New PL/SQL Language Features After 12c (Up to 19c)

Oracle 19c is intentionally conservative — it's a long-term support release — so the list is short but meaningful.

1. SQL Macros (Preview in 19c)

This is the *only* major new PL/SQL syntax feature introduced after 12c.

New keyword: `SQL_MACRO`

Used in function declarations:

```
CREATE FUNCTION f(p NUMBER)
RETURN VARCHAR2 SQL_MACRO IS
BEGIN
    RETURN 'SELECT * FROM emp WHERE deptno = p';
END;
```

Two types exist (in 21c+):

- `SQL_MACRO(TABLE)`
- `SQL_MACRO(SCALAR)`

In 19c, it's a **preview**, but fully functional.

2. Improved PL/SQL Inlining Controls

No new keywords, but 19c improved the optimizer's use of:

```
PRAGMA INLINE(subprogram, 'YES');
```

The pragma existed earlier, but 19c made it more effective.

□ 3. UTL_CALL_STACK Enhancements

No new keywords, but new subprograms and better stack reporting.

PL/SQL-Adjacent Features Added After 12c That Affect PL/SQL

These aren't new PL/SQL keywords, but they significantly change how PL/SQL behaves in production.

1. SQL Quarantine (19c)

Prevents PL/SQL jobs from repeatedly executing crashing SQL.

2. Automatic Indexing (19c)

PL/SQL code benefits from better execution plans without code changes.

3. Real-Time Statistics (19c)

Improves dynamic SQL and PL/SQL query performance.

4. Automatic SQL Plan Management Enhancements

Better baseline evolution and plan stability.

5. Native Compilation Improvements

Faster PL/SQL native code generation.

What Did *Not* Change (No New Keywords)

Oracle did **not** add new PL/SQL keywords in 18c or 19c besides `SQL_MACRO`.

No new:

- `DECLARE`-level keywords
- Loop constructs
- Exception keywords
- Cursor syntax
- Collection syntax
- Object type syntax

PL/SQL has been extremely stable since 12c because Oracle considers it a mature, complete language.

Bottom Line

Between 12c and 19c:

- **Only one major new PL/SQL keyword was added:** `SQL_MACRO`
- **Most improvements were runtime, optimizer, or operational enhancements**
- **PL/SQL syntax remained intentionally stable**

SQL Macros

SQL Macros in Oracle 19c *feel* like dynamic SQL at first glance, but they are **not** dynamic SQL. They solve a similar problem—generating SQL at runtime—but they do it in a fundamentally different, safer, and more optimizer-friendly way.

Let's break this down cleanly so you can benchmark the differences.

Short Answer

SQL Macros are *not* dynamic SQL. They are **compile-time SQL text generators** that expand into regular SQL before execution.

Dynamic SQL = SQL text built and executed at runtime SQL Macros = SQL text generated at parse time, then treated as normal SQL

What SQL Macros Actually Do

A SQL Macro is a PL/SQL function that returns a **SQL fragment**, not a value.

Example:

```
CREATE FUNCTION filter_dept(p_deptno NUMBER)
RETURN VARCHAR2 SQL_MACRO IS
BEGIN
    RETURN 'SELECT * FROM emp WHERE deptno = p_deptno';
END;
```

When you call it:

```
SELECT * FROM filter_dept(10);
```

Oracle **expands the macro into SQL text** before optimization.

This is similar to a C preprocessor macro or an inline SQL template.

How SQL Macros Differ from Dynamic SQL

1. Dynamic SQL is executed at runtime

You build a string and run it:

```
EXECUTE IMMEDIATE 'SELECT ...';
```

- Parsed at runtime
- Harder for the optimizer to reuse
- Requires bind variables to avoid SQL injection
- More overhead

2. SQL Macros are expanded at parse time

Oracle rewrites the SQL before execution:

- No runtime string building
- No `EXECUTE IMMEDIATE`
- Fully visible to the optimizer
- Bind variables handled automatically
- Safer and faster

Why SQL Macros Are Not Dynamic SQL

Feature	SQL Macros	Dynamic SQL
When SQL text is created	Parse time	Runtime
Uses `EXECUTE IMMEDIATE`	□	✔□
Optimizer visibility	Full	Limited
SQL injection risk	Very low	High if careless
Bind variable support	Automatic	Manual
Performance	High	Medium/low
Purpose	Reusable SQL templates	Runtime SQL construction

SQL Macros behave more like **inline views**, **WITH clause generators**, or **parameterized SQL templates**, not dynamic SQL engines.

Let’s make this crystal clear with a **side-by-side comparison** of the *same logic* implemented three different ways:

- **Static SQL**
- **Dynamic SQL**
- **SQL Macro (19c preview)**

This will show you exactly how SQL Macros differ from dynamic SQL — and why they’re not the same thing.

1. Static SQL Version

Classic, simple, fully optimized.

```
SELECT *
FROM emp
WHERE deptno = 10;
```

- Hard-coded
- Fully visible to the optimizer
- No flexibility

2. Dynamic SQL Version

SQL text is built and executed at **runtime**.

```
DECLARE
    l_sql   VARCHAR2(200);
    l_dept  NUMBER := 10;
BEGIN
    l_sql := 'SELECT * FROM emp WHERE deptno = :x';

    EXECUTE IMMEDIATE l_sql
    INTO ...
    USING l_dept;
END;
```

Characteristics

- SQL text is a **string**
- Parsed at runtime

- Requires `EXECUTE IMMEDIATE`
- Must manage bind variables manually
- SQL injection risk if careless
- Optimizer sees only the final SQL, not the PL/SQL logic that built it

3. SQL Macro Version (Oracle 19c Preview)

SQL text is generated at **parse time**, not runtime.

Define the macro

```
CREATE FUNCTION filter_dept(p_deptno NUMBER)
RETURN VARCHAR2 SQL_MACRO IS
BEGIN
    RETURN 'SELECT * FROM emp WHERE deptno = p_deptno';
END;
```

Use it like a table function

```
SELECT *
FROM filter_dept(10);
```

What actually happens internally

Oracle expands it into:

```
SELECT *
FROM emp
WHERE deptno = 10;
```

Characteristics

- No `EXECUTE IMMEDIATE`
- No runtime string building
- Fully visible to the optimizer
- Bind variables handled automatically
- Safe, fast, and composable
- Behaves like a reusable SQL template

Benchmark-Style Comparison

Feature	Static SQL	Dynamic SQL	SQL Macro
When SQL text is created	Compile time	Runtime	Parse time
Uses `EXECUTE IMMEDIATE`	□	✔□	□
Optimizer visibility	Full	Partial	Full
SQL injection risk	None	High if careless	Very low
Bind variable handling	Automatic	Manual	Automatic
Flexibility	Low	Very high	High
Performance	Excellent	Medium	Excellent

SQL Macros give you **dynamic-SQL-like flexibility** but with **static-SQL-like performance and safety**.

Bottom Line

SQL Macros are **not** dynamic SQL. They are **compile-time SQL generators** that let you write reusable, parameterized SQL without ever touching `EXECUTE IMMEDIATE`.

If you want, I can also show:

- A **scalar SQL macro** example
- A **table SQL macro** that builds a WHERE clause
- A **macro that generates JOINs dynamically**
- A **macro vs dynamic SQL performance comparison**

PRAGMA INLINE(subprogram, 'YES');

`PRAGMA INLINE` is Oracle's way of telling the PL/SQL compiler **"inline this subprogram's code directly into the caller instead of performing a normal function call."**

It's a performance-tuning directive — not a requirement — and it can make small, frequently-called functions dramatically faster by eliminating call overhead.

Let's break it down cleanly.

What `PRAGMA INLINE` Actually Does

When you write:

```
PRAGMA INLINE(my_func, 'YES');
```

you're telling Oracle:

- Don't call `my_func` normally
- Instead, **copy the function's logic directly into the calling code**
- Let the optimizer treat it like native code

This avoids:

- Stack frame setup
- Parameter passing
- Return value handling

For tiny functions, this overhead is often larger than the function itself.

Where You Put It

You place the pragma **inside the calling block**, not inside the function being inlined.

Example:

```
DECLARE
    FUNCTION add_tax(p NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN p * 1.07;
    END;
BEGIN
    PRAGMA INLINE(add_tax, 'YES');

    DBMS_OUTPUT.PUT_LINE(add_tax(100));
END;
```

Oracle will inline `add_tax` into the anonymous block.

What Oracle Does Under the Hood

Without inlining:

Code
```
CALL add_tax
  → push parameters
  → execute function
  → return value
```

With inlining:

Code
```
p * 1.07
```

The function's body is substituted directly into the caller.

When You Should Use It

✔ Good candidates

- Very small functions
- Functions called inside tight loops
- Functions used in performance-critical code paths
- Utility functions that do simple arithmetic or checks

Poor candidates

- Large functions
- Functions with complex logic
- Functions that are rarely called
- Functions that allocate memory or use exceptions heavily

Inlining a big function can **slow things down** because it bloats the calling code.

Does Oracle Inline Automatically?

Yes — starting in 12c and improved in 19c.

Oracle's optimizer will inline small functions **even without the pragma**.

`PRAGMA INLINE` is a way to *force* the optimizer's hand when you know better than the heuristics.

Can You Turn Inlining Off?

Yes:

```
PRAGMA INLINE(my_func, 'NO');
```

Useful when debugging or when inlining causes unexpected plan changes.

The overhead is **essentially the same each time**, yes — but with one important nuance that's worth understanding if you're benchmarking PL/SQL performance.

Let's break it down cleanly.

What Happens When You Call a Function Normally (No Inlining)

Every function call — no matter how small — triggers the same sequence:

1. Allocate a stack frame
2. Pass parameters
3. Transfer control to the function
4. Execute the function body

5. Return the value
6. Tear down the stack frame

This cost is **fixed per call**.

So if you call the function 10,000 times, you pay that overhead 10,000 times.

Does Oracle Reduce This Overhead Automatically?

Oracle does optimize PL/SQL execution internally, but:

- It **does not eliminate** the call overhead
- It **does not inline** the function unless the optimizer decides to (12c+), and even then only for very small, simple functions

If the optimizer chooses *not* to inline, the overhead remains identical for each call.

What Inlining Changes

Inlining removes the call entirely.

Instead of:

Code
```
CALL my_func → do work → return
```

Oracle substitutes the function body directly into the caller:

Code
```
-- inlined
<function logic here>
```

So the overhead becomes **zero** for that call.

So Is the Overhead the Same Every Time Without Inlining?

Yes — the overhead is:

- **Fixed**
- **Predictable**

- **Paid on every call**
- **Independent of the function's logic**

The only thing that changes the cost is:

- Inlining (manual or automatic)
- Native compilation (reduces cost but doesn't eliminate it)
- Function complexity (affects execution time, not call overhead)