Bones pràctiques en SQL

Codificació de procediments i funcions

PID_00253044

Alexandre Pereiras Magariños

Universitat Oberta de Catalunya

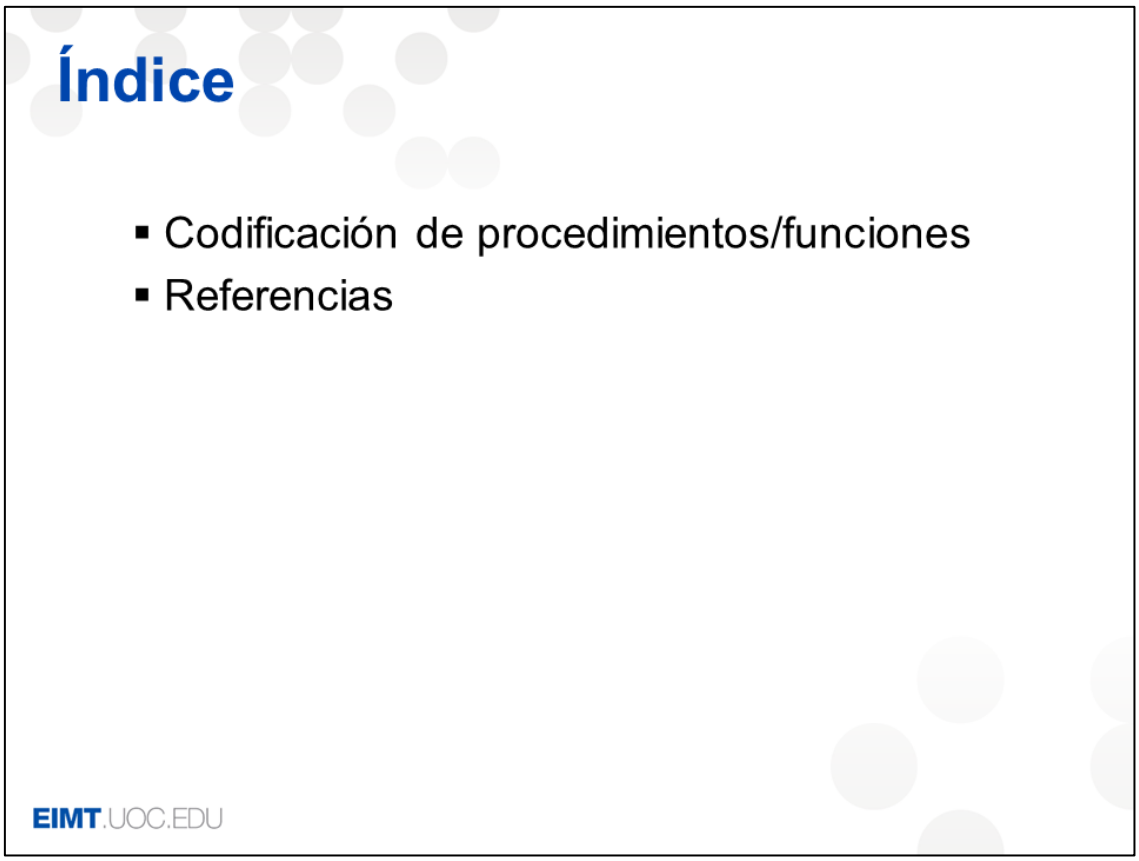
Benvinguts al tercer vídeo de la sèrie *Bones pràctiques en SQL*, sèrie en què veurem un conjunt de bones pràctiques a l'hora de programar en SQL en entorns de bases de dades/*data warehouse*.

Aquesta sèrie de vídeos està dividida en 4 parts:

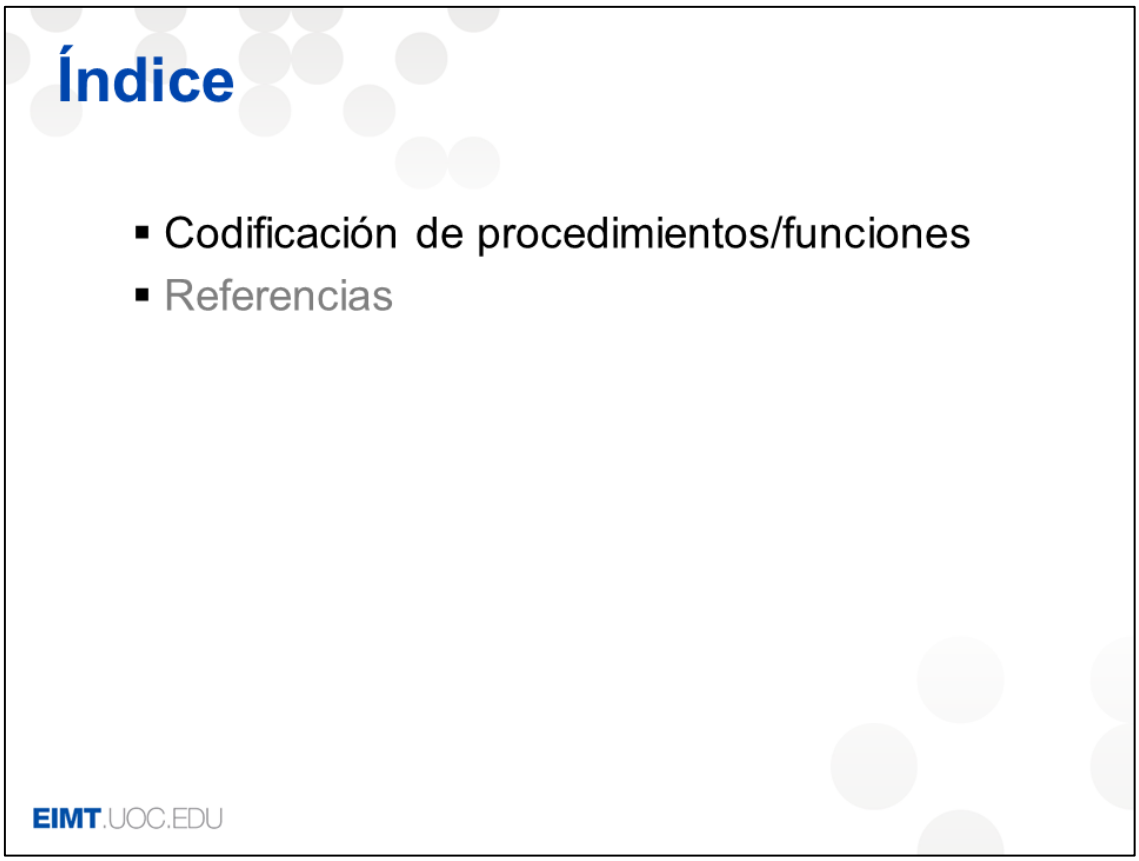
- *Codificació SQL*: centrada en aquelles pràctiques des del punt de vista genèric en SQL, que afecten la llegibilitat i la portabilitat del codi SQL.
- *Codificació de consultes*: en què veurem pràctiques que ens ajudaran a generar consultes SQL més eficients i llegibles.
- *Codificació de procediments i funcions*: en què pararem esment en aquelles pràctiques que ens permetran escriure el codi SQL en un servidor més eficient i gestionar errors de forma controlada.
- *Codificació de transaccions*: en què detallarem pràctiques per a assegurar un control correcte de les transaccions que codifiquem.

És important destacar que, dins de cada categoria, no es proporcionen totes les possibles bones pràctiques del mercat, sinó que es tracta d'un conjunt concret que des de la UOC hem considerat més rellevants.

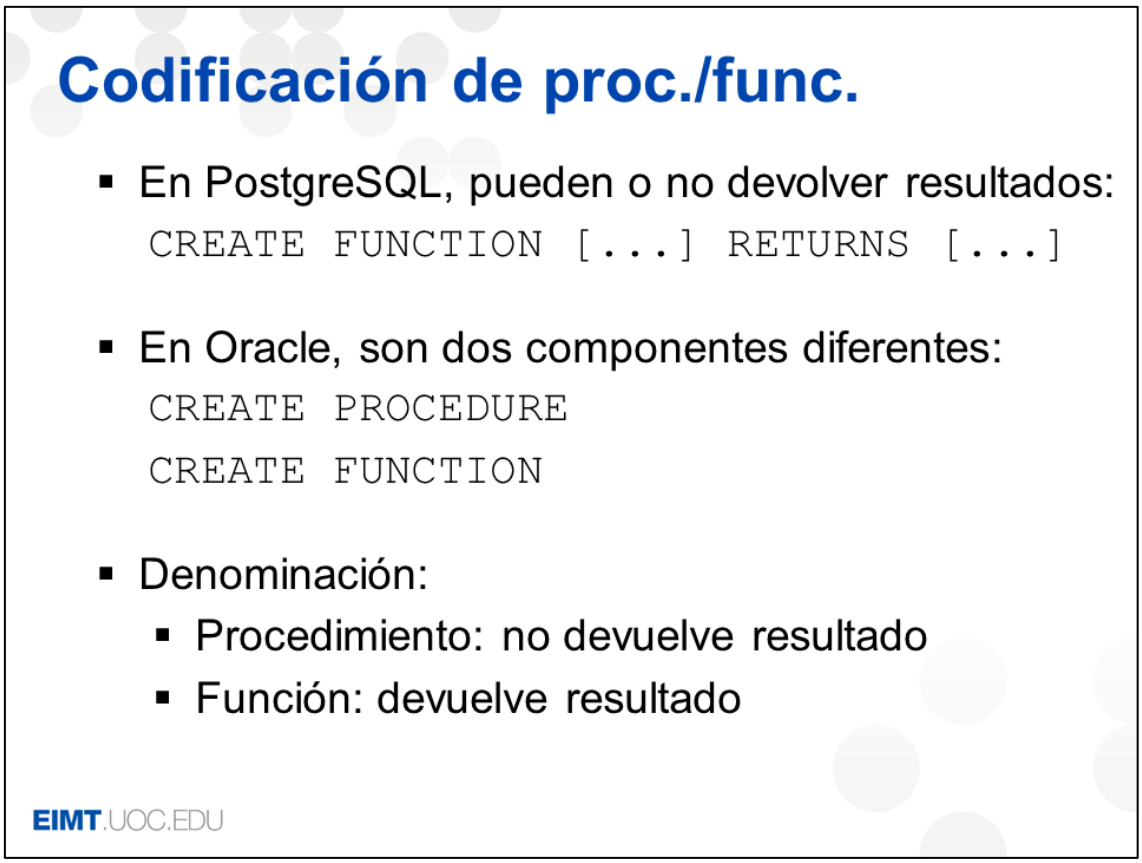
Aquest vídeo afrontarà la tercera part, *Codificació de procediments i funcions*.



Aquest tercer vídeo se centrarà a presentar una sèrie de bones pràctiques que ens permetran codificar procediments i funcions en codi SQL. A més, al final del vídeo us proporcionarem les referències bibliogràfiques utilitzades.



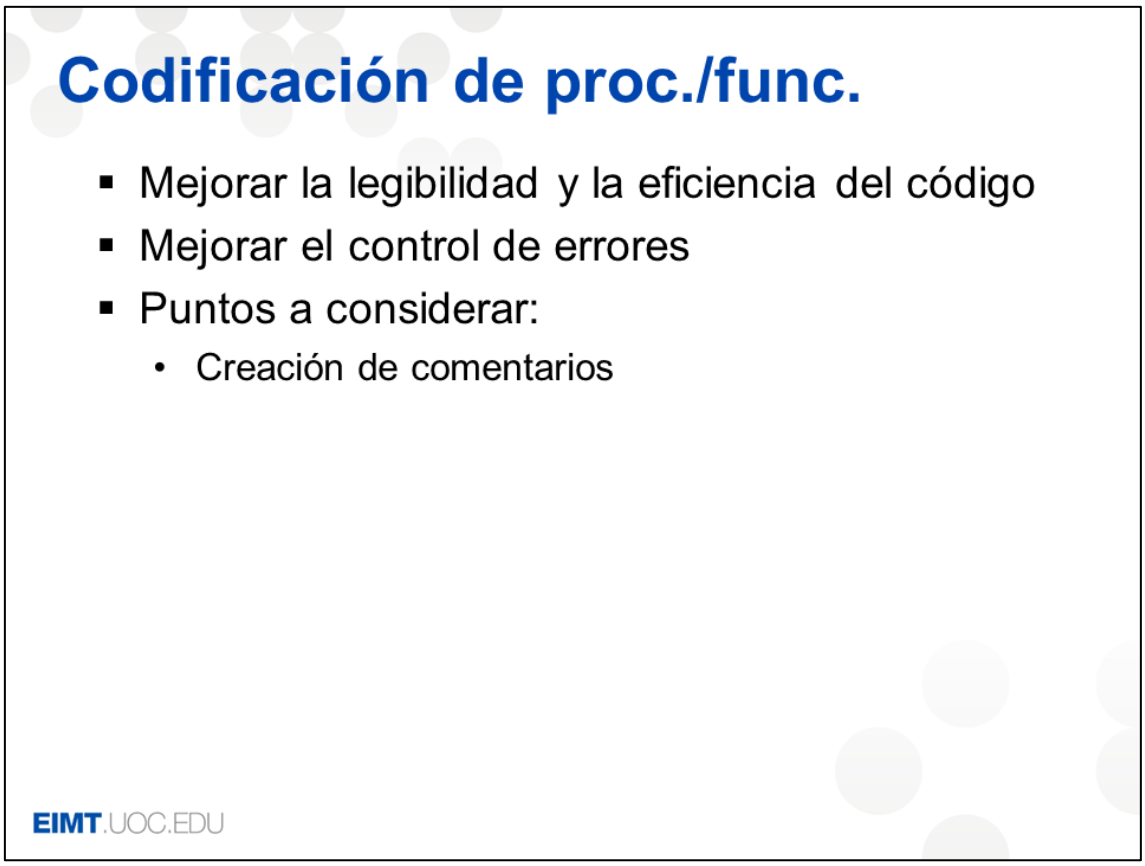
Vegem, doncs, la tercera categoria de bones pràctiques: codificació de procediments i funcions.



Abans de començar amb la secció de procediments i funcions, és necessari aclarir la terminologia utilitzada de **procediment** i **funció**.

En PostgreSQL, els procediments emmagatzemats poden o no retornar un resultat. Aquesta funcionalitat difereix d'altres SGBD, com ara Oracle, en què tots dos tipus requereixen clàusules SQL concretes (`CREATE PROCEDURE` o `CREATE FUNCTION`, on la primera no retorna cap resultat i la segona sí) en oposició amb PostgreSQL (`CREATE FUNCTION [...] RETURNS [...]`).

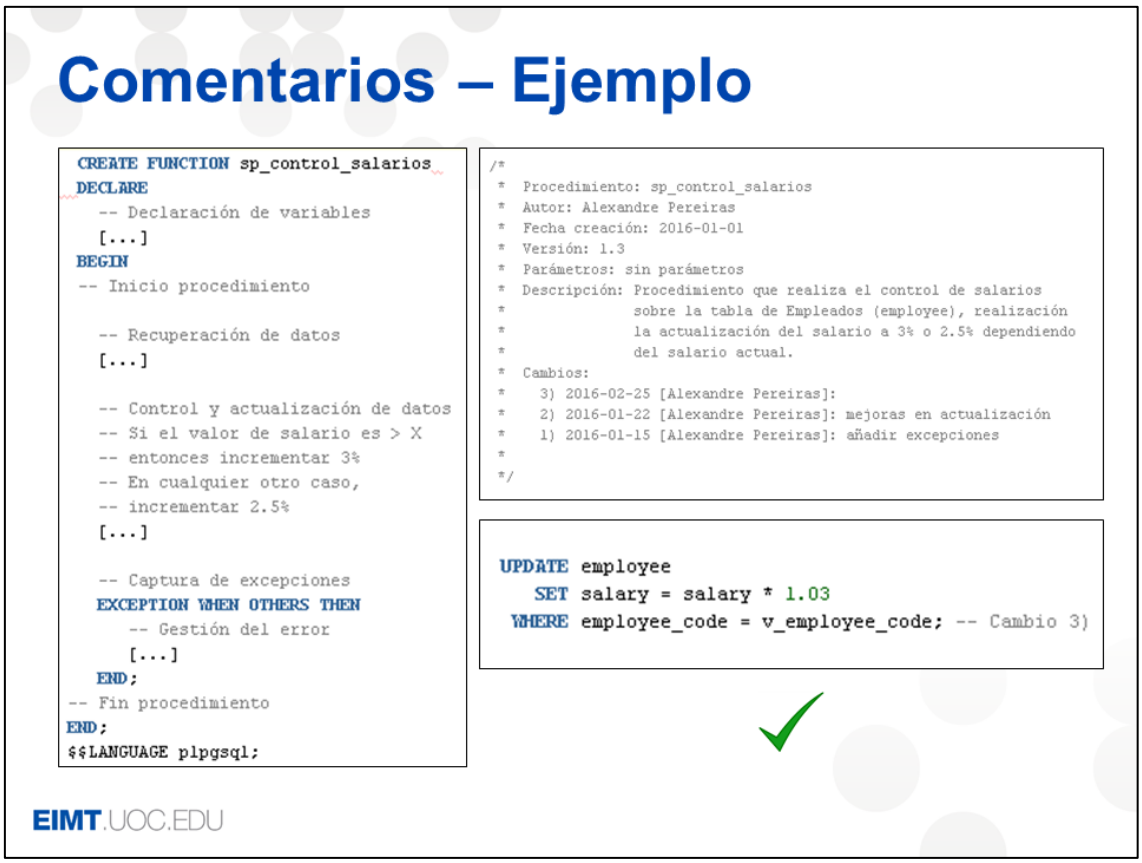
Amb la finalitat de diferenciar tots dos tipus en PostgreSQL, definirem com a **procediment** aquells procediments emmagatzemats que **no retornen cap resultat** i com a **funció** aquells procediments emmagatzemats que **sí que retornen un resultat**.



Després d'aclarir els termes de procediment i funció, revisarem les pràctiques que ens permetran codificar procediments i funcions en codi SQL. Procediments emmagatzemats mal codificats o ineficients ens poden causar problemes de rendiment en les aplicacions i, fins i tot, errors a l'hora de gestionar transaccions.

Des d'aquest punt de vista, es recomanen els punts següents:

- *Crear comentaris que ajudin a entendre el codi generat*: el fet de comentar el codi en un procediment emmagatzemat ajuda a altres programadors a entendre el codi de forma més clara quan un bloc de codi no resulta tan obvi d'entendre. Sempre es recomana comentar, tan bé com sigui possible, el que el codi SQL realitza dins d'un procediment o funció. També es recomana que, a la capçalera d'aquest procediment i com a comentari, s'afegeixin detalls de l'autor del procediment: data de creació i versió actual, així com un registre dels canvis que aquest procediment ha sofert al llarg de la seva vida: autor i data de la modificació, descripció de la modificació, etc.

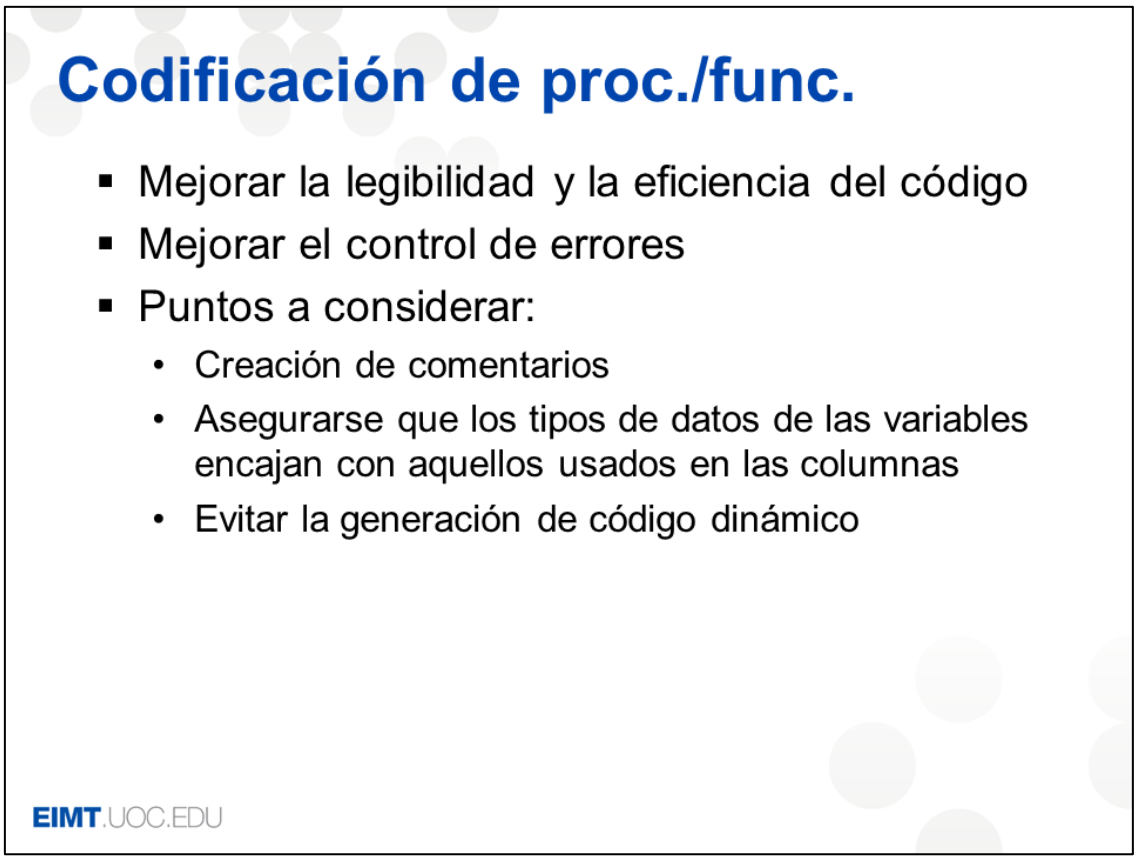


En aquesta transparència, podem veure 3 exemples de comentaris dins d'un procediment emmagatzemat:

- A la imatge de l'esquerra, podem veure exemples de com podem comentar el cos d'un procediment emmagatzemat, per a facilitar el seu manteniment en cas que siguin altres programadors els que s'encarreguin d'aquesta tasca.

- A la imatge superior dreta, podem veure un comentari típic de la capçalera d'un procediment, en què es detallen l'autor original del procediment, el nom del procediment, la data de creació, la versió actual, els paràmetres definits, la descripció del que realitza el procediment i la llista de canvis realitzats en aquest procediment, juntament amb la data, l'autor i la descripció de cada canvi.

- Finalment, a la imatge inferior dreta, podem veure un comentari en una de les línies de la sentència SQL que indica on s'ha realitzat el canvi núm. 3.



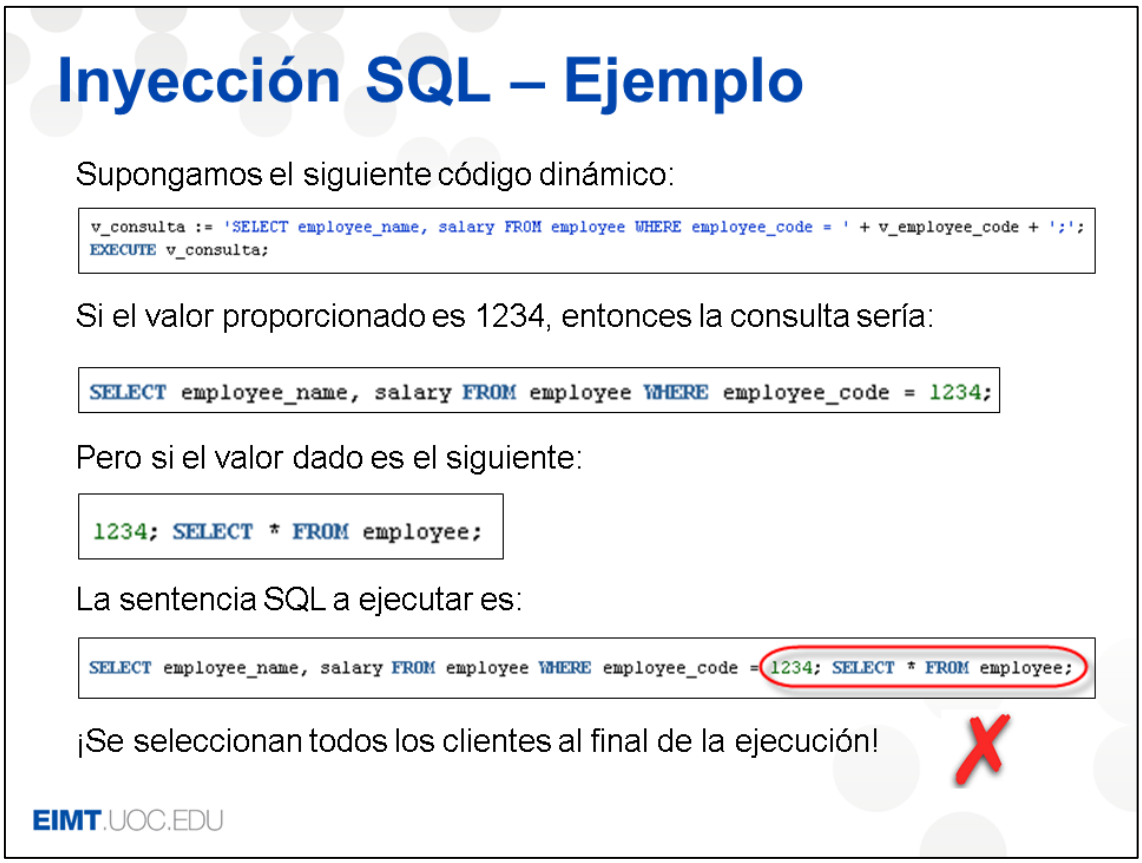
Continuem amb els punts següents:

- *Assegurar-se que els tipus de dades de les variables encaixen amb els de les columnes*: aquesta pràctica ens assegura que ens evitem problemes de compilació quan generem el procediment o funció, a més d'evitar problemes d'optimització perquè es delega a l'SGBD la transformació d'un tipus de dada a un altre.

- *Utilitzar crides a procediments emmagatzemats des de les aplicacions*: en la mesura que es pugui, es recomana que s'utilitzin procediments emmagatzemats per a l'execució del codi SQL des de les aplicacions. D'aquesta manera, tot el codi que gestiona el model de dades de l'aplicació es troba en la base de dades, actuant com una caixa negra i protegint les aplicacions dels possibles canvis en les estructures o en la lògica de les consultes. D'aquesta forma, solament podem modificar els procediments emmagatzemats sense que es facin canvis en les aplicacions.

- *Evitar la generació de codi dinàmic*: és a dir, en la mesura que es pugui, es recomana que s'evitin sentències SQL dinàmiques per a crear altres sentències SQL dins de procediments o funcions. La raó principal és que l'SQL dinàmic no està compilat i aquest pot generar errors a llarg termini si les estructures de les dades han canviat, a més de ser un codi SQL que és difícil de gestionar. Així mateix, el codi dinàmic comporta un problema de seguretat, denominat

injecció SQL. Aquest mecanisme permet a un atacant injectar un codi SQL dins d'un codi SQL programat, tractant-se generalment d'un codi SQL programat dinàmic. Vegem un exemple d'injecció SQL per a explicar aquesta vulnerabilitat.

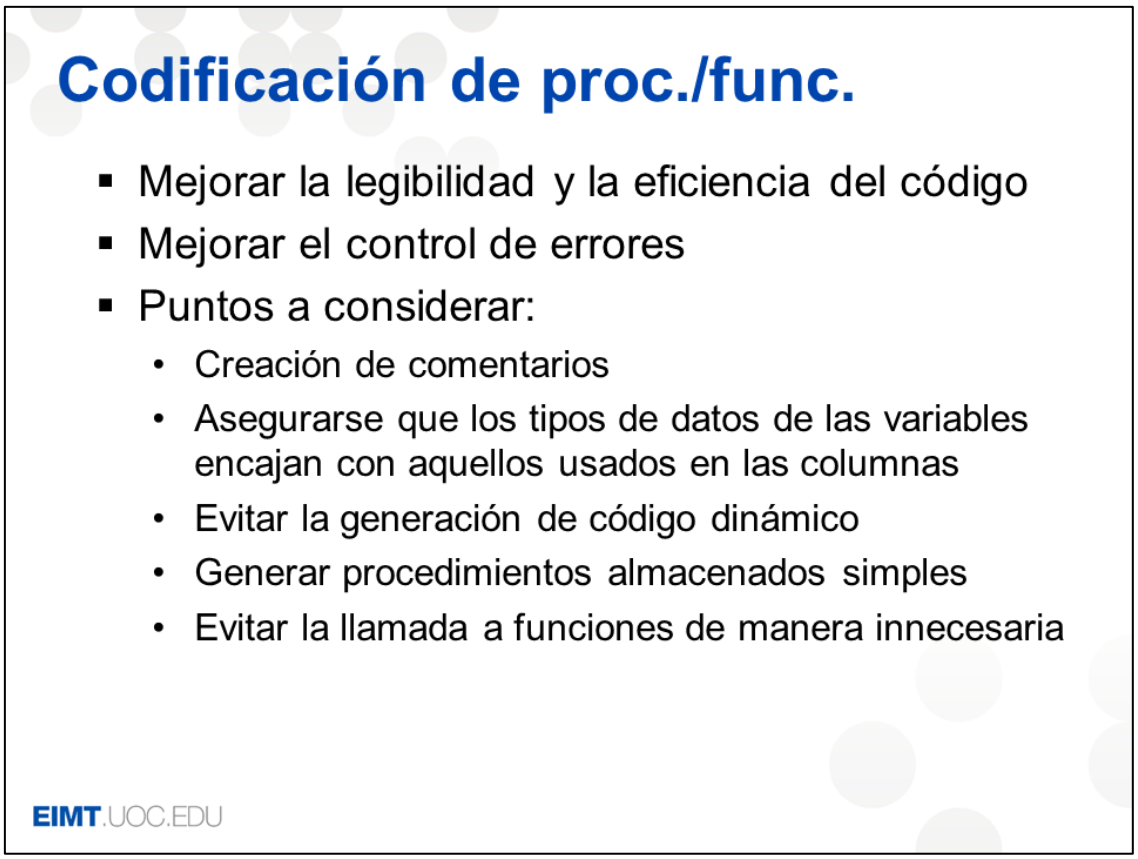


Suposem el codi dinàmic següent, on es construeix una consulta SQL per a obtenir detalls d'un empleat en concret sobre la base del codi d'empleat proporcionat per un usuari a través d'una aplicació web. Si l'usuari proporciona el valor 1234, llavors la consulta a executar seria la següent: `SELECT employee_name, salary FROM employee WHERE employee_code = 1234;`

Si l'aplicació web no està preparada per a evitar la injecció SQL, un usuari maliciós podria proporcionar el valor següent (`1234; SELECT * FROM employee;`), per la qual cosa la sentència a executar seria la següent: `SELECT employee_name, salary FROM employee WHERE employee_code = 1234; SELECT * FROM employee;`

Podem veure que el que s'executa són dues sentències SQL diferents, la primera d'obtenció de detalls per a l'empleat amb el codi 1234 i la segona per a obtenir tots els empleats de l'empresa, incloent informació que podria ser confidencial. Imagineu si en lloc d'un `SELECT`, l'usuari afegeix un `DROP TABLE`!

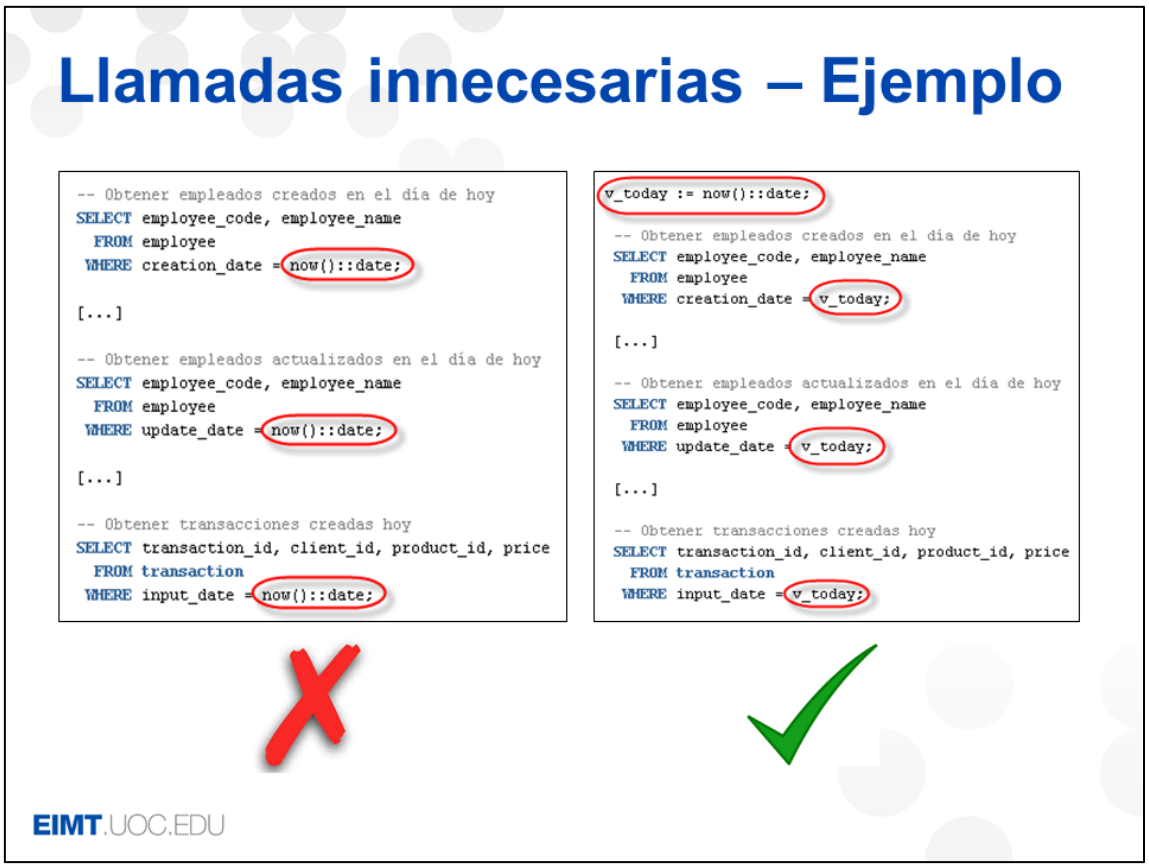
Per tant, sempre en la mesura que es pugui, és preferible evitar l'SQL dinàmic o, si no és possible, proporcionar els mecanismes necessaris per a evitar vulnerabilitats provocades per la injecció SQL.



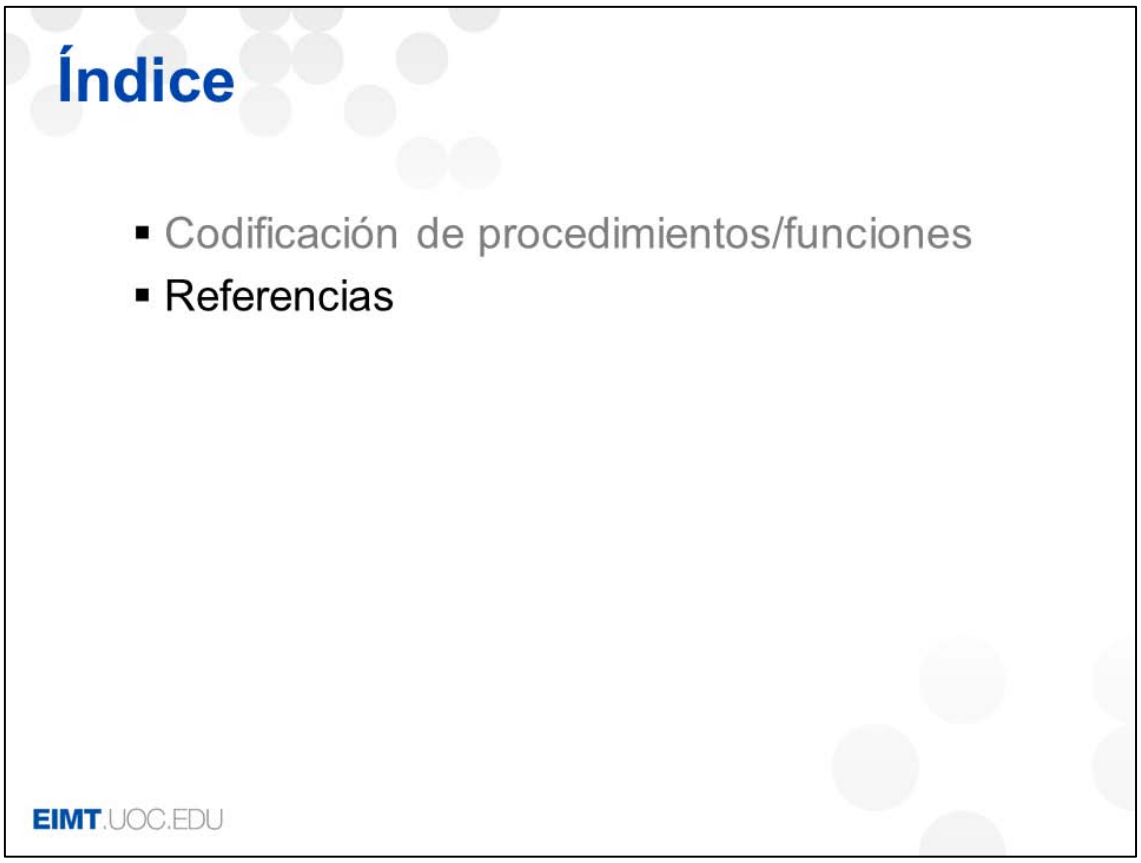
Continuem amb els últims punts:

- *Generar procediments emmagatzemats simples*: és recomanable generar procediments emmagatzemats modulars que permetin la reusabilitat. Per exemple, si volem auditar les operacions d'una base de dades mitjançant la generació d'informació en taules d'auditoria, és recomanable generar els procediments emmagatzemats necessaris per a inserir la informació necessària (que es passaria per paràmetres) i que aquests siguin cridats des d'altres procediments emmagatzemats, en lloc d'implementar la mateixa lògica en tots aquests. D'aquesta manera, aconseguim que els procediments siguin reusables.

- *Evitar la crida a funcions de manera innecessària*: si realitzem la crida a una funció des de diferents punts d'un procediment per a obtenir sempre el mateix resultat, és recomanable emmagatzemar aquest resultat en una variable una sola vegada i fer ús d'aquesta variable al llarg de tot el procediment, perquè el fet que obviem crides a una mateixa funció evita que utilitzem cicles de CPU de forma innecessària. Vegem un exemple.



Imaginem que tenim el codi següent que pertany al cos d'un procediment en què cridem a la funció `now()::date` de PostgreSQL per a obtenir la data en diferents punts, com es pot veure a la imatge de l'esquerra. Aquesta manera de codificar és ineficient, ja que obliguem l'SGBD a realitzar una crida a la mateixa funció diverses vegades, consumint cicles de CPU de forma innecessària. En aquests casos, és millor emmagatzemar el valor d'aquesta funció en una variable al principi i, així, fer ús de la variable al llarg del procediment, com es pot veure a la imatge de la dreta.



I fins aquí hem arribat amb aquest tercer vídeo de la sèrie *Bones pràctiques en SQL*. Esperem que us hagi agradat la presentació i que aquesta us hagi servit de molta ajuda.

Ara us presentarem un conjunt d'enllaços i referències d'interès sobre aquest tema.

Que tingueu un bon dia!