**Define en el documento que significado tienen las terminaciones ci, ai y que otras opciones existen:**

CI: Case insensitive.

AI: Accent insensitive.

CS: Case sensitive.

AS : Accent sensitive

[Collation and Unicode support - SQL Server | Microsoft Learn](https://learn.microsoft.com/en-us/sql/relational-databases/collations/collation-and-unicode-support?view=sql-server-ver16)

**Para que se usa cada uno de los tipos, su tamaño en bytes, las diferencias entre campos con un tipo similar y describe cuando es mejor usar uno u otro, compara los tipos numéricos, string, datetimes, unique identifier, etc.**

| **Tipo de Dato** | **Descripción** | **Cuándo Usarlo** | **Pros** | **Contras** |
| --- | --- | --- | --- | --- |
| **BIT** | Un solo bit que almacena valores 0 o 1. | Cuando se necesita almacenar valores lógicos. | Pequeño almacenamiento, eficiente en espacio. | Limitado a dos valores (0 y 1). |
| **TINYINT** | Entero de 1 byte sin signo que varía de 0 a 255. | Cuando se necesita almacenar números pequeños. | Uso eficiente de almacenamiento para valores pequeños. | Rango limitado de valores. |
| **SMALLINT** | Entero de 2 bytes que varía de -32,768 a 32,767. | Cuando se necesita almacenar números enteros pequeños. | Almacenamiento eficiente para enteros pequeños. | Rango limitado de valores. |
| **INT** | Entero de 4 bytes que varía de -2,147,483,648 a 2,147,483,647. | Para almacenar enteros medianos. | Amplio rango de valores enteros. | Puede ser excesivo para valores pequeños. |
| **BIGINT** | Entero de 8 bytes que varía de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807. | Cuando se necesita almacenar números grandes. | Rango masivo de valores enteros. | Utiliza más espacio de almacenamiento. |
| **DECIMAL(p,s)** | Número decimal fijo de precisión **p** y escala **s**. | Cuando se necesita precisión decimal exacta. | Precisión decimal exacta. | Mayor uso de almacenamiento. |
| **NUMERIC(p,s)** | Sinónimo de **DECIMAL**. | Cuando se necesita precisión decimal exacta. | Precisión decimal exacta. | Mayor uso de almacenamiento. |
| **FLOAT(n)** | Número de coma flotante de precisión simple. | Para almacenar números reales aproximados. | Amplio rango de valores reales. | Pérdida de precisión en cálculos complejos. |
| **REAL** | Número de coma flotante de precisión simple. | Para almacenar números reales aproximados. | Buena precisión para cálculos científicos. | Menos precisión que **FLOAT**. |
| **DATETIME** | Fecha y hora desde el 1 de enero de 1753 hasta el 31 de diciembre de 9999, con una precisión de 3.33 milisegundos. | Para almacenar fechas y horas. | Amplio rango de fechas y horas. | Precisión limitada. |
| **DATE** | Fecha sin hora, desde el 1 de enero de 0001 hasta el 31 de diciembre de 9999. | Para almacenar solo fechas sin la parte de tiempo. | Uso eficiente de almacenamiento. | No almacena la hora del día. |
| **TIME** | Hora del día desde las 00:00:00.0000000 hasta las 23:59:59.9999999, con una precisión de 100 nanosegundos. | Para almacenar solo la parte de tiempo del día. | Alta precisión en la parte de tiempo. | No almacena la fecha. |
| **CHAR(n)** | Cadena de longitud fija con un máximo de 8,000 caracteres. | Cuando se necesita una cadena de longitud fija. | Búsqueda más rápida en comparación con **VARCHAR**. | Usa espacio de almacenamiento incluso para valores más cortos. |
| **VARCHAR(n)** | Cadena de longitud variable con un máximo de 8,000 caracteres. | Cuando se necesita una cadena de longitud variable. | Usa solo el espacio necesario para los datos. | Búsqueda más lenta en comparación con **CHAR**. |
| **TEXT** | Cadena de longitud variable con un máximo de 2^31-1 (2,147,483,647) caracteres. | Para almacenar texto largo. | Manejo de grandes volúmenes de texto. | No se pueden realizar operaciones de búsqueda eficientes. |
| **BINARY(n)** | Datos binarios de longitud fija con un máximo de 8,000 bytes. | Cuando se necesita una longitud fija de datos binarios. | Búsqueda más rápida en comparación con **VARBINARY**. | Usa espacio de almacenamiento incluso para valores más cortos. |
| **VARBINARY(n)** | Datos binarios de longitud variable con un máximo de 8,000 bytes. | Cuando se necesita una longitud variable de datos binarios. | Usa solo el espacio necesario para los datos. | Búsqueda más lenta en comparación con **BINARY**. |
| **IMAGE** | Datos binarios de longitud variable con un máximo de 2^31-1 (2,147,483,647) bytes. | Para almacenar imágenes u otros datos binarios grandes. | Capacidad para almacenar grandes volúmenes de datos binarios. | Tipo de dato obsoleto, se recomienda usar **VARBINARY(MAX)**. |
| **UNIQUEIDENTIFIER** | Identificador único global (GUID). | Cuando se necesita un identificador único global. | Unicidad garantizada en todo el sistema. | Uso ineficiente de almacenamiento y no secuencial. |
| **XML** | Datos XML de hasta 2GB en tamaño. | Cuando se necesitan almacenar datos XML estructurados. | Capacidad para almacenar datos complejos. | Puede ser difícil de consultar y manipular. |
| **JSON** | Datos JSON introducidos en SQL Server 2016 (compatibilidad nivel 130) y versiones posteriores. | Cuando se necesitan almacenar datos en formato JSON. | Intercambio de datos flexible y fácil integración con aplicaciones web. | Puede tener un mayor consumo de almacenamiento en comparación con otros formatos de datos. |

<https://learn.microsoft.com/es-es/sql/t-sql/data-types/data-types-transact-sql?view=sql-server-ver16>

**Ejemplos char vs nchar, char vs varchar, varchar vs nvarchar:**

Char es una cadena caracteres – Nchar es similar pero para datos Unicode.

Si es Var, es de longitud variable.

**datetime vs smalldatetime:**

La diferencia es la precisión, 3,33 MS vs 1 minuto. Esto acarrea que ocupe mas almacenamiento 8 bytes vs 4 bytes. Y que tenga más rango de fechas, desde el 1 de enero de 1753 hasta 31 de diciembre del 9999 contra el 1 de enero de 1900 hasta el 6 de junio de 2079.

**Datetimeoffset:**

Fecha + desplazamiento(offset). 10 bytes para la fecha mas 5 adicionales para el offset.

**int vs bigint**

Tamaño y rango, 4 bytes vs 8, -2,147,483,648 hasta 2,147,483,647, mientras que **BIGINT** tiene un rango de valores mucho más amplio, desde -9,223,372,036,854,775,808

**Decimal vs money vs smallmoney,**

decimal es exacto.

Money precisión fija de 4 digitos decimales

Smallmoney más pequeño

**Describe las ventajas y desventajas de usar un int autonumérico vs unique identifer.**

Int autonumérico la única ventaja que tiene el espacio que ocupa, 4 bytes y que sigue un patrón secuencial.

El GUID ocupa 16 bytes de almacenamiento, pero es un Id único global, no sigue patrón secuencial y es prácticamente imposible la colisión incluso con GUIDs de otras bases de datos.  
Por ejemplo, si consideramos el caso de generar 10^18 (1 trillón) de GUIDs aleatorios, la probabilidad de que dos GUIDs colisionen sería aproximadamente de 1 en 2^68, lo cual es extremadamente improbable.

**Describe cuando es adecuado usar default values en el diseño de una tabla, escribe distintos ejemplos de default values usando distintos tipos de campos.**

Muy útil para el bit(booleano) para marcar por defecto el 1(true).

En tablas de auditoría para marcar la fecha de inserción de la fila donde será la fecha y hora del momento de la inserción.

**Analiza que columnas deben usar nulos y cuales no, define ventajas y desventajas usando columnas con o sin nulos.**

**Sin nulos – Ventajas** : Garantiza el valor válido de la columna, simplifica queries. Columnas NULL pueden requerir espacio adicional por tanto pueden ocupar menos las columnas NOT NULL.

**Sin nulos – Desventajas** : Obliga a la entrada del dato en la columna y quita flexibilidad, pero para que se necesita flexibilidad a coste de romper la integridad del dato. ¿Qué sentido tiene en este caso dejar nulo el Código del País, si es obligatorio y rompe tu modelo de datos y de dominio?.

**Con nulos – Ventajas** : Permite dejarlo nulo si el valor es desconocido en el momento de la inserción, en este caso si tiene sentido. Si es con motivo de simplificar proceso de inserción para evitar añadir columnas al insert, ERROR, estamos insertando una fila con columnas con valores nulos, provocando inconsistencia de datos, aún sabiéndolos o siendo requeridos por el dominio, por vagancia o por qué al usuario le parecen muchos a la hora de crear el registro, esto a mi me parece un error de diseño.

**Con nulos – Desventajas** : Consultas deben contemplar nulos, inconsistencia y falta de integridad en los datos. Otro ejemplo, ¿que sentido tiene crear un Producto sin precio unitario?

**Crea un diagrama de base de datos y verifica las reglas de integridad referencial, ten en cuenta que no podrás eliminar un cliente que tenga facturas o un artículo que tenga líneas de detalle:**

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated

¿Actualizar Management Studio? Con azure data studio no encuentro donde generar el diagrama.

**Los campos clave pueden estar compuestos de varios campos, describe cuando usarías esta estrategia y si te parece una buena práctica.**

Si el negocio lo requiere, reduce duplicidad de datos y puede agilizar consultas.

Si la relación entre las entidades no se puede representar con una clave primaria.

Ejemplo Pedido – Productos donde 1 mismo pedido contiene varios productos y cada producto puede estar en N pedidos. KEY(Id Pedido, Id Producto).

Por el contrario ocupan mas espacio de almacenamiento, aumenta la complejidad a la hora de formular las consultas.

**Usando triggers o campos calculados haz que los campos Total y TotalLine de las tablas se actualice cada vez que un campo relacionado con estos cálculos se actualice. Razona y describe en el document que ventajas y desventajas representa el uso de campos calculados o triggers.**

**Campos calculados**: El valor se calcula a partir de otros, siempre se mantiene actualizado, simple y muy buen rendimiento pero tenemos una limitación de las operaciones de cálculo. Por ejemplo en Fundraising, la Donación tiene importe total y un importe neto que es calculado, en base a las filas de una entidade relacionada, que contienen el % de deducción y que esos % se aplica para calcular el importe neto. En este caso no nos serviría.

**Triggers**: Se puede implementar mayor lógica y mas flexible, por el contrario es mas complejo el mantenimiento especialmente si la base de datos contiene múltiples triggers.

Además el principal problema de los triggers es que a cada cambio en la tabla se ejecuta y si es una tabla con gran volumen de operaciones va a causar un retardo y perdida de rendimiento en las inserciones y las actualizaciones.

En algunos casos puede ser buena opción no tener el dato en base de datos, es redundante pues se puede calcular a partir de otros y puede ser el propio código fuente quien lo haga cuando lo necesite.