ORACLE

SQL関連新機能

Oracle Database 23c新機能セミナー

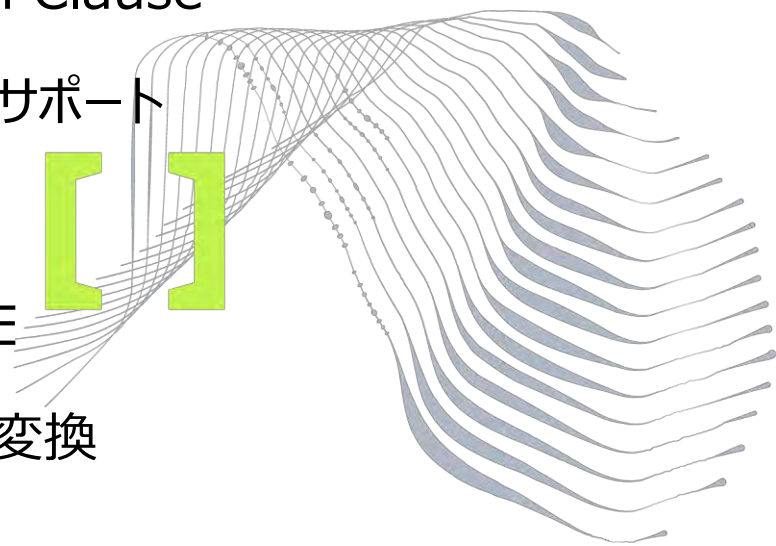
佐々木 経行

日本オラクル株式会社

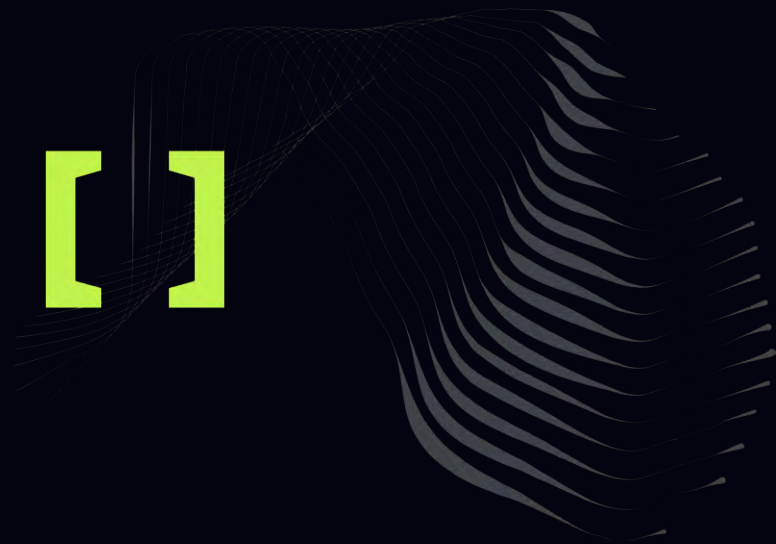
2023年10月20日

Agenda

1. アプリケーション使用用途注釈
2. アプリケーション使用用途ドメイン
3. BOOLEANデータ型
4. 別名と位置によるグループ集計
5. INTERVAL データ型の集計
6. 表の結合による簡単な更新または削除
7. DEFAULT ON NULL for UPDATE Statements
8. 拡張された CASE の制御
9. RETURNING句拡張による更新前後のデータを両方取得
10. SELECT Without FROM Clause
11. IF [NOT] EXISTS 句 のサポート
12. 表値コンストラクタ
13. DB_DEVELOPER_ROLE
14. PL/SQLからSQLへの自動変換



アプリケーション使用用途注釈



アプリケーション使用用途注釈

- 概要
  - 列やオブジェクトにメタデータを設定できるようになりました。
    - テーブル、ビュー、テーブル/ビュー・カラム、マテリアライズド・ビュー、インデックス、ドメインなど
- メリット
  - 列や表に対して、機密情報の注意事項の注釈をつけることができる。
  - アプリケーション開発者に、非表示などデータの扱いについて注釈をつけることができる。
  - テーブル設計書やデータ設計書などのメタデータ文書の作成負荷の軽減が期待できます。

```
CREATE TABLE employee (
  id       NUMBER(5)
              ANNOTATIONS (Identity, Display 'Employee ID', "Group" 'Emp_Info'),
  name     VARCHAR2(50)
              ANNOTATIONS (Display 'Employee Name',  "Group" 'Emp_Info'),
  salary   NUMBER
              ANNOTATIONS (Display 'Employee Salary', UI_Hidden)
)
ANNOTATIONS (Display 'Employee Table');
```

アプリケーション使用用途注釈

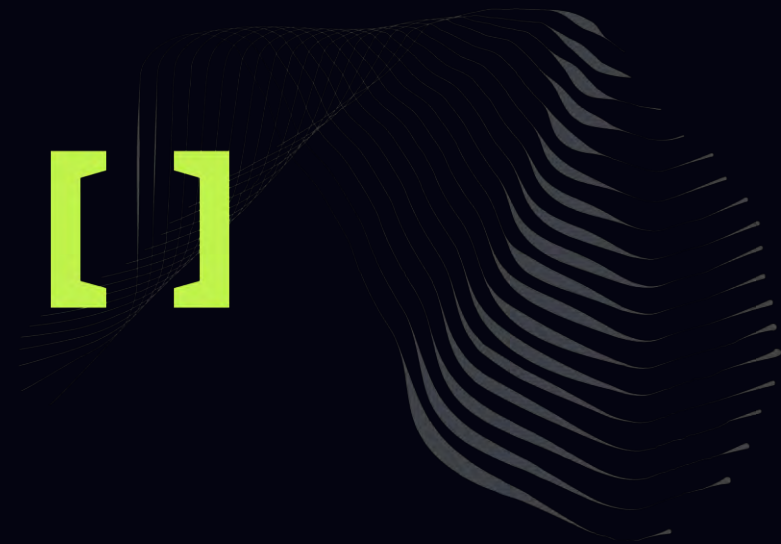
- 表のアノテーションの検索例

```
SELECT ANNOTATION_NAME, ANNOTATION_VALUE from USER_ANNOTATIONS_USAGE WHERE Object_Name =
'EMPLOYEE' AND Object_Type = 'TABLE' AND Column_Name IS NULL;

ANNOTATION_NAME ANNOTATION_VALUE
--------------- ----------------
DISPLAY         Employee Table
```

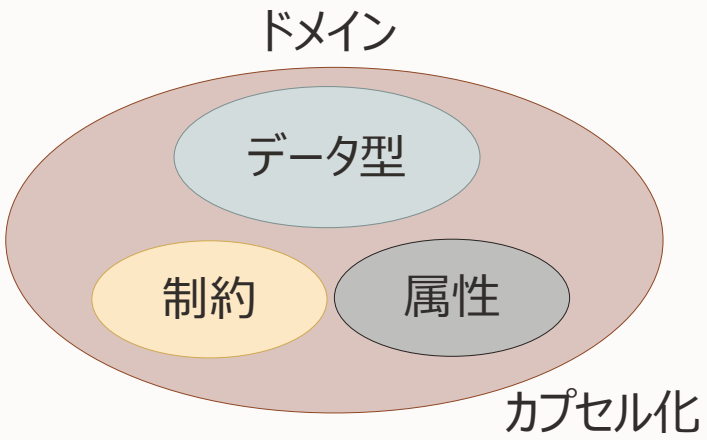
- 列のアノテーションの検索例

```
SELECT COLUMN_NAME, ANNOTATION_NAME, ANNOTATION_VALUE from USER_ANNOTATIONS_USAGE WHERE
Object_Name = ‘EMPLOYEE’ AND Object_Type = 'TABLE' AND Column_Name IS NOT NULL;

COLUMN_NAME ANNOTATION_NAME ANNOTATION_VALUE
----------- --------------- ----------------
ID          IDENTITY
ID          DISPLAY         Employee ID
ID          Group           Emp_Info
ENAME       DISPLAY         Employee Name
...
```

アプリケーション使用用途ドメイン



アプリケーション使用用途ドメイン



- 概要
  - ユーザーがデータ型の属性と制約を定義できるようになった
- メリット
  - これまでデータ型の定義は、VARCHAR2, NUMBER など汎用的なものしか用意されていなかったので
    クレジット カード番号や電子メールアドレスなどは、アプリケーション側で入力チェックする必要あった。
  - アプリケーション固有のクレジット カード番号や電子メール アドレスなどのデータ型をドメインとして
    制約して定義すると、データをテーブルに挿入、更新時にデータが検証チェックされるので、
    データの一貫性が保たれ、データの品質が向上します。
  - これまでは、データの検証チェックは、主にアプリケーション側でされていたが、ドメインでの検証チェックすることにより
    アプリケーションの開発工数の削減が期待できます。
  - また、ドメインではデータ表示方法と表示順序の属性を設定できるようになるので、
    これもアプリケーションの品質向上と開発工数の削減が期待できます。

アプリケーション使用用途ドメイン

- ドメイン構文
  - ユーザーがデータ型の制約と表示を定義

```
CREATE DOMAIN DomainName AS <Data Type>
[ DEFAULT <expression> [ ON NULL ] ] [ NOT NULL ]
[ CONSTRAINT [ name ] CHECK (<expression>) [ ENABLE | DISABLE ] ]
[ COLLATE collation ]
[ DISPLAY <expression> ]
[ ORDER <expression> ]
[ ANNOTATIONS ( annotations ) ]
```

  - 例として、電子メールアドレスをドメインとして制約、表示方法、表示順序を定義

```
CREATE DOMAIN email AS VARCHAR2(255) NOT NULL
CONSTRAINT email_c CHECK (REGEXP_LIKE (email, '^(\S+)\@(\S+)\.(\S+)$'))
DISPLAY '---' || SUBSTR(email, INSTR(email, '@'))
ORDER SUBSTR(email, INSTR(email, '@')+1) || SUBSTR(email, 1, INSTR(email, '@'));
```

アプリケーション使用用途ドメイン

- ドメインの使用例
  - 電子メールアドレスのドメインの列を含む表を作成

```
CREATE TABLE customers (
  cust_id        NUMBER          NOT NULL PRIMARY KEY,
  name           VARCHAR2(4000) NOT NULL,
  contact_email  VARCHAR2(1000) DOMAIN email,
  invoice_email  email
);
```

  - 電子メールアドレスのドメインの列を含む表に、制約違反になる行を挿入し、エラー確認

```
INSERT INTO customers values (1, 'TEST', 'abc', 'abc');

ORA-02290: check constraint (EMAIL_C) violated
```


アプリケーション使用用途ドメイン

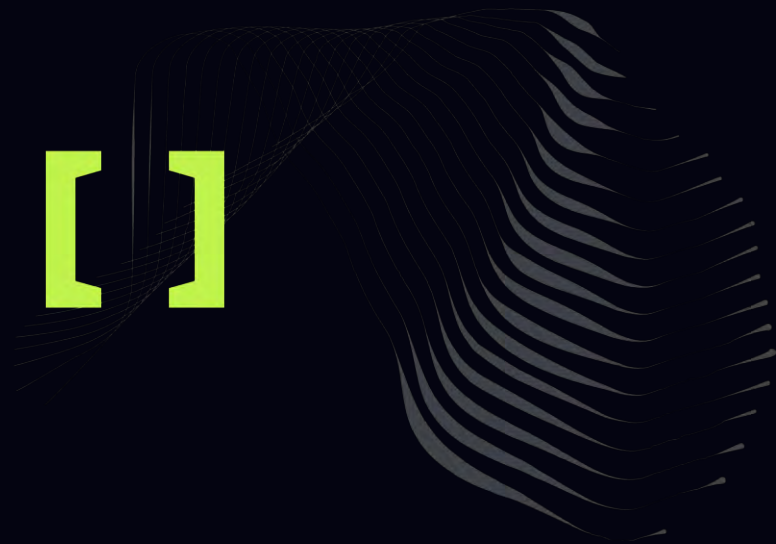
- ドメイン関数の使用例
  - ドメイン表示関数

```
SELECT DOMAIN_DISPLAY(invoice_email)
 AS email
  FROM customers;

EMAIL
-------------
---@aldi.com
---@swarovski.com
---@shell.com
```

  - ドメイン順関数

```
SELECT name FROM customers
ORDER BY DOMAIN_ORDER(contact_email);


NAME
-------------
Aldi
Shell
Swarovski
```

BOOLEANデータ型

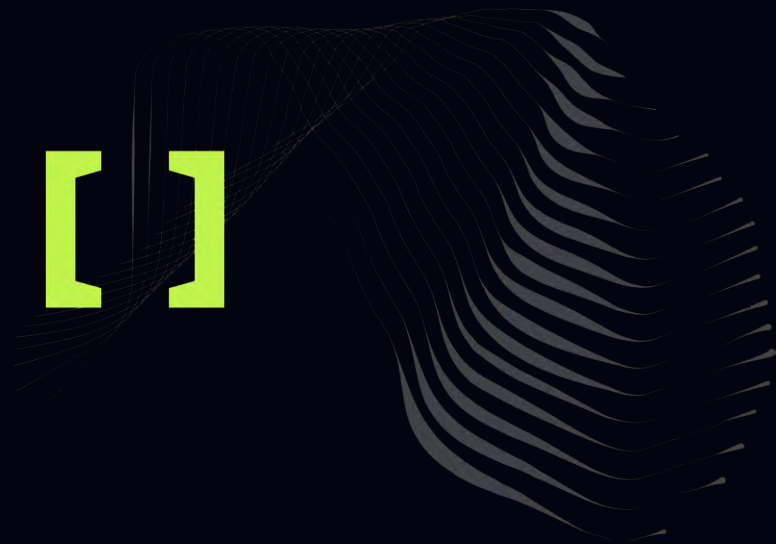


BOOLEANデータ型

- 概要
  - 真偽値のみのブーリアン型のデータを使えるようになった
- メリット
  - アプリケーションで使われるフラグ用のデータなどを、ブーリアン型のデータとして作成すれば、SQL文も短く、わかりやすいようになる

```
CREATE TABLE emails (address VARCHAR2(1000), active BOOLEAN);
INSERT INTO emails VALUES ('joe.doe@gmail.com', TRUE);
INSERT INTO emails VALUES ('jame.doe@yahoo.com', FALSE);
INSERT INTO emails VALUES ('mary.smith@yahoo.com', 'YES');
INSERT INTO emails VALUES ('jim.watson@bt.co.uk', 0);
SELECT address FROM emails WHERE active;

ADDRESS
--------------------
joe.doe@gmail.com
mary.smith@yahoo.com
```

別名と位置によるグループ集計



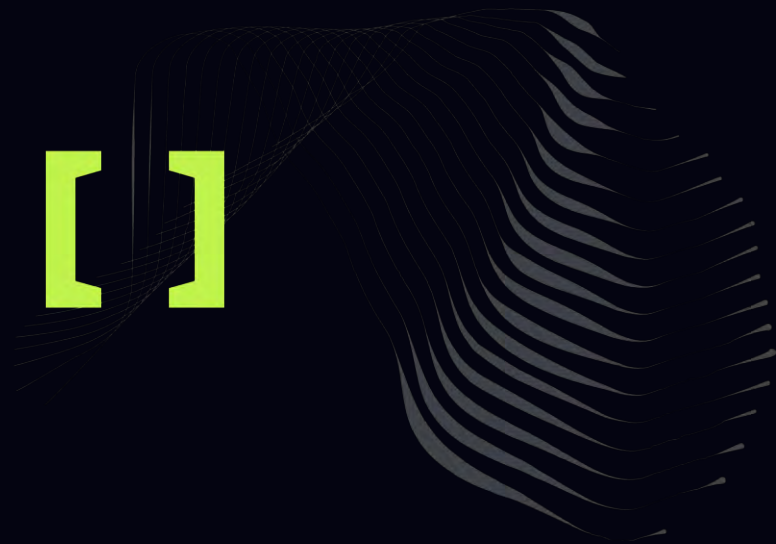
別名と位置によるグループ集計

- 概要
  - GROUP BY句で列の別名や位置の記載で、グループ集計できるようになった
- メリット
  - 今までは、GROUP BY句で正しい列名を正確に記述しなければならずSQL文が長く煩わしかったのが、簡略でできるようになった

```
SQL> SELECT extract(year FROM hiredate) AS
hired_year, COUNT(*)
FROM emp
GROUP BY extract(year FROM hiredate)
HAVING extract(year FROM hiredate) > 1985;
```

```
SQL> SELECT extract(year FROM
hiredate) AS hired_year, COUNT(*)
FROM emp
GROUP BY hired_year
HAVING hired_year > 1985;
```

INTERVAL データ型の集計



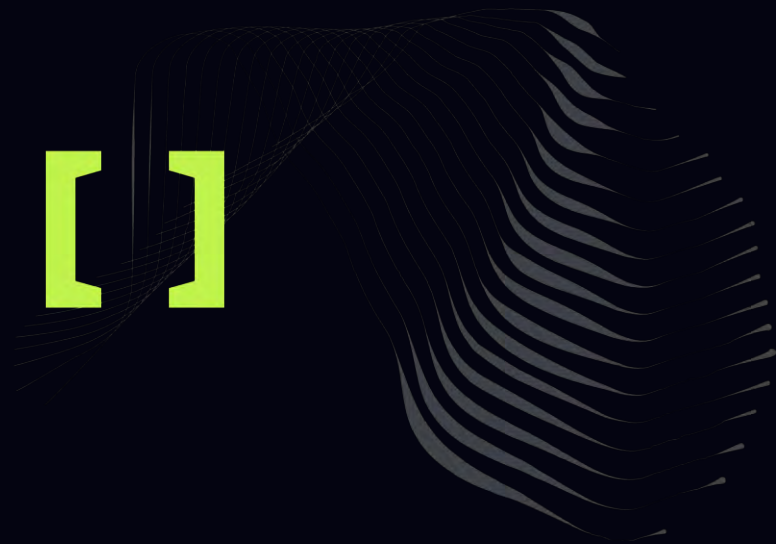
INTERVAL データ型の集計

- 概要
  - INTERVALデータ型を合計、平均、分析関数で使うことができるようになりました。
  - GROUP集計などにも使える
- メリット
  - INTERVALデータ型がより使いやすくなった

```
SQL> SELECT job_name, SUM( cpu_used )
  FROM DBA_SCHEDULER_JOB_RUN_DETAILS
  GROUP BY job_name
  HAVING SUM ( cpu_used ) > interval '5'
minute;
```

※cpu_usedはINTERVALデータ型

表の結合による簡単な更新または削除



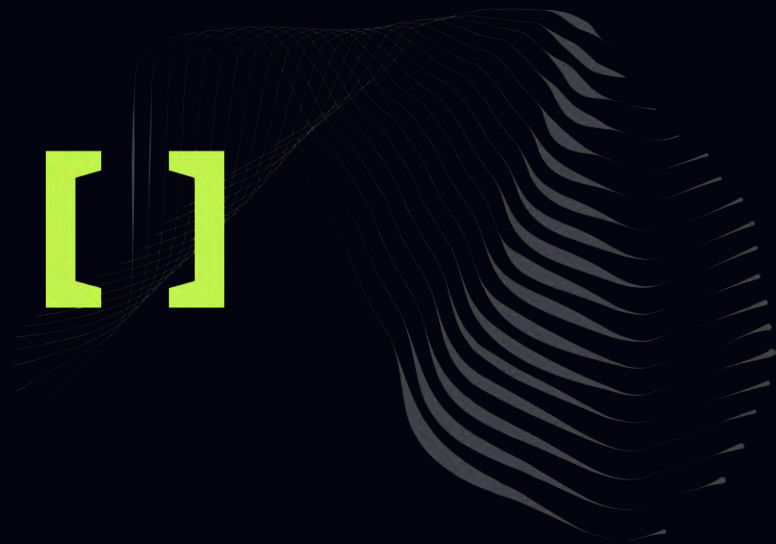
表の結合による簡単な更新または削除

- 概要
  - 他の表の値を使用してUPDATEやDELETE文を、表の結合で実行できるようになった
- メリット
  - 今までは、他の表の値を使用してUPDATEやDELETE文を実行する場合、
    サブクエリを使用して複雑なSQL文が必要だったが、簡略できるように

```
SQL> UPDATE employees e
SET e.salary = (
  SELECT j.max_salary FROM jobs j
  WHERE e.job_id = j.job_id
)
WHERE e.employee_id=100;
```

```
SQL> UPDATE employees e
SET e.salary = j.max_salary
FROM jobs j
WHERE  j.job_id = e.job_id
AND e.employee_id=100;
```

DEFAULT ON NULL for UPDATE Statements



DEFAULT ON NULL for UPDATE Statements

- 概要
  - DEFAULT ON NULL句が従来のINSERTに加え、UPDATEにも適用できるようになりました。

- メリット
  - 本機能によりアプリケーション・コードをより簡素化できます。これまではUPDATE時にデフォルト値で更新したい場合、アプリケーション・コード側でデフォルト値を保持し、コード内のUPDATE処理で明示的に指定する必要がありました。本機能によってUPDATE時もデータベース側で事前に定義したデフォルト値で更新できるため、アプリケーション・コードにおける考慮要素を削減できます。

DEFAULT ON NULL for UPDATE Statements

- 実装例

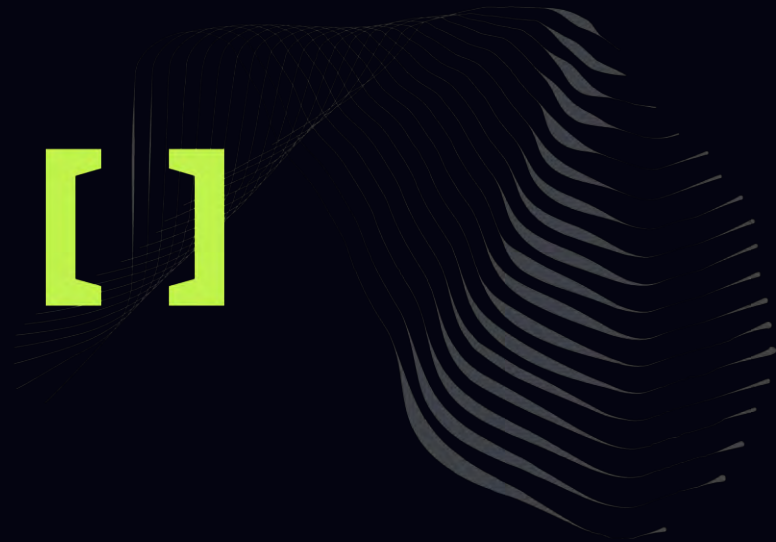
DEFAULT ON NULL for UPDATEを定義したテーブルの作成

```
-- テーブル作成
SQL> CREATE TABLE t1 (
    id          NUMBER,
    c1          VARCHAR2(15) DEFAULT ON NULL FOR
INSERT AND UPDATE 'yellow’
);

-- INSERT時のDEFAULT ON NULLの動作
SQL> INSERT INTO t1 (id) VALUES (2);
SQL> SELECT * FROM t1 ORDER BY 1;

        ID C1
---------- ---------------
         1 yellow
         2 yellow
```

DEFAULT ON NULL for UPDATEの動作確認

```
-- 一旦値を明示的に指定して更新
SQL> UPDATE t1 SET c1 = 'white';
2行が更新されました。

SQL> SELECT * FROM t1 ORDER BY 1;
        ID C1
---------- ---------------
         1 white
         2 white

-- UPDATE時のDEFAULT ON NULLの動作
SQL> UPDATE t1 SET c1 = null;
2行が更新されました。

SQL> SELECT * FROM t1 ORDER BY 1;
        ID C1
---------- ---------------
         1 yellow
         2 yellow
```

拡張された CASE の制御



拡張された CASE の制御
概要

- 概要
  - CASE文はPL/SQLで拡張され、SQL:2003 Standard [ISO03a、ISO03b]で更新されたCASE式およびCASE文の定義と一致するようになりました。

- メリット
  - ダングリング述語を使用すると、単純 CASE操作で等価以外のテストを実行できます。WHEN句に複数の選択肢があると、重複の少ないコードでCASE操作を記述できます。

- 注意
  - SQL文でのCASE式はダングリング述語は未対応です。

拡張された CASE の制御
ダングリング述語を使用した、単純 CASE文での等価以外のテスト

- 23cより前のPL/SQLでCASE文を使って等価以外の判定をする場合、単純CASE文ではなく、サーチCASE文を利用する必要がありましたが、23cではダングリング述語を用い単純CASE文で記載できるようになりました。

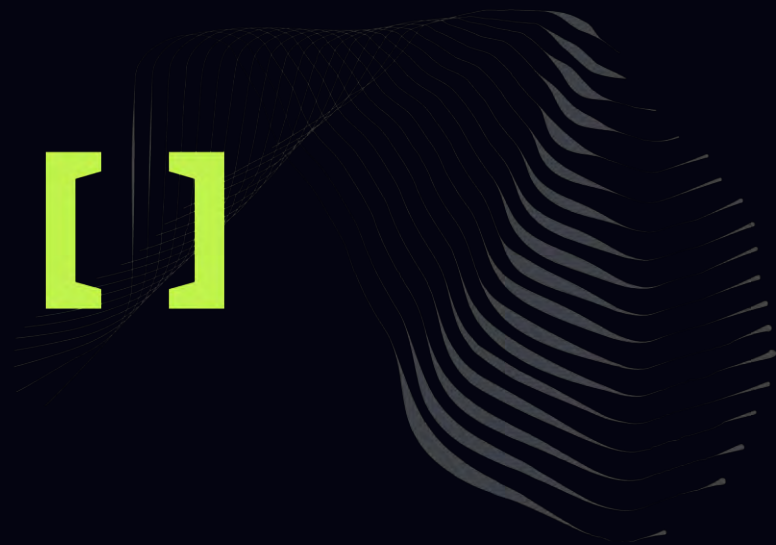
23cより前（サーチCASE文で等価以外のテスト）

```
DECLARE
  grade NUMBER;
BEGIN
  grade := 85;
  CASE
    WHEN grade > 89 THEN DBMS_OUTPUT.PUT_LINE('A');
    WHEN grade > 59 THEN DBMS_OUTPUT.PUT_LINE('B');
    ELSE DBMS_OUTPUT.PUT_LINE('C');
  END CASE;
END;
/
```

23c（ダングリング述語を用いた単純CASE文）

```
DECLARE
  grade NUMBER;
BEGIN
  grade := 85;
  CASE grade
    WHEN > 89 THEN DBMS_OUTPUT.PUT_LINE('A');
    WHEN > 59 THEN DBMS_OUTPUT.PUT_LINE('B');
    ELSE DBMS_OUTPUT.PUT_LINE('C');
  END CASE;
END;
/
```

RETURNING句拡張による更新前後のデータを両方取得



RETURNING句拡張による更新前後のデータを両方取得

- 概要
  - DMLのRETURNING句が拡張され、DML(INSERT/UPDATE/DELETE/MERGE)によって更新された列の更新前と更新後の値が取得できるようになりました。
- メリット
  - これまでRETURNING句とともに使用した際に取得できる列の値は、DMLによって更新された後の値のみでしたが更新前のデータも取得できるようになった。
  - データと更新前と更新後のデータをチェックしやすくなることにより、データの品質向上が期待できます。
  - UPDATE文の更新前後の取得例

```
SELECT salary INTO :old_salary
FROM employees
WHERE country = 'Austria'
```

```
UPDATE employees SET salary=salary*2
 WHERE country = 'Austria'
RETURNING salary
 INTO :new_salary;
```

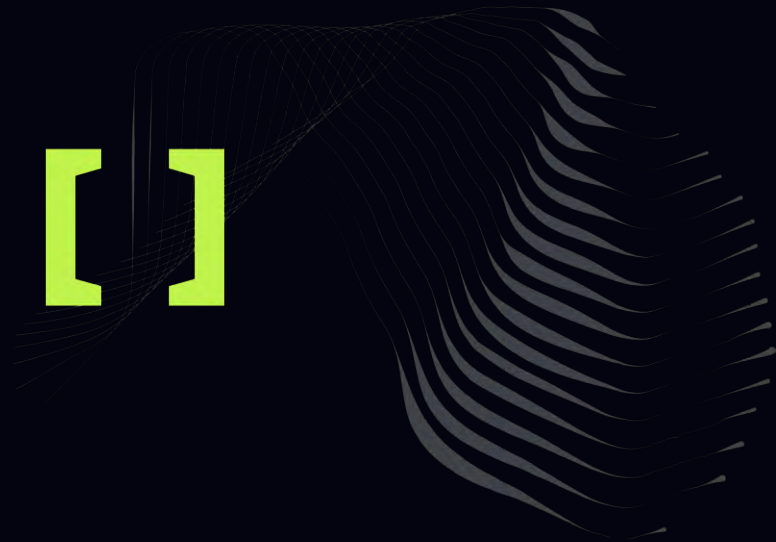
```
UPDATE employees SET salary=salary*2
 WHERE country = 'Austria'
RETURNING OLD salary, NEW salary
 INTO :old_salary, :new_salary;
```

RETURNING句拡張による更新前後のデータを両方取得

- MERGE文の使用例

```
MERGE INTO sales s USING
   (SELECT account, sale FROM ext) e
    ON (e.account=s.account)
  WHEN MATCHED THEN
    UPDATE SET s.sale=e.sale
  WHEN NOT MATCHED THEN
    INSERT (s.account, s.sale)
       VALUES (e.account, e.sale)
RETURNING s.account, e.sale
 INTO :n1, :n2;
```

SELECT Without FROM Clause



SELECT Without FROM Clause

- 概要
  - DUAL表を使う問合せに関して"FROM DUAL"句の記述が不要となりました。
- メリット
  - FROM句のないSELECTを使用することで、SQLをより簡素に表現できます。
- 実行例

```
SQL> SELECT SYSDATE;

SYSDATE
--------
23-06-07

SQL> SELECT 1+2;

       1+2
----------
         3
```


参考：FROM DUAL句を省略したSQLの内部動作

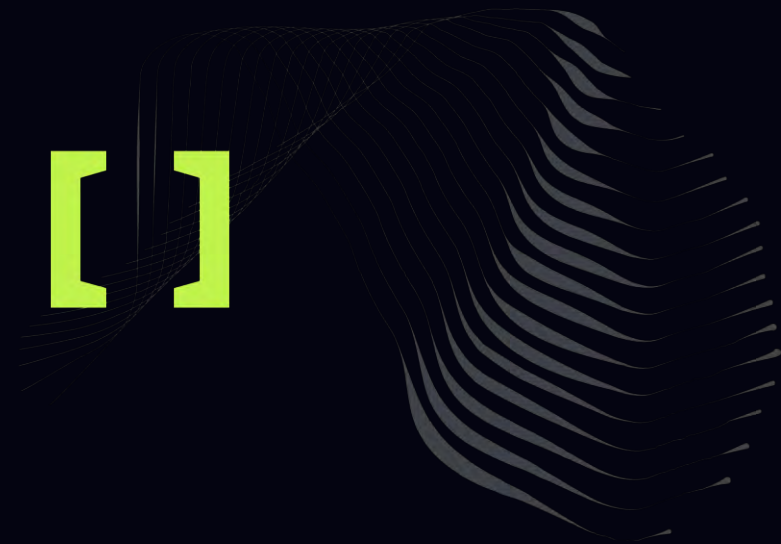
10053 SQLトレースを確認すると、FROM DUAL句の付いたSQLに問合せ変換されていることが分かります。

```
SQL> ALTER SYSTEM FLUSH SHARED_POOL;
SQL> ALTER SESSION SET EVENTS '10053 trace name context forever';
SQL> SELECT SYSDATE;
SYSDATE
--------
23-06-07
SQL> SELECT 1+2;
       1+2
----------
         3
SQL> ALTER SESSION SET EVENTS '10053 trace name context off';
SQL> SELECT VALUE FROM V$DIAG_INFO WHERE NAME = 'Default Trace File';
VALUE
--------------------------------------------------------------------------------
/opt/oracle/diag/rdbms/free/FREE/trace/FREE_ora_3339190.trc
```

上記で確認したトレースファイル内に問合せ変換の情報が出力されています。

```
...(略)...
Final query after transformations:******* UNPARSED QUERY IS *******
SELECT SYSDATE@! "SYSDATE" FROM "SYS"."DUAL" "DUAL"
...(略)...
Final query after transformations:******* UNPARSED QUERY IS *******
SELECT 3 "1+2" FROM "SYS"."DUAL" "DUAL"
```

IF [NOT] EXISTS 句 のサポート



IF [NOT] EXISTS 句 のサポート
概要

- 概要
  - DDLオブジェクトの作成、変更および削除では、IF EXISTSおよびIF NOT EXISTS句修飾子がサポートされるようになりました。
  - これにより、特定のオブジェクトが存在する場合や存在しない場合にエラーを発生させるかどうかを制御できます。

- メリット
  - IF [NOT] EXISTS句を使用すると、スクリプトおよびアプリケーションでのエラー処理を簡略化できます。

- 注意
  - IF NOT EXISTS句は、CREATE OR REPLACEコマンドでは使用できません。

IF [NOT] EXISTS 構文 のサポート
IF NOT EXISTS 構文を利用したCreate文

- IF NOT EXISTS句をCREATEコマンドとともに使用すると、特定のオブジェクトが存在する場合にエラーを抑制できます。オブジェクトがすでに存在する場合、IF NOT EXISTSを指定したCREATEコマンドはエラーを返しません。オブジェクトが存在しない場合は、CREATEコマンドによって新しいオブジェクトが作成されます。

■実行例　Create Tableで試す。

同名テーブルが無い場合実行

```
SQL>select * from t1;

行1でエラーが発生しました。:
ORA-00942: 表またはﾋﾞｭｰが存在しません。
SQL>
SQL>CREATE TABLE IF NOT EXIST t1 (c1 number);

表が作成されました。
SQL>
```

カラム構成が同一の
同名テーブルがある場合

```
SQL>select * from t1;

       C1
---------
        1
SQL>
SQL>CREATE TABLE IF NOT EXIST t1 (c1 number);

表が作成されました。
SQL>
```

テーブルが既に存在するが、
Create文からエラーは返らない

カラム構成が異なる同名テーブルがある場合

```
SQL>select * from t1;

       C1
---------
        1
SQL>
SQL>CREATE TABLE IF NOT EXIST t1 (c1 number,c2 varchar(20));

表が作成されました。
SQL>select * from t1;

       C1
---------
        1
SQL>
```

「表が作成されました」と表示するが、
カラム構成は変わらない。

IF [NOT] EXISTS 構文 のサポート
IF EXISTS 構文を利用したDrop文、Alter文

- IF EXISTS句をALTERコマンド又はDROPコマンドとともに使用すると、特定のオブジェクトが存在しない場合にエラーを抑制できます。オブジェクトが存在する場合、IF EXISTS句を指定したALTERコマンド又はDROPコマンドは正常に実行されます。

■実行例　Drop Tableの例

・同名テーブルがある場合

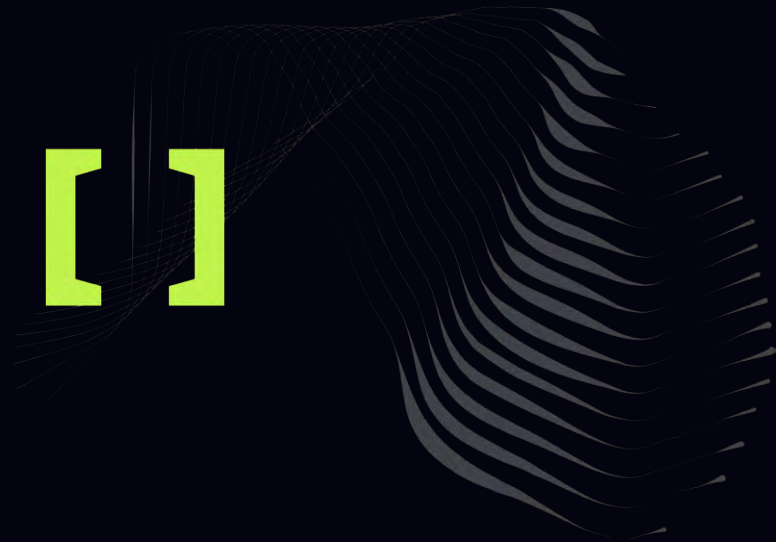
```
SQL>select * from t1;

       C1
---------
        1
SQL>
SQL>DROP TABLE IF EXIST t1;

表が削除されました。
SQL>
```

・同名テーブルが無い場合

```
SQL>select * from t1;

行1でエラーが発生しました。:
ORA-00942: 表またはﾋﾞｭｰが存在しません。
SQL>
SQL>DROP TABLE IF EXIST t1;

表が削除されました。
SQL>
```

テーブルが無いが、Drop文からエラーは返らない

表値コンストラクタ

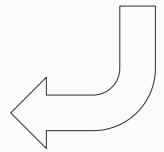


表値コンストラクタ(ISO SQL標準)

- 概要
  - VALUES 句によって複数行をSQLで記述になりました。
- メリット
  - 従来はVALUES 句に複数行記述できなかったので、複数行INSERT文では複雑なSQLを記述する必要があったが、この機能により簡単に記述できるようになった。
  - 一時的に使いたい表があるときに、実体の表を作らなくても、SQL文中にVALUES 句による一時表を記述することができるようになり、開発効率の向上が期待できます。

```
INSERT ALL
INSERT INTO bookings VALUES (12113, 'Vienna', '2022-09-21')
INSERT INTO bookings VALUES (62361, 'San Francisco', '2022-10-12')
INSERT INTO bookings VALUES (08172, 'Berlin', '2022-12-15’)
SELECT * FROM DUAL;
```

```
INSERT INTO bookings
 VALUES (12113, 'Vienna', '2022-09-21'),
        (62361, 'San Francisco', '2022-10-12'),
        (08172, 'Berlin', '2022-12-15');
```



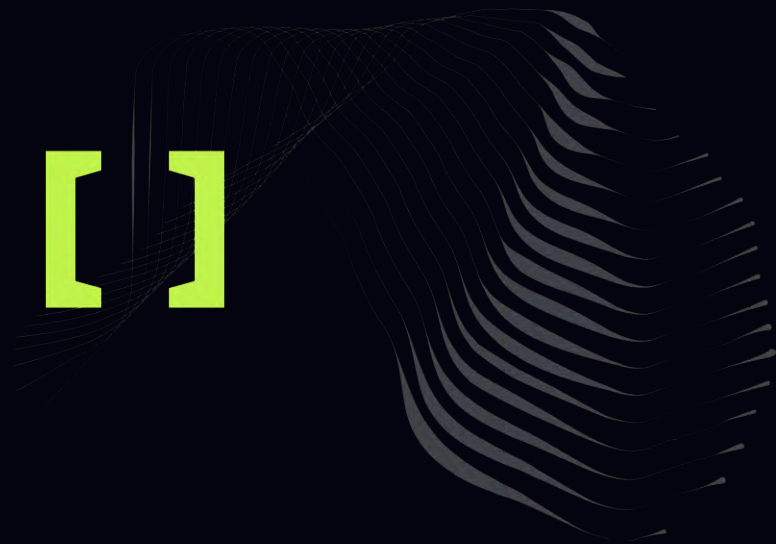
表値コンストラクタ(ISO SQL標準)

- DMLでの表バリューコンストラクターによる一時的な表の使用例

```
CREATE TABEL t1(
 employee_id NUMBER(1,0),
 first       VARCHAR2(10));

INSERT INTO t1 VALUES (1,'SCOTT’);
INSERT INTO t1 VALUES (2,'SMITH’);
INSERT INTO t1 VALUES (3,'JOHN’);

SELECT FROM t1;

EMPLOYEE_ID FIRST
----------- -----
          1 SCOTT
          2 SMITH
          3 JOHN
```

```
SELECT *
FROM (VALUES (1,'SCOTT'),
             (2,'SMITH'),
             (3,'JOHN')
) t1 (employee_id, first);

EMPLOYEE_ID FIRST
----------- -----
          1 SCOTT
          2 SMITH
          3 JOHN
```

アプリケーション開発者用の DB_DEVELOPER_ROLE



アプリケーション開発者用のDB_DEVELOPER_ROLE

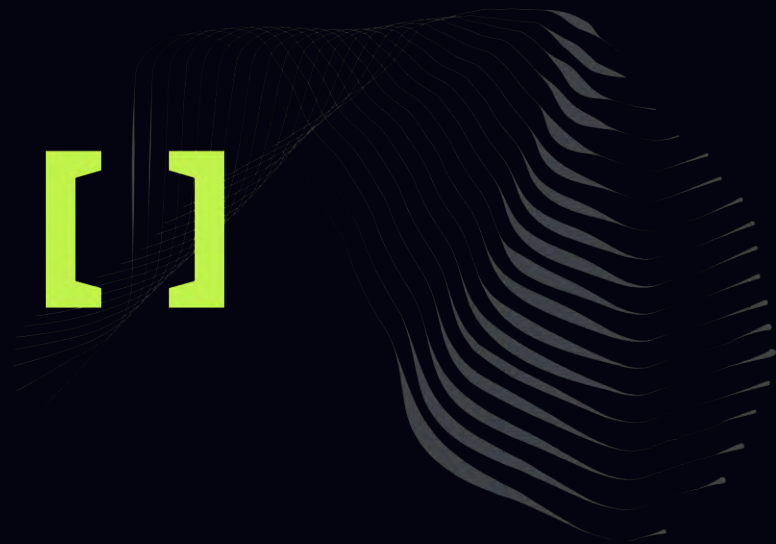
- 概要
  - 開発者権限のロールができた
- メリット
  - データモデルの構築に必要なシステム権限とアプリケーションの監視とデバッグに必要なオブジェクト権限を一括で権限付与できるようなった

```
SQL> GRANT  DB_DEVELOPER_ROLE TO   dev_user;
SQL> REVOKE DB_DEVELOPER_ROLE FROM dev_user;
```

参考： DB_DEVELOPER_ROLEに含まれる権限

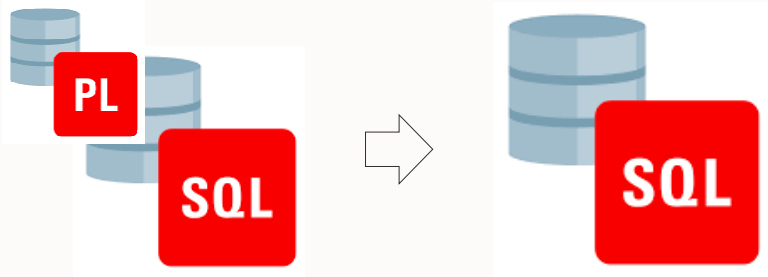
- ADMINISTER SQL TUNING SET
- CREATE ANALYTIC VIEW
- CREATE ATTRIBUTE DIMENSION
- CREATE CUBE
- CREATE CUBE BUILD PROCESS
- CREATE CUBE DIMENSION
- CREATE DIMENSION
- CREATE DOMAIN
- CREATE HIERARCHY
- CREATE JOB
- CREATE MATERIALIZED VIEW
- CREATE MINING MODEL
- CREATE MLE
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE SESSION
- CREATE SYNONYM
- CREATE TABLE
- CREATE TRIGGER
- CREATE TYPE
- CREATE VIEW
- DEBUG CONNECT SESSION
- EXECUTE DYNAMIC MLE
- EXECUTE ON JAVASCRIPT
- FORCE TRANSACTION
- ON COMMIT REFRESH

PL/SQLからSQLへの自動変換



PL/SQLからSQLへの自動変換

- 概要
  - SQL 文内の PL/SQL 関数は、可能な限り自動的に SQL 式に変換 (トランスパイル) されます。
- メリット
  - PL/SQL 関数を SQL 文に変換されると、全体の実行時間を短縮できます。
  - SQL トランスパイラは、透過的に、可能な限り自動的にSQL内のPL/SQL関数をSQL式に 変換(トランスパイル)します。
  - 以前は、SQL式では、PL/SQL 関数を呼び出すときに、PL/SQL ランタイムを呼び出す必要があるため、オーバーヘッドが発生して（遅くなって）いました。
  - PL/SQL関数を意味的に同等のSQL式に自動的に変換します。
  - トランスパイラがPL/SQLファンクションをSQLに変換できない場合、ファンクションの実行はPL/SQLランタイムにフォールバックします。すべての PL/SQL 構造がトランスパイラでサポートされているわけではありません。



PL/SQLからSQLへの自動変換

- 利用イメージ
  - 使用例として、 employees 表をPL/SQL 関数を使って文字列置換するSQLを実行

PL/SQL自動変換せず

```
SELECT employee_id, first_name,last_name
FROM employees
WHERE get_month_abbreviation ( hire_date )
= 'MAY';
...
Elapsed: 00:00:00.14
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("GET_MONTH_ABBREVIATION"("HIRE_DATE")='MAY')
```

PL/SQL自動変換

```
SELECT employee_id, first_name,last_name
FROM employees
WHERE get_month_abbreviation ( hire_date )
= 'MAY';
...
Elapsed: 00:00:00.02
```

PL/SQL 関数呼び出しの
オーバーヘッドがないので
処理が速い

```
Predicate Information (identified by operation id):
---------------------------------------------------
 1 - filter(TO_CHAR(INTERNAL_FUNCTION("HIRE_DATE"),'MON','NLS_DATE_LAN
            GUAGE=English')='MAY')
```

SQL実行計画の述語情報では、
PL/SQL 関数が、
SQLに変換（トランスパイル）されている

get_month_abbreviation関数定義

```
create function get_month_abbreviation ( date_value date ) return
varchar2 is begin return to_char ( date_value, 'MON',
'NLS_DATE_LANGUAGE=English' );
end; /
```

PL/SQLからSQLへの自動変換

- SQL トランスパイラの有効化または無効化
  - SQL トランスパイラはデフォルトでは無効になっています。ALTER SYSTEM コマンドを使用して有効にできます。
  - パラメータはシステム レベルまたはセッション レベルで変更できます。

SQL トランスパイラの有効化

```
ALTER SESSION set sql_transpiler = ON;
```

参考：トランスパイルの対象となるPL/SQL構造

SQL 自動変換 では、次の PL/SQL 言語要素がサポートされています。

- 基本的な SQL スカラー型: CHARACTER、DATETIME、および NUMBER
- 文字列型 (CHAR、VARCHAR、VARCHAR2、NCHAR など)
- 数値型 (NUMBER、BINARY DOUBLE など)
- 日付タイプ (DATE、INTERVAL、および TIMESTAMP)
- ローカル変数 (宣言時のオプションの初期化あり) と定数
- オプションの (単純な) デフォルト値を持つパラメータ
- 変数代入ステートメント
- 同等の SQL 式に変換できる式
- IF-THEN-ELSE ステートメント
- RETURN ステートメント
- BOOLEAN 型の式とローカル変数

ありがとうございました