auditar las inserciones:

```
CREATE TRIGGER trg_T1_insert_audit ON dbo.T1 AFTER INSERT AS
SET NOCOUNT ON;

INSERT INTO dbo.T1_Audit(keycol, datacol)
SELECT keycol, datacol FROM inserted;
GO
```

Como puede ver, el trigger simplemente inserta en la tabla de auditoría el resultado de una consulta en la tabla *inserted*. Los valores de las columnas en la tabla de auditoría que no se enumeran explícitamente en la instrucción *INSERT* son generados por las expresiones predeterminadas descritas anteriormente. Para probar el disparador, ejecute el siguiente código:

```
INSERT INTO dbo.T1(keycol, datacol)
VALUES(10, 'a');

INSERT INTO dbo.T1(keycol, datacol)
VALUES(30, 'x');

INSERT INTO dbo.T1(keycol, datacol)
VALUES(20, 'g');
```

El trigger se dispara luego de cada sentencia. A continuación, consultemos la tabla de auditoría:

```
SELECT audit_lsn, dt, login_name, keycol, datacol
FROM dbo.T1_Audit;
```



Obtendremos la siguiente salida:

audit_lsn	dt	login_name	keycol	datacol
1	2018-11-04 21:22:12.706	PRUEBA\user1	10	a
2	2018-11-04 21:22:12.706	PRUEBA\user1	30	x
3	2018-11-04 21:22:12.706	PRUEBA\user1	20	g

Cuando hayas terminado, ejecuta el siguiente código para la limpieza:

```
DROP TABLE dbo.T1_Audit, dbo.T1;
```

TRIGGERS DDL

SQL Server admite los triggers DDL, que se pueden usar para fines como la auditoría, la aplicación de políticas y la administración de cambios. El producto SQL Server box admite la creación de triggers DDL en dos ámbitos, el ámbito de la base de datos y el ámbito del servidor, en función del ámbito del evento. La base de datos SQL de Azure actualmente solo admite triggers de base de datos.

Crea un trigger de base de datos para eventos con un ámbito de base de datos, como *CREATE TABLE*. Crea un trigger de servidor para eventos con un ámbito de servidor, como *CREATE DATABASE*. SQL Server admite solo triggers *after* DDL; no admite triggers *instead of* DDL.

Dentro del trigger, obtiene información sobre el evento que provocó que el activador se activara al consultar una función llamada *EVENTDATA*, que devuelve la información del evento como una instancia XML. Puede usar expresiones de XQuery para extraer atributos de eventos como la hora de publicación, el tipo de evento y el nombre de inicio de sesión de la instancia XML.

El siguiente código crea la tabla *dbo.AuditDDLEvents*, que contiene la información de auditoría:

```
DROP TABLE IF EXISTS dbo.AuditDDLEvents;

CREATE TABLE dbo.AuditDDLEvents (
audit_lsn      INT      NOT NULL IDENTITY,
posttime       DATETIME2(3) NOT NULL,
eventtype      sysname   NOT NULL,
loginname      sysname   NOT NULL,
schemaname     sysname   NOT NULL,
objectname     sysname   NOT NULL,
targetobjectname sysname  NULL,
eventdata      XML      NOT NULL,
CONSTRAINT PK_AuditDDLEvents PRIMARY KEY(audit_lsn)
);
```

Observe que la tabla tiene una columna llamada *eventdata* que tiene un tipo de datos XML. Además de los atributos individuales que el activador extrae de la información del evento y almacena en atributos individuales, también almacena la información completa del evento en la columna *eventdata*.

Ejecute el siguiente código para crear el trigger de auditoría *trg_audit_ddl_events* en la base de datos utilizando el grupo de eventos *DDL_DATABASE_LEVEL_EVENTS*, que representa todos los eventos DDL en el nivel de la base de datos:



```
CREATE TRIGGER trg_audit_ddl_events
ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS
AS
SET NOCOUNT ON;

DECLARE @eventdata AS XML = eventdata();

INSERT INTO dbo.AuditDDEvents(
posttime, eventtype, loginname, schemaname, objectname, targetobjectname, eventdata)
VALUES(
@eventdata.value('/EVENT_INSTANCE/PostTime')[1],      'VARCHAR(23)'),
@eventdata.value('/EVENT_INSTANCE/EventType')[1],       'sysname'),
@eventdata.value('/EVENT_INSTANCE/LoginName')[1],       'sysname'),
@eventdata.value('/EVENT_INSTANCE/SchemaName')[1],     'sysname'),
@eventdata.value('/EVENT_INSTANCE/ObjectName')[1],     'sysname'),
@eventdata.value('/EVENT_INSTANCE/TargetObjectName')[1], 'sysname'),
@eventdata);
GO
```

El código del trigger primero almacena la información del evento obtenida de la función *EVENTDATA* en la variable *@eventdata*. El código luego inserta una fila en la tabla de auditoría con los atributos extraídos mediante el uso de expresiones XQuery por el método *.value* de la información del evento, más la instancia XML con la información completa del evento. (Para obtener detalles sobre el lenguaje de XQuery, consulte el siguiente artículo de Wikipedia: <https://en.wikipedia.org/wiki/XQuery>).

Para probar el trigger, ejecute el siguiente código, que contiene algunas declaraciones DDL:

```
CREATE TABLE dbo.T1(
col1 INT NOT NULL PRIMARY KEY
);

ALTER TABLE dbo.T1 ADD col2 INT NULL;

ALTER TABLE dbo.T1 ALTER COLUMN col2 INT NOT NULL;

CREATE NONCLUSTERED INDEX idx1 ON dbo.T1(col2);
```

A continuación, ejecute el siguiente código para consultar la tabla de auditoría:

```
SELECT * FROM dbo.AuditDDEvents;
```

Obtenemos la siguiente salida:

audit_lsn	posttime	eventtype	loginname	schemaname	objectname	targetobjectname	eventdata
1	2018-11-04 21:44:44.930	CREATE_TABLE	PRUEBA\user11	dbo	T1	NULL	<EVENT_INSTANCE>...
2	2018-11-04 21:44:44.977	ALTER_TABLE	PRUEBA\user11	dbo	T1	NULL	<EVENT_INSTANCE>...
3	2018-11-04 21:44:44.977	ALTER_TABLE	PRUEBA\user11	dbo	T1	NULL	<EVENT_INSTANCE>...
4	2018-11-04 21:44:44.980	CREATE_INDEX	PRUEBA\user11	dbo	idx1	T1	<EVENT_INSTANCE>...

Al finalizar, ejecuten el siguiente código para limpieza:

```
DROP TRIGGER IF EXISTS trg_audit_ddl_events ON DATABASE;

DROP TABLE IF EXISTS dbo.AuditDDEvents;
```



MANEJO DE ERRORES

SQL Server proporciona herramientas para manejar los errores en el código T-SQL. La principal herramienta utilizada para el manejo de errores es una construcción llamada *TRY...CATCH*. SQL Server también proporciona un conjunto de funciones que podemos invocar para obtener información sobre el error. Comenzaremos con un ejemplo básico que demuestra el uso de *TRY...CATCH*, seguido de un ejemplo más detallado que demuestra el uso de las funciones de error.

Trabajamos con el *TRY...CATCH* colocando el código T-SQL habitual en un bloque *TRY* (entre las palabras clave *BEGIN TRY* y *END TRY*) y colocando todo el código de manejo de errores en el bloque *CATCH* adyacente (entre las palabras clave *BEGIN CATCH* y *END CATCH*). Si el bloque *TRY* no tiene ningún error, el bloque *CATCH* simplemente se omite. Si el bloque *TRY* tiene un error, el control se pasa al bloque **CATCH** correspondiente. Tengan en cuenta que si un bloque *TRY...CATCH* captura y maneja un error, en lo que concierne a la persona que llama, no hubo ningún error.

Ejecute el siguiente código para demostrar un caso sin error en el bloque *TRY*:

```
BEGIN TRY
    PRINT 10/2;
    PRINT 'No error';
END TRY
BEGIN CATCH
    PRINT 'Error';
END CATCH;
```

Todo el código en el bloque *TRY* se completó con éxito; por lo tanto, se omitió el bloque *CATCH*.

Este código genera la siguiente salida:

```
5
No error
```

A continuación, corremos un código similar, pero esta vez con una división por cero. Un error ocurre:

```
BEGIN TRY
    PRINT 10/0;
    PRINT 'No error';
END TRY
BEGIN CATCH
    PRINT 'Error';
END CATCH;
```

Cuando ocurrió el error de división por cero en la primera instrucción *PRINT* en el bloque *TRY*, el control se pasó al bloque *CATCH* correspondiente. La segunda instrucción *PRINT* en el bloque *TRY* no se ejecutó. Por lo tanto, este código genera el siguiente resultado:

```
Error
```

Típicamente, el manejo de errores involucra algún trabajo en el bloque *CATCH* investigando la causa del error y tomando un curso de acción. SQL Server le proporciona información sobre el error a través de un conjunto de funciones. La función *ERROR_NUMBER* devuelve un entero con el número del error. El bloque *CATCH* generalmente incluye un código de flujo que inspecciona el número de error para determinar qué medidas tomar. La función *ERROR_MESSAGE* devuelve texto de mensaje de error. Para obtener la lista de números de error y mensajes, consulte la vista del catálogo *sys.messages*. Las funciones *ERROR_SEVERITY* y *ERROR_STATE* devuelven la gravedad y el estado del error. La función *ERROR_LINE* devuelve el número de línea en el código donde ocurrió el error.



Finalmente, la función *ERROR_PROCEDURE* devuelve el nombre del procedimiento en el que ocurrió el error y devuelve *NULL* si el error no ocurrió dentro de un procedimiento.

Para demostrar un ejemplo de manejo de errores más detallado que incluye el uso de las funciones de error, primero ejecute el siguiente código, que crea una tabla llamada *dbo.Empleados* en la base de datos actual:

```
DROP TABLE IF EXISTS dbo.Empleados;

CREATE TABLE dbo.Empleados (
empid INT NOT NULL,
nombre VARCHAR(25) NOT NULL,
jefeid INT NULL,
CONSTRAINT PK_Empleados PRIMARY KEY(empid),
CONSTRAINT CHK_Empleados_empid CHECK(empid > 0),
CONSTRAINT FK_Empleados_Empleados
FOREIGN KEY(jefeid) REFERENCES dbo.Empleados(empid)
);
```

El siguiente código inserta una nueva fila en la tabla *Empleados* en un bloque *TRY*, y si ocurre un error, muestra cómo identificar el error al inspeccionar la función *ERROR_NUMBER* en el bloque *CATCH*. El código utiliza el control de flujo para identificar y manejar los errores que desea tratar en el bloque *CATCH* y, de lo contrario, vuelve a lanzar el error.

El código también imprime los valores de las otras funciones de error simplemente para mostrar qué información está disponible cuando se produce un error:

```
BEGIN TRY

    INSERT INTO dbo.Empleados(empid, nombre, jefeid)
    VALUES(1, 'Emp1', NULL);
    -- Intentar también con empid = 0, 'A', NULL

END TRY

BEGIN CATCH
    IF ERROR_NUMBER() = 2627
        BEGIN
            PRINT 'Manejando violacion de PK...';
        END;
    ELSE
        IF ERROR_NUMBER() = 547
            BEGIN
                PRINT 'Manejando violacion de constraint CHECK/FK...';
            END;
        ELSE
            IF ERROR_NUMBER() = 515
                BEGIN
                    PRINT 'Manejando violaciona NULL...';
                END;
            ELSE
                IF ERROR_NUMBER() = 245
                    BEGIN
                        PRINT 'Manejando error de conversion...';
                    END;
                ELSE
                    BEGIN
                        PRINT 'Re-lanzando error...';
                        THROW;
                    END;
    END;
    PRINT 'Numero de Error : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
```



```
PRINT 'Mensaje de Error : ' + ERROR_MESSAGE();
PRINT 'Severidad de Error: ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT 'Estado de Error : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT 'Linea de Error : ' + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT 'Proc de Error : ' + COALESCE(ERROR_PROCEDURE(), 'No fue dentro de un proc');
END CATCH;
```

Cuando ejecuta este código por primera vez, la nueva fila se inserta en la tabla de empleados con éxito y, por lo tanto, se omite el bloque `CATCH`. Obtenemos la siguiente salida:

```
(1 row affected)
```

Cuando ejecuta el mismo código por segunda vez, la instrucción `INSERT` falla, el control pasa al bloque `CATCH` y se identifica un error de violación de clave principal. Obtenemos la siguiente salida:

```
(0 rows affected)
    Manejando violacion de PK...
    Numero de Error : 2627
    Mensaje de Error : Violation of PRIMARY KEY constraint 'PK_Empieados'. Cannot insert
duplicate key in object 'dbo.Empieados'. The duplicate key value is (1).
    Severidad de Error: 14
    Estado de Error : 1
    Linea de Error : 3
    Proc de Error : No fue dentro de un proc
```

Para ver otros errores, ejecute el código con los valores 0, 'A' y NULL como ID de empleado.

Aquí, para fines de demostración, usamos sentencias `PRINT` como las acciones cuando se identificó un error. Por supuesto, el manejo de errores generalmente involucra más que solo imprimir un mensaje que indica que el error fue identificado.

Tenga en cuenta que puede crear un procedimiento almacenado que encapsule un código de manejo de errores reutilizable como este:

```
DROP PROC IF EXISTS dbo.ErrInsertHandler;
GO

CREATE PROC dbo.ErrInsertHandler AS
SET NOCOUNT ON;

IF ERROR_NUMBER() = 2627
BEGIN
    PRINT 'Manejando violacion de PK...';
END;
ELSE
    IF ERROR_NUMBER() = 547
        BEGIN
            PRINT 'Manejando violacion de constraint CHECK/FK...';
        END;
    ELSE
        IF ERROR_NUMBER() = 515
            BEGIN
                PRINT 'Manejando violacion NULL...';
            END;
        ELSE
            IF ERROR_NUMBER() = 245
                BEGIN
                    PRINT 'Manejando error de conversion...';
                END;
```



```
PRINT 'Número de Error : ' + CAST(ERROR_NUMBER() AS VARCHAR(10));
PRINT 'Mensaje de Error : ' + ERROR_MESSAGE();
PRINT 'Severidad de Error: ' + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT 'Estado de Error : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT 'Línea de Error : ' + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT 'Proc de Error : ' + COALESCE(ERROR_PROCEDURE(), 'No fue en un proc');
GO
```

En el bloque *CATCH*, verificamos si el número de error es uno de los que deseamos tratar localmente. Si es así, simplemente ejecute el procedimiento almacenado; De lo contrario, vuelve a lanzar el error:

```
BEGIN TRY
    INSERT INTO dbo.Empleados(empid, nombre, jefeid)
    VALUES(1, 'Emp1', NULL);
END TRY

BEGIN CATCH
    IF ERROR_NUMBER() IN (2627, 547, 515, 245)
        EXEC dbo.ErrInsertHandler;
    ELSE
        THROW;
END CATCH;
```

De este modo, podemos mantener rutinas de manejo de error reusables.



ACCESO A BASES DE DATOS DESDE LENGUAJES DE PROGRAMACIÓN

Otro aspecto de las bases de datos y la programación es el acceso a las bases de datos desde los programas que realizamos.

Vamos a ver como ejemplo dos lenguajes, Python¹⁴ y Node.js¹⁵, y veamos cómo podemos realizar las operaciones básicas sobre la base de datos.

El primer aspecto para considerar es la conexión. En Python podemos utilizar una librería llamada pyodbc¹⁶. En Node.js utilizaremos un paquete llamado mssql¹⁷.

Python	Node.js
<pre>import pyodbc # Configura la conexión server = 'localhost' database = 'TSQLV6ES' username = 'demo' password = 'Y8esAMa7dW' connection_string = f'DRIVER={{SQL Server}};SERVER={server};DATABASE={database};UID={username};PWD={password}' # Conectate a la base de datos conn = pyodbc.connect(connection_string)</pre>	<pre>const sql = require('mssql'); // Configura la conexión const config = { user: 'demo', password: 'Y8esAMa7dW', server: 'localhost', database: 'TSQLV6ES' }; // Conectate a la base de datos sql.connect(config).then(pool => { return pool.request().query('SELECT clienteid, nombreempresa FROM Ventas.Clientes'); }).then(result => { console.log(result.recordset); }).catch(err => { console.error('Error al conectar a la base de datos:', err); });</pre>

Noten que en Python simplemente establecemos la conexión, pero en Node.js además ya realizamos una consulta a la base y mostramos el resultado por consola.

Una vez establecida la conexión podemos recuperar información de la base de datos mediante una estructura llamada cursor en Python y pool en Node.js. Por ejemplo, si queremos obtener la lista de clientes, podemos utilizar:

Python	Node.js
<pre>try: cursor = conn.cursor() cursor.execute('SELECT clienteid, nombreempresa FROM Ventas.Clientes') for row in cursor.fetchall(): print(f'{row.clienteid}: {row.nombreempresa}') except pyodbc.Error as e: print(e)</pre>	<pre>const query = 'SELECT clienteid, nombreempresa FROM Ventas.Clientes'; pool.request().query(query).then(result => { console.log(result.recordset); }).catch(err => { console.error('Error en la consulta:', err); });</pre>

¹⁴ <https://www.python.org/>

¹⁵ <https://nodejs.org/>

¹⁶ <https://pypi.org/project/pyodbc/>

¹⁷ <https://www.npmjs.com/package/mssql>



En el ejemplo de Python se define el cursor, y mediante execute se le pasa la instrucción SQL a ejecutar, luego mediante for y fetchall se recorre el resultado de la consulta y se realiza la impresión de los datos obtenidos.

En Node.js en query se define la consulta y mediante pool.request().query se obtiene el resultado en result.

Para agregar un nuevo cliente, podemos ejecutar el siguiente ejemplo:

Python

```
nuevo_cliente = ('Empresa XX', 'Juan Perez',  
'Ventas', 'Blanco 123', 'Buenos Aires',  
'Argentina', '541155555555')  
try:  
    cursor.execute('INSERT INTO Ventas.Clientes  
(nombreempresa, nombrecontacto, cargocontacto,  
direccion, ciudad, pais, telefono) VALUES (?,  
?, ?, ?, ?, ?)', nuevo_cliente)  
    conn.commit()  
except pyodbc.Error as e:  
    print(e)
```

Node.js

```
const nuevo_cliente = {  
    nombreempresa: 'Empresa XX',  
    nombrecontacto: 'Juan Perez',  
    cargocontacto: 'Ventas',  
    direccion: 'Blanco 123',  
    ciudad: 'Buenos Aires',  
    pais: 'Argentina',  
    telefono: '541155555555'  
};  
const insertQuery = 'INSERT INTO  
Ventas.Clientes (nombreempresa, nombrecontacto,  
cargocontacto, direccion, ciudad, pais,  
telefono) VALUES (@nombreempresa,  
@nombrecontacto, @cargocontacto, @direccion,  
@ciudad, @pais, @telefono)';  
pool.request().input('nombreempresa',  
sql.VarChar, nuevo_cliente.nombreempresa)  
    .input('nombrecontacto', sql.VarChar,  
nuevo_cliente.nombrecontacto)  
    .input('cargocontacto', sql.VarChar,  
nuevo_cliente.cargocontacto)  
    .input('direccion', sql.VarChar,  
nuevo_cliente.direccion)  
    .input('ciudad', sql.VarChar,  
nuevo_cliente.ciudad)  
    .input('pais', sql.VarChar,  
nuevo_cliente.pais)  
    .input('telefono', sql.VarChar,  
nuevo_cliente.telefono)  
    .query(insertQuery).then(result => {  
        console.log('Cliente agregado  
correctamente');  
    }).catch(err => {  
        console.error('Error al insertar  
cliente:', err);  
    });
```

Para modificar la información, por ejemplo, la dirección podemos:

Python

```
direccion_actualizada = 'Mitre 456'  
try:  
    cursor.execute('UPDATE Ventas.Clientes SET  
direccion = ? WHERE clienteid = ?',  
direccion_actualizada, 92)  
    conn.commit()  
except pyodbc.Error as e:  
    print(e)
```

Node.js

```
const direccion_actualizada = 'Mitre 456';  
const updateQuery = 'UPDATE Ventas.Clientes SET  
direccion = @direccion WHERE clienteid =  
@clienteid';  
pool.request().input('direccion', sql.VarChar,  
direccion_actualizada)  
    .input('clienteid', sql.Int, 92)  
    .query(updateQuery).then(result => {  
        console.log('Email actualizado  
correctamente');  
    }).catch(err => {  
        console.error('Error al actualizar  
direccion:', err);  
    });
```

Y para eliminar un cliente podemos:



Python	Node.js
<pre>cliente_a_borrar = 92 try: cursor.execute('DELETE FROM Ventas.Clientes WHERE clienteid = ?', cliente_a_borrar) conn.commit() except pyodbc.Error as e: print(e)</pre>	<pre>const cliente_a_borrar = 92; const deleteQuery = 'DELETE FROM Ventas.Clientes WHERE clienteid = @clienteid'; pool.request().input('clienteid', sql.Int, cliente_a_borrar) .query(deleteQuery).then(result => { console.log('Cliente eliminado correctamente'); }).catch(err => { console.error('Error al eliminar cliente:', err); });</pre>

Observen el detalle que en Python luego de cada una de las operaciones que modifican la base de datos se realiza una operación de commit. Esto es porque al abrir una conexión se considera que se está comenzando una transacción, y mediante el commit se confirma lo realizado hasta ese momento. Si no se realiza el commit, al finalizar la ejecución del programa se considera que la transacción fue fallida y la base de datos regresa al estado en que estaba.

En cambio, en Node.js se maneja el concepto de transacción implícita, en el que luego de cada operación se realiza el commit de manera automática. No obstante lo cual, en Node.js se pueden definir y marcar el comienzo y fin de las transacciones de manera explícita, como podemos ver en el siguiente ejemplo:

```
// Conectate a la base de datos
sql.connect(config).then(pool => {
    // Inicia una transaccion
    const transaction = new sql.Transaction(pool);
    transaction.begin().then(() => {
        // Realiza la insercion
        return new sql.Request(transaction)
            .query('INSERT INTO Ventas.Clientes (nombreampresa) VALUES (@nombreampresa)', {
                nombreampresa: 'Empresa XX'
            });
    }).then(() => {
        // Confirma la transaccion
        transaction.commit().then(() => {
            console.log('Insercion exitosa');
        });
    }).catch(err => {
        // Revierte la transaccion en caso de error
        transaction.rollback();
        console.error('Error en la insercion:', err);
    });
});
```

Para más información ver la sección Transacciones y Concurrencia

Al realizar consultas también podemos realizar consultas que combinen el resultado de varias tablas, es decir, consultas que realicen JOIN, por ejemplo, si necesitamos los productos y quien los suministra lo podemos obtener de la siguiente manera:



Python	Node.js
<pre>try: cursor.execute('SELECT p.nombreproducto, s.nombreempresa FROM Produccion.Productos p INNER JOIN Produccion.Proveedores s ON p.proveedorid = s.proveedorid') for row in cursor.fetchall(): print(f'{row.nombreproducto} - {row.nombreempresa}') except pyodbc.Error as e: print(e)</pre>	<pre>const joinQuery = 'SELECT p.nombreproducto, s.nombreempresa FROM Produccion.Productos p INNER JOIN Produccion.Proveedores s ON p.proveedorid = s.proveedorid'; pool.request().query(joinQuery).then(result => { console.log(result.recordset); }).catch(err => { console.error('Error en la consulta JOIN:', err); });</pre>

Estos son ejemplos simples en dos lenguajes. Tengan en cuenta que cada lenguaje puede tener sus particularidades en cuanto a la manera de establecer las conexiones, paso de parámetros, manipulación de los resultados, gestión de los errores y de las transacciones.

Por ejemplo, en C# se establece la conexión de esta manera:

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = "Data Source=localhost;Initial Catalog=TSQLV6ES;User
ID=demo;Password=Y8esAMa7dW";
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();
            // Ejemplo: consulta los nombres de los clientes
            string query = "SELECT clienteid, nombreempresa FROM Ventas.Clientes";
            using (SqlCommand command = new SqlCommand(query, connection))
            {
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        Console.WriteLine($"{reader["clienteid"]} - {reader["nombreempresa"]}");
                    }
                }
            }
        }
    }
}
```

En cambio, en Java, se puede hacer así:

```
import java.sql.*;

public class TSQVL6ESExample {
    public static void main(String[] args) {
        String url = "jdbc:sqlserver://localhost:1433;databaseName=TSQLV6ES";
        String user = "demo";
        String password = "Y8esAMa7dW";

        try (Connection connection = DriverManager.getConnection(url, user, password)) {
            // Ejemplo: consulta los nombres de los clientes
            String query = "SELECT clienteid, nombreempresa FROM Ventas.Clientes";
            try (Statement statement = connection.createStatement();
                 ResultSet resultSet = statement.executeQuery(query)) {
                while (resultSet.next()) {
```



```
        System.out.println(resultSet.getString("clienteid") + " - " +
resultSet.getString("nombreempresa"));
    }
}
} catch (SQLException e) {
    e.printStackTrace();
}
}
}
```

Y en GO:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/denisenkom/go-mssql"
)

func main() {
    connectionString := "server=localhost;user id=demo;password=Y8esAMa7dW;database=TSQLV6ES"
    db, err := sql.Open("mssql", connectionString)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    // Ejemplo: consulta los nombres de los clientes
    query := "SELECT clienteid, nombreempresa FROM Ventas.Clientes"
    rows, err := db.Query(query)
    if err != nil {
        panic(err)
    }
    defer rows.Close()

    for rows.Next() {
        var clienteid, nombreempresa string
        err := rows.Scan(&clienteid, &nombreempresa)
        if err != nil {
            panic(err)
        }
        fmt.Printf("%s - %s\n", clienteid, nombreempresa)
    }
}
```

Y, por último, así podría ser en COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TSQVL6ESEExample.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SQL-STATEMENT PIC X(200).
01 CLIENTEID PIC 9(5) VALUE '99999'.
01 NOMBREEMPRESA PIC X(40) VALUE 'Empresa XX'.
01 DB-CONNECTION.
    05 DB-HANDLE POINTER.
    05 DB-STATUS PIC S9(4) COMP.
01 SQL-ERROR-MSG PIC X(100).
LINKAGE SECTION.
01 CUSTOMER-INFO.
    05 CLIENTEID-IN PIC 9(5).
    05 NOMBREEMPRESA-IN PIC X(40).
PROCEDURE DIVISION.
```



```
EXEC SQL INCLUDE SQLCA END-EXEC.  
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
  
EXEC SQL CONNECT TO 'TSQLV6ES' USER 'demo' USING 'Y8esAMa7dW' END-EXEC.  
IF SQLCODE NOT = 0  
    MOVE SQLERRMC TO SQL-ERROR-MSG  
    DISPLAY 'Error al conectar a la base de datos: ' SQL-ERROR-MSG  
    GOBACK  
END-IF.  
  
EXEC SQL DECLARE C1 CURSOR FOR  
    SELECT clienteid, nombreempresa FROM Ventas.Clientes END-EXEC.  
EXEC SQL OPEN C1 END-EXEC.  
PERFORM UNTIL SQLCODE NOT = 0  
    EXEC SQL FETCH C1 INTO :CLIENTEID-IN, :NOMBREEMPRESA-IN-IN END-EXEC.  
    IF SQLCODE = 0  
        DISPLAY CLIENTEID-IN ' - ' NOMBREEMPRESA-IN-IN  
    END-IF  
END-PERFORM.  
EXEC SQL CLOSE C1 END-EXEC.  
  
EXEC SQL COMMIT END-EXEC.  
EXEC SQL DISCONNECT END-EXEC.  
  
GOBACK.
```

Los ejemplos que proporcioné pueden adaptarse a varios lenguajes de programación populares. Aquí tienen una lista de algunos de ellos:

1. **Ruby**: Pueden utilizar la gema pg para conectarte a la base de datos PostgreSQL o la gema mysql2 para MySQL. Los ejemplos de consultas y modificaciones serían similares a los de **Python**.
2. **PHP**: Utiliza la extensión mysqli o PDO para conectarte a la base de datos. Los ejemplos de consultas y modificaciones serían similares a los de **Python**.
3. **Perl**: Utiliza el módulo DBI para conectarte a la base de datos. Los ejemplos de consultas y modificaciones serían similares a los de **Python**.
4. **Swift** (para aplicaciones iOS/macOS): Utiliza el framework SQLite.swift para trabajar con bases de datos SQLite en aplicaciones Swift. Los ejemplos de consultas y modificaciones serían similares a los de **Python**.
5. **Scala**: Utiliza la biblioteca slick para interactuar con bases de datos en Scala. Los ejemplos de consultas y modificaciones serían similares a los de **Python**.

Recuerden que cada lenguaje tiene su propia sintaxis y peculiaridades, pero los conceptos básicos de acceso a bases de datos (consultas, inserciones, actualizaciones, eliminaciones) son similares en la mayoría de los casos.



ANEXO

FUNCIONES DE SISTEMA¹⁸

SQL Server incluye una serie de "Funciones del sistema", así como funciones más típicas con el producto. Algunas de estas se usan con frecuencia y son bastante claras desde el principio en términos de cómo usarlas. Otras, sin embargo, son de uso más raro y de naturaleza más críptica.

En este anexo, intentaremos aclarar el uso de la mayoría de estas funciones de manera breve y concisa.

Para tu información, en versiones anteriores, muchas funciones del sistema a menudo se denominaban "Variables globales". Este fue un nombre inapropiado, y Microsoft se ha esforzado por solucionarlo en las últimas versiones, cambiando la documentación para referirse a ellos por el nombre más apropiado de "función del sistema". De todos modos, es probable que aún exista documentación en la que se las refiera como variables globales.

Las funciones de T-SQL disponibles en SQL Server 2008 se dividen en las siguientes categorías:

- Funciones heredadas (legacy) del "sistema"
- Funciones agregadas
- Funciones de configuración
- Funciones criptográficas
- Funciones de cursor
- Funciones de fecha y hora
- Funciones matemáticas
- Funciones de metadatos
- Funciones de ranking
- Funciones de conjunto de filas (rowset)
- Funciones de seguridad
- Funciones de cadena
- Funciones del sistema
- Funciones de texto e imagen

Además, tenemos el operador **OVER**, que, si bien funciona en gran medida como una herramienta de clasificación, se puede aplicar a otras formas de funciones T-SQL (sobre todo de agregación). Si bien solo lo analizaremos como parte de las funciones de clasificación, es posible que veas que se hace referencia a varios otros lugares en este anexo.

FUNCIONES HEREDADAS (LEGACY) DEL "SISTEMA" (ANTES CONOCIDAS COMO VARIABLES GLOBALES)

@@CONNECTIONS

Devuelve el número de intentos de conexión desde la última vez que se inició SQL Server.

Este es el total de todos los intentos de conexión realizados desde la última vez que se inició SQL Server. La clave para recordar aquí es que estamos hablando de intentos, no de conexiones reales, y que estamos hablando de conexiones en lugar de usuarios.

¹⁸ Beginning Microsoft SQL Server® 2008 Programming – Robert Vieira. Apéndice A



Cada intento realizado para crear una conexión incrementa este contador independientemente de si esa conexión fue exitosa o no. El único problema con esto es que el intento de conexión debe haber llegado hasta el servidor. Si la conexión falló debido a las diferencias de NetLib o algún otro problema de red, entonces el servidor SQL ni siquiera sabría que necesita aumentar el conteo; solo cuenta si el servidor vio el intento de conexión. No importa si el intento tuvo éxito o fracasó.

También es importante comprender que estamos hablando de conexiones en lugar de intentos de inicio de sesión. Dependiendo de las aplicaciones, se pueden crear varias conexiones a el servidor, pero probablemente solo pedirá información al usuario una vez. De hecho, incluso Query Analyzer hace esto. Cuando hacemos clic en una nueva ventana, crea automáticamente otra conexión basada en la misma información de inicio de sesión.

*Esto, como una serie de otras funciones del sistema, a menudo es mejor atendido por un procedimiento almacenado del sistema, **sp_monitor**. Este procedimiento, en un solo comando, produce la información desde el número de conexiones, la CPU ocupada, hasta el número total de escrituras de SQL Server. Por lo tanto, si lo que se busca es información básica, entonces **sp_monitor** puede ser mejor; si necesitamos datos discretos que podemos manipular, entonces **@@CONNECTIONS** proporciona una pieza de datos agradable, ordenada y escalar.*

@@CPU_BUSY

Devuelve el tiempo en milisegundos que la CPU ha estado trabajando activamente desde que se inició SQL Server por última vez. Este número se basa en la resolución del temporizador del sistema, que puede variar, y por lo tanto puede variar en precisión.

Esta es otra de las funciones del tipo “desde que se inició el servidor”. Esto significa que no siempre puede contar con que el número aumente a medida que se ejecuta la aplicación. Es posible, en base a este número, calcular un porcentaje de CPU que SQL Server está ocupando. Sin embargo, de manera realista, preferiría acceder directamente al Monitor de rendimiento si lo necesitara con urgencia. La conclusión es que esta es una de esas cosas realmente geniales desde el punto de vista de "caramba, ¿no es genial saber eso?", Pero no tiene tantos usos prácticos en la mayoría de las aplicaciones.

@@IDLE

Devuelve el tiempo en milisegundos (basado en la resolución del temporizador del sistema) que SQL Server ha estado inactivo desde que se inició por última vez.

Podemos pensar en este como algo inverso a **@@CPU_BUSY**. Esencialmente, dice cuánto tiempo SQL Server ha pasado sin hacer nada. Si alguien encuentra un uso programático para este, envíeme un correo electrónico; me encantaría saberlo (no puedo pensar en uno).

@@IO_BUSY

Devuelve el tiempo en milisegundos (según la resolución del temporizador del sistema) que SQL Server ha dedicado a realizar operaciones de entrada y salida desde que se inició por última vez. Este valor se restablece cada vez que se inicia SQL Server.

Este realmente no tiene ninguna ciencia espacial, y es otro de los que encuentro que cae en la categoría de "sin uso programático real".

@@PACK RECEIVED AND @@PACK_SENT

Devuelve respectivamente el número de paquetes de entrada leídos/escritos desde/hacia la red por SQL Server desde que se inició por última vez.

Principalmente, estas son herramientas de solución de problemas de red.



@@PACKET_ERRORS

Devuelve la cantidad de errores de paquetes de red que se han producido en las conexiones a SQL Server desde la última vez que se inició SQL Server.

Principalmente una herramienta de solución de problemas de red.

@@TIMETICKS

Devuelve el número de microsegundos por tick. Esto varía según las máquinas y es otro de los que entran en la categoría de “sin uso programático real”.

@@TOTAL_ERRORS

Devuelve el número de errores de lectura/escritura de disco encontrados por SQL Server desde que se inició por última vez.

No confunda esto con errores de tiempo de ejecución o que tenga alguna relación con **@@ERROR**. Se trata de problemas con la E/S física. Este es otro de los de la variedad “no real programmatic use”. El uso principal aquí sería más similar a los scripts de diagnóstico del sistema. En términos generales, usaría el Monitor de rendimiento para esto.

@@TOTAL_READ AND @@TOTAL_WRITE

Devuelve respectivamente el número total de lecturas/escrituras de disco por parte de SQL Server desde que se inició por última vez.

Los nombres aquí son un poco engañosos, ya que no incluyen ninguna lectura del caché, son solo E/S físicas.

@@TRANCOUNT

Devuelve el número de transacciones activas, esencialmente el nivel de anidamiento de transacciones, para la conexión actual.

Este puede ser muy útil cuando estás haciendo transacciones. Normalmente no soy un gran admirador de las transacciones anidadas, pero hay ocasiones en las que son difíciles de evitar. Como tal, puede ser importante saber exactamente dónde nos encontramos en el anidamiento de transacciones (por ejemplo, puede tener una lógica que solo inicie una transacción si aún no está en una).

Si no está en una transacción, entonces **@@TRANCOUNT** es **0**. A partir de ahí, veamos un breve ejemplo:

SELECT @@TRANCOUNT As TransactionNestLevel	--Esto será cero --en este punto
BEGIN TRAN	
SELECT @@TRANCOUNT As TransactionNestLevel	--Esto será uno --en este punto
BEGIN TRAN	
SELECT @@TRANCOUNT As TransactionNestLevel	--Esto será dos --en este punto
COMMIT TRAN	
SELECT @@TRANCOUNT As TransactionNestLevel	--Esto volverá a uno --en este punto
ROLLBACK TRAN	
SELECT @@TRANCOUNT As TransactionNestLevel	--Esto volverá a cero --en este punto

Tengan en cuenta que, en este ejemplo, el **@@TRANCOUNT** al final también habría llegado a cero si tuviéramos un COMMIT como última sentencia.



FUNCIONES DE CONFIGURACIÓN

Bueno, estoy seguro de que será una completa sorpresa (bueno, no realmente...), pero las funciones de configuración son aquellas funciones que nos informan sobre las opciones tal como están configuradas para el servidor o la base de datos actual (según corresponda).

@@DATEFIRST

Devuelve el valor numérico que corresponde al día de la semana que el sistema considera el primer día de la semana.

El valor predeterminado en los Estados Unidos es 7, que equivale al domingo. Los valores se convierten de la siguiente manera:

- 1 – lunes (el primer día para la mayor parte del mundo)
- 2 — martes
- 3 — miércoles
- 4 — jueves
- 5 — viernes
- 6 — sábado
- 7 — domingo

Esto puede ser realmente útil cuando se trata de problemas de localización, por lo que se puede diseñar correctamente cualquier calendario u otra información que tenga que dependa del día de la semana.

Se usa la función **SET DATEFIRST** para modificar esta configuración.

@@DBTS

Devuelve la última marca de tiempo (timestamp) utilizada para la base de datos actual.

A primera vista, este parece actuar muy parecido a **@@IDENTITY** en el sentido de que le brinda la oportunidad de recuperar el último valor establecido por el sistema (esta vez, es la última marca de tiempo en lugar del último valor de identidad). Las cosas para tener en cuenta en este incluyen:

- El valor cambia según cualquier cambio en la base de datos, no solo en la tabla en la que está trabajando.
- Se refleja *cualquier* cambio de marca de tiempo en la base de datos, no solo los de la conexión actual.

Debido a que no se puede contar con que este valor sea realmente el último que usó (alguien más puede haber hecho algo que lo cambiaría), personalmente encuentro muy poco uso práctico para este.

@@LANGID AND @@LANGUAGE

Devuelve respectivamente el ID y el nombre del idioma actualmente en uso.

Estos pueden ser útiles para averiguar si su producto se instaló en una situación de localización o no y, de ser así, qué idioma es el predeterminado.

Para obtener una lista completa de los idiomas admitidos actualmente por SQL Server, podemos usar el procedimiento almacenado del sistema, **sp_helplanguage**.

@@LOCK_TIMEOUT

Devuelve la cantidad de tiempo actual en milisegundos antes de que el sistema agote el tiempo de espera de un recurso bloqueado.



Si un recurso (una página, una fila, una tabla, lo que sea) está bloqueado, su proceso se detendrá y esperará a que se elimine el bloqueo. Esto determina cuánto tiempo esperará su proceso antes de que se cancele el estado de cuenta.

El tiempo de espera predeterminado es **0** (que equivale a indefinidamente) a menos que alguien lo haya cambiado a nivel del sistema (usando **sp_configure**). Independientemente de cómo se establezca el valor predeterminado del sistema, obtendrá un valor de **-1** de este global a menos que haya establecido manualmente el valor para la conexión actual mediante **SET LOCK_TIMEOUT**.

@@MAX_CONNECTIONS

Devuelve el número máximo de conexiones de usuario simultáneas permitidas en su servidor SQL.

No confundamos esto con el mismo significado que vería en la propiedad Conexiones máximas en la Consola de administración. Este se basa en licencias y mostrará un número muy alto si ha seleccionado licencias "por puesto".

Tengamos en cuenta que la cantidad real de conexiones de usuario permitidas también depende de la versión de SQL Server que esté utilizando y de los límites de su(s) aplicación(es) y hardware.

@@MAX_PRECISION

Devuelve el nivel de precisión establecido actualmente para los tipos de datos numéricos y decimales.

El valor predeterminado es 38 lugares, pero el valor se puede cambiar usando la opción **/p** cuando inicia su SQL Server. El **/p** se puede agregar iniciando SQL Server desde una línea de comandos o agregándolo a los parámetros de inicio para el servicio MSSQLServer en el subprograma de servicios del sistema operativo Windows.

@@NESTLEVEL

Devuelve el nivel de anidamiento actual para los procedimientos almacenados anidados.

El primer procedimiento almacenado (sproc) que se ejecuta tiene un **@@NESTLEVEL** de **0**. Si ese sproc llama a otro, se dice que el segundo sproc está anidado en el primer sproc (y **@@NESTLEVEL** se incrementa a un valor de **1**). Asimismo, el segundo sproc podrá llamar a un tercero, y así hasta un máximo de 32 niveles de profundidad. Si supera el nivel de 32 niveles de profundidad, no solo se cancelará la transacción, sino que deberá revisar el diseño de su aplicación.

@@OPTIONS

Devuelve información sobre las opciones que se han aplicado mediante el comando **SET**.

Dado que solo obtiene un valor, pero puede tener muchas opciones configuradas, SQL Server usa banderas binarias para indicar qué valores están configurados. Para probar si la opción que nos interesa está configurada, debemos usar el valor de la opción junto con un operador bit a bit. Por ejemplo:

```
IF (@@OPTIONS & 2)
```

Si esto se evalúa como Verdadero, sabremos que **IMPLICIT_TRANSACTIONS** se ha activado para la conexión actual. Los valores son:



Bit	Opción SET	Descripción
1	DISABLE_DEF_CNST_CHK	Comprobación de restricciones provisionales frente a diferidas.
2	IMPLICIT_TRANSACTIONS	Una transacción se inicia implícitamente cuando se ejecuta una declaración.
4	CURSOR_CLOSE_ON_COMMIT	Controla el comportamiento de los cursosres después de que se haya realizado una operación COMMIT.
8	ANSI_WARNINGS	Advierte de truncamiento y NULL en agregación.
16	ANSI_PADDING	Controla el relleno de variables de longitud fija.
32	ANSI_NULLS	Determina el manejo de valores nulos cuando se usan operadores de igualdad.
64	ARITHABORT	Termina una consulta cuando se produce un desbordamiento o un error de división por cero durante la ejecución de la consulta.
128	ARITHIGNORE	Devuelve NULL cuando se produce un error de desbordamiento o división por cero durante una consulta.
256	QUOTED_IDENTIFIER	Diferencia entre comillas simples y dobles al evaluar una expresión.
512	NOCOUNT	Desactiva el mensaje de la(s) fila(s) afectada(s) devuelto(s) al final de cada sentencia.
1024	ANSI_NULL_DFLT_ON	Altera el comportamiento de la sesión para usar la compatibilidad ANSI para la nulabilidad. Las columnas creadas con tablas nuevas o añadidas a tablas antiguas sin una configuración de opción nula explícita se definen para permitir valores nulos. Mutuamente excluyente con ANSI_NULL_DFLT_OFF.
2048	ANSI_NULL_DFLT_OFF	Altera el comportamiento de la sesión para no usar la compatibilidad ANSI para la nulabilidad. Las nuevas columnas definidas sin nulabilidad explícita se definen para no permitir nulos. Mutuamente excluyente con ANSI_NULL_DFLT_ON.
4096	CONCAT_NULL_YIELDS_NULL	Devuelve un NULL al concatenar un NULL con una cadena.
8192	NUMERIC_ROUNDABORT	Genera un error cuando se produce una pérdida de precisión en una expresión.

@@REMSERVER

Devuelve el valor del servidor (como aparece en el registro de inicio de sesión) que llamó al procedimiento almacenado.

Se usa solo en procedimientos almacenados. Este es útil cuando desea que el sproc se comporte de manera diferente según el servidor remoto (a menudo una ubicación geográfica) desde el que se llamó al sproc.

@@SERVERNAME

Devuelve el nombre del servidor local desde el que se ejecuta el script.

Si tenemos varias instancias de SQL Server instaladas (un buen ejemplo sería un servicio de hospedaje web que usa una instalación de SQL Server separada para cada cliente), entonces **@@SERVERNAME** devuelve la siguiente información del nombre del servidor local si el nombre del servidor local no ha sido cambiado desde la instalación:

Instancia	Información del Servidor
Default instance	<servername>
Named instance	<servername\instancename>
Virtual server – default instance	<virtualservername>
Virtual server – named instance	<virtualservername\instancename>

@@SERVICENAME

Devuelve el nombre de la clave de registro con la que se ejecuta SQL Server. Será MSSQLService si es la instancia predeterminada de SQL Server o el nombre de la instancia, si corresponde.

@@SPID

Devuelve el ID de proceso del servidor (SPID) del proceso de usuario actual.



Esto equivale al mismo ID de proceso que ve si ejecuta **sp_who**. Lo bueno es que podemos indicarle el **SPID** de nuestra conexión actual, que puede ser utilizado por el DBA para monitorear y, si es necesario, finalizar esa tarea.

@@TEXTSIZE

Devuelve el valor actual de la opción **TEXTSIZE** de la sentencia **SET**, que especifica la longitud máxima, en bytes, devuelta por una sentencia **SELECT** cuando se trata de datos de texto o imagen.

El valor predeterminado es 4096 bytes (4 KB). Puede cambiar este valor utilizando la sentencia **SET TEXTSIZE**.

@@VERSION

Devuelve la versión actual de SQL Server, así como el tipo de procesador y la arquitectura del sistema operativo. Por ejemplo:

```
SELECT @@VERSION
```

da:

```
-----  
Microsoft SQL Server 2022 (RTM-CU9) (KB5030731) - 16.0.4085.2 (X64)  
Sep 27 2023 12:05:43  
Copyright (C) 2022 Microsoft Corporation  
Developer Edition (64-bit) on Linux (Ubuntu 20.04.6 LTS) <X64>
```

Desafortunadamente, esto no devuelve la información estructurada, por lo que debemos analizarla si deseamos usarla para probar información específica.

Sería mejor usar el sproc del sistema **xp_msver** en su lugar; devuelve información de tal manera que puede recuperar más fácilmente información específica de los resultados.

FUNCIONES DE CURSOR

Estas proporcionan diversa información sobre el estado o la naturaleza de un cursor dado.

@@CURSOR_ROWS

Devuelve cuántas filas hay actualmente en el último conjunto de cursores abierto en la conexión actual. Tengamos en cuenta que esto es para cursores, no para tablas temporales.

Este número se restablece cada vez que abre un nuevo cursor. Si necesitamos abrir más de un cursor a la vez y necesitamos saber la cantidad de filas en el primer cursor, deberemos mover este valor a una variable de retención antes de abrir los cursores posteriores.

Es posible usar esto para configurar un contador para controlar un bucle **WHILE** cuando se trata de cursores, pero recomiendo encarecidamente que no se haga esta práctica: el valor contenido en **@@CURSOR_ROWS** puede cambiar según el tipo de cursor y si SQL Server está llenando el cursor asíncrónicamente o no. Usar **@@FETCH_STATUS** será mucho más confiable y al menos igual de fácil de usar.

Si el valor devuelto es un número negativo mayor que **-1**, debe estar trabajando con un cursor asíncrono y el número negativo es el número de registros creados hasta el momento en el cursor. Sin embargo, si el valor es **-1**, entonces el cursor es un cursor dinámico, ya que el número de filas cambia constantemente. Un valor devuelto de **0** le informa que no se ha abierto ningún cursor o que el último cursor abierto ya no está abierto. Finalmente, cualquier número positivo indica el número de filas dentro del cursor.



Para crear un cursor asíncrono, establecemos el umbral del cursor con **sp_configure cursor threshold** en un valor superior a **0**. Luego, cuando el cursor supera esta configuración, se devuelve el cursor, mientras que los registros restantes se colocan en el cursor de forma asíncrona.

@@FETCH_STATUS

Devuelve un indicador del estado de la última operación **FETCH** del cursor.

Si usamos cursos, usaremos **@@FETCH_STATUS**. Así es como sabemos el éxito o el fracaso del intento de navegar a un registro en el cursor. Devolverá una constante dependiendo de si SQL Server tuvo éxito en su última operación **FETCH** o no y, si **FETCH** falló, por qué. Las constantes son:

- **0** – Éxito
- **-1** – Error. Por lo general, porque está más allá del principio o el final del conjunto de cursos.
- **-2** – Error. No se encontró la fila que estaba buscando, generalmente porque se eliminó entre el momento en que se creó el conjunto de cursos y cuando navegó a la fila actual. Solo debe ocurrir en cursos desplazables y no dinámicos.

Para facilitar la lectura, a menudo configuraremos algunas constantes antes de usar **@@FETCH_STATUS**. Por ejemplo:

```
DECLARE @NOTFOUND int  
DECLARE @BEGINEND int
```

```
SELECT @NOTFOUND = -2  
SELECT @BEGINEND = -1
```

Luego podemos usarlos en el condicional en la declaración **WHILE** del bucle de cursor en lugar de solo el número entero de fila. Esto puede hacer que el código sea un poco más legible.

CURSOR_STATUS

La función **CURSOR_STATUS** permite que la persona que llama a un procedimiento almacenado determine si ese procedimiento ha devuelto un cursor y un conjunto de resultados. La sintaxis es la siguiente:

```
CURSOR_STATUS (  
{ '<Local>', '<cursor name>' }  
| { '<global>', '<cursor name>' }  
| { '<variable>', '<cursor variable>' }  
)
```

local, **global** y **variable** especifican constantes que indican el origen del cursor. **local** equivale a un nombre de cursor local, **global** a un nombre de cursor global y **variable** a una variable local.

Si estamos utilizando la forma de **nombre del cursor**, hay cuatro posibles valores de retorno:

- **1** – el cursor está abierto. Si el cursor es dinámico, su conjunto de resultados tiene cero o más filas. Si el cursor no es dinámico, tiene una o más filas.
- **0** – el conjunto de resultados del cursor está vacío.
- **-1** – el cursor está cerrado.
- **-3** – no existe un cursor con ese **nombre de cursor**.

Si está utilizando la forma de **variable de cursor**, hay cinco posibles valores de retorno:

- **1** – el cursor está abierto. Si el cursor es dinámico, su conjunto de resultados tiene cero o más filas. Si el cursor no es dinámico, tiene una o más filas.



- **0** – el conjunto de resultados está vacío.
- **-1** – el cursor está cerrado.
- **-2** – no hay ningún cursor asignado a la **variable de cursor**.
- **-3** – la variable con nombre cursor variable no existe o, si existe, aún no se le ha asignado un cursor.

FUNCIONES BÁSICAS DE METADATOS

Las funciones de metadatos proporcionan información sobre la base de datos y los objetos de la base de datos. Ellas son:

- COL_LENGTH
- COL_NAME
- COLUMNPROPERTY
- DATABASEPROPERTY
- DATABASEPROPERTYEX
- DB_ID
- DB_NAME
- FILE_ID
- FILE_NAME
- FILEGROUP_ID
- FILEGROUP_NAME
- FILEGROUOPROPERTY
- FILEPROPERTY
- FULLTEXTCATALOGPROPERTY
- FULLTEXTSERVICEPROPERTY
- INDEX_COL
- INDEXKEY_PROPERTY
- INDEXPROPERTY
- OBJECT_ID
- OBJECT_NAME
- OBJECTPROPERTY
- OBJECTPROPERTYEX
- @@PROCID
- SCHEMA_ID
- SCHEMA_NAME
- SQL_VARIANT_PROPERTY
- TYPE_ID
- TYPE_NAME
- TYPEPROPERTY

COL_LENGTH

La función **COL_LENGTH** devuelve la longitud definida de una columna. La sintaxis es la siguiente:

`COL_LENGTH('<tabla>', '<columna>')`

El parámetro de **columna** especifica el nombre de la columna para la que se va a determinar la longitud. El parámetro de **tabla** especifica el nombre de la tabla que contiene esa columna.

COL_NAME

La función **COL_NAME** toma un número de ID de tabla y un número de ID de columna y devuelve el nombre de la columna de la base de datos. La sintaxis es la siguiente:

`COL_NAME(<tabla_id>, <columna_id>)`



El parámetro **columna_id** especifica el número de ID de la columna. El parámetro **tabla_id** especifica el número de ID de la tabla que contiene esa columna.

COLUMNPROPERTY

La función **COLUMNPROPERTY** devuelve datos sobre un parámetro de columna o procedimiento. La sintaxis es la siguiente:

COLUMNPROPERTY(<id>, <columna>, <propiedad>)

El parámetro **id** especifica el ID de la tabla/procedimiento. El parámetro de **columna** especifica el nombre de la columna/parámetro. El parámetro de **propiedad** especifica los datos que deben devolverse para el parámetro de columna o procedimiento. El parámetro de propiedad puede ser uno de los siguientes valores:

- **AllowsNull**: permite valores **NULL**.
- **IsComputed**: la columna es una columna calculada.
- **IsCursorType**: el procedimiento es de tipo **CURSOR**.
- **IsFullTextIndexed**: la columna se indexó en full-text.
- **IsIdentity**: la columna es una columna **IDENTITY**.
- **IsIdNotForRepl**: la columna comprueba **IDENTITY NOT FOR REPLICATION**.
- **IsOutParam**: el parámetro de procedimiento es un parámetro de salida.
- **IsRowGuidCol**: la columna es una columna **ROWGUIDCOL**.
- **Precision**: la precisión del tipo de datos de la columna o el parámetro.
- **Scale**: la escala para el tipo de datos de la columna o el parámetro.
- **UseAnsiTrim**: la configuración de relleno ANSI estaba activada cuando se creó la tabla.

El valor de retorno de esta función será **1** para **Verdadero**, **0** para **Falso** y **NULL** si la entrada no fue válida, excepto para **Precision** (donde se devolverá la precisión para el tipo de datos) y **Scale** (donde se devolverá la escala).

DATABASEPROPERTY

La función **DATABASEPROPERTY** devuelve la configuración para la base de datos y el nombre de propiedad especificados. La sintaxis es la siguiente:

DATABASEPROPERTY(‘<database>’ , ‘<propiedad>’)

El parámetro **database** especifica el nombre de la base de datos para la que se devolverán los datos de la propiedad nombrada. El parámetro de **propiedad** contiene el nombre de una propiedad de la base de datos y puede tener uno de los siguientes valores:

- **IsAnsiNullDefault**: la base de datos sigue el estándar ANSI-92 para valores **NULL**.
- **IsAnsiNullsEnabled**: no se pueden evaluar todas las comparaciones realizadas con **NULL**.
- **IsAnsiWarningsEnabled**: se emiten mensajes de advertencia cuando se producen condiciones de error estándar.
- **IsAutoClose**: la base de datos libera recursos después de que el último usuario haya salido.
- **IsAutoShrink**: los archivos de la base de datos se pueden reducir de forma automática y periódica.
- **IsAutoUpdateStatistics**: se ha habilitado la opción de actualización automática de estadísticas.
- **IsBulkCopy**: la base de datos permite operaciones no *logueadas* (como las realizadas con el Programa de copia masiva – Bulk Copy).
- **IsCloseCursorsOnCommitEnabled**: se cerrarán todos los cursos que estén abiertos cuando se confirme una transacción.
- **IsDboOnly**: solo el **dbo** puede acceder a la base de datos.
- **IsDetached**: la base de datos se separó mediante una operación de separación.



- **IsEmergencyMode:** la base de datos está en modo de emergencia.
- **IsFulltextEnabled:** la base de datos se ha habilitado para full-text.
- **IsInLoad:** la base de datos se está cargando.
- **IsInRecovery:** la base de datos se está recuperando.
- **IsInStandby:** la base de datos es de solo lectura y se permite el registro de restauración.
- **IsLocalCursorsDefault:** las declaraciones de cursor están predeterminadas en **LOCAL**.
- **IsNotRecovered:** la base de datos no se pudo recuperar.
- **IsNullConcat:** la concatenación a **NULL** da como resultado **NULL**.
- **IsOffline:** la base de datos está fuera de línea.
- **IsQuotedIdentifiersEnabled:** los identificadores se pueden delimitar con comillas dobles.
- **IsReadOnly:** la base de datos está en modo de solo lectura.
- **IsRecursiveTriggersEnabled:** la activación recursiva de disparadores (Triggers) está habilitada.
- **IsShutDown:** la base de datos encontró un problema durante el inicio.
- **IsSingleUser:** la base de datos está en modo de usuario único.
- **IsSuspect:** la base de datos es sospechosa.
- **IsTruncLog:** la base de datos trunca sus puntos de control de inicio de sesión.
- **Version:** el número de versión interna del código de SQL Server con el que se creó la base de datos.

El valor de retorno de esta función será **1** para **verdadero**, **0** para **falso** y **NULL** si la entrada no fue válida, excepto para **Version** (donde la función devolverá el número de versión si la base de datos está abierta y **NULL** si la base de datos está cerrada).

DATABASEPROPERTYEX

La función **DATABASEPROPERTYEX** es básicamente un superconjunto de **DATABASEPROPERTY** y también devuelve la configuración para la base de datos y el nombre de propiedad especificados. La sintaxis es prácticamente la misma que **DATABASEPROPERTY** y es la siguiente:

DATABASEPROPERTYEX(‘<database>’ , ‘<propiedad>’)

DATABASEPROPERTYEX solo tiene algunas propiedades más disponibles, que incluyen:

- **Collation:** devuelve la intercalación predeterminada para la base de datos (recuerde, las intercalaciones también se pueden sobreescribir en el nivel de columna).
- **ComparisonStyle:** indica el estilo de comparación de Windows (por ejemplo, la distinción entre mayúsculas y minúsculas) de la intercalación en particular.
- **IsAnsiPaddingEnabled:** indica si las cadenas se rellenan con la misma longitud antes de la comparación o la inserción.
- **IsArithmaticAbortEnabled:** indica si las consultas finalizan cuando se produce un desbordamiento de datos o un error de división por cero.

DB_ID

La función **DB_ID** devuelve el número de identificación de la base de datos. La sintaxis es la siguiente:

DB_ID([‘<database_name>’])

El parámetro opcional **database_name** especifica qué número de ID de base de datos se requiere. Si no se proporciona el nombre de la base de datos, se usará la base de datos actual en su lugar.



DB_NAME

La función **DB_NAME** devuelve el nombre de la base de datos que tiene el número de identificación especificado. La sintaxis es la siguiente:

```
DB_NAME([<database_id>])
```

El parámetro opcional **database_id** especifica qué nombre de base de datos se devolverá. Si no se proporciona **database_id**, se devolverá el nombre de la base de datos actual.

FILE_ID

La función **FILE_ID** devuelve el número de ID de archivo para el nombre de archivo especificado en la base de datos actual. La sintaxis es la siguiente:

```
FILE_ID('<file_name>')
```

El parámetro **file_name** especifica el nombre del archivo para el que se requiere el ID.

FILE_NAME

La función **FILE_NAME** devuelve el nombre de archivo para el archivo con el número de ID de archivo especificado. La sintaxis es la siguiente:

```
FILE_NAME(<file_id>)
```

El parámetro **file_id** especifica el número de ID del archivo para el que se requiere el nombre.

FILEGROUP_ID

La función **FILEGROUP_ID** devuelve el número de identificación del grupo de archivos para el nombre del grupo de archivos especificado. La sintaxis es la siguiente:

```
FILEGROUP_ID('<filegroup_name>')
```

El parámetro **filegroup_name** especifica el nombre del grupo de archivos del ID de grupo de archivos requerido.

FILEGROUP_NAME

La función **FILEGROUP_NAME** devuelve el nombre del grupo de archivos para el número de identificación del grupo de archivos especificado. La sintaxis es la siguiente:

```
FILEGROUP_NAME(<filegroup_id>)
```

El parámetro **filegroup_id** especifica el ID del grupo de archivos del nombre del grupo de archivos requerido.

FILEGROUOPROPERTY

FILEGROUOPROPERTY devuelve la configuración de una propiedad de grupo de archivos especificada, dado el grupo de archivos y el nombre de la propiedad. La sintaxis es la siguiente:

```
FILEGROUOPROPERTY(<filegroup_name>, <propiedad>)
```

El parámetro **filegroup_name** especifica el nombre del grupo de archivos que contiene la propiedad que se consulta. El parámetro de **propiedad** especifica la propiedad que se consulta y puede tener uno de los siguientes valores:

- **IsReadOnly**: el nombre del grupo de archivos es de solo lectura.
- **IsUserDefinedFG**: el nombre del grupo de archivos es un grupo de archivos definido por el usuario.
- **IsDefault**: el nombre del grupo de archivos es el grupo de archivos predeterminado.



El valor de retorno de esta función será **1** para **Verdadero**, **0** para **Falso** y **NULL** si la entrada no fue válida.

FILEPROPERTY

La función **FILEPROPERTY** devuelve la configuración de una propiedad de nombre de archivo especificada, dado el nombre de archivo y el nombre de propiedad. La sintaxis es la siguiente:

FILEPROPERTY(<file_name>, <propiedad>)

El parámetro **file_name** especifica el nombre del grupo de archivos que contiene la propiedad que se consulta. El parámetro de **propiedad** especifica la propiedad que se consulta y puede tener uno de los siguientes valores:

- **IsReadOnly**: el archivo es de solo lectura.
- **IsPrimaryFile**: el archivo es el archivo principal.
- **IsLogFile**: el archivo es un archivo de registro.
- **SpaceUsed**: la cantidad de espacio utilizado por el archivo especificado.

El valor de retorno de esta función será **1** para **Verdadero**, **0** para **Falso** y **NULL** si la entrada no fue válida, excepto **SpaceUsed** (que devolverá el número de páginas asignadas en el archivo).

FULLTEXTCATALOGPROPERTY

La función **FULLTEXTCATALOGPROPERTY** devuelve datos sobre las propiedades del catálogo full-text. La sintaxis es la siguiente:

FULLTEXTCATALOGPROPERTY(<catalog_name>, <propiedad>)

El parámetro **catalog_name** especifica el nombre del catálogo full-text. El parámetro de **propiedad** especifica la propiedad que se consulta. Las propiedades que se pueden consultar son:

- **PopulateStatus**: para los cuales los posibles valores de retorno son: **0** (inactivo), **1** (relleno en curso), **2** (pausado), **3** (limitado), **4** (recuperación), **5** (apagado), **6** (relleno incremental en curso), **7** (actualizando índices).
- **ItemCount**: devuelve el número de elementos indexados full-text que se encuentran actualmente en el catálogo full-text.
- **IndexSize**: devuelve el tamaño del índice full-text en megabytes.
- **UniqueKeyCount**: devuelve el número de palabras únicas que componen el índice full-text en este catálogo.
- **LogSize**: devuelve el tamaño (en bytes) del conjunto combinado de registros de errores asociados con un catálogo full-text.
- **PopulateCompletionAge**: devuelve la diferencia (en segundos) entre la finalización del último llenado del índice full-text y el 01/01/1990 00:00:00.

FULLTEXTSERVICEPROPERTY

La función **FULLTEXTSERVICEPROPERTY** devuelve datos sobre las propiedades de nivel de servicio full-text. La sintaxis es la siguiente:

FULLTEXTSERVICEPROPERTY(<propiedad>)

El parámetro de **propiedad** especifica el nombre de la propiedad de nivel de servicio que se va a consultar. El parámetro de propiedad puede ser uno de los siguientes valores:

- **ResourceUsage**: devuelve un valor de **1** (background) a **5** (dedicado).



- **ConnectTimeOut:** devuelve el número de segundos que el servicio de búsqueda esperará a que todas las conexiones a SQL Server completen el índice de texto completo antes de que se agote el tiempo de espera.
- **IsFulltextInstalled:** devuelve **1** si el servicio full-text está instalado en la computadora y **0** en caso contrario.

INDEX_COL

La función **INDEX_COL** devuelve el nombre de la columna indexada. La sintaxis es la siguiente:

```
INDEX_COL('<tabla>', <index_id>, <key_id>)
```

El parámetro **tabla** especifica el nombre de la tabla, **index_id** especifica el ID del índice y **key_id** especifica el ID de la clave.

INDEXKEY_PROPERTY

Esta función devuelve información sobre la clave de índice.

```
INDEXKEY_PROPERTY(<tabla_id>, <index_id>, <key_id>, <propiedad>)
```

El parámetro **tabla_id** es el ID numérico del tipo de datos int, que define la tabla que desea inspeccionar. Se puede usar **OBJECT_ID** para encontrar el **tabla_id** numérico. **index_id** especifica el ID del índice y también es del tipo de datos int. **key_id** especifica la posición de la columna de índice de la clave; por ejemplo, con una clave de tres columnas, establecer este valor en 2 determinará que desea inspeccionar la columna central. Finalmente, **propiedad** es el identificador de cadena de caracteres de una de las dos propiedades cuya configuración desea encontrar. Los dos valores posibles son **ColumnId**, que devolverá el ID de la columna física, e **IsDescending**, que devolverá el orden en que se ordena la columna (**1** es descendente y **0** ascendente).

INDEXPROPERTY

La función **INDEXPROPERTY** devuelve la configuración de una propiedad de índice especificada, dado el ID de la tabla, el nombre del índice y el nombre de la propiedad. La sintaxis es la siguiente:

```
INDEXPROPERTY(<tabla_ID>, <index>, <propiedad>)
```

El parámetro de **propiedad** especifica la propiedad del índice que se va a consultar. El parámetro de propiedad puede ser uno de estos valores posibles:

- **IndexDepth:** la profundidad del índice.
- **IsAutoStatistic:** el índice se creó mediante la opción de creación automática de estadísticas de **sp_dboption**.
- **IsClustered:** el índice está agrupado.
- **IsStatistics:** el índice fue creado por la instrucción **CREATE STATISTICS** o por la opción de creación automática de estadísticas de **sp_dboption**.
- **IsUnicode:** el índice es único.
- **IndexFillFactor:** el índice especifica su propio factor de relleno.
- **IsPadIndex:** el índice especifica el espacio para dejar abierto en cada nodo interior.
- **IsFulltextKey:** el índice es la clave de texto completo de una tabla.
- **IsHypothetical:** el índice es hipotético y no se puede utilizar directamente como ruta de acceso a datos.

El valor devuelto por esta función será **1** para **Verdadero**, **0** para **Falso** y **NULL** si la entrada no fue válida, excepto **IndexDepth** (que devolverá el número de niveles que tiene el índice) e **IndexFillFactor** (que devolverá el factor de relleno utilizado cuando se creó el índice o se reconstruyó por última vez).



OBJECT_ID

La función **OBJECT_ID** devuelve el número de ID del objeto de base de datos especificado. La sintaxis es la siguiente:

```
OBJECT_ID('<objeto>')
```

OBJECT_NAME

La función **OBJECT_NAME** devuelve el nombre del objeto de base de datos especificado. La sintaxis es la siguiente:

```
OBJECT_NAME(<object_id>)
```

OBJECTPROPERTY

La función **OBJECTPROPERTY** devuelve datos sobre objetos en la base de datos actual. La sintaxis es la siguiente:

```
OBJECTPROPERTY(<id>, <propiedad>)
```

El parámetro **id** especifica el ID del objeto requerido. El parámetro de **propiedad** especifica la información requerida sobre el objeto. Se permiten los siguientes valores de **propiedad**:

CnstIsClustKey	CnstIsColumn
CnstIsDeleteCascade	CnstIsDisabled
CnstIsNonclustKey	CnstIsNotRepl
CnstIsNotTrusted	CnstIsUpdateCascade
ExecIsAfterTrigger	ExecIsAnsiNullsOn
ExecIsDeleteTrigger	ExecIsFirstDeleteTrigger
ExecIsFirstInsertTrigger	ExecIsFirstUpdateTrigger
ExecIsInsertTrigger	ExecIsInsteadOfTrigger
ExecIsLastDeleteTrigger	ExecIsLastInsertTrigger
ExecIsLastUpdateTrigger	ExecIsQuotedIdentOn
ExecIsStartup	ExecIsTriggerDisabled
ExecIsTriggerNotForRepl	ExecIsUpdateTrigger
HasAfterTrigger	HasDeleteTrigger
HasInsertTrigger	HasInsteadOfTrigger
HasUpdateTrigger	IsAnsiNullsOn
IsCheckCnst	IsConstraint
IsDefault	IsDefaultCnst
IsDeterministic	IsExecuted
IsExtendedProc	IsForeignKey
IsIndexable	IsIndexed
IsInlineFunction	IsMSShipped
IsPrimaryKey	IsProcedure
IsQueue	IsQuotedIdentOn
IsReplProc	IsRule
IsScalarFunction	IsSchemaBound
IsSystemTable	IsTable
IsTableFunction	IsTrigger
IsUniqueCnst	IsUserTable
IsView	OwnerId
TableDeleteTrigger	TableDeleteTriggerCount
TableFullTextBackgroundUpdateIndexOn	TableFulltextCatalogId
TableFullTextChangeTrackingOn	TableFulltextDocsProcessed
TableFulltextFailCount	TableFulltextItemCount
TableFulltextKeyColumn	TableFulltextPendingChanges
TableFulltextPopulateStatus	TableHasActiveFulltextIndex
TableHasCheckCnst	TableHasClustIndex
TableHasDefaultCnst	TableHasDeleteTrigger



TableHasForeignKey	TableHasForeignRef
TableHasIdentity	TableHasIndex
TableHasInsertTrigger	TableHasNonclustIndex
TableHasPrimaryKey	TableHasRowGuidCol
TableHasTextImage	TableHasTimestamp
TableHasUniqueCnst	TableHasUpdateTrigger
TableInsertTrigger	TableInsertTriggerCount
TableIsFake	TableIsLockedOnBulkLoad
TableIsPinned	TableTextInRowLimit
TableUpdateTrigger	TableUpdateTriggerCount

El valor de retorno de esta función será **1** para **Verdadero**, **0** para **Falso** y **NULL** si la entrada no fue válida, excepto por:

- **OwnerId:** devuelve el ID de usuario de la base de datos del propietario de ese objeto. Es diferente del **SchemaID** del objeto y probablemente no será tan útil en SQL Server 2005 y versiones posteriores.
- **TableDeleteTrigger, TableInsertTrigger, TableUpdateTrigger:** devuelve el ID del primer activador (trigger) con el tipo especificado. Se devuelve cero si no existe ningún trigger de ese tipo.
- **TableDeleteTriggerCount, TableInsertTriggerCount, TableUpdateTriggerCount:** devuelve el número del tipo de trigger especificado que existe para la tabla en cuestión.
- **TableFulltextCatalogId:** devuelve el ID del catálogo full-text, si lo hay, y cero si no existe un catálogo full-text para esa tabla.
- **TableFulltextKeyColumn:** devuelve el ID de columna de la columna que se utiliza como índice único para ese índice full-text.
- **TableFulltextPendingChanges:** el número de entradas que han cambiado desde que se ejecutó el último análisis full-text para esta tabla. El seguimiento de cambios debe estar habilitado para que esta función devuelva resultados útiles.
- **TableFulltextPopulateStatus:** este tiene varios valores de retorno posibles:
 - **0:** indica que el proceso full-text está actualmente inactivo.
 - **1:** actualmente se está realizando una corrida de población completa.
 - **2:** actualmente se está ejecutando una población incremental.
 - **3:** actualmente se están analizando y agregando cambios al catálogo full-text.
 - **4:** actualmente se está ejecutando algún tipo de actualización en segundo plano (como la realizada por el mecanismo de seguimiento automático de cambios).
 - **5:** una operación full-text está en curso, pero se ha acelerado (para permitir que otras solicitudes del sistema se realicen según sea necesario) o se ha pausado.

Puede usar los comentarios de esta opción para tomar decisiones sobre qué otras opciones relacionadas con full-text son apropiadas (para verificar si un relleno está en progreso para saber si otras funciones, como **TableFulltextDocsProcessed**, son válidas).

- **TableFulltextDocsProcessed:** válido solo mientras se está ejecutando la indexación full-text, devuelve el número de filas procesadas desde que se inició la tarea de procesamiento del índice full-text. Un resultado cero indica que la indexación full-text no se está ejecutando actualmente (un resultado nulo significa que la indexación full-text no está configurada para esta tabla).
- **TableFulltextFailCount:** válido solo mientras se está ejecutando la indexación full-text, devuelve el número de filas que la indexación full-text ha omitido, por algún motivo (sin indicación del motivo). Al igual que con **TableFulltextDocsProcessed**, un resultado cero indica que la tabla no se está analizando en busca de full-text y un valor nulo indica que full-text no está configurado para esta tabla.
- **TableIsPinned:** se deja solo por compatibilidad con versiones anteriores y siempre devolverá "**0**" en SQL Server 2005 y versiones posteriores.



OBJECTPROPERTYEX

OBJECTPROPERTYEX es una versión extendida de la función **OBJECTPROPERTY**.

OBJECTPROPERTYEX(<id>, <propiedad>)

Al igual que **OBJECTPROPERTY**, el parámetro id especifica el **ID** del objeto requerido. El parámetro de **propiedad** especifica la información requerida sobre el objeto. **OBJECTPROPERTYEX** admite todos los mismos valores de propiedad que **OBJECTPROPERTY** pero agrega los siguientes valores de propiedad como opciones adicionales:

- **BaseType**: devuelve el tipo de datos base de un objeto.
- **IsPrecise**: indica que su objeto no contiene cálculos imprecisos. Por ejemplo, un int o decimal es preciso, pero un float no lo es: se debe suponer que los cálculos que utilizan tipos de datos imprecisos arrojan resultados imprecisos. Tenga en cuenta que puede marcar específicamente cualquier ensamblaje .NET que produzca como preciso o no.
- **IsSystemVerified**: indica si el propio SQL Server puede verificar las propiedades **IsPrecise** e **IsDeterministic** (en lugar de que las haya configurado el usuario).
- **Schemaid**: justo lo que parece: devuelve el ID del sistema interno para un objeto dado. A continuación, puede utilizar **SCHEMA_NAME** para poner un nombre más fácil de usar en el ID del esquema.
- **SystemDataAccess**: indica si el objeto en cuestión depende de los datos de la tabla del sistema.
- **UserDataAdapter**: indica si el objeto en cuestión utiliza alguna de las tablas de usuario o datos de usuario del sistema.

@@PROCID

Devuelve el ID de procedimiento almacenado del procedimiento que se está ejecutando actualmente.

Principalmente una herramienta de resolución de problemas cuando un proceso se está ejecutando y consume una gran cantidad de recursos. Se utiliza principalmente como una función DBA.

SCHEMA_ID

Dado un nombre de esquema, devuelve el ID del sistema interno para ese esquema. Utiliza la sintaxis:

SCHEMA_ID(<schema_name>)

SCHEMA_NAME

Dado un ID de sistema de esquema interno, devuelve el nombre fácil de usar para ese esquema. La sintaxis es:

SCHEMA_NAME(<schema_id>)

SQL_VARIANT_PROPERTY

SQL_VARIANT_PROPERTY es una función poderosa y devuelve información sobre un **sql_variant**. Esta información podría ser de **BaseType, Precision, Scale, TotalBytes, Collation** o **MaxLength**. La sintaxis es:

SQL_VARIANT_PROPERTY (expression, propiedad)

Expression es una expresión de tipo **sql_variant**. La **propiedad** puede ser cualquiera de los siguientes valores:



Valor	Descripción	Tipo base de sql_variant devuelto
BaseType	Data types include: char, int, money, nchar, ntext, numeric, nvarchar, real, smalldatetime, smallint, smallmoney, text, timestamp, tinyint, uniqueidentifier, varbinary, varchar	sysname
Precision	La precisión del tipo de datos base numérico: datetime = 23 smalldatetime = 16 float = 53 real = 24 decimal (p,s) y numeric (p,s) = p money = 19 smallmoney = 10 int = 10 smallint = 5 tinyint = 3 bit = 1 Todos los demás tipos = 0	int
Scale	El número de dígitos a la derecha del punto decimal del tipo de datos de base numérica: decimal (p,s) y numeric (p,s) = s money y smallmoney = 4 datetime = 3 Todos los demás tipos = 0	int
TotalBytes	El número de bytes necesarios para contener tanto los metadatos como los datos del valor. Si el valor es superior a 900, la creación del índice fallará.	int
Collation	La intercalación del valor particular de sql_variant.	sysname
MaxLength	La longitud máxima del tipo de datos, en bytes.	int

TYPEPROPERTY

La función **TYPEPROPERTY** devuelve información sobre un tipo de datos. La sintaxis es la siguiente:

TYPEPROPERTY(<tipo>, <propiedad>)

El parámetro de **tipo** especifica el nombre del tipo de datos. El parámetro de **propiedad** especifica la propiedad del tipo de datos que se va a consultar; puede ser uno de los siguientes valores:

- **Precision**: devuelve el número de dígitos/caracteres.
- **Scale**: devuelve el número de lugares decimales.
- **AllowsNull**: devuelve **1** para **verdadero** y **0** para **falso**.
- **UsesAnsiTrim**: devuelve **1** para **Verdadero** y **0** para **Falso**.

FUNCIONES DE CONJUNTO DE FILAS (ROWSET)

Las funciones de conjunto de filas devuelven un objeto que se puede usar en lugar de una referencia de tabla en una instrucción T-SQL. Las funciones del conjunto de filas son:

- CHANGETABLE
- CONTAINSTABLE
- FREETEXTTABLE
- OPENDATASOURCE
- OPENQUERY
- OPENROWSET



- OPENXML

CHANGETABLE

Devuelve información de seguimiento de cambios para una tabla. Puede usar esta declaración para devolver todos los cambios de una tabla o cambiar la información de seguimiento de una fila específica. La sintaxis es la siguiente:

CHANGETABLE (

```
{ CHANGES table , last_sync_version  
| VERSION table , <primary_key_values> } ) [AS] table_alias [ ( column_aliases [ ,...n ] )
```

CONTAINSTABLE

La función **CONTAINSTABLE** se utiliza en consultas full-text. La sintaxis es la siguiente:

CONTAINSTABLE (<table>, {<column> | *}, '<contains_search_condition>')

FREETEXTTABLE

La función **FREETEXTTABLE** se utiliza en consultas full-text. La sintaxis es la siguiente:

FREETEXTTABLE (<table>, {<column> | *}, '<freetext_string>')

OPENDATASOURCE

La función **OPENDATASOURCE** proporciona información de conexión ad hoc. La sintaxis es la siguiente:

OPENDATASOURCE (<provider_name>, <init_string>)

El **provider_name** es el nombre registrado como ProgID del proveedor OLE DB utilizado para acceder a la fuente de datos. **Init_string** debería ser familiar para los programadores de VB, ya que es la cadena de inicialización del proveedor OLE DB. Por ejemplo, **init_string** podría verse así:

```
"User Id=wonderison;Password=JuniorBlues;DataSource=MyServerName"
```

OPENQUERY

La función **OPENQUERY** ejecuta la **consulta** de paso a través especificada en el **servidor_vinculado** especificado. La sintaxis es la siguiente:

OPENQUERY(<servidor_vinculado>, '<consulta>')

OPENROWSET

La función **OPENROWSET** accede a datos remotos desde una fuente de datos OLE DB. La sintaxis es la siguiente:

```
OPENROWSET('<provider_name>'  
{  
'<datasource>;'<user_id>;'<password>'  
| '<provider_string>'  
},  
{  
['<catalog.>'][<schema.>]<object>  
| '<query>'  
})
```

El parámetro **provider_name** es una cadena que representa el nombre descriptivo de OLE DB proporcionado como se especifica en el registro. El parámetro **data_source** es una cadena correspondiente a la fuente de datos



OLE DB requerida. El parámetro **user_id** es un nombre de usuario relevante para pasar al proveedor OLE DB. El parámetro de **password** es la contraseña asociada con el id_usuario.

El parámetro **Provider_string** es una cadena de conexión específica del proveedor y se usa en lugar de la combinación de fuente de datos, ID de usuario y contraseña.

El parámetro **catalog** es el nombre del catálogo/base de datos que contiene el objeto requerido. El parámetro **schema** es el nombre del propietario del esquema o del objeto del objeto requerido. El parámetro **object** es el nombre del objeto.

El parámetro **query** es una cadena que ejecuta el proveedor y se utiliza en lugar de una combinación de **catálogo, esquema y objeto**.

OPENXML

Al pasar un documento XML como parámetro, o al recuperar un documento XML y definir el documento dentro de una variable, **OPENXML** permite inspeccionar la estructura y devolver datos, como si el documento XML fuera una tabla. La sintaxis es la siguiente:

```
OPENXML(<idoc      int>      [in],<rowpattern>      nvarchar[in],[<flags>      byte[in]])      [WITH
(<SchemaDeclaration> | <TableName>)]
```

El parámetro **idoc_int** es la variable definida mediante el sproc del sistema **sp_xml_preparedocument**. **Rowpattern** es la definición del nodo. El parámetro **flags** especifica la asignación entre el documento XML y el conjunto de filas para devolver dentro de la instrucción **SELECT**. **SchemaDeclaration** define el esquema XML para el documento XML; si hay una tabla definida dentro de la base de datos que sigue el esquema XML, se puede usar **TableName** en su lugar.

Antes de poder utilizar el documento XML, debe prepararse mediante el procedimiento del sistema **sp_xml_preparedocument**.

FUNCIONES DE SEGURIDAD

Las funciones de seguridad devuelven información sobre usuarios y roles. Ellas son:

- HAS_DBACCESS
- IS_MEMBER
- IS_SRVROLEMEMBER
- SUSER_ID
- SUSER_NAME
- SUSER_SID
- USER
- USER_ID
- USER_NAME

HAS_DBACCESS

La función **HAS_DBACCESS** se utiliza para determinar si el usuario que ha iniciado sesión tiene acceso a la base de datos que se está utilizando. Un valor de retorno de **1** significa que el usuario tiene acceso y un valor de retorno de **0** significa que no lo tiene. Un valor de retorno **NULL** significa que el nombre de la base de datos proporcionado no es válido. La sintaxis es la siguiente:

```
HAS_DBACCESS ('<database name>')
```



IS_MEMBER

La función **IS_MEMBER** devuelve si el usuario actual es miembro del grupo de Windows NT/función de SQL Server especificado. La sintaxis es la siguiente:

```
IS_MEMBER ({'<group>' | '<role>'})
```

El parámetro **group** especifica el nombre del grupo NT y debe tener la forma **dominio\grupo**. El parámetro **role** especifica el nombre de la función de SQL Server. El rol puede ser un rol fijo de base de datos o un rol definido por el usuario, pero no puede ser un rol de servidor.

Esta función devolverá un **1** si el usuario actual es miembro del grupo o rol especificado, un **0** si el usuario actual no es miembro del grupo o rol especificado y **NULL** si el grupo o rol especificado no es válido.

IS_SRVROLEMEMBER

La función **IS_SRVROLEMEMBER** devuelve si un usuario es miembro del rol de servidor especificado. La sintaxis es la siguiente:

```
IS_SRVROLEMEMBER ('<role>' [, '<Login>'])
```

El parámetro **login** opcional es el nombre de la cuenta de inicio de sesión para verificar; el valor predeterminado es el usuario actual. El parámetro **role** especifica la función del servidor y debe ser uno de los siguientes valores posibles:

- sysadmin
- dbcreator
- diskadmin
- processadmin
- serveradmin
- setupadmin
- securityadmin

Esta función devuelve **1** si la cuenta de inicio de sesión especificada es miembro del rol especificado, **0** si el inicio de sesión no es miembro del rol y **NULL** si el rol o el inicio de sesión no son válidos.

SUSER_ID

La función **SUSER_ID** devuelve el número de ID de inicio de sesión del usuario especificado. La sintaxis es la siguiente:

```
SUSER_ID(['<Login>'])
```

El parámetro **login** es el nombre de ID de inicio de sesión del usuario especificado. Si no se proporciona ningún valor para el inicio de sesión, en su lugar se utilizará el valor predeterminado del usuario actual.

*La función del sistema **SUSER_ID** se reemplazó hace mucho tiempo por **SUSER_SID** y permanece en el producto únicamente con fines de compatibilidad con versiones anteriores. Evite usar **SUSER_ID**, ya que el valor interno que devuelve puede cambiar de un servidor a otro (el SID es mucho más confiable cuando considera que una base de datos puede restaurarse en un nuevo servidor donde un inicio de sesión dado puede tener un **SUSER_ID** diferente).*

SUSER_NAME

La función **SUSER_NAME** devuelve el nombre de ID de inicio de sesión del usuario especificado. La sintaxis es la siguiente:

```
SUSER_NAME([<server_user_id>])
```



El parámetro **server_user_id** es el número de ID de inicio de sesión del usuario especificado. Si no se proporciona ningún valor para **server_user_id**, en su lugar se utilizará el valor predeterminado del usuario actual.

La función del sistema **SUSER_NAME** se incluye en SQL Server 2000 solo para compatibilidad con versiones anteriores, por lo que, si es posible, debe usar **SUSER_SNAME** en su lugar.

SUSER_SID

La función **SUSER_SID** devuelve el número de identificación de seguridad (SID) para el usuario especificado. La sintaxis es la siguiente:

SUSER_SID([‘<login>’])

El parámetro **login** es el nombre de inicio de sesión del usuario. Si no se proporciona ningún valor para el inicio de sesión, se utilizará el usuario actual en su lugar.

SUSER_SNAME

La función **SUSER_SNAME** devuelve el nombre de identificación de inicio de sesión para el número de identificación de seguridad (SID) especificado. La sintaxis es la siguiente:

SUSER_SNAME([<server_user_sid>])

El parámetro **server_user_sid** es el SID del usuario. Si no se proporciona ningún valor para **server_user_sid**, se utilizará el del usuario actual en su lugar.

USER

La función **USER** permite insertar en una tabla un valor proporcionado por el sistema para el nombre de usuario de la base de datos del usuario actual si no se ha proporcionado ningún valor predeterminado. La sintaxis es la siguiente:

USER

USER_ID

La función **USER_ID** devuelve el número de identificación de la base de datos del usuario especificado. La sintaxis es la siguiente:

USER_ID([‘<user>’])

El parámetro **user** es el nombre de usuario que se utilizará. Si no se proporciona ningún valor para el usuario, se utiliza el usuario actual.

USER_NAME

La función **USER_NAME** es el reverso funcional de **USER_ID** y devuelve el nombre de usuario del usuario especificado en la base de datos dado un número de ID de base de datos. La sintaxis es la siguiente:

USER_NAME([‘<user_id>’])

El parámetro **user_id** es la identificación del usuario para el que desea el nombre. Si no se proporciona ningún valor para el ID de usuario, se asume que es el usuario actual.

FUNCIONES DEL SISTEMA

Las funciones del sistema, la forma más antigua de referirse a lo que Microsoft ahora llama simplemente "otros", se pueden usar para devolver información sobre valores, objetos y configuraciones con SQL Server. Las funciones son las siguientes:



- APP_NAME
- CASE
- CAST and CONVERT
- COALESCE
- COLLATIONPROPERTY
- CURRENT_TIMESTAMP
- CURRENT_USER
- DATALENGTH
- FORMATMESSAGE
- GETANSINULL
- HOST_ID
- HOST_NAME
- IDENT_CURRENT
- IDENT_INCR
- IDENT_SEED
- IDENTITY
- ISDATE
- ISNULL
- ISNUMERIC
- NEWID
- NULLIF
- PARSENAME
- PERMISSIONS
- ROWCOUNT_BIG
- SCOPE_IDENTITY
- SERVERPROPERTY
- SESSION_USER
- SESSIONPROPERTY
- STATS_DATE
- SYSTEM_USER
- USER_NAME

APP_NAME

La función **APP_NAME** devuelve el nombre de la aplicación para la sesión actual si la aplicación ha establecido uno como tipo **nvarchar**. Tiene la siguiente sintaxis:

`APP_NAME()`

CASE

La función **CASE** evalúa una lista de condiciones y devuelve uno de múltiples resultados posibles. Tiene dos formatos:

- La función **CASE simple** compara una expresión con un conjunto de expresiones simples para determinar el resultado.
- La función **CASE buscada** evalúa un conjunto de expresiones booleanas para determinar el resultado.

Ambos formatos admiten un argumento **ELSE** opcional.

FUNCIÓN CASE SIMPLE:

```
CASE <input expression>
WHEN <when expression> THEN <result expression> ELSE <else result expression>
END
```



FUNCIÓN CASE BUSCADA:

```
CASE  
WHEN <Boolean expression> THEN <result expression> ELSE <else result expression>  
END
```

CAST AND CONVERT

Estas dos funciones proporcionan una funcionalidad similar en el sentido de que ambas convierten un tipo de datos en otro tipo.

CAST:

```
CAST(<expression> AS <data type>)
```

CONVERT:

```
CONVERT (<data type>[(<length>)], <expression> [, <style>])
```

Donde **style** se refiere al estilo del formato de fecha cuando se convierte a un tipo de datos de caracteres.

COALESCE

A la función **COALESCE** se le pasa un número indefinido de argumentos y prueba la primera expresión no nula entre ellos. La sintaxis es la siguiente:

```
COALESCE(<expression> [,...n])
```

Si todos los argumentos son **NULL**, **COALESCE** devuelve **NULL**.

COLLATIONPROPERTY

La función **COLLATIONPROPERTY** devuelve la propiedad de una cotejo determinada. La sintaxis es la siguiente:

```
COLLATIONPROPERTY(<collation name>, <property>)
```

El parámetro **collation_name** es el nombre de la intercalación que desea utilizar y **property** es la propiedad de la intercalación que desea determinar. Puede ser uno de tres valores:

Propiedad	Descripción
CodePage	La página de códigos no Unicode de la intercalación.
LCID	El LCID de Windows de la intercalación. Devuelve NULL para intercalaciones de SQL.
ComparisonStyle	El estilo de comparación de Windows de la intercalación. Devuelve NULL para intercalaciones binarias o SQL.

CURRENT_USER

La función **CURRENT_USER** simplemente devuelve el usuario actual como un tipo de nombre de sistema. Es equivalente a **USER_NAME()**. La sintaxis es la siguiente:

```
CURRENT_USER
```

DATALENGTH

La función **DATALENGTH** devuelve el número de bytes utilizados para representar la **expresión** como un número entero. Es especialmente útil con los tipos de datos **varchar**, **varbinary**, **text**, **image**, **nvarchar** y **ntext** porque estos tipos de datos pueden almacenar datos de longitud variable. La sintaxis es la siguiente:



DATALENGTH(<expresión>)

@@ERROR

Devuelve el código de error de la última instrucción T-SQL que se ejecutó en la conexión actual. Si no hay error, entonces el valor será cero.

Al escribir procedimientos almacenados o disparadores, esta es una función de sistema básica: prácticamente no puede vivir sin ella.

Lo que debemos recordar con @@ERROR es que su vida útil es solo una declaración. Esto significa que, si deseamos usarla para verificar un error después de una declaración dada, entonces debemos hacer que su prueba sea la siguiente declaración o debemos moverla a una variable de retención. En general, recomiendo usar ERROR_NUMBER() en un bloque TRY...CATCH a menos que necesite soportar código anterior a SQL Server 2005.

Se puede ver una lista de todos los errores del sistema utilizando la tabla del sistema **sys.messages** en la base de datos **master**.

Para crear tus propios errores personalizados, podemos usar **sp_addmessage**.

FORMATMESSAGE

La función **FORMATMESSAGE** utiliza mensajes existentes en **sysmessages** para construir un mensaje. La sintaxis es la siguiente:

FORMATMESSAGE(<msg_number>, <param_value>[,...n])

Donde **msg_number** es el ID del mensaje en **sysmessages**.

FORMATMESSAGE busca el mensaje en el idioma actual del usuario. Si no hay una versión localizada del mensaje, se utiliza la versión en inglés de EE. UU.

GETANSINULL

La función **GETANSINULL** devuelve la nulabilidad predeterminada para una base de datos como un número entero. La sintaxis es la siguiente:

GETANSINULL([‘<database>’])

El parámetro **database** es el nombre de la base de datos para la que se devolverá la información de nulabilidad.

Cuando la nulabilidad de la base de datos dada permite valores **NULL** y la columna o el tipo de datos no está definido explícitamente, **GETANSINULL** devuelve **1**. Este es el valor predeterminado ANSI NULL.

HOST_ID

La función **HOST_ID** devuelve el ID de la estación de trabajo. La sintaxis es la siguiente:

HOST_ID()

HOST_NAME

La función **HOST_NAME** devuelve el nombre de la estación de trabajo. La sintaxis es la siguiente:

HOST_NAME()



IDENT_CURRENT

La función **IDENT_CURRENT** devuelve el último valor de identidad creado para una tabla, dentro de cualquier sesión o ámbito de esa tabla. Esto es exactamente como **@@IDENTITY** y **SCOPE_IDENTITY**; sin embargo, esto no tiene límite en el alcance de su búsqueda para devolver el valor.

La sintaxis es la siguiente:

```
IDENT_CURRENT('<table_name>')
```

table_name es la tabla para la que desea encontrar la identidad actual.

IDENT_INCR

La función **IDENT_INCR** devuelve el valor de incremento especificado durante la creación de una columna de identidad en una tabla o vista que tiene una columna de identidad. La sintaxis es la siguiente:

```
IDENT_INCR('<table_or_view>')
```

El parámetro **table_or_view** es una expresión que especifica la tabla o vista para verificar un valor de incremento de identidad válido.

IDENT_SEED

La función **IDENT_SEED** devuelve el valor inicial especificado durante la creación de una columna de identidad en una tabla o una vista que tiene una columna de identidad. La sintaxis es la siguiente:

```
IDENT_SEED('<table_or_view>')
```

El parámetro **table_or_view** es una expresión que especifica la tabla o vista para verificar un valor de incremento de identidad válido.

@@IDENTITY

Devuelve el último valor de identidad creado por la conexión actual.

Si usamos columnas IDENTITY y luego hacemos referencia a ellas como una clave externa en otra tabla, nos encontraremos usando esta todo el tiempo. Podemos crear el registro principal (generalmente el que tiene la identidad que necesitamos recuperar), y luego seleccionamos **@@IDENTITY** para saber con qué valor necesitamos relacionar los registros secundarios.

Si realizamos inserciones en varias tablas con valores IDENTITY, recordemos que el valor en **@@IDENTITY** solo será para el último valor de identidad insertado; todo lo anterior se habrá perdido, a menos que movamos el valor a una variable de retención después de cada inserción. Además, si la última tabla en la que insertamos no tenía una columna IDENTITY, **@@IDENTITY** se establecerá en **NULL**.

IDENTITY

La función **IDENTITY** se utiliza para insertar una columna de identidad en una nueva tabla. Se usa solo con una instrucción SELECT con una cláusula de tabla INTO. La sintaxis es la siguiente:

```
IDENTITY(<data_type>[, <seed>, <increment>]) AS <column_name>
```

Donde:

- **Data_type** es el tipo de datos de la columna de identidad.



- **Seed** es el valor que se asignará a la primera fila de la tabla. A cada fila subsiguiente se le asigna el siguiente valor de identidad, que es igual al último valor de **IDENTITY** más el valor de **increment**. Si no se especifica **seed** ni **increment**, ambos tienen el valor predeterminado 1.
- **Increment** es el incremento que se agrega al valor inicial para las filas sucesivas de la tabla.
- **Column_name** es el nombre de la columna que se va a insertar en la nueva tabla.

ISNULL

La función **ISNULL** comprueba una expresión para un valor **NULL** y lo reemplaza con un valor de reemplazo especificado. La sintaxis es la siguiente:

`ISNULL(<check_expression>, <replacement_value>)`

ISNUMERIC

La función **ISNUMERIC** determina si una expresión es un tipo numérico válido. La sintaxis es la siguiente:

`ISNUMERIC(<expression>)`

NEWID

La función **NEWID** crea un valor único de tipo **uniqueidentifier**. La sintaxis es la siguiente:

`NEWID()`

NULLIF

La función **NULLIF** compara dos expresiones y devuelve un valor **NULL** si las dos expresiones son iguales. La sintaxis es la siguiente:

`NULLIF(<expression1>, <expression2>)`

PARSENAME

La función **PARSENAME** devuelve la parte especificada de un nombre de objeto. La sintaxis es la siguiente:

`PARSENAME('<object_name>', <object_piece>)`

El parámetro **object_name** especifica el nombre del objeto de la parte que se va a recuperar. El parámetro **object_piece** especifica la parte del objeto a devolver. El parámetro **object_piece** toma uno de estos valores posibles:

- 1 — Nombre del objeto
- 2 — Nombre del propietario
- 3 — Nombre de la base de datos
- 4 — Nombre del servidor

PERMISSIONS

La función **PERMISSIONS** devuelve un valor que contiene un mapa de bits, que indica los permisos de declaración, objeto o columna para el usuario actual. La sintaxis es la siguiente:

`PERMISSIONS([<objectid> [, '<column>']]])`

El parámetro **objectid** especifica el ID de un objeto. El parámetro **column** opcional especifica el nombre de la columna para la que se devuelve la información de permisos.

@@ROWCOUNT

Devuelve el número de filas afectadas por la última declaración.



Uno de los globales más utilizados, uno de los usos más comunes para este es verificar errores que no son de tiempo de ejecución, es decir, elementos que son errores lógicos para un programa pero con los que SQL Server no verá ningún problema. Un ejemplo es una situación en la que estamos realizando una actualización basada en una condición, pero descubrimos que afecta a cero filas. Lo más probable es que, si el cliente envió una modificación para una fila en particular, esperaba que esa fila coincidiera con los criterios dados: cero filas afectadas es indicativo de que algo anda mal.

Sin embargo, si prueba esta función del sistema en cualquier declaración que no devuelva filas, también devolverá un valor de **0**.

ROWCOUNT_BIG

La función **ROWCOUNT_BIG** es muy similar a **@@ROWCOUNT** en que devuelve el número de filas de la última declaración. Sin embargo, el valor devuelto es del tipo de datos bigint. La sintaxis es la siguiente:

```
ROWCOUNT_BIG()
```

SCOPE_IDENTITY

La función **SCOPE_IDENTITY** devuelve el último valor insertado en una columna IDENTITY en el mismo ámbito (es decir, dentro del mismo sproc, trigger, función o batch). Esto es similar a **IDENT_CURRENT**, discutido anteriormente, aunque no se limitó a las inserciones de identidad realizadas en el mismo ámbito.

Esta función devuelve un tipo de datos **sql_variant** y la sintaxis es la siguiente:

```
SCOPE_IDENTITY()
```

SERVERPROPERTY

La función **SERVERPROPERTY** devuelve información sobre el servidor en el que se está ejecutando. La sintaxis es la siguiente:

```
SERVERPROPERTY('<propertyname>')
```

Los valores posibles para **propertyname** son:

Property Name	Valores Devueltos
Collation	El nombre de la intercalación predeterminada para el servidor.
Edition	La edición de la instancia de SQL Server instalada en el servidor. Devuelve uno de los siguientes resultados de nvarchar: 'Desktop Engine' 'Developer Edition' 'Enterprise Edition' 'Enterprise Evaluation Edition' 'Personal Edition' 'Standard Edition'
Engine Edition	La edición del motor de la instancia de SQL Server instalada en el servidor:: 1. Personal or Desktop Engine 2. Standard 3. Enterprise (returned for Enterprise, Enterprise Evaluation, and Developer)
InstanceName	El nombre de la instancia a la que está conectado el usuario.
IsClustered	Determinará si la instancia del servidor está configurada en un clúster de conmutación por error: 1 – Agrupado 0 – No agrupado



	NULL – Entrada no válida o error
IsFullText Installed	Para determinar si el componente full-text está instalado con la instancia actual de SQL Server: 1: full-text está instalado. 0: full-text no está instalado NULL – Entrada no válida o error
IsIntegrated SecurityOnly	Para determinar si el servidor está en modo de seguridad integrada: 1 – Seguridad integrada 0 – Seguridad no integrada NULL – Entrada no válida o error
IsSingleUser	Para determinar si el servidor es una instalación de un solo usuario: 1 – Usuario único 0 – No es un solo usuario NULL – Entrada no válida o error
IsSync WithBackup	Para determinar si la base de datos es una base de datos publicada o una base de datos de distribución y se puede restaurar sin interrumpir la replicación transaccional actual: 1 – Verdadero 0 – Falso
LicenseType	Qué tipo de licencia está instalada para esta instancia de SQL Server: PER SEAT: modo por puesto PER PROCESSOR: modo por procesador DISABLED: las licencias están deshabilitadas
MachineName	Devuelve el nombre del equipo con Windows NT en el que se ejecuta la instancia del servidor. Para una instancia en clúster (una instancia de SQL Server que se ejecuta en un servidor virtual en Microsoft Cluster Server), devuelve el nombre del servidor virtual.
NumLicenses	Número de licencias de cliente registradas para esta instancia de SQL Server, si está en modo por puesto. Número de procesadores con licencia para esta instancia de SQL Server, si está en modo por procesador.
ProcessID	Id. de proceso del servicio de SQL Server. (El ProcessID es útil para identificar qué sqlservr.exe pertenece a esta instancia).
ProductVersion	Muy parecido a los proyectos de Visual Basic, en el sentido de que se devuelven los detalles de la versión de la instancia de SQL Server, en forma de ' major.minor.build '.
ProductLevel	Devuelve el valor de la versión de la instancia de SQL Server que se está ejecutando actualmente. Devoluciones: 'RTM' : versión de envío 'SPn' : versión de Service Pack 'Bn' : versión Beta
ServerName	Tanto el servidor de Windows NT como la información de la instancia asociada con una instancia específica de SQL Server.



La función **SERVERPROPERTY** es muy útil para corporaciones con múltiples sitios donde los desarrolladores necesitan obtener información de un servidor.

SESSION_USER

La función **SESSION_USER** permite insertar en una tabla un valor proporcionado por el sistema para el nombre de usuario de la sesión actual si no se ha especificado ningún valor predeterminado. La sintaxis es la siguiente:

SESSION_USER

SESSIONPROPERTY

La función **SESSIONPROPERTY** se utiliza para devolver las opciones SET para una sesión. La sintaxis es la siguiente:

SESSIONPROPERTY (<option>)

Esta función es útil cuando hay procedimientos almacenados que alteran las propiedades de la sesión en escenarios específicos. Esta función rara vez debe usarse, ya que no debe modificar demasiadas opciones de **SET** durante el tiempo de ejecución.

STATS_DATE

La función **STATS_DATE** devuelve la fecha en que se actualizaron por última vez las estadísticas del índice especificado. La sintaxis es la siguiente:

STATS_DATE(<table id>, <index id>)

SYSTEM_USER

La función **SYSTEM_USER** permite insertar en una tabla un valor proporcionado por el sistema para el nombre de usuario actual del sistema si no se ha especificado ningún valor predeterminado. La sintaxis es la siguiente:

SYSTEM_USER

USER_NAME

USER_NAME devuelve un nombre de usuario de la base de datos. La sintaxis es la siguiente:

USER_NAME([<id>])

El parámetro **id** especifica el número de identificación del nombre de usuario requerido; si no se da ningún valor, se asume que el usuario actual.



UNIDAD 3

CONSULTAS MULTI-TABLA

En esta unidad nos enfocaremos en las consultas multi-tabla. Abarcaremos Subconsultas, Expresiones de Tabla, Joins y los operadores relacionales UNION, EXCEPT e INTERSECT.

SUBCONSULTAS

SQL soporta escribir consultas dentro de consultas o consultas anidadas. La consulta exterior es una consulta cuyo conjunto de resultados se devuelve al llamador y se conoce como la consulta externa. La consulta interna es una consulta cuyo resultado es utilizado por la consulta externa y se conoce como subconsulta. La consulta interna actúa en lugar de una expresión que se basa en constantes o variables y se evalúa en tiempo de ejecución. A diferencia de los resultados de expresiones que utilizan constantes, el resultado de una subconsulta puede cambiar, debido a cambios en las tablas consultadas. Cuando utilizamos subconsultas, podemos evitar la necesidad de pasos separados en las soluciones que almacenan resultados de consultas intermedias en variables.

Una subconsulta puede ser **autocontenido** o **correlacionada**. Una subconsulta autocontenido no tiene dependencia de tablas de la consulta externa, mientras que una subconsulta correlacionada sí. Una subconsulta puede ser **escalar**, **multivaluada** o de **expresión de tabla**. Es decir, una subconsulta puede devolver un valor único, varios valores o una tabla completa como resultado.

Tanto las subconsultas autocontenidoas como las subconsultas correlacionadas pueden devolver un escalar o valores múltiples. Primero describiremos subconsultas autocontenidoas y veremos ejemplos escalares y multivaluados. Luego describiremos subconsultas correlacionadas.

SUBCONSULTAS AUTOCONTENIDAS

Las subconsultas autocontenidoas son independientes de la consulta principal a la que pertenecen, es decir, pueden ser ejecutadas de manera independiente. Por lo tanto, son simples para probar, ya que pueden probarse por separado

SUBCONSULTAS AUTOCONTENIDAS ESCALARES

Dado que devuelve un único valor y es independiente de la consulta principal, puede aparecer en cualquier lugar de la consulta principal en que se necesite como en el SELECT o WHERE.

Por ejemplo, si necesitamos obtener los datos de la orden con el mayor id (ordenid) podemos ejecutar la siguiente consulta:

```
USE TSQLV6ES;
SELECT ordenid, fechaorden, empid, clienteid
FROM Ventas.Ordenes
WHERE ordenid = (SELECT MAX(0.ordenid)
FROM Ventas.Ordenes AS 0);

ordenid      fechaorden    empid      clienteid
-----  -----  -----
11077        2022-05-06  1          65
```

Para que una subconsulta escalar sea válida debe retornar a lo sumo un valor. Si llega a devolver más de un valor se producirá un error en tiempo de ejecución.



La siguiente consulta se ejecuta sin problema:

```
USE TSQLV6ES;
SELECT ordenid
FROM Ventas.Ordenes
WHERE empid = (SELECT E.empid
                FROM RH.Empleados AS E
                WHERE E.apellido LIKE 'C%');

ordenid
-----
10262
10268
10276
10278
...
11065
11068
11075
```

Devuelve las órdenes de los empleados cuyo apellido comienza con C. Como hay un solo empleado cuyo apellido comienza con C, entonces no da error, pero esta consulta podría devolver más de un valor. Esto pasa si en lugar de pedir que empiecen con C pedimos que empiecen con D:

```
USE TSQLV6ES;
SELECT ordenid
FROM Ventas.Ordenes
WHERE empid = (SELECT E.empid
                FROM RH.Empleados AS E
                WHERE E.apellido LIKE 'D%');

ordenid
-----
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=,
<, <= , >, >= or when the subquery is used as an expression.
```

Si una subconsulta escalar no devuelve ningún valor, devuelve NULL, por lo tanto, el predicado evalúa una comparación con NULL, que da como resultado Desconocido y por lo tanto la consulta principal no devuelve resultados.

Por ejemplo, si buscamos empleado cuyo apellido empieza con A:

```
USE TSQLV6ES;
SELECT ordenid
FROM Ventas.Ordenes
WHERE empid = (SELECT E.empid
                FROM RH.Empleados AS E
                WHERE E.apellido LIKE 'A%');

ordenid
-----
```

SUBCONSULTAS AUTOCONTENIDAS MULTIVALUADAS

Son consultas que devuelven múltiples valores para una misma columna. Los resultados de estas subconsultas deben evaluarse con predicados como IN.

El formato de un predicado utilizando IN es:



<expresión escalar> [NOT] IN (<subconsulta multivaluada>)

El predicado devuelve verdadero si la expresión escalar coincide con alguno de los valores devueltos por la subconsulta.

Si reescribimos uno de los ejemplos anteriores utilizando IN, ya no tendríamos problemas en tiempo de ejecución:

```
USE TSQLV6ES;
SELECT ordenid
FROM Ventas.Ordenes
WHERE empid IN (SELECT E.empid
                 FROM RH.Empleados AS E
                 WHERE E.apellido LIKE 'D%');
```

```
ordenid
-----
10258
10270
10275
10285
...
11017
11022
11058
```

SUBCONSULTAS CORRELACIONADAS

Las subconsultas correlacionadas son subconsultas en las que se hace referencia a atributos que forman parte de la consulta principal. Esto significa que la subconsulta es dependiente de la consulta principal y no puede ser ejecutada de manera independiente. Lógicamente, es como si la subconsulta es evaluada por cada fila de la consulta principal.

Por ejemplo, la siguiente consulta que devuelve las órdenes con el máximo número para cada cliente (clienteid):

```
USE TSQLV6ES;
SELECT clienteid, ordenid, fechaorden, empid
FROM Ventas.Ordenes AS O1
WHERE ordenid =
(SELECT MAX(O2.ordenid)
FROM Ventas.Ordenes AS O2
WHERE O2.clienteid = O1.clienteid);
```

La consulta principal se realiza sobre una instancia de la tabla Ordenes que llamamos O1, que devuelve los valores para los que el campo ordenid coinciden con el resultado de la subconsulta.

La subconsulta filtra los resultados de una segunda instancia de la tabla Ordenes que llamamos O2, en donde el clienteid de la tabla de la subconsulta es igual al clienteid de la tabla principal (O1).

clienteid	ordenid	fechaorden	empid
91	11044	2022-04-23	4
90	11005	2022-04-07	2
89	11066	2022-05-01	7
88	10935	2022-03-09	4
...			
3	10856	2022-01-28	3
2	10926	2022-03-04	4
1	11011	2022-04-09	3



EXISTS

También contamos en T-SQL con el predicado EXISTS, que devuelve verdadero (true) en caso de que la subconsulta devuelva alguna fila, de lo contrario devuelve falso (false).

El formato de un predicado utilizando EXISTS es:

[NOT] EXISTS (<subconsulta multivaluada>)

Por ejemplo, una consulta para obtener los clientes de España (*Spain*) que han realizado órdenes:

```
SELECT clienteid, nombreempresa
FROM Ventas.Clientes AS C
WHERE pais = 'Spain'
AND EXISTS
(SELECT * FROM Ventas.Ordenes AS O
WHERE O.clienteid = C.clienteid);

clienteid    nombreempresa
-----
8            Customer QUHWH
29           Customer MDLWA
30           Customer KSLQF
69           Customer SIUIH
```

SUBCONSULTAS CON MAL COMPORTAMIENTO

Hay tareas comunes que se manejan con subconsultas por las cuales podemos fácilmente tener problemas con errores si no seguimos ciertas prácticas recomendadas. Describiremos dos de estos problemas: uno que implica un error de sustitución en el nombre de una columna de subconsulta y otro que implica un manejo poco intuitivo de NULL en una subconsulta. También proporcionaremos las mejores prácticas que, si se siguen, pueden ayudarnos a evitar meterse en tales problemas.

Considere las siguientes tablas T1 y T2 y los datos de muestra con los que se completan:

```
DROP TABLE IF EXISTS dbo.T1;
DROP TABLE IF EXISTS dbo.T2;
GO
CREATE TABLE dbo.T1(col1 INT NOT NULL);
CREATE TABLE dbo.T2(col2 INT NOT NULL);
INSERT INTO dbo.T1(col1) VALUES(1);
INSERT INTO dbo.T1(col1) VALUES(2);
INSERT INTO dbo.T1(col1) VALUES(3);
INSERT INTO dbo.T2(col2) VALUES(2);
```

Supongamos que necesitamos devolver los valores en T1 que también aparecen en el conjunto de valores en T2. Escribimos la siguiente consulta (pero no la ejecuten todavía).

```
SELECT col1 FROM dbo.T1 WHERE col1 IN (SELECT col1 FROM dbo.T2);
```

Antes de ejecutar esta consulta, examinemos los valores en ambas tablas y respondamos la pregunta, ¿qué valores esperamos ver en el resultado? Naturalmente, esperamos ver solo el valor 2 en el resultado. Ahora ejecutemos la consulta. Obtenemos el siguiente resultado:

```
col1
-----
1
2
3
```



¿Pueden explicar por qué obtenemos todos los valores de T1 y no solo 2?

Un examen detenido de las definiciones de la tabla y el código de consulta revelará que la subconsulta contra T2 se refiere a `col1` por error, en lugar de `col2`. Ahora la pregunta es, ¿por qué no falló el código? La forma en que SQL resuelve a qué tabla pertenece la columna es buscar primero la columna en la tabla en la consulta inmediata. Si no puede encontrarlo allí, como en nuestro caso, lo busca en la tabla de la consulta externa. En nuestro caso, hay una columna llamada `col1` en la tabla exterior; por lo tanto, ese se usa. Sin querer, la referencia interna a `col1` se convirtió en una referencia correlacionada. Entonces, cuando el valor externo de `col1` es 1, la consulta interna selecciona un 1 para cada fila en T2. Cuando el valor de `col1` externo es 2, la consulta interna selecciona un 2 para cada fila en T2. Te das cuenta de que siempre que haya filas en T2 y el valor de `col1` no sea NULL, siempre obtendremos una coincidencia.

Por lo general, este tipo de problema ocurre cuando no somos consistentes al nombrar atributos de la misma manera en diferentes tablas cuando representan lo mismo. Por ejemplo, supongamos que en una tabla Clientes le asignamos un nombre a la columna que contiene el ID de cliente `clienteid`, y en una tabla de Ordenes relacionada le damos el nombre `clienid` a la columna ID de cliente. Luego, escribimos una consulta similar a la consulta de nuestro ejemplo en busca de clientes que realizaron pedidos, pero por error especificamos `clienteid` en ambas tablas. Recuperaremos a todos los clientes en lugar de solo a los que realmente hicieron pedidos. Cuando la declaración externa es `SELECT`, la consulta devuelve un resultado incorrecto con todas las filas, pero prestemos atención porque cuando la declaración externa es `DELETE`, terminaremos eliminando todas las filas.

Hay dos mejores prácticas que pueden ayudarnos a evitar tales errores en el código: una es una recomendación a largo plazo y la otra es una recomendación a corto plazo. La mejor práctica a largo plazo es prestar más atención a nombrar atributos en diferentes tablas de la misma manera cuando representan lo mismo. La recomendación a corto plazo es simplemente anteponer al nombre de la columna el nombre de la tabla (o alias, si asignó uno), y así no permitiremos la resolución implícita. Apliquemos esta práctica a nuestra consulta de muestra:

```
SELECT col1 FROM dbo.T1 WHERE col1 IN(SELECT T2.col1 FROM dbo.T2);
```

Msg 207, Level 16, State 1, Line 12
Invalid column name 'col1'.

Al ver este error, por supuesto, resolveremos el problema y solucionaremos la consulta especificando el nombre de la columna correcta, `col2`, en la subconsulta.

Otro caso clásico en el que las personas se meten en problemas con las subconsultas tiene que ver con las complejidades relacionadas con el tratamiento NULL. Para ver el problema, volvamos a crear y llenar las tablas T1 y T2 con el siguiente código:

```
DROP TABLE IF EXISTS dbo.T1;
DROP TABLE IF EXISTS dbo.T2;
GO
CREATE TABLE dbo.T1(col1 INT NULL);
CREATE TABLE dbo.T2(col1 INT NOT NULL);
INSERT INTO dbo.T1(col1) VALUES(1);
INSERT INTO dbo.T1(col1) VALUES(2);
INSERT INTO dbo.T1(col1) VALUES(NULL);
INSERT INTO dbo.T2(col1) VALUES(2);
INSERT INTO dbo.T2(col1) VALUES(3);
```

Observe los valores en ambas tablas. Supongamos que deseamos devolver solo los valores que aparecen en T2 pero no en T1. Escribimos el siguiente código en un intento de lograrlo (pero no lo ejecuten todavía):

```
SELECT col1 FROM dbo.T2 WHERE col1 NOT IN(SELECT col1 FROM dbo.T1);
```



Antes de ejecutar el código, respondamos la pregunta, ¿qué valores esperamos ver en el resultado? La mayoría de la gente esperaría recuperar el valor 3. Ahora ejecutamos la consulta. Obtenemos un resultado vacío:

```
col1
```

```
-----
```

Una clave para comprender por qué obtenemos un conjunto vacío es recordar que para que un filtro WHERE devuelva una fila, el predicado debe evaluarse como verdadero; obtener falso o desconocido hace que la fila se descarte. Está claro por qué no recupera el valor 2: porque el valor 2 aparece en T1 (el predicado IN devuelve verdadero), no quiere verlo (el NO IN es falso). La parte más complicada es averiguar por qué no recupera el valor 3. Se desconoce la respuesta a la pregunta de si aparece 3 en T1 porque NULL puede representar cualquier valor. Más técnicamente, el predicado IN se traduce en $3 = 1 \text{ O } 3 = 2 \text{ O } 3 = \text{NULL}$, y el resultado de esta disyunción de predicados es el valor lógico desconocido. Ahora aplicamos NO al resultado. Cuando negamos lo desconocido, el resultado sigue siendo desconocido. En otras palabras, al igual que se desconoce que 3 aparece en T1, también se desconoce que 3 no aparece en T1.

Todo esto significa que cuando usamos NOT IN con una subconsulta y al menos uno de los miembros del resultado de la subconsulta es NULL, la consulta devolverá un conjunto vacío. Desde una perspectiva de SQL, no deseamos ver los valores de T2 que aparecen en T1 exactamente, porque sabemos con certeza que aparecen en T1, y no deseamos ver el resto de los valores de T2, porque no sabemos con certeza que no aparecen en T1. Esta es una de las implicaciones absurdas del manejo NULL y la lógica de tres valores.

Podemos hacer ciertas cosas para evitar meternos en tales problemas. Por un lado, si se supone que una columna no permite NULL, asegúrenos de aplicar una restricción NOT NULL. Si no lo hace, lo más probable es que los NULL lleguen a esa columna.

Por otro lado, si se supone que la columna permite NULL, pero, por ejemplo, esos NULL representan un valor faltante e inaplicable, debemos revisar su solución para ignorarlos. Podemos lograr este comportamiento de dos maneras.

Una opción es eliminar explícitamente los NULL en la consulta interna agregando el filtro WHERE col1 NOT IS NULL, así:

```
SELECT col1 FROM dbo.T2 WHERE col1 NOT IN(SELECT col1 FROM dbo.T1 WHERE col1 IS NOT NULL);
```

```
col1
```

```
-----
```

```
3
```

La otra alternativa es utilizar NOT EXISTS en lugar de NOT IN, así:

```
SELECT col1
FROM dbo.T2
WHERE NOT EXISTS(SELECT * FROM dbo.T1 WHERE T1.col1 = T2.col1);
```

Aquí, cuando el filtro de la consulta interna compara un NULL con cualquier cosa, el resultado del predicado del filtro es desconocido y las filas para las que el predicado evalúa como desconocido se descartan. En otras palabras, la eliminación implícita de los NULL por el filtro en esta solución tiene el mismo efecto que la eliminación explícita de los NULL en la solución anterior.



EXPRESIONES DE TABLA

Una expresión de tabla (Table Expression) es consulta nominada que representa una tabla relacional válida. Se pueden utilizar en otras consultas del mismo modo que otras tablas.

Microsoft SQL Server soporta cuatro tipos de expresiones de tabla: Tablas Derivadas, Expresiones Comunes de Tabla, Vistas y Funciones con Valores de tabla en línea.

Las expresiones de tabla no están alojadas físicamente, son virtuales. Cuando se consulta una expresión de tabla, la consulta que la utiliza llama a la consulta de la expresión de tabla, es decir, se van anidando las ejecuciones.

Los beneficios de utilizar expresiones de tabla están asociados generalmente a cuestiones de lógica en el código más que a cuestiones de performance. Por ejemplo, permiten simplificar consultas realizando una solución más modular. También permiten salvar algunas limitaciones del lenguaje, como la imposibilidad de referirse a un alias definido en el SELECT en la cláusula WHERE (ya que el WHERE se evalúa antes que el SELECT).

TABLAS DERIVADAS

Las tablas derivadas (también conocidas como tablas de subconsultas) son definidas en la cláusula FROM de una consulta.

La consulta que genera la tabla derivada se especifica entre paréntesis, seguida por la cláusula AS y el nombre que se asigna a la tabla derivada. Por ejemplo, la siguiente tabla derivada llamada USAClies está basada en una consulta que devuelve los clientes de Estados Unidos (USA), y la consulta que la utiliza devuelve todas las filas de la tabla derivada.

```
USE TSQLV6ES;
SELECT *
FROM (SELECT clienteid, nombreempresa
      FROM Ventas.Clientes
     WHERE pais = N'USA') AS USAClies;
```

clienteid	nombreempresa
32	Customer YSIQX
36	Customer LVJSO
43	Customer UISOJ
45	Customer QXPPT
48	Customer DVFMB
55	Customer KZQZT
65	Customer NYUHS
71	Customer LCOUJ
75	Customer XQJYP
77	Customer LCYBZ
78	Customer NLTYP
82	Customer EYHKM
89	Customer YBQTI

En este caso no tiene mucho sentido utilizar una tabla derivada, ya que la consulta que la usa no realiza ninguna operación sobre la tabla derivada, es simplemente un ejemplo.

Para poder ser utilizada en una tabla derivada una consulta debe cumplir tres requisitos:

- 1) El orden no está garantizado. Una expresión de tabla se supone que representa una tabla relacional, y las filas de una tabla relacional no tienen un orden garantizado. Por este motivo no se puede utilizar ORDER BY en una consulta que define una expresión de tabla, a menos que el ORDER BY tenga otro propósito distinto de sólo la presentación de los datos (por ejemplo, si se utiliza TOP)



- 2) Todas las columnas deben tener nombre. Si alguna columna corresponde a una expresión o función deberá asignarse un alias en el SELECT que define la consulta
- 3) Los nombres de las columnas deben ser únicos. Se debe tener particular cuidado cuando se realizan joins. En caso de que esto ocurra se deberá asignar un alias a alguna de las repeticiones.

Todas estas restricciones son consecuencia de que deben cumplir con las reglas del modelo relacional.

ASIGNANDO ALIAS A COLUMNAS

Desde la consulta que llama a la expresión de tabla se puede hacer referencia a los alias definidos en la expresión de tabla. En consultas normales esto podría dar error, ya que el SELECT se ejecuta a posteriori.

Por ejemplo, supongamos que necesitamos escribir una consulta contra la tabla *Ventas.Ordenes* y devolver la cantidad de clientes (distintos) que se atendieron cada año.

```
USE TSQLV6ES;
SELECT YEAR(fechaorden) AS ordenanio, COUNT(DISTINCT clienteid) AS numclies
FROM Ventas.Ordenes
GROUP BY ordenanio;
```

```
Msg 207, Level 16, State 1, Line 3
Invalid column name 'ordenanio'.
```

Obviamente esto podemos resolverlo reemplazando el alias en el GROUP BY por la función definida en el SELECT, y resultaría simple porque la función es corta, pero ¿qué pasa si la función que debemos aplicar es compleja? Esto nos obliga a mantener dos copias de la función, y en caso de modificaciones o correcciones debemos asegurarnos de corregir en ambas copias.

Para solucionar esto con una sola copia de la función podemos utilizar una expresión de tabla.

```
USE TSQLV6ES;
SELECT ordenanio, COUNT(DISTINCT clienteid) AS numclies
FROM (SELECT YEAR(fechaorden) AS ordenanio, clienteid
      FROM Ventas.Ordenes) AS D
GROUP BY ordenanio;
```

ordenanio	numclies
2020	67
2021	86
2022	81

Existe una segunda manera de asignar alias a la expresión de tabla, y es definirlos luego del alias de la expresión de tabla entre paréntesis. Por ejemplo:

```
USE TSQLV6ES;
SELECT ordenanio, COUNT(DISTINCT clienteid) AS numclies
FROM (SELECT YEAR(fechaorden), clienteid
      FROM Ventas.Ordenes) AS D(ordenanio, clienteid)
GROUP BY ordenanio;
```

ANIDANDO

En caso de ser necesario por la lógica de la consulta, las tablas derivadas se pueden anidar, es decir, puedo utilizar una tabla derivada dentro de la definición de una tabla derivada.

El anidamiento es una técnica que puede ser muy útil en algunos casos, pero tiende a complicar la legibilidad de la consulta.



Por ejemplo, la siguiente consulta devuelve el año y la cantidad de clientes atendidos, pero sólo para aquellos años en los que se han atendido más de 70 clientes:

```
USE TSQLV6ES;
SELECT ordenanio, numclies
FROM (SELECT ordenanio, COUNT(DISTINCT clienteid) AS numclies
      FROM (SELECT YEAR(fechaorden) AS ordenanio, clienteid
            FROM Ventas.Ordenes) D1
      GROUP BY ordenanio) AS D2
WHERE numclies > 70;
```

ordenanio	numclies
2021	86
2022	81

Como dijimos el anidamiento incrementa la complejidad. ¿Qué consulta les parece más simple, la anterior o la que sigue?

```
USE TSQLV6ES;
SELECT YEAR(fechaorden) AS ordenanio, COUNT(DISTINCT clienteid) AS numclies
FROM Ventas.Ordenes
GROUP BY YEAR(fechaorden)
HAVING COUNT(DISTINCT clienteid) > 70;
```

REFERENCIAS MÚLTIPLES

Imaginemos que necesito con fines estadísticos obtener la siguiente información

Año	Clientes Año Actual	Clientes Año Anterior	Crecimiento
2020	67		
2021	86	67	19
2022	81	86	-5

¿Cómo podemos hacer para obtener esta información?

```
USE TSQLV6ES;
SELECT Act.ordenanio AS Año, Act.numclies AS ClientesAñoActual, Prv.numclies AS ClientesAñoAnterior, Act.numclies - Prv.numclies AS Crecimiento
FROM (SELECT YEAR(fechaorden) AS ordenanio, COUNT(DISTINCT clienteid) AS numclies
      FROM Ventas.Ordenes GROUP BY YEAR(fechaorden)) AS Act
LEFT OUTER JOIN
(SELECT YEAR(fechaorden) AS ordenanio, COUNT(DISTINCT clienteid) AS numclies
      FROM Ventas.Ordenes GROUP BY YEAR(fechaorden)) AS Prv
ON Act.ordenanio = Prv.ordenario + 1;
```

Año	ClientesAñoActual	ClientesAñoAnterior	Crecimiento
2020	67	NULL	NULL
2021	86	67	19
2022	81	86	-5

Como podemos ver en la solución, no nos queda otra alternativa que definir la tabla derivada cada vez que necesitamos utilizarla.



EXPRESIONES COMUNES DE TABLA (CTE DE COMMON TABLE EXPRESSIONS)

Son similares a las tablas derivadas, con algunas ventajas. Se definen con WITH con la siguiente sintaxis:

```
WITH <NombreCTE>[(<lista_de_alias_de_columna>)]
AS
(
<consulta que define la CTE>
)
<consulta que utiliza la CTE>;
```

La consulta que define la CTE debe cumplir con los mismos requisitos que las tablas derivadas.

```
USE TSQLV6ES;
WITH UsaClies AS
(
SELECT clienteid, nombreempresa
FROM Ventas.Clientes
WHERE pais = N'USA'
)
SELECT * FROM UsaClies;
```

clienteid	nombreempresa
32	Customer YSIQX
36	Customer LVJSO
43	Customer UISOJ
45	Customer QXPPT
48	Customer DVFMB
55	Customer KZQZT
65	Customer NYUHS
71	Customer LCOUJ
75	Customer X0JYP
77	Customer LCYBZ
78	Customer NLTYP
82	Customer EYHKM
89	Customer YBQTI

ASIGNANDO ALIAS A COLUMNAS EN CTE

Las CTE soportan dos maneras, inline y externa.

INLINE	EXTERNA
<pre>USE TSQLV6ES; WITH C AS (SELECT YEAR(fechaorden) AS ordenanio, clienteid FROM Ventas.Ordenes) SELECT ordenanio, COUNT(DISTINCT clienteid) AS numclies FROM C GROUP BY ordenanio;</pre>	<pre>USE TSQLV6ES; WITH C(ordenanio, clienteid) AS (SELECT YEAR(fechaorden), clienteid FROM Ventas.Ordenes) SELECT ordenanio, COUNT(DISTINCT clienteid) AS numclies FROM C GROUP BY ordenanio;</pre>

DEFINIENDO MÚLTIPLES CTE

A priori la diferencia con las tablas derivadas parece meramente semántica, sin embargo, el hecho de tener que definir primero la CTE antes de utilizarlas les da a las CTE algunas ventajas importantes sobre las tablas derivadas.



Una ventaja es que, si se necesita referenciar dos CTEs, no es necesario anidarlas como a las tablas derivadas. En lugar de eso, se definen múltiples CTEs dentro del WITH separadas por comas. Cada CTE puede hacer referencia a cualquier CTE definida previamente.

```
USE TSQLV6ES;
WITH C1 AS
  (SELECT YEAR(fechaorden) AS ordenanio, clienteid
   FROM Ventas.Ordenes),
  C2 AS
  (SELECT ordenanio, COUNT(DISTINCT clienteid) AS numclies
   FROM C1
   GROUP BY ordenanio)

  SELECT ordenanio, numclies
  FROM C2
  WHERE numclies > 70;

ordenanio      numclies
-----  -----
2021            86
2022            81
```

REFERENCIAS MÚLTIPLES EN CTEs

Aquí viene una diferencia substancial. Como las CTE existen en el momento de ser utilizadas, no tengo que definirlas múltiples veces si quiero utilizarlas múltiples veces, como tenía que hacer con las tablas derivadas.

```
USE TSQLV6ES;
WITH CuentaAnual AS
  (SELECT YEAR(fechaorden) AS ordenanio, COUNT(DISTINCT clienteid) AS numclies
   FROM Ventas.Ordenes
   GROUP BY YEAR(fechaorden))

  SELECT Act.ordenanio, Act.numclies AS actnumclies, Prv.numclies AS prvnumclies,
  Act.numclies - Prv.numclies AS crecimiento
  FROM CuentaAnual AS Act
  LEFT OUTER JOIN CuentaAnual AS Prv
  ON Act.ordenanio = Prv.ordenanio + 1;
```

CTE RECURSIVAS

Las CTE soportan el uso de recursividad. Una CTE recursiva es definida por al menos dos consultas (podrían ser más), donde al menos una es la consulta ancla y la otra es el miembro recursivo.

La forma básica de una CTE recursiva es:

```
WITH <Nombre_CTE>[(<alias de columnas>)]
AS
(
<consulta ancla>
UNION ALL
<consulta recursiva>
)
<consulta que utiliza la CTE>;
```

La consulta ancla es una consulta que devuelve una tabla relacional válida. La consulta ancla se invoca una sola vez.

El miembro recursivo es una consulta que hace referencia a la CTE y es invocada repetidamente hasta que devuelve un conjunto vacío. La referencia a la CTE representa el conjunto resultante anterior. La primera vez que



el miembro recursivo es invocado, el resultado anterior corresponde al resultado de la consulta ancla. Ambas consultas deben ser compatibles en términos de cantidad y tipos de columnas.

VISTAS

Los tipos descriptos anteriormente son expresiones utilizadas para una consulta puntal, y en el momento en que finaliza la ejecución, tanto las tablas derivadas como las CTEs desaparecen, es decir, no son reusables.

En cambio, las vistas sí son reusables. Su definición es almacenada en la base de datos, y permanece en la base hasta que son eliminadas de manera explícita.

En cualquier otro aspecto, son similares a las tablas derivadas y las CTEs.

Para crear una vista con los clientes de Estados Unidos (ClientesUSA) realizamos lo siguiente:

```
USE TSQLV6ES;
DROP VIEW IF EXISTS Ventas.ClientesUSA;
GO
CREATE VIEW Ventas.ClientesUSA
AS
SELECT
clienteid, nombreempresa, nombrecontacto, cargocontacto, direccion,
ciudad, region, codigopostal, pais, telefono, fax
FROM Ventas.Clientes
WHERE pais = N'USA';
GO
```

En primer término, verificamos si la vista existe. En caso de que exista, primero borramos la definición mediante DROP.

Los alias de columnas en este ejemplo se definen de modo inline, pero también pueden definirse de modo externo declarándolos entre paréntesis a continuación del nombre de la vista.

Una vez creada la vista, puede ser consultada como cualquier otra tabla.

```
USE TSQLV6ES;
SELECT clienteid, nombreempresa
FROM Ventas.ClientesUSA;
```

clienteid	nombreempresa
32	Customer YSIQX
36	Customer LVJSO
43	Customer UISOJ
45	Customer QXPPT
48	Customer DVFMB
55	Customer KZQZT
65	Customer NYUHS
71	Customer LCOUJ
75	Customer X0JYP
77	Customer LCYBZ
78	Customer NLTYP
82	Customer EYHKM
89	Customer YBQTI



FUNCIONES CON VALORES DE TABLA EN LÍNEA

Son expresiones de tabla reutilizables como las vistas, pero que soportan parámetros de entrada. Podemos pensarlas en definitiva como vistas parametrizables.

Veamos el siguiente ejemplo:

```
USE TSQLV6ES;
DROP FUNCTION IF EXISTS dbo.GetClieOrdenes;
GO
CREATE FUNCTION dbo.GetClieOrdenes
(@cid AS INT) RETURNS TABLE
AS
RETURN
SELECT ordenid, clienteid, empid, fechaorden, fecharequerida,
fechadespacho, transporteid, flete, nombreenvio, direccionenvio, ciudadenvio,
regionenvio, codigopostalenvio, paisenvio
FROM Ventas.Ordenes
WHERE clienteid = @cid;
GO
```

Esta expresión admite un parámetro de entrada llamado @cid, que representa el ID de cliente, y devuelve las órdenes de este cliente.

Entonces para obtener las órdenes del cliente 1, podemos hacerlo de la siguiente manera:

```
USE TSQLV6ES;
SELECT ordenid, clienteid
FROM dbo.GetClieOrdenes(1) AS O;
```

ordenid	clienteid
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1

Es una buena práctica asignar un alias. Estas expresiones pueden ser utilizadas dentro de una estructura de JOIN, como podemos ver en el siguiente ejemplo:

```
SELECT O.ordenid, O.clienteid, OD.productoid, OD.cantidad
FROM dbo.GetClieOrdenes(1) AS O
JOIN Ventas.DetallesOrden AS OD
ON O.ordenid = OD.ordenid;
```

ordenid	clienteid	productoid	cantidad
10643	1	28	15
10643	1	39	21
10643	1	46	2
10692	1	63	20
10702	1	3	6
10702	1	76	15
10835	1	59	15
10835	1	77	2
10952	1	6	16
10952	1	28	2
11011	1	58	40
11011	1	71	20



EJEMPLOS

```
/* Obtener los id de clientes que fueron atendidos por todos los empleados */
USE TSQLV6ES;
SELECT clienteid
FROM Ventas.Ordenes
GROUP BY clienteid
HAVING COUNT(DISTINCT empid) = (SELECT COUNT(*) FROM RH.Empleados);

clienteid
-----
71
```

Considerando la siguiente consulta en la que obtenemos las fechas de la última orden de cada mes:

```
/* Obtener para cada año y mes, cual fue la fecha de la ultima orden del periodo */
USE TSQLV6ES;
SELECT MAX(fechaorden) AS ultimafecha
FROM Ventas.Ordenes
GROUP BY YEAR(fechaorden), MONTH(fechaorden);

ultimafecha
-----
2020-11-29
2022-02-27
2021-07-31
...

/* Obtener las ultimas ordenes emitidas de cada mes */
USE TSQLV6ES;
SELECT ordenid, fechaorden, clienteid, empid
FROM Ventas.Ordenes
WHERE fechaorden IN
(SELECT MAX(fechaorden)
FROM Ventas.Ordenes
GROUP BY YEAR(fechaorden), MONTH(fechaorden));

ordenid    fechaorden clienteid   empid
-----  -----
10269      2020-07-31 89          5
10294      2020-08-30 65          4
10317      2020-09-30 48          6
10343      2020-10-31 44          4
...
11075      2022-05-06 68          8
11076      2022-05-06 9           4
11077      2022-05-06 65          1
```



JOINS

JOIN es, por lejos, el tipo de operador de tabla más utilizado y una de las características más utilizadas en SQL en general. La mayoría de las consultas implican la combinación de datos de varias tablas y una de las principales herramientas que se utilizan para este propósito es el JOIN.

Esta unidad cubre los tipos de JOIN fundamentales (CROSS, INNER y OUTER), joins auto, equi y non-equi. También cubre aspectos lógicos y de optimización de consultas de múltiples JOINs, semi joins y anti semi joins, así como algoritmos de join. La unidad concluye con la cobertura de una tarea común llamada elementos de separación y demuestra una solución eficiente para la tarea basada en joins.

CROSS JOIN

Entre los tres tipos de join fundamentales (cross, inner y outer), el primero es el más simple, aunque se usa con menos frecuencia que los demás. Un cross join produce un producto cartesiano de las dos tablas de entrada. Si una tabla contiene M filas y la otra N filas, obtiene un resultado con $M \times N$ filas.

En términos de sintaxis, SQL estándar (así como T-SQL) admite dos sintaxis para combinaciones cruzadas. Una es una sintaxis anterior en la que especifica una coma entre los nombres de las tablas, así:

```
USE TSQLV6ES;
SELECT E1.nombre AS nombre1, E1.apellido AS apellido1,
E2.nombre AS nombre2, E2.apellido AS apellido2
FROM RH.Empleados AS E1, RH.Empleados AS E2;
```

Otra es una sintaxis más nueva que se agregó en el estándar SQL-92, en la que especifica una palabra clave que indica el tipo de unión (CROSS, en nuestro caso) seguida de la palabra clave JOIN entre los nombres de las tablas, así:

```
USE TSQLV6ES;
SELECT E1.nombre AS nombre1, E1.apellido AS apellido1,
E2.nombre AS nombre2, E2.apellido AS apellido2
FROM RH.Empleados AS E1 CROSS JOIN RH.Empleados AS E2;
```

Ambas sintaxis son estándar y ambas son compatibles con T-SQL. No hay una diferencia lógica ni una diferencia de optimización entre los dos. Quizás se pregunte, entonces, ¿por qué SQL estándar se molestó en crear dos sintaxis diferentes si tienen el mismo significado? La razón de esto no tiene nada que ver con los cross e inner joins; más bien, está relacionado con los outer joins. Un outer join implica un predicado coincidente, que tiene un papel muy diferente al de un predicado de filtrado. El primero define solo qué filas del lado no preservado de la combinación coinciden con las filas del lado preservado, pero no puede filtrar las filas del lado preservado. Este último se utiliza como filtro básico y ciertamente puede descartar filas de ambos lados.

El comité de estándares SQL decidió agregar soporte para outer joins en el estándar SQL-92, pero pensaron que la sintaxis tradicional basada en comas no se presta para admitir el nuevo tipo de join. Esto se debe a que esta sintaxis no le permite definir un predicado coincidente como parte del operador de tabla de join, que, como se mencionó, se supone que juega un papel diferente al predicado de filtrado que especifica en la cláusula WHERE. Así que terminaron creando la sintaxis más nueva con la palabra clave JOIN con una cláusula ON designada donde uno indica el predicado coincidente. Esta sintaxis más nueva con la palabra clave JOIN se conoce comúnmente como sintaxis SQL-92, y la sintaxis antigua basada en comas se conoce como sintaxis SQL-89.

Para permitir un estilo de codificación consistente para los diferentes tipos de combinaciones, el estándar SQL-92 también agregó soporte para una sintaxis similar basada en palabras clave JOIN para cross e inner joins. Esa es la historia que explica por qué ahora tiene dos sintaxis estándar para cross e inner joins y solo una sintaxis estándar para outer joins. Se recomienda encarecidamente que ceñirse a utilizar la sintaxis basada en palabras clave JOIN en todos los ámbitos. Una razón es que da como resultado un estilo de codificación consistente que es mucho



más fácil de mantener que un estilo mixto. Además, la sintaxis basada en comas es más propensa a errores, como veremos más adelante cuando analicemos los inner joins.

Veremos un par de ejemplos de casos de uso de cross join. Uno genera datos de muestra y el otro evita un problema de optimización relacionado con las subconsultas.

Supongamos que necesitamos generar datos de muestra que representen pedidos. Un pedido tiene un ID de pedido, se realizó en una fecha determinada, lo realizó un cliente determinado y lo manejó un empleado determinado. Obtenemos como entradas el rango de fechas de pedido, así como la cantidad de clientes y empleados distintos que necesitamos brindar soporte. Con la función GetNums en la base de datos de muestra, podemos generar conjuntos de fechas, ID de cliente e ID de empleado. Realizaremos cross join entre esos conjuntos para obtener todas las combinaciones posibles y, utilizando una función ROW_NUMBER, calcularemos los ID de la orden, así:

```
USE TSQLV6ES;
DECLARE @s AS DATE = '20210101', @e AS DATE = '20210131',
@numcusts AS INT = 50, @numemps AS INT = 10;
SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS ordenid,
DATEADD(day, D.n, @s) AS fechaorden, C.n AS clienteid, E.n AS empid
FROM dbo.GetNums(0, DATEDIFF(day, @s, @e)) AS D
CROSS JOIN dbo.GetNums(1, @numcusts) AS C
CROSS JOIN dbo.GetNums(1, @numemps) AS E;
```

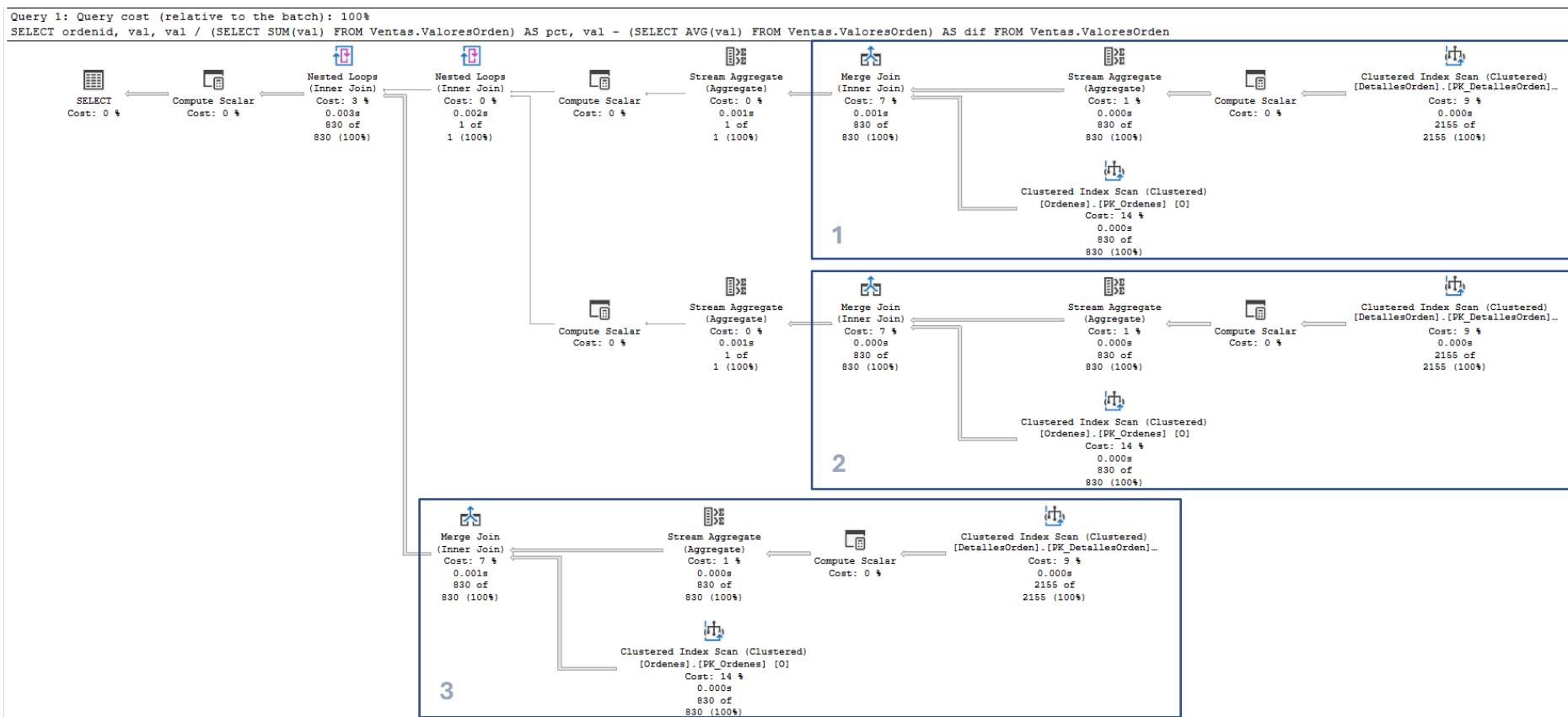
La especificación de orden inusual ORDER BY (SELECT NULL) en la función ROW_NUMBER es un truco que se utiliza para calcular los números de fila en función de un orden arbitrario. Utilizamos este truco cuando necesitamos generar valores únicos y no nos importa el orden en que se generan.

Tengamos en cuenta que, por lo general, cuando generamos datos de muestra, necesitamos una solución más sofisticada que dé como resultado una distribución más realista de los datos. En realidad, normalmente no tenemos una distribución uniforme de las fechas de los pedidos, los ID de clientes y los ID de empleados. Pero a veces necesitamos una forma muy básica de datos de muestra para ciertas pruebas, y esta sencilla técnica puede ser suficiente.

Otro caso de uso de las combinaciones cruzadas tiene que ver con una cierta deficiencia de optimización en SQL Server relacionada con las subconsultas. Consideremos la siguiente consulta:

```
USE TSQLV6ES;
SELECT ordenid, val,
val / (SELECT SUM(val) FROM Ventas.ValoresOrden) AS pct,
val - (SELECT AVG(val) FROM Ventas.ValoresOrden) AS dif
FROM Ventas.ValoresOrden;
```

Esta consulta se realiza en la vista ValoresOrden. Devuelve para cada pedido el porcentaje del valor del pedido actual con respecto al total general, así como la diferencia con respecto al promedio general. El total general y el promedio general se calculan mediante subconsultas agregadas escalares. Es cierto que la consulta se refiere a la vista ValoresOrden tres veces, pero es evidente que ambas subconsultas deben operar en exactamente el mismo conjunto de valores. Desafortunadamente, actualmente no existe una lógica en el optimizador para evitar la duplicación de trabajo al contraer todas las subconsultas coincidentes en una actividad que alimentará el mismo conjunto de valores a todos los cálculos agregados. En nuestro caso, el problema se amplía aún más porque ValoresOrden es una vista. La vista une las tablas Ventas.Ordenes y Ventas.DetallesOrden, y agrupa y agrega los datos para producir información a nivel de pedido. Todo este trabajo se repite tres veces en el plan de ejecución de nuestra consulta, como puede ver en la siguiente imagen:



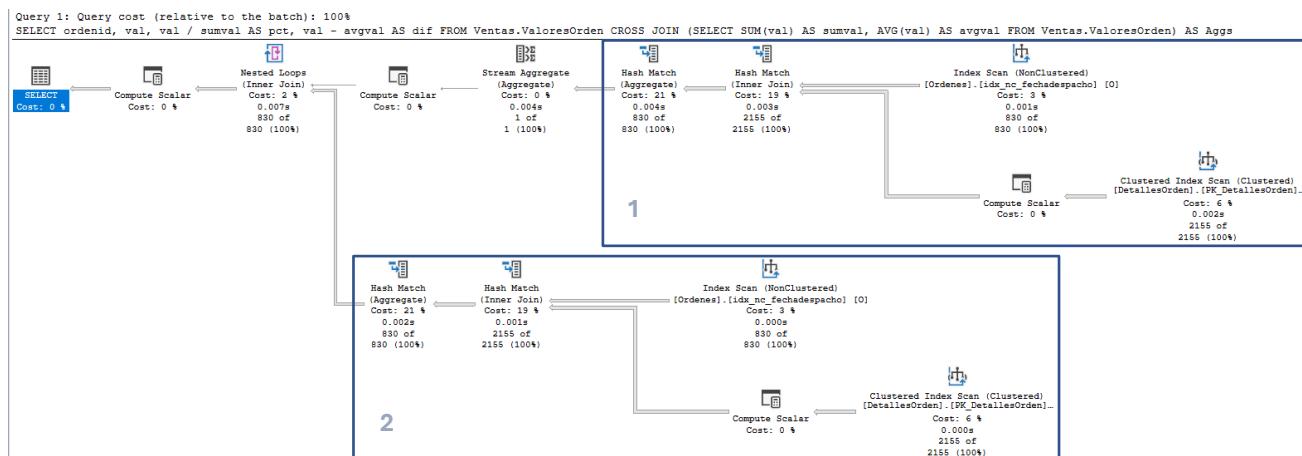


Las ramas marcadas como 1 y 2 representan el trabajo de las dos subconsultas que alimentan los valores a los dos cálculos agregados. La rama marcada como 3 representa la referencia a la vista en la consulta externa para obtener la información del pedido, en otras palabras, el detalle. Como podemos imaginar, cuantos más agregados tenga, peor será el rendimiento de esta solución. Imaginemos que tenemos 10 agregados diferentes para calcular. Esto requeriría 11 repeticiones del trabajo: 10 para los agregados y una para los detalles.

Afortunadamente, existe una forma de evitar la repetición innecesaria del trabajo. Para lograr esto, usaremos una consulta para calcular todos los agregados que necesita. SQL Server optimizará esta consulta aplicando el trabajo solo una vez, alimentando los mismos valores a todos los cálculos agregados. Definimos una expresión de tabla basada en esa consulta (que llamaremos Aggs) y realizaremos una combinación cruzada entre la vista ValoresOrden y Aggs. Luego, en la lista SELECT, realizaremos todos sus cálculos relacionados con el detalle y los agregados. Aquí está la consulta de la solución completa:

```
USE TSQLV6ES;
SELECT ordenid, val,
val / sumval AS pct,
val - avgval AS dif
FROM Ventas.ValoresOrden
CROSS JOIN (SELECT SUM(val) AS sumval, AVG(val) AS avgval
FROM Ventas.ValoresOrden) AS Aggs;
```

Observemos el plan de ejecución para esta consulta.



Observemos que esta vez el trabajo representado por la vista ocurre solo una vez para todos los agregados (rama 1) y una vez para el detalle (rama 2). Lo bueno de esta solución es que no importa cuántos agregados tenga, seguirá teniendo una sola actividad que recopile los valores de todos ellos.

Nos enfrentaremos a un problema similar cuando necesitemos dividir el cálculo. Por ejemplo, supongamos que en lugar de totales generales, necesitamos totales de clientes. En otras palabras, debemos calcular el porcentaje del valor actual del pedido con respecto al total del cliente (no el total general) y la diferencia con el promedio del cliente. Usando el enfoque de subconsulta, agregamos correlaciones, así:

```
USE TSQLV6ES;
SELECT ordenid, val,
val / (SELECT SUM(val) FROM Ventas.ValoresOrden AS I
WHERE I.clienteid = O.clienteid) AS pct,
val - (SELECT AVG(val) FROM Ventas.ValoresOrden AS I
WHERE I.clienteid = O.clienteid) AS dif
FROM Ventas.ValoresOrden AS O;
```



Pero nuevamente, debido a que estamos utilizando subconsultas, el trabajo representado por la vista se repetirá tres veces. La solución es similar a la anterior, solo que en lugar de usar un cross join, usaremos un inner join y hacemos coincidir las ID de cliente entre la vista ValoresOrden y Aggs, así:

```
USE TSQLV6ES;
SELECT ordenid, val,
val / sumval AS pct,
val - avgval AS dif
FROM Ventas.ValoresOrden AS O
INNER JOIN (SELECT clienteid, SUM(val) AS sumval, AVG(val) AS avgval
FROM Ventas.ValoresOrden
GROUP BY clienteid) AS Aggs
ON O.clienteid = Aggs.clienteid;
```

INNER JOIN

Desde una perspectiva de procesamiento lógico de consultas, un inner join implica dos pasos. Comienza con un producto cartesiano entre las dos tablas de entrada, como en un cross join. Luego aplica un filtro que generalmente involucra elementos coincidentes de ambos lados.

Desde una perspectiva de procesamiento físico de consultas, por supuesto, las cosas se pueden hacer de manera diferente, siempre que el resultado sea correcto.

Al igual que con los cross joins, los inner joins actualmente tienen dos sintaxis estándar. Una es la sintaxis SQL-89, donde especifica comas entre los nombres de las tablas y luego aplica todos los predicados de filtrado en la cláusula WHERE, así:

```
USE TSQLV6ES;
SELECT C.clienteid, C.nombreempresa, O.ordenid
FROM Ventas.Clientes AS C, Ventas.Ordenes AS O
WHERE C.clienteid = O.clienteid
AND C.pais = 'USA';
```

Como podemos ver, la sintaxis no distingue realmente entre un cross join y un inner join; más bien, expresa un inner join como un cross join con una condición de filtro.

La otra sintaxis es SQL-92, que es más compatible con la sintaxis de cross join estándar. Especificamos las palabras clave INNER JOIN entre los nombres de las tablas (o simplemente JOIN porque INNER es el predeterminado) y el predicado de join en la cláusula obligatoria ON. Si tenemos condiciones de filtro adicionales, podemos especificarlas en la cláusula ON del join o en la cláusula WHERE habitual, así:

```
USE TSQLV6ES;
SELECT C.clienteid, C.nombreempresa, O.ordenid
FROM Ventas.Clientes AS C
INNER JOIN Ventas.Ordenes AS O
ON C.clienteid = O.clienteid
WHERE C.pais = 'USA';
```

A pesar de que la cláusula ON es obligatoria en la sintaxis SQL-92 para inner joins, no hay distinción entre un predicado coincidente y un predicado de filtrado como en los outer joins. Ambos predicados se tratan como predicados de filtrado. Entonces, aunque desde una perspectiva de procesamiento lógico de consultas, se supone que la cláusula WHERE se evalúa después de la cláusula FROM con todos sus joins, el optimizador podría muy bien decidir procesar los predicados de la cláusula WHERE antes de comenzar a procesar los joins en la cláusula FROM. De hecho, si examinamos los planes para las dos consultas que se acaban de mostrar, notaremos que en ambos casos el plan comienza con una exploración del índice agrupado en la tabla Clientes y aplica el filtro de país como parte de la exploración.



Mencionamos anteriormente que se recomienda ceñirse a la sintaxis SQL-92 para los joins y evitar la sintaxis SQL-89 aunque es estándar para los cross e inner joins. Ya proporcionamos una razón para esta recomendación: usar un estilo consistente para todo tipo de joins. Otra razón es que la sintaxis SQL-89 es más propensa a errores; específicamente, es más probable que se olvide un predicado de join y que tal error pase desapercibido. Con la sintaxis SQL-92, generalmente especificamos cada predicado de join justo después del respectivo join, así:

```
FROM T1
INNER JOIN T2
ON T2.col1 = T1.col1
INNER JOIN T3
ON T3.col2 = T2.col2
INNER JOIN T4
ON T4.col3 = T3.col3
```

Con esta sintaxis, la probabilidad de que pierda un predicado de join es baja, e incluso si lo hace, el analizador generará un error porque la cláusula ON es obligatoria en los inner joins (y outer). Por el contrario, con la sintaxis SQL-89, especificamos todos los nombres de tabla separados por comas en la cláusula FROM y todos los predicados como una conjunción en la cláusula WHERE, así:

```
FROM T1, T2, T3, T4
WHERE T2.col1 = T1.col1 AND T3.col2 = T2.col2 AND T4.col3 = T3.col3
```

Claramente, la probabilidad de que olvidemos un predicado por error es mayor, y si lo hacemos, terminaremos con un cross join involuntario.

Una pregunta común es si existe una diferencia entre usar las palabras clave INNER JOIN y simplemente JOIN. No hay ninguna. SQL estándar hizo que los inner join sean los predeterminados porque probablemente son el tipo de join más utilizado. De manera similar, SQL estándar hizo que la palabra clave OUTER fuera opcional en los outer joins. Por ejemplo, LEFT OUTER JOIN y LEFT JOIN son equivalentes. En cuanto a lo que se recomienda usar, algunos prefieren la sintaxis completa y la encuentran más clara. Otros prefieren la sintaxis breve porque da como resultado un código más corto. Creo que lo que es más importante que utilizar la sintaxis completa o breve es ser coherente entre los miembros del equipo de desarrollo. Cuando diferentes personas usan diferentes estilos, puede resultar difícil mantener el código de los demás. Una buena idea es tener reuniones para discutir opciones de estilo, como sintaxis completa o breve, sangría, mayúsculas, etc., y elaborar un documento de estilo de codificación que todos deban seguir. A veces, las personas tienen opiniones sólidas sobre ciertos aspectos de estilo en función de sus preferencias personales, por lo que cuando surgen conflictos, se puede votar para determinar qué opción prevalecerá. Una vez que todos comienzan a usar un estilo coherente, es mucho más fácil para diferentes personas mantener el mismo código.

OUTER JOIN

Desde una perspectiva de procesamiento lógico de consultas, los outer joins comienzan con los mismos dos pasos que los inner joins. Pero, además, tienen un tercer paso lógico que garantiza que se devuelvan todas las filas de la tabla o tablas que marque como conservadas, incluso si no tienen coincidencias en la otra tabla según el predicado de join. Con las palabras clave LEFT (o LEFT OUTER), RIGHT o FULL, marca la tabla de la izquierda, la tabla de la derecha o ambas tablas como conservadas. Las filas de salida que representan no coincidencias (también conocidas como filas externas) tienen NULL utilizados como marcadores de posición en las columnas del lado no preservado.

Como ejemplo, la siguiente consulta devuelve clientes y sus pedidos, incluidos los clientes sin pedidos:



```
USE TSQLV6ES;
SELECT C.clienteid, C.nombreempresa, C.pais,
O.ordenid, O.paisenvio
FROM Ventas.Clientes AS C
LEFT OUTER JOIN Ventas.Ordenes AS O
ON C.clienteid = O.clienteid;
```

En los outer joins, el papel del predicado en la cláusula ON es diferente que en los inner joins. Con el último, sirve como una herramienta de filtrado simple, mientras que con el primero sirve como una herramienta de comparación más sofisticada. Todas las filas del lado preservado se devolverán tanto si encuentran filas coincidentes según el predicado como si no. Si necesitamos que se apliquen filtros simples, los especificamos en la cláusula WHERE. Por ejemplo, una forma clásica de identificar solo las no coincidencias es usar un outer join y agregar una cláusula WHERE que filtra solo las filas que tienen un NULL en una columna del lado no preservado que no permite NULL normalmente. La columna de clave principal es una buena opción para este propósito. Por ejemplo, la siguiente consulta solo devuelve clientes que no realizaron pedidos:

```
USE TSQLV6ES;
SELECT C.clienteid, C.nombreempresa, O.ordenid
FROM Ventas.Clientes AS C
LEFT OUTER JOIN Ventas.Ordenes AS O
ON C.clienteid = O.clienteid
WHERE O.ordenid IS NULL;
```

Es mucho más común que las personas utilicen left outer joins que right outer joins. Podría deberse a que la mayoría de las personas tienden a especificar tablas referenciadas antes de hacer referencia a las de la consulta y, por lo general, la tabla referenciada es la que puede tener filas sin coincidencias en la referencia. Es cierto que en un caso de combinación única, como T1 LEFT OUTER JOIN T2, puede expresar lo mismo con un T2 RIGHT OUTER JOIN T1 simétrico. Sin embargo, en ciertos casos más complejos con múltiples joins involucrados, encontraremos que usar un right outer join permite una solución más simple que usar left outer joins. Veremos un caso de este tipo en breve en la sección "Consultas Multi-Join".

SELF JOIN

Un self join (auto join) es un join entre varias instancias de la misma tabla. Como ejemplo, la siguiente consulta relaciona a los empleados con sus gerentes, que también son empleados:

```
USE TSQLV6ES;
SELECT E.nombre + ' ' + E.apellido AS emp, J.nombre + ' ' + J.apellido AS jefe
FROM RH.Empleados AS E
LEFT OUTER JOIN RH.Empleados AS J
ON E.jefeid = J.empid;
```

Lo que tiene de especial los self join es que es obligatorio asignar un alias a las instancias de la tabla de forma diferente; de lo contrario, terminaremos con nombres de columna duplicados, incluidos los prefijos de la tabla.

Desde una perspectiva de optimización, el optimizador considera las dos instancias como dos tablas diferentes. Por ejemplo, puede usar diferentes índices para cada una de las instancias. En el plan de consulta, puede identificar en cada objeto físico al que se accede qué instancia representa en función del alias de tabla que le asignamos.

EQUI Y NONEQUI JOINS

Los términos equi y nonequi joins se refieren al tipo de operador que utiliza en el predicado de combinación. Cuando utilizamos un operador de igualdad, como en la gran mayoría de las consultas, la combinación se denomina equi join. Cuando utiliza un operador que no sea la igualdad, el join se denomina nonequi join.



Como ejemplo, la siguiente consulta usa un nonequi join con un operador menor que (<) para identificar pares únicos de empleados:

```
USE TSQLV6ES;
SELECT E1.empid, E1.nombre, E1.apellido, E2.empid, E2.apellido, E2.nombre
FROM RH.Empleados AS E1
INNER JOIN RH.Empleados AS E2
ON E1.empid < E2.empid;
```

Si te preguntas por qué deberíamos usar un nonequi join aquí en lugar de un cross join, es porque no deseamos obtener los “auto” pares (el mismo empleado x, x en ambos lados) y no deseamos obtener pares “espejo” (x, y así como y, x). Si deseamos producir triples únicos, simplemente agregamos otro nonequi join a una tercera instancia de la tabla (llamémosla E3), con el predicado de combinación E2.empid < E3.empid.

Desde una perspectiva de optimización, es importante identificar el join como equi o nonequi. Más adelante, veremos la optimización de joins con los algoritmos de join, bucle anidado, combinación y hash. Bucle anidado es el único algoritmo compatible con los nonequi joins. La combinación y el hash requieren un equi join o, más precisamente, al menos un predicado que se basa en un operador de igualdad.

CONSULTAS MULTI-JOIN

Las consultas multi-join son consultas con múltiples joins (valga la redundancia). Hay una serie de consideraciones interesantes sobre estas consultas, algunas relacionadas con la optimización y otras relacionadas con el tratamiento lógico.

Como ejemplo, la siguiente consulta une cinco tablas para devolver pares de cliente-proveedor que tuvieron actividad juntos:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
INNER JOIN Ventas.Ordenes AS O ON O.clienteid = C.clienteid
INNER JOIN Ventas.DetallesOrden AS DO ON DO.ordenid = O.ordenid
INNER JOIN Produccion.Productos AS P ON P.productoid = DO.productoid
INNER JOIN Produccion.Proveedores AS PV ON PV.proveedorid = P.proveedorid;
```

El nombre de la empresa del cliente se obtiene de la tabla Clientes. La tabla Clientes se une a la tabla Ordenes para encontrar las cabeceras de las órdenes que realizaron los clientes. El resultado se une con DetallesOrden para encontrar las líneas de pedido dentro de esas órdenes. Las líneas de pedido contienen los ID de los productos que se pidieron. El resultado se une a Productos, donde se encuentran los ID de los proveedores de esos productos. Finalmente, el resultado se une con Proveedores para obtener el nombre de la empresa proveedora.

Dado que el mismo par cliente-proveedor puede aparecer más de una vez en el resultado, la consulta tiene una cláusula DISTINCT para eliminar los duplicados. Aquí había una buena razón para usar DISTINCT. Sin embargo, tengan en cuenta que si observan un uso excesivo de DISTINCT en las consultas de join de alguien, podría indicar que no comprenden correctamente las relaciones en el modelo de datos. Al combinar tablas basándose en relaciones incorrectas, podemos terminar con duplicados, y el uso de DISTINCT en tal caso podría ser un intento de ocultarlos.

Las siguientes secciones tratan aspectos del procesamiento físico y lógico de esta consulta.

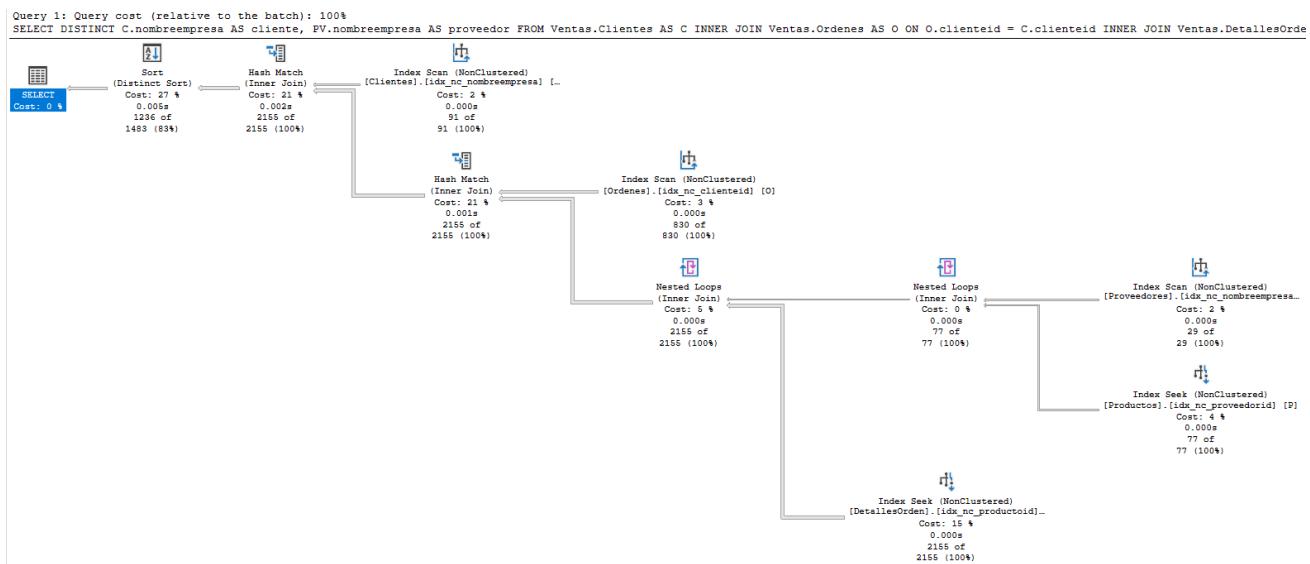
CONTROLAR EL ORDEN DE EVALUACIÓN FÍSICO DEL JOIN

Recordemos que los operadores de tabla se evalúan de izquierda a derecha.



Ese es el caso desde el punto de vista del procesamiento lógico. Sin embargo, desde el punto de vista del procesamiento físico, bajo ciertas condiciones, el orden puede modificarse sin cambiar el significado de la consulta. Con los outer joins, cambiar el orden puede cambiar el significado, y esto es algo que el optimizador no puede hacer. Pero con inner y cross joins, cambiar el orden no cambia el significado. Entonces, con ellos, el optimizador aplicará la optimización del orden de join. Explorará diferentes órdenes de join y seleccionará la que, según sus estimaciones, se supone que es la más rápida.

Como ejemplo, nuestra consulta de multi-join obtuvo el plan de la siguiente imagen, que muestra que el orden de join físico es diferente al lógico.



Observemos que el orden en el plan es el siguiente: Clientes join (Ordenes join ((Proveedores join Productos) join DetallesOrden)). A modo de ilustración, la siguiente consulta aplica un orden lógico de joins que refleja el orden físico de la imagen:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
INNER JOIN ( Ventas.Ordenes AS O
INNER JOIN ( Produccion.Proveedores AS PV
INNER JOIN Produccion.Productos AS P
ON P.proveedorid = PV.proveedorid
INNER JOIN Ventas.DetallesOrden AS DO
ON DO.productoid = P.productoid )
ON DO.orderid = O.orderid )
ON O.clienteid = C.clienteid;
```

Hay una serie de razones que podrían llevar al optimizador a elegir un orden de join subóptimo. Al igual que con otras opciones subóptimas, las opciones de orden de join pueden verse afectadas por estimaciones de cardinalidad inexactas.

Además, el número teórico de posibles órdenes de join, o árboles de join, puede ser muy grande incluso con una pequeña cantidad de tablas.

Con N tablas, el número de árboles posibles es $(2N - 2)! / (N - 1)!$, donde “!” indica factorial. Como ejemplo, en nuestro caso tenemos 5 tablas que dan como resultado 1.680 árboles posibles. Con 10 tablas, ¡el número es 17,643,225,600! Aquí el “!” es para asombro, no para representar factorial. Como podemos darnos cuenta, si el optimizador realmente intentara explorar todos los árboles de join posibles, el proceso de optimización tomaría demasiado tiempo y se volvería contraproducente. Entonces, el optimizador hace una serie de cosas para reducir



el tiempo de optimización. Calcula los umbrales basados en el costo y el tiempo en función de los tamaños de las tablas involucradas y, si se alcanza uno de ellos, la optimización se detiene. El optimizador tampoco considera normalmente los árboles de join tupidos, donde se unen los resultados de los joins; tales árboles representan la mayoría de los árboles. Lo que todo esto significa es que es posible que el optimizador no proporcione el orden de join verdaderamente óptimo.

Si sospechamos que ese es el caso y deseamos forzar al optimizador a usar un orden determinado, podemos hacerlo escribiendo los joins en el orden que creamos que es óptimo y agregando la sugerencia de consulta (query hint) FORCE ORDER, así:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
INNER JOIN Ventas.Ordenes AS O
ON O.clienteid = C.clienteid
INNER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid
INNER JOIN Produccion.Productos AS P
ON P.productoid = DO.productoid
INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid
OPTION (FORCE ORDER);
```

Esta sugerencia (hint) es ciertamente buena como herramienta de resolución de problemas porque podemos usarla para probar el rendimiento de diferentes órdenes y ver si el optimizador, de hecho, tomó una decisión subóptima. El problema de usar tal sugerencia en el código de producción es que hace que esta parte de la optimización sea estática. Normalmente, los cambios en la indexación, las características de los datos y otros factores pueden provocar un cambio en la elección del orden de join del optimizador, pero obviamente no cuando se usa la sugerencia. Por lo tanto, la sugerencia es buena para usar como herramienta de resolución de problemas y, a veces, como una solución breve y temporal en producción en casos críticos.

Sin embargo, generalmente no es bueno usarla como una solución a largo plazo. Por ejemplo, supongamos que la causa de la elección de orden de join subóptima fue una estimación de cardinalidad incorrecta. Cuando tengamos tiempo para hacer una investigación más exhaustiva del problema, espero que podamos encontrar formas de ayudar al optimizador a generar estimaciones más precisas que, naturalmente, conduzcan a un plan más óptimo.

CONTROLAR EL ORDEN DE EVALUACIÓN LÓGICA DEL JOIN

Acabamos de ver cómo controlar el orden físico de los join. También hay casos en los que es necesario controlar el orden lógico de los joins para lograr un cierto cambio en el significado de la consulta. Como ejemplo, consideremos la consulta que devuelve pares cliente-proveedor de la sección anterior:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
INNER JOIN Ventas.Ordenes AS O
ON O.clienteid = C.clienteid
INNER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid
INNER JOIN Produccion.Productos AS P
ON P.productoid = DO.productoid
INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid;
```

Esta consulta devuelve 1236 filas. Todos los joins de la consulta son inner. Esto significa que los clientes que no realizaron ningún pedido no serán devueltos. Hay dos clientes de este tipo en nuestros datos de muestra.



Supongamos que necesitamos cambiar la solución para incluir clientes sin pedidos. Lo intuitivo es cambiar el tipo de join entre Clientes y Pedidos a un left outer join, así:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
LEFT OUTER JOIN Ventas.Ordenes AS O
ON O.clienteid = C.clienteid
INNER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid
INNER JOIN Produccion.Productos AS P
ON P.productoid = DO.productoid
INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid;
```

Sin embargo, si ejecutamos la consulta, notaremos que todavía devuelve el mismo resultado que antes. Lógicamente, el left outer join entre Clientes y Pedidos devuelve los dos clientes que no realizaron pedidos con NULL en los atributos de pedido. Luego, al combinar el resultado con DetallesOrden usando un inner join, esas filas externas se eliminan. Esto se debe a que al comparar las marcas NULL en la columna O.ordenid en esas filas con cualquier valor DO.ordenid, el resultado es el valor lógico desconocido. Podemos generalizar este caso y decir que cualquier left outer join que sea seguido posteriormente por un inner join o un right outer join hace que el left outer join se convierta efectivamente en un inner join. Suponiendo que comparamos los elementos NULL del lado no conservado del left outer join con elementos de otra tabla. De hecho, si observamos el plan de ejecución de la consulta, notaremos que, de hecho, el optimizador convirtió el left outer join en un inner join.

Una forma común para intentar resolver el problema es hacer que todos los joins sean left outer, así:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
LEFT OUTER JOIN Ventas.Ordenes AS O
ON O.clienteid = C.clienteid
LEFT OUTER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid
LEFT OUTER JOIN Produccion.Productos AS P
ON P.productoid = DO.productoid
LEFT OUTER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid;
```

Hay dos problemas con esta solución. Un problema es que en realidad cambió el significado de la consulta del deseado. Buscaba un left outer join entre Clientes y el resultado de los inner join entre las tablas restantes. No se supone que las filas sin coincidencias en las tablas restantes se conserven; sin embargo, con esta solución, si quedan filas, se conservarán. El otro problema es que el optimizador tiene menos flexibilidad para alterar el orden de join cuando se trata de outer joins. Prácticamente la única flexibilidad que tiene es cambiar T1 LEFT OUTER JOIN T2 a T2 RIGHT OUTER JOIN T1.

Una mejor solución es comenzar con los inner join entre las tablas además de Clientes y luego aplicar un right outer join con los Clientes, así:



```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Ordenes AS O
INNER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid
INNER JOIN Produccion.Productos AS P
ON P.productoid = DO.productoid
INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid
RIGHT OUTER JOIN Ventas.Clientes AS C
ON C.clienteid = O.clienteid;
```

Ahora la solución es semánticamente correcta y deja más flexibilidad al optimizador para aplicar la optimización del orden de join. Esta consulta devuelve 1238 filas, que incluyen a los dos clientes que no realizaron pedidos.

Existe una solución aún más elegante que le permite expresar los join de una manera similar a como pensamos en la tarea. Recordemos que deseamos aplicar un left outer join entre Clientes y una unidad que represente el resultado de los inner joins entre las tablas restantes. Para definir dicha unidad, simplemente encerramos los inner join entre paréntesis y movemos el predicado de join que relaciona a los Clientes y esa unidad después del paréntesis, así:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
LEFT OUTER JOIN
( Ventas.Ordenes AS O
INNER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid
INNER JOIN Produccion.Productos AS P
ON P.productoid = DO.productoid
INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid)
ON O.clienteid = C.clienteid;
```

Curiosamente, los paréntesis no son realmente necesarios. Si los eliminamos y ejecutamos el código, veremos que se ejecuta correctamente y devuelve el resultado correcto. Lo que realmente marca la diferencia es la disposición parecida a paréntesis de las unidades y la ubicación de las cláusulas ON que reflejan esta disposición. Simplemente debemos asegurarnos de que la cláusula ON que se supone que relaciona dos unidades aparezca justo después de ellas; de lo contrario, la consulta no será válida. Con esto en mente, a continuación, se presenta otra disposición válida de las unidades, logrando el resultado deseado para nuestra tarea:

```
USE TSQLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
LEFT OUTER JOIN Ventas.Ordenes AS O
INNER JOIN Ventas.DetallesOrden AS DO
INNER JOIN Produccion.Productos AS P
INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid
ON P.productoid = DO.productoid
ON DO.ordenid = O.ordenid
ON O.clienteid = C.clienteid;
```

El uso de paréntesis acompañados de una sangría adecuada da como resultado un código mucho más claro, por lo que recomendamos usarlos, aunque no sean necesarios.

Anteriormente mencionamos que una de las heurísticas que usa el optimizador para reducir el tiempo de optimización es no considerar planes complicados (tupidos) que involucran joins entre resultados de joins. Si



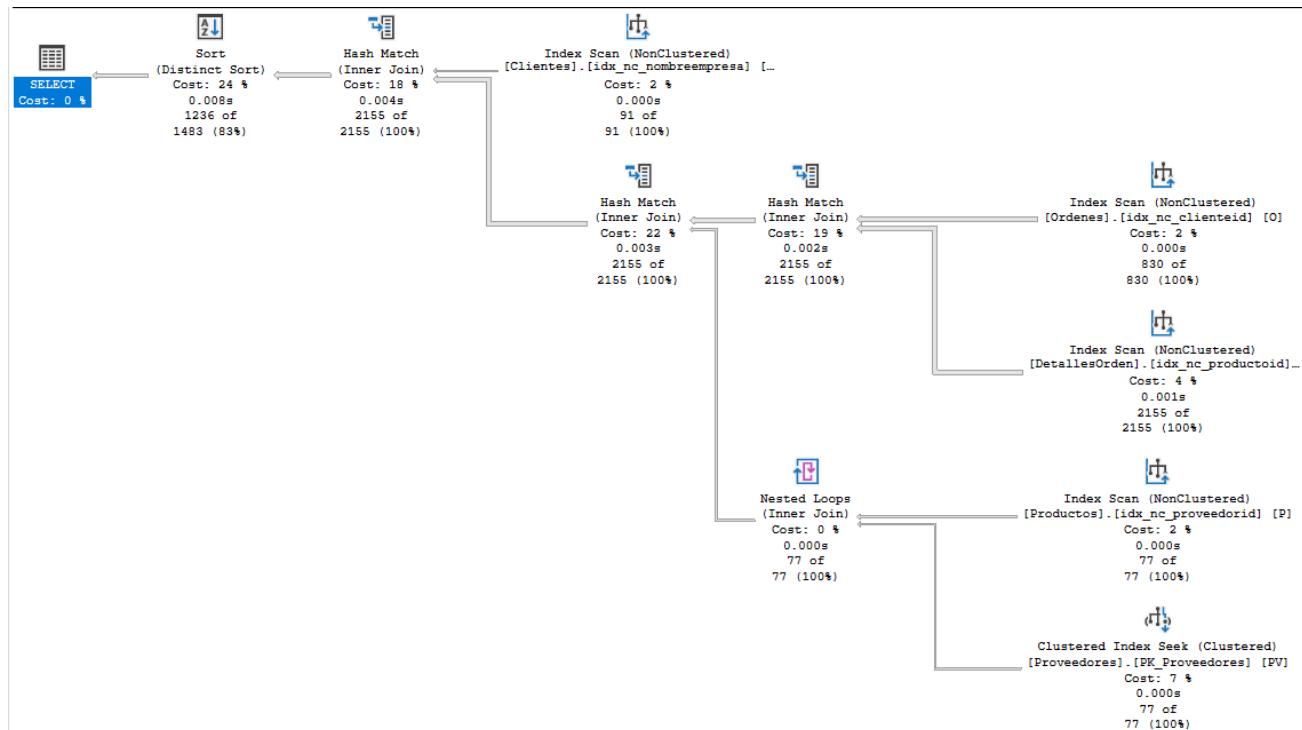
sospechamos que un plan tan complicado es el más eficiente y deseamos que el optimizador lo use, debemos forzarlo. La forma de lograr esto es definiendo dos unidades, cada una basada en una combinación con la cláusula ON respectiva, seguida de una cláusula ON que relaciona las dos unidades. También debemos especificar la opción de consulta FORCE ORDER.

A continuación, se muestra un ejemplo de cómo forzar un plan tan complicado:

```
USE TSQVLV6ES;
SELECT DISTINCT C.nombreempresa AS cliente, PV.nombreempresa AS proveedor
FROM Ventas.Clientes AS C
INNER JOIN
(Ventas.Ordenes AS O INNER JOIN Ventas.DetallesOrden AS DO
ON DO.ordenid = O.ordenid)
INNER JOIN
(Produccion.Productos AS P INNER JOIN Produccion.Proveedores AS PV
ON PV.proveedorid = P.proveedorid)
ON P.productoid = DO.productoid
ON O.clienteid = C.clienteid
OPTION (FORCE ORDER);
```

Acá tenemos una unidad (llamémosla U1) basada en un join entre Ordenes y DetallesOrden. Tenemos otra unidad (llamémosla U2) basada en un join entre Productos y Proveedores. Entonces tenemos un join entre las dos unidades (llamemos al resultado U1-2).

Luego tenemos un join entre Clientes y U1-2. El plan de ejecución de esta consulta se muestra en la siguiente imagen:



Observe que en el plan se logró el diseño tupido deseado.

SEMI Y ANTI SEMI JOINS

Normalmente, un join combina filas de dos tablas y devuelve elementos de ambos lados. Lo que hace que un join sea un semi join es que devuelve elementos de solo uno de los lados. El lado desde el que devuelve los elementos determina si se trata de un semi join izquierdo o derecho.



Como ejemplo, considere una solicitud para devolver el ID de cliente y el nombre de la empresa de los clientes que realizaron pedidos.

Devuelve información solo de la tabla Clientes, siempre que se encuentre una fila coincidente en la tabla Pedidos.

Considerando la tabla Clientes como la tabla de la izquierda, la operación es un semi join izquierdo. Hay varias formas de implementar la tarea. Una es usar un inner join y aplicar una cláusula DISTINCT para eliminar la información del cliente duplicada, así:

```
USE TSQLV6ES;
SELECT DISTINCT C.clienteid, C.nombreempresa
FROM Ventas.Clientes AS C
INNER JOIN Ventas.Ordenes AS O
ON O.clienteid = C.clienteid;
```

Otra es usar el predicado EXISTS, así:

```
USE TSQLV6ES;
SELECT clienteid, nombreempresa
FROM Ventas.Clientes AS C
WHERE EXISTS(SELECT *
FROM Ventas.Ordenes AS O
WHERE O.clienteid = C.clienteid);
```

Ambas consultas obtienen el mismo plan. Por supuesto, hay más formas de lograr la tarea.

Un anti semi join es aquel en el que devuelve elementos de una tabla si no se puede encontrar una fila coincidente en una tabla relacionada. También acá, dependiendo de si devuelve información de la tabla de la izquierda o de la derecha, la combinación es un anti-semi join izquierdo o derecho. A modo de ejemplo, una solicitud de devolución de clientes que no realizaron pedidos se cumple con una operación anti-semi join izquierda.

Hay varias formas clásicas de lograr tal anti-semi join. Una es usar un left outer join entre Clientes y Ordenes y filtrar solo filas externas, así:

```
USE TSQLV6ES;
SELECT C.clienteid, C.nombreempresa
FROM Ventas.Clientes AS C
LEFT OUTER JOIN Ventas.Ordenes AS O
ON O.clienteid = C.clienteid
WHERE O.orderid IS NULL;
```

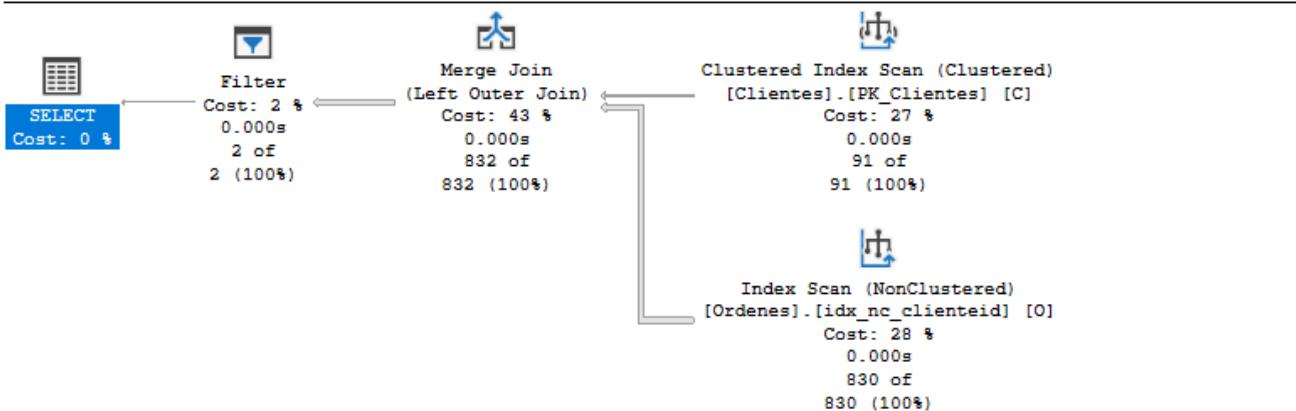
Otra es usar el predicado NOT EXISTS, así:

```
USE TSQLV6ES;
SELECT clienteid, nombreempresa
FROM Ventas.Clientes AS C
WHERE NOT EXISTS(SELECT *
FROM Ventas.Ordenes AS O
WHERE O.clienteid = C.clienteid);
```

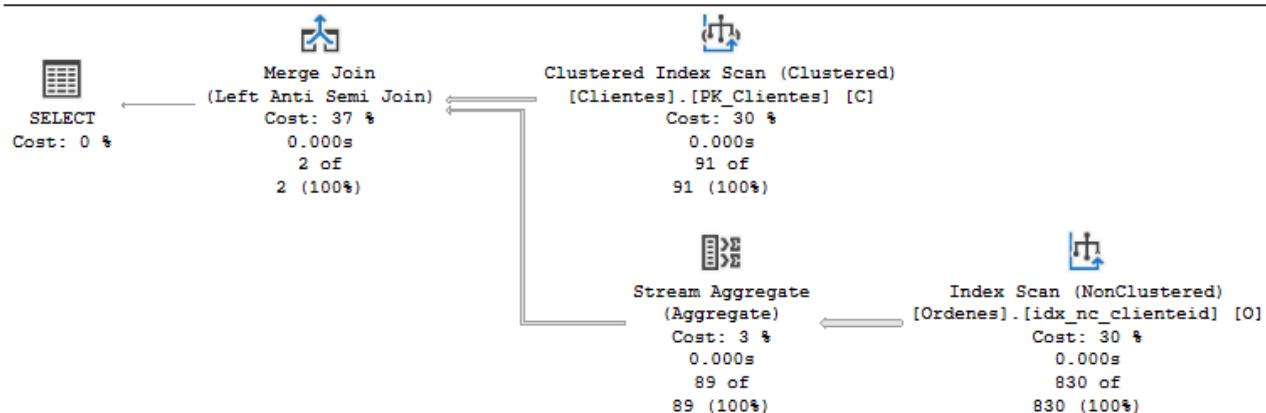
Curiosamente, parece que actualmente el optimizador no tiene lógica para detectar que la solución de outer join está aplicando realmente un anti-semi join, pero lo hace con la solución NOT EXISTS. Esto se puede ver en los planes de ejecución para estas consultas que se muestran en las siguientes imágenes:



Query 1: Query cost (relative to the batch): 52%
SELECT C.clienteid, C.nombreempresa FROM Ventas.Clientes AS C LEFT OUTER JOIN Ventas.Ordenes



Query 2: Query cost (relative to the batch): 48%
SELECT clienteid, nombreempresa FROM Ventas.Clientes AS C WHERE NOT EXISTS(SELECT * FROM Vent



Por supuesto, hay varias formas adicionales de implementar anti-semi joins. Los que vimos son dos clásicos.

ALGORITMOS DE JOIN

Los algoritmos de join son las estrategias de join físico que utiliza SQL Server para procesar los join. SQL Server admite tres algoritmos de join: bucles anidados, combinación y hash. Los bucles anidados es el más antiguo de los tres algoritmos y el respaldo cuando los demás no se pueden utilizar. Por ejemplo, el cross join solo se puede procesar con bucles anidados. La combinación y el hash requieren que el join sea un equi join o, más precisamente, que tenga al menos un predicado de join que se base en un operador de igualdad.

En el gráfico del plan de consulta, el operador de join tiene un icono distinto para cada algoritmo. Además, la propiedad Operación física indica el algoritmo de join (Bucles anidados, Coincidencia de hash o Combinación de join) y la propiedad Operación lógica indica el tipo de combinación lógica (inner join, left outer join, left anti semi join, etc.). Además, el nombre del algoritmo físico aparece justo debajo del operador y el tipo de join lógico aparece debajo del nombre del algoritmo entre paréntesis.

Las siguientes secciones describen los algoritmos de join individuales, las circunstancias en las que tienden a ser eficientes y las pautas de indexación para respaldarlos.

BUCLES ANIDADOS (NESTED LOOPS)

El algoritmo de bucles anidados es bastante sencillo. Ejecuta la entrada externa (superior) solo una vez y, utilizando un bucle, ejecuta la entrada interna (inferior) para cada fila de la entrada externa para identificar coincidencias.



Los bucles anidados tienden a funcionar bien cuando la entrada externa es pequeña y la entrada interna tiene un índice con la lista de claves basada en la columna de join más cualquier columna filtrada adicional, si es relevante. Si es fundamental que el índice sea de cobertura depende de la cantidad de coincidencias que obtenga en total. Si el número total de coincidencias es pequeño, algunas búsquedas no son demasiado costosas y, por lo tanto, el índice no tiene por qué ser de cobertura. Si la cantidad de coincidencias es grande, las búsquedas se vuelven costosas y luego se vuelve más crítico evitarlas haciendo que el índice sea de cobertura.

Como ejemplo, considere la siguiente consulta¹⁹:

```
USE PerformanceV3ES;
SELECT C.clienteid, C.nombreciente, O.ordenid, O.empid, O.transporteid, O.fechaorden
FROM dbo.Clientes AS C
INNER JOIN dbo.Ordenes AS O
ON O.clienteid = C.clienteid
WHERE C.nombreciente LIKE 'Cli_1000%'
AND O.fechaorden >= '20200101'
AND O.fechaorden < '20200401';
```

Actualmente, las tablas no tienen una buena indexación para admitir una combinación de bucles anidados eficientes. Debido a que el lado más pequeño (Clientes, en nuestro caso) generalmente se usa como entrada externa y se ejecuta solo una vez, no es tan crítico respaldarlo con un buen índice. En el peor de los casos, la mesa pequeña se escanea por completo. El índice mucho más crítico aquí está en el lado más grande (Órdenes, en nuestro caso). No obstante, si desea evitar una exploración completa de la tabla Clientes, puede preparar un índice en la columna filtrada nombreciente como clave e incluir la columna clienteid para la cobertura. En cuanto al índice ideal en la tabla Órdenes, piense en el trabajo involucrado en una sola iteración del ciclo para algún cliente X. Es como enviar la siguiente consulta:

```
SELECT ordenid, empid, transporteid, fechaorden
FROM dbo.Ordenes
WHERE clienteid = X
AND fechaorden >= '20200101'
AND fechaorden < '20200401';
```

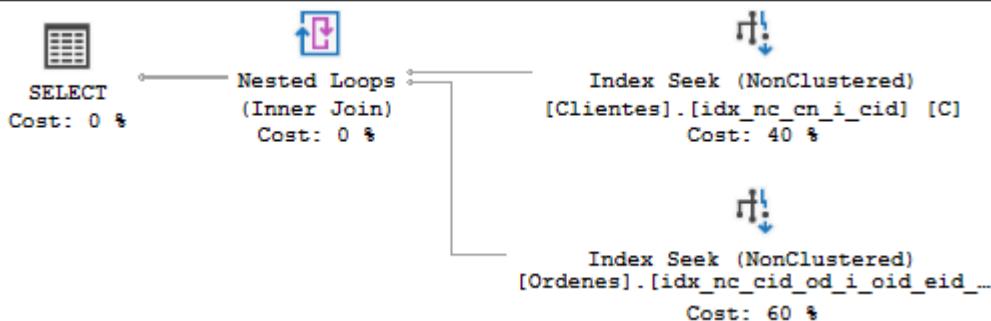
Debido a que tiene un predicado de igualdad y un predicado de rango, es bastante sencillo encontrar aquí un índice ideal. Definimos el índice con la lista de claves basada en clienteid y fechaorden, e incluye las columnas restantes (ordenid, empid y transporte) para la cobertura. Cuando tenemos un predicado de igualdad y un predicado de rango, primero debemos definir la lista de claves de índice con la columna de igualdad. De esta manera, las filas calificadas aparecerán en un rango consecutivo en la hoja de índice. Ese no será el caso si primero definimos la lista de claves con la columna de rango.

Ejecute el siguiente código para crear una indexación óptima para nuestra consulta:

```
USE PerformanceV3ES;
CREATE INDEX idx_nc_cn_i_cid ON dbo.Clientes(nombreciente) INCLUDE(clienteid);
CREATE INDEX idx_nc_cid_od_i_oid_eid_sid
ON dbo.Ordenes(clienteid, fechaorden) INCLUDE(ordenid, empid, transporteid);
```

Ahora ejecute la consulta y examine el plan de consulta de la imagen:

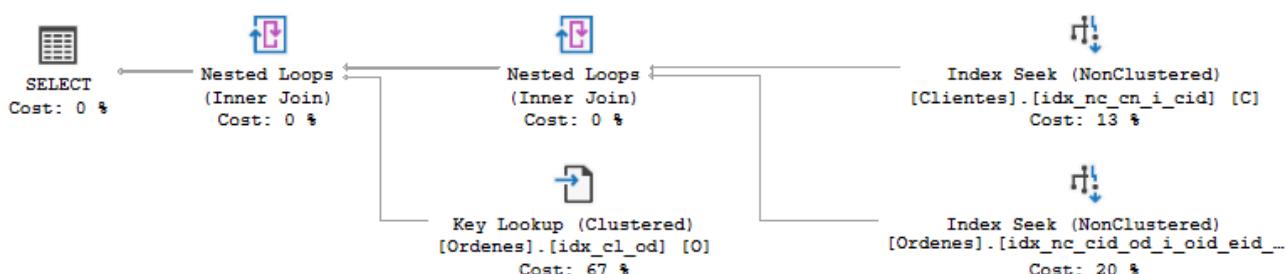
¹⁹ Para los ejemplos de lo que queda de la unidad se utilizará la base PerformanceV3ES



Como podemos ver, el optimizador eligió usar el algoritmo de bucles anidados con la tabla más pequeña, Clientes, como entrada externa y la tabla más grande, Ordenes, como entrada interna. El plan realiza una búsqueda y un escaneo de rango en el índice de cobertura de Clientes para recuperar a los clientes que califiquen. (Hay 11 en nuestro caso). Luego, usando un bucle, el plan realiza para cada cliente una búsqueda y un escaneo de rango en el índice de cobertura en Pedidos para recuperar pedidos coincidentes para el cliente actual. Hay 29 pedidos coincidentes en total.

A continuación, analizaremos los efectos que tendrán los diferentes cambios en la consulta en su optimización.

Como se mencionó, si tiene una pequeña cantidad de coincidencias en total, no es tan crítico que el índice en la tabla interna sea de cobertura. Puede probar tal caso agregando O.relleno a la lista SELECT de la consulta. El plan seguirá utilizando el algoritmo de bucles anidados, pero agregará búsquedas de claves para recuperar la columna relleno del índice agrupado porque no forma parte del índice idx_nc_cid_od_i_oid_eid_sid. Cuando haya terminado de probar, asegúrese de eliminar la columna O.relleno de la consulta antes de aplicar el siguiente cambio.



Cuando se devuelven suficientes filas de la entrada externa, el optimizador generalmente aplica una captación previa (una lectura anticipada) para acelerar las ejecuciones de la entrada interna. Por lo general, el optimizador considera una captación previa cuando estima que se devolverán más de un par de docenas de filas de la entrada externa. El plan de nuestra consulta actual no implica una captación previa porque el optimizador estima que los Clientes solo devolverán unas pocas filas. Pero si cambia el filtro de consulta por Clientes de C.nombrecliente LIKE 'Cli_1000%' a C.nombrecliente LIKE 'Cli_100%', se utilizará una captación previa. Después de aplicar el cambio, encontrará que el operador Nested Loops tiene una propiedad llamada WithUnorderedPrefetch, que se establece en True. La razón por la que se trata de una captación previa desordenada es que no es relevante devolver las filas ordenadas por los ID de cliente. Si agrega ORDER BY cliente a la consulta, la propiedad cambiará a WithOrderedPrefetch. Cuando haya terminado de probar, asegúrese de volver a la consulta original antes de aplicar el siguiente cambio.

Mencioné que el operador Nested Loops suele ser eficiente cuando la entrada externa es pequeña. Cambie el filtro de Clientes a C.nombrecliente LIKE 'Cust_10%'. Ahora, el número estimado de coincidencias de los Clientes aumenta de unas pocas a más de mil, y el optimizador cambia su elección del algoritmo de join a hash.



Existe una técnica interesante que el optimizador puede usar cuando el plan es paralelo, pero solo hay unas pocas filas externas. En tal caso, sin la intervención del optimizador, el trabajo podría distribuirse de manera desigual entre los diferentes subprocessos.

Por lo tanto, cuando el optimizador detecta un caso de algunas filas externas, agrega un operador Redistribute Streams después de obtener las filas de la entrada externa. Este operador redistribuye las filas entre los subprocessos de manera más uniforme.

MERGE (COMBINACIÓN)

El algoritmo de combinación requiere que ambas entradas estén ordenadas por la columna de join. Luego fusiona los dos como una cremallera. Si la combinación es de uno a varios, como es probable que sea el caso en la mayoría de las consultas, solo hay una pasada por cada entrada. Si el tipo de combinación es de varios a varios, una entrada se escanea una vez y la otra implica rebobinados.

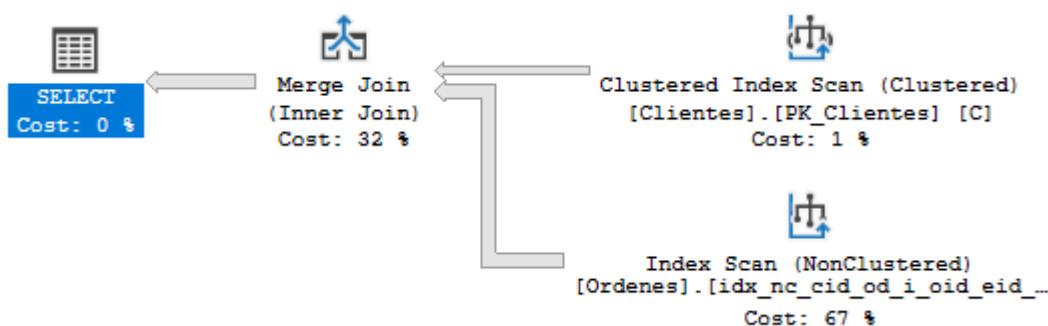
Generalmente, el optimizador tiene dos formas de obtener una entrada ordenada. Si los datos ya están ordenados en un índice, el optimizador puede usar un escaneo de orden de índice. La otra forma es agregar una operación de clasificación explícita al plan, pero luego está el costo adicional asociado con la clasificación.

Considere la siguiente consulta como un ejemplo de obtención de datos ordenados de índices:

```
USE PerformanceV3ES;
SELECT C.clienteid, C.nombrecliente, O.ordenid, O.empid, O.transporteid, O.fechaorden
FROM dbo.Clientes AS C
INNER JOIN dbo.Ordenes AS O
ON O.clienteid = C.clienteid;
```

El índice PK_Clientes de la tabla Clientes es un índice agrupado definido en clienteid como clave. Por lo tanto, los datos de Clientes están cubiertos por el índice y ordenados como lo requiere el algoritmo de combinación de join. El índice no agrupado idx_nc_cid_od_i_oid_eid_sid en las Ordenes que creamos para el ejemplo en la sección "Bucles anidados" es bastante eficaz para una combinación de join. Tiene los datos ordenados por clienteid como clave principal y cubre los elementos de la consulta de Ordenes.

Las condiciones son óptimas para un algoritmo de combinación de join; por lo tanto, el optimizador usa esta estrategia, como puede ver en el plan de nuestra consulta que se muestra en la siguiente imagen:



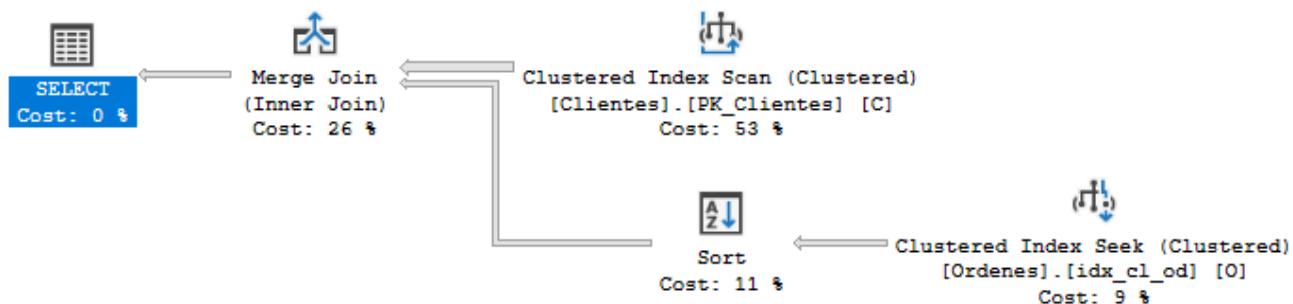
Como ejemplo de una combinación de join que implica una ordenación explícita, considere la siguiente consulta:

```
USE PerformanceV3ES;
SELECT C.clienteid, C.nombrecliente, O.ordenid, O.empid, O.transporteid, O.fechaorden
FROM dbo.Clientes AS C
INNER JOIN dbo.Ordenes AS O
ON O.clienteid = C.clienteid
WHERE O.fechaorden >= '20200101'
AND O.fechaorden < '20200102';
```



Los datos se pueden obtener ordenados por clienteid del índice agrupado en la tabla Clientes. Sin embargo, con el filtro de rango adicional contra Órdenes, es imposible crear un índice que ordene los datos principalmente por fecha de pedido para admitir el filtro y clienteid para admitir una combinación de join. Aún así, el filtro de intervalo de fechas es selectivo: solo hay unos pocos cientos de filas calificadas. Además, el índice agrupado de Órdenes se define con la fecha de la orden como clave. Entonces, el optimizador calcula que puede usar una búsqueda y un escaneo de rango contra el índice agrupado en Órdenes para recuperar los pedidos calificados, y luego, con una operación de clasificación explícita, ordenar las filas por clienteid.

Con una pequeña cantidad de filas, el costo de clasificación es bastante bajo. El plan para nuestra consulta, de hecho, implementa esta estrategia como puede ver en la siguiente imagen:



Observemos el pequeño porcentaje de costo asociado con la operación de Sort.

Tengamos en cuenta que dicho plan es sensible a las inexactitudes en la estimación de cardinalidad de la actividad que conduce al operador de Sort. Ese es especialmente el caso cuando la estimación es para una pequeña cantidad de filas, pero la real es una gran cantidad, porque la clasificación no se escala bien. Por un lado, una estimación más precisa podría llevar al optimizador a utilizar un algoritmo de join diferente por completo. Por otro lado, la estimación baja podría resultar en una concesión de memoria insuficiente para la actividad de ordenación, y esto podría provocar derrames en tempdb. Debemos estar atentos a estos problemas y, si detectamos uno, intentemos encontrar formas de ayudar al optimizador a obtener una estimación más precisa.

HASH

El algoritmo hash implica la creación de una tabla hash basada en una de las entradas, generalmente la más pequeña. La razón por la que se prefiere el lado pequeño como entrada de compilación es que la huella de memoria suele ser menor de esta manera. Si la tabla hash cabe en la memoria, es ideal. De lo contrario, la tabla hash se puede particionar, pero el intercambio de particiones dentro y fuera de la memoria hace que el algoritmo sea menos eficiente. Los elementos de la entrada de compilación se organizan en depósitos hash según la función hash elegida. Con fines ilustrativos, supongamos que la columna de combinación es un número entero y el optimizador elige una función de módulo 50 como función hash. Se crearán hasta 50 depósitos hash, con todos los elementos que obtuvieron el mismo resultado de la función hash organizados en una lista vinculada en el mismo depósito. Luego, se prueba la otra entrada (conocida como la entrada de la sonda), la función hash se aplica a los valores de la columna de unión y, en función del resultado, el algoritmo escanea el contenedor hash respectivo para buscar coincidencias.

El algoritmo hash sobresale en consultas de datawarehouse, que tienden a involucrar grandes cantidades de datos en la tabla de hechos y tablas de dimensiones mucho más pequeñas. Tiende a escalar bien con este tipo de consultas, beneficiándose enormemente del procesamiento en paralelo. Sin embargo, tengan en cuenta que la tabla hash requiere una concesión de memoria y, por lo tanto, este algoritmo es sensible a las inexactitudes con las estimaciones de cardinalidad. Si hay una concesión de memoria insuficiente, la tabla hash se derramará en tempdb y esto dará como resultado un procesamiento menos eficiente.



Cuando se usan índices de almacén de columnas, el algoritmo hash puede usar el modo de ejecución por lotes. En SQL Server 2012, solo los inner join procesados con un algoritmo hash podrían usar la ejecución por lotes. Además, si el operador se derramaba en tempdb, el modo de ejecución cambiaba al modo de fila. En SQL Server 2014, los outer joins procesados con un algoritmo hash pueden usar la ejecución por lotes, al igual que los operadores que se derraman en tempdb.

Además de las consultas clásicas de datawarehouse, existe otro caso interesante en el que el optimizador tiende a elegir el algoritmo hash, cuando no hay buenos índices para admitir los otros algoritmos. Si lo pensamos, una tabla hash es una estructura de búsqueda alternativa a un árbol B. Cuando no hay buenos índices existentes, generalmente es más eficiente en general crear una tabla hash, usarla y soltarla en cada ejecución de consulta en comparación con hacer lo mismo con un árbol B.

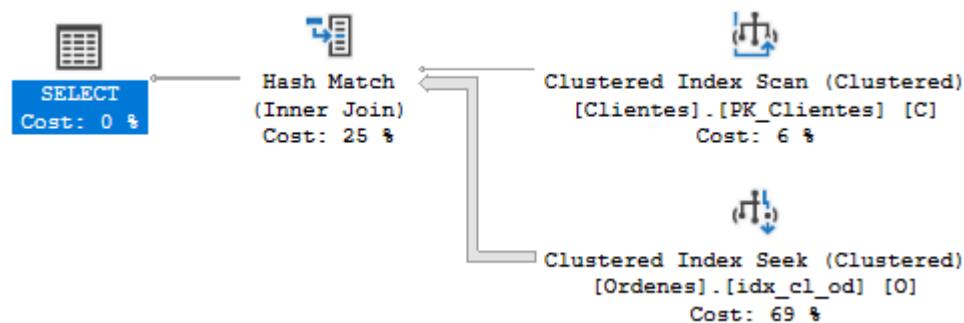
Como ejemplo, ejecute el siguiente código para eliminar los dos índices que creó anteriormente para admitir ejemplos anteriores:

```
USE PerformanceV3ES;
DROP INDEX idx_nc_cn_i_cid ON dbo.Clientes;
DROP INDEX idx_nc_cid_od_i_oid_eid_sid ON dbo.Ordenes;
```

Luego, ejecutemos la misma consulta que ejecutamos anteriormente en la sección "Bucles anidados":

```
USE PerformanceV3ES;
SELECT C.clienteid, C.nombreciente, O.ordenid, O.empid, O.transporteid, O.fechaorden
FROM dbo.Clientes AS C
INNER JOIN dbo.Ordenes AS O
ON O.clienteid = C.clienteid
WHERE C.nombreciente LIKE 'Cli_1000%'
AND O.fechaorden >= '20200101'
AND O.fechaorden < '20200401';
```

Esta vez, como resultado de la ausencia de índices eficientes para admitir un algoritmo de bucles anidados, el optimizador elige usar una join hash con una tabla hash basada en los clientes calificados. El plan para esta consulta se muestra en la siguiente imagen:



Por lo tanto, cuando veamos que el optimizador usa un join de hash en los casos en los que no parece natural, tenemos que preguntarnos si faltan índices importantes, especialmente si la consulta es frecuente en el sistema.

FORZAR LA ESTRATEGIA DE JOIN

Si sospechamos que el optimizador eligió un algoritmo de join subóptimo, podemos probar un par de métodos para forzar el que creamos que es más óptimo. Una es usar una sugerencia (hint) de join donde especificamos el algoritmo de join (LOOP, MERGE o HASH) justo antes de la palabra clave JOIN, usando la sintaxis de combinación completa (INNER <hint> JOIN, LEFT OUTER <hint> JOIN, etc.) A continuación, se muestra un ejemplo para forzar un algoritmo de bucles anidados:



```
USE PerformanceV3ES;
SELECT C.clienteid, C.nombreciente, O.ordenid, O.empid, O.transporteid, O.fechaorden
FROM dbo.Clientes AS C
INNER LOOP JOIN dbo.Ordenes AS O
ON O.clienteid = C.clienteid;
```

Tengamos en cuenta que con este método también forzamos el orden de join. La tabla de la izquierda se usa como entrada externa y la tabla de la derecha se usa como la interna.

Otro método consiste en utilizar sugerencias de consulta en la cláusula OPTION de consulta. La ventaja de este método es que puede especificar más de un algoritmo. Debido a que solo hay tres algoritmos admitidos, al enumerar dos, se excluye el tercero. Por ejemplo, supongamos que deseamos evitar que el optimizador elija el algoritmo de combinación, pero desea dejarlo con cierta flexibilidad para elegir entre bucles anidados y hash. Logramos esto así:

```
USE PerformanceV3ES;
SELECT C.clienteid, C.nombreciente, O.ordenid, O.empid, O.transporteid, O.fechaorden
FROM dbo.Clientes AS C
INNER JOIN dbo.Ordenes AS O
ON O.clienteid = C.clienteid
OPTION(LOOP JOIN, HASH JOIN);
```

Sin embargo, tengamos en cuenta que al utilizar este método, todos los join de la consulta se ven afectadas. También tengamos en cuenta que, a diferencia de la sugerencia de join, las sugerencias de consulta para los algoritmos de join no afectan el orden de join. Si deseamos forzar el orden de join, debe agregar la sugerencia de consulta FORCE ORDER.

Estos consejos son útiles como herramienta de resolución de problemas de rendimiento; sin embargo, su uso en consultas de producción elimina la naturaleza dinámica de la optimización que normalmente puede reaccionar ante circunstancias cambiantes cambiando las opciones de optimización. Si se encuentra utilizando tales sugerencias en el código de producción para resolver problemas críticos, debería considerarlos una solución temporal. Cuando tenga tiempo, podrá investigar más a fondo el problema y encontrar una solución que ayude al optimizador a tomar decisiones más óptimas de forma natural.



OPERADORES UNION, EXCEPT E INTERSECT

Los operadores UNION, EXCEPT e INTERSECT son operadores relacionales que combinan filas de los resultados de dos consultas. La forma general de utilizarlos es:

```
<consulta 1>
<operador>
<consulta 2>
[ORDER BY <lista_de_ordenamiento>];
```

El operador UNION unifica las filas de las dos entradas. El operador INTERSECT devuelve solo las filas que son comunes a ambas entradas. El operador EXCEPT devuelve las filas que aparecen en la primera entrada, pero no en la segunda.

Las consultas de entrada no pueden tener una cláusula ORDER BY porque se supone que deben devolver un resultado relacional. Sin embargo, se permite una cláusula ORDER BY contra el resultado del operador.

Los esquemas de las consultas de entrada deben ser compatibles. El número de columnas tiene que ser el mismo, y los tipos deben ser implícitamente convertibles de la que tiene la precedencia de tipo de datos más baja a la más alta. Los nombres de las columnas de resultados se definen mediante la primera consulta.

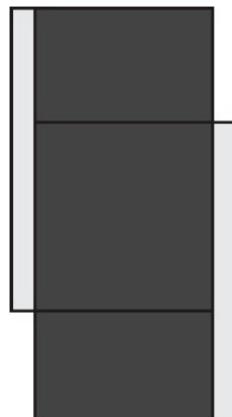
Al comparar filas entre las entradas, estos operadores utilizan implícitamente el predicado estándar DISTINCT. Según este predicado, un NULL no es distinto de otro NULL y un NULL es distinto de un valor no NULL. Por el contrario, según el operador de igualdad, comparar dos NULL da como resultado el valor lógico desconocido, y lo mismo se aplica al comparar un valor NULL con un valor no NULL. Este hecho hace que estos operadores sean sencillos e intuitivos de usar, incluso cuando son posibles NULL en los datos.

Cuando se utilizan varios operadores en una consulta sin paréntesis, INTERSECT precede a UNION y EXCEPT. Los dos últimos tienen la misma precedencia, por lo que se evalúan según el orden de aparición. Siempre podemos forzar el orden de evaluación deseado utilizando paréntesis.

Las siguientes secciones proporcionan más detalles sobre estos operadores.

EL OPERADOR UNION

El operador UNION consolida los resultados de las dos consultas de entrada. Si una fila aparece en alguno de los conjuntos de entrada, aparecerá en el resultado de la operación UNION. T-SQL implementa el operador UNION ALL y UNION (con la opción DISTINCT implícita).



La siguiente figura ilustra al operador UNION. El área sombreada representa el resultado del operador. Las áreas no sombreadas representan el hecho de que el operador puede no incluir todos los atributos de las relaciones originales.

EL OPERADOR UNION ALL

El operador UNION ALL consolida los resultados de las dos consultas sin intentar remover duplicados del resultado final. Si la consulta 1 devuelve m filas y la consulta 2 devuelve n filas, el resultado de UNION ALL tendrá m + n filas.

Por ejemplo, en la siguiente consulta utilizaremos UNION ALL para consolidar las ubicaciones de empleados y clientes.

```
SELECT pais, region, ciudad FROM RH.Empleados
UNION ALL
SELECT pais, region, ciudad FROM Ventas.Clientes;
```



El resultado tiene 100 filas – 9 de empleados y 91 de clientes –

pais	region	ciudad
USA	WA	Seattle
USA	WA	Tacoma
USA	WA	Kirkland
USA	WA	Redmond
UK	NULL	London
UK	NULL	London
UK	NULL	London
...		
Finland	NULL	Oulu
Brazil	SP	Resende
USA	WA	Seattle
Finland	NULL	Helsinki
Poland	NULL	Warszawa

Como UNION ALL no elimina los duplicados, el resultado es un multiset, es decir, que una misma fila puede aparecer varias veces, como de hecho podemos ver en el ejemplo.

EL OPERADOR UNION (DISTINCT)

El operador UNION (con la opción DISTINCT implícita) consolida los resultados de las dos consultas de entrada y elimina los duplicados. Si una fila aparece en las dos consultas, sólo aparecerá una vez en el resultado final. En otras palabras, el resultado es un conjunto, y no un multiset.

Veamos el mismo ejemplo de antes, pero aplicando UNION:

```
SELECT pais, region, ciudad FROM RH.Empleados  
UNION  
SELECT pais, region, ciudad FROM Ventas.Clientes;
```

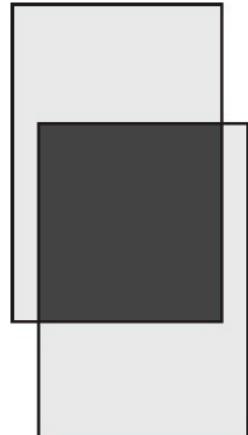
Esta vez obtenemos 71 filas en lugar de 100.

pais	region	ciudad
Argentina	NULL	Buenos Aires
Austria	NULL	Graz
Austria	NULL	Salzburg
Belgium	NULL	Bruxelles
Belgium	NULL	Charleroi
Brazil	RJ	Rio de Janeiro
Brazil	SP	Campinas
Brazil	SP	Resende
Brazil	SP	Sao Paulo
Canada	BC	Tsawassen
...		
USA	WA	Seattle
USA	WA	Tacoma
USA	WA	Walla Walla
USA	WY	Lander
Venezuela	DF	Caracas
Venezuela	Lara	Barquisimeto
Venezuela	Nueva Esparta	I. de Margarita
Venezuela	Táchira	San Cristóbal



EL OPERADOR INTERSECT

El operador INTERSECT devuelve sólo las filas que son comunes al resultado de las consultas de entrada.



En la figura podemos verlo gráficamente.

EL OPERADOR INTERSECT (DISTINCT)

El operador INTERSECT (con la opción DISTINCT implícita) devuelve las filas que aparecen en los resultados de ambas consultas de entrada, eliminando duplicados. Si una fila aparece al menos una vez en cada resultado, formará parte del resultado final.

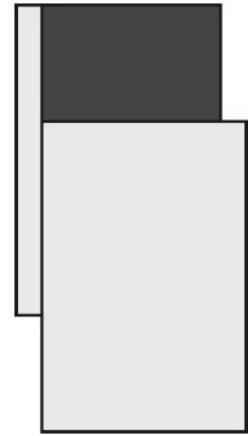
La siguiente consulta devuelve las ubicaciones en las que coinciden empleados y clientes.

```
SELECT pais, region, ciudad FROM RH.Empleados  
INTERSECT  
SELECT pais, region, ciudad FROM Ventas.Clientes;
```

pais	region	ciudad
UK	NULL	London
USA	WA	Kirkland
USA	WA	Seattle

EL OPERADOR EXCEPT

El operador EXCEPT implementa la diferencia de conjuntos. Opera sobre los resultados de dos consultas, y devuelve las filas que aparecen como resultado de la primera consulta, pero no en el resultado de la segunda consulta.



EL OPERADOR EXCEPT (DISTINCT)

El operador EXCEPT (con la opción DISTINCT implícita) devuelve las filas que aparecen como resultado de la primera consulta, pero no en el resultado de la segunda consulta, eliminando duplicados.

Una fila de la primera consulta terminará como parte del resultado si aparece al menos una vez en la primera consulta y no aparece en la segunda.

Noten que en las operaciones de UNION e INTERSECT el orden en que se presenten las consultas no influye en el resultado final, pero en EXCEPT sí.

Por ejemplo, la siguiente consulta devuelve las ubicaciones de empleados en las que no hay clientes.

```
SELECT pais, region, ciudad FROM RH.Empleados  
EXCEPT  
SELECT pais, region, ciudad FROM Ventas.Clientes;
```

pais	region	ciudad
USA	WA	Redmond
USA	WA	Tacoma

Si damos vuelta las consultas, obtenemos las ubicaciones de clientes, en la que no hay empleados, y en ese caso obtendríamos 66.



PRECEDENCIA

SQL define precedencia entre los operadores de conjuntos. El operador INTERSECT precede a los operadores UNION y EXCEPT, y UNION y EXCEPT son evaluados en orden de aparición.

Por ejemplo:

```
SELECT pais, region, ciudad FROM Produccion.Proveedores  
EXCEPT  
SELECT pais, region, ciudad FROM RH.Empleados  
INTERSECT  
SELECT pais, region, ciudad FROM Ventas.Clientes;
```

Devuelve 28 filas. Pero la siguiente consulta devuelve 3:

```
(SELECT pais, region, ciudad FROM Produccion.Proveedores  
EXCEPT  
SELECT pais, region, ciudad FROM RH.Empleados)  
INTERSECT  
SELECT pais, region, ciudad FROM Ventas.Clientes;
```

pais	region	ciudad
Canada	Québec	Montréal
France	NULL	Paris
Germany	NULL	Berlin

En el primer caso se realiza primero la operación INTERSECT, y luego EXCEPT.

En el segundo caso, mediante el uso de paréntesis, se realiza primero la operación EXCEPT, y luego INTERSECT.

EJEMPLOS

Realizar un reporte para obtener el empid y el ordenid de la tabla Ordenes no es complicado:

```
SELECT empid, ordenid  
FROM Ventas.Ordenes;
```

Pero para que pueda brindar información más útil, podemos hacer lo siguiente:

```
-- Crear un reporte que muestre las ordenes de un empleado.  
SELECT RH.Empleados.empid, RH.Empleados.nombre,  
       RH.Empleados.apellido, Ventas.Ordenes.ordenid, Ventas.Ordenes.fechaorden  
  FROM RH.Empleados JOIN Ventas.Ordenes ON  
        (RH.Empleados.empid = Ventas.Ordenes.empid)  
 ORDER BY Ventas.Ordenes.fechaorden;
```

empid	nombre	apellido	ordenid	fechaorden
5	Sven	Mortensen	10248	2020-07-04
6	Paul	Suurs	10249	2020-07-05
4	Yael	Peled	10250	2020-07-08
3	Judy	Lew	10251	2020-07-08
4	Yael	Peled	10252	2020-07-09
3	Judy	Lew	10253	2020-07-10
5	Sven	Mortensen	10254	2020-07-11
...				
8	Maria	Cameron	11075	2022-05-06
4	Yael	Peled	11076	2022-05-06
1	Sara	Davis	11077	2022-05-06



```
/* Crear un reporte que muestre OrdenID, y nombre de la empresa que realizó la orden, y el nombre y apellido del empleado que la ingresó.  
Sólo mostrar órdenes efectuadas después del 1 de Enero de 2008 que fueron despachadas luego de la fecha requerida.  
Ordenar por Nombre de empresa */
```

```
SELECT o.ordenid, c.nombreempresa, e.nombre, e.apellido  
FROM Ventas.Ordenes o  
JOIN RH.Empleados e ON (e.empid = o.empid)  
JOIN Ventas.Clientes c ON (c.clienteid = o.clienteid)  
WHERE o.fechadespacho > o.fecharequerida AND o.fechaorden > '20080101'  
ORDER BY c.nombreempresa;
```

ordenid	nombreempresa	nombre	apellido
10523	Customer AHPOP	Russell	King
10264	Customer CYZTN	Paul	Suurs
10927	Customer EFFTCA	Yael	Peled
10687	Customer FRXZL	Patricia	Doyle
10309	Customer FRXZL	Judy	Lew
10380	Customer FRXZL	Maria	Cameron
10280	Customer HGVLZ	Don	Funk
10924	Customer HGVLZ	Judy	Lew
10515	Customer IRRVL	Don	Funk
10451	Customer IRRVL	Yael	Peled
10847	Customer LCOUJ	Yael	Peled
10727	Customer LHANT	Don	Funk
10749	Customer LJUCA	Yael	Peled
10427	Customer LOLJO	Yael	Peled
10660	Customer LVJSO	Maria	Cameron
10828	Customer LWGMD	Patricia	Doyle
10593	Customer OXFNU	Russell	King
10779	Customer PZNLA	Judy	Lew
10970	Customer QUHWH	Patricia	Doyle
10433	Customer QZURI	Judy	Lew
10726	Customer RFNQC	Yael	Peled
10827	Customer RTXGC	Sara	Davis
10663	Customer RTXGC	Don	Funk
10302	Customer SFOGW	Yael	Peled
10578	Customer UBHAU	Yael	Peled
10545	Customer UISOJ	Maria	Cameron
10705	Customer UMTLM	Patricia	Doyle
10960	Customer UMTLM	Judy	Lew
10807	Customer WMFEA	Yael	Peled
10271	Customer XOJYP	Paul	Suurs
10483	Customer YBQTI	Russell	King
10596	Customer YBQTI	Maria	Cameron
10709	Customer YJC BX	Sara	Davis
10777	Customer YJC BX	Russell	King
10423	Customer YJC BX	Paul	Suurs
10816	Customer YSIQX	Yael	Peled
10320	Customer ZHYOS	Sven	Mortensen

```
/* Crear un reporte que muestre el número de empleados y clientes de cada ciudad que tenga empleados en ella */  
SELECT COUNT(DISTINCT e.empid) AS numEmpleados,  
       COUNT(DISTINCT c.clienteid) AS numEmpresas,  
       e.ciudad, c.ciudad  
  FROM RH.Empleados e JOIN Ventas.Clientes c ON  
    (e.ciudad = c.ciudad)  
 GROUP BY e.ciudad, c.ciudad  
 ORDER BY numEmpleados DESC;
```



numEmpleados	numEmpresas	ciudad	ciudad
4	6	London	London
2	1	Seattle	Seattle
1	1	Kirkland	Kirkland

/* Crear un reporte que muestre el número de empleados y clientes de cada ciudad que tenga empleados en ella */

```
SELECT COUNT(DISTINCT e.empid) AS numEmployees,
       COUNT(DISTINCT c.clienteid) AS numCompanies,
       e.ciudad, c.ciudad
  FROM RH.Empleados e LEFT JOIN Ventas.Clientes c ON
    (e.ciudad = c.ciudad)
 GROUP BY e.ciudad, c.ciudad
 ORDER BY numEmployees DESC;
```

numEmployees	numCompanies	ciudad	ciudad
4	6	London	London
2	1	Seattle	Seattle
1	0	Tacoma	NULL
1	0	Redmond	NULL
1	1	Kirkland	Kirkland

Warning: Null value is eliminated by an aggregate or other SET operation.

/* Crear un reporte que muestre el número de empleados y clientes de cada ciudad que tenga clientes en ella */

```
SELECT COUNT(DISTINCT e.empid) AS numEmployees,
       COUNT(DISTINCT c.clienteid) AS numCompanies,
       e.ciudad, c.ciudad
  FROM RH.Empleados e RIGHT JOIN Ventas.Clientes c ON
    (e.ciudad = c.ciudad)
 GROUP BY e.ciudad, c.ciudad
 ORDER BY numEmployees DESC;
```

numEmployees	numCompanies	ciudad	ciudad
4	6	London	London
2	1	Seattle	Seattle
1	1	Kirkland	Kirkland
0	1	NULL	Kobenhavn
0	1	NULL	Köln
0	1	NULL	Lander
...			
0	1	NULL	Graz
0	1	NULL	Helsinki
0	1	NULL	I. de Margarita

Warning: Null value is eliminated by an aggregate or other SET operation.

/* Crear un reporte que muestre el número de empleados y clientes de cada ciudad */

```
SELECT COUNT(DISTINCT e.empid) AS numEmpleados,
       COUNT(DISTINCT c.clienteid) AS numClientes,
       e.ciudad, c.ciudad
  FROM RH.Empleados e FULL JOIN Ventas.Clientes c ON
    (e.ciudad = c.ciudad)
 GROUP BY e.ciudad, c.ciudad
 ORDER BY numEmpleados DESC;
```



numEmpleados	numClientes	ciudad	ciudad
4	6	London	London
2	1	Seattle	Seattle
1	0	Tacoma	NULL
1	0	Redmond	NULL
1	1	Kirkland	Kirkland
0	1	NULL	Aachen
0	1	NULL	Albuquerque
0	1	NULL	Anchorage
...			
0	1	NULL	Versailles
0	1	NULL	Walla Walla
0	1	NULL	Warszawa

Warning: Null value is eliminated by an aggregate or other SET operation.



UNIDAD 4

Esta unidad cubre temas de consultas que van más allá de las fundamentales y el objetivo es ir más allá en su compresión. Comenzaremos con las potentes funciones de ventana, que se pueden utilizar para aplicar cálculos de análisis de datos de una manera flexible y eficiente. Continuaremos con técnicas para pivotear y despivotear datos. El pivoteo rota los datos de un estado de filas a columnas, y el despivoteo rota datos de columnas a filas, similar a las tablas dinámicas en Excel. Y finalizaremos la unidad con una discusión sobre los datos agrupados, que son los conjuntos de expresiones resultantes de aplicar GROUP BY. Veremos técnicas para definir varios grupos de datos en la misma consulta.

Lo que tienen en común las funciones que veremos en esta unidad, además de ser más avanzadas que las funciones vistas hasta el momento, es que se utilizan principalmente con fines analíticos.

Para más información sobre estos temas se puede consultar: T-SQL Querying (Microsoft Press, 2015) y Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions (Microsoft Press, 2012).

FUNCIONES DE VENTANA²⁰

Al igual que las funciones de grupo, las funciones de ventana también le permiten realizar cálculos de análisis de datos. La diferencia entre los dos está, en cómo define el conjunto de filas para la función con la que trabaja. Con funciones de grupo, utiliza consultas agrupadas para organizar las filas consultadas en grupos, y luego las funciones de grupo son aplicadas a cada grupo. Obtiene una fila de resultado por grupo, no por fila originaria. Con funciones de ventana, se define el conjunto de filas por función, y luego retorna un valor de resultado por cada función y fila originaria. Se define el conjunto de filas para la función a trabajar con el uso de una cláusula llamada OVER.

FUNCIONES AGGREGATE DE VENTANA

Las funciones agregadas de ventana son las mismas, que las funciones agregadas de agrupación (por ejemplo, SUM, COUNT, AVG, MIN y MAX), excepto que las funciones agregadas de ventana son aplicadas a una ventana de filas definidas por la cláusula OVER.

Uno de los beneficios de usar funciones de ventana es que, a diferencia de las consultas agrupadas, las consultas con ventanas no ocultan el detalle, retornan una fila por cada fila de consulta originaria. Esto significa que puede mezclar detalles y elementos agregados en la misma consulta, e incluso en la misma expresión. Usando la cláusula OVER, se define un conjunto de filas para la función a trabajar por cada fila originaria. En otras palabras, una consulta de ventana define una ventana de filas por cada función y fila en la consulta originaria.

Como se ha mencionado, se utiliza una cláusula OVER para definir una ventana de filas para la función. La ventana es definida con respecto a la fila actual. Cuando utiliza paréntesis vacíos, la cláusula OVER representa el conjunto resultado de la consulta originaria completa. Por ejemplo, la expresión SUM(val) OVER() representa el total general de todas las filas en la consulta originaria. Puede utilizar una cláusula de partición de ventana para restringir la ventana. Por ejemplo, la expresión SUM(val) OVER(PARTITION BY clienteid) representa el total de clientes actuales. Como un ejemplo, si la fila actual tiene un clienteid de 1, la cláusula OVER filtra sólo estas filas del conjunto resultado de la consulta originaria, donde el clienteid es 1; por lo tanto, la expresión retorna el total para el cliente 1.

²⁰ <http://blogtsqlcastellano.blogspot.com.ar/2016/03/usando-funciones-de-ventana.html>



He aquí un ejemplo de una consulta sobre la vista Ventas.ValoresOrden, retornando para cada pedido el clienteid, ordenid, y el valor de pedido; utilizando funciones de ventana, la consulta también retorna el total general de todos los valores y el total por cliente.

```
USE TSQLV6ES;
SELECT clienteid, ordenid, val,
SUM(val) OVER(PARTITION BY clienteid) AS clientotal,
SUM(val) OVER() AS totalgeneral
FROM Ventas.ValoresOrden;
```

clienteid	ordenid	val	clientotal	totalgeneral
1	10643	814.50	4273.00	1265793.22
1	10692	878.00	4273.00	1265793.22
1	10702	330.00	4273.00	1265793.22
1	10835	845.80	4273.00	1265793.22
1	10952	471.20	4273.00	1265793.22
1	11011	933.50	4273.00	1265793.22
2	10926	514.40	1402.95	1265793.22
2	10759	320.00	1402.95	1265793.22
2	10625	479.75	1402.95	1265793.22
2	10308	88.80	1402.95	1265793.22
...				
91	10906	427.50	3531.95	1265793.22
91	10870	160.00	3531.95	1265793.22
91	10998	686.00	3531.95	1265793.22
91	11044	591.60	3531.95	1265793.22
91	10611	808.00	3531.95	1265793.22
91	10792	399.85	3531.95	1265793.22
91	10374	459.00	3531.95	1265793.22

El total general es, por supuesto, el mismo para todas las filas. El total del cliente es el mismo para todas las filas con el mismo clienteid.

Puede mezclar elementos de detalle y agregados de ventana en la misma expresión. Por ejemplo, la siguiente consulta, calcula para cada pedido el porcentaje del valor de pedido actual sobre el total del cliente, y también el porcentaje del total general.

```
USE TSQLV6ES;
SELECT clienteid, ordenid, val,
CAST(100.0*val/SUM(val) OVER(PARTITION BY clienteid) AS NUMERIC(5,2)) AS pctcliente,
CAST(100.0*val/SUM(val) OVER() AS NUMERIC(5,2)) AS pcttotal
FROM Ventas.ValoresOrden;
```

clienteid	ordenid	val	pctcliente	pcttotal
1	10643	814.50	19.06	0.06
1	10692	878.00	20.55	0.07
1	10702	330.00	7.72	0.03
1	10835	845.80	19.79	0.07
1	10952	471.20	11.03	0.04
1	11011	933.50	21.85	0.07
2	10926	514.40	36.67	0.04
2	10759	320.00	22.81	0.03
2	10625	479.75	34.20	0.04
2	10308	88.80	6.33	0.01
...				
91	10906	427.50	12.10	0.03
91	10870	160.00	4.53	0.01
91	10998	686.00	19.42	0.05
91	11044	591.60	16.75	0.05
91	10611	808.00	22.88	0.06
91	10792	399.85	11.32	0.03
91	10374	459.00	13.00	0.04

La suma de todos los porcentajes sobre el total general es 100. La suma de todos los porcentajes sobre el total de clientes es 100, por cada partición de filas con el mismo cliente.

Las funciones agregadas de ventana soportan, otra opción de filtrado llamado enmarcado. La idea es que defina el ordenamiento en la partición utilizando una cláusula de orden de ventana, y luego basado en ese orden, puede confinar un marco de filas entre dos delimitadores. Define los delimitadores utilizando una cláusula marco de



ventana. La cláusula marco de ventana requiere que una cláusula de orden de ventana esté presente, debido a que un conjunto no tiene orden; y sin orden, el limitar las filas entre dos delimitadores podría no tener sentido.

En la cláusula marco de ventana, indica las unidades marco de ventana (ROWS o RANGE) y las extensiones marco de ventana (los delimitadores). Con la unidad marco de ventana ROWS, puede indicar los delimitadores como una de las tres opciones:

- UNBOUNDED PRECEDING o FOLLOWING, es decir, el principio o el fin de la partición, respectivamente
- CURRENT ROW, obviamente, representando la fila actual
- $<n>$ ROWS PRECEDING o FOLLOWING, es decir, n filas antes o después de la actual, respectivamente

A modo de ejemplo, suponga que quiere consultar la vista Ventas.ValoresOrden y calcular los valores totales acumulados, desde el comienzo de la actividad del cliente actual hasta el pedido actual. Necesita utilizar el agregado SUM. Particionar la ventana por clienteid. Ordenar la ventana por fechaorden, ordenid. Luego, enmarcar las filas desde el inicio de la partición (UNBOUNDED PRECEDING) hasta la fila actual. Su consulta debería ser similar a la siguiente.

```
USE TSQLV6ES;
SELECT clienteid, ordenid, fechaorden, val,
       SUM(val) OVER(PARTITION BY clienteid
                      ORDER BY fechaorden, ordenid
                     ROWS BETWEEN UNBOUNDED PRECEDING
                           AND CURRENT ROW) AS totalacumulado
FROM Ventas.ValoresOrden;
```

clienteid	ordenid	fechaorden	val	totalacumulado
1	10643	2021-08-25	814.50	814.50
1	10692	2021-10-03	878.00	1692.50
1	10702	2021-10-13	330.00	2022.50
1	10835	2022-01-15	845.80	2868.30
1	10952	2022-03-16	471.20	3339.50
1	11011	2022-04-09	933.50	4273.00
2	10308	2020-09-18	88.80	88.80
2	10625	2021-08-08	479.75	568.55
2	10759	2021-11-28	320.00	888.55
2	10926	2022-03-04	514.40	1402.95
...				
91	10374	2020-12-05	459.00	459.00
91	10611	2021-07-25	808.00	1267.00
91	10792	2021-12-23	399.85	1666.85
91	10870	2022-02-04	160.00	1826.85
91	10906	2022-02-25	427.50	2254.35
91	10998	2022-04-03	686.00	2940.35
91	11044	2022-04-23	591.60	3531.95

Observe cómo los valores se van acumulando desde el principio de la partición de cliente hasta la fila actual. Por cierto, en vez de la forma detallada de la extensión marco ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, puede utilizar la forma más corta ROWS UNBOUNDED PRECEDING, y mantener el mismo significado.

Usando las funciones agregadas de ventana, para realizar cálculos como totales acumulados, normalmente obtiene mucho mejor rendimiento en comparación con el uso de joins o subconsultas y funciones agregadas de grupo. Las funciones de ventana nos dan una buena optimización, especialmente cuando se utiliza UNBOUNDED PRECEDING como el primer delimitador.



En términos del procesamiento de consulta lógico, el resultado de una consulta es logrado cuando lo obtiene de la fase SELECT, después de que las fases FROM, WHERE, GROUP BY y HAVING han sido procesadas. Debido a que supuestamente las funciones de ventana operan, sobre el conjunto resultado de la consulta originaria, son permitidos sólo en las cláusulas SELECT y ORDER BY. Si necesita referirse al resultado de una función de ventana en cualquier cláusula, que es evaluada antes de la cláusula SELECT, necesita utilizar una expresión de tabla. Se invoca la función de ventana en la cláusula SELECT de la consulta interna, asignando la expresión con un alias de columna. Luego, puede referirse a ese alias de columna en la consulta externa en todas las cláusulas.

Por ejemplo, suponga que necesita filtrar el resultado de la última consulta, retornando sólo esas filas donde el total acumulado es inferior a 1.000. El siguiente código logra esto definiendo una expresión de tabla común (CTE), basada en la consulta anterior y luego haciendo el filtrado en la consulta externa.

```
USE TSQLV6ES;
WITH TotalesAcumulados AS
(
    SELECT clienteid, ordenid, fechaorden, val,
           SUM(val) OVER(PARTITION BY clienteid
                          ORDER BY fechaorden, ordenid
                          ROWS BETWEEN UNBOUNDED PRECEDING
                                 AND CURRENT ROW) AS totalacumulado
      FROM Ventas.ValoresOrden
)
SELECT *
  FROM TotalesAcumulados
 WHERE totalacumulado < 1000;
```

clienteid	ordenid	fechaorden	val	totalacumulado
1	10643	2021-08-25	814.50	814.50
2	10308	2020-09-18	88.80	88.80
2	10625	2021-08-08	479.75	568.55
2	10759	2021-11-28	320.00	888.55
3	10365	2020-11-27	403.20	403.20
...				
87	10266	2020-07-26	346.56	346.56
88	10256	2020-07-15	517.80	517.80
89	10269	2020-07-31	642.20	642.20
90	10615	2021-07-30	120.00	120.00
90	10673	2021-09-18	412.35	532.35
91	10374	2020-12-05	459.00	459.00

Como otro ejemplo de una extensión marco de ventana, si quiere que el marco incluya sólo las tres últimas filas, debería utilizar la forma ROWS BETWEEN 2 PRECEDING AND CURRENT ROW.

FUNCIONES RANKING DE VENTANA

Con funciones ranking de ventana, puede rankear las filas en una partición basada en un ordenamiento especificado. Al igual que con las otras funciones de ventana, si no indica una cláusula de partición de ventana, el resultado de consulta originario completo es considerado una partición. La cláusula de orden de ventana es obligatoria. Las funciones ranking de ventana no soportan una cláusula marco de ventana. T-SQL soporta cuatro funciones ranking de ventana: ROW_NUMBER, RANK, DENSE_RANK y NTILE.

La consulta siguiente demuestra el uso de estas funciones.



```
USE TSQLV6ES;
SELECT clienteid, ordenid, val,
       ROW_NUMBER() OVER(ORDER BY val) AS rounum,
       RANK() OVER(ORDER BY val) AS rnk,
       DENSE_RANK() OVER(ORDER BY val) AS densernk,
       NTILE(100) OVER(ORDER BY val) AS ntile100
FROM Ventas.ValoresOrden;
```

clienteid	ordenid	val	rounum	rnk	densernk	ntile100
12	10782	12.50	1	1	1	1
27	10807	18.40	2	2	2	1
66	10586	23.80	3	3	3	1
76	10767	28.00	4	4	4	1
54	10898	30.00	5	5	5	1
88	10900	33.75	6	6	6	1
48	10883	36.00	7	7	7	1
41	11051	36.00	8	7	7	1
71	10815	40.00	9	9	8	1
38	10674	45.00	10	10	9	2
53	11057	45.00	11	10	9	2
75	10271	48.00	12	12	10	2
...						
63	10540	10191.70	822	822	787	99
65	10479	10495.60	823	823	788	100
37	10897	10835.24	824	824	789	100
39	10817	10952.85	825	825	790	100
73	10417	11188.40	826	826	791	100
65	10889	11380.00	827	827	792	100
71	11030	12615.05	828	828	793	100
34	10981	15810.00	829	829	794	100
63	10865	16387.50	830	830	795	100

Importante: Orden de Presentación versus Orden de Ventana

La consulta de muestra no tiene una cláusula ORDER BY de presentación, y, por lo tanto, no hay garantía de que las filas serán presentadas en algún orden en particular. La cláusula de orden de ventana sólo determina el ordenamiento para el cálculo de función de ventana. Si se invoca una función de ventana en su consulta, pero no especifica una cláusula ORDER BY de presentación, no hay garantía de que las filas serán presentadas en el mismo orden como el orden de función de ventana. Si necesita una garantía, necesita agregar una cláusula ORDER BY de presentación.

La función ROW_NUMBER calcula un entero secuencial único a partir de 1 en la partición de ventana basado en el orden de ventana. Debido a que la consulta de ejemplo no tiene una cláusula de partición de ventana, la función considera el conjunto resultado de consulta completa, como una partición; por lo tanto, la función asigna números de fila únicos a través del conjunto de resultados de consulta completo.

Note que, si el ordenamiento no es único, la función ROW_NUMBER no es determinística. Por ejemplo, note en el resultado, que dos filas tienen el mismo valor de ordenamiento de 36.00, pero las dos filas obtienen diferentes números de fila. Eso es porque la función debe generar enteros únicos, en la partición. Actualmente, no hay un desempate explícito, y por lo tanto la elección de que fila obtiene el número de fila más alto es arbitraria (dependiente de la optimización). Si necesita un cálculo determinístico (resultados repetibles garantizados), necesita agregar un desempate. Por ejemplo, puede agregar la llave primaria para hacer el ordenamiento único, como en ORDER BY val, ordenid.

RANK y DENSE_RANK difieren de ROW_NUMBER en el sentido que asigna el mismo valor rango a todas las filas que comparten el mismo valor de ordenamiento. La función RANK retorna el número de filas en la partición que tiene un valor de ordenamiento menor que el actual, más 1. Por ejemplo, considere las filas en el resultado de consulta de ejemplo que tiene un valor de ordenamiento de 45.00. Nueve filas tienen valores de ordenamiento que son menores que 45.00; por lo tanto, estas filas obtuvieron el rango de 10 (9 + 1).

La función DENSE_RANK retorna el número de valores de ordenamiento distintos que son más bajos que el actual, más 1. Por ejemplo, las mismas filas que obtuvieron el rango de 10 obtuvieron el rango denso de 9. Eso es porque



estas filas tienen un valor de ordenamiento de 45.00, y hay ocho valores de ordenamiento distintos que son inferiores que 45.00. Debido a que RANK considera filas y DENSE_RANK considera valores distintos, el primero puede tener brechas entre los valores ranking de resultado, y el último no puede tener brechas. Debido a que las funciones RANK y DENSE_RANK calculan el mismo valor ranking a las filas con el mismo valor de ordenamiento, ambas funciones son determinísticas, incluso cuando el ordenamiento no es único. De hecho, si utiliza el ordenamiento único, ambas funciones retornan el mismo resultado como la función ROW_NUMBER. Así que usualmente estas funciones son interesantes de utilizar cuando el ordenamiento no es único.

Con la función NTILE, puede organizar las filas en la partición, en un número solicitado de fichas de igual tamaño, basado en el orden especificado. Se especifica el número deseado de fichas como entrada a la función. En la consulta de ejemplo, solicitó 100 fichas. Hay 830 filas en el conjunto resultado, y por lo tanto el tamaño de ficha base es $830/100 = 8$ con un resto de 30. Debido a que hay un resto de 30, las primeras 30 fichas son asignados con una fila adicional.

A saber, las fichas 1 al 30 tendrán nueve filas cada una, y todas las fichas restantes (31 a 100) tendrán ocho filas cada una. Observe en el resultado de esta consulta de muestra que las primeras nueve filas (de acuerdo al ordenamiento val) son asignados con la ficha número 1, entonces las siguientes nueve filas son asignadas con la ficha número 2, y así sucesivamente. Como ROW_NUMBER, la función NTILE no es determinística cuando el ordenamiento no es único. Si necesita garantizar el determinismo, necesita definir el ordenamiento único.

Tema Clave

Como se explica en la discusión de las funciones agregadas de ventana, las funciones de ventana son permitidas sólo en las cláusulas SELECT y ORDER BY de la consulta. Si necesita referirse a ellas en otras cláusulas, por ejemplo, en la cláusula WHERE, necesita usar una expresión de tabla, tal como un CTE. Se invoca la función de ventana en la cláusula SELECT de la consulta interna, asignando la expresión con un alias de columna. Luego se refiere a ese alias de columna en la consulta WHERE de la consulta externa.

FUNCIONES OFFSET DE VENTANA

Las funciones offset de ventana retornan un elemento de una sola fila que está en un offset dado de la fila actual en la partición de ventana, o de la primera o la última fila en el marco de ventana. T-SQL soporta las siguientes funciones offset de ventana: LAG, LEAD, FIRST_VALUE y LAST_VALUE. Las funciones LAG y LEAD confían en un offset con respecto a la fila actual, y las funciones FIRST_VALUE y LAST_VALUE operan en la primera o la última fila en el marco, respectivamente.

Las funciones LAG y LEAD soportan cláusulas de partición y de ordenamiento de ventana. No soportan una cláusula marco de ventana. La función LAG retorna un elemento de la fila en la partición actual, que es un número solicitado de filas antes de la fila actual (basado en la ordenación de ventana), con 1 asumido como el offset predeterminado. La función LEAD retorna un elemento de la fila que está en el offset solicitado después de la fila actual.

A modo de ejemplo, la siguiente consulta utiliza las funciones LAG y LEAD para retornar junto con cada pedido, el valor del pedido del cliente anterior, además del valor del pedido del cliente posterior.

```
USE TSQLV6ES;
SELECT clienteid, ordenid, fechaorden, val,
LAG(val) OVER(PARTITION BY clienteid
               ORDER BY fechaorden, ordenid) AS prev_val,
LEAD(val) OVER(PARTITION BY clienteid
               ORDER BY fechaorden, ordenid) AS sigu_val
FROM Ventas.ValoresOrden;
```



clienteid	ordenid	fechaorden	val	prev_val	sigu_val
1	10643	2021-08-25	814.50	NULL	878.00
1	10692	2021-10-03	878.00	814.50	330.00
1	10702	2021-10-13	330.00	878.00	845.80
1	10835	2022-01-15	845.80	330.00	471.20
1	10952	2022-03-16	471.20	845.80	933.50
1	11011	2022-04-08	933.50	471.20	NULL
2	10308	2020-09-18	88.80	NULL	479.75
2	10625	2021-08-08	479.75	88.80	320.00
2	10759	2021-11-28	320.00	479.75	514.40
2	10926	2022-03-04	514.40	320.00	NULL
...					

Debido a que un offset explícito no fue especificado, ambas funciones confían en el offset predeterminado de 1. Si quiere un offset diferente de 1, se le especifica como el segundo argumento, como en LAG(val, 3). Note que, si una fila no existe en el offset solicitado, la función retorna un NULL por defecto. Si desea retornar un valor diferente en tal caso, especifíquelo como el tercer argumento, como en LAG(val, 3, 0).

Las funciones FIRST_VALUE y LAST_VALUE retornan una expresión de valor de la primera o la última fila en el marco de ventana, respectivamente. Naturalmente, las funciones soportan cláusulas de partición de ventana, de orden y de marco.

En SQL Server 2022 se incorporaron las opciones IGNORE NULLS y RESPECT NULLS.

IGNORE NULLS: se omiten los valores NULL del conjunto de datos al calcular el primer o último valor en una partición.

RESPECT NULLS: se respetan los valores NULL del conjunto de datos al calcular el primer o último valor en una partición.

Las sintaxis quedan:

```
FIRST_VALUE ( [expresion_escalar] ) [ IGNORE NULLS | RESPECT NULLS ]  
OVER ( [ clausula_partition ] clausula_order_by [ clausula_rango_filas ] )  
  
LAST_VALUE ( [expresion_escalar] ) [ IGNORE NULLS | RESPECT NULLS ]  
OVER ( [ clausula_partition ] clausula_order_by [ clausula_rango_filas ] )
```

A modo de ejemplo, la siguiente consulta retorna junto con cada pedido los valores del primer y último pedido del cliente.

```
USE TSQLV6ES;  
SELECT clienteid, ordenid, fechaorden, val,  
FIRST_VALUE(val) OVER(PARTITION BY clienteid  
                      ORDER BY fechaorden, ordenid  
                      ROWS BETWEEN UNBOUNDED PRECEDING  
                      AND CURRENT ROW) AS primer_val,  
LAST_VALUE(val) OVER(PARTITION BY clienteid  
                      ORDER BY fechaorden, ordenid  
                      ROWS BETWEEN CURRENT ROW  
                      AND UNBOUNDED FOLLOWING) AS ultimo_val  
FROM Ventas.ValoresOrden;
```



clienteid	ordenid	fechaorden	val	primer_val	ultimo_val
1	11811	2022-04-05	933.50	814.50	933.50
1	10952	2022-03-16	471.20	814.50	933.50
1	10835	2022-01-15	845.80	814.50	933.50
1	10702	2021-10-13	330.00	814.50	933.50
1	10692	2021-10-03	878.00	814.50	933.50
1	10643	2021-08-25	814.50	814.50	933.50
2	10926	2022-03-04	514.40	88.80	514.40
2	10759	2021-11-28	320.00	88.80	514.40
2	10625	2021-08-08	479.75	88.80	514.40
2	10308	2020-09-18	88.80	88.80	514.40
...					

SELECT – WINDOW

Como pudimos ver en algunos ejemplos anteriores, en casos en que tenemos expresiones que utilizan las mismas ventanas, tenemos que escribir el código de la ventana cada vez, con las consabidas contras que esto genera.

En SQL Server 2022 se implementó la cláusula WINDOW, que permite definir la ventana una sola vez y reutilizarla. Algo parecido a lo vimos con las Tablas derivadas y las CTE.

La definición de la ventana con nombre en la cláusula WINDOW determina las particiones y el orden de un conjunto de filas antes de que se aplique la función de ventana que usa la ventana en la cláusula OVER.

```
WINDOW nombre_ventana AS (
    [ nombre_de_ventana_referenciada ]
    [ <cláusula PARTITION BY> ]
    [ <cláusula ORDER BY> ]
    [ <cláusula ROW o RANGE> ]
)

<cláusula PARTITION BY> ::= 
PARTITION BY expresion , ... [ n ]

<cláusula ORDER BY> ::= 
ORDER BY expresion_orden
[ COLLATE nombre_collation ]
[ ASC | DESC ]
[ ,...n ]

<cláusula ROW o RANGE> ::= 
{ ROWS | RANGE } <extensión del marco de ventana>
```

En el siguiente ejemplo podemos ver que la misma ventana está definida tres veces en la consulta:

```
SELECT ordenid, productoid, cantidad
    ,SUM(cantidad) OVER (PARTITION BY ordenid ORDER BY ordenid, productoid) AS Total
    ,AVG(cantidad) OVER (PARTITION BY ordenid ORDER BY ordenid, productoid) AS Promedio
    ,COUNT(cantidad) OVER (PARTITION BY ordenid ORDER BY ordenid, productoid) AS Cantidad
FROM Ventas.DetallesOrden
WHERE ordenid IN(10847,10979);
```

Mediante la cláusula WINDOW se puede definir una única vez y reutilizarla mediante el nombre que se le ha dado:

```
SELECT ordenid, productoid, cantidad
    ,SUM(cantidad) OVER ven1 AS Total
    ,AVG(cantidad) OVER ven1 AS Promedio
    ,COUNT(cantidad) OVER ven1 AS Cantidad
FROM Ventas.DetallesOrden
WHERE ordenid IN(10847,10979)
WINDOW ven1 AS (PARTITION BY ordenid ORDER BY ordenid, productoid);
```



PIVOTEANDO DATOS

Pivoteando datos consiste en rotar datos que están en forma de filas, a columnas, posiblemente realizando algún tipo de agregación en el proceso. En muchos casos el pivoteo de datos es manejado en la capa de presentación, para realizar reportes, por ejemplo, pero ahora vamos a ver cómo podemos hacerlo en la base de datos.

Para realizar los ejemplos, vamos a crear una tabla de ejemplo (dbo.Ordenes) y agregarle algunos datos.

```
USE TSQLV6ES;
DROP TABLE IF EXISTS dbo.Ordenes;
CREATE TABLE dbo.Ordenes
(
ordenid INT NOT NULL,
fechaorden DATE NOT NULL,
empid INT NOT NULL,
clienteid VARCHAR(5) NOT NULL,
cantidad INT NOT NULL,
CONSTRAINT PK_Orders PRIMARY KEY(ordenid)
);
INSERT INTO dbo.Ordenes(ordenid, fechaorden, empid, clienteid, cantidad) VALUES
(30001, '20200802', 3, 'A', 10),
(10001, '20201224', 2, 'A', 12),
(10005, '20201224', 1, 'B', 20),
(40001, '20210109', 2, 'A', 40),
(10006, '20210118', 1, 'C', 14),
(20001, '20210212', 2, 'B', 12),
(40005, '20220212', 3, 'A', 10),
(20002, '20220216', 1, 'C', 20),
(30003, '20220418', 2, 'B', 15),
(30004, '20200418', 3, 'C', 22),
(30007, '20220907', 3, 'D', 30);

SELECT * FROM dbo.Ordenes;
```

ordenid	fechaorden	empid	clienteid	cantidad
10001	2020-12-24	2	A	12
10005	2020-12-24	1	B	20
10006	2021-01-18	1	C	14
20001	2021-02-12	2	B	12
20002	2022-02-16	1	C	20
30001	2020-08-02	3	A	10
30003	2022-04-18	2	B	15
30004	2020-04-18	3	C	22
30007	2022-09-07	3	D	30
40001	2021-01-09	2	A	40
40005	2022-02-12	3	A	10

Podemos obtener el total por cada empleado y cliente, con la siguiente consulta:

```
SELECT empid, clienteid, SUM(cantidad) AS sumqty
FROM dbo.Ordenes
GROUP BY empid, clienteid;
```

empid	clienteid	sumqty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30

Pero los datos necesitamos verlos en el siguiente formato:



empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

Lo que vemos es una tabla pivoteada con agregación. Cada consulta de pivoteo involucra tres etapas lógicas de procesamiento.

- 1) Una fase de agrupamiento sobre los elementos de filas
- 2) Una fase de propagación sobre los elementos de columnas
- 3) Una fase de agregación, con una función de agregación asociada.

En este ejemplo, necesitamos generar una fila única, para cada empleado, esto significa que las filas de la tabla dbo.Ordenes tienen que ser agrupadas por empid.

La tabla dbo.Ordenes tiene una columna única que contiene todos los clienteid con sus cantidades ordenadas. La tarea de pivoteo requiere generar una columna diferente para cada id de cliente, que contenga la suma de las cantidades para cada uno. Este es el proceso de propagación de cantidades por clienteid.

Por último, se deben “intersectar” los valores entre cada empleado y cliente, utilizando la función de agrupamiento necesaria, en este caso SUM.

En resumen, el pivoteo implica agrupar, propagar y agregar. En este ejemplo, agrupa por empid, propaga (cantidades) por custid y agrega con SUM (cantidad). Después de identificar los elementos implicados en el pivoteo, el resto es sólo una cuestión de incorporar esos elementos en los lugares correctos en una plantilla de consulta genérica para pivotar.

PIVOTEANDO CON UNA CONSULTA AGRUPADA

La solución que utiliza una consulta agrupada maneja las tres fases de una manera explícita y directa.

La fase de agrupamiento se realiza mediante GROUP BY, en este caso GROUP BY empid.

La fase de propagación se realiza en el SELECT, mediante expresiones CASE para cada columna resultante, en este caso para cada cliente se deberá utilizar una expresión como: CASE WHEN clienteid = 'A' THEN cantidad END.

Por último, la fase de agregación se realiza aplicando la función de agregación correspondiente, SUM en este caso, al resultado de cada expresión CASE, quedando la expresión del cliente A como: SUM(CASE WHEN clienteid = 'A' THEN cantidad END) AS A

La consulta final queda de la siguiente manera:

```
SELECT empid,
SUM(CASE WHEN clienteid = 'A' THEN cantidad END) AS A,
SUM(CASE WHEN clienteid = 'B' THEN cantidad END) AS B,
SUM(CASE WHEN clienteid = 'C' THEN cantidad END) AS C,
SUM(CASE WHEN clienteid = 'D' THEN cantidad END) AS D
FROM dbo.Ordenes
GROUP BY empid;
```

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30



PIVOTEANDO CON EL OPERADOR PIVOT

El operador PIVOT no es estándar, es una solución propietaria de T-SQL, que permite pivotear de una manera más concisa.

Es un operador de tabla, que opera en el contexto del FROM. Opera sobre la tabla de origen a su izquierda, pivotea los datos, y devuelve una tabla. El operador PIVOT realiza los mismos pasos lógicos descriptos anteriormente (agrupamiento, propagación y agregación), solo que requiere menos código que la solución anterior.

La forma general es la siguiente:

```
SELECT ...
FROM <tabla_origen>
PIVOT(<función_agregacion>(<elemento_agregación>)
FOR <elemento_propagacion> IN (<lista_columnas_objetivo>)) AS
<alias_tabla_resultado>
WHERE ...;
```

Entre los paréntesis del operador PIVOT, se especifica la función de agregación, SUM en este caso, el elemento de agregación (cantidad), el elemento de propagación (clienteid), y la lista de columnas objetivo (A, B, C, D). Luego de los paréntesis del operador PIVOT, se especifica un alias para la tabla resultante.

En el operador PIVOT no se especifica explícitamente los elementos de agrupamiento, por lo que no es necesario utilizar GROUP BY en esta consulta. El operador PIVOT determina los elementos de agrupamiento implícitamente, por eliminación. Los elementos de agrupamiento son todos los atributos de la tabla origen que no fueron especificados como elementos de propagación ni como elementos de agregación. Por lo tanto, debemos asegurarnos que la tabla de origen sólo tenga los atributos de agrupamiento, propagación y agregación, y ningún otro. Para esto podemos utilizar una expresión de tabla.

```
SELECT empid, A, B, C, D
FROM (SELECT empid, clienteid, cantidad
      FROM dbo.Ordenes) AS D
PIVOT(SUM(cantidad) FOR clienteid IN(A, B, C, D)) AS P;
```

En lugar de operar directamente sobre la tabla dbo.Ordenes, el operador PIVOT trabaja sobre la tabla derivada D, que incluye sólo los elementos empid, clienteid y cantidad.

¿Qué pasa si realizamos la consulta directamente sobre dbo.Ordenes?

```
SELECT empid, A, B, C, D
FROM dbo.Ordenes
PIVOT(SUM(cantidad) FOR clienteid IN(A, B, C, D)) AS P;
```

empid	A	B	C	D
2	12	NULL	NULL	NULL
1	NULL	20	NULL	NULL
1	NULL	NULL	14	NULL
2	NULL	12	NULL	NULL
1	NULL	NULL	20	NULL
3	10	NULL	NULL	NULL
2	NULL	15	NULL	NULL
3	NULL	NULL	22	NULL
3	NULL	NULL	NULL	30
2	40	NULL	NULL	NULL
3	10	NULL	NULL	NULL

Como la tabla contiene más elementos, el operador PIVOT toma como atributos de agrupamiento a ordenid, fechaorden y empid.



Es como si en el enfoque original, hubiéramos utilizado la siguiente consulta:

```
SELECT empid,
SUM(CASE WHEN clienteid = 'A' THEN cantidad END) AS A,
SUM(CASE WHEN clienteid = 'B' THEN cantidad END) AS B,
SUM(CASE WHEN clienteid = 'C' THEN cantidad END) AS C,
SUM(CASE WHEN clienteid = 'D' THEN cantidad END) AS D
FROM dbo.Ordenes
GROUP BY ordenid, fechaorden, empid;
```

Como una buena práctica para el operador PIVOT, es conveniente utilizar siempre una expresión de tabla, incluso cuando la tabla de origen tiene sólo los atributos necesarios, ya que, en el futuro, se podrían agregar campos a la tabla.

Si ahora necesitamos mostrar los empleados como filas y los clientes como columnas, el elemento de agrupamiento ahora sería clienteid, y el de propagación empid.

```
SELECT clienteid, [1], [2], [3]
FROM (SELECT empid, clienteid, cantidad
      FROM dbo.Ordenes) AS D
PIVOT(SUM(cantidad) FOR empid IN([1], [2], [3])) AS P;
clienteid 1          2          3
-----
A        NULL       52       20
B        20        27       NULL
C        34        NULL      22
D        NULL       NULL     30
```

Recuerden que cuando los identificadores son irregulares, por ejemplo, cuando empiezan con dígitos, deben ser delimitados, por eso utilizamos los corchetes.

DESPIVOTEANDO DATOS

El despivoteo es una técnica que rota los datos de un estado de columnas a un estado de filas. Por lo general, implica consultar un estado pivotante de los datos y producir a partir de cada fila de origen varias filas de resultados, cada una con un valor de columna de origen diferente. Un caso de uso común es desvincular los datos que importó de una hoja de cálculo a la base de datos para facilitar la manipulación.

Para realizar los ejemplos, vamos a crear una tabla de ejemplo (dbo.EmpClieOrdenes) y agregarle algunos datos.

```
USE TSQLV6ES;

DROP TABLE IF EXISTS dbo.EmpClieOrdenes;

CREATE TABLE dbo.EmpClieOrdenes
(
empid INT NOT NULL
CONSTRAINT PK_EmpClieOrdenes PRIMARY KEY,
A VARCHAR(5) NULL,
B VARCHAR(5) NULL,
C VARCHAR(5) NULL,
D VARCHAR(5) NULL
);

INSERT INTO dbo.EmpClieOrdenes(empid, A, B, C, D)
SELECT empid, A, B, C, D
FROM (SELECT empid, clienteid, cantidad
      FROM dbo.Ordenes) AS D
PIVOT(SUM(cantidad) FOR clienteid IN(A, B, C, D)) AS P;

SELECT * FROM dbo.EmpClieOrdenes;
```



empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

La tabla tiene una fila por cada empleado, y una columna por cada cliente A, B, C y D, y la cantidad pedida por cada empleado y cliente. Podemos ver que las intersecciones irrelevantes (combinaciones de empleado-cliente que no tienen actividad) son representadas por NULLs. Supongamos que tenemos el pedido de despivotear la información, requiriendo devolver una fila para cada empleado y cliente, junto con la cantidad solicitada. El resultado a obtener debería ser algo como esto:

empid	clienteid	cantidad
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

DESPIVOTEANDO CON EL OPERADOR APPLY

Depivotear implica tres fases de procesamiento lógico: producir copias, extraer valores y eliminar filas irrelevantes.

El primer paso de la solución consiste en producir varias copias de cada fila de origen, una para cada columna que necesite anular despivotear. En este caso, debemos producir una copia para cada una de las columnas A, B, C y D, que representan los ID de cliente. Podemos lograr este paso aplicando un CROSS JOIN entre la tabla EmpClieOrdenes y una tabla que tiene una fila para cada cliente.

Si ya tenemos una tabla de clientes en la base de datos, podemos usar esa tabla en el CROSS JOIN. Si no tenemos una tabla de clientes, podemos crear una virtual sobre la marcha utilizando un constructor de valores de tabla basado en la cláusula VALUES. Aquí está el código que implementa este paso:

```
SELECT *
FROM dbo.EmpClieOrdenes
CROSS JOIN (VALUES('A'),('B'),('C'),('D')) AS C(clienteid);
```

empid	A	B	C	D	clienteid
1	NULL	20	34	NULL	A
1	NULL	20	34	NULL	B
1	NULL	20	34	NULL	C
1	NULL	20	34	NULL	D
2	52	27	NULL	NULL	A
2	52	27	NULL	NULL	B
2	52	27	NULL	NULL	C
2	52	27	NULL	NULL	D
3	20	NULL	22	30	A
3	20	NULL	22	30	B
3	20	NULL	22	30	C
3	20	NULL	22	30	D

La cláusula VALUES define un conjunto de cuatro filas, cada una con un único valor de ID de cliente. El código define una tabla derivada llamada C basada en esta cláusula y nombra la única columna en ella clienteid. Luego, el código aplica un CROSS JOIN entre EmpClieOrdenes y C.



Como puede ver, se produjeron cuatro copias para cada fila de origen, una para los clientes A, B, C y D.

El segundo paso en la solución es extraer un valor de una de las columnas de cantidad del cliente original (A, B, C o D) para devolver una columna de valor único (llamémosla cantidad en nuestro caso). Debemos extraer el valor de la columna que corresponde al valor clienteid actual. Si clienteid es "A", la columna cantidad debe devolver el valor de la columna A, si clienteid es "B", cantidad debe devolver el valor de la columna B, y así sucesivamente. Para lograr este paso, podríamos pensar que simplemente podemos agregar la columna cantidad como una segunda columna a cada fila en el constructor de valores de la tabla (la cláusula VALUES), así:

```
SELECT empid, clienteid, cantidad
FROM dbo.EmpClieOrdenes
CROSS JOIN (VALUES ('A', A), ('B', B), ('C', C), ('D', D)) AS C(clienteid, cantidad);
```

Sin embargo, recordemos que un JOIN trata sus dos entradas como un conjunto; por lo tanto, no hay orden entre esas entradas. No podemos hacer referencia a los elementos de ninguna de las entradas al construir la otra. En nuestro caso, el constructor de valores de tabla en el lado derecho del JOIN tiene referencias a las columnas A, B, C y D del lado izquierdo del JOIN (EmpClieOrdenes). En consecuencia, cuando intentamos ejecutar este código, obtenemos los siguientes errores:

```
Msg 207, Level 16, State 1, Line 35
Invalid column name 'A'.
Msg 207, Level 16, State 1, Line 35
Invalid column name 'B'.
Msg 207, Level 16, State 1, Line 35
Invalid column name 'C'.
Msg 207, Level 16, State 1, Line 35
Invalid column name 'D'.
```

La solución es utilizar el operador CROSS APPLY en lugar del operador CROSS JOIN.

Son similares, pero el primero evalúa primero el lado izquierdo y luego aplica el lado derecho a cada fila izquierda, haciendo que los elementos del lado izquierdo sean accesibles al lado derecho. Aquí está el código que implementa este paso con el operador CROSS APPLY:

```
SELECT empid, clienteid, cantidad
FROM dbo.EmpClieOrdenes
CROSS APPLY (VALUES('A', A), ('B', B), ('C', C), ('D', D)) AS C(clienteid, cantidad);

empid      clienteid  cantidad
-----  -----  -----
1          A          NULL
1          B          20
1          C          34
1          D          NULL
2          A          52
2          B          27
2          C          NULL
2          D          NULL
3          A          20
3          B          NULL
3          C          22
3          D          30
```

Recordemos que, en la tabla original, los NULL representan intersecciones irrelevantes. Como resultado, normalmente no hay razón para mantener filas irrelevantes donde cantidad es NULL. Lo bueno en nuestro caso es que el operador CROSS APPLY, que creó la columna cantidad, se procesó en la cláusula FROM, y la cláusula FROM se evalúa antes que la cláusula WHERE. Esto significa que la columna cantidad es accesible para expresiones en



la cláusula WHERE. Para eliminar filas irrelevantes, agregamos un filtro en la cláusula WHERE que descarte filas con un NULL en la columna cantidad, así:

```
SELECT empid, clienteid, cantidad
FROM dbo.EmpClieOrdenes
CROSS APPLY (VALUES('A', A), ('B', B), ('C', C), ('D', D)) AS C(clienteid, cantidad)
WHERE cantidad IS NOT NULL;
```

empid	clienteid	cantidad
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

DESPIVOTEANDO CON EL OPERADOR UNPIVOT

Despivotear los datos implica producir dos columnas de resultados a partir de cualquier número de columnas de origen: una para contener los nombres de las columnas de origen como cadenas y otra para contener los valores de la columna de origen. En este ejemplo, debemos anular la división de las columnas de origen A, B, C y D, lo que genera la columna de nombre clienteid de resultado y la columna cantidad de valores. Similar al operador PIVOT, T-SQL también admite el operador UNPIVOT para permitirle despivotear datos. La forma general de una consulta con el operador UNPIVOT es la siguiente:

```
SELECT ...
FROM <input_table>
UNPIVOT(<values_column> FOR <names_column> IN(<source_columns>)) AS
<result_table_alias>
WHERE ...;
```

Al igual que el operador PIVOT, UNPIVOT también se implementó como operador de tabla en el contexto de la cláusula FROM. Opera en una tabla fuente o expresión de tabla (EmpClieOrdenes en este caso). Dentro de los paréntesis del operador UNPIVOT, especificamos el nombre que deseamos asignar a la columna que contendrá los valores de la columna fuente (cantidad aquí), el nombre que deseamos asignar a la columna que contendrá los nombres de la columna fuente (clienteid), y la lista de nombres de columna fuente (A, B, C y D). Después de los paréntesis, proporcionamos un alias a la tabla resultante del operador de tabla.

Aquí está la consulta que usa el operador UNPIVOT para manejar nuestra tarea de despivoteo:

```
SELECT empid, clienteid, cantidad
FROM dbo.EmpClieOrdenes
UNPIVOT(cantidad FOR clienteid IN(A, B, C, D)) AS U;
```

Tengamos en cuenta que el operador UNPIVOT implementa las mismas fases de procesamiento lógico descritas anteriormente: generar copias, extraer elementos y eliminar las intersecciones NULL.

Sin embargo, la última fase no es opcional como en la solución con el operador APPLY. Cuando necesitemos aplicar la tercera fase, es conveniente utilizar la solución con el operador UNPIVOT porque es más concisa. Cuando necesitemos mantener las filas con NULL, usaremos la solución con el operador APPLY.



CONJUNTOS DE AGRUPACIÓN

Esta sección describe qué es un conjunto de agrupación y las características de T-SQL que admiten conjuntos de agrupación.

Un conjunto de agrupación es un conjunto de expresiones por las que se agrupan los datos en una consulta agrupada (una consulta con una cláusula GROUP BY). La razón para usar el término "conjunto" aquí es que el orden en el que especificamos las expresiones en la cláusula GROUP BY no tiene importancia.

Tradicionalmente en SQL, una única consulta agrupada define un único conjunto de agrupaciones. Por ejemplo, cada una de las siguientes cuatro consultas define un solo conjunto de agrupaciones:

```
SELECT empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY empid, clienteid;

SELECT empid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY empid;

SELECT clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY clienteid;

SELECT SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes;
```

La primera consulta define el conjunto de agrupaciones (empid, clienteid); la segunda (empid), la tercera (clienteid) y la última consulta define lo que se conoce como el conjunto de agrupación vacío, (). Este código devuelve cuatro conjuntos de resultados, uno para cada una de las cuatro consultas.

Supongamos, para fines de informes, que en lugar de querer que se devuelvan cuatro conjuntos de resultados separados, deseamos un único conjunto de resultados unificado. Podemos lograr esto utilizando el operador UNION ALL entre las consultas, después de plantar NULL como marcadores de posición para las columnas que aparecen en una consulta, pero no en otras. Así es como se ve el código:

```
SELECT empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY empid, clienteid
UNION ALL
SELECT empid, NULL, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY empid
UNION ALL
SELECT NULL, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY clienteid
UNION ALL
SELECT NULL, NULL, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes;
```



empid	clienteid	sumcantidad
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30
1	NULL	54
2	NULL	79
3	NULL	72
NULL	A	72
NULL	B	47
NULL	C	56
NULL	D	30
NULL	NULL	205

Aunque logramos obtener lo que buscamos, esta solución tiene dos problemas principales: la longitud del código y el rendimiento. Es largo porque tiene una consulta independiente para cada conjunto de agrupaciones. Además, SQL Server necesita aplicar un escaneo separado de los datos para cada consulta.

T-SQL admite funciones estándar que podemos utilizar para definir varios conjuntos de agrupaciones en la misma consulta. Estas son las subcláusulas GROUPING SETS, CUBE y ROLLUP de la cláusula GROUP BY, y las funciones GROUPING y GROUPING_ID. Los principales casos de uso son informes y análisis de datos. Estas características generalmente necesitan que la capa de presentación utilice controles GUI más sofisticados para mostrar los datos que el control de cuadrícula típico con sus columnas y filas.

LA SUBCLÁUSULA GROUPING SETS

La subcláusula GROUPING SETS es una potente mejora de la cláusula GROUP BY. Podemos utilizarlo para definir varios conjuntos de agrupaciones en la misma consulta. Simplemente enumeramos los conjuntos de agrupación que deseamos, separados por comas dentro de los paréntesis de la subcláusula GROUPING SETS, y para cada conjunto de agrupaciones enumeramos los miembros, separados por comas, entre paréntesis. Por ejemplo, la siguiente consulta define cuatro conjuntos de agrupaciones: (empid, clienteid), (empid), (clienteid) y ():

```
SELECT empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY
GROUPING SETS
(
(empid, clienteid),
(empid),
(clienteid),
()
);
```

El último conjunto de agrupaciones es el conjunto de agrupaciones vacío que representa el total general. Esta consulta es un equivalente lógico de la solución anterior que unificó los conjuntos de resultados de cuatro consultas agregadas. Solo que este es mucho más corto y además se optimiza mejor. SQL Server normalmente necesita menos escaneos de datos que el número de conjuntos de agrupaciones porque puede acumular agregados internamente.



LA SUBCLÁUSULA CUBE

La subcláusula CUBE de la cláusula GROUP BY proporciona una forma abreviada de definir múltiples conjuntos de agrupaciones. En los paréntesis de la subcláusula CUBE, proporcionamos un conjunto de miembros separados por comas y obtenemos todos los conjuntos de agrupación posibles que se pueden definir en función de los miembros de entrada. Por ejemplo, CUBE (a, b, c) es equivalente a GROUPING SETS((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ()). En la teoría de conjuntos, el conjunto de todos los subconjuntos de elementos que se pueden producir a partir de un conjunto particular se denomina conjunto de potencias. Podemos pensar que la subcláusula CUBE produce el conjunto de potencias de conjuntos de agrupación que se pueden formar a partir del conjunto de elementos dado.

En lugar de usar la subcláusula GROUPING SETS en la consulta anterior para definir los cuatro conjuntos de agrupamiento (empid, clienteid), (empid), (clienteid) y (), simplemente podemos usar CUBE (empid, clienteid). Aquí está la consulta completa:

```
SELECT empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY CUBE(empid, clienteid);
```

LA SUBCLÁUSULA ROLLUP

La subcláusula ROLLUP de la cláusula GROUP BY también proporciona una forma abreviada de definir múltiples conjuntos de agrupaciones. Sin embargo, a diferencia de la subcláusula CUBE, ROLLUP no produce todos los conjuntos de agrupaciones posibles. ROLLUP asume una jerarquía entre los miembros de entrada y produce solo conjuntos de agrupación que forman combinaciones iniciales de los miembros de entrada. Por ejemplo, mientras que CUBE (a, b, c) produce los ocho posibles conjuntos de agrupaciones, ROLLUP (a, b, c) produce solo cuatro según la jerarquía a>b>c. Es el equivalente a especificar GROUPING SETS((a, b, c), (a, b), (a), ()) .

Por ejemplo, supongamos que deseamos devolver las cantidades totales para todos los conjuntos de agrupación que se pueden definir en función de la jerarquía de tiempo del año del pedido, el mes del pedido y el día del pedido. Puede utilizar la subcláusula GROUPING SETS y enumerar explícitamente los cuatro posibles conjuntos de agrupaciones:

```
GROUPING SETS(
(YEAR(fechaorden), MONTH(fechaorden), DAY(fechaorden)),
(YEAR(fechaorden), MONTH(fechaorden)),
(YEAR(fechaorden)),
())
```

El equivalente lógico que utiliza la subcláusula ROLLUP es mucho más conciso:

```
ROLLUP(YEAR(fechaorden), MONTH(fechaorden), DAY(fechaorden))
```

A continuación, está la consulta completa que necesitamos ejecutar:

```
SELECT
YEAR(fechaorden) AS anioorden,
MONTH(fechaorden) AS mesorden,
DAY(fechaorden) AS diaorden,
SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY ROLLUP(YEAR(fechaorden), MONTH(fechaorden), DAY(fechaorden));
```



anioorden	mesorden	diaorden	sumcantidad
2020	4	18	22
2020	4	NULL	22
2020	8	2	10
2020	8	NULL	10
2020	12	24	32
2020	12	NULL	32
2020	NULL	NULL	64
2021	1	9	40
2021	1	18	14
2021	1	NULL	54
2021	2	12	12
2021	2	NULL	12
2021	NULL	NULL	66
2022	2	12	10
2022	2	16	20
2022	2	NULL	30
2022	4	18	15
2022	4	NULL	15
2022	9	7	30
2022	9	NULL	30
2022	NULL	NULL	75
NULL	NULL	NULL	205

LAS FUNCIONES GROUPING Y GROUPING_ID

Cuando tenemos una sola consulta que define varios conjuntos de agrupación, es posible que debamos asociar filas de resultados y conjuntos de agrupación. Siempre que todos los elementos de agrupación estén definidos como NOT NULL, esto es fácil. Por ejemplo, consideremos la siguiente consulta:

```
SELECT empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY CUBE(empid, clienteid);
```

empid	clienteid	sumcantidad
2	A	52
3	A	20
NULL	A	72
1	B	20
2	B	27
NULL	B	47
1	C	34
3	C	22
NULL	C	56
3	D	30
NULL	D	30
NULL	NULL	205
1	NULL	54
2	NULL	79
3	NULL	72

Debido a que las columnas empid y clienteid se definieron en la tabla dbo.Ordenes como NOT NULL, un NULL en esas columnas solo puede representar un marcador de posición, lo que indica que la columna no participó en el conjunto de agrupación actual. Por ejemplo, todas las filas en las que empid no es NULL y clienteid no es NULL están asociadas con el conjunto de agrupaciones (empid, clienteid). Todas las filas en las que empid no es NULL y clienteid es NULL están asociadas con el conjunto de agrupación (empid), y así sucesivamente.



Sin embargo, si una columna de agrupación permite NULL en la tabla, no puede saber con certeza si un NULL en el conjunto de resultados se originó a partir de los datos o es un marcador de posición para un miembro no participante en un conjunto de agrupación. Una forma de resolver este problema es utilizar la función GROUPING.

Esta función acepta el nombre de una columna y devuelve 0 si es un miembro del conjunto de agrupación actual (un elemento de detalle) y 1 en caso contrario (un elemento agregado).

```
SELECT
GROUPING(empid) AS grpemp,
GROUPING(clienteid) AS grpcli,
empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY CUBE(empid, clienteid);
```

grpemp	grpcli	empid	clienteid	sumcantidad
0	0	2	A	52
0	0	3	A	20
1	0	NULL	A	72
0	0	1	B	20
0	0	2	B	27
1	0	NULL	B	47
0	0	1	C	34
0	0	3	C	22
1	0	NULL	C	56
0	0	3	D	30
1	0	NULL	D	30
1	1	NULL	NULL	205
0	1	1	NULL	54
0	1	2	NULL	79
0	1	3	NULL	72

Ahora ya no necesitamos depender de los valores NULL para averiguar la asociación entre las filas de resultados y los conjuntos de agrupación. Por ejemplo, todas las filas en las que grpemp es 0 y grpcli es 0 están asociadas con el conjunto de agrupaciones (empid, clienteid). Todas las filas en las que grpemp es 0 y grpcli es 1 están asociadas con el conjunto de agrupación (empid), y así sucesivamente.

T-SQL admite otra función, denominada GROUPING_ID, que puede simplificar aún más el proceso de asociación de filas de resultados y conjuntos de agrupaciones. Proporcionamos a la función todos los elementos que participan en cualquier agrupación establecida como entradas, por ejemplo, GROUPING_ID(a, b, c, d), y la función devuelve un mapa de bits entero en el que cada bit representa un elemento de entrada diferente, el elemento más a la derecha representado por el bit más a la derecha. Por ejemplo, el conjunto de agrupaciones (a, b, c, d) está representado por el entero 0 ($0 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1$). El conjunto de agrupación (a, c) está representado por el número entero 5 ($0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$), y así sucesivamente.

En lugar de llamar a la función GROUPING para cada elemento de agrupación como en la consulta anterior, puede llamar a la función GROUPING_ID una vez y proporcionarle todos los elementos de agrupación como entrada, como se muestra aquí:

```
SELECT
GROUPING_ID(empid, clienteid) AS groupingset,
empid, clienteid, SUM(cantidad) AS sumcantidad
FROM dbo.Ordenes
GROUP BY CUBE(empid, clienteid);
```



groupingset	empid	clienteid	sumcantidad
0	2	A	52
0	3	A	20
2	NULL	A	72
0	1	B	20
0	2	B	27
2	NULL	B	47
0	1	C	34
0	3	C	22
2	NULL	C	56
0	3	D	30
2	NULL	D	30
3	NULL	NULL	205
1	1	NULL	54
1	2	NULL	79
1	3	NULL	72

Ahora podemos averiguar fácilmente con qué conjunto de agrupaciones está asociada cada fila. El entero 0 (binario 00) representa el conjunto de agrupaciones (empid, custid); el número entero 1 (binario 01) representa (empid); el número entero 2 (binario 10) representa (clienteid); y el número entero 3 (11 binario) representa () .

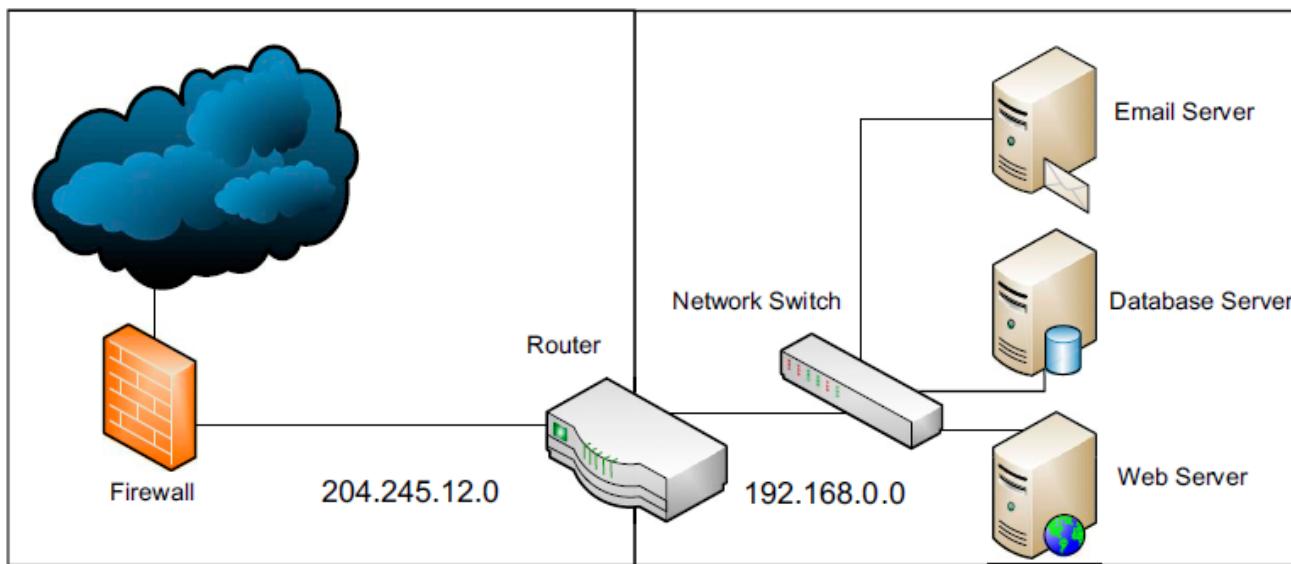


UNIDAD 5

SEGURIDAD DE LA INFORMACIÓN

ASEGURAR LA RED

Si bien está fuera del alcance de la materia, es fundamental tener en claro desde que puestos de la red se puede acceder a la base de datos.



Desde ya debemos contar con firewall que protejan el acceso desde Internet, pero también pueden ser necesarios firewalls en la red interna para proteger accesos indebidos a la base de datos.

También puede ser necesario configurar el firewall de Windows del servidor en donde está instalada la instancia de SQL Server.

SEGURIDAD FÍSICA

Muchas veces nos preocupamos en proteger el acceso a la base de datos a través de la red, y descuidamos el acceso físico:

- 1) Una persona no autorizada accede al centro de cómputos y accede directamente al servidor
- 2) Una persona no autorizada accede a la red de la compañía, generalmente vía WIFI y accede a los recursos de red, entre ellos el servidor de base de datos
- 3) Una persona no autorizada accede al puesto de un empleado (pc o notebook) desatendido y sin bloquear.

CIFRADO DE BASES DE DATOS

Una forma clave de proteger los datos dentro de su base de datos es usar el cifrado de la base de datos. Sin embargo, ninguna solución de cifrado es correcta para todas las bases de datos. Los requisitos de encriptación de la aplicación determinarán la solución de encriptación a seleccionar. Una cosa que debemos recordar acerca del cifrado de la base de datos es que cuantos más datos cifremos y más fuerte sea el cifrado, más potencia de CPU se necesitará para cifrar y descifrar los datos que se necesitan. Por lo tanto, es fundamental equilibrar los requisitos de cifrado con la mayor carga del sistema.

Existen dos técnicas para proteger los datos: hashing y cifrado. El cifrado se realiza mediante uno de varios algoritmos diferentes que dan un valor que se puede descifrar cuando se utiliza la clave de descifrado correcta.



Cada una de las diferentes opciones de cifrado proporciona una fuerza de cifrado diferente. A medida que usemos un nivel de cifrado más fuerte, se utilizará más carga de CPU en el servidor Microsoft SQL. Microsoft SQL Server sólo admite un subconjunto de los algoritmos de cifrado disponibles; sin embargo, admite algunos de los algoritmos más populares, desde el más débil al más fuerte, que son DES, TRIPLE_DES, TRIPLE_DES_3KEY, RC2, RC4, RC4_128, DESX, AES_128, AES_192 y AES_256. La lista completa de algoritmos disponibles no ha cambiado desde Microsoft SQL Server 2005. La lista le ofrece una variedad de opciones que brindarán una opción de cifrado para casi todos.

A partir de SQL Server 2016 (13.x), todos los algoritmos distintos de AES_128, AES_192 y AES_256 están en desuso. Para usar algoritmos más antiguos (no recomendado), debemos configurar la base de datos con un nivel de compatibilidad de base de datos 120 o inferior.

AES

Se pueden usar tres tamaños diferentes de cifradores con el algoritmo de Estándar de cifrado avanzado (AES). Estos cifrados pueden tener un tamaño de 128, 192 y 256 bits, representados por AES_128, AES_192 y AES_256, respectivamente, dentro de Microsoft SQL Server. Los tamaños de clave variable se utilizan para combinar datos que son bloques de 128 bits. Los atacantes han tenido cierto éxito al romper el algoritmo de cifrado AES cuando utilizan el cifrado AES de extremo inferior. Hasta la fecha, las versiones superiores de AES se han mantenido estables.

HASHING

Por otro lado, tenemos algoritmos de hash. Los algoritmos de hash proporcionan una técnica unidireccional que podemos usar para enmascarar datos, con la mínima posibilidad de que alguien pueda revertir el valor de hash al valor original. Y con las técnicas de hash, cada vez que *hasheamos* el valor original obtenemos el mismo valor de hash. Microsoft SQL Server admite los siguientes algoritmos de hash: MD2 | MD4 | MD5 | SHA | SHA1 | SHA2_256 | SHA2_512

Siempre que utilicemos el mismo algoritmo de hashing cada vez que hagamos hash de un valor, siempre obtendremos el mismo valor de hash. Por ejemplo, si usamos el algoritmo hash MD5 para aplicar al valor "Lujan Buen Viaje", siempre devolverá el valor de "0xB0560603BB56A009FAB2D0246803DAFA".

El hash se realiza, independientemente del algoritmo utilizado, a través de la función del sistema HASHBYTES. La función HASHBYTES acepta dos valores: el algoritmo a usar y el valor para obtener el hash. El problema cuando se utiliza la función del sistema HASHBYTES es que no admite todos los tipos de datos que admite Microsoft SQL Server. El mayor problema con esta falta de soporte es que la función HASHBYTES no admite cadenas de caracteres de más de 8000 bytes. Cuando se usan cadenas ASCII con los tipos de datos CHAR o VARCHAR, la función del sistema HASHBYTES aceptará hasta 8000 caracteres. Cuando se usan cadenas Unicode con los tipos de datos NCHAR o NVARCHAR, la función del sistema HASHBYTES aceptará hasta 4000 caracteres. Al pasar datos binarios utilizando el tipo de datos VARBINARY, la función HASHBYTES aceptará hasta 8000 bytes de datos binarios.

Hay dos formas en que se puede usar un valor hash para encontrar el valor original. El primero es bastante simple: simplemente cree una base de datos que almacene todos los valores potenciales. Luego tome el valor hash y compárelo con los valores en la base de datos en busca de coincidencias.

De hecho, hay sitios web como <http://tools.benramsey.com/md5/> que manejan esta búsqueda en varias bases de datos disponibles en Internet. El segundo método de ataque contra MD5 se llama ataque de colisión. Un ataque de colisión se produce cuando se encuentran dos valores diferentes que se pueden hashear con el mismo valor hash, lo que permite que la verificación de los valores pase. Matemáticamente, podrías expresar el ataque como hash (value1) = hash (value2).



La encriptación se puede realizar en diferentes lugares y “momentos”:

- 1) Encriptar datos dentro de las tablas
- 2) Encriptar datos en reposo (en disco)
- 3) Encriptar datos en los cables (en el movimiento de datos por la red)
- 4) Encriptar datos a través de drivers MPIO (cuando usamos discos conectados por fibra)
- 5) Encriptar datos a través HBAs (adaptador de bus de host, para discos)

El cifrado de datos se puede realizar en muchos, muchos puntos diferentes de la aplicación, dependiendo del objetivo que estemos intentando cumplir. Algunas de estas configuraciones son más complejas de configurar, como el cifrado con el controlador PowerPath MPIO, que otras, como el Cifrado de datos transparente. No hay una respuesta única a la pregunta “¿Cómo debo cifrar mi base de datos?” Porque cada base de datos es diferente. Por eso es tan importante que haya tantas opciones en cuanto a cómo podemos cifrar la base de datos. Cada opción se cargará en alguna parte de la aplicación; solo depende de en qué parte de la aplicación deseemos colocar la carga adicional de la CPU. Podemos seleccionar desde el equipo cliente, el nivel medio, la CPU del servidor de la base de datos o los HBA en el servidor SQL, siempre y cuando deseemos colocar la carga de trabajo del procesador correspondiente a la capa en la que deseamos cifrar los datos del servidor SQL.

SEGURIDAD EN LAS CONTRASEÑAS

Una de las formas clave para proteger SQL Server es usar contraseñas seguras y robustas para las cuentas de inicio de sesión de SQL Server. Uno de los mayores agujeros de seguridad en SQL Server 2000 y versiones anteriores de Microsoft SQL Server fue que el servidor se instalaba con una contraseña de administrador de sistema (SA) en blanco de manera predeterminada y le permitiría usar una contraseña en blanco, permitiendo así que cualquier persona se conecte sin mucho trabajo.

Incluso con las versiones más recientes de Microsoft SQL Server, la cuenta SA aún es una debilidad potencial, como lo es cualquier inicio de sesión basado en autenticación de SQL Server. Esto se debe a que las cuentas de SQL se pueden romper fácilmente mediante ataques de contraseña de fuerza bruta.

Cuando usamos SQL Azure no hay una cuenta de SA disponible con la que trabajar desde el cliente de Microsoft. La cuenta SA está reservada para el uso exclusivo de Microsoft.

Al utilizar SQL Azure como instancia de base de datos, solo está disponible la autenticación SQL. SQL Azure no admite la autenticación de Windows para uso de los clientes de Microsoft, ya que el servidor de base de datos SQL Azure no admite la adición a un dominio de la empresa.

Los inicios de sesión de autenticación de SQL son más susceptibles a estos ataques de inicio de sesión que un inicio de sesión de autenticación de Windows debido a la forma en que se procesan estos inicios de sesión. Con un inicio de sesión de autenticación de SQL, cada conexión a la base de datos SQL pasa el nombre de usuario y la contraseña reales del equipo cliente al motor de SQL Server. Debido a esto, un atacante puede simplemente sentarse allí pasando nombres de usuario y contraseñas al servidor hasta que se establezca la conexión con éxito.

Con un inicio de sesión de autenticación de Windows, el proceso es muy diferente del proceso de autenticación de SQL. Cuando el cliente solicita un inicio de sesión utilizando la autenticación de Windows, se muestran varios componentes dentro de la red de Active Directory de Windows necesarios para completar la solicitud. Esto incluye el Centro de distribución de claves (KDC) de Kerberos para cuando se utiliza Kerberos para la autenticación y el Controlador de dominio de Active Directory de Windows para cuando se usa la autenticación NTLM (NT LAN Manager). El KDC de Kerberos se ejecuta en cada controlador de dominio dentro de un dominio de Active Directory que tiene instalada la función de Servicios de dominio de Active Directory (AD DS).

El proceso que ocurre cuando se establece una conexión de autenticación de Windows es bastante sencillo una vez que conocemos los componentes involucrados. Cuando el cliente solicita una conexión, el SQL Server Native



Client se pone en contacto con el KDC y solicita un ticket de Kerberos para el Nombre principal del servicio (SPN) del Motor de base de datos. Si la solicitud al KDC falla, el SQL Server Native Client intentará la solicitud de un ticket nuevamente usando la autenticación NTLM. Este ticket contendrá el Identificador de seguridad (SID) de la cuenta de dominio de Windows, así como los SID de los grupos de Windows de los que la cuenta de dominio es miembro.

Una vez que el SQL Server Native Client ha recibido el ticket del KDC, el ticket se pasa al servicio de SQL Server.

El SQL Server luego verifica el ticket contra el servicio del servidor Kerberos o NTLM en el controlador de dominio para verificar que el SID existe y está activo, y fue generado por la computadora solicitante. Una vez que se confirma la ID de Windows en el dominio, los SID para los grupos de servidores locales de los que el usuario es miembro se agregan al ticket de Kerberos y se inicia el proceso dentro del SQL Server. Si falla alguna de estas comprobaciones, se rechaza la conexión. Lo primero que el servidor de SQL verificará es si hay un inicio de sesión autenticado de Windows que coincida con el usuario. Si no hay un inicio de sesión específico de Windows, SQL Server comprueba si hay un grupo de dominios de Windows o un grupo local de Windows al que pertenece el usuario. La siguiente comprobación es para ver si el inicio de sesión o el grupo de dominio que tiene el inicio de sesión como miembro está habilitado y se le ha otorgado el derecho de conexión. La siguiente verificación es asegurarse de que el grupo de inicio de sesión o dominio tenga derecho a conectarse a un punto final específico. En este punto, el inicio de sesión de Windows se ha conectado correctamente a la instancia de SQL Server. El siguiente paso en el proceso es asignar el ID de inicio de sesión del inicio de sesión de Windows, así como cualquier grupo de dominio autorizado. Estas ID de inicio de sesión se unen dentro de una matriz interna dentro del motor de SQL Server para ser utilizadas en el último paso del proceso de autenticación, así como en varios procesos a medida que el usuario interactúa con los objetos dentro de las bases de datos de SQL Server. El último paso del proceso de conexión toma el nombre de la base de datos que se incluyó dentro de la cadena de conexión (o el inicio de sesión predeterminado de base de datos si no se especifica una base de datos de cadena de conexión) y comprueba si alguno de los ID de inicio de sesión contenidos con la matriz interna que se acaba de crear existe dentro de la base de datos como usuario. Si una de las ID de inicio de sesión existe dentro de la base de datos, entonces el inicio de sesión en el servidor SQL se completa. Si no existe ninguna de las ID de inicio de sesión dentro de la base de datos y la base de datos tiene el usuario invitado habilitado, entonces el usuario se conectará con el permiso del usuario invitado. Si no existe ninguna de las ID de inicio de sesión dentro de la base de datos y el inicio de sesión de invitado no está habilitado, la conexión se rechaza con un mensaje de error específico predeterminado de la base de datos.

PASSWORDS ROBUSTAS

Hoy no hay excusa para tener una contraseña insegura para un servidor SQL. La mayoría de los sitios web a los que nos conectamos, como los sitios web de bancos y tarjetas de crédito, requieren que utilicemos una contraseña segura de algún tipo. Es sorprendente la cantidad de compañías que no toman estas mismas técnicas en serio para su seguridad interna.

Una contraseña segura suele definirse como una contraseña que contiene al menos tres de las siguientes cuatro categorías y tiene una longitud de al menos ocho caracteres, aunque algunas empresas pueden requerir contraseñas más largas.

1. Letras minúsculas
2. Mayúsculas
3. Números
4. Caracteres especiales

Ahora, cuando se trata de contraseñas para cuentas como la cuenta de SA, que rara vez son utilizadas por personas, no hay razón para detenerse allí. Cuanto más larga sea la contraseña y los caracteres más especiales que usemos en la contraseña, menor será la posibilidad de que alguien pueda ingresar a nuestro servidor SQL.



utilizando esta cuenta. Este mismo uso de contraseñas seguras debe usarse para cualquier inicio de sesión de SQL que creemos para asegurar mejor estos inicios de sesión de SQL contra ataques de fuerza bruta.

Una cosa que podemos hacer para proteger realmente la cuenta SA es usar algunos caracteres ASCII (Código Estándar Americano para el Intercambio de Información) dentro de la contraseña. Básicamente, esto hará que la cuenta sea irrompible para la mayoría de las personas que usan scripts automatizados para atacar la contraseña de SA, ya que casi todos usan los caracteres estándar del alfabeto latino.

Con un poco de creatividad, de hecho, podríamos convertir la palabra "Password" en una contraseña verdaderamente segura y sólida. Por ejemplo, llevando al extremo, reemplazando la letra S con la Lamad hebrea, la letra O con una carita sonriente y la letra D con un signo Dong obtendríamos "Paַלְוָwׁׂrdּ"

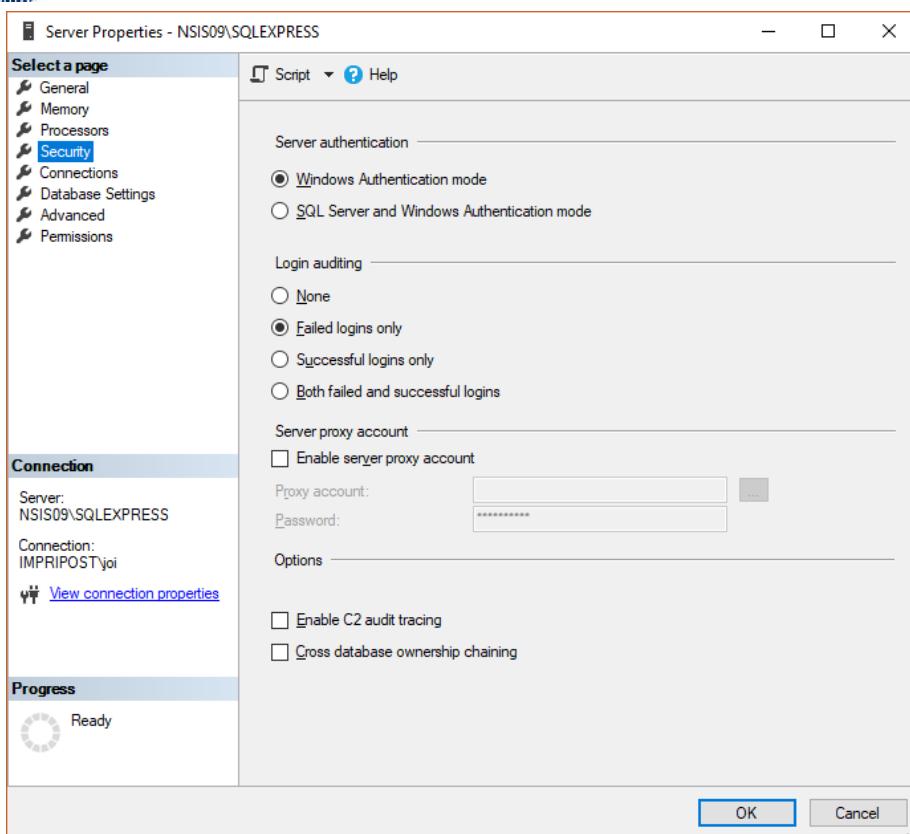
Podemos obtener más ideas sobre las formas de reemplazar los caracteres con caracteres ASCII altos en el mapa de caracteres que se puede encontrar dentro de Windows. Puede encontrar el mapa de caracteres haciendo clic en Inicio> Programas> Accesorios> Herramientas del sistema> Mapa de caracteres. Después de que se cargue la aplicación, simplemente nos desplazamos hacia abajo en la lista de caracteres disponibles hasta que encontremos los que deseamos usar.

Ahora hay un problema con el uso de estos altos caracteres ASCII para la contraseña de SA: si alguna vez necesitamos iniciar sesión en el servidor SQL usando la cuenta de SA, deberemos usar el mapa de caracteres para obtener los caracteres, o necesitamos conocer los códigos <ALT> para acceder a estos caracteres.

La cuenta SA debe ser la cuenta más segura en el servidor SQL por varias razones, la más importante de las cuales es que la cuenta SA tiene derechos sobre todo y no podemos revocar sus derechos para hacer lo que quiera. La segunda razón es que la cuenta de SA tiene un nombre de usuario conocido, ya que no podemos cambiar el nombre de usuario de SA a otra cosa. Debido a esto, alguien que está intentando ingresar a su servidor SQL no necesita adivinar el nombre de usuario; solamente necesita adivinar la contraseña que reduce la cantidad de trabajo necesario para ingresar a SQL Server a la mitad.

La forma más segura de proteger la cuenta de SA es no habilitar la autenticación de SQL, lo que requiere que todas las conexiones al SQL Server provengan de una computadora confiable que esté autenticada en el dominio de Windows. Deshabilitar la autenticación de SQL es un cambio muy fácil de realizar en su servidor SQL. Sin embargo, antes de deshabilitar la autenticación de SQL en una instancia de SQL que ya está en producción, deberemos asegurarnos de que no haya aplicaciones que inicien sesión en el servidor SQL mediante la autenticación de SQL.

Una vez hecho esto, podemos deshabilitar la autenticación de SQL. Siempre que sea posible, las nuevas instalaciones de SQL Server deben configurarse para usar solo la autenticación de Windows. La autenticación SQL puede deshabilitarse conectando el explorador de objetos en SQL Server Management Studio a la instancia en cuestión, luego haciendo clic derecho en el servidor y seleccionando las propiedades. Seleccione la pestaña Seguridad a la derecha. En la sección Autenticación del servidor, seleccionamos el botón de opción Autenticación de Windows como se muestra en la figura y hacemos clic en Aceptar. También hay código T-SQL disponible para cambiar esta configuración.



Sin embargo, el cambio no es un simple cambio a través de la configuración sp_configure como la mayoría de las configuraciones de todo el servidor. Debemos actualizar el registro utilizando el procedimiento almacenado del sistema xp_instance_regwrite desde la base de datos master. El código T-SQL necesario para cambiar esta configuración se muestra a continuación.

```
USE [master]
GO
EXEC xp_instance_regwrite N'HKEY_LOCAL_MACHINE',
    N'Software\Microsoft\MSSQLServer\MSSQLServer',
    N'LoginMode',
    REG_DWORD,
    1
GO
```

Al igual que con todos los cambios realizados en el registro (ya sea directamente o a través de este script T-SQL), los valores o cambios incorrectos causarán que el servidor SQL se comporte incorrectamente o que no se inicie en absoluto.

En resumen, uno de los mayores problemas en el mundo de TI de hoy es que una vez que hayamos creado nuestras contraseñas bonitas y seguras, ¿cómo las recordamos? Esos nombres de usuario y contraseñas probablemente se documentarán en algún lugar, generalmente dentro de una hoja de Excel que se guarda en un recurso compartido de red para que todos los administradores de bases de datos del grupo tengan acceso rápido y fácil a ellos. Sin embargo, al hacer esto, ahora hemos colocado todas las contraseñas que tanto tiempo nos tomó para asegurarnos de que sean seguras y robustas en sus archivos web.config y app.config para que cualquier persona que tenga acceso a la red compartida pueda leerlas y utilizarlas fácilmente. Por lo general, no solo los administradores de la base de datos tendrían acceso al recurso compartido de red. Además de los administradores de la base de datos, las SA, el software de backup y el sistema de monitoreo tendrían acceso al recurso compartido de red. Y esto es además de quien haya encontrado el backup perdido en nuestro servidor de



archivos. En otras palabras, asegurémonos de almacenar esa lista de contraseñas en un lugar robusto y seguro, y no en la arena pública disponible para que todos puedan leer y compartir en red.

ASEGURANDO LA INSTANCIA

- ¿Qué instalar y cuándo?: Entre más puertas y ventanas tiene nuestra casa, más puntos de acceso y vulnerabilidad. Lo mismo pasa con la instancia de SQL Server. Debemos tener cuidado y solo instalar los servicios y protocolos que necesitamos, no instalar todo lo disponibles “por las dudas”.
- Políticas de cambio de contraseña: Es importante fijar políticas de “Edad mínima de los password” y “Edad máxima de los password”
- Auditoría de inicios de sesión fallidos: Es fundamental revisar los intentos fallidos de inicio de sesión para detectar posibles ataques por fuerza bruta.
- Procedimientos almacenados como medida de seguridad: En lugar de acceder a los datos en las tablas de manera directa, es una buena medida limitar el acceso, permitiendo que se realice solamente a través de Stored Procedures
- Permisos mínimos posibles: No otorgar permisos de más “por las dudas”. Sólo otorgar los mínimos permisos necesarios para el correcto funcionamiento.
- Servidores vinculados: Los servidores vinculados permiten acceder a datos que están alojados en otros servidores y son una fuente potencial de peligro, aunque muchas veces son indispensables para compartir información.

ATAQUES SQL INJECTION

Un ataque de inyección SQL es probablemente el ataque más fácil de prevenir, mientras que es uno de los menos protegidos contra las formas de ataque. El núcleo del ataque es que un comando SQL se agrega al final de un campo de formulario en la web o en frontend de la aplicación (generalmente a través de un sitio web), con la intención de romper la secuencia de comandos SQL original y luego ejecutar la secuencia de comandos SQL que fue inyectado en el campo de formulario. Esta inyección de SQL ocurre con mayor frecuencia cuando genera SQL de forma dinámica dentro de su aplicación de front-end. Estos ataques son más comunes con las aplicaciones heredadas de páginas Active Server (ASP) y procesador de hipertexto (PHP), pero siguen siendo un problema con las aplicaciones basadas en web ASP.NET y Java. La razón principal detrás de un ataque de Inyección de SQL se debe a las prácticas de codificación deficientes, tanto dentro de la aplicación de front-end como dentro de los procedimientos almacenados de la base de datos. Muchos desarrolladores han aprendido mejores prácticas de desarrollo desde que se lanzó ASP.NET, pero la Inyección SQL sigue siendo un gran problema entre la cantidad de aplicaciones legacy existentes y las aplicaciones más nuevas creadas por desarrolladores que no tomaron en serio la Inyección SQL al crear la aplicación.

Como ejemplo, supongamos que la aplicación web de front-end crea una secuencia de comandos SQL dinámica que termina ejecutando una secuencia de comandos SQL similar a la siguiente:

```
SELECT * FROM Ventas.Ordenes WHERE ordenid=25
```

Este script SQL se crea cuando el cliente accede a la parte del historial de pedidos de venta del sitio web de la empresa. El valor pasado como ordenid se toma de la cadena de consulta en la URL, por lo que la consulta que se muestra arriba se crea cuando el cliente va a la URL <http://www.ejemplo.com/ordenes/historialordenes.aspx?Id=25>.

Dentro del código .NET, se realiza una concatenación de cadena simple para armar la consulta SQL. Por lo tanto, cualquier valor que se ponga al final de la cadena de consulta se pasa a la base de datos al final de la instrucción de selección. Si el atacante cambiara la cadena de consulta a algo como "/historialordenes.aspx?id=25; delete from Ventas.Ordenes", entonces la consulta enviada al SQL Server será un poco más peligrosa de ejecutar, como vemos a continuación:



```
SELECT * FROM Ventas.Ordenes WHERE ordenid=25; delete from Ventas.Ordenes
```

La forma en que funciona la consulta en el ejemplo anterior es que a la base de datos SQL se le indica el fin de un comando a través del punto y coma ";" y que hay otra sentencia que debe ejecutarse. El SQL Server luego procesa la siguiente declaración como se indica. Mientras que la consulta inicial se ejecuta con normalidad, y sin que se genere ningún error, pero cuando observemos la tabla Ventas.Ordenes, no veremos ningún registro en la tabla porque la segunda consulta en ese lote se ejecutará contra la base de datos a continuación. Incluso si el atacante omite el valor que la consulta espera, pueden pasar "; delete Ventas.Ordenes;" y mientras la primera consulta intenta devolver los datos de la tabla fallará, el lote continuará avanzando a la siguiente declaración, que eliminará todos los registros en la tabla.

Muchas personas inspeccionarán el texto de los parámetros en busca de varias palabras clave para evitar estos ataques de inyección SQL. Sin embargo, esto solo proporciona la protección más rudimentaria ya que hay muchas, muchas formas de forzar estos ataques para que funcionen. Algunas de estas técnicas incluyen pasar datos binarios, hacer que SQL Server vuelva a convertir los datos binarios en una cadena de texto y luego ejecutar la cadena. Esto se puede probar ejecutando la instrucción T-SQL que se muestra a continuación:

```
DECLARE @v varchar(255)
SELECT @v = cast(0x73705F68656C706462 as varchar(255))
EXEC (@v)
```

Cuando se aceptan datos de un usuario, ya sea un cliente o un empleado, una buena manera de asegurarse de que el valor no se utilizará para un ataque de inyección de SQL es validar que los datos que se devuelven son del tipo de datos esperado. Si se espera un número, la aplicación de front-end debe garantizar que haya un número dentro del valor. Si se espera una cadena de texto, asegúrese de que la cadena de texto tenga la longitud correcta y que no contenga datos binarios dentro de ella. La aplicación de front-end debe poder validar todos los datos que se transmiten por parte del usuario, ya sea informando al usuario del problema y permitiéndole corregir el problema, o mediante un bloqueo correcto de tal manera que se devuelva un error y no se envíen comandos a la base de datos o al sistema de archivos. El hecho de que los usuarios deban enviar datos válidos no significa que vayan a hacerlo. Si se pudiera confiar en los usuarios, la mayoría de esta unidad no sería necesaria.

¿POR QUÉ SON EXITOSOS LOS ATAQUES SQL INJECTION?

Los ataques de inyección de SQL son tan exitosos por varias razones, la más común de las cuales es que muchos desarrolladores más nuevos simplemente no conocen el problema. Dado que los plazos de los proyectos son tan cortos, estos desarrolladores juniors no tienen tiempo para investigar implicaciones de seguridad del uso de SQL dinámico. Estas aplicaciones se dejan en producción durante meses o años, con poco o ningún mantenimiento. Estos desarrolladores pueden avanzar en su carrera sin que nadie les brinde la orientación necesaria para evitar estos problemas.

Ahora los desarrolladores no son los únicos culpables de los problemas de ataque de inyección de SQL. La Administración de TI debe tener políticas implementadas para garantizar que los desarrolladores más nuevos que ingresan no tengan la capacidad de escribir SQL en línea dinámico contra el motor de la base de datos. Estas políticas deben incluir reglas como las siguientes:

1. Toda la interacción de la base de datos debe resumirse a través de procedimientos almacenados.
2. Ningún procedimiento almacenado debe tener SQL dinámico a menos que no haya otra opción.
3. Las aplicaciones no deben tener acceso a objetos de tabla o vista a menos que lo requiera el SQL dinámico, que está permitido bajo la regla # 2.
4. Todas las llamadas a la base de datos deben estar parametrizadas en lugar de ser un SQL dinámico en línea.
5. Ningún comentario del usuario debe ser confiable y considerado como seguro; Todas las interacciones del usuario son sospechosas.



Con la introducción de asignaciones relacionales de objetos (ORM), como el enlace a SQL y nHibernate, los problemas de inyección de SQL se reducen considerablemente, ya que el código ORM correctamente realizado parametrizará automáticamente las consultas SQL. Sin embargo, si el ORM llama a procedimientos almacenados, y esos procedimientos almacenados tienen SQL dinámico dentro de ellos, la aplicación aún es susceptible a ataques de Inyección de SQL.

Los ataques de inyección SQL plantean algunos de los mayores peligros para la base de datos y los clientes, ya que generalmente se usan para afectar directamente la información que el cliente ve y se pueden usar con bastante facilidad para intentar enviar códigos maliciosos a las computadoras de los clientes. Estos ataques son muy populares entre los atacantes porque son una forma relativamente fácil de explotar el diseño de sistemas. También son populares porque son fáciles de reproducir una vez que se considera que un sitio es comprometible, ya que generalmente toma mucho tiempo corregir todos los puntos de ataque potenciales en un sitio web. Este período de tiempo deja el sitio web y la base de datos abiertos al ataque durante un largo período de tiempo, ya que las empresas generalmente no están dispuestas a cerrar sus sitios web que miran hacia los clientes mientras se repara el diseño del sitio web.

Debido a la forma en que funcionan los ataques de inyección de SQL, el Administrador de la base de datos, el Desarrollador de la base de datos, el Desarrollador de la aplicación y el Administrador de sistemas deben trabajar juntos para garantizar que están protegiendo correctamente los datos dentro de la base de datos y la red de la compañía en general. A medida que los desarrolladores de base de datos y de aplicaciones comienzan a acostumbrarse a escribir código que no es susceptible a los ataques de inyección de SQL, el proyecto actual se volverá más seguro, al igual que los proyectos futuros en los que trabajen los miembros del equipo.

CRIPTOLOGÍA

HISTORIA

En el año 500 a.C. los griegos utilizaron un cilindro llamado "escíptala" alrededor del cual enrollaban una tira de cuero. Al escribir un mensaje sobre el cuero y desenrollarlo se veía una lista de letras sin sentido. El mensaje correcto sólo podía leerse al enrollar el cuero nuevamente en un cilindro de igual diámetro.



Durante el Imperio Romano Julio Cesar

empleó un sistema de cifrado consistente en sustituir la letra a encriptar por otra letra distanciada a tres posiciones más adelante. Durante su reinado, los mensajes de Julio Cesar nunca fueron desencriptados.

En el S. XII Roger Bacon y en el S. XV León Batista Alberti inventaron y publicaron sendos algoritmos de encriptación basados en modificaciones del método de Julio César.

Durante la segunda guerra mundial en un lugar llamado Bletchley Park (70 Km al norte de Londres) un grupo de científicos trabajaba en Enigma, la máquina encargada de cifrar los mensajes secretos alemanes.

En este grupo se encontraban tres matemáticos polacos llamados Marian Rejewski, Jerzy Różycki, Henryk Zygalski y "un joven que se mordía siempre las uñas, iba con ropa sin planchar y era más bien bajito. Este joven retraído se llamaba Alan Turing y había sido reclutado porque unos años antes había creado un ordenador binario. Probablemente poca gente en los servicios secretos ingleses sabía lo que era un ordenador (y mucho menos binario) ... pero no cabía duda de que sólo alguien realmente inteligente podía inventar algo así, cualquier cosa que eso fuese..."



Era mucho más abstracto que todos sus antecesores y sólo utilizaba 0 y 1 como valores posibles de las variables de su álgebra.²¹

Sería Turing el encargado de descifrar el primer mensaje de Enigma y, cambiar el curso de la guerra, la historia y de... la Seguridad Informática actual.

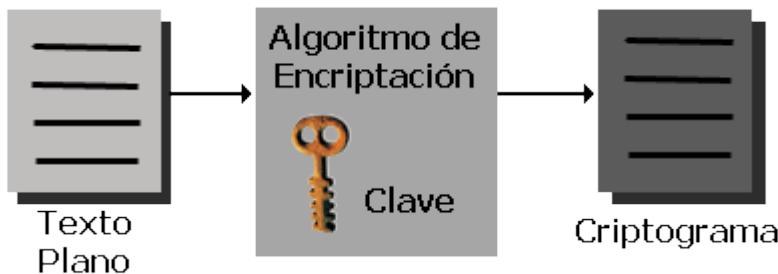
CRIPTOGRAFÍA

La palabra **Criptografía** proviene etimológicamente del griego Kruiptoz (Kriptos-Oculto) y Grajein (Grafo-Escritura) y significa "arte de escribir con clave secreta o de un modo enigmático"²²

Aportando luz a la definición cabe aclarar que la Criptografía hace años que dejó de ser un arte para convertirse en una técnica (o conjunto de ellas) que tratan sobre la protección (ocultamiento ante personas no autorizadas) de la información. Entre las disciplinas que engloba cabe destacar la Teoría de la Información, la Matemática Discreta, la Teoría de los Grandes Números y la Complejidad Algorítmica.

Es decir que la Criptografía es la ciencia que consiste en transformar un mensaje inteligible en otro que no lo es (mediante claves que sólo el emisor y el destinatario conocen), para después devolverlo a su forma original, sin que nadie que vea el mensaje cifrado sea capaz de entenderlo.

El mensaje cifrado recibe el nombre **Criptograma**



La importancia de la Criptografía radica en que es el único método actual capaz de hacer cumplir el objetivo de la Seguridad Informática: "mantener la Privacidad, Integridad, Autenticidad..." y hacer cumplir con el **No Rechazo**, relacionado a no poder negar la autoría y recepción de un mensaje enviado.

CRIPTOANÁLISIS

Es el arte de estudiar los mensajes ilegibles, encriptados, para transformarlos en legibles sin conocer la clave, aunque el método de cifrado empleado siempre es conocido.

CRIPTOSISTEMA

"Un Criptosistema se define como la quíntupla (m, C, K, E, D) , donde:

- m representa el conjunto de todos los mensajes sin cifrar (texto plano) que pueden ser enviados.
- C Representa el conjunto de todos los posibles mensajes cifrados, o criptogramas.
- K representa el conjunto de claves que se pueden emplear en el Criptosistema.

²¹ Extraído de <http://www.kriptopolis.org>

²² LUCENA LÓPEZ, Manuel José. Criptografía y Seguridad en Computadores. Dpto. de Informática Universidad de Jaén. Edición virtual. España. 1999. <http://www.kriptopolis.org>. Capítulo 2-Página 23.



- E es el conjunto de transformaciones de cifrado o familia de funciones que se aplica a cada elemento de m para obtener un elemento de C. Existe una transformación diferente E_k para cada valor posible de la clave K.
- D es el conjunto de transformaciones de descifrado, análogo a E.

Todo Criptosistema cumple la condición $D_k(E_k(m))=m$ es decir, que si se tiene un mensaje m, se cifra empleando la clave K y luego se descifra empleando la misma clave, se obtiene el mensaje original m.²³

Existen dos tipos fundamentales de Criptosistemas utilizados para cifrar datos e información digital y ser enviados posteriormente después por medios de transmisión libre.

- a. **Simétricos o de clave privada:** se emplea la misma clave K para cifrar y descifrar, por lo tanto, el emisor y el receptor deben poseer la clave. El mayor inconveniente que presentan es que se debe contar con un canal seguro para la transmisión de dicha clave.
- b. **Asimétricos o de llave pública:** se emplea una doble clave conocidas como K_p (clave privada) y K_p (clave Pública). Una de ellas es utilizada para la transformación E de cifrado y la otra para el descifrado D. En muchos de los sistemas existentes estas claves son intercambiables, es decir que si empleamos una para cifrar se utiliza la otra para descifrar y viceversa.

Los sistemas asimétricos deben cumplir con la condición de que la clave Pública (al ser conocida y sólo utilizada para cifrar) no debe permitir calcular la privada. Como puede observarse este sistema permite intercambiar claves en un canal inseguro de transmisión ya que lo único que se envía es la clave pública.

Los algoritmos asimétricos emplean claves de longitud mayor a los simétricos. Así, por ejemplo, suele considerarse segura una clave de 128 bits para estos últimos, pero se recomienda claves de 1024 bits (como mínimo) para los algoritmos asimétricos. Esto permite que los algoritmos simétricos sean considerablemente más rápidos que los asimétricos.

En la práctica actualmente se emplea una combinación de ambos sistemas ya que los asimétricos son computacionalmente más costosos (mayor tiempo de cifrado). Para realizar dicha combinación se cifra el mensaje m con un sistema simétrico y luego se encripta la clave K utilizada en el algoritmo simétrico (generalmente más corta que el mensaje) con un sistema asimétrico.

Después de estos Criptosistemas modernos podemos encontrar otros no menos importantes utilizados desde siempre para cifrar mensajes de menos importancia o domésticos, y que han ido perdiendo su eficacia por ser fácilmente criptoanalizables y por tanto "reventables". Cada uno de los algoritmos clásicos descritos a continuación utilizan la misma clave K para cifrar y descifrar el mensaje.

TRANSPOSICIÓN

Son aquellos que alteran el orden de los caracteres dentro del mensaje a cifrar. El algoritmo de transposición más común consiste en colocar el texto en una tabla de n columnas. El texto cifrado serán los caracteres dados por columna (de arriba hacia abajo) con una clave K consistente en el orden en que se leen las columnas.

²³ LUCENA LÓPEZ, Manuel José. Criptografía y Seguridad en Computadores. Dpto. de Informática Universidad de Jaén. Edición virtual. España. 1999. <http://www.kriptopolis.org>. Capítulo 2-Página 24



Ejemplo: Si $n = 3$ columnas, la clave K es (3,1,2) y el mensaje a cifrar "SEGURIDAD INFORMATICA".

1	2	3
S	E	G
U	R	I
D	A	D
	I	N
F	O	R
M	A	T
I	C	A

El mensaje cifrado será: "GIDNRTASUD FMIERAIOAC"

CIFRADOS MONOALFABÉTICOS

Sin desordenar los símbolos del lenguaje, se establece una correspondencia única para todos ellos en todo el mensaje. Es decir que, si al carácter A le corresponde carácter D, esta correspondencia se mantiene durante todo el mensaje.

ALGORITMO DE CÉSAR

Es uno de los algoritmos criptográficos más simples. Consiste en sumar 3 al número de orden de cada letra. De esta forma a la A le corresponde la D, a la B la E, y así sucesivamente. Puede observarse que este algoritmo ni siquiera posee clave, puesto que la transformación siempre es la misma.

Obviamente, para descifrar basta con restar 3 al número de orden de las letras del criptograma.

Ejemplo: Si el algoritmo de cifrado es:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Entonces el mensaje cifrado será:

S E G U R I D A D I N F O R M A T I C A

V H J X U L G D G L Q I R U P D W L F D

SUSTITUCIÓN GENERAL

Es el caso general del algoritmo de César. El sistema consiste en sustituir cada letra por otra aleatoria. Esto supone un grado más de complejidad, aunque como es de suponer las propiedades estadísticas del texto original se conservan en el criptograma y por lo tanto el sistema sigue siendo criptoanalizable.

ALGORITMOS SIMÉTRICOS MODERNOS (LLAVE PRIVADA)

La mayoría de los algoritmos simétricos actuales se apoyan en los conceptos de Confusión y Difusión vertidos por Claude Shannon sobre la Teoría de la Información a finales de los años cuarenta.

Estos métodos consisten en ocultar la relación entre el texto plano, el texto cifrado y la clave (Confusión); y repartir la influencia de cada bit del mensaje original lo más posible entre el mensaje cifrado (Difusión).

El objetivo del presente no es entrar en detalles de cada uno de los muchos algoritmos existentes, por lo que sólo se dará una idea de su funcionamiento y complejidad.



REDES DE FEISTEL

Este algoritmo no es un algoritmo de cifrado per se, pero muchos de los vistos a continuación lo utilizan como parte vital en su funcionamiento. Se basa en dividir un bloque de longitud n (generalmente el texto a cifrar) en dos mitades, L y R. Luego se define un cifrado de producto iterativo en el que la salida de cada ronda es la entrada de la siguiente.

DES

Data Encryption Standard es el algoritmo simétrico más extendido mundialmente. A mediados de los setenta fue adoptado como estándar para las comunicaciones seguras (Estándar AES) del gobierno de EE. UU. En su principio fue diseñado por la NSA (National Security Agency)²⁴ para ser implementado en hardware, pero al extenderse su algoritmo se comenzó a implementar en software.

DES utiliza bloques de 64 bits, los cuales codifica empleando claves de 56 bits y aplicando permutaciones a nivel de bit en diferentes momentos (mediante tablas de permutaciones y operaciones XOR). Es una red de Feistel de 16 rondas, más dos permutaciones, una que se aplica al principio y otra al final.

La flexibilidad de DES reside en que el mismo algoritmo puede ser utilizado tanto para cifrar como para descifrar, simplemente invirtiendo el orden de las 16 subclaves obtenidas a partir de la clave de cifrado.

En la actualidad no se ha podido romper el sistema DES criptoanalíticamente (deducir la clave simétrica a partir de la información interceptada). Sin embargo, una empresa española sin fines de lucro llamado Electronic Frontier Foundation (EFF)²⁵ construyó en enero de 1999 una máquina capaz de probar las 2^{56} claves posibles en DES y romperlo sólo en tres días con fuerza bruta.

A pesar de su caída DES sigue siendo utilizado por su amplia extensión de las implementaciones vía hardware existentes (en cajeros automáticos y señales de video por ejemplo) y se evita tener que confiar en nuevas tecnologías no probadas. En vez de abandonar su utilización se prefiere suplantar a DES con lo que se conoce como cifrado múltiple, es decir aplicar varias veces el mismo algoritmo para fortalecer la longitud de la clave.

DES MÚLTIPLE

Consiste en aplicar varias veces el algoritmo DES (con diferentes claves) al mensaje original. El más conocidos de todos ellos el Triple-DES (T-DES), el cual consiste en aplicar 3 veces DES de la siguiente manera:

1. Se codifica con la clave K¹.
2. Se decodifica el resultado con la clave K².
3. Lo obtenido se vuelve a codificar con K¹.

La clave resultante es la concatenación de K¹ y K² con una longitud de 112 bits.

En 1998 el NIST (National Institute of Standards Technology) convocó a un concurso para poder determinar un algoritmo simétricos seguro y próximo sustituto de DES. Se aceptaron 15 candidatos y a principios del año 2000 los 5 finalistas fueron MARS, RC-6, Serpent y TwoFish y Rijndael (que en octubre sería el ganador).

IDEA

El International Data Encryption Algorithm fue desarrollado en Alemania a principios de los noventa por James L. Massey y Xuejia Lai.

²⁴ National Security Agency (NSA): <http://www.nsa.gov>

²⁵ Electronic Frontier Foundation (EFF) actualmente ya no se encuentra trabajando, pero puede visitarse su sitio en <http://www.eff.com>



Trabaja con bloques de 64 bits de longitud empleando una clave de 128 bits y, como en el caso de DES, se utiliza el mismo algoritmo tanto para cifrar como para descifrar.

El proceso de encriptación consiste ocho rondas de cifrado idéntico, excepto por las subclaves utilizadas (segmentos de 16 bits de los 128 de la clave), en donde se combinan diferentes operaciones matemáticas (XORs y Sumas Módulo 16) y una transformación final.

"En mi opinión, él es el mejor y más seguro algoritmo de bloques disponible actualmente al público."²⁶

BLOWFISH

Este algoritmo fue desarrollado por Bruce Schneier en 1993. Para la encriptación emplea bloques de 64 bits y permite claves de encriptación de diversas longitudes (hasta 448 bits).

Generalmente, utiliza valores decimales de Π (aunque puede cambiarse a voluntad) para obtener las funciones de encriptación y desencriptación. Estas funciones emplean operaciones lógicas simples y presentes en cualquier procesador. Esto se traduce en un algoritmo "liviano", que permite su implementación, vía hardware, en cualquier controlador (como teléfonos celulares, por ejemplo).

RC5

Este algoritmo, diseñado por RSA²⁷, permite definir el tamaño del bloque a encriptar, el tamaño de la clave utilizada y el número de fases de encriptación. El algoritmo genera una tabla de encriptación y luego procede a encriptar o desencriptar los datos.

CAST

Es un buen sistema de cifrado en bloques con una clave CAST-128 bits, es muy rápido y es gratuito. Su nombre deriva de las iniciales de sus autores, Carlisle, Adams, Stafford Tavares, de la empresa Northern Telecom (NorTel).

CAST no tiene claves débiles o semidébiles y hay fuertes argumentos acerca que CAST es completamente inmune a los métodos de criptoanálisis más potentes conocidos.

También existe una versión con clave CAST-256 bits que ha sido candidato a AES.

RIJNDAEL (EL NUEVO ESTÁNDAR AES)

Rijndael, el nuevo algoritmo belga mezcla de Vincent Rijmen y Joan Daemen (sus autores) sorprende tanto por su innovador diseño como por su simplicidad práctica; aunque tras él se esconde un complejo trasfondo matemático.

Su algoritmo no se basa en redes de Feistel, y en su lugar se ha definido una estructura de "capas" formadas por funciones polinómicas reversibles (tienen inversa) y no lineales. Es fácil imaginar que el proceso de descifrado consiste en aplicar las funciones inversas a las aplicadas para cifrar, en el orden contrario.

Las implementaciones actuales pueden utilizar bloques de 128, 192 y 256 bits de longitud combinadas con claves de 128, 192 y 256 bits para su cifrado; aunque tanto los bloques como las claves pueden extenderse en múltiplos de 32 bits.

²⁶ SCHNEIER, Bruce. *Applied Cryptography*. Segunda Edición. EE. UU. 1996

²⁷ RSA Labs: <http://www.rsa.com>. No confundir con el algoritmo de clave pública del mismo nombre RSA (Rivest-Shamir-Adleman)



Si bien su joven edad no permite asegurar nada, según sus autores, es altamente improbable que existan claves débiles en el nuevo AES. También se ha probado la resistencia al criptoanálisis tanto lineal como diferencial, asegurando así la desaparición de DES.

CRIPTOANÁLISIS DE ALGORITMOS SIMÉTRICOS

El Criptoanálisis comenzó a extenderse a partir de la aparición de DES por sospechas (nunca confirmadas) de que el algoritmo propuesto por la NSA contenía puertas traseras. Entre los ataques más potentes a la criptografía simétrica se encuentran:

- **Criptoanálisis Diferencial:** Ideado por Biham y Shamir en 1990, se basa en el estudio de dos textos codificados para estudiar las diferencias entre ambos mientras se los está codificando. Luego puede asignarse probabilidades a ciertas claves de cifrado.
- **Criptoanálisis Lineal:** Ideado por Mitsuru Matsui, se basa en tomar porciones del texto cifrado y porciones de otro texto plano y efectuar operaciones sobre ellos de forma tal de obtener probabilidades de aparición de ciertas claves.

Sin embargo, estos métodos, no han podido ser muy eficientes en la práctica. En el momento después de que un sistema criptográfico es publicado y se muestra inmune a estos dos tipos de ataques (y otros pocos) la mayor preocupación es la longitud de las claves.

ALGORITMOS ASIMÉTRICOS (LLAVE PRIVADA-PÚBLICA)

Ideado por los matemáticos Whitfield Diffie y Martín Hellman (DH) con el informático Ralph Merkle a mediados de los 70, estos algoritmos han demostrado su seguridad en comunicaciones inseguras como Internet. Su principal característica es que no se basa en una única clave sino en un par de ellas: una conocida (Pública) y otra Privada.

Actualmente existen muchos algoritmos de este tipo, pero han demostrado ser poco utilizables en la práctica ya sea por la longitud de las claves, la longitud del texto encriptado generado o su velocidad de cifrado extremadamente largos.

DH está basado en las propiedades y en el tiempo necesario para calcular el valor del logaritmo de un número extremadamente alto y primo.

RSA

Este algoritmo fue ideado en 1977 por Ron Rivest, Adi Shamir y Leonard Adleman (RSA). Es el más empleado en la actualidad, sencillo de comprender e implementar, aunque la longitud de sus claves es bastante considerable (ha pasado desde sus 200 bits originales a 2048 actualmente).

RSA es la suma de dos de los algoritmos más importantes de la historia: el Máximo Común Divisor de Euclides (Grecia 450-377 A.C.) y el último teorema de Fermat (Francia 1601-1665).

Se emplean las ventajas proporcionadas por las propiedades de los números primos cuando se aplican sobre ellos operaciones matemáticas basadas en la función módulo. En concreto, emplea la función exponencial discreta para cifrar y descifrar, y cuya inversa, el logaritmo discreto, es muy difícil de calcular.

Los cálculos matemáticos de este algoritmo emplean un número denominado Módulo Público, N, que forma parte de la clave pública y que se obtiene a partir de la multiplicación de dos números primos, p y q, diferentes y grandes (del orden de 512 bits) y que forman parte de la clave privada. La gran propiedad de RSA es que, mientras que N es público, los valores de p y q se pueden mantener en secreto debido a la dificultad que entraña la factorización de un número grande.

La robustez del algoritmo se basa en la facilidad para encontrar dos números primos grandes frente a la enorme dificultad que presenta la factorización de su producto. Aunque el avance tecnológico hace que cada vez sea más



rápido un posible ataque por fuerza bruta, el simple hecho de aumentar la longitud de las claves empleadas supone un incremento en la carga computacional lo suficientemente grande para que este tipo de ataque sea inviable.

Sin embargo, se ha de notar que, aunque el hecho de aumentar la longitud de las claves RSA no supone ninguna dificultad tecnológica, las leyes de exportación de criptografía de EE. UU. imponían, hasta el 20 de septiembre de 2000, un límite a dicha longitud por lo que el uso comercial de RSA no estaba permitido, ya que la patente pertenecía a los laboratorios RSA. Desde esta fecha su uso es libre.

ATAQUES A RSA

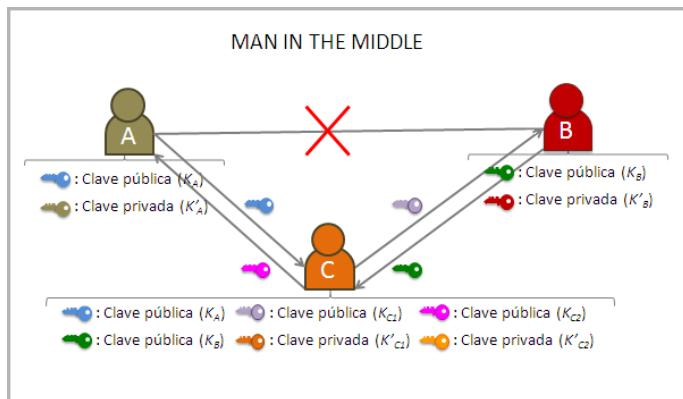
Si un atacante quiere recuperar la clave privada a partir de la pública debe obtener p y q a partir de N , lo cual actualmente es un problema intratable si los números primos son lo suficientemente grandes (alrededor de 200 dígitos).

Vale decir que nadie ha demostrado que no pueda existir un método que permita descifrar un mensaje sin usar la clave privada y sin factorizar N . Así, aunque el algoritmo es bastante seguro conceptualmente, existen algunos ataques que pueden ser efectivos al apoyarse sobre deficiencias en la implementación y uso de este.

El ataque con mayores probabilidades de éxito es el **ataque de intermediario**, que en realidad puede darse sobre cualquier algoritmo de clave pública. Supongamos:

... que A quiere establecer una comunicación con B, y que C quiere espiarla. Cuando A le solicite a B su clave pública K_B , C se interpone, obteniendo la clave de B y enviado a A una clave falsa K_C , creada por él. Cuando A codifique el mensaje, C lo intercepta de nuevo, lo decodifica con su clave propia y emplea K_B para codificarlo y enviarlo a B... ni A ni B sospecharán nunca de lo sucedido.

La única manera de evitar esto consiste en asegurar a A que la clave pública de B es auténtica. Para ello esta debería ser firmada por un amigo común que, actuando como Autoridad Certificadora, certifique su autenticidad.



Otros ataques (como el de claves débiles, el de texto plano escogido, el de módulo común, y el de exponente bajo) aprovechan vulnerabilidades específicas de algunas implementaciones.

CURVAS ELÍPTICAS (CEE)

Las curvas elípticas fueron propuestas por primera vez para ser usadas en aplicaciones criptográficas en 1985 de forma independiente por Miller y Koblitz. Las curvas elípticas en sí llevan estudiándose durante muchos siglos y están entre los objetos más ricamente estructurados y estudiados de la teoría de números.

La eficiencia de este algoritmo radica en la longitud reducida de las claves, lo cual permite su implementación en sistemas de bajos recursos como teléfonos celulares y Smart Cards. Puede hacerse la siguiente comparación con RSA, obteniendo el mismo nivel de seguridad:

- CCE de 163 bits = RSA de 1024 bits
- CCE de 224 bits = RSA de 2048 bits



Otros algoritmos asimétricos conocidos son ElGamal (basado en el Problema de los Logaritmos Discretos de Diffie-Hellman DH), Rabin (basado en el problema del cálculo de raíces cuadradas módulo un número compuesto), DSS y LUC.

AUTENTIFICACIÓN

Se entiende por Autentificación cualquier método que permita garantizar alguna característica sobre un objeto dado. Interesa comprobar la autentificación de:

- a. Un Mensaje mediante una firma: se debe garantizar la procedencia de un mensaje conocido, de forma de poder asegurar que no es una falsificación. A este mecanismo se lo conoce como **Firma Digital** y consiste en asegurar que el mensaje m proviene del emisor E y no de otro.
- b. Un Usuario mediante una contraseña: se debe garantizar la presencia de un usuario autorizado mediante una contraseña secreta.
- c. Un Dispositivo: se debe garantizar la presencia de un dispositivo válido en el sistema, por ejemplo, una llave electrónica.

FIRMA DIGITAL

Una firma digital se logra mediante una Función Hash de Resumen. Esta función se encarga de obtener una "muestra única" del mensaje original. Dicha muestra es más pequeña y es muy difícil encontrar otro mensaje que tenga la misma firma. Suponiendo que B envía un mensaje m firmado a A , el procedimiento es:

- a. B genera un resumen del mensaje $r(m)$ y lo cifra con su clave privada.
- b. B envía el criptograma.
- c. A genera su propia copia de $r(m)$ usando la clave pública de B asociada a la privada.
- d. A compara su criptograma con el recibido y si coinciden el mensaje es auténtico.

Cabe destacar que:

1. Cualquiera que posea la clave pública de B puede constatar que el mensaje proviene realmente de B .
2. La firma digital es distinta en todos los documentos: si A firma dos documentos, produce dos criptogramas distintos y; si A y B firman el mismo documento m también se producen dos criptogramas diferentes.

Las funciones Hash están basadas en que un mensaje de longitud arbitraria se transforma en un mensaje de longitud constante dividiendo el mensaje en partes iguales, aplicando la función de transformación a cada parte y sumando todos los resultados obtenidos.

Actualmente se recomienda utilizar firmas de al menos 128 bits (38 dígitos) siendo 160 bits (48 dígitos) el valor más utilizado.

- **MD5**

El Message Digest 5 (resultado mejorado sobre el MD4 original de Ron Rivest) procesa los mensajes de entrada en bloques de 512, y que produce una salida de 128 bits

Siendo m un mensaje de b bits de longitud, se alarga m hasta que su longitud sea 64 bits inferior a un múltiplo de 512. Esto se realiza agregando un 1 y tantos ceros como sea necesario. A continuación, se agregan 64 bits con el valor de b comenzando por el byte menos significativo.

Posteriormente, se realizan 64 operaciones divididas en 4 rondas sobre estos bloques de 512 bits. Finalmente, se suman y concatenan los bloques obteniendo la firma deseada de m .

- **SHA-1**

El Secure Hash Algorithm fue desarrollado por la NSA, y genera firmas de 160 bits a partir de bloques de 512 bits del mensaje original.



Su funcionamiento es similar al MD5, solo variando la longitud de los bloques y la cantidad de operaciones realizadas en las 5 rondas en las que se divide el proceso.

Otros algoritmos utilizados para obtener firmas digitales son: DSA (Digital Signature Logarithm) y el RIPE-MD160.

PGP (PRETTY GOOD PRIVACY)

Este proyecto de "Seguridad Bastante Buena" pertenece a Phill Zimmerman quien decidió crearlo en 1991 "por falta de herramientas criptográficas sencillas, potentes, baratas y al alcance del usuario común. Es personal. Es privado. Y no es de interés para nadie más que no sea usted... Existe una necesidad social en crecimiento para esto. Es por eso que lo creé."²⁸

Actualmente PGP es la herramienta más popular y fiable para mantener la seguridad y privacidad en las comunicaciones tanto para pequeños usuarios como para grandes empresas.

FUNCIONAMIENTO DE PGP

1. Anillos de Claves

Un anillo es una colección de claves almacenadas en un archivo. Cada usuario tiene dos anillos, uno para las claves públicas y otro para las claves privadas.

Cada una de las claves, además, posee un identificador de usuario, fecha de expiración, versión de PGP y una huella digital única hexadecimal suficientemente corta que permita verificar la autenticidad de la clave.

2. Codificación de Mensajes

Como ya se sabe, los algoritmos simétricos de cifrado son más rápidos que los asimétricos. Por esta razón PGP cifra primero el mensaje empleando un algoritmo simétrico con una clave generada aleatoriamente (clave de sesión) y posteriormente codifica la clave haciendo uso de la llave pública del destinatario. Dicha clave es extraída convenientemente del anillo de claves públicas a partir del identificador suministrado por el usuario.

Nótese que para que el mensaje pueda ser leído por múltiples destinatarios basta con que se incluya en la cabecera cada una de las claves públicas correspondientes.

3. Decodificación de Mensajes

Cuando se trata de decodificar el mensaje, PGP simplemente busca en la cabecera las claves públicas con las que está codificado, pide una contraseña para abrir el anillo de claves privadas y comprueba si se tiene una clave que permita decodificar el mensaje.

Nótese que siempre que se quiere hacer uso de una clave privada, habrá que suministrar la contraseña correspondiente, por lo que, si este anillo quedara comprometido, el atacante tendría que averiguar dicha contraseña para descifrar los mensajes.

No obstante, si el anillo de claves privadas quedara comprometido, es recomendable revocar todas las claves almacenadas y generar otras nuevas.

4. Compresión de Archivos

PGP generalmente comprime el texto plano antes de encriptar el mensaje (y lo descomprime después de desencriptarlo) para disminuir el tiempo de cifrado, de transmisión y de alguna manera fortalecer la seguridad del cifrado ante el criptoanálisis que explotan las redundancias del texto plano.

PGP utiliza rutinas de compresión de dominio público creadas por Gailly-Adler-Wales (basadas en los algoritmos de Liv-Zemple) funcionalmente semejantes a las utilizadas en los softwares comerciales de este tipo.

²⁸ "Porqué escribí PGP". Declaraciones de Phill Zimmerman. <http://www.pgp.com> - <http://pgp.org>



5. Algoritmos Utilizados por PGP

Las diferentes versiones de PGP han ido adoptando diferentes combinaciones de algoritmos de firma y cifrado eligiendo entre los estudiados. Las firmas se realizan mediante MD5, SHA-1 y/o RIPE-MD6. Los algoritmos simétricos utilizados pueden ser IDEA, CAST y TDES y los asimétricos RSA y ElGamal.

ESTEGANOGRÁFÍA

Consiste en ocultar en el interior de información aparentemente inocua, otro tipo de información (cifrada o no). El texto se envía como texto plano, pero entremezclado con mucha cantidad de "basura" que sirve de camuflaje al mensaje enviado. El método de recuperación y lectura sólo es conocido por el destinatario del mensaje y se conoce como "separar el grano de la paja".

Los mensajes suelen ir ocultos entre archivos de sonido o imágenes y ser enormemente grandes por la cantidad extra de información enviada (a comparación del mensaje original).



UNIDAD 6

BASES DE DATOS NoSQL o NO RELACIONALES

Llevamos ya varios años en el mundo de la informática empresarial. Hemos visto cambiar muchas cosas en lenguajes, arquitecturas, plataformas y procesos. Pero durante todo este tiempo una cosa se ha mantenido constante: las bases de datos relacionales almacenan los datos. Ha habido desafíos, algunos de los cuales han tenido éxito en algunos nichos, pero en general, la cuestión del almacenamiento de datos para los arquitectos ha sido la cuestión de qué base de datos relacional usar.

Hay mucho valor en la estabilidad de este reinado. Los datos de una organización duran mucho más que sus programas (al menos eso es lo que la gente nos dice, hemos visto muchos programas muy antiguos). Es valioso tener un almacenamiento de datos estable que sea bien entendido y accesible desde muchas plataformas de programación de aplicaciones.

Ahora, sin embargo, hay un nuevo retador en el bloque bajo la etiqueta de confrontación de NoSQL. Nace de la necesidad de manejar grandes volúmenes de datos, lo que obligó a un cambio fundamental a construir grandes plataformas de hardware a través de clústeres de servidores básicos. Esta necesidad también ha generado preocupaciones de larga data sobre las dificultades de hacer que el código de la aplicación funcione bien con el modelo de datos relacionales.

El término “NoSQL” está muy mal definido. Por lo general, se aplica a varias bases de datos no relacionales recientes, como Cassandra, Mongo, Neo4J y Riak. Adoptan datos sin esquema, se ejecutan en clústeres y tienen la capacidad de intercambiar la coherencia tradicional por otras propiedades útiles.

Los defensores de las bases de datos NoSQL afirman que pueden crear sistemas que tienen un mayor rendimiento, se escalan mucho mejor y son más fáciles de programar.

¿Es este el primer toque de difuntos para las bases de datos relacionales, o es otro pretendiente al trono? Nuestra respuesta a eso es "ninguno". Las bases de datos relacionales son una herramienta poderosa que esperamos usar durante muchas décadas más, pero vemos un cambio profundo en el sentido de que las bases de datos relacionales no serán las únicas bases de datos en uso. Nuestra opinión es que estamos entrando en un mundo de persistencia políglota donde las empresas, e incluso las aplicaciones individuales, utilizan múltiples tecnologías para la gestión de datos. Como resultado, los arquitectos deberán estar familiarizados con estas tecnologías y ser capaces de evaluar cuáles usar para diferentes necesidades. Si no hubiéramos pensado en eso, no habríamos invertido el tiempo y el esfuerzo en escribir este apunte.

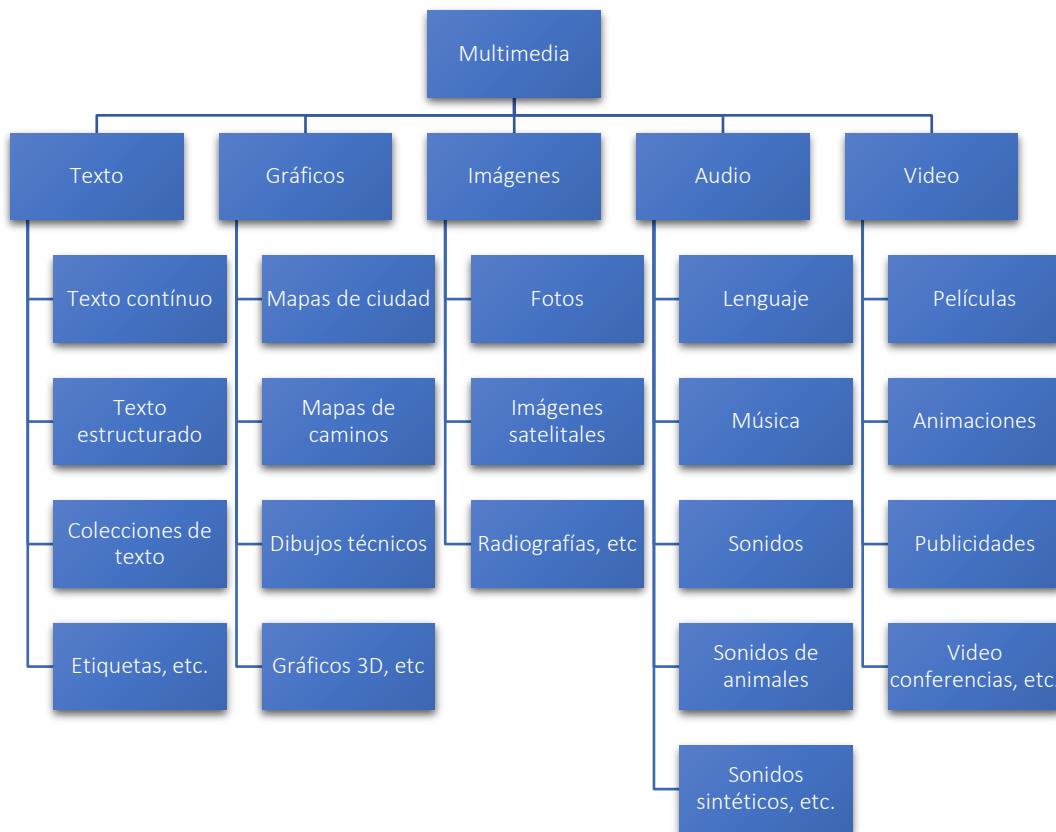
Esta unidad busca brindarte suficiente información para responder a la pregunta de si vale la pena considerar seriamente las bases de datos NoSQL para tus proyectos futuros. Cada proyecto es diferente y no hay forma de que podamos escribir un árbol de decisión simple para elegir el almacén de datos correcto. En cambio, lo que intentamos aquí es brindarte suficiente información sobre cómo funcionan las bases de datos NoSQL, para que puedas hacer esos juicios vos mismo sin tener que rastrear toda la web.

BIG DATA

El término Big Data se utiliza para etiquetar grandes volúmenes de datos que superan los límites del software convencional. Estos datos generalmente no están estructurados y pueden provenir de una amplia variedad de fuentes: publicaciones en redes sociales, correos electrónicos, archivos electrónicos con contenido multimedia, consultas de motores de búsqueda, repositorios de documentos de sistemas de administración de contenido, datos de sensores de varios tipos., evolución de tarifas en bolsas de valores, datos de flujo de tráfico e imágenes



de satélite, contadores inteligentes en electrodomésticos, procesos de pedido, compra y pago en tiendas online, aplicaciones de e-salud, sistemas de monitorización, etc.



Todavía no existe una definición vinculante para Big Data, pero la mayoría de los especialistas en datos estarán de acuerdo en tres v: **volumen** (grandes cantidades de datos), **variedad** (múltiples formatos, datos estructurados, semiestructurados y no estructurados) y **velocidad** (procesamiento de alta velocidad y en tiempo real). El glosario de TI de Gartner Group ofrece la siguiente definición:

Big data son activos de información de gran volumen, alta velocidad y/o gran variedad que exigen formas innovadoras y rentables de procesamiento de la información que permiten una mejor comprensión, toma de decisiones y automatización de procesos.²⁹

Con esta definición, Gartner Group posiciona el Big Data como un activo de información para las empresas.

De hecho, es vital para las empresas y organizaciones generar conocimiento relevante para la toma de decisiones con el fin de sobrevivir. Además de los sistemas de información internos, utilizan cada vez más los numerosos recursos disponibles en línea para anticipar mejor los desarrollos económicos, ecológicos y sociales en los mercados.

Big Data es un desafío que enfrentan no solo las empresas con fines de lucro en los mercados digitales, sino también los gobiernos, las autoridades, las ONG (organizaciones no gubernamentales) y las NPO (organizaciones sin fines de lucro).

²⁹ Gartner Group. Glosario de IT – Big data; <https://www.gartner.com/en/information-technology/glossary/big-data> consultado el 5 de marzo de 2022



Un buen ejemplo son los programas para crear ciudades inteligentes o ciudades ubicuas, es decir, utilizar tecnologías de Big Data en ciudades y aglomeraciones urbanas para el desarrollo sostenible de los aspectos sociales y ecológicos de los espacios de vida humana. Incluyen proyectos que faciliten la movilidad, el uso de sistemas inteligentes para el abastecimiento de agua y energía, la promoción de redes sociales, la ampliación de la participación política, el fomento del emprendimiento, la protección del medio ambiente y el aumento de la seguridad y la calidad de vida.

Todo uso de aplicaciones de Big Data requiere una gestión exitosa de las tres V mencionadas anteriormente:

- **Volumen:** hay cantidades masivas de datos involucrados, que van desde tera a zettabytes (megabyte = 10^6 bytes, gigabyte = 10^9 bytes, terabyte = 10^{12} bytes, petabyte = 10^{15} bytes, exabyte = 10^{18} bytes, zettabyte = 10^{21} bytes).
- **Variedad:** Big Data implica el almacenamiento de datos multimedia estructurados, semiestructurados y no estructurados (texto, gráficos, imágenes, audio y video).
- **Velocidad:** las aplicaciones deben poder procesar y analizar flujos de datos en tiempo real a medida que se recopilan los datos.

Como en la definición de Gartner Group, Big Data puede considerarse un activo de información, por lo que en ocasiones se le añade otra V:

- **Valor:** Las aplicaciones de Big Data están destinadas a aumentar el valor de la empresa, por lo que se realizan inversiones en personal e infraestructura técnica donde aportarán apalancamiento o se podrá generar valor agregado.

Existen numerosas soluciones de código abierto para bases de datos NoSQL, y las tecnologías no requieren hardware costoso, mientras que también ofrecen una buena escalabilidad. Sin embargo, falta personal especializado, ya que la profesión de científico de datos apenas está surgiendo, y la educación profesional en este sector aún se encuentra en su fase piloto o solo en discusión.

Para completar nuestra consideración del concepto de Big Data nos fijaremos en otra V:

- **Veracidad:** dado que muchos datos son vagos o inexactos, se necesitan algoritmos específicos que evalúen la validez y evalúen la calidad de los resultados. Grandes cantidades de datos no significan automáticamente mejores análisis.

La veracidad es un factor importante en Big Data, donde los datos disponibles son de calidad variable, lo que debe tenerse en cuenta en los análisis. Además de los métodos estadísticos, existen métodos difusos de computación blanda que asignan un valor de verdad entre 0 (falso) y 1 (verdadero) a cualquier resultado o declaración (bases de datos difusas).

BASES DE DATOS NOSQL

BASADAS EN MODELOS DE GRAFO

Las bases de datos NoSQL admiten varios modelos de bases de datos. Escogimos bases de datos de grafos como ejemplo para observar y analizar sus características.

Grafo de propiedades Los grafos de propiedades consisten en nodos (conceptos, objetos) y aristas dirigidas (relaciones) que conectan los nodos. Tanto los nodos como las aristas reciben una etiqueta y pueden tener propiedades. Las propiedades se dan como pares de atributo-valor siguiendo el patrón (atributo: valor) con los nombres de los atributos y los respectivos valores.



Un grafo presenta de forma abstracta los nodos y las aristas con sus propiedades. La Ilustración 1 muestra parte de una colección de películas como ejemplo. Contiene los nodos PELÍCULA con atributos Título y Año (de estreno), GÉNERO con el respectivo Tipo (p. ej., crimen, misterio, comedia, drama, suspense, western, ciencia ficción, documental, etc.), ACTOR con Nombre y Año de Nacimiento, y DIRECTOR con Nombre y Nacionalidad.

El ejemplo utiliza tres aristas dirigidas: La arista ACTED_IN muestra qué artista del nodo ACTOR protagonizó qué película del nodo MOVIE. Esta arista también tiene una propiedad, el Rol del actor en la película. Las otras dos aristas, HAS y DIRECTED_BY, van del nodo PELÍCULA al nodo GÉNERO y DIRECTOR, respectivamente.

En el nivel de manifestación, es decir, la base de datos de grafos, el grafo de propiedades contiene los valores concretos (Ilustración 2).

El modelo de grafos de propiedades para bases de datos se basa formalmente en la teoría de grafos. Dependiendo de su madurez, los productos de software relevantes pueden ofrecer algoritmos para calcular las siguientes características:

- **Conectividad:** un grafo está conectado cuando cada nodo del grafo está conectado a todos los demás nodos por al menos una ruta.
- **Ruta más corta:** La ruta más corta entre dos nodos de un grafo es la que tiene menos aristas.
- **Vecino más cercano:** en grafos con aristas ponderadas (por ejemplo, por distancia o tiempo en una red de transporte), los vecinos más cercanos de un nodo se pueden determinar encontrando los intervalos mínimos (ruta más corta en términos de distancia o tiempo).
- **Coincidencia:** la coincidencia en la teoría de grafos significa encontrar un conjunto de aristas que no tienen nodos comunes.

Estas características de grafos son significativas en muchos tipos de aplicaciones. Encontrar el camino más corto o el vecino más cercano, por ejemplo, es de gran importancia en el cálculo de rutas de viaje o transporte. Los algoritmos enumerados también pueden ordenar y analizar las relaciones en las redes sociales por la longitud de la ruta.

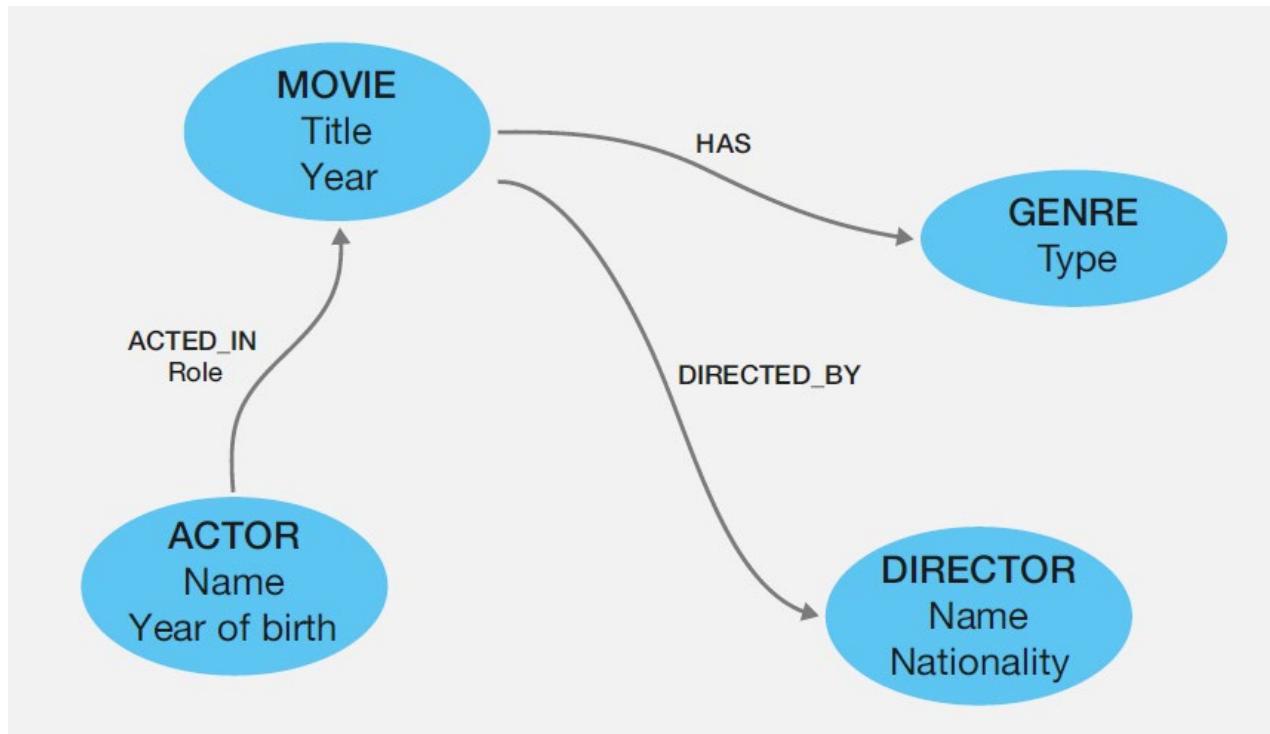


ILUSTRACIÓN 1 SECCIÓN DE UN GRAFO DE PROPIEDADES DE PELÍCULAS

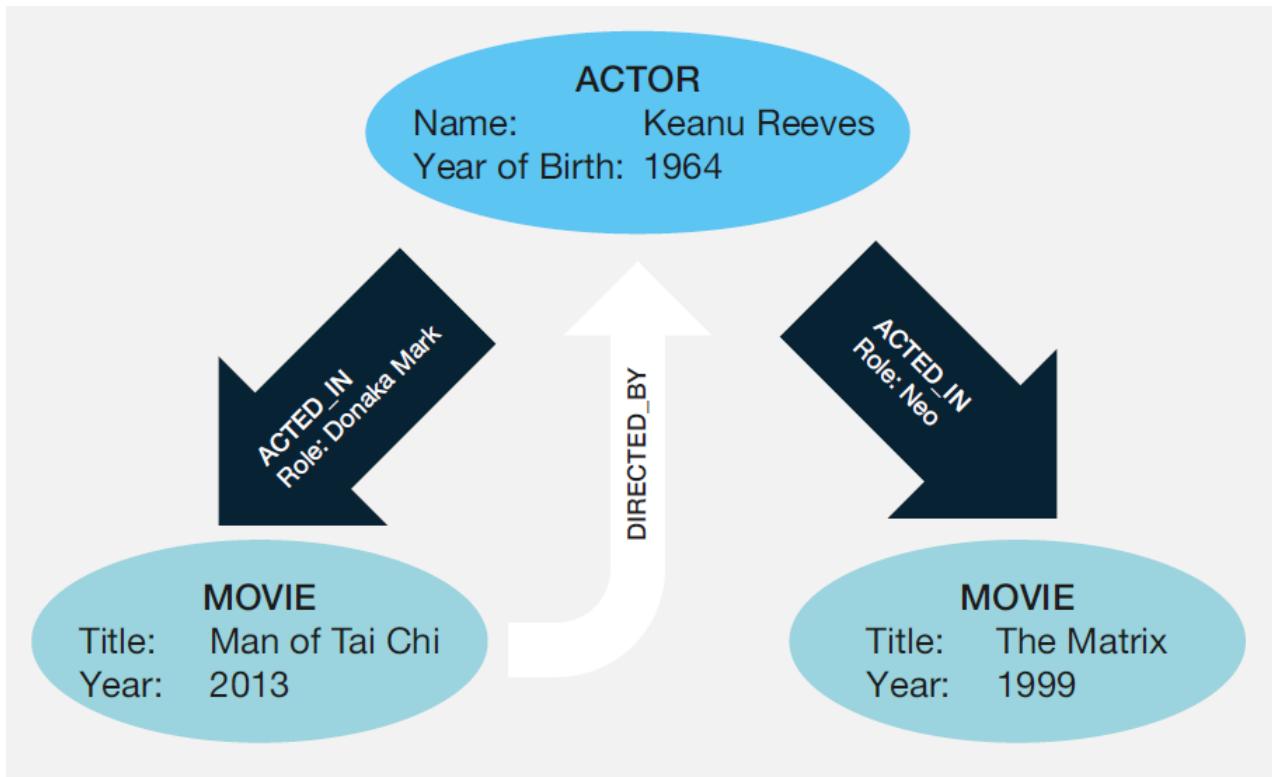


ILUSTRACIÓN 2 SECCIÓN DE UNA BASE DE DATOS DE GRAFOS SOBRE PELÍCULAS

LENGUAJE DE CONSULTA DE GRAFOS CYpher

Cypher es un lenguaje de consulta declarativo para extraer patrones de bases de datos de grafos.

Los usuarios definen su consulta especificando nodos y aristas. A continuación, el sistema de gestión de la base de datos calcula todos los patrones que cumplen los criterios mediante el análisis de las posibles rutas (conexiones entre nodos a través de aristas). En otras palabras, el usuario declara la estructura del patrón deseado y los algoritmos del sistema de administración de la base de datos atraviesan todas las conexiones necesarias (caminos) y ensamblan los resultados.

El modelo de datos de una base de datos de grafos consta de nodos (conceptos, objetos) y aristas dirigidas (relaciones entre nodos). Además de su nombre, tanto los nodos como las aristas pueden tener un conjunto de propiedades. Estas propiedades están representadas por pares de atributo-valor.

La Ilustración 2 muestra un segmento de una base de datos de grafos sobre películas y actores. Para simplificar las cosas, solo se muestran dos tipos de nodos: ACTOR y MOVIE. Los nodos ACTOR contienen dos pares de atributo-valor, específicamente (Nombre: Nombre Apellido) y (Año de nacimiento: Año).

El segmento de la Ilustración 2 incluye diferentes tipos de aristas: La relación ACTED_IN representa qué actores protagonizaron qué películas. Las aristas también pueden tener propiedades si se les agregan pares de atributo-valor. Para la relación ACTED_IN, se enumeran los roles respectivos de los actores en las películas. Por ejemplo, Keanu Reeves fue elegido como el hacker Neo en 'The Matrix'.

Los nodos se pueden conectar mediante múltiples ejes de relación. La película 'Man of Tai Chi' y el actor Keanu Reeves están vinculados no solo por el papel del actor (ACTED_IN), sino también por el puesto de director (DIRECTED_BY). Por lo tanto, el diagrama muestra que Keanu Reeves dirigió la película 'Man of Tai Chi' y la protagonizó como Donaka Mark.



Si queremos analizar esta base de datos de gráficos en películas, podemos usar Cypher. Utiliza los siguientes elementos básicos de consulta:

- MATCH: Especificación de nodos y aristas, así como declaración de patrones de búsqueda.
- WHERE: Condiciones para el filtrado de resultados.
- RETURN: Especificación del resultado de búsqueda deseado, agregado si es necesario

Por ejemplo, la consulta de Cypher para el año en que se estrenó la película 'The Matrix' sería:

```
MATCH (m: Movie {Title: "The Matrix"})  
RETURN m.Year
```

La consulta envía la variable m para la película 'The Matrix' para devolver el año de lanzamiento de la película por m.Year. En Cypher, los paréntesis siempre indican nodos, es decir, (m: Movie) declara la variable de control m para el nodo MOVIE. Además de las variables de control, se pueden incluir pares de atributo-valor individuales entre corchetes. Dado que estamos específicamente interesados en la película 'The Matrix', podemos agregar {Title: "The Matrix"} al nodo (m: Movie).

Las consultas sobre las relaciones dentro de la base de datos de grafos son un poco más complicadas.

Las relaciones entre dos nodos arbitrarios (a) y (b) se expresan en Cypher con el símbolo de flecha “->”, es decir, la ruta de (a) a (b) se declara como “(a)->(b)”. Si la relación específica entre (a) y (b) es importante, la arista [r] se puede insertar en el medio de la flecha. Los corchetes representan aristas, y r es nuestra variable para relaciones

Ahora, si queremos saber quién interpretó a Neo en 'The Matrix', usamos la siguiente consulta para analizar la ruta ACTED_IN entre ACTOR y MOVIE:

```
MATCH (a: Actor)-[: Acted_In {Role: "Neo"}]->  
(: Movie {Title: "The Matrix"})  
RETURN a.Name
```

Cypher devolverá el resultado Keanu Reeves.

Para obtener una lista de títulos de películas (m), nombres de actores (a) y roles respectivos (r), la consulta tendría que ser:

```
MATCH (a: Actor)-[r: Acted_In] -> (m: Movie)  
RETURN m.Title, a.Name, r.Role
```

Dado que nuestra base de datos de grafos de ejemplo solo contiene un actor y dos películas, el resultado sería la película 'Man of Tai Chi' con el actor Keanu Reeves en el papel de Donaka Mark y la película 'The Matrix' con Keanu Reeves como Neo.

En la vida real, sin embargo, una base de datos de grafos de actores, películas y roles tiene innumerables entradas. Por lo tanto, una consulta manejable tendría que permanecer limitada, por ejemplo, al actor Keanu Reeves, y tendría este aspecto:

```
MATCH (a: Actor)-[r: Acted_In] -> (m: Movie)  
WHERE (a.Name = "Keanu Reeves")  
RETURN m.Title, a.Name, r.Role
```

Al igual que SQL, Cypher utiliza consultas declarativas en las que el usuario especifica las propiedades deseadas del patrón de resultados (Cypher) o la tabla de resultados (SQL), y el sistema de gestión de bases de datos respectivo luego calcula los resultados. Sin embargo, el análisis de redes de relaciones, el uso de estrategias de búsqueda recursivas o el análisis de propiedades de gráficos son casi imposibles con SQL.



SISTEMAS DE GESTIÓN DE BASES DE DATOS NoSQL

Antes de la introducción del modelo relacional por parte de Ted Codd, existían bases de datos no relacionales, como bases de datos jerárquicas o similares a redes. Después del desarrollo de los sistemas de gestión de bases de datos relacionales, los modelos no relacionales todavía se usaban en aplicaciones técnicas o científicas. Por ejemplo, ejecutar sistemas CAD (diseño asistido por computadora) para componentes estructurales o de máquinas en tecnología relacional es bastante difícil. Dividir objetos técnicos en una multitud de tablas resultó problemático, ya que las manipulaciones geométricas, topológicas y gráficas tenían que ejecutarse en tiempo real.

La llegada de Internet y numerosas aplicaciones basadas en la web ha dado un gran impulso a la relevancia de los conceptos de datos no relacionales frente a los relacionales, ya que administrar aplicaciones de Big Data con tecnología de bases de datos relacionales es difícil o imposible.

Si bien "no relacional" sería una mejor descripción que NoSQL, este último se ha establecido con los investigadores y proveedores de bases de datos en el mercado durante los últimos años.

NoSQL El término NoSQL ahora se usa para cualquier enfoque de gestión de datos no relacional que cumpla con dos criterios:

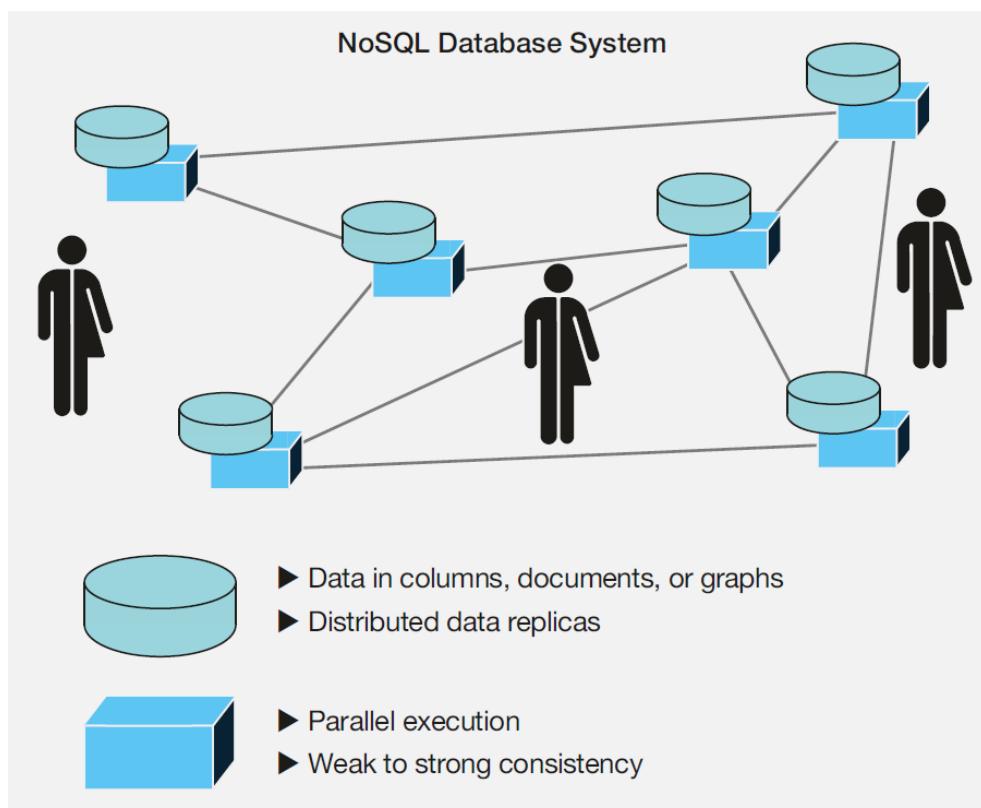
En primer lugar: Los datos no se almacenan en tablas.

En segundo lugar: el lenguaje de la base de datos no es SQL.

NoSQL también se interpreta a veces como 'No solo SQL' para expresar que otras tecnologías además de la tecnología de datos relacionales se utilizan en aplicaciones web distribuidas masivamente.

Las tecnologías NoSQL son especialmente necesarias si el servicio web requiere alta disponibilidad. Veremos ejemplos en los que se utilizan varias tecnologías NoSQL además de una base de datos relacional.

La estructura básica de un sistema de gestión de base de datos NoSQL puede verse en la siguiente figura:





Los sistemas de administración de bases de datos NoSQL utilizan principalmente una arquitectura de almacenamiento distribuida masivamente.

Los datos reales se almacenan en pares clave-valor, columnas o familias de columnas, almacenes de documentos o grafos.

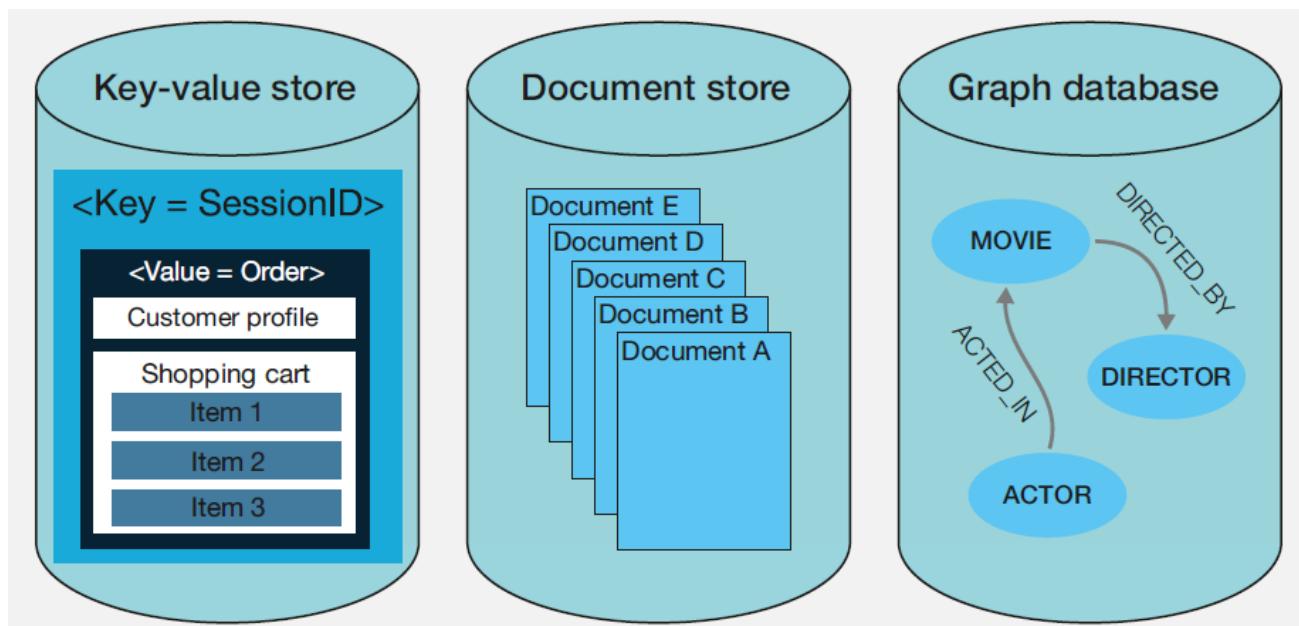


ILUSTRACIÓN 3 DIFERENTES BASES NOSQL

Para garantizar una alta disponibilidad y evitar interrupciones en los sistemas de base de datos NoSQL, se admiten varios conceptos de redundancia (como el "hashing consistente").

La arquitectura replicada y distribuida masivamente también permite análisis paralelos ("MapReduce"). Especialmente los análisis de grandes volúmenes de datos o la búsqueda de información específica pueden acelerarse significativamente con procesos de computación distribuida.

En el método map/reduce, las subtareas se delegan a varios nodos de computación y se extraen pares clave-valor simples (map), luego los resultados parciales se agregan y devuelven (reduce).

También existen modelos de consistencia múltiple o redes informáticas distribuidas masivamente. Una coherencia sólida significa que el sistema de gestión de bases de datos NoSQL garantiza una coherencia total en todo momento. Los sistemas con coherencia débil toleran que los cambios se copien en los nodos replicados con un retraso, lo que genera incoherencias temporales. Es posible una mayor diferenciación, por ejemplo, consistencia por quórum.

La siguiente definición de bases de datos NoSQL está guiada por el Archivo NoSQL³⁰ en la web:

Sistema de base de datos NoSQL: Los sistemas de almacenamiento basados en web se consideran sistemas de base de datos NoSQL si cumplen los siguientes requisitos:

- **Modelo:** el modelo de base de datos subyacente no es relacional.
- **Al menos tres V:** el sistema de base de datos incluye una gran cantidad de datos (volumen), estructuras de datos flexibles (variedad) y procesamiento en tiempo real (velocidad).

³⁰ NoSQL Archive; <http://nosql-database.org/>, consultado el 17 de febrero de 2015



- **Esquema:** el sistema de administración de la base de datos no está sujeto a un esquema de base de datos fijo.
- **Arquitectura:** la arquitectura de la base de datos admite aplicaciones web de distribución masiva y escalado horizontal.
- **Replicación:** el sistema de administración de bases de datos admite la replicación de datos.
- **Garantía de coherencia:** De acuerdo con el teorema CAP, la coherencia puede garantizarse con un retraso para priorizar la alta disponibilidad y la tolerancia a la partición.

Los investigadores y operadores de NoSQL Archive enumeran más de 225 productos de base de datos NoSQL en su sitio web, la mayoría de ellos de código abierto. Sin embargo, la multitud de soluciones indica que el mercado de productos NoSQL aún no es completamente seguro.

Además, la implementación de tecnologías NoSQL adecuadas requiere especialistas que conozcan no solo los conceptos subyacentes, sino también los diversos enfoques y herramientas de arquitectura.

La Ilustración 3 muestra tres sistemas de gestión de bases de datos NoSQL diferentes.

Los *almacenes de clave-valor* son la versión más simple. Los datos se almacenan como una clave de identificación <clave = “clave”> y una lista de valores <valor = “valor 1”, “valor 2”, ...>. Un buen ejemplo es una tienda online con gestión de sesiones y cesta de la compra. El ID de sesión es la clave de identificación; los artículos individuales de la cesta se almacenan como valores además del perfil del cliente.

En los almacenes de documentos, los registros se gestionan como documentos dentro de la base de datos NoSQL. Estos documentos son archivos de texto estructurados, por ejemplo, en formato JSON o XML, que se pueden buscar mediante una clave única o atributos dentro de los documentos. A diferencia de los almacenes de clave-valor, los documentos tienen cierta estructura; sin embargo, no tiene esquemas, es decir, las estructuras de los registros individuales (documentos) pueden variar.

El tercer ejemplo vuelve a visitar la base de datos de grafos sobre películas y actores.

CYPHER

La base de datos de grafos Neo4J³¹ utiliza el lenguaje Cypher como lenguaje para ingresar comandos para interactuar con la base de datos. Cypher se basa en un mecanismo de coincidencia de patrones.

Similar a SQL, Cypher tiene comandos de lenguaje para consultas de datos y manipulación de datos (lenguaje de manipulación de datos, DML); sin embargo, la definición del esquema en Cypher se realiza de forma implícita, es decir, los tipos de nodos y aristas se definen insertando instancias de ellos en la base de datos como nodos y aristas específicos reales.

El lenguaje de definición de datos (DDL) de Cypher solo puede describir índices, restricciones únicas y estadísticas. Cypher no incluye elementos lingüísticos directos para los mecanismos de seguridad, para los cuales los lenguajes relacionales tienen declaraciones, como GRANT y REVOKE.

A continuación, veremos más de cerca el lenguaje Cypher; todos los ejemplos se refieren al conjunto de datos Northwind³².

Cypher tiene tres comandos básicos:

- MATCH para definir patrones de búsqueda

³¹ <https://neo4j.com/>

³² <https://neo4j.com/developer/guide-importing-data-and-etl/>



- WHERE para condiciones para filtrar los resultados
- RETURN para generar propiedades, vértices, relaciones o caminos.

La cláusula RETURN puede generar vértices o tablas de propiedades. Los siguientes ejemplos devuelven el nodo con el nombre de producto Chocolade:

```
MATCH (p:Product)
WHERE p.productName = "Chocolade"
RETURN p
```

The screenshot shows the Neo4j Browser interface. On the left, there's a sidebar with icons for Graph, Table, Test, and Code. The main area displays a single node highlighted with a blue border. The node has the label 'Product' and the name 'Chocolade'. To the right, the 'Node Properties' panel is open, showing the following properties for the node:

Property	Value
<id>	47
categoryID	3
discontinued	false
productID	48
productName	Chocolade
quantityPerUnit	10 pkgs.
reorderLevel	25
supplierID	22
unitPrice	12.75
unitsInStock	15
unitsOnOrder	70

El retorno de nodos completos es similar a 'SELECT *' en SQL. Cypher también puede devolver propiedades como valores de atributos de nodos y aristas en forma de tablas:

```
MATCH (p:Product)
WHERE p.unitPrice > 55
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice
```

The screenshot shows the Neo4j Browser interface with a table result. The table has two columns: 'p.productName' and 'p.unitPrice'. The data is as follows:

p.productName	p.unitPrice
"Camarón Tigres"	62.5
"Sir Rodney's Marmalade"	81.0
"Mishi Kobe Niku"	97.0
"Thüringer Rostbratwurst"	123.79
"Côte de Blaye"	263.5

At the bottom of the results, it says: 'Started streaming 5 records after 6 ms and completed after 9 ms.'

Esta consulta incluye una selección, una proyección y una ordenación. La cláusula MATCH define una coincidencia de patrones que filtra el grafo para el nodo del tipo 'Producto'; la cláusula WHERE selecciona todos los productos con un precio superior a 55; y la cláusula RETURN proyecta esos nodos en las propiedades nombre y precio del producto, con la cláusula ORDER BY ordenando los productos por precio.

El producto cartesiano de dos nodos se puede generar en Cypher con la siguiente sintaxis:

```
MATCH (p:Product), (c:Category)
RETURN p.productName, c.categoryName
```



```
neo4j$ MATCH (p:Product), (c:Category) RETURN p.productName, c.categoryName
```

p.productName	c.categoryName
1 "Chai"	"Beverages"
2 "Chang"	"Beverages"
3 "Aniseed Syrup"	"Beverages"
4 "Chef Anton's Cajun Seasoning"	"Beverages"
5 "Chef Anton's Gumbo Mix"	"Beverages"
6 "Grandma's Boysenberry Spread"	"Beverages"

Started streaming 616 records after 6 ms and completed after 8 ms.

Este comando enumera todas las combinaciones posibles de nombres de productos y nombres de categorías. Sin embargo, las uniones de nodos, es decir, las selecciones en el producto cartesiano se ejecutan en base a grafos haciendo coincidir los patrones de ruta por tipos de arista:

```
MATCH (p:Product) -[:PART_OF]-> (c:Category)  
RETURN p.productName, c.categoryName
```

```
neo4j$ MATCH (p:Product) -[:PART_OF]-> (c:Category) RETURN p.productName, c.categoryName
```

p.productName	c.categoryName
1 "Sasquatch Ale"	"Beverages"
2 "Outback Lager"	"Beverages"
3 "Laughing Lumberjack Lager"	"Beverages"
4 "Steeleye Stout"	"Beverages"
5 "Côte de Blaye"	"Beverages"
6 "Lakkaliköör"	"Beverages"

Started streaming 77 records after 6 ms and completed after 6 ms.

Para cada producto, esta consulta enumera la categoría a la que pertenece, considerando solo los nodos de producto y categoría conectados por aristas del tipo PART_OF. Esto es igual al INNER JOIN del tipo de nodo 'Producto' con el tipo de nodo 'Categoría' a través del tipo de arista PART_OF.

Hay tipos de nodos en los que solo un subconjunto de los nodos tiene una arista de un tipo de arista específico. Por ejemplo, no todos los empleados tienen subordinados, es decir, solo un subconjunto de los nodos del tipo 'Empleado' tiene una arista de tipo REPORTS_TO entrante.

Entonces, ¿qué pasa si el usuario quiere generar una lista de todos los empleados junto con el número de subordinados para cada empleado, incluso si ese número es cero? Una consulta `MATCH (e:Employee) <- [:REPORTS_TO]-(sub)` solo devolvería aquellos empleados que realmente tienen subordinados, es decir, donde el número es mayor que cero:

```
MATCH (e:Employee) <- [:REPORTS_TO]-(sub)  
RETURN e.employeeID, count(sub.employeeID)
```



neo4j\$ MATCH (e:Employee) < -[:REPORTS_TO]-(sub) RETURN e.employeeID, count(sub.employeeID)		
	e.employeeID	count(sub.employeeID)
1	"2"	5
2	"5"	3

Started streaming 2 records in less than 1 ms and completed after 1 ms.

Una cláusula OPTIONAL MATCH permite enumerar a todos los empleados, incluidos aquellos sin subordinados:

```
MATCH (e:Employee)
OPTIONAL MATCH (e) < -[:REPORTS_TO]-(sub)
RETURN e.employeeID, count(sub.employeeID)
```

neo4j\$ MATCH (e:Employee) OPTIONAL MATCH (e) < -[:REPORTS_TO]-(sub) RETURN e.employeeID, count(sub.employeeID)		
	e.employeeID	count(sub.employeeID)
1	"1"	0
2	"2"	5
3	"3"	0
4	"4"	0
5	"5"	3
6	"6"	0

Started streaming 9 records after 7 ms and completed after 9 ms.

Al igual que SQL, Cypher tiene operadores, funciones integradas y funciones agregadas. La siguiente consulta devuelve el nombre completo y la inicial del apellido de cada empleado, junto con el número de subordinados:

```
MATCH (e:Employee)
OPTIONAL MATCH (e) < -[:REPORTS_TO]-(sub)
RETURN
e.firstName + " "
+ left(e.lastName, 1) + "." as name,
count(sub.employeeID)
```

neo4j\$ MATCH (e:Employee) OPTIONAL MATCH (e) < -[:REPORTS_TO]-(sub) RETURN e.firstName + " " + left(e.lastName, 1) + "." as name, count(sub.employeeID)		
	name	count(sub.employeeID)
1	"Nancy D."	0
2	"Andrew F."	5
3	"Janet L."	0
4	"Margaret P."	0
5	"Steven B."	3
6	"Michael S."	0

Started streaming 9 records after 9 ms and completed after 12 ms.



El operador + se puede usar en valores de datos de tipo 'texto' para concatenarlos. La función incluida LEFT devuelve los primeros n caracteres de un texto. Finalmente, el agregado 'Count' determina el número de nodos para una combinación de valores de datos en la instrucción RETURN que existen en el conjunto de la instrucción MATCH.

```
MATCH (e:Employee)
OPTIONAL MATCH (e) < -[:REPORTS_TO]-(sub)
RETURN
e.firstName + " "
+ left(e.lastName, 1) + "." as name,
collect(sub.employeeID)
```

	name	collect(sub.employeeID)
1	"Nancy D."	[]
2	"Andrew F."	["4", "8", "5", "1", "3"]
3	"Janet L."	[]
4	"Margaret P."	[]
5	"Steven B."	["6", "9", "7"]
6	"Michael S."	[]

Started streaming 9 records after 11 ms and completed after 12 ms.

A diferencia de SQL, los agregados en Cypher no requieren una cláusula GROUP BY. Los agregados adicionales, similares a SQL, son total (sum), mínimo (min) y máximo (max). Un agregado no atómico útil es 'collect', que genera una matriz a partir de valores existentes.

La expresión anterior, por ejemplo, enumera todos los empleados con su nombre y la inicial de su apellido, así como una lista de los números de empleado de sus subordinados (que puede estar vacía).

La definición de esquema en Cypher, a diferencia de SQL, se realiza implícitamente, es decir, las clases de datos abstractos (metadatos), como los tipos o atributos de nodos y aristas, se crean usándolos en la inserción de valores de datos concretos. El siguiente ejemplo inserta nuevos datos en la base de datos:

```
CREATE
(p:Product {
productName:"SQL & NoSQL Databases",
year:2016})
-[:PUBLISHER]->
(o:Organization {
name:"Springer Vieweg"})
```

Esta expresión amerita un análisis más profundo porque aquí suceden múltiples cosas implícitamente. Se crean y conectan dos nuevos nodos, uno para el producto “Bases de datos SQL y NoSQL” y otro para el editor “Springer Vieweg”. Esto implica la generación implícita del nuevo tipo de nodo “Organization” que antes no existía. Esos nodos reciben valores de datos en forma de pares de atributo-valor ingresados en los nodos. Tanto los atributos “year” como “name” no existían antes y, por lo tanto, se agregan al esquema implícitamente; en SQL, esto requeriría un comando CREATE TABLE y ALTER TABLE.

Además, se crea una arista con el tipo “PUBLISHER” entre los nodos del libro y del editor, agregando no solo la arista en sí, sino también ese tipo de arista al esquema de la base de datos.



Las cláusulas SET se utilizan para cambiar los valores de datos que coinciden con un patrón específico. La expresión del siguiente ejemplo establece un nuevo precio para el producto “Chocolate”:

```
MATCH (p:Product)
WHERE p.productName = "Chocolade"
SET p.unitPrice = 13.75
```

Con DELETE, es posible eliminar nodos y aristas como se especifica. Dado que las bases de datos de grafos garantizan la integridad referencial, los vértices solo se pueden eliminar si no tienen aristas adjuntas. Antes de poder eliminar un nodo, el usuario, por lo tanto, debe eliminar todas las aristas entrantes y salientes.

A continuación, se muestra una expresión que primero reconoce todas las aristas conectadas al producto “Tunnbröd”, luego elimina esas aristas y finalmente elimina el nodo del producto en sí.

```
MATCH
()-[r1]->(p:Product),
(p)-[r2]->()
WHERE p.productName = "Tunnbröd"
DELETE r1, r2, p
```

The screenshot shows the Neo4j browser interface. The code input field contains the Cypher query for deleting the 'Tunnbröd' product node and its relationships. The results pane shows the command executed: 'neo4j\$ MATCH ()-[r1]->(p:Product), (p)-[r2]->() WHERE p.productName = "Tunnbröd" DELETE r1, r2, p'. Below the results, a message indicates 'Deleted 1 node, deleted 22 relationships, completed after 8 ms.'

Cabe señalar aquí que, si bien Cypher ofrece algunas funciones para analizar rutas dentro de gráficos (incluido el casco de Kleene para tipos de aristas), no es compatible con la gama completa de álgebra de Kleene para rutas en grafos, como se requiere en lenguajes basados en la teoría de grafos.

Sin embargo, Cypher es un lenguaje muy adecuado para uso práctico.



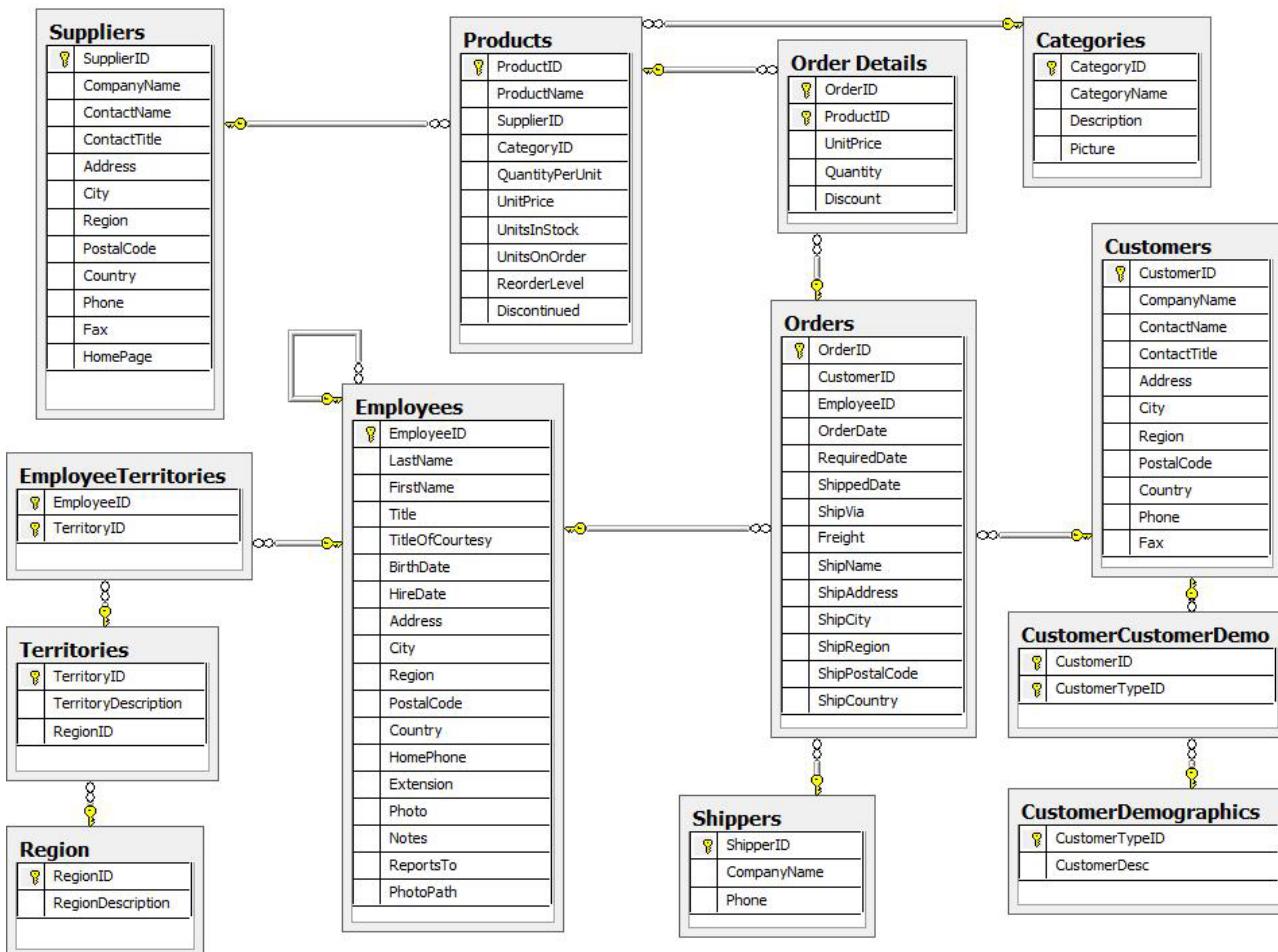
NORTHWIND EN NEO4J

Esta guía le enseñará el proceso para exportar datos desde una base de datos relacional e importarlos a una base de datos de grafos (Neo4j). Aprenderá a tomar datos del sistema relacional y al grafo mediante la traducción del esquema y el uso de herramientas de importación. Este tutorial utiliza un conjunto de datos específico, pero los principios de este tutorial se pueden aplicar y reutilizar con cualquier dominio de datos.

ACERCA DEL DOMINIO DE DATOS

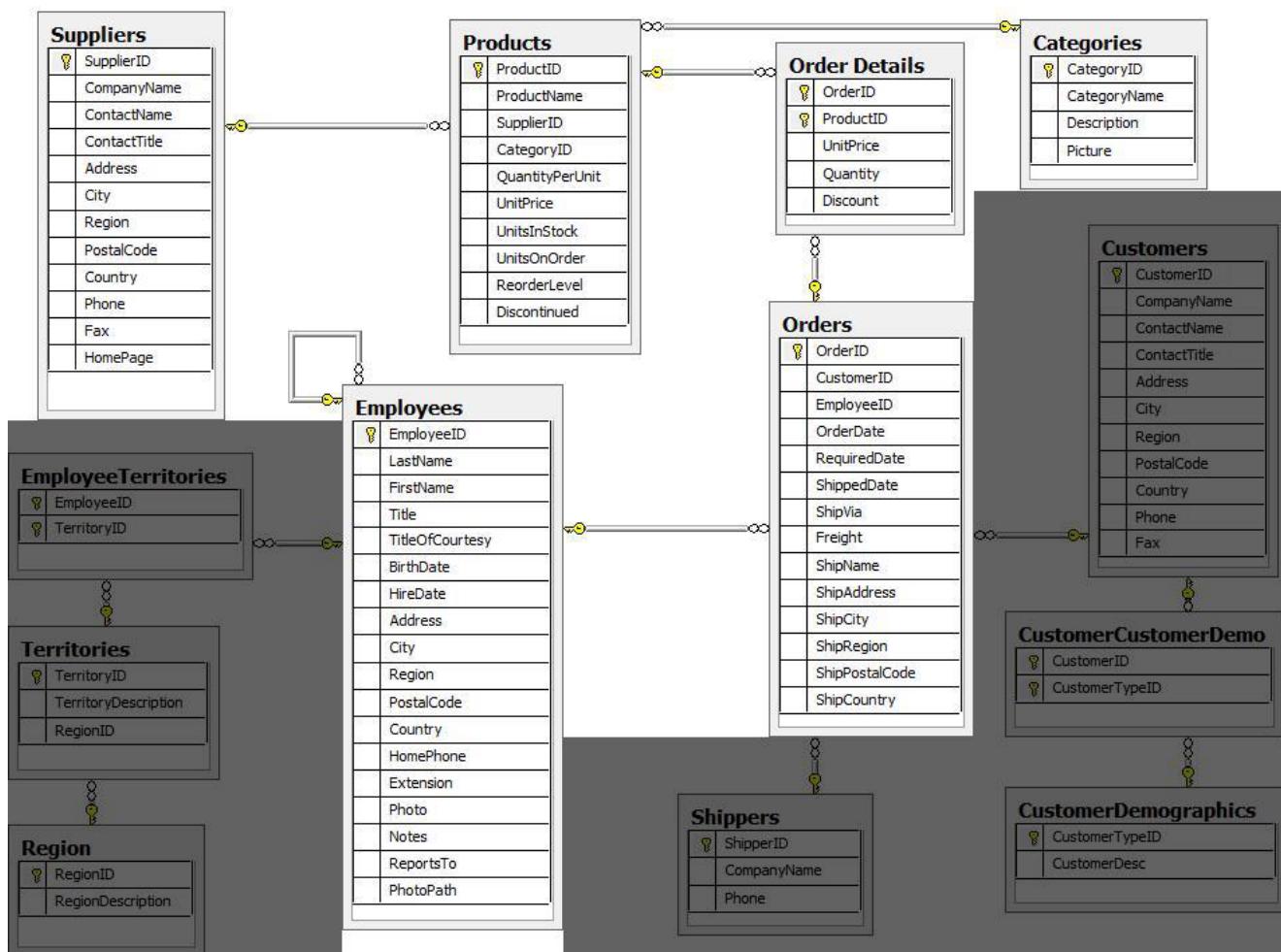
En esta guía, utilizaremos el conjunto de datos NorthWind, un conjunto de datos SQL de uso frecuente. Estos datos representan un sistema de venta de productos: almacenamiento y seguimiento de clientes, productos, pedidos de clientes, existencias en almacén, envíos, proveedores e incluso empleados y sus territorios de ventas. Aunque el conjunto de datos de NorthWind se usa a menudo para demostrar SQL y bases de datos relacionales, los datos también se pueden estructurar como un grafo.

A continuación, se muestra un diagrama entidad-relación (ERD) del conjunto de datos Northwind.



Primero, este es un modelo bastante grande y detallado. Podemos reducir esto un poco para nuestro ejemplo y elegir las entidades que son más críticas para nuestro grafo; en otras palabras, aquellas que podrían beneficiarse más al ver las conexiones. Para nuestro caso de uso, realmente queremos optimizar las relaciones con los pedidos: qué productos estaban involucrados (con las categorías y proveedores de esos productos), qué empleados trabajaron en ellos y los jefes de esos empleados.

Usando estos requisitos comerciales, podemos reducir nuestro modelo a estas entidades esenciales.



DESARROLLO DE UN MODELO DE GRAFOS

Lo primero que deberá hacer para obtener datos de una base de datos relacional en un gráfico es traducir el modelo de datos relacionales a un modelo de datos de gráfico. Determinar cómo desea estructurar tablas y filas como nodos y relaciones puede variar según lo que sea más importante para sus necesidades comerciales.

Al derivar un modelo de grafos a partir de un modelo relacional, debe tener en cuenta un par de pautas generales.

1. Una fila es un nodo.
2. Un nombre de tabla es un nombre de etiqueta.
3. Una combinación o clave foránea es una relación.

Con estos principios en mente, podemos mapear nuestro modelo relacional a un grafo con los siguientes pasos:

FILAS A NODOS, NOMBRES DE TABLA A ETIQUETAS

1. Cada fila de nuestra tabla Orders se convierte en un nodo en nuestro grafo con Order como etiqueta.
2. Cada fila de nuestra tabla Products se convierte en un nodo con Product como etiqueta.
3. Cada fila de nuestra tabla Suppliers se convierte en un nodo con Supplier como etiqueta.
4. Cada fila de nuestra tabla Categories se convierte en un nodo con Category como etiqueta.
5. Cada fila de nuestra tabla Employees se convierte en un nodo con Employee como etiqueta.

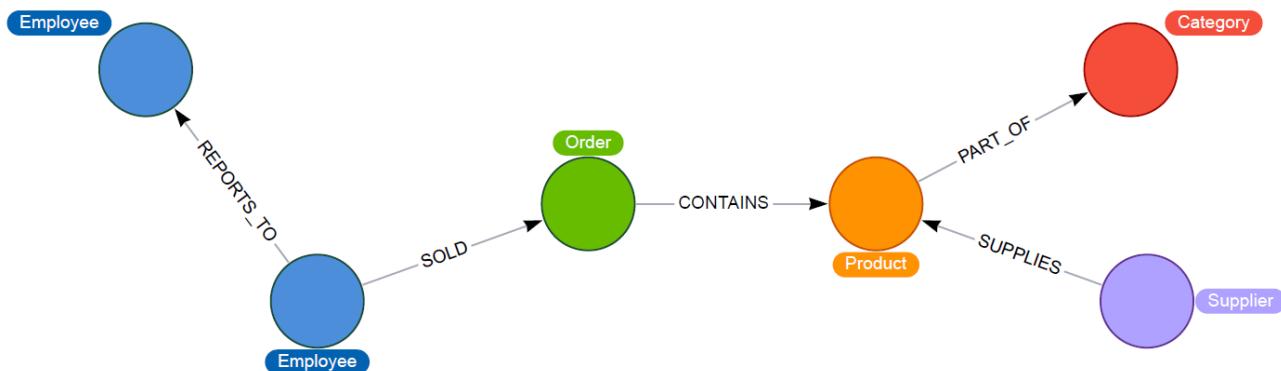
JOINS A RELACIONES

1. Join entre Suppliers y Products se convierte en una relación nombrada SUPPLIES (donde el proveedor suministra el producto).



2. Join entre Products y Categories se convierte en una relación nombrada PART_OF (donde el producto es parte de una categoría).
3. Join entre Employees y Orders se convierte en una relación nombrada SOLD (donde el empleado vendió un pedido).
4. Join entre Employees y sí mismo (relación unaria) se convierte en una relación nombrada REPORTS_TO (donde los empleados tienen un gerente).
5. Join con la tabla (Order Details) entre Orders y Products se convierte en una relación nombrada CONTAINS con propiedades unitPrice, quantity y discount (donde order contiene un producto).

Si dibujamos nuestra traducción en la pizarra, tenemos este modelo de datos de grafos



Ahora, podemos, por supuesto, decidir que queremos incluir el resto de las entidades de nuestro modelo relacional, pero por ahora, nos limitaremos a este modelo de grafo más pequeño.

¿EN QUÉ SE DIFERENCIA EL MODELO GRAFO DEL MODELO RELACIONAL?

No hay nulos. Las entradas de valor no existentes (propiedades) simplemente no están presentes.

Describe las relaciones con más detalle. Por ejemplo, sabemos que un empleado vendió (SOLD) un pedido en lugar de tener una relación de clave foránea entre las tablas Order y Employees. También podríamos optar por agregar más metadatos sobre esa relación, si así lo deseamos.

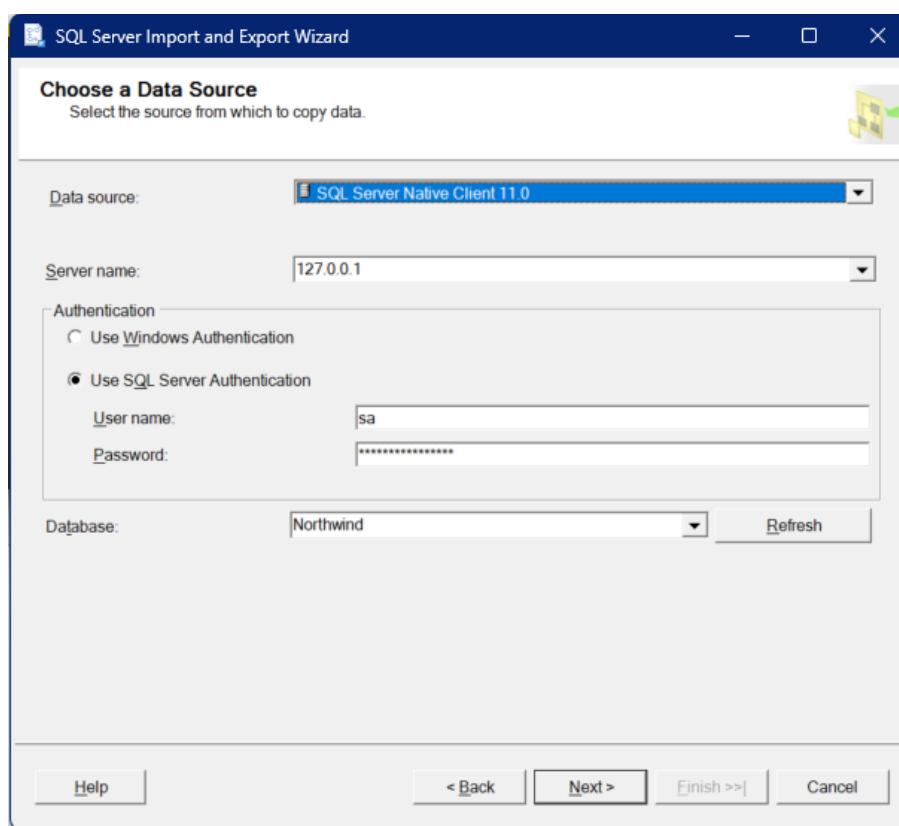
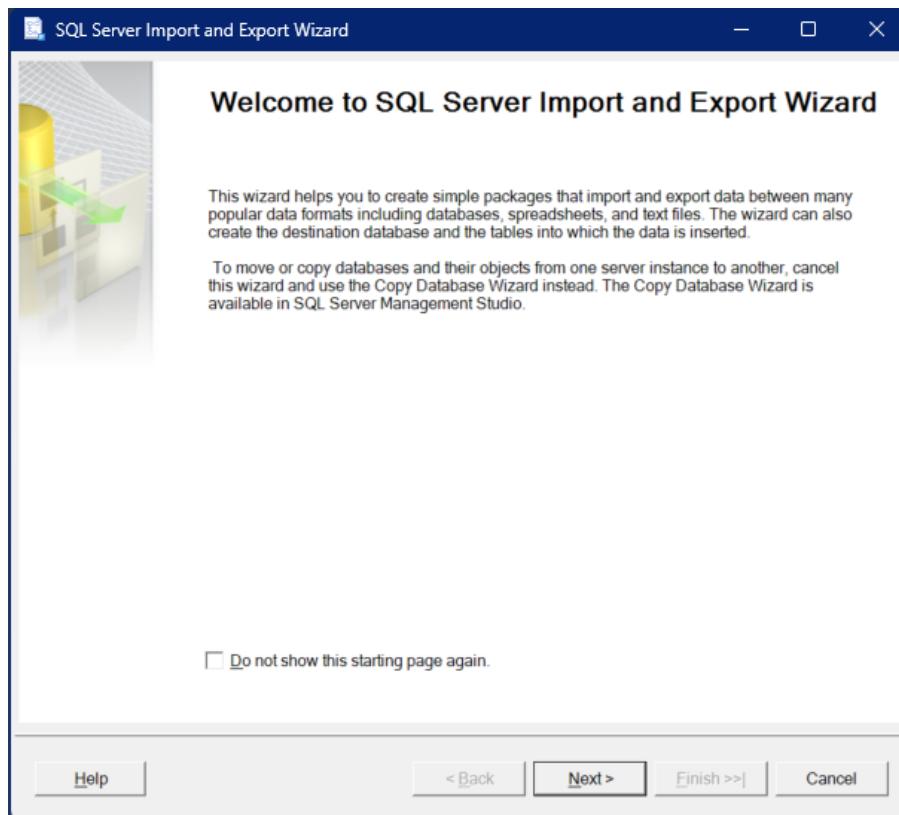
Cualquiera de los modelos puede ser más normalizado. Por ejemplo, las direcciones se han desnormalizado en varias de las tablas, pero podrían haber estado en una tabla separada. En una versión futura de nuestro modelo de grafos, también podríamos optar por separar las direcciones de las entidades Order (o Supplier o Employee) y crear nodos Address separados.

EXPORTACIÓN DE TABLAS RELACIONALES A CSV

Afortunadamente, este paso ya se ha realizado con los datos de Northwind que usará más adelante en esta guía.

Sin embargo, si está trabajando con otro dominio de datos, deberá tomar los datos de las tablas relacionales y ponerlos en otro formato para cargarlos en el grafo. Un formato común que muchos sistemas pueden manejar es un archivo plano de valores separados por comas (CSV).

Para exportar las tablas, vamos a utilizar la utilidad de exportación de datos en SQL Management Studio.





SQL Server Import and Export Wizard

Choose a Destination

Specify where to copy data to.

Destination: **Flat File Destination**

Select a file and specify the file properties and the file format.

File name: **C:\Temp\customers.csv**

Locale: **English (United Kingdom)** Unicode

Code page: **1252 (ANSI - Latin I)**

Format: **Delimited**

Text qualifier: **<none>**

Column names in the first data row

Help | < Back | **Next >** | Finish >> | Cancel

SQL Server Import and Export Wizard

Configure Flat File Destination

Source table or view: **[dbo].[Customers]**

Specify the characters that delimit the destination file:

Row delimiter: **{(CR)}{(LF)}**

Column delimiter: **Comma {,}**

Help | < Back | **Next >** | Finish >> | Cancel

Y repetimos el proceso por cada tabla.

Nota: no necesita ejecutar esta exportación a menos que desee ejecutarlo en su propio RDBMS de Northwind.



IMPORTAR LOS DATOS USANDO CYPHER

Utilizará el comando LOAD CSV de Cypher para transformar el contenido del archivo CSV en una estructura de grafos.

Cuando usa LOAD CSV para crear nodos y relaciones en la base de datos, tiene dos opciones para dónde residen los archivos CSV:

- En la carpeta de importación para la instancia de Neo4j que puede administrar.
- Desde una ubicación disponible públicamente, como un repositorio de S3 o una ubicación de github. Debe utilizar esta opción si utiliza Neo4j AuraDB o Neo4j Sandbox.

Si desea utilizar los archivos CSV para la instancia de Neo4j que administra, puede copiar los archivos CSV desde northwind zip desde github y colocarlos en la carpeta de importación de su Neo4j DBMS.

Ya hemos colocado estos archivos CSV en Gihub para que pueda acceder a ellos.

Utiliza la sentencia LOAD CSV de Cypher para leer cada archivo y agrega cláusulas de Cypher después para tomar los datos de fila/columna y transformarlos en el grafo.

A continuación, ejecutará el código Cypher para:

1. Cargue los nodos desde los archivos CSV.
2. Cree los índices y la restricción para los datos en el grafo.
3. Cree las relaciones entre los nodos.

CREACIÓN DE NODOS ORDER

Ejecute este bloque Cypher para crear los nodos de pedido en la base de datos:

```
// Create orders
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/orders.csv'
AS row
MERGE (order : Order {orderID: row.OrderID})
ON CREATE SET order.shipName = row.ShipName;
```

Si ha colocado los archivos CSV en la carpeta de importación, debe usar esta sintaxis de código para cargar los archivos CSV desde un directorio local:

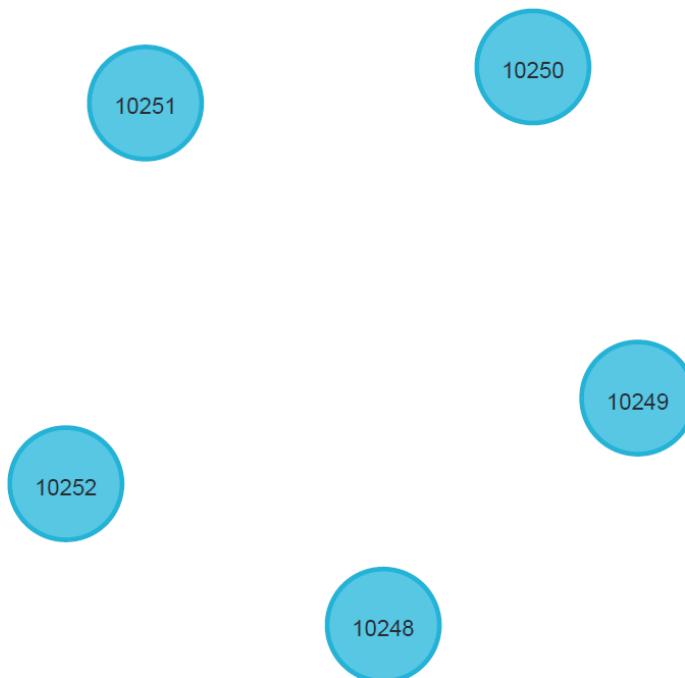
```
// Create orders
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
MERGE (order:Order {orderID: row.OrderID})
ON CREATE SET order.shipName = row.ShipName;
```

Este código crea 830 nodos de pedido en la base de datos.

Puede ver algunos de los nodos en la base de datos ejecutando este código:

```
MATCH (o: Order) return o LIMIT 5 ;
```

La vista del grafo es:



La vista de tabla contiene estos valores para las propiedades del nodo:

"o"
{"orderID": "10248", "shipName": "Vins et alcools Chevalier"}
{"orderID": "10249", "shipName": "Toms Spezialitäten"}
{"orderID": "10250", "shipName": "Hanari Carnes"}
{"orderID": "10251", "shipName": "Victuailles en stock"}
{"orderID": "10252", "shipName": "Suprêmes délices"}

Es posible que observe que no ha importado todas las columnas de campo en el archivo CSV. Con sus declaraciones, puede elegir qué propiedades se necesitan en un nodo, cuáles se pueden omitir y cuáles podrían necesitar importarse a otro tipo de nodo o relación. También puede notar que usó la palabra clave MERGE, en lugar de CREATE. Aunque estamos bastante seguros de que no hay duplicados en nuestros archivos CSV, podemos usar MERGE como una buena práctica para garantizar entidades únicas en nuestra base de datos.

CREACIÓN DE NODOS DE PRODUCTOS

Ejecute este código para crear los nodos de Producto en la base de datos:

```
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/products.csv'
AS row
MERGE (product:Product {productID: row.ProductID})
ON CREATE SET product.productName = row.ProductName, product.unitPrice = toFloat
(row.UnitPrice) ;
```



Este código crea 77 nodos de Producto en la base de datos.

Puede ver algunos de estos nodos en la base de datos ejecutando este código:

```
MATCH (p: Product) return p LIMIT 5 ;
```

La vista del grafo es:



La vista de tabla contiene estos valores para las propiedades del nodo:

"p"
{"unitPrice":18.0,"productID":"1","productName":"Chai"}
{"unitPrice":19.0,"productID":"2","productName":"Chang"}
{"unitPrice":10.0,"productID":"3","productName":"Aniseed Syrup"}
{"unitPrice":22.0,"productID":"4","productName":"Chef Anton's Cajun Seasoning"}
{"unitPrice":21.35,"productID":"5","productName":"Chef Anton's Gumbo Mix"}

CREACIÓN DE NODOS DE PROVEEDORES

Ejecute este código para crear los nodos de proveedor en la base de datos:

```
// Create suppliers
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/suppliers.csv' AS row
MERGE (supplier:Supplier {supplierID: row.SupplierID})
ON CREATE SET supplier.companyName = row.CompanyName;
```

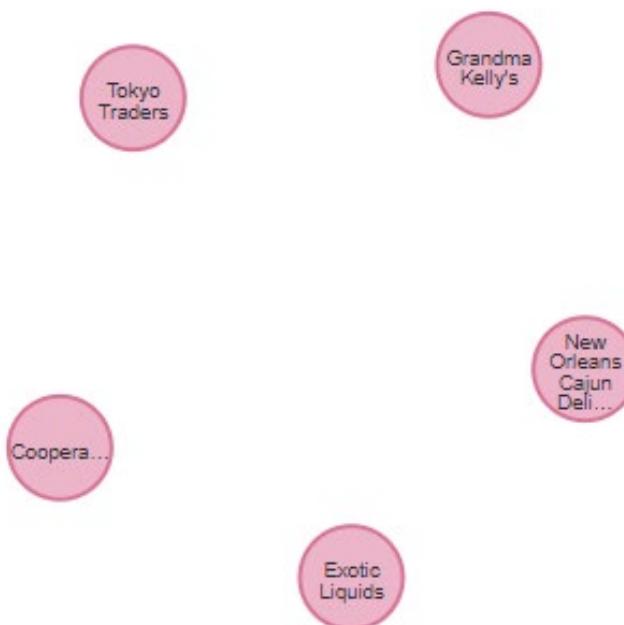


Este código crea 29 nodos de proveedores en la base de datos.

Puede ver algunos de estos nodos en la base de datos ejecutando este código:

```
MATCH (s:Supplier) return s LIMIT 5;
```

La vista del grafo es:



La vista de tabla contiene estos valores para las propiedades del nodo:

"s"
{"supplierID": "1", "companyName": "Exotic Liquids"}
{"supplierID": "2", "companyName": "New Orleans Cajun Delights"}
{"supplierID": "3", "companyName": "Grandma Kelly's Homestead"}
{"supplierID": "4", "companyName": "Tokyo Traders"}
{"supplierID": "5", "companyName": "Cooperativa de Quesos 'Las Cabras'"}

CREACIÓN DE NODOS DE EMPLEADOS

Ejecute este código para crear los nodos de proveedor en la base de datos:

```
// Create employees
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/employees.csv' AS row
MERGE (e:Employee {employeeID:row.EmployeeID})
    ON CREATE SET e.firstName = row.FirstName, e.lastName = row.LastName, e.title =
row.Title;
```

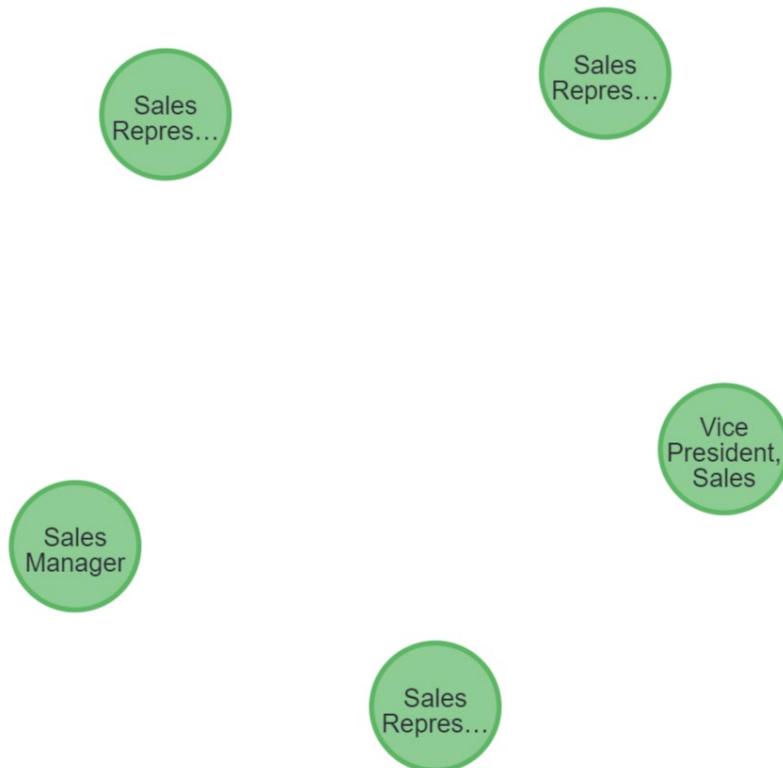


Este código crea 9 nodos de empleados en la base de datos.

Puede ver algunos de estos nodos en la base de datos ejecutando este código:

```
MATCH (e:Employee) return e LIMIT 5;
```

La vista de grafo es:



La vista de tabla contiene estos valores para las propiedades del nodo:

"e"
{ "lastName": "Davolio", "firstName": "Nancy", "employeeID": "1", "title": "Sales Representative"}
{ "lastName": "Fuller", "firstName": "Andrew", "employeeID": "2", "title": "Vice President, Sales"}
{ "lastName": "Leverling", "firstName": "Janet", "employeeID": "3", "title": "Sales Representative"}
{ "lastName": "Peacock", "firstName": "Margaret", "employeeID": "4", "title": "Sales Representative"}
{ "lastName": "Buchanan", "firstName": "Steven", "employeeID": "5", "title": "Sales Manager"}

CREACIÓN DE NODOS DE CATEGORÍA

```
// Create categories
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/categories.csv' AS row
MERGE (c:Category {categoryID: row.CategoryID})
ON CREATE SET c.categoryName = row.CategoryName, c.description = row.Description;
```

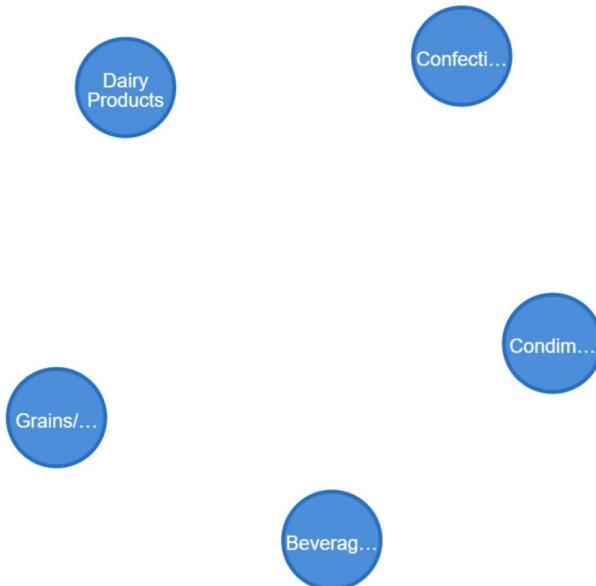


Este código crea 8 nodos de categoría en la base de datos.

Puede ver algunos de estos nodos en la base de datos ejecutando este código:

```
MATCH (c:Category) return c LIMIT 5;
```

La vista de grafo es:



La vista de tabla contiene estos valores para las propiedades del nodo:

"c"
{"description":"Soft drinks, coffees, teas, beers, and ales","categoryName":"Beverages","categoryID":"1"}
{"description":"Sweet and savory sauces, relishes, spreads, and seasonings","categoryName":"Condiments","categoryID":"2"}
{"description":"Desserts, candies, and sweet breads","categoryName":"Confections","categoryID":"3"}
{"description":"Cheeses","categoryName":"Dairy Products","categoryID":"4"}
{"description":"Breads, crackers, pasta, and cereal","categoryName":"Grains/Cereals","categoryID":"5"}

Para conjuntos de datos comerciales o empresariales muy grandes, es posible que encuentre errores de falta de memoria, especialmente en máquinas más pequeñas. Para evitar estas situaciones, puede anteponer la declaración con :auto USING PERIODIC COMMIT para confirmar datos en lotes. Esta práctica no es una recomendación estándar para conjuntos de datos más pequeños, pero solo se recomienda cuando existen amenazas de problemas de memoria.



CREACIÓN DE ÍNDICES Y RESTRICCIONES PARA LOS DATOS EN EL GRAFO

Una vez creados los nodos, debe crear las relaciones entre ellos. Importar las relaciones significará buscar los nodos que acaba de crear y agregar una relación entre esas entidades existentes. Para asegurarse de que la búsqueda de nodos esté optimizada, creará índices para cualquier propiedad de nodo utilizada en las búsquedas (a menudo, la identificación u otro valor único).

También queremos crear una restricción (también crea un índice con ella) que impedirá la creación de pedidos con la misma identificación, evitando duplicados. Finalmente, como los índices se crean después de que se insertan los nodos, su población ocurre de forma asíncrona, por lo que llamamos db.awaitIndexes() a bloquear hasta que se llenen.

Ejecute este bloque de código:

```
CREATE INDEX product_id FOR (p:Product) ON (p.productID);
CREATE INDEX product_name FOR (p:Product) ON (p.productName);
CREATE INDEX supplier_id FOR (s:Supplier) ON (s.supplierID);
CREATE INDEX employee_id FOR (e:Employee) ON (e.employeeID);
CREATE INDEX category_id FOR (c:Category) ON (c.categoryID);
CREATE CONSTRAINT order_id FOR (o:Order) REQUIRE o.orderID IS UNIQUE;
CALL db.awaitIndexes();
```

Después de ejecutar este código, puede ejecutar este código para ver los índices (y la restricción) en la base de datos:

```
SHOW INDEXES;
```

Debería ver estos índices (y restricciones) en la base de datos:

"id"	"name"	"state"	"populationPercent"	"uniqueness"	"type"	"entityType"	"labelsOrTypes"	"properties"	"provider"
7	"category_id"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["Category"]	["categoryID"]	"native-btree-1.0"
6	"employee_id"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["Employee"]	["employeeID"]	"native-btree-1.0"
1	"index_343aff4e"	"ONLINE"	100.0	"NONUNIQUE"	"LOOKUP"	"NODE"	[]	[]	"token-lookup-1.0"
2	"index_f7700477"	"ONLINE"	100.0	"NONUNIQUE"	"LOOKUP"	"RELATIONSHIP"	[]	[]	"token-lookup-1.0"
8	"order_id"	"ONLINE"	100.0	"UNIQUE"	"BTREE"	"NODE"	["Order"]	["orderID"]	"native-btree-1.0"
3	"product_id"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["Product"]	["productID"]	"native-btree-1.0"
4	"product_name"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["Product"]	["productName"]	"native-btree-1.0"
5	"supplier_id"	"ONLINE"	100.0	"NONUNIQUE"	"BTREE"	"NODE"	["Supplier"]	["supplierID"]	"native-btree-1.0"

CREANDO LAS RELACIONES ENTRE LOS NODOS.

A continuación, creará relaciones:

1. Entre órdenes y empleados.
2. Entre productos y proveedores y entre productos y categorías.
3. Entre empleados.

CREAR RELACIONES ENTRE PEDIDOS Y EMPLEADOS.

Con los nodos e índices iniciales en su lugar, ahora puede crear las relaciones de pedidos a productos y pedidos a empleados.

Ejecute este bloque de código:



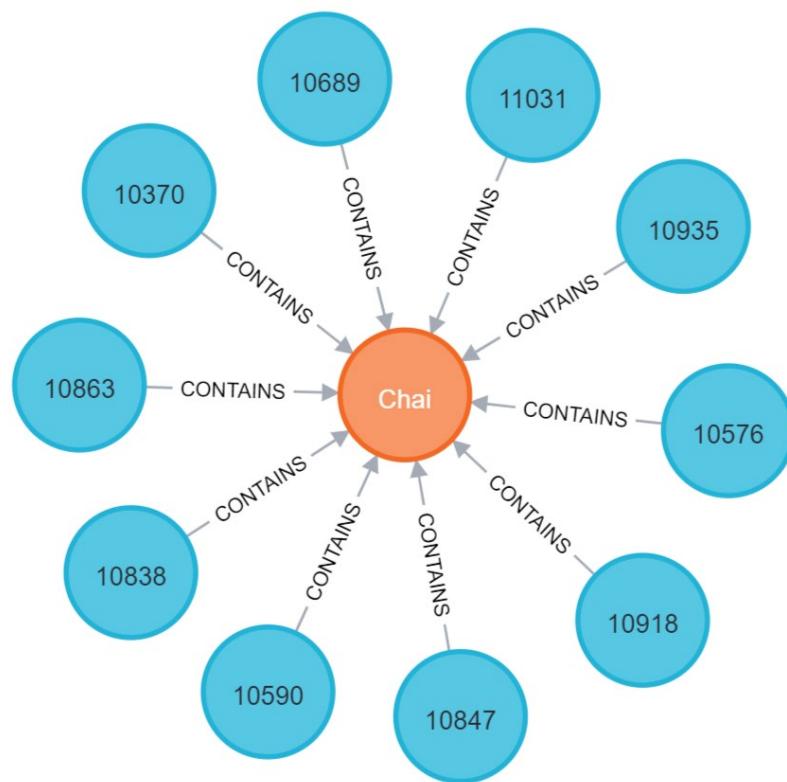
```
// Create relationships between orders and products
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/orders.csv' AS row
MATCH (order:Order {orderId: row.OrderID})
MATCH (product:Product {productId: row.ProductID})
MERGE (order)-[op:CONTAINS]->(product)
    ON CREATE SET op.unitPrice =toFloat(row.UnitPrice), op.quantity = toFloat(row.Quantity);
```

Este código crea 2155 relaciones en el gráfico.

Puedes ver algunos de ellos ejecutando este código:

```
MATCH (o:Order)-[]-(p:Product)
RETURN o,p LIMIT 10;
```

La vista de grafo es:



Luego, ejecuta este bloque de código:

```
// Create relationships between orders and employees
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/orders.csv' AS row
MATCH (order:Order {orderId: row.OrderID})
MATCH (employee:Employee {employeeID: row.EmployeeID})
MERGE (employee)-[:SOLD]->(order);
```

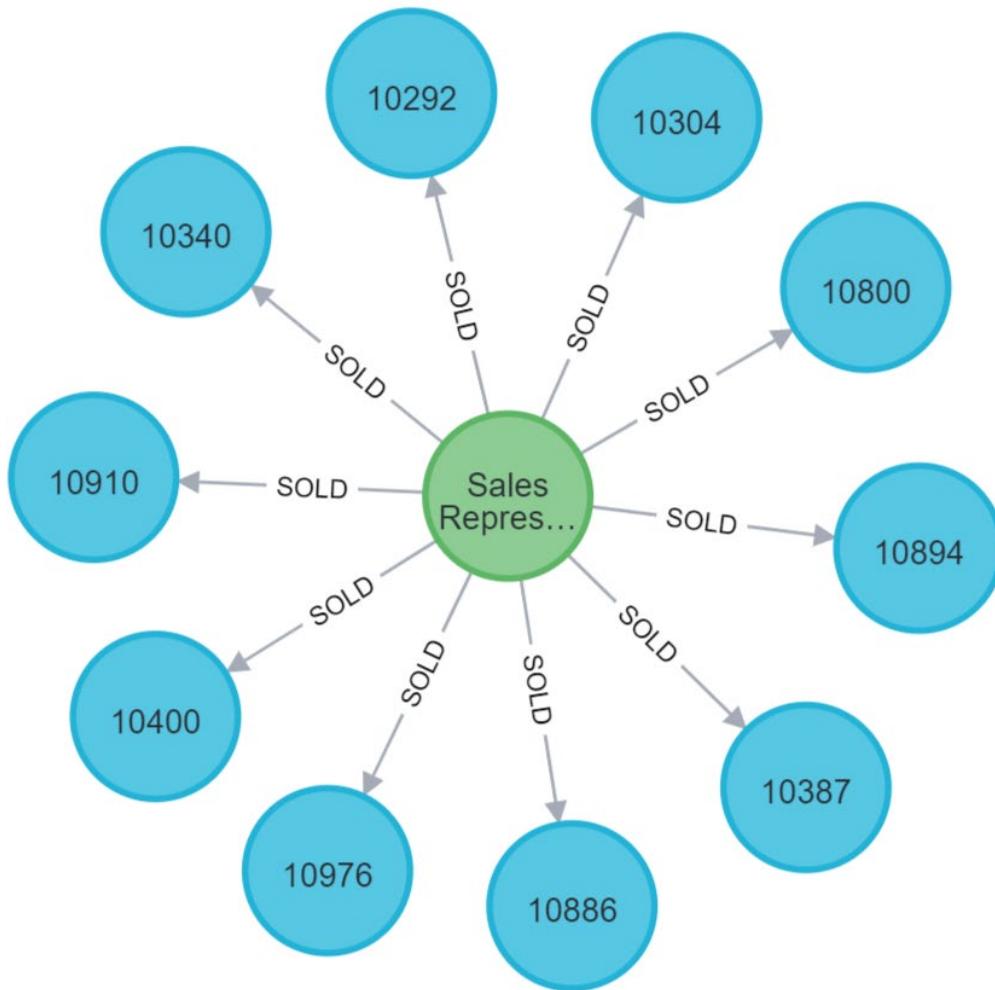
Este código crea 830 relaciones en el gráfico.



Puedes ver algunos de ellos ejecutando este código:

```
MATCH (o:Order)-[]-(e:Employee)
RETURN o,e LIMIT 10;
```

La vista de grafo es:



CREAR RELACIONES ENTRE PRODUCTOS Y PROVEEDORES Y ENTRE PRODUCTOS Y CATEGORÍAS.

A continuación, cree relaciones entre productos, proveedores y categorías:

Ejecute este bloque de código:

```
// Create relationships between products and suppliers
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/products.csv'
AS row
MATCH (product:Product {productId: row.ProductID})
MATCH (supplier:Supplier {supplierID: row.SupplierID})
MERGE (supplier)-[:SUPPLIES]->(product);
```

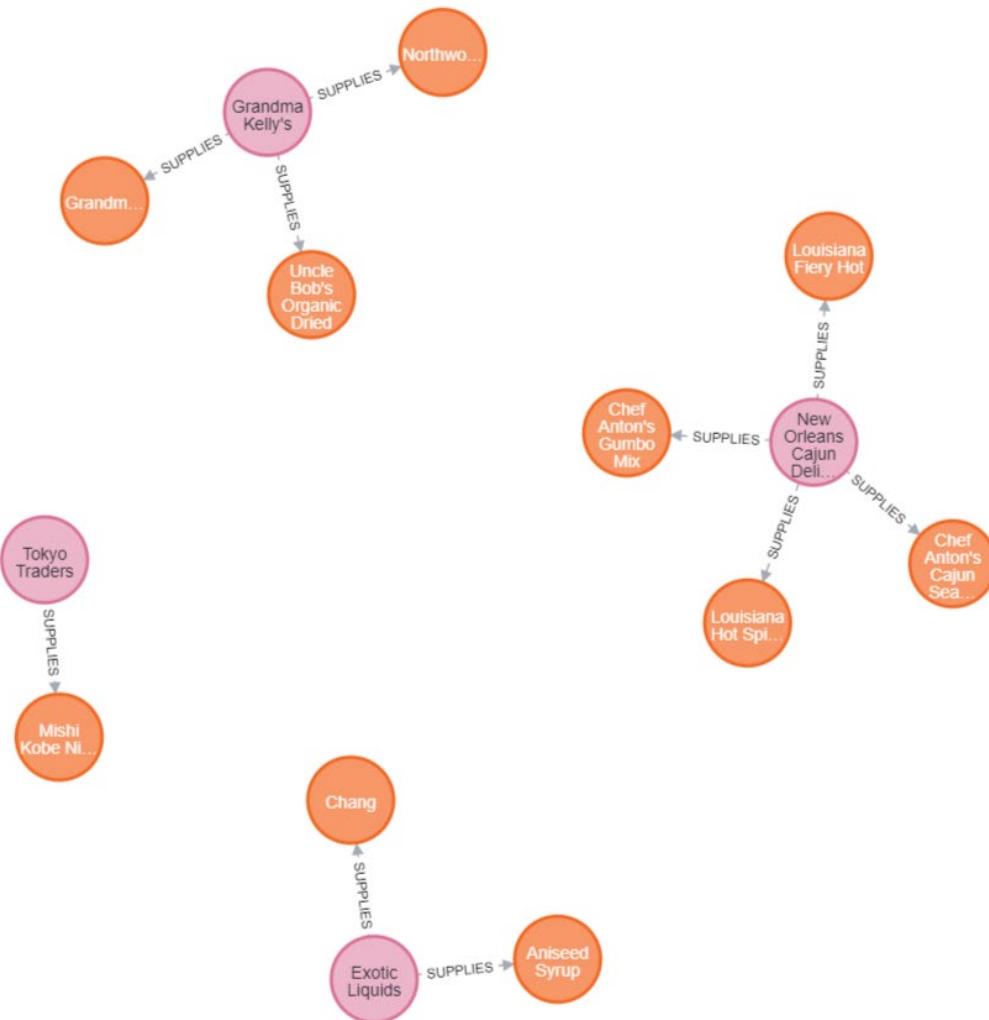
Este código crea 77 relaciones en el gráfico.

Puedes ver algunos de ellos ejecutando este código:



```
MATCH (s:Supplier)-[]-(p:Product)
RETURN s,p LIMIT 10;
```

El grafo sería algo así:



Luego, ejecuta este bloque de código:

```
// Create relationships between products and categories
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/products.csv'
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (category:Category {categoryID: row.CategoryID})
MERGE (product)-[:PART_OF]->(category);
```

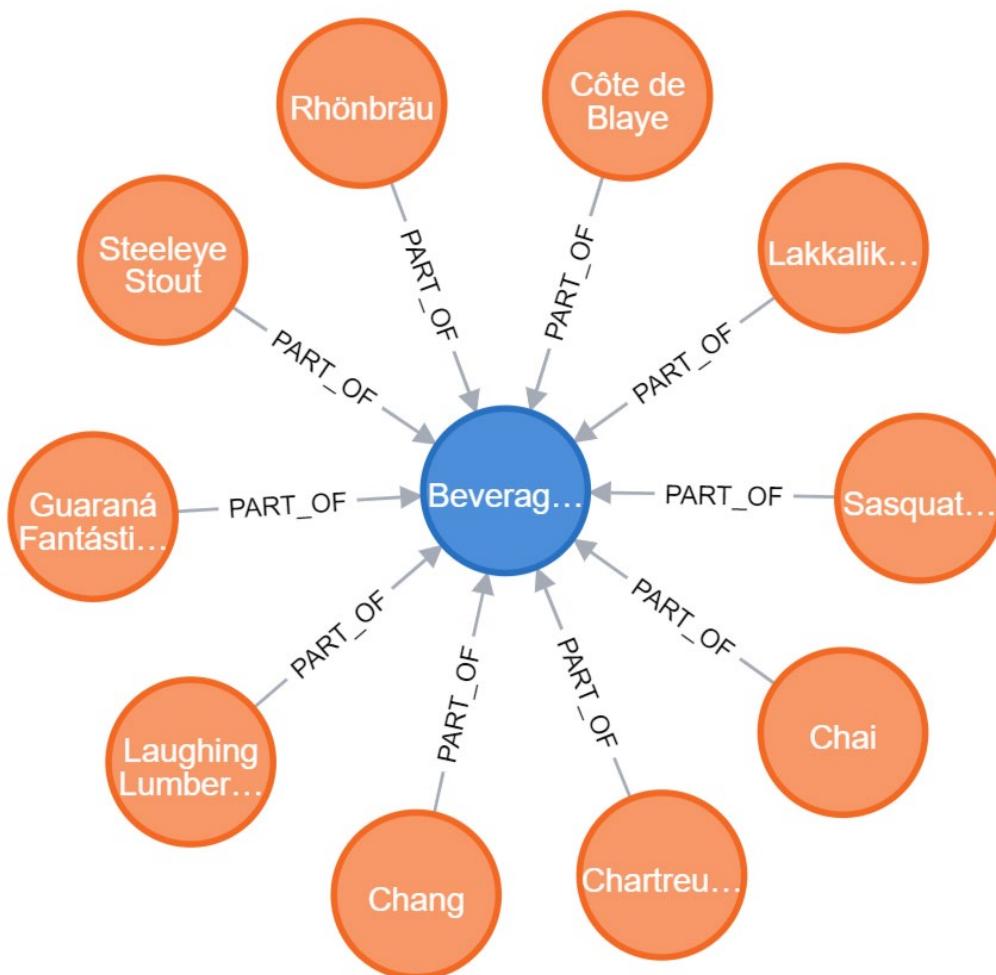
Este código crea 77 relaciones en el grafo.

Puedes ver algunos de ellos ejecutando este código:

```
MATCH (c:Category)-[]-(p:Product)
RETURN c,p LIMIT 10;
```



La vista de grafo debería verse así:



CREAR RELACIONES ENTRE LOS EMPLEADOS.

Por último, creará la relación 'REPORTS_TO' entre los empleados para representar la estructura de reporte:

Ejecute este bloque de código:

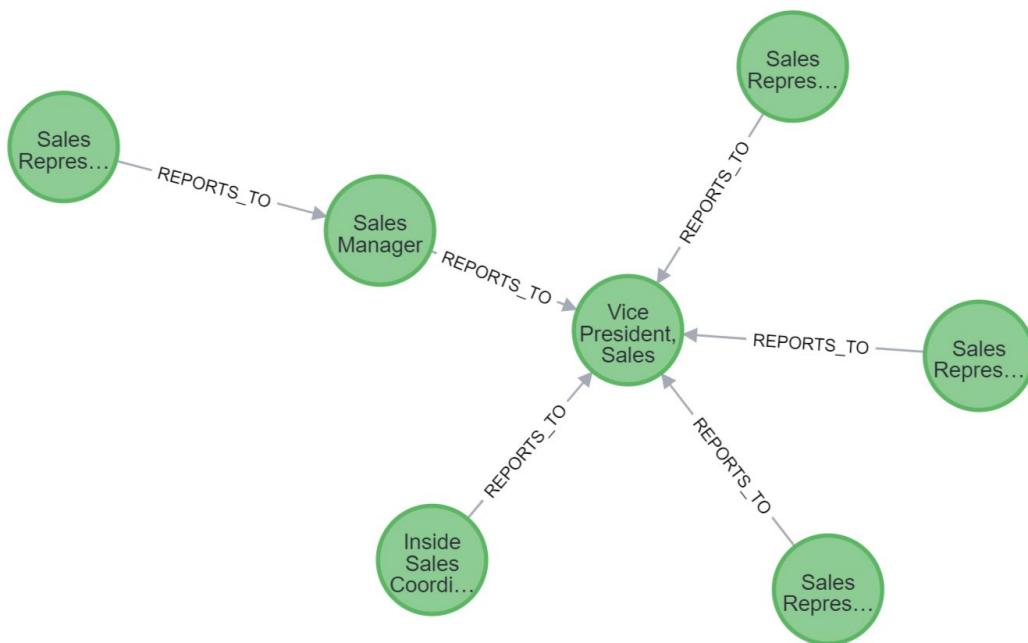
```
// Create relationships between employees (reporting hierarchy)
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc8
8381995e6823ff3f419b5a0cb8ac4f/employees.csv' AS row
MATCH (employee:Employee {employeeID: row.EmployeeID})
MATCH (manager:Employee {employeeID: row.ReportsTo})
MERGE (employee)-[:REPORTS_TO]->(manager);
```

Este código crea 8 relaciones en el grafo.

Puedes ver algunos de ellos ejecutando este código:

```
MATCH (e1:Employee)-[]-(e2:Employee)
RETURN e1,e2 LIMIT 10;
```

El grafo debería verse así:



A continuación, consultará el grafo resultante para averiguar qué puede decirnos sobre nuestros datos recién importados.

CONSULTANDO EL GRAFO

Podríamos comenzar con un par de consultas generales para verificar que nuestros datos coincidan con el modelo que diseñamos anteriormente en la guía. Aquí hay algunos ejemplos de consultas.

Ejecute este bloque de código:

```
//find a sample of employees who sold orders with their ordered products
MATCH (e:Employee)-[rel:SOLD]->(o:Order)-[rel2:CONTAINS]->(p:Product)
RETURN e, rel, o, rel2, p LIMIT 25;

//find the supplier and category for a specific product
MATCH (s:Supplier)-[r1:SUPPLIES]->(p:Product {productName: 'Chocolade'})-[r2:PART_OF]-
>(c:Category)
RETURN s, r1, p, r2, c;
```

Una vez que esté seguro de que los datos se alinean con nuestro modelo de datos y todo parece correcto, puede comenzar a consultar para recopilar información y conocimientos para las decisiones comerciales.

¿QUÉ EMPLEADO TUVO LA MAYOR CANTIDAD DE VENTAS CRUZADAS DE 'CHOCOLATE' Y OTRO PRODUCTO?

Ejecute este bloque de código:

```
MATCH (choc:Product {productName: 'Chocolade'})-<-[ :CONTAINS ]-(:Order)-<-[ :SOLD ]-(employee),
      (employee)-[:SOLD]->(o2)-[:CONTAINS]->(other:Product)
RETURN employee.employeeID as employee, other.productName as otherProduct, count(distinct o2) as count
ORDER BY count DESC
LIMIT 5;
```

Parece que el empleado n.º 4 estuvo ocupado, ¡aunque al empleado n.º 1 también le fue bien! Sus resultados deberían ser algo como esto:



"employee"	"otherProduct"	"count"
"4"	"Gnocchi di nonna Alice"	14
"3"	"Gumbär Gummibärchen"	12
"4"	"Pâté chinois"	12
"1"	"Flotemysost"	12
"1"	"Pavlova"	11

¿CÓMO SE ORGANIZAN LOS EMPLEADOS? ¿QUIÉN REPORTA A QUIÉN?

Ejecute este bloque de código:

```
MATCH (e:Employee)-[:REPORTS_TO]-(sub)
RETURN e.employeeID AS manager, sub.employeeID AS employee;
```

"manager"	"employee"
"2"	"5"
"2"	"3"
"2"	"8"
"2"	"1"
"2"	"4"
"5"	"7"
"5"	"9"
"5"	"6"

Note que el empleado 5 tiene empleados que reportan a él, pero él reporta al empleado 2.

Veamos un poco más.

¿QUÉ EMPLEADOS REPORTAN A QUIÉN INCLUSO DE MANERA INDIRECTA?

Ejecutemos este código:

```
MATCH path = (e:Employee)-[:REPORTS_TO*]-(sub)
WITH e, sub, [person in NODES(path) | person.employeeID][1..-1] AS path
RETURN e.employeeID AS manager, path as middleManager, sub.employeeID AS employee
ORDER BY size(path);
```

El resultado es:



"manager"	"middleManager"	"employee"
"2"	[]	"4"
"2"	[]	"1"
"2"	[]	"8"
"2"	[]	"3"
"2"	[]	"5"
"5"	[]	"6"
"5"	[]	"9"
"5"	[]	"7"
"2"	["5"]	"6"
"2"	["5"]	"9"
"2"	["5"]	"7"

¿CUÁNTAS ÓRDENES FUERON REALIZADAS POR CADA PARTE DE LA JERARQUÍA?

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO*0..]-(sub)-[:SOLD]->(order)
RETURN e.employeeID as employee, [x IN COLLECT(DISTINCT sub.employeeID) WHERE x <> e.employeeID] AS reportsTo, COUNT(distinct order) AS totalOrders
ORDER BY totalOrders DESC;
```

El resultado es:

"employee"	"reportsTo"	"totalOrders"
"2"	["4", "1", "8", "3", "5", "6", "9", "7"]	830
"5"	["6", "9", "7"]	224
"4"	[]	156
"3"	[]	127
"1"	[]	123
"8"	[]	104
"7"	[]	72
"6"	[]	67
"9"	[]	43



BASES DE DATOS DOCUMENTALES

El modelo de datos documental está fuertemente orientado a agregados, dado que una base de datos consiste en un conjunto de agregados denominados «documentos». Las bases de datos documentales se caracterizan por que definen un conjunto de estructuras y tipos permitidos que pueden ser almacenados, y además es posible acceder a la estructura del agregado, teniendo como ventaja que se consigue más flexibilidad en el acceso. En este sentido se pueden realizar consultas a la base de datos según los campos del agregado, se pueden recuperar partes del agregado en vez del agregado completo, y además se pueden crear índices basados en el contenido del agregado. Dado que cada agregado tiene asociado un identificador, es posible realizar búsquedas del estilo clave-valor.

MongoDB es una base de datos NoSQL orientada a documentos creada por la compañía 10gen en el año 2007. Se caracteriza por que almacena los datos en documentos de tipo JSON con un esquema dinámico denominado «BSON».

A continuación, se exponen los conceptos básicos y la terminología fundamental necesaria para poder entender qué es una base de datos en MongoDB y qué elementos la forman. En este sentido se definen los conceptos de documento, colección y base de datos, así como los tipos de datos soportados por MongoDB. Asimismo se realiza una introducción a la Shell de comandos que constituye la principal forma de interactuar con MongoDB.

DOCUMENTOS

Los documentos son la unidad básica de organización de la información en MongoDB, y desempeñan un papel equivalente a una fila en las bases de datos relacionales. Un documento es un conjunto ordenado de claves que tienen asociados valores, y que se corresponden con algunas estructuras de datos típicas de los lenguajes de programación tales como tablas hash o diccionarios. En general, los documentos contendrán múltiples pares clave-valor, como, por ejemplo, {“Nombre”:”Juan”, “País”:”España”}.

Sus principales características son:

- Las claves son cadenas, por lo que se permite cualquier carácter con un par de excepciones:
 - La clave no pueden contener el carácter nulo «\0».
 - El punto «.» y el «\$» deben evitarse, pues tienen propiedades especiales.
- MongoDB es sensible tanto a las mayúsculas/minúsculas como a los tipos de datos. Así, por ejemplo, los siguientes documentos se consideran distintos: {“Edad”:3}, {“Edad”: “3”}, {“edad”:3}, {“edad”:”3”}.
- Los documentos no pueden tener claves duplicadas. Así, por ejemplo, el siguiente documento es incorrecto: {“edad”:3,”edad”:56}.
- Los pares clave-valor están ordenados en los documentos. Por ejemplo, el documento {“x”:3,”y”:5} no es igual que {“y”:5,”x”:3}. Es importante no definir las aplicaciones pensando en el orden de los campos, pues MongoDB puede reordenarlos automáticamente en determinadas situaciones.
- Los valores de un documento pueden ser de diferentes tipos.

TIPOS DE DATOS

Los principales tipos de datos soportados por los documentos en MongoDB son:

- Nulo: representa el valor nulo o bien un campo que no existe. Por ejemplo, {“x”:null}.
- Booleanos: representa el tipo booleano, que puede tomar los valores de true o false. Por ejemplo, {“x”:true}.
- Números: distingue entre números reales, como, por ejemplo, {“x”:3.14}, y números enteros, como, por ejemplo, {“x”:45}.
- Cadenas: cualquier cadena de caracteres, como, por ejemplo, {“x”：“Ejemplo”}.
- Fechas: almacena la fecha en milisegundos, pero no la zona horario. Por ejemplo, {“x”:new Date()}.
- Expresiones regulares: se pueden usar expresiones regulares para realizar consultas.



- Arrays: se representa como un conjunto o lista de valores. Por ejemplo, {"x": ["a", "b", "c"]}.
- Documentos embebidos: los documentos pueden contener documentos embebidos como valores de un documento padre. Por ejemplo, {"x": {"y": 45}}.
- Identificadores de objetos: es un identificador de 12 bytes para un documento. Por ejemplo, {"x": ObjectId()}).
- Datos binarios: es una cadena de bytes arbitraria que no puede ser manipulada directamente desde el Shell y que sirve para representar cadenas de caracteres no UTF8.
- Código Javascript: los documentos y las consultas pueden contener código JavaScript. Por ejemplo, {"x": function () { ... }}.

Hay que observar:

- a) Fechas. Para crear un objeto de tipo fecha se usa el comando new Date (). Sin embargo, si se llama sin new (solo Date ()) , se retorna una cadena que representa la fecha. Y por tanto se trata de diferentes tipos de datos. Las fechas en el Shell son mostradas usando la configuración local de la zona horaria, sin embargo, la base de datos las almacena como un valor en milisegundos sin referencia a la zona horaria (aunque podría almacenarse este valor definiendo una clave para el mismo).
- b) Arrays. Pueden ser usados tanto en operaciones en las que el orden es importante, tales como listas, pilas o colas, como en operaciones en las que el orden no es importante, tales como conjuntos. Los arrays pueden contener diferentes tipos de valores, como, por ejemplo, {"Cosas": ["edad", 45]} (de hecho, soporta cualquiera de los tipos de valores soportados para los documentos, por lo que se pueden crear arrays anidados). Una propiedad importante en MongoDB es que reconoce la estructura de los arrays y permite navegar por el interior de los arrays para realizar operaciones sobre sus contenidos, como consultas, o crear índices sobre sus contenidos. En el ejemplo anterior se podría crear una consulta para recuperar todos aquellos documentos donde 3.14 es un elemento del array «Cosas», y si, por ejemplo, esta fuera una consulta habitual entonces incluso se podría crear un índice sobre la clave «Cosas» y mejorar el rendimiento de la consulta. Asimismo MongoDB permite realizar actualizaciones que modifican los contenidos de los arrays, tales como cambiar un valor del array por otro.
- c) Documentos embebidos. Los documentos pueden ser usados como valores de una clave, y en este caso se denominan «documentos embebidos». Se suelen usar para organizar los datos de una manera lo más natural posible. Por ejemplo, si se tiene un documento que representa a una persona y se quiere almacenar su dirección, podría crearse anidando un documento «dirección» al documento asociado a una persona, como, por ejemplo:

```
{  
  "nombre": "Juan",  
  "dirección": {  
    "calle": "Mayor 3",  
    "ciudad": "Madrid",  
    "País": "España"  
  }}
```

MongoDB es capaz de navegar por la estructura de los documentos embebidos y realizar operaciones con sus valores, como, por ejemplo, crear índices, consultas o actualizaciones. La principal desventaja de los documentos embebidos se debe a la repetición de datos. Por ejemplo, supóngase que las direcciones se encuentran en una tabla independiente y que hay que rectificar un error tipográfico, entonces, al rectificar la dirección en la tabla direcciones, se rectifica para todos los usuarios que comparten la misma dirección. Sin embargo, en MongoDB habría que rectificar las direcciones una a una en cada documento, aunque se compartan direcciones.

- d) Identificador de objetos

Cada documento tiene que tener un clave denominada «_id»:



- El valor de esta clave puede ser de cualquier tipo pero por defecto será de tipo ObjectId.
- En una colección, cada documento debe tener un valor único y no repetido para la clave «_id», lo que asegura que cada documento en la colección pueda ser identificado de manera única. Así, por ejemplo, dos colecciones podrían tener un documento con «_id» con el valor 123, pero en una misma colección no podría haber dos documentos con valor de «_id» de 123.
- El tipo ObjectId es el tipo por defecto para los valores asociados a la clave «_id». Es un tipo de datos diseñado para ser usado en ambientes distribuidos de manera que permita disponer de valores que sean únicos globalmente. Cada valor usa 12 bytes, lo que permite representar una cadena de 24 dígitos hexadecimales (2 dígitos por cada byte). Si se crean múltiples valores del tipo ObjectId sucesivamente, solo cambian unos pocos dígitos del final y una pareja de dígitos de la mitad. Esto se debe a la forma en la que se crean los valores del tipo ObjectIDs. Los 12 bytes se generan así:

Timestamp				Machine				PID		Increment		
0	1	2	3	4	5	6	7	8	9	10	11	

Hay que observar que:

- Los primeros 4 bytes son una marca de tiempo en segundos, que, combinados con los siguientes 4 bytes, proporciona unicidad a nivel de segundo y que identifican de manera implícita cuando el documento fue creado. Por otro lado, a causa de que la marca de tiempo aparece en primer lugar, los ObjectIDs se ordenan obligatoriamente en orden de inserción, lo que hace que la indexación sobre ObjectIDs sea eficiente.
- Los siguientes 3 bytes son un identificador único de la máquina que lo genera, lo que garantiza que diferentes máquinas no generan colisiones.
- Para conseguir unicidad entre diferentes procesos que generan ObjectIDs concurrentemente en una misma máquina, se usan los siguientes 2 bytes, que son tomados del identificador del proceso que genera un ObjectId.
- Los primeros 9 bytes aseguran la unicidad a través de máquinas y procesos para un segundo, y los últimos 3 bytes son un contador que se incrementa y que es el responsable de la unicidad dentro de un segundo en un proceso.
- Este sistema permite generar 2563 únicos ObjectIDs por proceso en un segundo.

Cuando un documento se va a insertar, si no tiene un valor para la clave «_id», es generado automáticamente por MongoDB.

COLECCIONES

Una colección es un grupo de documentos, y desempeña el papel análogo a las tablas en las bases de datos relacionales.

Las colecciones tienen esquemas dinámicos, lo que significa que dentro de una colección puede haber cualquier número de documentos con diferentes estructuras. Por ejemplo, en una misma colección podrían estar los siguientes documentos diferentes: {"edad":34}, {"x":"casa"} que tienen diferentes claves y diferentes tipos de valores.

Dado que cualquier documento, se puede poner en cualquier colección, y dado que no es necesario disponer de esquemas distintos para los diferentes tipos de documentos, entonces surge la pregunta de por qué se necesita usar más de una colección y tener que separar los documentos mediante colecciones separadas:



- Si se mantuvieran diferentes tipos de documentos en la misma colección, produciría problemas para asegurar que cada consulta solo recupera documentos de un cierto tipo o que en el código de la aplicación implementa consultas para cada tipo de documento.
- Es más rápido obtener una lista de colecciones que extraer una lista de los tipos de documentos en una colección.
- La agrupación de documentos del mismo tipo juntos en la misma colección permite la localidad de los datos.
- Cuando se crean índices se impone cierta estructura a los documentos (especialmente en los índices únicos). Estos índices están definidos por colección de forma que poniendo documentos de un solo tipo en la misma colección entonces se podrán indexar las colecciones de una forma más eficiente.

Por tanto, estas razones hacen razonable crear un esquema y agrupar los tipos relacionados de documentos juntos, aunque MongoDB no lo imponga como obligatorio.

Una colección se identifica por su nombre, que es una cadena con las siguientes restricciones:

- La cadena vacía no es un nombre válido para una colección.
- Los nombres de las colecciones no pueden contener el carácter nulo «\0», pues este símbolo se usa para indicar el fin del nombre de una colección.
- No se debe crear ninguna colección que empiece con «system», dado que es un prefijo reservado para las colecciones internas. Por ejemplo, la colección system.users contiene los usuarios de la base de datos, la colección system.namespaces contiene información acerca de todas las colecciones de la base de datos.
- Las colecciones creadas por los usuarios no deben contener el carácter reservado «\$» en su nombre.

Una convención para organizar las colecciones consiste en definir subcolecciones usando espacios de nombres separados por el carácter «.». Por ejemplo, una aplicación que contuviese un blog podría tener una colección denominada «blog.posts» y otra colección denominada «blog.autores» con un propósito organizativo y que, sin embargo, ni existe la colección blog y en caso de existir no exista una relación entre la colección padre blog y las subcolecciones.

Aunque las subcolecciones no tienen propiedades especiales, son útiles por algunas razones:

- Existe un protocolo en MongoDB para almacenar archivos muy extensos denominado «GridFS» que usa las subcolecciones para almacenar los archivos de metadatos separados de los datos.
- Algunas librerías proporcionan funciones para acceder de una forma simple a las subcolecciones de una colección. Por ejemplo, db.blog proporciona acceso a la colección blog y db.blog.posts permite acceder a la colección blog.posts

BASES DE DATOS

Las colecciones se agrupan en bases de datos, de manera que una única instancia de MongoDB puede gestionar varias bases de datos cada una agrupando cero o más colecciones.

Hay que observar que:

- Cada base de datos tiene sus propios permisos y se almacena en ficheros del disco separados.
- Una buena regla general consiste en almacenar todos los datos de una aplicación en la misma base de datos.
- Las bases de datos separadas son útiles cuando se almacenan datos para aplicaciones o usuarios diferentes que usan el mismo servidor de MongoDB.

Las bases de datos se identifican mediante nombres que son cadenas con las siguientes restricciones:



- La cadena vacía no es un nombre válido para una base de datos.
- El nombre de una base de datos no puede contener ninguno de los siguientes caracteres: \, /, ., , *, <, >, :, |, ?, \$, espacio o \0(valor nulo).
- Los nombres de las bases de datos son sensitivos a mayúsculas y minúsculas incluso sobre sistemas de archivos que no lo sean. Una regla práctica es usar siempre nombres en minúscula.
- Los nombres están limitados a un máximo de 64 bytes.
- Existen nombres que no pueden usarse para las bases de datos por estar reservados:
 - 1) admin. Es el nombre de la base de datos «root» en términos de autenticación. Si un usuario es añadido a esta base de datos, entonces el usuario hereda los permisos para todas las bases de datos. Existen determinados comandos que solo pueden ser ejecutados desde esta base de datos, tales como listar todas las bases de datos o apagar el servidor.
 - 2) local. Esta base de datos nunca será replicada y sirve para almacenar cualquier colección que debería ser local a un servidor.
 - 3) configura. Cuando en MongoDB se usa una configuración con sharding, se usa esta base de datos para almacenar información acerca de los fragmentos o shards que se crean.

Mediante la concatenación del nombre de una base de datos con una colección de la base de datos se consigue una cualificación entera del nombre de la colección denominado «espacio de nombres». Por ejemplo, si se usa la colección blog.posts en la base de datos cms, el espacio de nombres de esa colección sería cms.blog.posts. Los espacios de nombres están limitados a 121 bytes de longitud, aunque en la práctica es mejor que sean menores de 100 bytes.

MONGODB EN DOCKER

Los ejemplos que veremos a continuación se realizarán utilizando MongoDB en un contenedor de Docker.

Los pasos a seguir para realizar la instalación son:

- 1) Obtener la imagen de Docker de MongoDB
docker pull mongodb/mongodb-community-server
- 2) Ejecutar la imagen como un contenedor
docker run --name testmongo -p 27017:27017 -d mongodb/mongodb-community-server:latest
- 3) Verificar que el contenedor se está ejecutando
docker container ls

```
PS C:\Users\joi> docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
355221019cff mongodb/mongodb-community-server:latest "python3 /usr/local/..." 2 minutes ago Up About a minute 0.0.0.0:27017->27017/tcp testmongo
```

- 4) Conectar a MongoDB mediante mongosh

```
docker exec -it testmongo mongosh
```

```
PS C:\Users\joi> docker exec -it testmongo mongosh
Current Mongosh Log ID: 65e785f59a768d0815570a00
Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.1.5
Using MongoDB: 7.0.6
Using Mongosh: 2.1.5

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

-----
The server generated these startup warnings when booting
2024-03-05T20:48:31.693+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2024-03-05T20:48:33.276+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
2024-03-05T20:48:33.278+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
2024-03-05T20:48:33.278+00:00: vm.max_map_count is too low
```

LA SHELL DE COMANDOS DE MONGODB

La Shell es un cliente de MongoDB que permite interaccionar con una instancia de MongoDB desde una línea de comandos con el objetivo de realizar funciones administrativas, inspeccionar la instancia que se está ejecutando



o llevar a cabo determinadas operaciones de gestión sobre la base de datos. Se trata de la principal interfaz para comunicarse con una instancia de MongoDB.

La Shell intenta conectarse al servidor de MongoDB que se esté ejecutando, y en caso contrario se producirá un fallo. Si la conexión se ha realizado con éxito, lo primero que hace es conectarse a una base de datos por defecto denominada «test» y asignar esta conexión a la variable global db. Esta variable es el punto de acceso primario al servidor MongoDB a través de la Shell. Para ver la base de datos que tiene asignada en un momento dado, basta con teclear en la Shell «db».

```
test> db  
test  
test>
```

En particular, la Shell es un intérprete de JavaScript que es capaz de ejecutar cualquier programa en JavaScript, como, por ejemplo, ejecutar operaciones matemáticas.

```
test> 1000/200  
5  
test>
```

También es capaz de utilizar librerías de JavaScript.

```
test> Math.sin(Math.PI / 2);  
1  
test> new Date("2024/6/1")  
ISODate('2024-06-01T00:00:00.000Z')  
test> "¡Hola Mundo!".replace("Mundo", "Juan");  
¡Hola Juan!  
test>
```

Asimismo es posible definir y llamar a funciones de JavaScript.

```
test> function factorial (n) {  
... if (n <= 1) return 1;  
... return n * factorial(n - 1);  
... }  
[Function: factorial]  
test> factorial(5);  
120  
test>
```

En la Shell se pueden crear comandos que se encuentren en varias líneas, de manera que la Shell detectará si el comando está completo cuando se presiona Enter. En caso de no estar completo, la Shell permite continuar escribiendo en la siguiente línea. Si se presiona tres veces Enter, se cancela y se vuelve a una nueva línea.

La Shell contiene algunos comandos adicionales que no son válidos en JavaScript pero que simplifican las operaciones con las bases de datos, como, por ejemplo, el comando «use», que permite seleccionar o cambiar de base de datos.



```
test> use prueba
switched to db prueba
prueba>
```

Las colecciones pueden ser accedidas desde la variable db, como, por ejemplo, db.blog retornaría la colección blog en la base de datos actual.

OPERACIONES BÁSICAS EN LA SHELL

En la Shell se pueden realizar cuatro operaciones básicas para manipular y ver datos denominadas CRUD (crear, leer, actualizar o borrar):

- Crear. El comando insert añade un documento a una colección. Por ejemplo, si se quiere almacenar un post de un blog, primero se crea una variable post que representa al documento y que tendrá como claves título, contenido y fecha de publicación:

```
prueba> post= { "titulo": "Mi primer post de mi blog",
... "contenido": "Este es mi primer post de mi blog",
... "date": new Date()
{
  titulo: 'Mi primer post de mi blog',
  contenido: 'Este es mi primer post de mi blog',
  date: ISODate('2024-03-05T21:09:24.905Z')
}
prueba>
```

A continuación, se usa insertOne para añadirlo a la colección blog.

```
prueba> db.blog.insertOne(post)
{
  acknowledged: true,
  insertedId: ObjectId('65e78a64e4dac6b7dfb11658')
}
```

Ahora se puede recuperar con una llamada a la función find sobre la colección.

```
prueba> db.blog.find()
[
  {
    _id: ObjectId('65e78a64e4dac6b7dfb11658'),
    titulo: 'Mi primer post de mi blog',
    contenido: 'Este es mi primer post de mi blog',
    date: ISODate('2024-03-05T21:09:24.905Z')
  }
]
prueba>
```

En el ejemplo se puede observar que se ha añadido una clave «_id» y que el resto de las claves se han mantenido intactas.

- Leer. Los comandos find y findOne pueden ser usados para consultar una colección. Si solo se quiere recuperar un único documento de una colección, se usará findOne:



```
prueba> db.blog.findOne()
{
  _id: ObjectId('65e78a64e4dac6b7dfb11658'),
  titulo: 'Mi primer post de mi blog',
  contenido: 'Este es mi primer post de mi blog',
  date: ISODate('2024-03-05T21:09:24.905Z')
}
prueba>
```

Tanto a find () como a findOne () se les puede pasar como parámetro una condición de recuperación que permite restringir los documentos recuperados. La Shell mostrará automáticamente 20 documentos pero se pueden recuperar más.

- c) Actualizar. Para actualizar se dispone del comando update, que toma al menos dos parámetros: un criterio para encontrar el documento que se desea actualizar y el nuevo documento. Por ejemplo, supóngase que en el ejemplo anterior se quiere añadir una nueva clave que represente los comentarios a un post del blog, y que tomará como valor un array de comentarios. Entonces, lo primero que hay que hacer es modificar la variable post y añadir una clave «comentarios»:

```
prueba> post.comentarios = []
[]
prueba>
```

A continuación, se actualiza el documento sustituyendo el anterior documento por el nuevo:

```
prueba> db.blog.updateOne({"titulo": "Mi primer post de mi blog"}, {$set:post})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Ahora se puede consultar y ver los cambios mediante findOne:

```
prueba> db.blog.findOne()
{
  _id: ObjectId('65e78a64e4dac6b7dfb11658'),
  titulo: 'Mi primer post de mi blog',
  contenido: 'Este es mi primer post de mi blog',
  date: ISODate('2024-03-05T21:09:24.905Z'),
  comentarios: []
}
```

- d) Borrar. El comando remove elimina de forma permanente los documentos de una base de datos. Si se llama sin parámetros, elimina todos los documentos de una colección. También puede tomar un documento especificando criterios para su eliminación, como el siguiente ejemplo, que elimina los documentos que tenga en la clave «título» el valor «Mi primer post de mi blog»:

```
prueba> db.blog.remove({"titulo": "Mi primer post de mi blog"})
{ acknowledged: true, deletedCount: 1 }
prueba>
```

Además de las operaciones básicas de manipulación de la base de datos, la Shell dispone de otros comandos útiles más orientados a manipular la propia Shell:



- a) Configuración de la Shell. Por defecto, la Shell se conecta a una instancia local de MongoDB, sin embargo, es posible conectarse a una instancia diferente o puerto especificando el hostname, puerto y base de datos: mongo hostname: puerto/nombre_base_datos.

A veces puede interesar ejecutar la Shell sin conectarse a una instancia de MongoDB, para lo que se utiliza el comando mongosh --nodb.

```
PS C:\Users\joi> docker exec -it testmongo mongosh --nodb
Current Mongosh Log ID: 65e78e3d476e22ce012d353e
Using Mongosh:          2.1.5
```

For mongosh info see: <https://docs.mongodb.com/mongodb-shell/>

>

Una vez conectado se puede conectar a una instancia de MongoDB mediante el comando new Mongo (hostname) y a continuación conectarse a la base de datos:

```
> conn=new Mongo("localhost")
mongodbs://127.0.0.1:27017/localhost?directConnection=true&serverSelectionTimeoutMS=2000
> db=conn.getDB("prueba")
prueba
prueba>
```

A partir de este momento se puede usar de forma normal la base de datos. Hay que observar que con estos comandos es posible conectarse a diferentes bases de datos o servidores en cualquier momento.

- b) Ayuda. Dado que MongoDB es una Shell basado en JavaScript, se puede encontrar información sobre su funcionamiento consultando la documentación de JavaScript. Asimismo se puede encontrar información sobre la funcionalidad de la Shell específica de MongoDB utilizando el comando help.

```
prueba> help
Shell Help:

use                                         Set current database
show                                         'show databases'/'show dbs': Print a list of all available databases.
                                                'show collections'/'show tables': Print a list of all collections for current database.
                                                'show profile': Prints system.profile information.
                                                'show users': Print a list of all users for current database.
                                                'show roles': Print a list of all roles for current database.
                                                'show log <type>': log for current connection, if type is not set uses 'global'
                                                'show logs': Print all logs.

exit                                         Quit the MongoDB shell with exit()/exit()
quit                                         Quit the MongoDB shell with quit/quit()
Mongo                                         Create a new connection and return the Mongo object. Usage: new Mongo(URI, options [optional])
connect                                       Create a new connection and return the Database object. Usage: connect(URI, username [optional], password [optional])
it                                            result of the last line evaluated; use to further iterate
version                                       Shell version
load                                           Loads and runs a JavaScript file into the current shell environment
enableTelemetry                                Enables collection of anonymous usage data to improve the mongosh CLI
disableTelemetry                                Disables collection of anonymous usage data to improve the mongosh CLI
passwordPrompt                                 Prompts the user for a password
sleep                                         Sleep for the specified number of milliseconds
print                                         Prints the contents of an object to the output
printjson                                     Alias for print()
convertShardKeyToHashed                        Returns the hashed value for the input using the same hashing function as a hashed index.
cls                                           Clears the screen like console.clear()
isInteractive                                  Returns whether the shell will enter or has entered interactive mode

For more information on usage: https://docs.mongodb.com/manual/reference/method
prueba>
```

En cuanto a la base de datos también se puede encontrar ayuda mediante el comando db.help () y en cuanto a colección mediante el comando db.Nombre_Colección.help (). También se puede encontrar información sobre una función determinada introduciendo el nombre de la misma sin paréntesis. En este caso se imprimirá el código fuente en JavaScript de la función. Por ejemplo, si se quiere ver cómo funciona la función update, se utilizaría el comando mostrado en la imagen.



```
prueba> db.prueba.update
[Function: update] AsyncFunction {
  apiVersions: [ 1, Infinity ],
  serverVersions: [ '0.0.0', '3.2.0' ],
  deprecated: true,
  returnsPromise: true,
  topologies: [ 'ReplSet', 'Sharded', 'LoadBalanced', 'Standalone' ],
  returnType: { type: 'unknown', attributes: {} },
  platforms: [ 'Compass', 'Browser', 'CLI' ],
  isDirectShellCommand: false,
  acceptsRawInput: false,
  shellCommandCompleter: undefined,
  help: [Function (anonymous)] Help
}
prueba>
```

- c) Ejecución de JavaScript. Otra característica de la Shell es la posibilidad de ejecutar archivos JavaScript. Para ello se ejecutan en la Shell los scripts de la siguiente forma:

mongo Nombre_Script1.js Nombre_Script2.js.

La Shell ejecuta cada script listado y a continuación abandona la Shell. También es posible ejecutar los scripts en una conexión que no sea la que aparece por defecto especificando primero la dirección: host, puerto y una base de datos concreta, y a continuación la lista de scripts:

mongo --quiet hostname:puerto/base_datos Nombre_Script1.js
Nombre_Script2.js.

Hay que observar en el ejemplo anterior que las opciones de la línea de comandos para ejecutar la Shell van siempre antes de la dirección. En este caso se utiliza la opción –quiet, que evita que se imprima la información de cabecera «MongoDB Shell version...». En este sentido, esta opción se puede utilizar para canalizar la salida de un script de la Shell a otro comando. También es posible imprimir a la salida estándar los scripts usando la función print().

Otra forma de ejecutar scripts desde dentro de la Shell es usar la función load ("Nombre_Script.js").

Los scripts tienen acceso a la variable db como a otras variables globales, sin embargo, no funcionan algunos de los comandos propios de MongoDB, aunque existen otras alternativas a estos en JavaScript:

use prueba	db.getSiblingDB("prueba")
show dbs	db.getMongo().getDBs()
show collections	db.getCollectionNames()

También es posible usar scripts para añadir variables en la Shell. Por ejemplo, se podría tener un script conectarseA () para realizar una inicialización que conecte a una instancia local de MongoDB que se ejecuta sobre un puerto dado y configura la variable db a esa conexión. Si se carga este script en la Shell, el script está ya definido en MongoDB:



```
prueba> var conectarseA= function (puerto, dbnombre){  
... if (!puerto) {  
... puerto=27017;  
... }  
... if (!dbnombre) {  
... dbnombre="test";  
... }  
... db= connect("localhost:"+puerto+"/"+dbnombre);  
... return db;  
... };  
  
prueba> typeof conectarseA  
function  
prueba>
```

Los scripts también pueden usarse para automatizar tareas comunes y actividades administrativas. Por defecto, la Shell mirará en el directorio en que se inició, de manera que, si el script no está en el directorio actual, se puede especificar a la Shell una dirección relativa o absoluta. Por ejemplo, si se quiere poner los scripts de la Shell en el directorio ~/mis-scripts, para cargar el script conectarseA.js se usaría el comando: load ("~/home/usuario/mis-scripts/conectarseA.js"). Hay que observar que load () no es capaz de resolver el símbolo «~».

Asimismo si se quiere usar el comando run () para ejecutar programas desde la línea de comandos de la Shell, se pasan los argumentos a la función como parámetros:

```
- run ("ls", "-l", "/home/usuario/mis-scripts/")  
- sh70352| -rw-r--r-- 1 myUser myUser 2012-12-13 13:15 defineconectarseA.js  
- sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:10 script1.js  
- sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:12 script2.js  
- sh70532| -rw-r--r-- 1 myUser myUser 2013-02-22 15:13 script3.js
```

Hay que observar que su uso es limitado, dado que la salida se formatea de forma extraña y no soporta la canalización entre diferentes funciones.

OPERACIONES CRUD

A continuación veremos las principales operaciones de manipulación de datos, más conocidas como CRUD (crear, leer, actualizar o borrar). Mediante estas operaciones, el usuario puede gestionar los datos de una forma directa a través de la Shell de comandos de MongoDB. Asimismo, se hace una especial referencia a las operaciones de modificación sobre los arrays, al tratarse de un objeto que es considerado de primer orden en MongoDB.

INSERCIÓN DE DOCUMENTOS

Para insertar un documento en una colección se usa el método insertOne:

```
prueba> db.prueba.insertOne({"titulo":"El Quijote"})  
{  
  acknowledged: true,  
  insertedId: ObjectId('65e794f5476e22ce012d3540')  
}  
prueba>
```



Esta acción añadirá al documento el campo «_id» en caso de no existir en el documento, y almacenará el mismo en MongoDB.

Cuando es necesario insertar un conjunto de documentos, se utiliza el comando `insertMany` y se puede pasar como parámetro un array con el conjunto de documentos que deben ser insertados:

```
prueba> db.prueba.insertMany([{"_id": 0}, {"_id": 1}, {"_id": 2}])
{ acknowledged: true, insertedIds: { '0': 0, '1': 1, '2': 2 } }
```

Se pueden insertar mediante un array múltiples documentos siempre que se vayan almacenar en una única colección; en caso de varias colecciones no es posible y habría que usar otras herramientas tales como mongoimport.

Hay que observar:

- Cuando se inserta usando un array, si se produce algún fallo en algún documento, se insertan todos los documentos anteriores al que tuvo el fallo, y los que hay a continuación no se insertan. Este comportamiento se puede cambiar usando la opción «`continueOnError`», que, en caso de encontrarse un error en un documento, lo salta, y continúa insertando el resto de los documentos. Esta opción no está disponible directamente en la Shell, pero sí en los drivers de los lenguajes de programación.
- Actualmente existe un límite de longitud de 48 MB para las inserciones realizadas usando un array de documentos.

Cuando se inserta un documento MongoDB, se realizan una serie de operaciones con el objetivo de evitar inconsistencias tales como:

- Se añade el campo «`_id`» en caso de no tenerlo.
- Se comprueba la estructura básica. En particular se comprueba el tamaño del documento (debe ser más pequeño de 16 MB). Para saber el tamaño de un documento se puede usar el comando `Object.bsonsize(doc)`.
- Existencia de caracteres no válidos.

BORRADO DE DOCUMENTOS

El método `remove` elimina todos los documentos de una colección, pero no elimina la colección ni la metainformación acerca de la colección:

```
prueba> db.prueba.find()
[
  { _id: ObjectId('65e794f5476e22ce012d3540'), titulo: 'El Quijote' },
  { _id: 0 },
  { _id: 1 },
  { _id: 2 }
]
prueba> db.prueba.remove({})
{ acknowledged: true, deletedCount: 4 }
prueba>
```



El método permite opcionalmente tomar una condición de búsqueda, de forma que eliminará solo aquellos documentos que encajen con la condición dada. Por ejemplo, si se quisiera eliminar todos los documentos de la colección correo.lista donde el valor para el campo «salida» es cierto, entonces se usaría el siguiente comando:

```
db.correo.lista.remove ({“salida”: true})
```

Una vez que se ha realizado el borrado no se puede dar revertir y se pierden todos los documentos borrados. A veces, si se van a borrar todos los documentos, es más rápido eliminar toda la colección en vez de los documentos. Para ello se usa el método drop:

```
db.prueba.drop ()
```

ACTUALIZACIÓN DE DOCUMENTOS

Para modificar un documento almacenado se usa el método update, que toma dos parámetros: una condición de búsqueda que localiza el documento que actualizar y un conjunto de cambios que realizar sobre el documento.

Las actualizaciones son atómicas, de manera que, si se quieren realizar dos a la vez, la primera que llegue es la primera en realizarse y a continuación se hará la siguiente.

- Reemplazamiento de documentos

El tipo de actualización más simple consiste en reemplazar un documento por otro. Por ejemplo, que se quiere cambiar el siguiente documento:

```
{  
  “nombre”: “Juan”,  
  “amigos”: 32,  
  “enemigos”: 2  
}
```

Y se quiere crear un campo «relaciones» que englobe a los campos «amigos» y «enemigos» como subdocumentos, esta operación se puede llevar a cabo con un updateOne:

```
prueba> juan.relaciones = {"amigos":juan.amigos, "enemigos":juan.enemigos};  
{ amigos: 32, enemigos: 2 }  
prueba> juan.PrimerNombre=juan.nombre  
Juan  
prueba> delete juan.amigos  
true  
prueba> delete juan.enemigos  
true  
prueba> delete juan.nombre  
true  
  
prueba> db.prueba.updateOne({"nombre":"Juan"}, {$set:{juan}});  
{  
  acknowledged: true,  
  insertedId: null,  
  matchedCount: 1,  
  modifiedCount: 1,  
  upsertedCount: 0  
}  
prueba> db.prueba.findOne()  
{  
  _id: ObjectId('65e79904476e22ce012d3541'),  
  nombre: 'Juan',  
  amigos: 32,  
  enemigos: 2,  
  juan: {  
    _id: ObjectId('65e79904476e22ce012d3541'),  
    relaciones: { amigos: 32, enemigos: 2 },  
    PrimerNombre: 'Juan'  
  }  
}  
prueba>
```



Un error común es cuando encaja más de un documento con el criterio de búsqueda y se crea un campo duplicado «`_id`» con el segundo parámetro. En este caso, la base de dato genera un error y ningún documento es actualizado. Por ejemplo, supóngase que se crean varios documentos con el mismo valor para el campo «`nombre`», sea «Juan», y se quiere actualizar el valor del campo «`edad`» de uno de ellos (se quiere aumentar el valor de la edad del segundo «Juan»):

```
prueba> db.prueba.find()
[
  {
    _id: ObjectId('65e79bbb476e22ce012d3546'),
    nombre: 'Juan',
    edad: 32
  },
  {
    _id: ObjectId('65e79bc7476e22ce012d3547'),
    nombre: 'Juan',
    edad: 33
  },
  {
    _id: ObjectId('65e79bcc476e22ce012d3548'),
    nombre: 'Juan',
    edad: 45
  }
]
prueba> juan=db.prueba.findOne({"nombre":"Juan", "edad":33})
{ _id: ObjectId('65e79bc7476e22ce012d3547'), nombre: 'Juan', edad: 33 }
prueba> juan.edad++;
33
prueba> db.prueba.updateOne({"nombre": "Juan"}, {$set:juan});
MongoServerError: Performing an update on the path '_id' would modify the immutable field '_id'
prueba>
```

Se produce un error, dado que el método `updateOne` busca un documento que encaje con la condición de búsqueda y el primero que encuentra es el referido al «Juan», que tiene 32 años. Intenta cambiar ese documento por el actualizado, y se encuentra que, si hace el cambio, habría dos documentos con el mismo «`_id`», y eso no es posible (el «`_id`» debe ser único). Para evitar estas situaciones, lo mejor es usar el método `updateOne` con el campo «`_id`», que es único. En el ejemplo anterior se podría hacer la actualización si se hiciera de la siguiente forma:

```
prueba> db.prueba.updateOne({_id: ObjectId("65e79bc7476e22ce012d3547")}, {$set:juan});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba>
```

Otra ventaja de usar el campo «`_id`» es que el documento está indexado por este campo.

- Modificadores

En muchas ocasiones, el tipo de actualización que se quiere realizar consiste en añadir, modificar o eliminar claves, manipular arrays y documentos embebidos, etc. Para estos casos se va a usar un conjunto de operadores de modificación.

- `$inc`. Este operador permite cambiar el valor numérico de una clave que ya existe incrementando su valor por el especificado junto al operador, o bien puede crear una clave que no existía inicializándola al valor dado.



Por ejemplo, supóngase que se mantienen los datos estadísticos de un sitio web en una colección de manera que se incrementa un contador cada vez que alguien visita una página. Para ello se tiene un documento que almacena la URL y el número de visitas de la página.

```
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e79db6476e22ce012d3549'),
  URL: 'www.ejemplo.com',
  visitas: 35
}
prueba>
```

Cada vez que alguien visita una página se busca la página a partir de su URL y se incrementa el campo de «visitas» con el modificador «\$inc», que incrementa el campo dado en el valor descrito:

```
prueba> db.prueba.updateOne({"URL": "www.ejemplo.com"}, {"$inc": {"visitas": 1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e79db6476e22ce012d3549'),
  URL: 'www.ejemplo.com',
  visitas: 36
}
prueba> |
```

También sería posible incrementar el valor por un valor mayor que 1:

```
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e79db6476e22ce012d3549'),
  URL: 'www.ejemplo.com',
  visitas: 36
}
prueba> db.prueba.updateOne({"URL": "www.ejemplo.com"}, {"$inc": {"visitas": 30}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e79db6476e22ce012d3549'),
  URL: 'www.ejemplo.com',
  visitas: 66
}
prueba>
```

Y de la misma forma se podría decrementar usando números negativos:



```
prueba> db.prueba.updateOne({"URL": "www.ejemplo.com"}, {"$inc": {"visitas": -37}})  
{  
    acknowledged: true,  
    insertedId: null,  
    matchedCount: 1,  
    modifiedCount: 1,  
    upsertedCount: 0  
}  
prueba> db.prueba.findOne()  
{  
    _id: ObjectId('65e79db6476e22ce012d3549'),  
    URL: 'www.ejemplo.com',  
    visitas: 29  
}  
prueba>
```

Y por ejemplo, se podría añadir un nuevo campo para indicar el número de enlaces de la página:

```
prueba> db.prueba.updateOne({"URL": "www.ejemplo.com"}, {"$inc": {"enlaces": 30}})  
{  
    acknowledged: true,  
    insertedId: null,  
    matchedCount: 1,  
    modifiedCount: 1,  
    upsertedCount: 0  
}  
prueba> db.prueba.findOne()  
{  
    _id: ObjectId('65e79db6476e22ce012d3549'),  
    URL: 'www.ejemplo.com',  
    visitas: 29,  
    enlaces: 30  
}  
prueba>
```

Hay que observar:

- Cuando se usan operadores de modificación el valor del campo «_id», no puede ser cambiado (en cambio, cuando se reemplaza un documento entero, sí es posible cambiar el campo «_id»). Sin embargo, los valores para cualquier otra clave incluyendo claves indexadas únicas sí pueden ser modificadas.
- Este operador solo puede ser usado con números enteros, enteros largos o double, de manera que, si se usa con otro tipo de valores (incluido los tipos que algunos lenguajes tratan como números tales como booleanos, cadenas de números, nulos, etc.), producirá un fallo. En el ejemplo se intenta incrementar un campo con un valor no numérico:

```
< prueba> db.prueba.updateOne({"URL": "www.ejemplo.com"}, {"$inc": {"enlaces": "20"}})  
MongoServerError: Cannot increment with non-numeric argument: {enlaces: "20"}  
< prueba>
```

- «\$set» y «\$unset». Este operador establece un valor para un campo dado, y, si el campo dado no existe, lo crea. En este sentido, es útil para modificar el esquema de un documento o añadir claves definidas por el usuario. Por ejemplo, supóngase que se tiene el perfil de usuario almacenado en un documento:



```
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a09f476e22ce012d354a'),
  nombre: 'Juan',
  edad: 34,
  sexo: 'Varon',
  localizacion: 'Buenos Aires'
}
prueba>
```

Si el usuario desea añadir un campo sobre su libro favorito, se podría hacer usando el modificador «\$set»:

```
prueba> db.prueba.updateOne({"_id":ObjectId("65e7a09f476e22ce012d354a")}, {"$set":{"libroFavorito": "Martin Fierro"}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a09f476e22ce012d354a'),
  nombre: 'Juan',
  edad: 34,
  sexo: 'Varon',
  localizacion: 'Buenos Aires',
  libroFavorito: 'Martin Fierro'
}
prueba>
```

Nuevamente, si el usuario desea cambiar el valor del campo sobre su libro favorito por otro valor, se podría hacer usando también el modificador «\$set»:

```
prueba> db.prueba.updateOne({"_id":ObjectId("65e7a09f476e22ce012d354a")}, {"$set":{"libroFavorito": "El Quijote"}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a09f476e22ce012d354a'),
  nombre: 'Juan',
  edad: 34,
  sexo: 'Varon',
  localizacion: 'Buenos Aires',
  libroFavorito: 'El Quijote'
}
prueba>
```

También con el modificador «\$set» es posible cambiar el tipo de un campo que se modifica. Por ejemplo, si se quiere que el campo «libroFavorito» sea un array en vez de un valor único, también puede usarse el modificador «\$set»:

```
prueba> db.prueba.updateOne({"_id":ObjectId("65e7a09f476e22ce012d354a")}, {"$set":{"libroFavorito": ["Martin Fierro","El Quijote"]}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a09f476e22ce012d354a'),
  nombre: 'Juan',
  edad: 34,
  sexo: 'Varon',
  localizacion: 'Buenos Aires',
  libroFavorito: [ 'Martin Fierro', 'El Quijote' ]
}
prueba>
```

También mediante el operador «\$set» es posible realizar cambios en documentos embebidos, para lo cual solo es necesario indicar el campo en el que se encuentran. Por ejemplo, considerar el siguiente documento:



```
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a2f3476e22ce012d354b'),
  titulo: 'Post de un blog',
  contenido: 'el contenido',
  autor: { nombre: 'Isabel', email: 'isa@prueba.com' }
}
prueba>
```

Se quiere cambiar el valor del campo del nombre del autor del post, entonces se haría lo siguiente:

```
prueba> db.prueba.update({ "autor.nombre": "Isabel"}, {"$set": {"autor.nombre": "Isabel Sanz"}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a2f3476e22ce012d354b'),
  titulo: 'Post de un blog',
  contenido: 'el contenido',
  autor: { nombre: 'Isabel Sanz', email: 'isa@prueba.com' }
}
prueba>
```

Hay que observar:

- Existe un operador denominado «\$unset» que permite eliminar campos de un documento. En el ejemplo anterior, si se quiere eliminar el campo «libroFavorito»:

```
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a3e9476e22ce012d354c'),
  nombre: 'Juan',
  edad: 34,
  sexo: 'Varón',
  localizacion: 'Buenos Aires',
  libroFavorito: [ 'Martin Fierro', 'El Quijote' ]
}
prueba> db.prueba.updateOne({ "_id": ObjectId("65e7a3e9476e22ce012d354c") }, {"$unset": {"libroFavorito": 1}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
prueba> db.prueba.findOne()
{
  _id: ObjectId('65e7a3e9476e22ce012d354c'),
  nombre: 'Juan',
  edad: 34,
  sexo: 'Varón',
  localizacion: 'Buenos Aires'
}
prueba>
```

- Para añadir, modificar o eliminar claves se debe usar siempre los modificadores «\$». En este sentido, hay que observar que, si intentara hacer un cambio en las claves con un comando como el siguiente: db.prueba.update (criterio, {"edad": "país"}), tendría como efecto reemplazar el documento que encaje con el criterio de búsqueda por el documento {"edad": "país"}.



UNIDAD 7

TABLAS CRONOLÓGICAS

Cuando se modifican datos en tablas, normalmente se pierde cualquier rastro del estado anterior a la modificación de las filas. Sólo se puede acceder al estado actual. ¿Qué pasa si necesitamos tener acceso a información histórica de los datos? Tal vez necesitemos estos estados para una auditoría, un análisis en punto de tiempo determinado, comparando estados actuales con estados antiguos, cambiando lentamente las dimensiones (https://en.wikipedia.org/wiki/Slowly_changing_dimension), restaurar un estado anterior de filas debido a la supresión o actualización accidental, y así sucesivamente.

Podríamos intentar armar una solución manual personalizada, basada en Triggers. Pero por suerte, a partir de MS SQL Server 2016, se puede utilizar una característica llamada **tablas cronológicas versionadas por el sistema**. Esta característica proporciona una solución que es a la vez más simple y eficiente que una personalizada.

Una tabla cronológica versionada por el sistema tiene dos columnas que representan el período de validez de la fila, además de una tabla vinculada de historial con un esquema en espejo, que contiene estados anteriores de las filas modificadas. Cuando necesitamos modificar datos, interactuamos con la tabla actual, realizando instrucciones normales de modificación. SQL Server actualiza automáticamente las columnas de período y mueve las versiones anteriores de filas a la tabla de historial. Cuando necesitamos consultar datos, si deseamos el estado actual, simplemente consultamos la tabla actual como de costumbre. Si necesitamos acceso a estados anteriores, también consultamos la tabla actual, pero agregamos una cláusula que indica que deseamos ver un estado anterior de la tabla o un período de tiempo determinado. SQL Server consulta las tablas actuales e históricas detrás escena como sea necesario.

El estándar SQL admite tres tipos de tablas cronológicas:

- Las tablas cronológicas versionadas por el sistema que dependen del momento de la transacción para definir el período de validez de una fila.
- Las tablas de período de validez basados en aplicación, que se basan en la definición de aplicación para determinar el período de validez de una fila. Esto significa que puede definir un período de validez que será efectivo en el futuro.
- Bicronológicas, que combinan los dos tipos que acabamos de mencionar (transacción y período de aplicación).

SQL Server 2016 admite sólo tablas cronológicas versionadas por el sistema. Esperemos que Microsoft añada soporte para tablas de período de aplicación y tablas bicronológicas en futuras versiones de SQL Server.

CREACIÓN DE TABLAS

Cuando creamos una tabla cronológica versionada por el sistema, debemos asegurarnos de que la definición de la tabla tenga todos los elementos siguientes:

- Una clave primaria.
- Dos columnas definidas como *DATETIME2* con cualquier precisión, que no aceptan NULL y representan el inicio y el final del período de validez de la fila en la zona horaria UTC.
- Una columna de inicio que debe ser marcada con la opción *GENERATED ALWAYS AS ROW START*
- Una columna final que debe ser marcada con la opción *GENERATED ALWAYS AS ROW END*
- La designación de las columnas de tiempo con la opción *FOR SYSTEM_TIME (<startcol>, <endcol>)*
- La opción de tabla *SYSTEM_VERSIONING* debe ser puesta en ON



- Una tabla de historial vinculada (que SQL Server puede crear por nosotros) para mantener los estados anteriores de las filas modificadas.

Opcionalmente, podemos marcar las columnas de período como ocultas para que cuando estemos consultando la tabla con **SELECT *** no sean devueltas y cuando se está insertando datos sean ignoradas.

El siguiente código nos permite crear una tabla cronológica versionada por el sistema que llamaremos *Empleados* y una tabla de historia vinculada llamada *EmpleadosHistoria*:

```
CREATE TABLE dbo.Empleados
(
empid INT NOT NULL
CONSTRAINT PK_Empleados PRIMARY KEY NONCLUSTERED,
nombreemp VARCHAR(25) NOT NULL,
departamento VARCHAR(50) NOT NULL,
sueldo NUMERIC(10, 2) NOT NULL,
sysstart DATETIME2(0)
GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
sysend DATETIME2(0)
GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,
PERIOD FOR SYSTEM_TIME (sysstart, sysend),
INDEX ix_Empleados CLUSTERED(empid, sysstart, sysend)
) WITH (
SYSTEM_VERSIONING =
ON (
HISTORY_TABLE =
dbo.EmpleadosHistoria )
);
```

Revisen la lista de elementos necesarios y asegúrense de identificarlos en el código.

Suponiendo que la tabla de historial no existe cuando ejecuta este código, SQL Server la crea por nosotros. Si no especificamos un nombre para la tabla, SQL Server asigna uno con la forma *MSSQL_TemporalHistoryFor_<object_id>*, donde *object_id* es el ID de objeto de la tabla actual. SQL Server crea la tabla de historial con un esquema duplicado de la tabla actual, pero con las siguientes diferencias:

- Sin clave primaria
- Un índice agrupado en (*<endcol>*, *<startcol>*), con compresión de página, si es posible
- Columnas de período que no están marcados con las opciones especiales, como *GENERATED ALWAYS AS ROW START/END o HIDDEN*
- Sin designación de las columnas de período con la opción *PERIOD FOR SYSTEM_TIME*
- La tabla de historial no está marcada con la opción *SYSTEM_VERSIONING*

Si la tabla de historial ya existe cuando se crea la tabla actual, SQL Server valida la consistencia tanto del esquema (como se acaba de describir) como de los datos (sin superposición de períodos). Si la tabla de historial no pasa las comprobaciones de coherencia, SQL Server producirá un error en tiempo DDL y no creará la tabla actual. Opcionalmente podemos indicar que no deseamos que SQL Server realice la comprobación de coherencia de datos.

Si examinamos el árbol de objetos en el Explorador de objetos en SQL Server Management Studio (SSMS), encontraremos la tabla *Empleados* marcada como (*System-versioned*) y por debajo de ella la tabla vinculada *EmpleadosHistoria* marcada como (*History*), como se muestra en la siguiente imagen.



Object Explorer

Connect ▾

- TSQLV6ES
 - Database Diagrams
 - Tables
 - System Tables
 - FileTables
 - External Tables
 - Graph Tables
 - dbo.Empleados (System-Versioned)
 - dbo.EmpleadosHistoria (History)
 - Columns
 - empid (int, not null)
 - nombreemp (varchar(25), not null)
 - departamento (varchar(50), not null)
 - sueldo (numeric(10,2), not null)
 - sysstart (datetime2(0), not null)
 - sysend (datetime2(0), not null)
 - Constraints
 - Indexes
 - Statistics
 - Columns
 - empid (PK, int, not null)
 - nombreemp (varchar(25), not null)
 - departamento (varchar(50), not null)
 - sueldo (numeric(10,2), not null)
 - sysstart (datetime2(0), not null)
 - sysend (datetime2(0), not null)
 - Keys
 - Constraints
 - Triggers
 - Indexes
 - Statistics
 - dbo.Nums
 - Estadisticas.Pruebas
 - Estadisticas.Resultados
 - Produccion.Categorias
 - Produccion.Productos
 - Produccion.Proveedores
 - RH.Empleados
 - Ventas.Clientes
 - Ventas.DetallesOrden
 - Ventas.Ordenes
 - Ventas.Ordenes2
 - Ventas.Transportes
 - Dropped Ledger Tables
 - Views
 - External Resources
 - Synonyms

También podemos convertir una tabla no cronológica existente que ya tiene datos en una cronológica.



Por ejemplo, supongamos que tenemos una tabla llamada *Empleados* en la base de datos y deseamos convertirla en una tabla cronológica. Primero modificamos la tabla, agregando las columnas del período y designándolos como tales usando el código siguiente (pero no ejecutaremos realmente el código porque nuestra tabla *Empleados* ya es cronológica):

```
ALTER TABLE dbo.Empleados ADD
sysstart DATETIME2(0) GENERATED ALWAYS AS ROW START HIDDEN NOT NULL
CONSTRAINT DFT_Empuestos_sysstart DEFAULT('19000101'),
sysend DATETIME2(0) GENERATED ALWAYS AS ROW END HIDDEN NOT NULL
CONSTRAINT DFT_Empuestos_sysend DEFAULT('99991231 23:59:59'),
PERIOD FOR SYSTEM_TIME (sysstart, sysend);
```

Observemos los valores predeterminados que establecen el período de validez para las filas existentes. Podemos decidir lo que queramos como hora de inicio del período de validez, siempre y cuando no sea en el futuro. El fin tiene que ser el valor máximo soportado en el tipo de datos.

A continuación, modificamos la tabla para habilitar el control de versiones del sistema y vincularla a una tabla de historial utilizando el siguiente código (nuevamente, en realidad no ejecutar este código en nuestro caso):

```
ALTER TABLE dbo.Empleados
SET ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.EmpleadosHistoria ) );
```

Si marcamos las columnas de período como ocultas, cuando consultemos la tabla con **SELECT *** SQL Server no las devolverá. Intentemos esto con nuestra tabla *Empleados* mediante la ejecución del siguiente código:

```
SELECT *
FROM dbo.Empleados;
empid      nombreemp      departamento      sueldo
-----
(0 rows affected)
```

Si deseamos devolver las columnas de período, tenemos que mencionarlas explícitamente en la lista **SELECT**

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.Empleados;
empid      nombreemp      departamento      sueldo      sysstart      sysend
-----
(0 rows affected)
```

SQL Server permite realizar cambios de esquema en una tabla cronológica sin necesidad de deshabilitar el versionado del sistema primero. Realizamos el cambio de esquema a la tabla actual y SQL Server lo aplica a las tablas actual e histórica. Naturalmente, si deseamos agregar una columna que no admite *NULL*, deberemos agregarla con una restricción de *DEFAULT*. Por ejemplo, supongamos que queremos agregar una columna que no admita *NULL* llamada *fechaalta* a nuestra tabla *Empleados* y usando la fecha 1 de enero de 1900 como el valor *DEFAULT*. Para hacerlo, ejecutamos el siguiente código:

```
ALTER TABLE dbo.Empleados
ADD fechaalta DATE NOT NULL
CONSTRAINT DFT_Empuestos_fechaalta DEFAULT('19000101');
```

A continuación, podemos actualizar la fecha de contratación de los empleados existentes según sea necesario.

Consultamos la tabla *Empleados* después de añadir la columna de la *fechaalta*:

```
SELECT *
FROM dbo.Empleados;
```



Obtendremos el resultado siguiente:

```
empid      nombreemp      departamento      sueldo      fechaalta
-----      -----      -----      -----
(0 rows affected)
```

Consultamos la tabla *EmpleadosHistoria*:

```
SELECT *
FROM dbo.EmpleadosHistoria;
```



```
empid      nombreemp      departamento      sueldo      sysstart      sysend      fechaalta
-----      -----      -----      -----
(0 rows affected)
```

El resultado de la consulta tiene el campo *fechaalta*.

SQL Server agrega la columna *fechaalta* a ambas tablas, pero se añadió la restricción *DEFAULT* sólo en la tabla actual. Sin embargo, si hubiera alguna fila en la tabla de historial, SQL Server habría asignado el valor default a la columna de *fechaalta* en esas filas.

Supongamos que deseamos eliminar la columna *fechaalta* de ambas tablas. Primero eliminamos la restricción *DEFAULT* de la tabla actual ejecutando el siguiente código:

```
ALTER TABLE dbo.Empleados
DROP CONSTRAINT DFT_Empleados_fechaalta;
```

En segundo lugar, se quita la columna de la tabla actual ejecutando el código siguiente:

```
ALTER TABLE dbo.Empleados
DROP COLUMN fechaalta;
```

SQL Server elimina la columna de ambas tablas.

MODIFICACIÓN DE DATOS

La modificación de tablas cronológicas es similar a la modificación de tablas normales. Sólo modificamos la tabla actual con *INSERT*, *UPDATE*, *DELETE* y *MERGE*. (No hay soporte para *TRUNCATE* de tablas cronológicas en SQL Server 2016.) Detrás de escena, SQL Server actualiza las columnas de período y mueve las filas a la tabla de historial según sea necesario. Recordemos que las columnas de período reflejan el período de validez de la fila en la zona horaria UTC.

Si definimos las columnas de período como ocultas, como en nuestro caso, simplemente las ignoramos en las instrucciones *INSERT*. Si no las definimos como ocultas, siempre y cuando sigamos las mejores prácticas y mencionemos explícitamente los nombres de las columnas de destino, podemos ignorarlas. Si no las definimos como ocultas y no mencionamos los nombres de las columnas de destino, deberemos especificar la palabra clave *DEFAULT* como el valor para ellas.

En los siguientes ejemplos, vamos a ver modificaciones a la tabla *Empleados* y veremos el momento en el tiempo, en la zona horaria UTC, en la que los aplicamos. Naturalmente, el tiempo de modificación será diferente cuando ejecuten los ejemplos de código, por lo que podría ser una buena idea que tomen nota del momento en que realizan las consultas. Podemos consultar la función *SYSUTCDATETIME* para obtener esta información.

Ejecutamos el siguiente código para agregar algunas filas de la tabla *Empleados* (cuando ejecuté el ejemplo era 12/10/2017 01:23:14):



```
INSERT INTO dbo.Empleados(empid, nombreemp, departamento, sueldo)
VALUES(1, 'Sara', 'IT', 50000.00),
(2, 'Don', 'HR', 45000.00),
(3, 'Judy', 'Sales', 55000.00),
(4, 'Yael', 'Marketing', 55000.00),
(5, 'Sven', 'IT', 45000.00),
(6, 'Paul', 'Sales', 40000.00);
```

Consultamos los datos en las tablas actuales e históricas para ver lo que hizo SQL Server detrás de escena.

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.Empleados;
```

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.EmpleadosHistoria;
```

La tabla actual tiene las seis nuevas filas, con la columna de *sysstart* que refleja el momento de la modificación y *sysend* con el máximo valor posible en el tipo con la precisión elegido:

empid	nombreemp	departamento	sueldo	sysstart	sysend
1	Sara	IT	50000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
2	Don	HR	45000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
5	Sven	IT	45000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
6	Paul	Sales	40000.00	2017-10-12 01:23:14	9999-12-31 23:59:59

(6 rows affected)

El período de validez indica que las filas se consideran válidas desde el momento en que fueron insertadas y sin límite final.

La tabla de historial está vacía en este punto:

empid	nombreemp	departamento	sueldo	sysstart	sysend

(0 rows affected)

Ejecutemos el código siguiente para eliminar la fila donde el ID de empleado es 6 (la ejecuté el 12/10/2017 01:30:09):

```
DELETE FROM dbo.Empleados
WHERE empid = 6;
```

SQL Server mueve la fila eliminada a la tabla de historial, estableciendo el valor *sysend* con el momento del borrado. A continuación, se muestra el contenido de la tabla actual en este punto:

empid	nombreemp	departamento	sueldo	sysstart	sysend
1	Sara	IT	50000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
2	Don	HR	45000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
5	Sven	IT	45000.00	2017-10-12 01:23:14	9999-12-31 23:59:59

(5 rows affected)

A continuación, se muestra el contenido de la tabla de historial:

empid	nombreemp	departamento	sueldo	sysstart	sysend
6	Paul	Sales	40000.00	2017-10-12 01:23:14	2017-10-12 01:30:09

(1 row affected)

La actualización de una fila se trata como una eliminación más una inserción. SQL Server mueve la versión antigua de la fila a la tabla de historial con el momento de transacción como la hora de finalización del período, y mantiene la versión actual de la fila en la tabla actual con el tiempo de transacción como la hora de inicio del período y el



valor máximo en el tipo como el momento de finalización del período. Por ejemplo, ejecutemos la siguiente consulta para aumentar el sueldo de todos los empleados de IT en un cinco por ciento (se ejecutó el 12/10/2017 01:35:44):

```
UPDATE dbo.Empleados  
SET sueldo *= 1.05  
WHERE departamento = 'IT';
```

A continuación, se muestra el contenido de la tabla actual después de la actualización:

empid	nombreemp	departamento	suelo	sysstart	sysend
1	Sara	IT	52500.00	2017-10-12 01:35:44	9999-12-31 23:59:59
2	Don	HR	45000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
3	Judy	Sales	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
5	Sven	IT	47250.00	2017-10-12 01:35:44	9999-12-31 23:59:59

(5 rows affected)

Observemos los valores de la columna sueldos y plazos para los empleados de IT.

A continuación, se muestra el contenido de la tabla de historial:

empid	nombreemp	departamento	suelo	sysstart	sysend
6	Paul	Sales	40000.00	2017-10-12 01:23:14	2017-10-12 01:30:09
1	Sara	IT	50000.00	2017-10-12 01:23:14	2017-10-12 01:35:44
5	Sven	IT	45000.00	2017-10-12 01:23:14	2017-10-12 01:35:44

(3 rows affected)

Los tiempos de modificación en los registros que SQL Server pone en las columnas de período reflejan la hora de inicio de la transacción. Si tenemos una transacción de larga duración que comenzó en el punto en el tiempo T1 y terminó en T2, SQL Server registrará T1 como el momento de modificación para todas las sentencias. Por ejemplo, ejecutemos este código para abrir una transacción explícita y cambiar el departamento de empleado 5 a Ventas (se ejecutó 12/10/2017 01:43:20):

```
BEGIN TRAN;  
UPDATE dbo.Empleados  
SET departamento = 'Sales'  
WHERE empid = 5;
```

Esperamos unos segundos y luego ejecutamos el siguiente código para cambiar el departamento de empleado 3 a IT (se ejecutó 12/10/2017 01:44:49):

```
UPDATE dbo.Empleados  
SET departamento = 'IT'  
WHERE empid = 3;  
COMMIT TRAN;
```

A continuación, se muestra el contenido de la tabla actual después de ejecutar esta transacción:

empid	nombreemp	departamento	suelo	sysstart	sysend
1	Sara	IT	52500.00	2017-10-12 01:35:44	9999-12-31 23:59:59
2	Don	HR	45000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
3	Judy	IT	55000.00	2017-10-12 01:43:20	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2017-10-12 01:23:14	9999-12-31 23:59:59
5	Sven	Sales	47250.00	2017-10-12 01:43:20	9999-12-31 23:59:59

(5 rows affected)

A continuación, se muestra el contenido de la tabla de historial en este punto:



empid	nombreemp	departamento	sueldo	sysstart	sysend
6	Paul	Sales	40000.00	2017-10-12 01:23:14	2017-10-12 01:30:09
1	Sara	IT	50000.00	2017-10-12 01:23:14	2017-10-12 01:35:44
5	Sven	IT	45000.00	2017-10-12 01:23:14	2017-10-12 01:35:44
3	Judy	Sales	55000.00	2017-10-12 01:23:14	2017-10-12 01:43:20
5	Sven	IT	47250.00	2017-10-12 01:35:44	2017-10-12 01:43:20

(5 rows affected)

Observemos que, para las filas modificadas, el tiempo de modificación (*sysstart* para las filas actuales y *sysend* para las filas de historia) refleja el momento de inicio de la transacción.

CONSULTA DE DATOS

Consultar datos en tablas cronológicas es simple y elegante. Si deseamos consultar el estado actual de los datos, simplemente consultamos la tabla actual con una consulta de tabla normal. Si queremos consultar un estado pasado de los datos, también podemos consultar la tabla actual, pero agregamos una cláusula llamada *FOR SYSTEM_TIME* y una subcláusula que indica el punto validez o período de tiempo en que estemos interesados.

Antes de examinar las especificaciones de la consulta de tablas cronológicas, ejecutemos el siguiente código para re-crear las tablas *Empleados* y *EmpleadosHistoria* y los completamos con datos, para que cuando ejecutemos los ejemplos podamos obtener los mismos resultados que en el apunte:

```
USE TSQLV6ES;
-- Eliminamos las tablas si existen
IF OBJECT_ID(N'dbo.Empleados', N'U') IS NOT NULL
BEGIN
IF OBJECTPROPERTY(OBJECT_ID(N'dbo.Empleados', N'U'), N'TableTemporalType') = 2
ALTER TABLE dbo.Empleados SET ( SYSTEM_VERSIONING = OFF );
DROP TABLE IF EXISTS dbo.EmpleadosHistoria, dbo.Empleados;
END;
GO
-- Creamos y populamos las tablas
CREATE TABLE dbo.Empleados
(
empid INT NOT NULL
CONSTRAINT PK_Empelados PRIMARY KEY NONCLUSTERED,
nombreemp VARCHAR(25) NOT NULL,
departamento VARCHAR(50) NOT NULL,
sueldo NUMERIC(10, 2) NOT NULL,
sysstart DATETIME2(0) NOT NULL,
sysend DATETIME2(0) NOT NULL,
INDEX ix_Empelados CLUSTERED(empid, sysstart, sysend)
);
INSERT INTO dbo.Empleados(empid, nombreemp, departamento, sueldo, sysstart, sysend)
VALUES
(1 , 'Sara' , 'IT' , 52500.00, '2022-02-16 17:20:02' , '9999-12-31 23:59:59'),
(2 , 'Don' , 'HR' , 45000.00, '2022-02-16 17:08:41' , '9999-12-31 23:59:59'),
(3 , 'Judy' , 'IT' , 55000.00, '2022-02-16 17:28:10' , '9999-12-31 23:59:59'),
(4 , 'Yael' , 'Marketing' , 55000.00, '2022-02-16 17:08:41' , '9999-12-31 23:59:59'),
(5 , 'Sven' , 'Sales' , 47250.00, '2022-02-16 17:28:10' , '9999-12-31 23:59:59');
-- Creamos y populamos la tabla EmpleadosHistoria
CREATE TABLE dbo.EmpleadosHistoria
(
empid INT NOT NULL,
nombreemp VARCHAR(25) NOT NULL,
departamento VARCHAR(50) NOT NULL,
sueldo NUMERIC(10, 2) NOT NULL,
sysstart DATETIME2(0) NOT NULL,
sysend DATETIME2(0) NOT NULL,
INDEX ix_EmpeladosHistoria CLUSTERED(sysend, sysstart)
WITH (DATA_COMPRESSION = PAGE)
```



```
);

INSERT INTO dbo.EmpleadosHistoria(empid, nombreemp, departamento, sueldo, sysstart, sysend)
VALUES
(6 , 'Paul' , 'Sales' , 40000.00, '2022-02-16 17:08:41' , '2022-02-16 17:15:26'),
(1 , 'Sara' , 'IT' , 50000.00, '2022-02-16 17:08:41' , '2022-02-16 17:20:02'),
(5 , 'Sven' , 'IT' , 45000.00, '2022-02-16 17:08:41' , '2022-02-16 17:20:02'),
(3 , 'Judy' , 'Sales' , 55000.00, '2022-02-16 17:08:41' , '2022-02-16 17:28:10'),
(5 , 'Sven' , 'IT' , 47250.00, '2022-02-16 17:20:02' , '2022-02-16 17:28:10');

-- Habilitamos el versionado
ALTER TABLE dbo.Empleados ADD PERIOD FOR SYSTEM_TIME (sysstart, sysend);
ALTER TABLE dbo.Empleados ALTER COLUMN sysstart ADD HIDDEN;
ALTER TABLE dbo.Empleados ALTER COLUMN sysend ADD HIDDEN;
ALTER TABLE dbo.Empleados
SET ( SYSTEM_VERSIONING = ON ( HISTORY_TABLE = dbo.EmpleadosHistoria ) );
```

De esta manera, las salidas de las consultas en tu entorno serán las mismas que en el apunte.

Sólo recordemos que cuando una consulta no tiene *ORDER BY*, no hay garantía de orden de presentación específico en la salida. Así que es posible que el orden de las filas que obtengan cuando ejecutan las consultas sean diferentes que en el apunte.

Como se ha mencionado, si deseamos consultar el estado actual de las filas, simplemente consultamos la tabla actual:

```
SELECT *
FROM dbo.Empleados;
```

Esta consulta genera la siguiente salida:

empid	nombreemp	departamento	sueldo
1	Sara	IT	52500.00
2	Don	HR	45000.00
3	Judy	IT	55000.00
4	Yael	Marketing	55000.00
5	Sven	Sales	47250.00

Recordemos que debido a que las columnas de tiempo se definen como ocultas, una consulta *SELECT ** no las devuelve. Aquí, utilizamos *SELECT ** para fines de ilustración, pero recordemos que no es una buena práctica.

Si deseamos ver un estado pasado de los datos, apuntaremos a un cierto punto o período de tiempo, consultando la tabla actual seguido por la cláusula *FOR SYSTEM_TIME*, además de una subcláusula con datos específicos. SQL Server recuperará los datos tanto de la tabla actual como del historial, según sea necesario.

La siguiente es la sintaxis para utilizar la cláusula *FOR SYSTEM_TIME*:

```
SELECT ... FROM <table_or_view> FOR SYSTEM_TIME <subclause> AS <alias>;
```

De las cinco subcláusulas que admite la cláusula *SYSTEM_TIME*, es probable que utilicemos el *AS OF* más a menudo. Lo usamos para solicitar ver los datos en un punto específico en el tiempo que especifiquemos. La sintaxis de este subcláusula es *FOR SYSTEM_TIME AS OF <valor datetime2>*. La entrada puede ser una constante, una variable o un parámetro. Digamos que la entrada es una variable llamada *@datetime*. Obtenremos las filas donde *@datetime* está entre *sysstart* y *sysend*. En otras palabras, el período de validez comienza en o antes de *@datetime* y termina después de *@datetime*. El siguiente predicado identifica las filas que califican:

```
sysstart <= @datetime AND sysend > @datetime
```

Ejecutemos el código siguiente para devolver las filas de Empleados al momento 2022-02-16 17:00:00:



```
SELECT *
FROM dbo.Empleados FOR SYSTEM_TIME AS OF '2022-02-16 17:00:00';
```

Obtendremos un resultado vacío porque la primera inserción se realizó en 2022-02-16 17:08:41:

```
empid      nombreemp      departamento      sueldo
-----
(0 rows affected)
```

Consultamos la tabla de nuevo, esta vez a partir de 2022-02-16 17:10:00:

```
SELECT *
FROM dbo.Empleados FOR SYSTEM_TIME AS OF '2022-02-16 17:10:00';
```

Obtendremos el resultado siguiente:

```
empid      nombreemp      departamento      sueldo
-----
2          Don            HR                45000.00
4          Yael           Marketing        55000.00
6          Paul           Sales             40000.00
1          Sara            IT                50000.00
5          Sven            IT                45000.00
3          Judy            Sales             55000.00
(6 rows affected)
```

También podemos consultar varias instancias de la misma tabla, comparando diferentes estados de datos en diferentes momentos. Por ejemplo, la consulta siguiente devuelve el porcentaje de aumento de sueldo de los empleados que tuvieron un aumento salarial entre dos momentos:

```
SELECT T2.empid, T2.nombreemp,
CAST( (T2.sueldo / T1.sueldo - 1.0) * 100.0 AS NUMERIC(10, 2) ) AS pct
FROM dbo.Empleados FOR SYSTEM_TIME AS OF '2022-02-16 17:10:00' AS T1
INNER JOIN dbo.Empleados FOR SYSTEM_TIME AS OF '2022-02-16 17:25:00' AS T2
ON T1.empid = T2.empid
AND T2.sueldo > T1.sueldo;
```

Este código genera la siguiente salida:

```
empid      nombreemp      pct
-----
1          Sara            5.00
5          Sven            5.00
(2 rows affected)
```

La subcláusula *FROM @start TO @end* devuelve las filas que satisfacen el predicado *sysstart < @end AND sysend > @start*. En otras palabras, devuelve las filas con un período de validez que comienza antes de que finalice el intervalo de entrada y que termine después de que comience el intervalo de entrada. La siguiente consulta demuestra el uso de esta subcláusula:

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.Empleados
FOR SYSTEM_TIME FROM '2022-02-16 17:15:26' TO '2022-02-16 17:20:02';
```

Esta consulta genera la siguiente salida:

```
empid      nombreemp      departamento      sueldo      sysstart      sysend
-----
2          Don            HR                45000.00    2022-02-16 17:08:41 9999-12-31 23:59:59
4          Yael           Marketing        55000.00    2022-02-16 17:08:41 9999-12-31 23:59:59
1          Sara            IT                50000.00    2022-02-16 17:08:41 2022-02-16 17:20:02
5          Sven            IT                45000.00    2022-02-16 17:08:41 2022-02-16 17:20:02
3          Judy            Sales             55000.00    2022-02-16 17:08:41 2022-02-16 17:28:10
(5 rows affected)
```

Observemos que las filas con un valor de *sysstart* 02/16/2022 17:20:02 no están incluidos en la salida.



Si necesitamos que el valor de entrada @end sea inclusivo, debemos utilizar la subcláusula *BETWEEN* en lugar de *FROM*. La sintaxis de la subcláusula *BETWEEN* es *BETWEEN @start AND @end*, y devuelve las filas que satisfacen el predicado *sysstart <= @end AND sysend > @start*. Eso devuelve las filas con un período de validez que comienza *en o antes de* los extremos del intervalo de entrada y que termina *después de* que comience el intervalo de entrada. La siguiente consulta demuestra el uso de esta subcláusula con los mismos valores de entrada que en la consulta anterior:

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.Empleados
FOR SYSTEM_TIME BETWEEN '2022-02-16 17:15:26' AND '2022-02-16 17:20:02';
```

Se obtiene la siguiente salida, esta vez incluyendo filas con un valor *sysstart* de 02/16/2022 17:20:02:

empid	nombreemp	departamento	sueldo	sysstart	sysend
1	Sara	IT	52500.00	2022-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
1	Sara	IT	50000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
5	Sven	IT	45000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
3	Judy	Sales	55000.00	2022-02-16 17:08:41	2022-02-16 17:28:10
5	Sven	IT	47250.00	2022-02-16 17:20:02	2022-02-16 17:28:10

(7 rows affected)

La subcláusula *FOR SYSTEM_TIME CONTAINED IN (@Start, @end)* devuelve las filas que satisfacen el predicado *sysstart >= @start AND sysend <= @end*. Devuelve las filas con un período de validez que comience *en o después de* que comience el intervalo de entrada y que finaliza *cuando o antes que* termine el intervalo de entrada. En otras palabras, el período de validez debe estar contenido completamente en el período de entrada.

He aquí un ejemplo que demuestra el uso de esta cláusula:

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.Empleados
FOR SYSTEM_TIME CONTAINED IN ('2022-02-16 17:00:00', '2022-02-16 18:00:00');
```

Esta consulta genera la siguiente salida:

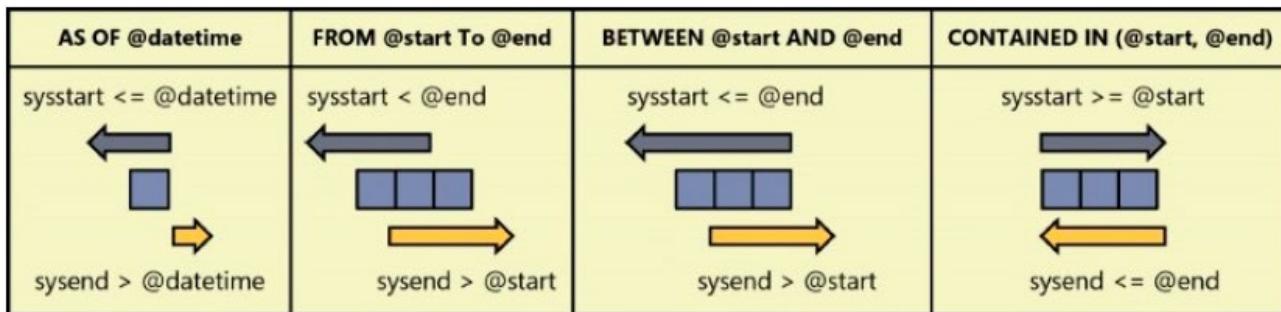
empid	nombreemp	departamento	sueldo	sysstart	sysend
6	Paul	Sales	40000.00	2022-02-16 17:08:41	2022-02-16 17:15:26
1	Sara	IT	50000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
5	Sven	IT	45000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
3	Judy	Sales	55000.00	2022-02-16 17:08:41	2022-02-16 17:28:10
5	Sven	IT	47250.00	2022-02-16 17:20:02	2022-02-16 17:28:10

(5 rows affected)

La siguiente tabla resume las subcláusulas antes mencionadas y los predicados que representan.

Subcláusula	Predicado
AS OF @datetime	sysstart <= @datetime AND sysend > @datetime
FROM @start TO @end	sysstart < @end AND sysend > @start
BETWEEN @start AND @end	sysstart <= @end AND sysend > @start
CONTAINED IN (@start, @end)	sysstart >= @start AND sysend <= @end

El siguiente gráfico tiene un resumen similar de las subcláusulas, con una representación gráfica de predicados y filas de calificación.

**Legend:**

@datetime Input date and time value

@start @end Input start and end date and time values

sysstart System period start column

sysend System period end column

T-SQL también es compatible con la subcláusula *ALL*, que simplemente devuelve todas las filas de ambas tablas. La siguiente consulta demuestra el uso de esta subcláusula:

```
SELECT empid, nombreemp, departamento, sueldo, sysstart, sysend
FROM dbo.Empleados FOR SYSTEM_TIME ALL;
```

Esta consulta genera la siguiente salida:

empid	nombreemp	departamento	suelo	sysstart	sysend
1	Sara	IT	52500.00	2022-02-16 17:20:02	9999-12-31 23:59:59
2	Don	HR	45000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
3	Judy	IT	55000.00	2022-02-16 17:28:10	9999-12-31 23:59:59
4	Yael	Marketing	55000.00	2022-02-16 17:08:41	9999-12-31 23:59:59
5	Sven	Sales	47250.00	2022-02-16 17:28:10	9999-12-31 23:59:59
6	Paul	Sales	40000.00	2022-02-16 17:08:41	2022-02-16 17:15:26
1	Sara	IT	50000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
5	Sven	IT	45000.00	2022-02-16 17:08:41	2022-02-16 17:20:02
3	Judy	Sales	55000.00	2022-02-16 17:08:41	2022-02-16 17:28:10
5	Sven	IT	47250.00	2022-02-16 17:20:02	2022-02-16 17:28:10

(10 rows affected)

Recordemos que las columnas de tiempo reflejan el período de validez de la fila como valores *datetime2* en la zona horaria UTC. Si deseamos volver a los valores como *datetimeoffset* en una determinada zona horaria, podemos utilizar la función *AT TIME ZONE*. Deberemos utilizar la función dos veces.

Una vez para convertir la entrada a *datetimeoffset*, lo que indica que está en la zona horaria UTC, y otra para convertir el valor al ejemplo zona de tiempo objetivo, *sysstart AT TIME ZONE 'UTC' AT TIME ZONE 'Argentina Standard Time'*. Si utilizamos sólo una conversión directamente objetivo, la función asumirá que el valor fuente ya está en el tiempo de destino y no realizará la comutación deseada.

Otra cosa a considerar es que para la columna de *sysend*, si el valor es el máximo en él, sólo querrá considerarlo como utilizando la zona horaria UTC. De lo contrario, convertirlo a la zona horaria de destino como con la columna de *sysstart*. Se puede utilizar una expresión *CASE* para aplicar esta lógica.

Como ejemplo, la consulta siguiente devuelve todas las filas y presenta las columnas de período en la zona horaria Hora estándar de Argentina:



```
SELECT empid, nombreemp, departamento, sueldo,
sysstart AT TIME ZONE 'UTC' AT TIME ZONE 'Argentina Standard Time' AS sysstart,
CASE
WHEN sysend = '9999-12-31 23:59:59'
THEN sysend AT TIME ZONE 'UTC'
ELSE sysend AT TIME ZONE 'UTC' AT TIME ZONE 'Argentina Standard Time'
END AS sysend
FROM dbo.Empleados FOR SYSTEM_TIME ALL;
```

Esta consulta genera la siguiente salida:

empid	nombreemp	departamento	suelo	sysstart	sysend
1	Sara	IT	52500.00	2022-02-16 14:20:02 -03:00	9999-12-31 23:59:59 +00:00
2	Don	HR	45000.00	2022-02-16 14:08:41 -03:00	9999-12-31 23:59:59 +00:00
3	Judy	IT	55000.00	2022-02-16 14:28:10 -03:00	9999-12-31 23:59:59 +00:00
4	Yael	Marketing	55000.00	2022-02-16 14:08:41 -03:00	9999-12-31 23:59:59 +00:00
5	Sven	Sales	47250.00	2022-02-16 14:28:10 -03:00	9999-12-31 23:59:59 +00:00
6	Paul	Sales	40000.00	2022-02-16 14:08:41 -03:00	2022-02-16 14:15:26 -03:00
1	Sara	IT	50000.00	2022-02-16 14:08:41 -03:00	2022-02-16 14:20:02 -03:00
5	Sven	IT	45000.00	2022-02-16 14:08:41 -03:00	2022-02-16 14:20:02 -03:00
3	Judy	Sales	55000.00	2022-02-16 14:08:41 -03:00	2022-02-16 14:28:10 -03:00
5	Sven	IT	47250.00	2022-02-16 14:20:02 -03:00	2022-02-16 14:28:10 -03:00

(10 rows affected)

Cuando haya terminado de experimentar con los datos, ejecute el siguiente código para la limpieza:

```
IF OBJECT_ID(N'dbo.Empleados', N'U') IS NOT NULL
BEGIN
IF OBJECTPROPERTY(OBJECT_ID(N'dbo.Empleados', N'U'), N'TableTemporalType') = 2
ALTER TABLE dbo.Empleados SET ( SYSTEM_VERSIONING = OFF );
DROP TABLE IF EXISTS dbo.EmpleadosHistoria, dbo.Empleados;
END;
```




CONTENIDO

Programa de Estudio 2024.....	1
SQL.....	5
Historia de Microsoft SQL Server.....	9
SQL Server 2005.....	10
SQL Server 2008.....	10
SQL Server 2008 R2.....	10
SQL Server 2012.....	10
SQL Server 2014.....	10
SQL Server 2016.....	10
SQL Server 2017.....	10
SQL Server 2019.....	11
SQL Server 2022.....	11
Ediciones Principales.....	11
Funciones del servidor	11
Tipos de Sistemas de Bases de Datos	12
Arquitectura de SQL Server.....	12
Alternativas ABC	12
Instancias de SQL Server	15
Bases de Datos	15
Esquemas y objetos	17
Categorías de las instrucciones SQL.....	17
Restricciones (Constraints)	17
Creando Tablas	19
Claves foráneas	20
Restricción CHECK	20
Restricción DEFAULT	21
Null y la lógica de tres valores.....	21
Orden Lógico de Ejecución	21
Base TSQLV6ES	22
DER.....	22
Diccionario de Datos.....	22
Diagrama de Base de Datos.....	24
Filtros Top y OFFSET-FETCH.....	25
Filtro TOP	25
Tiebreaker	26



Empates (TIES).....	27
Filtro OFFSET-FETCH	28
Predicados y Operadores.....	29
Precedencia de Tipos de Datos.....	30
Precedencia de operadores.....	31
Expresiones CASE.....	31
NULLs	32
Funciones GREATEST y LEAST	34
Operaciones Simultáneas	34
Trabajando con cadenas de caracteres.....	35
Tipos de Datos	35
Cotejo	36
Concatenación de cadenas (operador + FUNCIÓN CONCAT y Función CONCAT_WS)	37
Funciones de Cadena.....	39
Predicado LIKE.....	48
Trabajando con datos de fecha y hora.....	49
Tipos de datos de Fecha y Hora	49
Literales	49
Trabajando con fecha y hora por separado	52
Filtrando rangos de fechas	53
Funciones de Fecha y Hora	54
Obtener la fecha y hora actual	54
Funciones CAST, CONVERT y PARSE y sus contrapartes TRY_	55
Función DATEPART	55
Funciones YEAR, MONTH y DAY	56
Función DATENAME	56
Funciones FROMPARTS.....	57
Función SWITCHOFFSET	57
Función TODATETIMEOFFSET	57
Función AT TIME ZONE	58
Función DATEADD	59
Funciones DATEDIFF y DATEDIFF_BIG.....	59
Función DATE_BUCKET	60
Función ISDATE	60
Función EOMONTH	61
Función DATETRUNC	61
Función GENERATE_SERIES	61



Anexo Estilos de Fecha y Hora	62
Programando	65
Variables.....	65
Lotes (Batches).....	67
Un lote como unidad de análisis	67
Lotes y Variables.....	68
Sentencias que no se pueden combinar en el mismo lote	68
Un lote como unidad de resolución.....	69
La opción GO n	69
Elementos de Flujo	70
El elemento de Flujo IF...ELSE.....	70
WHILE	71
Transacciones y Concurrencia	73
Transacciones	73
Trabas y bloqueo.....	76
Trabas (Locks)	76
Interbloqueos (DeadLocks).....	97
Conclusión.....	100
Cursos	101
Tablas Temporales.....	104
Tablas Temporales Locales	104
Tablas Temporales Globales.....	105
Variables de Tabla.....	106
Tipos de Tabla.....	107
SQL Dinámico	109
El comando EXEC	109
EL Stored Procedure sp_executesql.....	109
Usando PIVOT con SQL Dinámico	110
Rutinas	112
Funciones Definidas por el Usuario.....	112
Stored procedures	113
Disparadores (Triggers)	115
Manejo de Errores.....	119
Funciones de Sistema.....	129
Funciones heredadas (legacy) del "sistema" (antes conocidas como variables globales)	129
@@CONNECTIONS	129
@@CPU_BUSY	130



@@IDLE	130
@@IO_BUSY.....	130
@@PACK_RECEIVED and @@PACK_SENT	130
@@PACKET_ERRORS.....	131
@@TIMETICKS.....	131
@@TOTAL_ERRORS.....	131
@@TOTAL_READ and @@TOTAL_WRITE	131
@@TRANCOUNT	131
Funciones de Configuración	132
@@DATEFIRST	132
@@DBTS.....	132
@@LANGID and @@LANGUAGE	132
@@LOCK_TIMEOUT.....	132
@@MAX_CONNECTIONS.....	133
@@MAX_PRECISION	133
@@NESTLEVEL.....	133
@@OPTIONS	133
@@REMSERVER	134
@@SERVNAME.....	134
@@SERVICENAME	134
@@SPID.....	134
@@TEXTSIZE	135
@@VERSION	135
Funciones de Cursor	135
@@CURSOR_ROWS	135
@@FETCH_STATUS	136
CURSOR_STATUS	136
Funciones Básicas de Metadatos.....	137
COL_LENGTH	137
COL_NAME	137
COLUMNPROPERTY.....	138
DATABASEPROPERTY.....	138
DATABASEPROPERTYEX.....	139
DB_ID	139
DB_NAME.....	140
FILE_ID	140
FILE_NAME.....	140



FILEGROUP_ID	140
FILEGROUP_NAME.....	140
FILEGROUOPROPERTY	140
FILEPROPERTY.....	141
FULLTEXTCATALOGPROPERTY	141
FULLTEXTSERVICEPROPERTY.....	141
INDEX_COL	142
INDEXKEY_PROPERTY	142
INDEXPROPERTY	142
OBJECT_ID	143
OBJECT_NAME.....	143
OBJECTPROPERTY	143
OBJECTPROPERTYEX	145
@@PROCID.....	145
SCHEMA_ID	145
SCHEMA_NAME.....	145
SQL_VARIANT_PROPERTY	145
TYPEPROPERTY	146
Funciones de Conjunto de Filas (Rowset).....	146
CHANGETABLE	147
CONTAINSTABLE	147
FREETEXTTABLE.....	147
OPENDATASOURCE.....	147
OPENQUERY	147
OPENROWSET.....	147
OPENXML.....	148
Funciones de Seguridad.....	148
HAS_DBACCESS	148
IS_MEMBER	149
IS_SRVROLEMEMBER.....	149
SUSER_ID	149
SUSER_NAME	149
SUSER_SID	150
SUSER_SNAME	150
USER.....	150
USER_ID	150
USER_NAME	150



Funciones del Sistema	150
APP_NAME	151
CASE.....	151
CAST and CONVERT	152
COALESCE	152
COLLATIONPROPERTY	152
CURRENT_USER	152
DATALENGTH	152
@@ERROR	153
FORMATMESSAGE	153
GETANSINULL	153
HOST_ID.....	153
HOST_NAME.....	153
IDENT_CURRENT	154
IDENT_INCR	154
IDENT_SEED	154
@@IDENTITY	154
IDENTITY	154
ISNULL	155
ISNUMERIC	155
NEWID	155
NULLIF	155
PARSENAME	155
PERMISSIONS.....	155
@@ROWCOUNT.....	155
ROWCOUNT_BIG	156
SCOPE_IDENTITY.....	156
SERVERPROPERTY.....	156
SESSION_USER	158
SESSIONPROPERTY	158
STATS_DATE	158
SYSTEM_USER	158
USER_NAME	158
Consultas Multi-tabla	159
Subconsultas	159
Subconsultas Autocontenidoas.....	159
Subconsultas Autocontenidoas Escalares	159



Subconsultas Autocontenidas Multivaluadas	160
Subconsultas Correlacionadas.....	161
EXISTS.....	162
Subconsultas con mal comportamiento	162
Expresiones de Tabla	165
Tablas Derivadas	165
Expresiones Comunes de Tabla (CTE de Common Table Expressions)	168
Vistas	170
Funciones con Valores de tabla en línea.....	171
Ejemplos	172
JOINS.....	173
CROSS JOIN	173
INNER JOIN.....	177
OUTER JOIN.....	178
SELF JOIN	179
EQUI Y NONEQUI JOINS	179
CONSULTAS MULTI-JOIN.....	180
Controlar el orden de evaluación físico del join	180
Controlar el orden de evaluación lógica del join	182
SEMI y ANTI SEMI JOINS.....	185
ALGORITMOS DE JOIN.....	187
Operadores UNION, EXCEPT e INTERSECT.....	194
El operador UNION	194
El operador UNION ALL	194
El operador UNION (DISTINCT)	195
El operador INTERSECT	196
El operador INTERSECT (DISTINCT)	196
El operador EXCEPT	196
El operador EXCEPT (DISTINCT)	196
Precedencia	197
EJEMPLOS	197
Funciones de Ventana	201
Funciones Aggregate de Ventana	201
Funciones Ranking de Ventana	204
Funciones Offset de Ventana.....	206
SELECT – WINDOW.....	208
Pivoteando Datos	209



Pivoteando con una consulta agrupada.....	210
Pivoteando con el operador PIVOT	211
Despivoteando Datos.....	212
Despivoteando con el operador APPLY.....	213
Despivoteando con el operador UNPIVOT	215
Conjuntos de Agrupación	216
La subcláusula GROUPING SETS	217
La subcláusula CUBE.....	218
La subcláusula ROLLUP	218
Las funciones GROUPING y GROUPING_ID	219
Seguridad de la Información.....	223
Asegurar la Red.....	223
Seguridad Física	223
Cifrado de Bases de Datos	223
AES	224
Hashing.....	224
Seguridad en las contraseñas	225
Passwords Robustas.....	226
Asegurando la Instancia.....	229
Ataques SQL Injection.....	229
¿Por qué son exitosos los ataques SQL Injection?.....	230
Criptología	231
Historia	231
Criptografía	232
Criptoanálisis	232
Criptosistema.....	232
Algoritmos Simétricos Modernos (Llave Privada).....	234
Criptoanálisis de Algoritmos Simétricos	237
Algoritmos Asimétricos (Llave Privada-Pública)	237
Autentificación	239
PGP (Pretty Good Privacy).....	240
Esteganografía	241
Bases de datos NoSQL o No Relacionales.....	243
Big Data	243
Bases de Datos NoSQL	245
Basadas en modelos de Grafo	245
Lenguaje de consulta de grafos Cypher	247



Sistemas de Gestión de Bases de Datos NoSQL	249
Cypher	251
Northwind en Neo4j	257
Acerca del dominio de datos.....	257
Desarrollo de un modelo de grafos.....	258
¿En qué se diferencia el modelo grafo del modelo relacional?	259
Exportación de tablas relacionales a CSV.....	259
Importar los datos usando Cypher	262
Creación de índices y restricciones para los datos en el grafo.....	268
Creando las relaciones entre los nodos.....	268
Consultando el grafo.....	273
Base de datos documentales	276
Documentos.....	276
Tipos de datos	276
Colecciones	278
Bases de datos	279
La Shell de comandos de MongoDB	280
Operaciones básicas en la Shell	282
Tablas Cronológicas	295
Creación de tablas.....	295
Modificación de datos.....	299
Consulta de datos.....	302