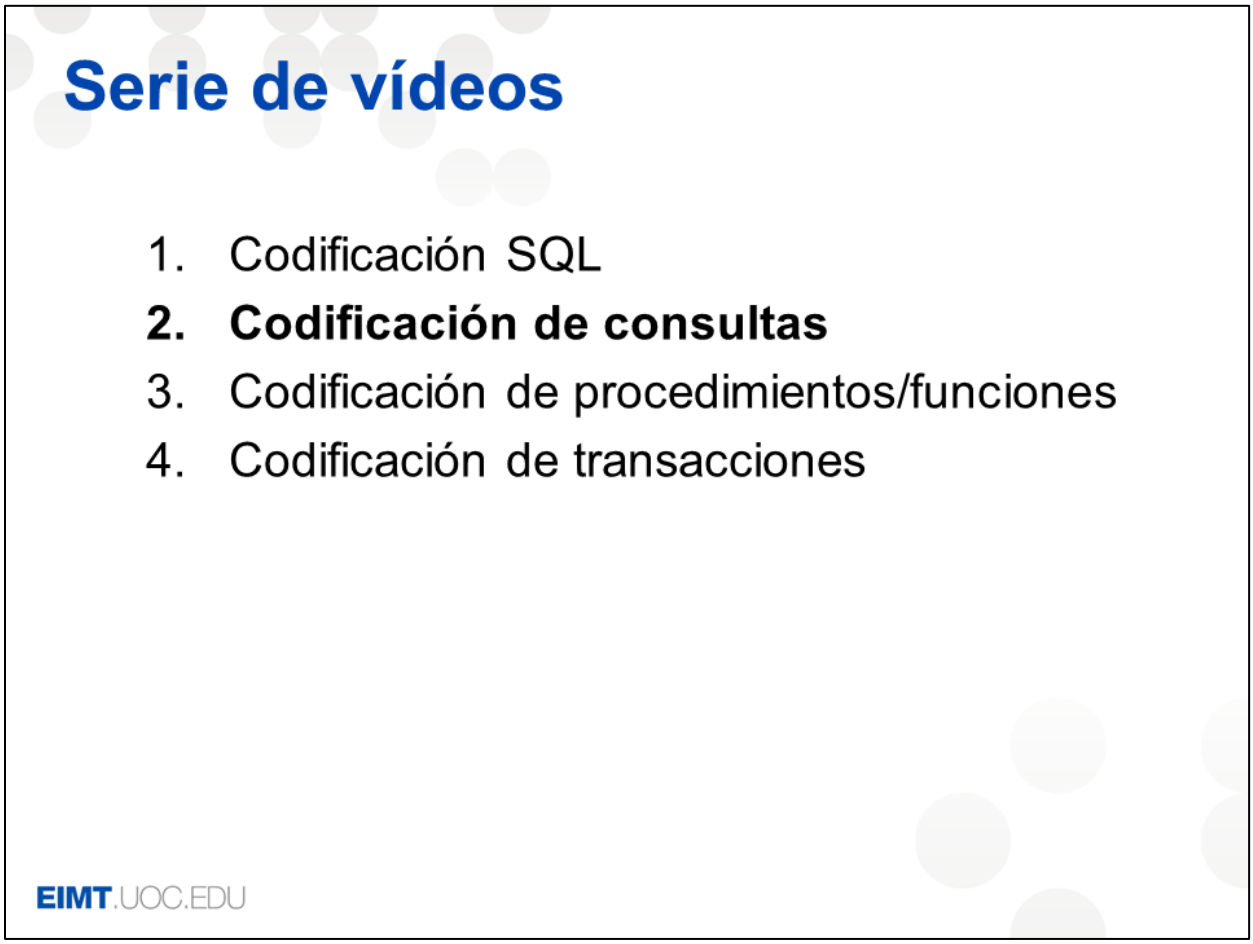
Bienvenidos al segundo vídeo de la serie *Buenas prácticas en SQL* de la asignatura de *Bases de datos para Data Warehouse*, serie en la que veremos un conjunto de buenas prácticas a la hora de programar SQL en entornos de bases de datos/*Data Warehouse*.

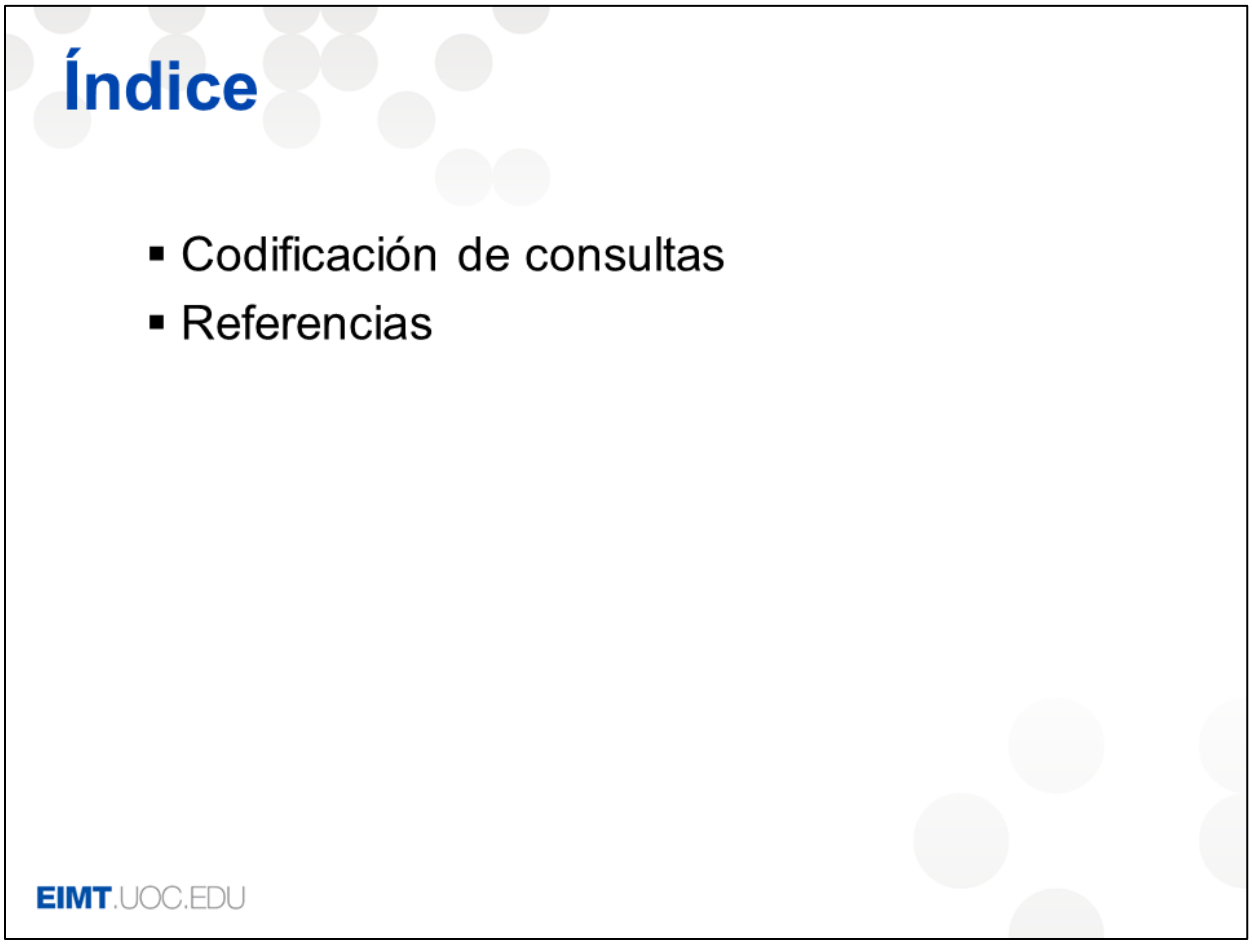


Esta serie de vídeos está dividida en 4 partes:

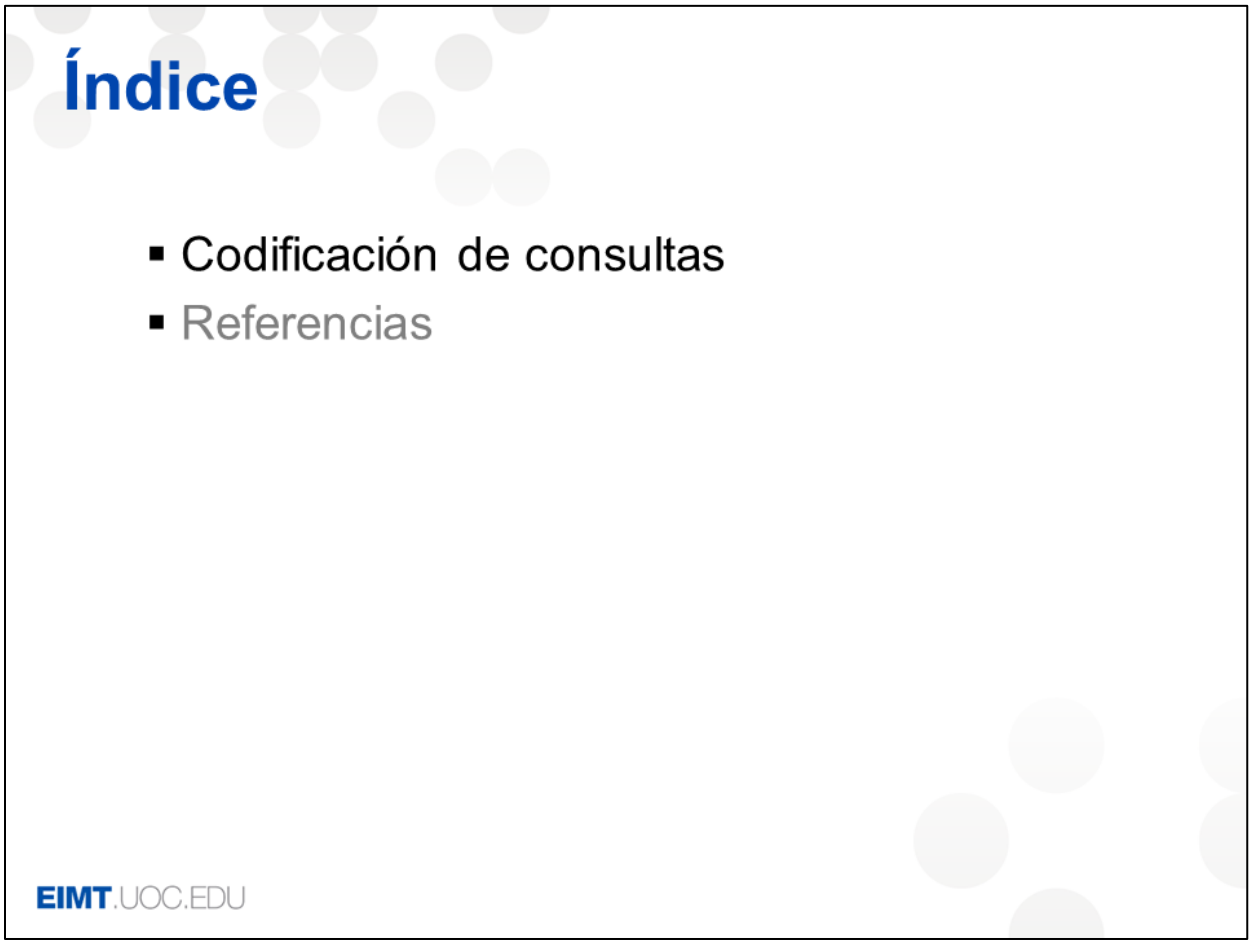
- *Codificación SQL*, centrada en aquellas prácticas desde un punto de vista genérico en SQL, que afectan a la legibilidad y portabilidad de código SQL.
- *Codificación de consultas*, en la que veremos prácticas que nos ayudarán a generar consultas SQL más eficientes y legibles.
- *Codificación de procedimientos y funciones*, donde pondremos atención en aquellas prácticas que nos permitirán escribir código SQL en servidor más eficiente y gestionar errores de forma controlada.
- *Codificación de transacciones*, en la que detallaremos prácticas para asegurar un correcto control de las transacciones que codificamos.

Es importante resaltar que, dentro de cada categoría, no se proporcionan todas las posibles buenas prácticas del mercado, sino que se trata de un conjunto concreto que desde la UOC hemos considerado como más relevantes.

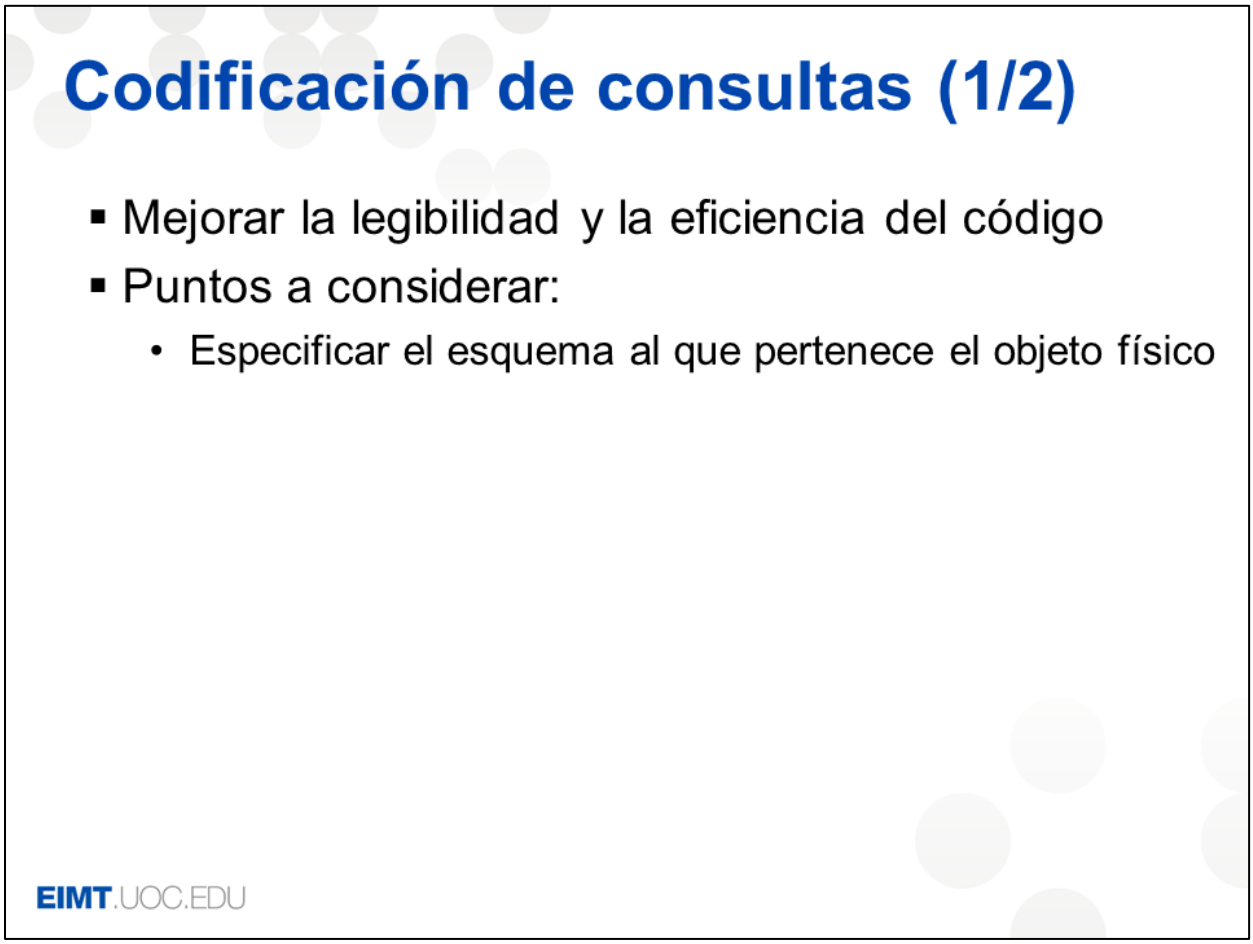
Este vídeo afrontará la segunda parte, *Codificación de consultas*.



Este segundo vídeo se centrará en presentar una serie de buenas prácticas que nos permitirán generar consultas SQL eficientes, legibles y portables, proponiendo en algunos casos ejemplos para facilitar su entendimiento. Al final del vídeo, se proporcionarán las referencias bibliográficas utilizadas.

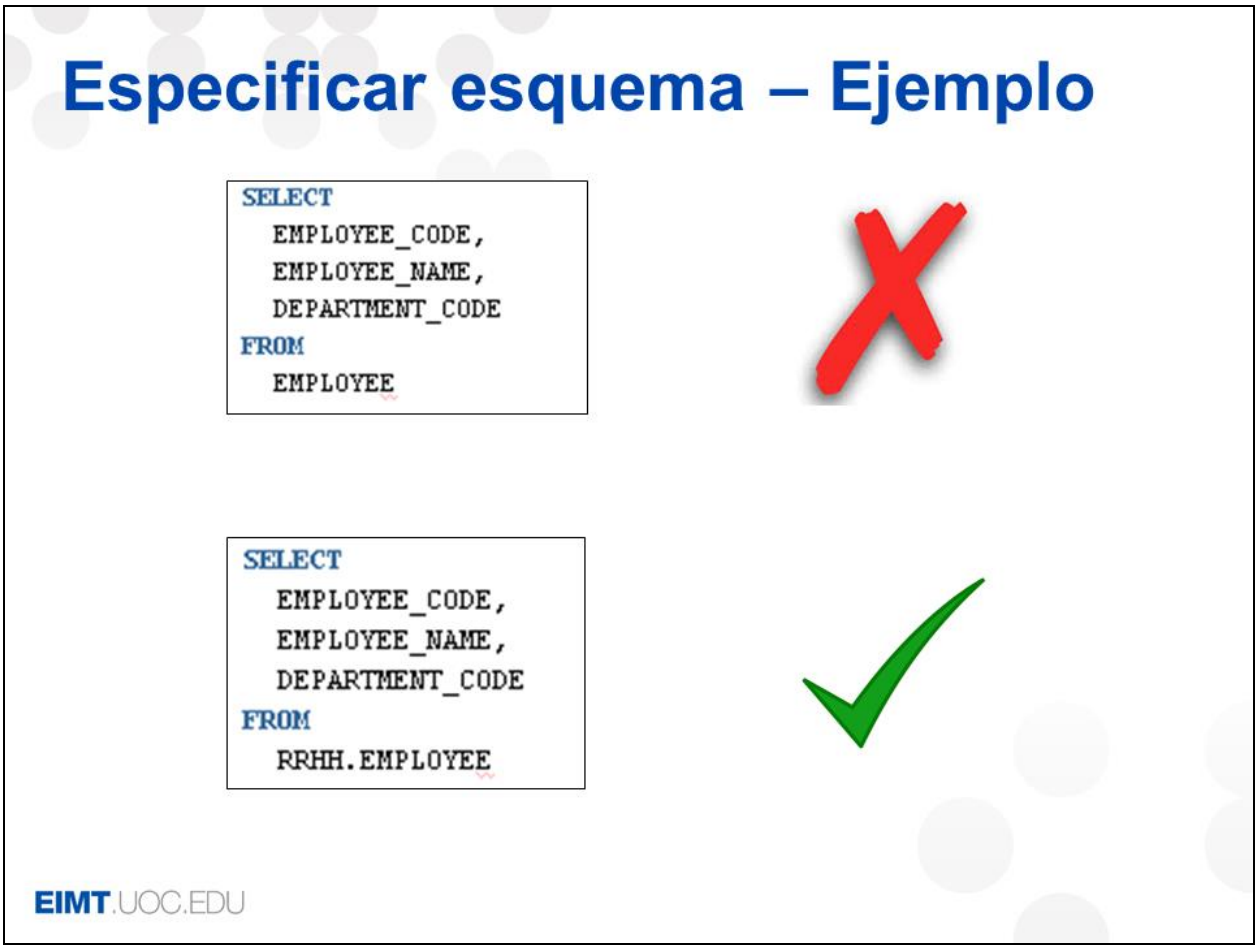


Vamos pues a ver la segunda categoría de buenas prácticas: codificación de consultas.

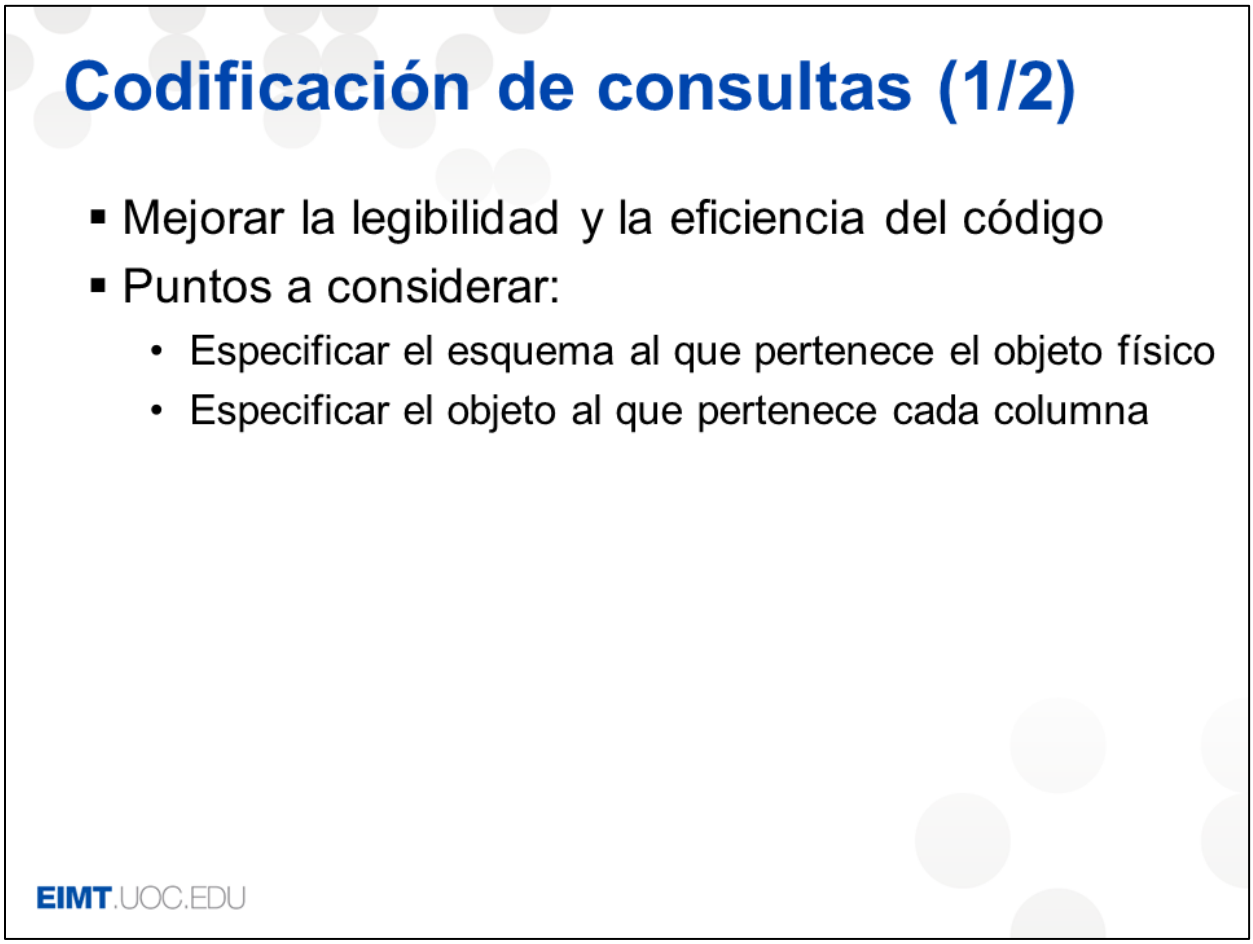


Esta sección presenta una serie de guías o prácticas que nos permitirán codificar consultas SQL más eficientes a la vez que nos permitan una mejor legibilidad de éstas. Desde este punto de vista, se recomiendan los siguientes puntos:

- *Especificar el esquema al que pertenece el objeto físico*: cuando realizamos una consulta SQL, generalmente utilizamos tablas, vistas o vistas materializadas (entre otros), los cuales son objetos que pertenecen a un esquema en concreto. A la hora de generar una consulta, se considera como buena práctica especificar el esquema al que pertenece dicho objeto. Veamos un ejemplo de este punto.

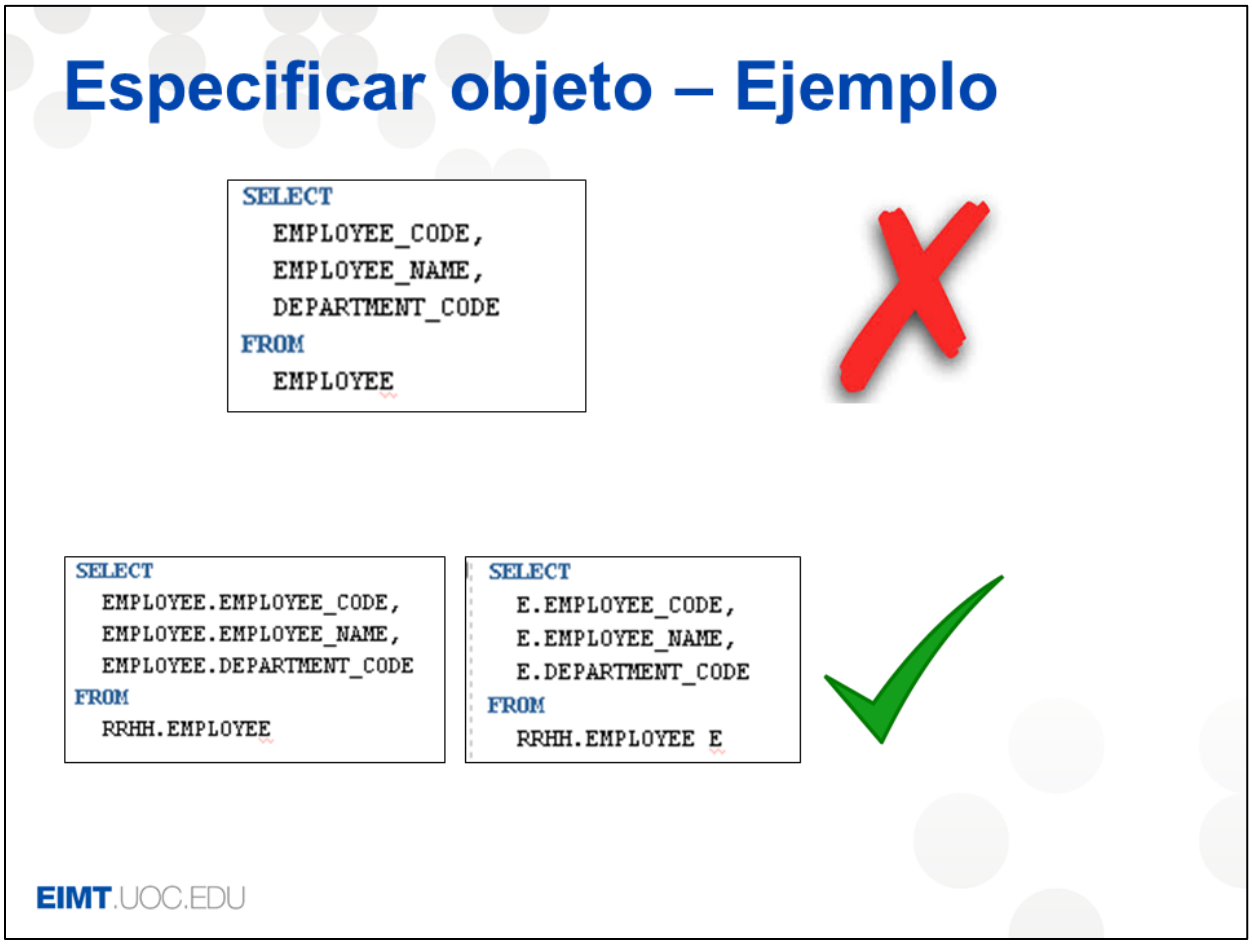


Supongamos que tenemos una tabla *employee* perteneciente al esquema *rrhh*. Una consulta que incluya una llamada a la tabla *employee* debería de realizarse mediante *rrhh.employee*. Esto es así para evitar problemas en el caso de que un usuario tenga acceso a dos tablas que existen con el mismo nombre pero en diferentes esquemas (por ejemplo, si existe una tabla *employee* en el esquema *taxes,* es decir, *taxes.employee*).

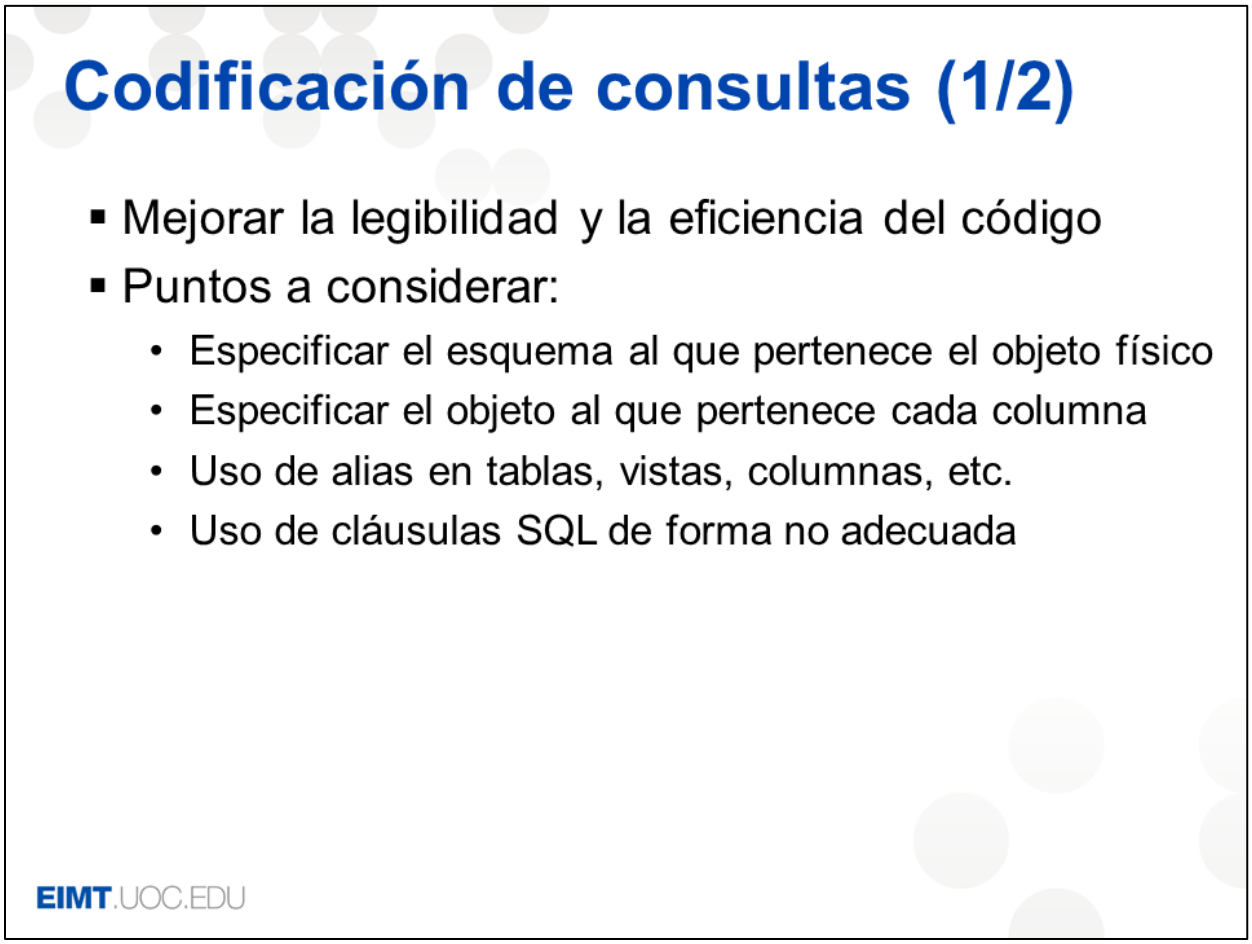


El siguiente punto es:

- *Especificar el objeto al que pertenece cada columna:* a la hora de especificar columnas en una consulta SQL (como parte de las cláusulas `SELECT`, `WHERE`, `GROUP BY`, `HAVING` u `ORDER BY`), se recomienda especificar a qué objeto pertenece dicha columna. Veamos un ejemplo de este punto.



Utilizando la misma tabla *employee* vista anteriormente, la proyección de las columnas código de empleado, nombre de empleado y código de departamento debería de especificarse con el nombre de la tabla o bien con un alias (en caso de que, por ejemplo, se utilice la tabla *employee* más de una vez en la misma consulta), tal y como se muestra en las imágenes.

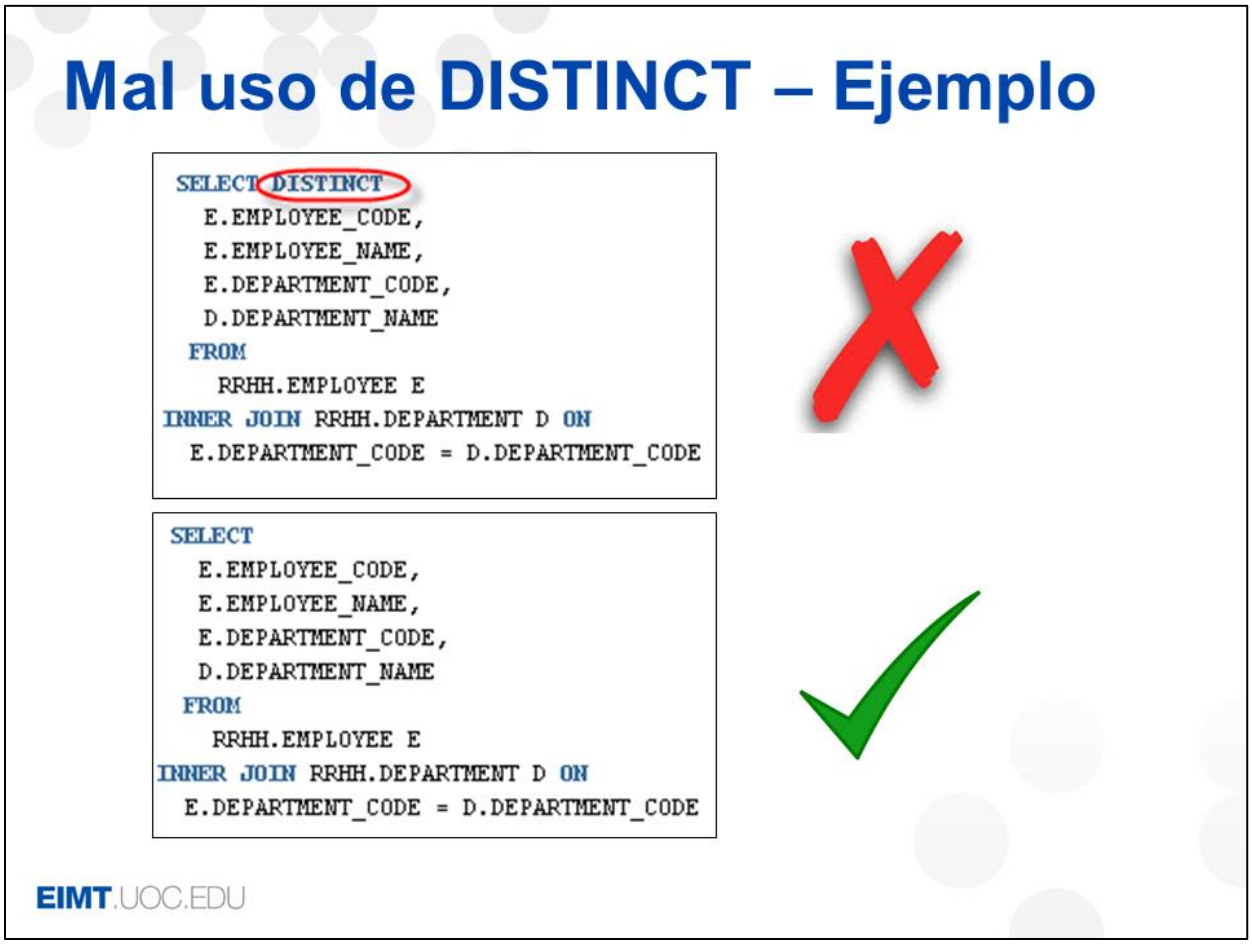


Continuamos con los siguientes puntos:

- *Uso de alias en tablas, vistas, columnas*: en consultas SQL, podemos especificar alias tanto en las tablas, vistas o subconsultas, como en las columnas que se proyectan. Se recomienda el uso de estos aliases para facilitar la legibilidad del código y también evitar problemas de funcionamiento, en caso de que una columna se llame igual en más de una tabla, vista o subconsulta.

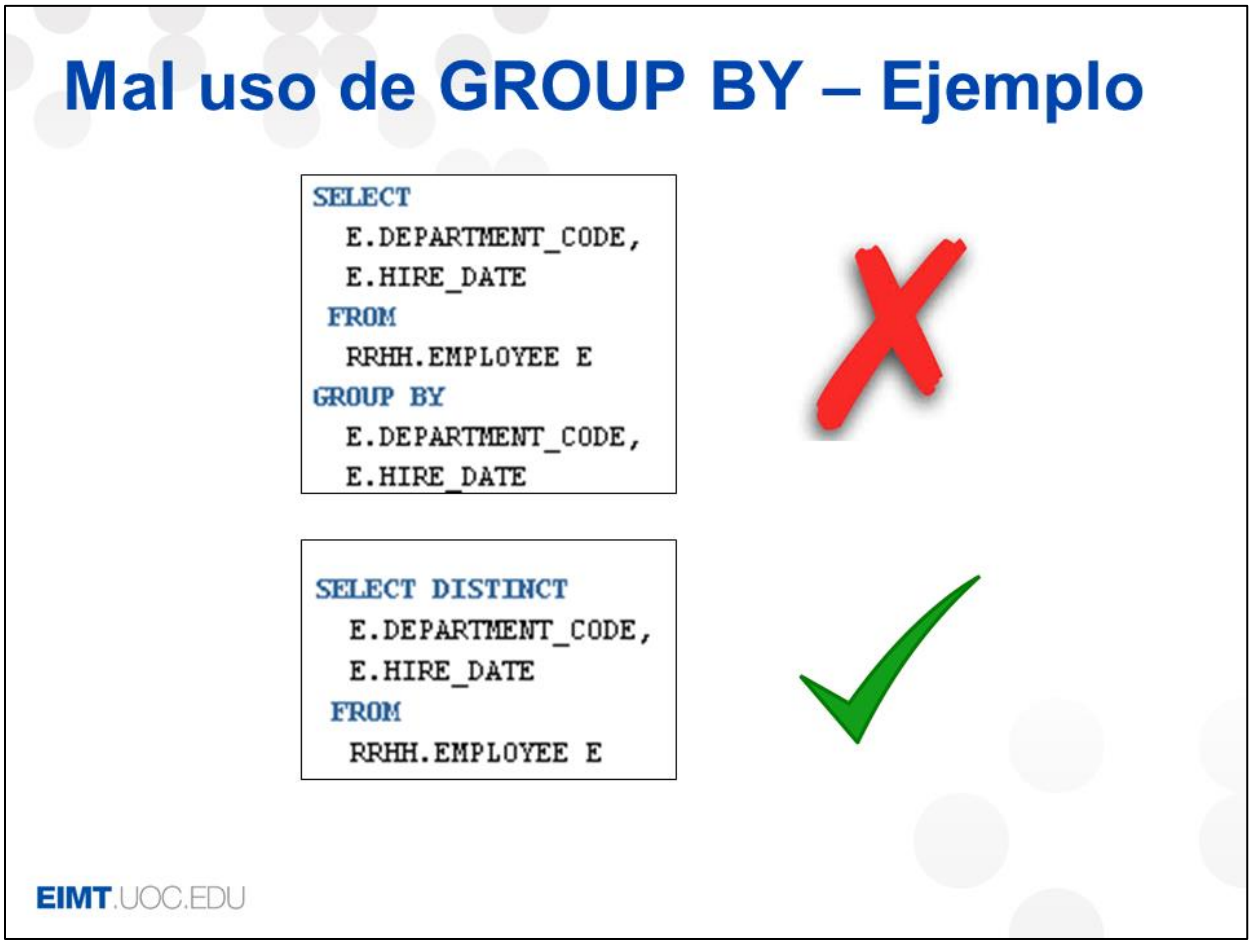
- *Uso de cláusulas SQL de forma no adecuada*: es muy importante que las cláusulas SQL sean utilizadas para el propósito que se les ha dado, ya que podrían causar los siguientes problemas:
    - Rendimiento: la consulta realiza de forma innecesaria operaciones, exigiendo al SGBD un trabajo extra realizando tareas que son innecesarias.
    - Ofuscación de código: aunque es posible obtener el mismo resultado utilizando cláusulas SQL diferentes, un uso de éstas inadecuado puede provocar problemas de legibilidad y entendimiento.

  Veamos una serie de ejemplos de un uso de cláusulas SQL de forma no adecuada.

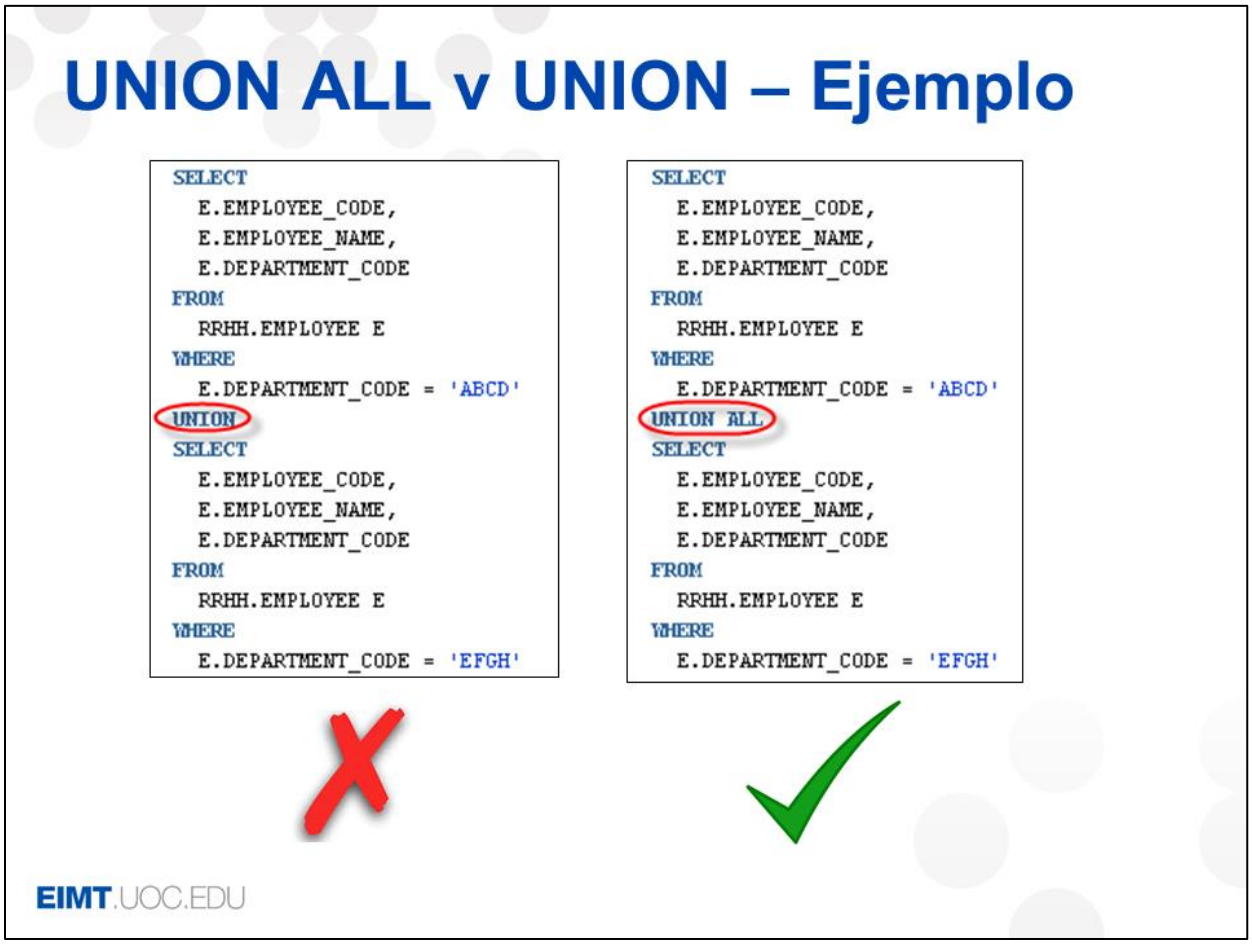


Suponiendo una consulta que obtiene información de empleados y departamentos, es un error añadir la cláusula `DISTINCT` para obtener detalles de los diferentes empleados cuando la clave primaria del empleado es parte de la lista de columnas proyectadas de la consulta. En estos casos, se le está exigiendo al SGBD que realice tareas para distinguir los diferentes empleados de forma innecesaria. Este tipo de problemas se suelen dar también con las cláusulas `ORDER BY` y `UNION/UNION ALL` (`UNION` elimina duplicados, `UNION ALL` los mantiene).

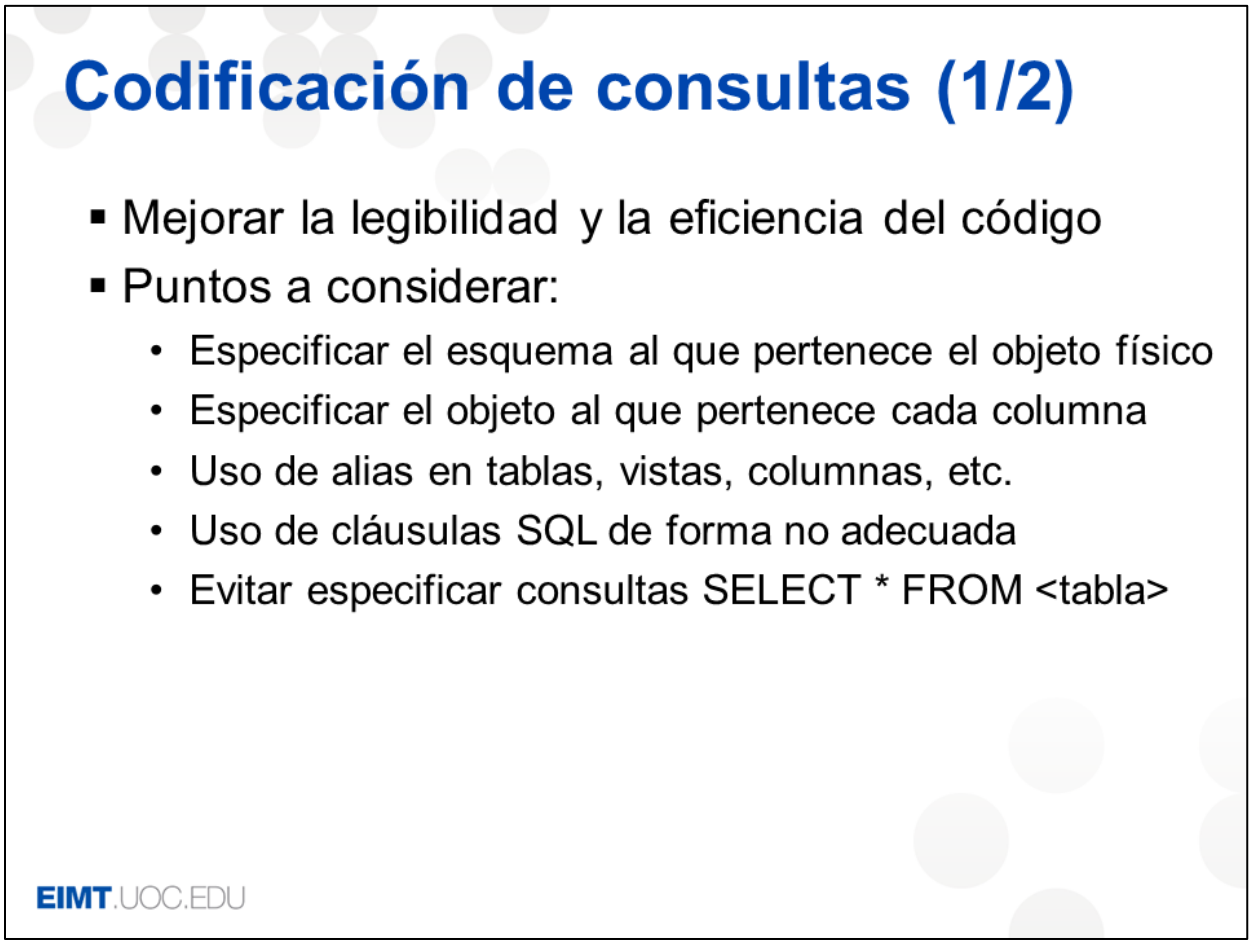
En resumen: si la ordenación de datos o eliminación de duplicados no son necesarias por la construcción/significado de la consulta, estas cláusulas deben ser eliminadas de la consulta.



Supongamos ahora que queremos obtener los diferentes códigos de departamento y la fecha de contratación de nuestros empleados para estudiar los períodos de contratación de cada departamento. La consulta mostrada en la parte superior realiza esta tarea mediante un `GROUP BY`. Si bien nos proporcionará los resultados esperados, el propósito de la cláusula `GROUP BY` es el de agregar información y no para obtener los diferentes valores de un conjunto de columnas. En estos casos, el propósito de la consulta es confuso, por lo que se recomienda la utilización de la cláusula `DISTINCT` para evitar malos entendidos.



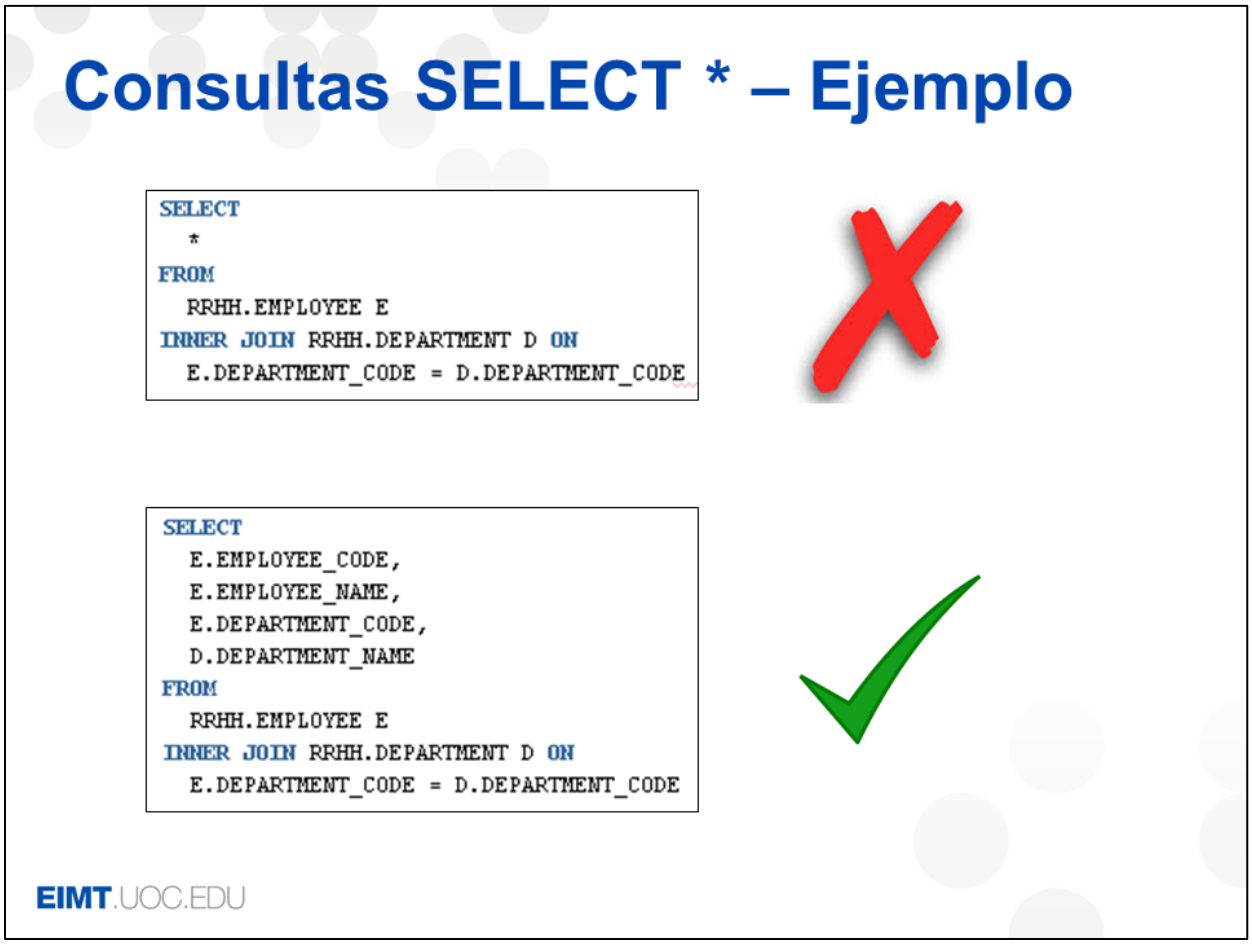
En cuanto al uso de `UNION ALL` y `UNION`: a la hora de realizar una operación `UNION`, es importante entender si el resultado esperado requiere que se eliminen las filas duplicadas. En caso de que no sea necesario eliminarlas, se recomienda el uso de `UNION ALL` en lugar de `UNION`, la primera cláusula no elimina duplicados y simplemente concatena los dos conjuntos de datos, sin necesidad de combinar ambos conjuntos con el fin de eliminar filas duplicadas. Esto se puede ver en el siguiente ejemplo, donde se obtiene la unión de empleados de dos departamentos diferentes, en los que nunca vamos a encontrar duplicados. El uso de `UNION` en la consulta de la izquierda requiere que el SGBD elimine los duplicados, mientras que `UNION ALL` en la consulta de la derecha simplemente concatena los resultados de ambas subconsultas.



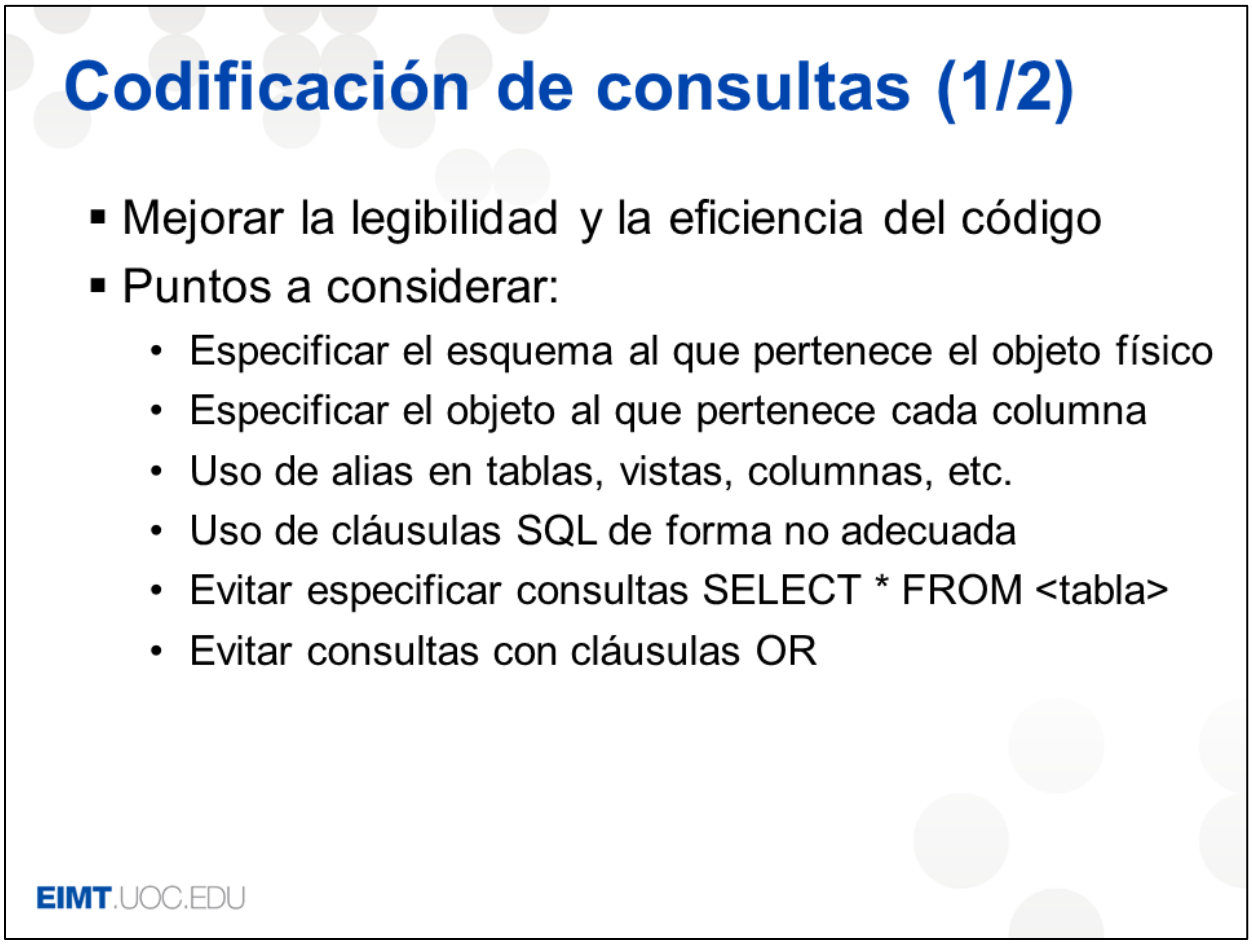
- *Evitar especificar consultas SELECT * FROM <tabla>*: el uso de consultas del estilo `SELECT * FROM <tabla>`, donde se obtienen todos los campos de las tablas a las que se hace referencia, es bastante común en aplicaciones de bases de datos. Consultas de este tipo deben evitarse, ya que suelen causar varios problemas, entre estos:

    - Generan más operaciones de E/S, al tener que obtener todos los valores de todas las columnas, aunque éstas no se vayan a utilizar por parte de la aplicación. Esto causa una pequeña deficiencia en el rendimiento de la consulta.
    - Pueden causar problemas cuando se eliminan o se añaden nuevas columnas en las tablas, y la aplicación que las utiliza no está diseñada de forma correcta para gestionar estos casos.

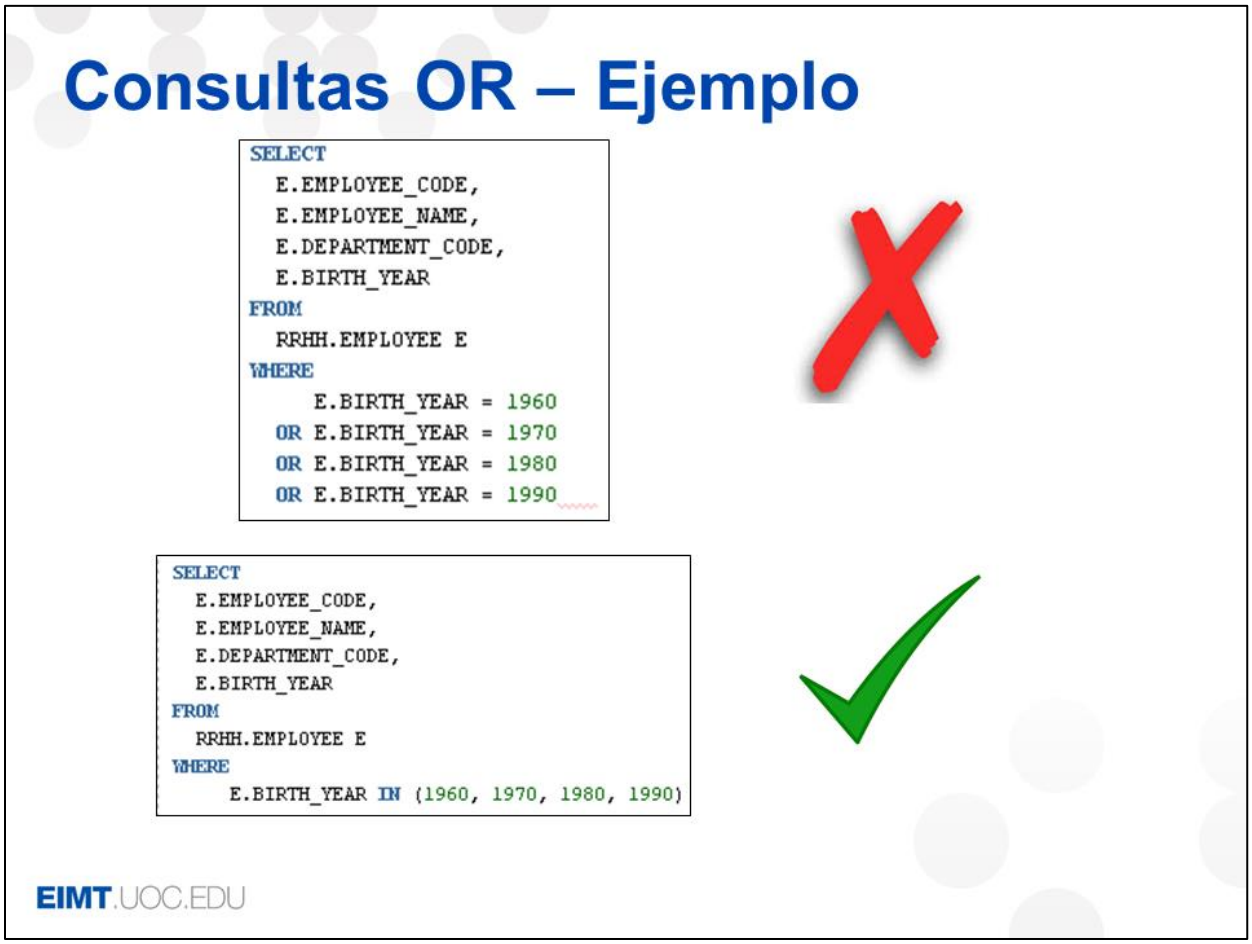
  Se recomienda proyectar solamente las columnas que se vayan a utilizar. Veamos un ejemplo de esta buena práctica.



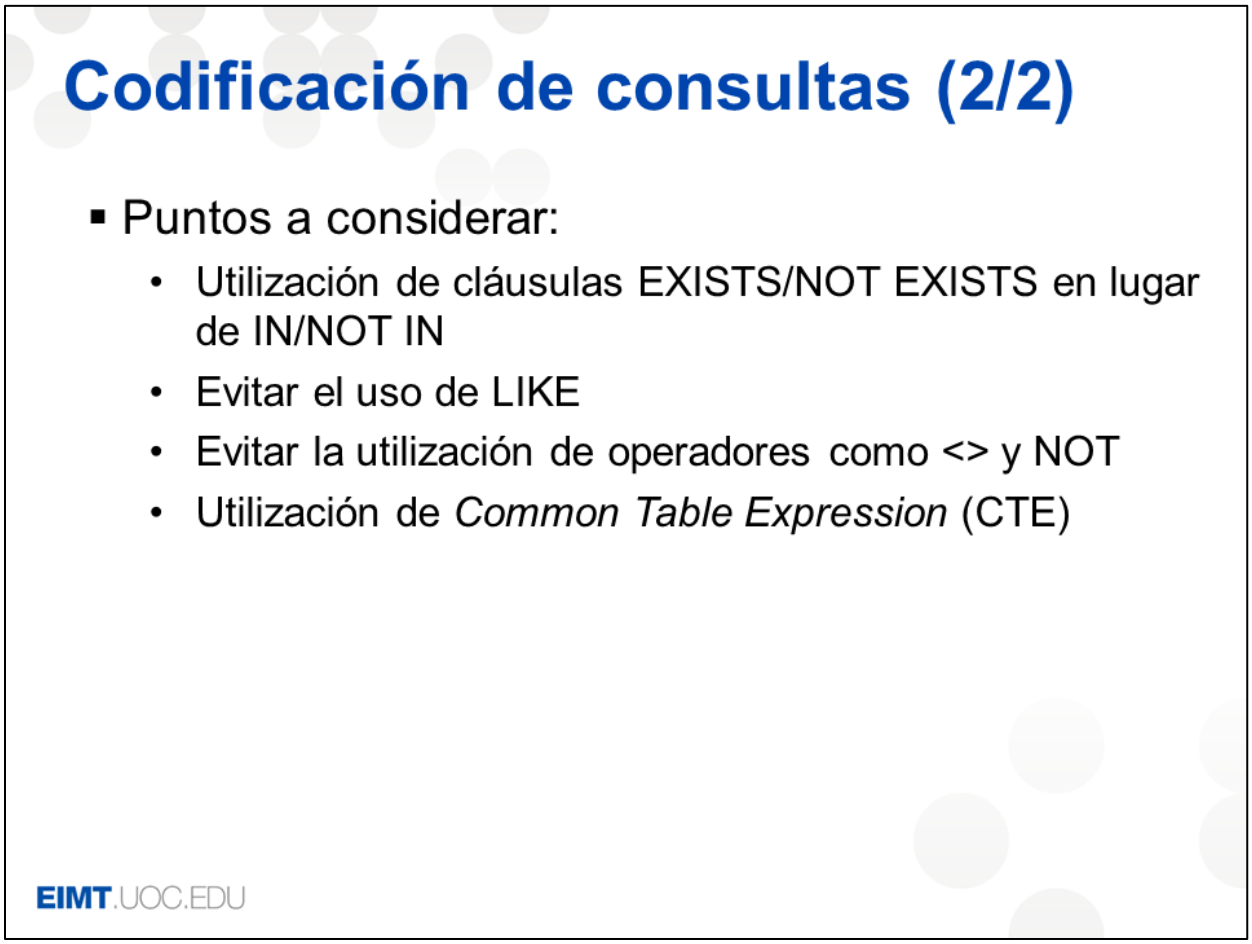
Supongamos que queremos obtener el código y nombre del empleado, y el código y el nombre del departamento en el que trabaja. Para obtener esta información, debemos de utilizar las tablas *employee* y *department*. En la imagen superior, la consulta proyecta todos los campos de ambas tablas (no solamente los que se piden). En estos casos, se considera una mala práctica el uso de *, ya que nos devuelve más información de la que necesitamos, además de que podrían causar problemas en las aplicaciones en caso de que dichas tablas sufriesen modificaciones (por ejemplo, nuevas columnas o columnas eliminadas). La consulta que debería de implementarse se muestra en la segunda imagen.



- *Evitar consultas con cláusulas OR*: a la hora de implementar consultas SQL, existen ocasiones en las que tenemos que implementar condiciones para obtener datos en base a una serie de valores que puede tomar una columna en concreto. Para la implementación de este tipo de consultas, se puede y se suele utilizar el operador `OR`. En casos como éste, se suele recomendar el uso de `IN` en lugar de `OR`, ya que no solamente es más legible, sino que además suele producir un mejor rendimiento, especialmente cuando la lista de valores es larga. Incluso se podría considerar la opción de `UNION ALL`, de forma que cada consulta que forma parte de la `UNION ALL` sea una consulta con una condición de igualdad sin `OR` para un valor/conjunto de valores en concreto. En cualquier caso, se recomienda siempre revisar el plan de ejecución de cada una de las consultas para verificar cuál ofrece el mejor plan de acceso.



Por ejemplo, si queremos obtener la lista de empleados que han nacido en los años 1990, 1980, 1970 o 1960 respectivamente, se podría crear la consulta utilizando el operador `OR` (como se muestra en la primera imagen). En esta consulta, si bien nos devolvería los resultados esperados, vemos que hay una repetición de la misma condición. Esto podría simplificarse utilizando el operador `IN` (segunda imagen), que generalmente, el SGBD implementa de manera más óptima que un `OR`.



Continuamos con la lista de puntos a considerar:

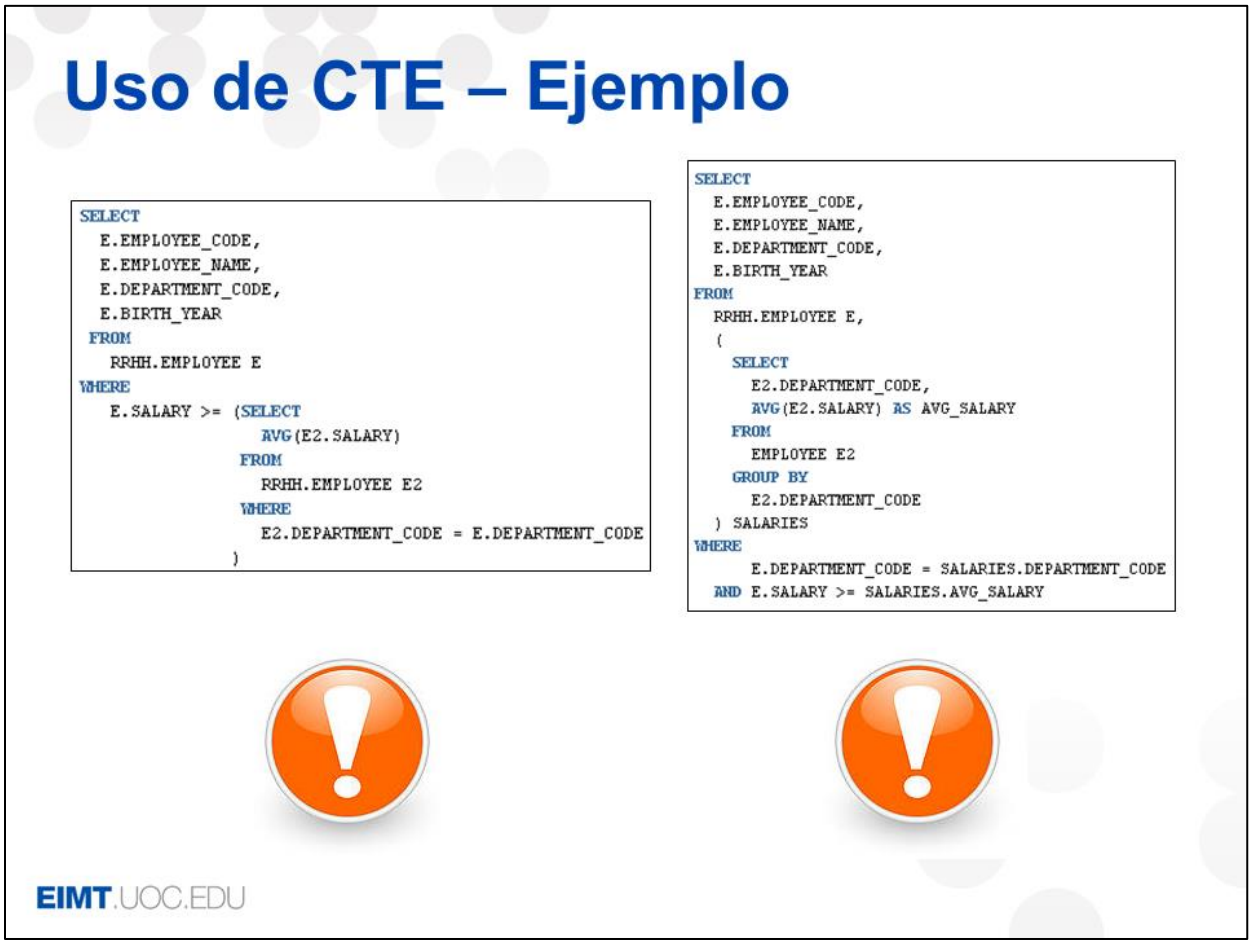
- *Utilización de cláusulas EXISTS/NOT EXISTS en lugar de IN/NOT IN*: generalmente, el uso de `EXISTS` en lugar de `IN` (y sus variante negadas) genera un mejor rendimiento, además de evitar la problemática de tratamiento de valores nulos, con la que podríamos estar obteniendo resultados incorrectos en caso de que existan valores nulos en la columna utilizada en la condición.

- *Evitar el uso de LIKE*: el operador `LIKE` nos permite verificar si columnas de tipo carácter cumplen con una determinada característica mediante el uso de patrones. El uso de `LIKE` es costoso, ya que requiere verificar dentro de cada cadena de caracteres si se cumple la condición especificada, y esto requiere acceso a las diferentes posiciones del valor comparado, por lo que el SGBD tardará más tiempo en devolver resultados. En la medida de lo posible, el uso de este operador debe evitarse para no producir posibles problemas de rendimiento.

- *Evitar la utilización de operadores como <> y NOT*: el uso de operadores con lógica negativa como <> y `NOT` en consultas suelen producir algún que otro problema de rendimiento, en comparación con aquellas consultas en las que se utilizan operadores con lógica positiva (por ejemplo, =), sobre todo en el caso de que existan índices especificados en dichas columnas, índices que el optimizador de

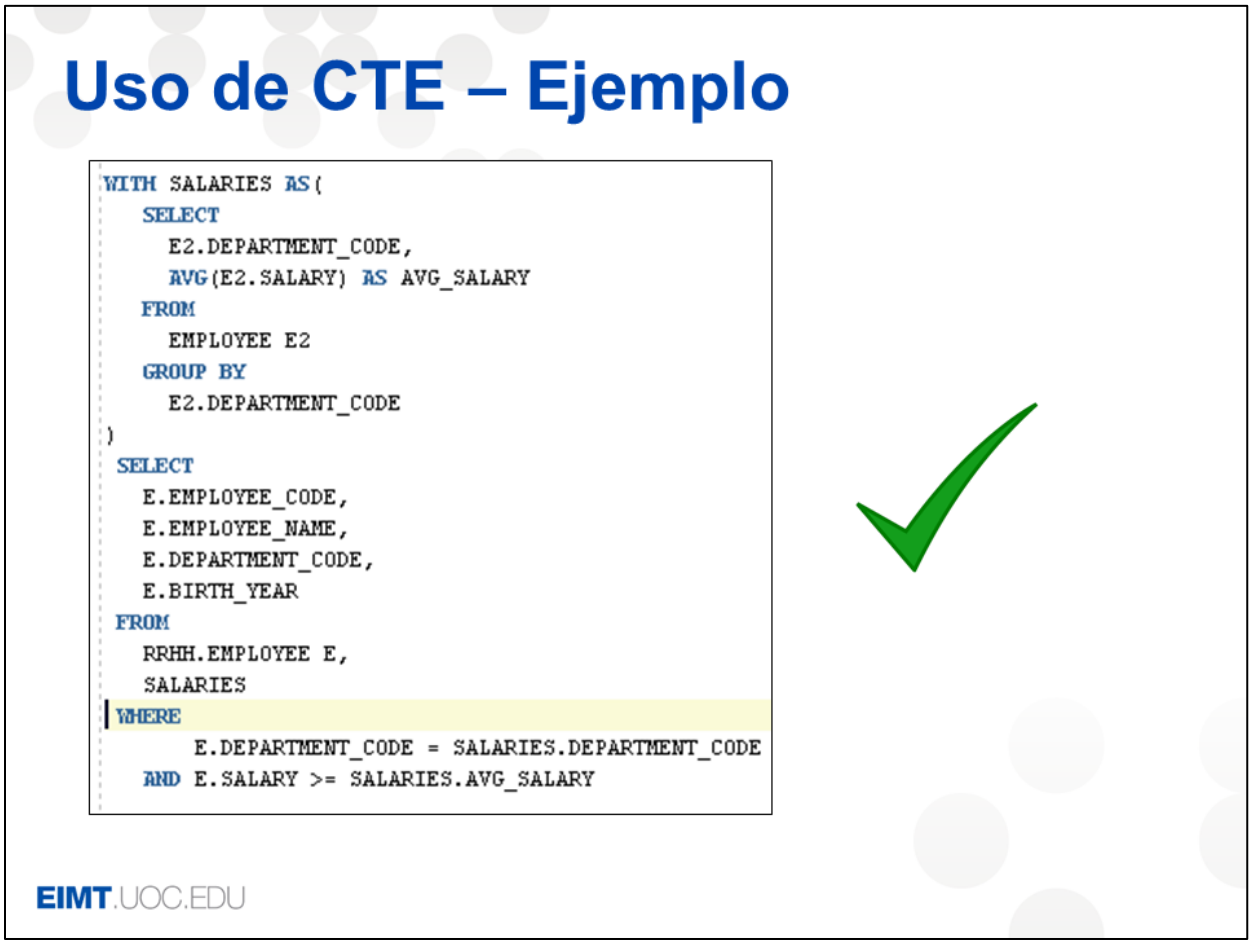
consultas no suele utilizar cuando se encuentra con un operador negativo. En la medida de lo posible, deberemos evitar este tipo de operadores.

- *Utilización de Common Table Expression (CTE) para facilitar la legibilidad del código*: las tablas derivadas (subconsultas que aparecen en la cláusula `FROM`) son difíciles de mantener, además de dificultar la lectura de la consulta. En estos casos, se recomienda el uso de CTE para facilitar la legibilidad y el mantenimiento de la consulta. Veamos un ejemplo de este punto.

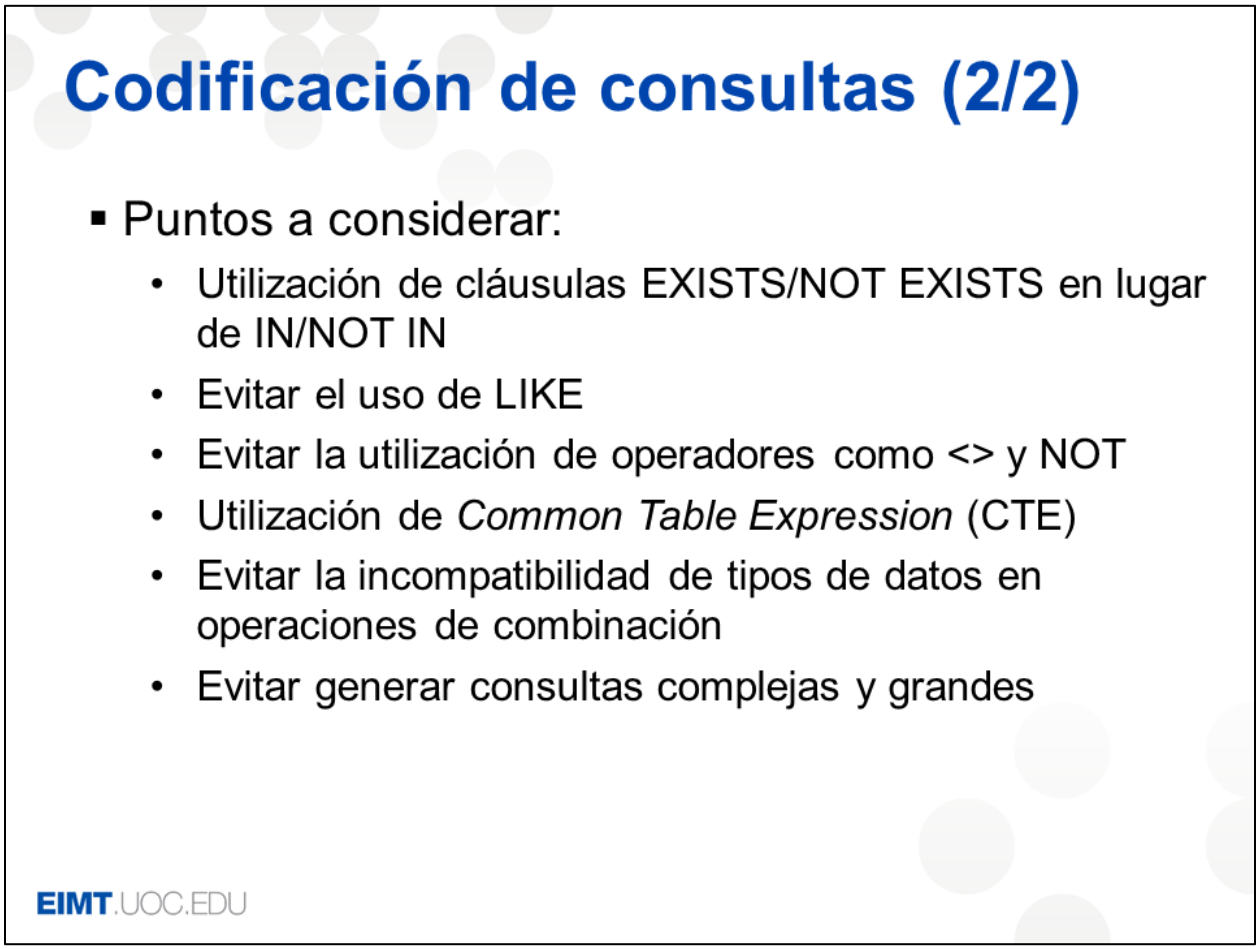


Supongamos que queremos implementar una consulta SQL que obtiene los empleados que tienen un salario igual o superior a la media del salario del departamento en el que trabajan. Esta consulta se podría implementar de diferentes formas mediante el uso de subconsultas. En la imagen de la izquierda vemos que para calcular la media por departamento, se ha utilizado una subconsulta como parte de la cláusula `WHERE`. En la consulta de la derecha, se ha utilizado una tabla derivada (una subconsulta como parte del `FROM`).

Ambas consultas, si bien generan los resultados esperados, podrían considerarse poco legibles o incluso podrían originar un mantenimiento más alto. Además, la primera de ellas no permite referenciar la información de la media de empleados por departamento en caso de que esta información necesite referenciarse en otras partes de la consulta principal, si bien la segunda podría referenciarse por ser parte de la cláusula `FROM`.



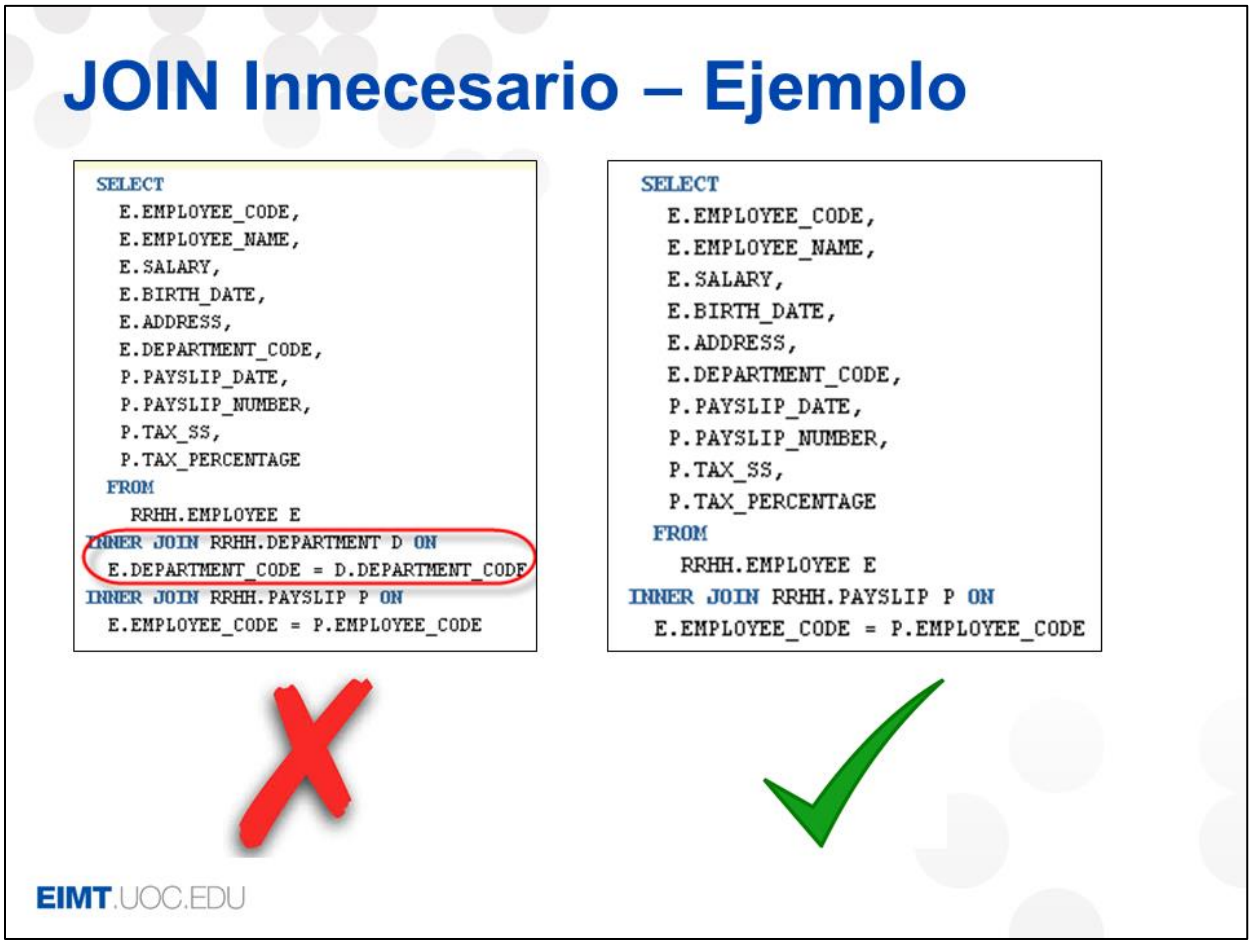
Estas dos consultas se podrían haber implementado mediante la técnica de CTE con la cláusula `WITH`, como se puede ver en esta transparencia. Vemos que, en este último caso, la consulta es más legible, permitiendo diferenciar la subconsulta que calcula la media de salario y la consulta principal, que obtiene el listado de empleados que cumple con la condición requerida.



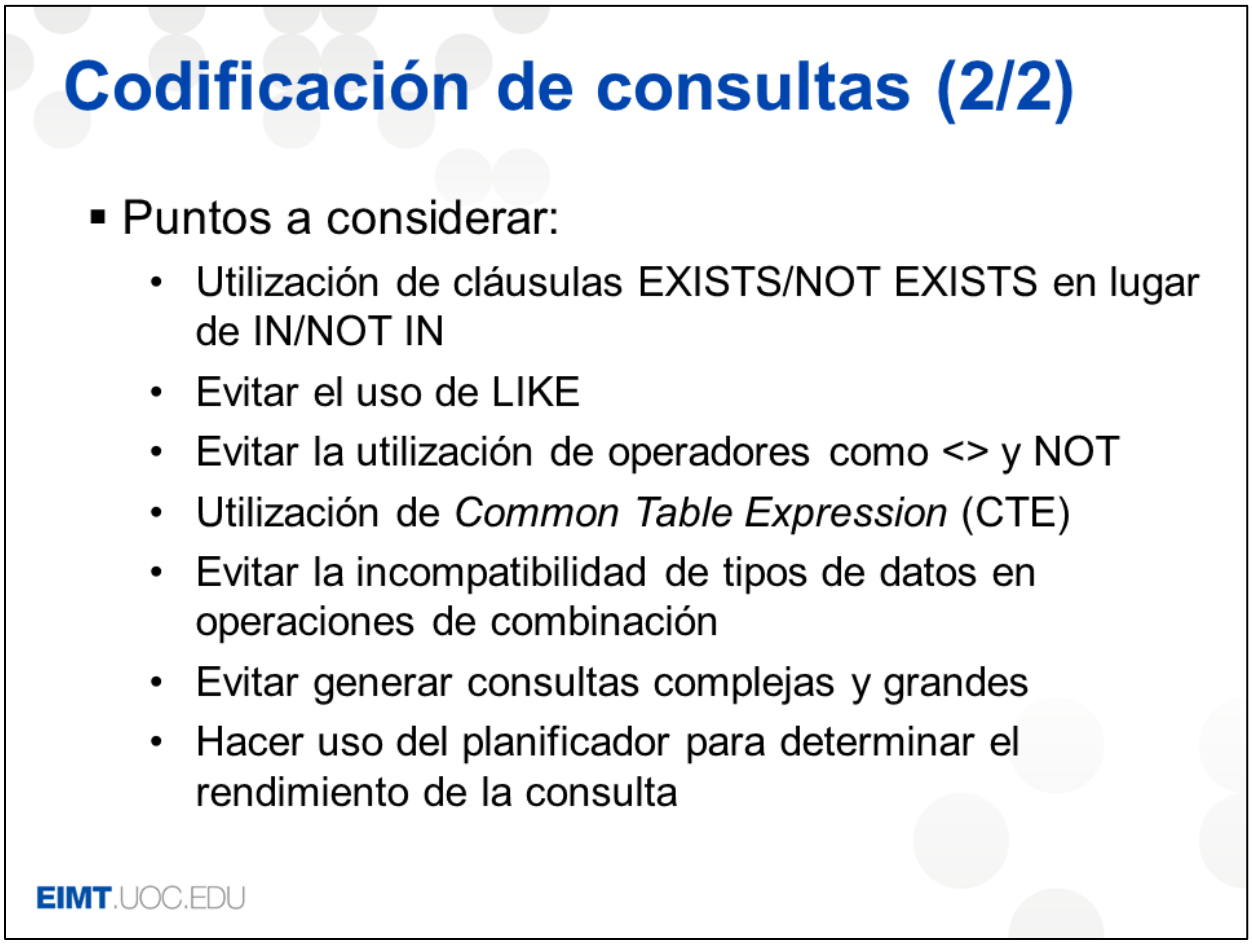
Continuamos con los siguientes puntos:

- *Evitar incompatibilidad de datos operaciones de combinación*: cuando realizamos operaciones de combinación entre tablas, utilizamos un conjunto de columnas para realizar la comparación entre ambas. Si los tipos de datos no son iguales, los SGBD suelen transformar el tipo de dato de una de las columnas para que sea compatible con el tipo de dato de la columna perteneciente a la otra tabla. Esta funcionalidad, aunque aparentemente sea útil, genera una degradación en el rendimiento de las consultas debido a la necesidad de transformar un tipo de dato a otro.

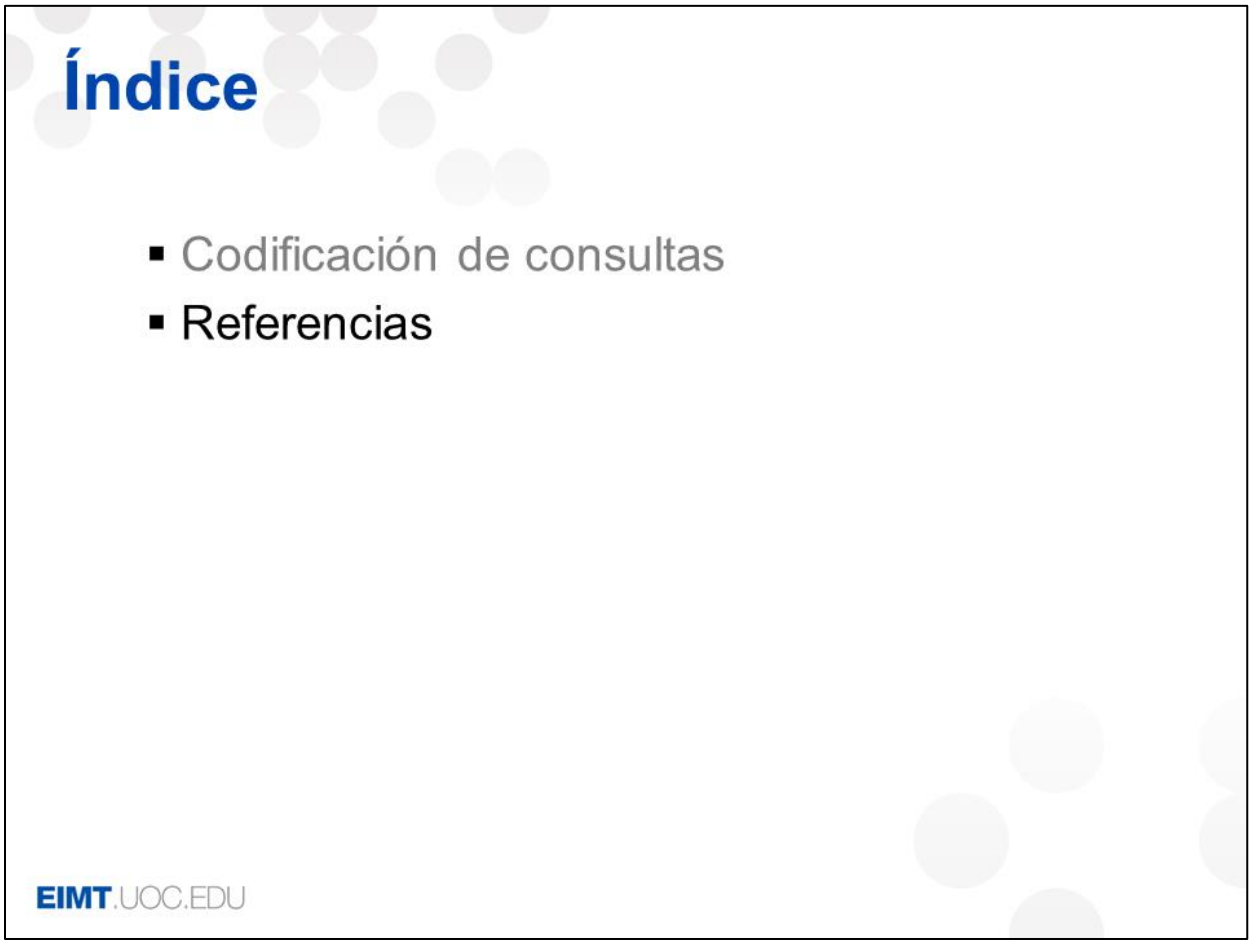
- *Evitar generar consultas complejas y grandes*: consultas complejas y de gran tamaño que utilizan muchas tablas y operaciones de combinación pueden requerir un tiempo considerable para conseguir que éstas sean lo más óptimas posibles. En estos casos, es posible que el optimizador de consultas tenga que utilizar heurísticas para generar el plan de consulta, descartando así otras estrategias más óptimas, por lo que es posible que el plan generado no sea el más óptimo. En estos casos, se debe, siempre en la medida de lo posible, revisar si la consulta requiere utilizar de todas los objetos y operaciones de combinación codificados. Veamos un ejemplo de esto último.



Supongamos que necesitamos una consulta que nos proporcione información de empleados (*employee*) y de las tasas aplicadas en las nóminas (*payslip*), además del número de nómina y de la fecha emitida. La consulta que se presenta en la izquierda nos proporciona dicha información, pero vemos que realiza además una operación de combinación con la tabla de departamentos (*department*) de forma innecesaria, ya que no se proyecta ninguna información procedente de esta tabla. La consulta de la derecha nos proporcionaría exactamente la misma información además de evitar combinaciones innecesarias.



- *Hacer uso del planificador para determinar el rendimiento de la consulta*: el planificador de consultas nos proporcionará información para entender el plan de consulta y determinar el rendimiento de esta. Cuando trabajemos con consultas SQL, es muy importante que utilicemos el planificador para revisar el plan de consulta y así decidir si nuestra consulta tiene un rendimiento aceptable o no.



Y hasta aquí hemos llegado con este segundo vídeo de la serie *Buenas prácticas en SQL*. Esperamos que os haya gustado la presentación y que esta os haya servido de mucha ayuda.

Presentaremos ahora un conjunto de enlaces y referencias de interés acerca de este tema.

Referencias

Rankins, R.; Bertucci, P.; Gallelli, C.; Silverstein, A.. (2013). *Microsoft® SQL Server 2012 Unleashed*. Sams.

PostgreSQL:

http://www.postgresql.org/docs/9.3/

SQL Server Performance:

http://www.sql-server-performance.com/2001/sql-best-practices/

EIMT.UOC.EDU

¡Que tengáis un buen día!