31

Developing Applications with Sessionless Transactions

In this chapter, learn about Sessionless transactions and how to implement them to manage transactional workloads in your applications.

31.1 Introduction to Managing Transactions and Sessionless Transactions

Using database resources efficiently is a challenge when you develop applications that use transactional workloads. Typically, managing a transaction requires the connection and session resources to be tied to the transaction throughout the transaction's lifecycle, and therefore the session or connection can be released only after the transaction has ended. Ideally, applications with think time in the application logic would want to grab a session from the database, submit a unit of work, release the session, attend to the application logic, grab another session, resume the transaction, submit another unit of work, and so on, until they commit the transaction in the end. As is evident, the latter approach enables applications to use resources more efficiently because the transaction is not tied to a session or connection, which can then be used by another client. The Sessionless Transactions feature is based on the latter approach. In Sessionless Transactions, after you start a transaction, you have the flexibility to suspend and resume the transaction across sessions during the transaction's lifecycle. Moreover, when you use Sessionless Transactions, the database internally coordinates the two-phase commit (2PC) protocol, without the need for any application-side logic. Also, Sessionless Transactions can be used in a single-instance configuration or multi-instance Real Application Clusters (RAC) configuration.

> **Note:**
>
> Sessionless Transactions are applicable only to a single database.

To get a better perspective of how important the Sessionless Transactions feature can be to your application, take a look at how an application would manage transactions with and without Sessionless Transactions.

Managing Transactions without Sessionless Transactions

Without Sessionless Transactions, to manage transactions in your application, you would need to choose between one of the following two approaches:

- Start a transaction on a session in an instance and continue to associate the transaction with the same session and instance until the transaction is finalized.

  Downside: When a transaction is associated with a session and a connection, the session and connection in the instance are held up for the time the transaction is active, even when the resource utilization is very low. With the addition of more and more connections, you risk the chances of the database running out of connections.

- Coordinate the eXtended Architecture (XA) protocol through a transaction manager. A transaction manager is required to coordinate an XA transaction across branches on different database instances and sessions, and also coordinate the 2PC process to commit or roll back the XA transaction.

  Downside: Although the connections are not held up and you can multiplex the connection across clients, you still need the transaction manager infrastructure to coordinate an XA transaction. The transaction manager must work in the middle tier and remember the status of all the branches. Any network failure can result in in-doubt transactions, which can only be resolved using a recovery process. Moreover, an additional mid-tier component like a transaction manager introduces yet another stateful service that has to be managed, and kept highly available - a huge management overhead. If the service fails, then locks are held in the database leading to cascading failures, hangs, and database outages.

There are solutions such as Oracle XA Transactions, which is based on the XA open standard, that allow you to multiplex transactions between connections and sessions. Oracle XA guarantees that in all the participating databases the transactional updates are either committed or fully rolled back. However, to use Oracle XA, you must deploy a transaction manager, such as MicroTX, Tuxedo, or WebLogic as a middle-tier component that resides outside Oracle Database.

There are other disadvantages of using XA Transactions, such as:

- XA transactions can result in high transaction latency owing to slow resource managers or unreliable network between a transaction manager and resource managers.
- XA transactions are only as reliable as the external transaction managers, even though the individual resource managers are highly reliable and available like Oracle Database.
- Transaction managers must create and manage branches across database instances or sessions, thereby requiring more memory space to keep the information on branches and more time to find the correct branch when resuming a transaction on a certain session and instance. Therefore, transaction managers must have the knowledge of individual instances and run the 2PC operation between the branches created across database instances. Any failure can lead to the transaction going in-doubt and blocking subsequent work.

> **See Also:**
>
> Developing Applications with Oracle XA for information about Oracle XA

On the contrary, when you use Sessionless Transactions, you do not need to use a transaction manager when communicating with a single database (single or multi-instance). The database does all the work of coordinating the transaction for you.

Managing Transactions with Sessionless Transactions

A Sessionless transaction breaks the coupling between the transaction and the session. Once you start a transaction, it does not need to be tied to a session or connection. You can free the session or connection, allowing it to be used by another client. Additionally, you do not need a transaction manager to coordinate the XA protocol because the 2PC process and the error recovery process are handled automatically by the database. Therefore, there is no risk of in-doubt transactions and no need for any recovery mechanism.

To sum up, the key differentiators of Sessionless Transactions are:

- Efficient use of sessions and connections because a transaction is not sticky on a session or connection.
- Reduced commit latency because the transaction finalization happens internally and fewer client-server round trips are needed.
- Improved transactional throughput because the recovery mechanism is handled internally, thereby eliminating the possibility of locks holding resources indefinitely.
- Transactions never go in-doubt.

31.2 Sessionless Transactions Overview

Sessionless Transactions is a feature for managing transactions efficiently in a database application.

The Sessionless Transactions feature enables users to start a transaction on a database session by providing a unique transaction identifier, submit a unit of work, suspend the transaction, and continue the same transaction on another session using the same transaction identifier. In the end, the same transaction can be committed from yet another session.

The Sessionless Transactions feature provides applications with a native mechanism to commit or roll back a transaction without employing an external transaction manager to coordinate the XA protocol. The transaction commit is coordinated internally by the database to ensure data integrity across multiple database instances or sessions.

You can use Sessionless Transactions on a RAC or a non-RAC deployment. If it is a RAC deployment, each unit of work could use sessions on different database instances. On a non-RAC deployment, you have a single instance, and each unit of work could use different sessions on the same database instance.

The following is an example illustrating the workflow on a RAC deployment.

1. Acquire a session on Instance 1.
2. Set a unique identifier for the transaction.
3. Start a new Sessionless transaction.
4. Submit the unit of work – DML or SQL statements.
5. Suspend the Sessionless transaction.
6. Release the session.
7. Acquire a session on Instance 2.
8. Set the same unique identifier for the transaction.

9. Resume the Sessionless transaction.
10. Submit the unit of work – DML or SQL statements.
11. Suspend the Sessionless transaction.
12. Release the session.
13. Acquire a session on Instance 3.
14. Set the same unique identifier for the transaction.
15. Resume the Sessionless transaction.
16. `COMMIT`.
17. Release the session.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information about Oracle RAC

To start, suspend, and resume a Sessionless transaction, you can use server-side (PL/SQL) or client-side APIs. The APIs enable you to:

- Set a user-provided or auto-generated unique transaction identifier on a transaction.
- Start a new transaction with the given transaction identifier.
- Suspend the transaction identified by the transaction identifier.
- Resume the transaction identified by the transaction identifier.

These APIs are discussed in detail in the subsequent sections of this document.

31.3 Sessionless Transactions Capabilities

Sessionless Transactions provides the following capabilities.

Multiplexing Transactions between Sessions and Connections

Sessionless Transactions provides native mechanisms to start, suspend, and resume a transaction, and in the end, commit or roll back the transaction, as shown in the following Figure 31-1 and Figure 31-2. Sessionless Transactions gives you the flexibility to eliminate the need for the XA protocol and an external transaction manager. Therefore, you can obtain the key advantages of reduced latency and Oracle Database's high availability.

Figure 31-1 Multiplexing Transactions across Sessions on a RAC Deployment

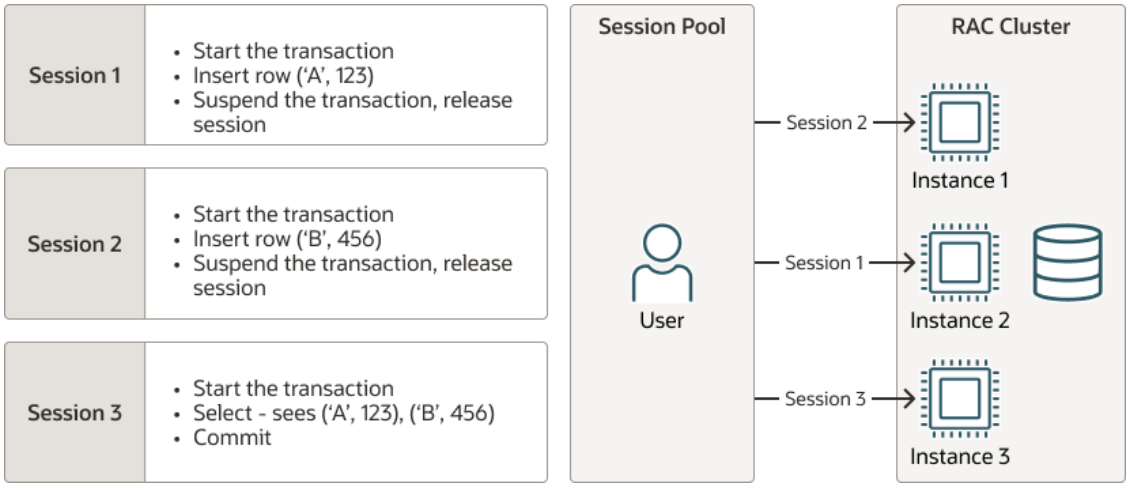
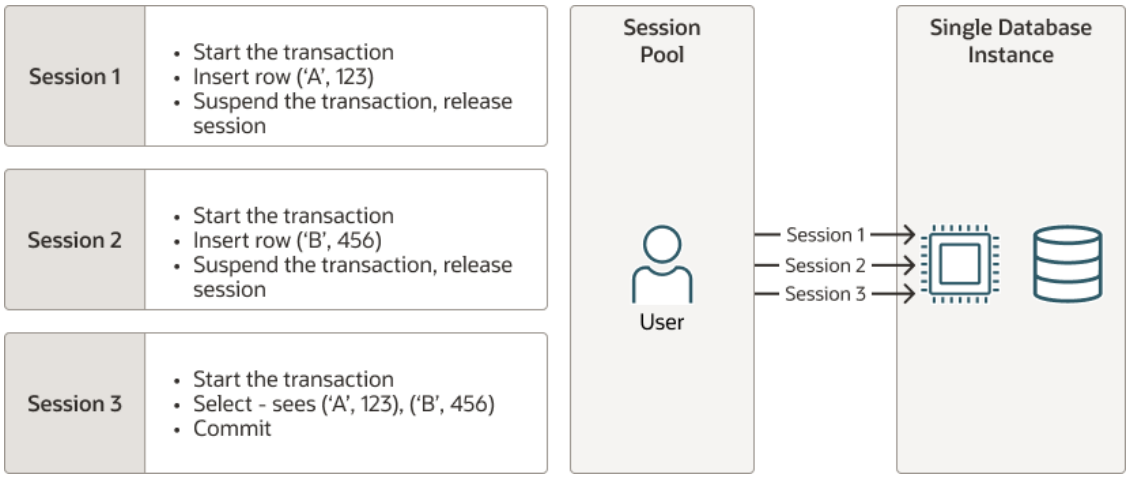


Figure 31-2 Multiplexing Transactions across Sessions on a Single-instance Deployment



Eliminating External Transaction Managers when Using a Single Database

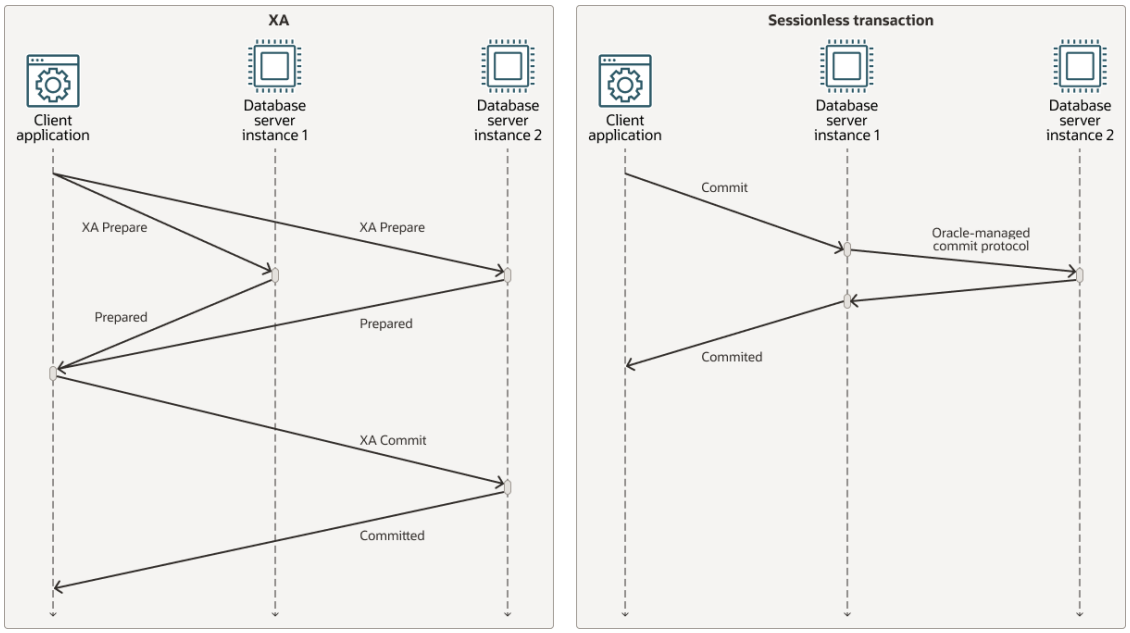
Enterprise applications enlist transaction managers to guarantee completion of a series of operations in transactional resource managers such as Oracle Database. Sessionless Transactions employs 2PC internally to commit or roll back a Sessionless transaction, thereby eliminating the need for an external transaction manager when communicating with the same database. However, a transaction manager is still needed when coordinating a transaction across different databases.

Improving Performance

The Sessionless Transactions feature implements the commit protocol inside Oracle Database, allowing the application or client to issue a simple commit command to finalize a transaction. In fact, the commit command could be issued from any connection or session having the right transaction identifier, thereby emphasizing the feature's connection and session multiplexing capability. The reduced commit latency of Sessionless transactions is in sharp contrast to the XA protocol, where the application or the transaction manager must drive the commit protocol,

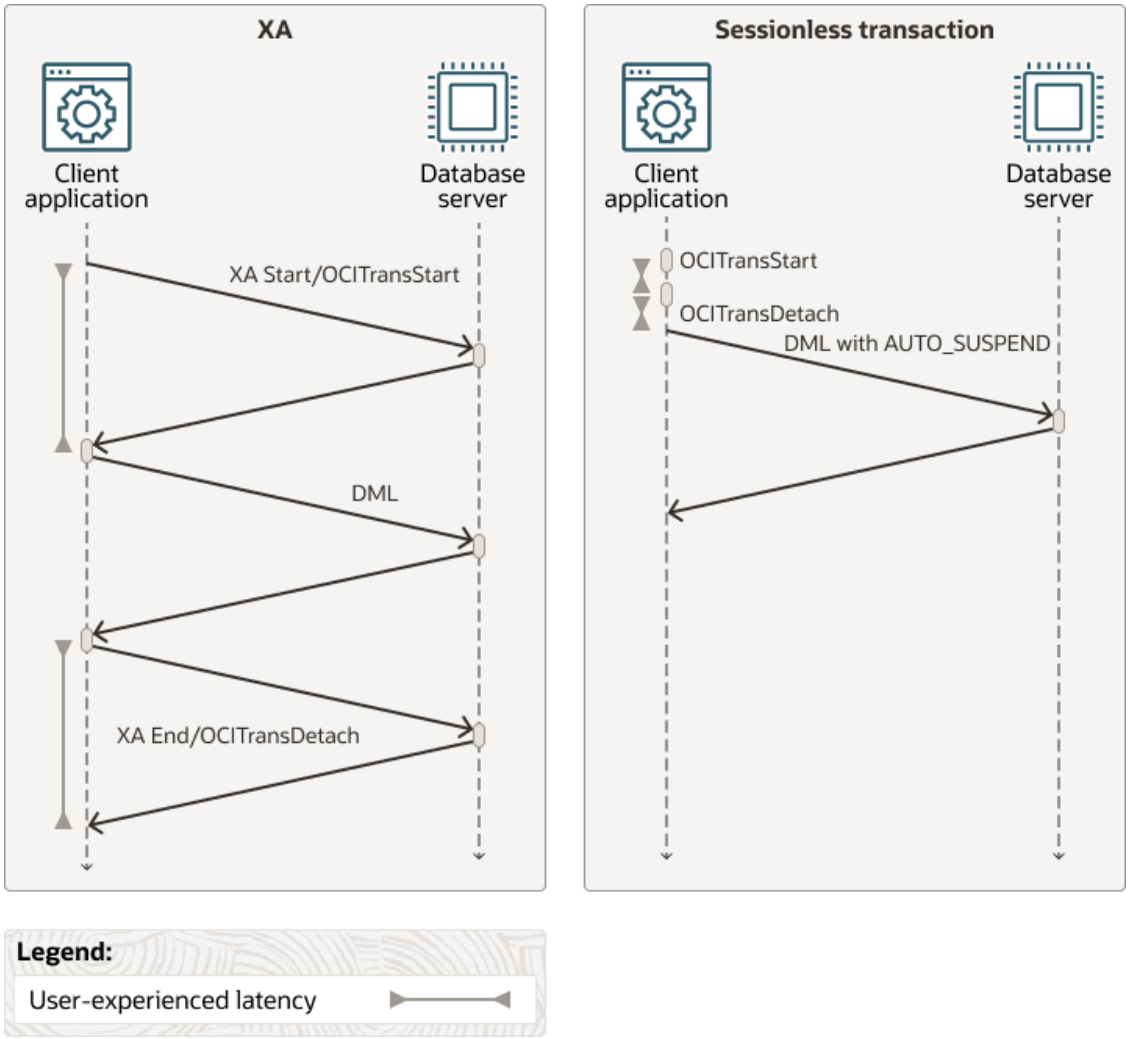
making several round trips to the database server, resulting in increased commit latency. As shown in the following Figure 31-3, Oracle internally coordinates 2PC for Sessionless transactions. The clients do not need to issue the prepare round-trip calls, which have higher latencies when compared to communication between Oracle server instances.

Figure 31-3 XA Transactions and Sessionless Transactions 2PC



In addition to reduced commit latency, the calls to start, resume, or suspend a Sessionless transaction can be attached to the next network round trip. As such, the user-conceived Sessionless transaction's start, resume, and suspend latencies are less than that of the counterparts of XA transactions. Figure 31-4 illustrates the starting and detaching/suspending latency differences between XA transactions and Sessionless transactions. With Sessionless transactions, the `OCITransStart` and `OCITransDetach` calls return immediately without making a round trip to the database server. These requests are sent to the server along with the subsequent server round trip to the database server.

Figure 31-4 XA Transactions and Sessionless Transactions



Building Highly Fault-tolerant and Auto-recoverable Applications

With Sessionless Transactions, the commit protocol is coordinated by the database. Hence, the burden of transaction recovery is removed from the applications (or the transaction managers). The transactions never go in-doubt or into a pending state. Therefore, applications do not need to deploy any recovery mechanisms.

> **See Also:**
>
> Managing Distributed Transactions in *Database Administrator's Guide* for information about managing in-doubt transactions

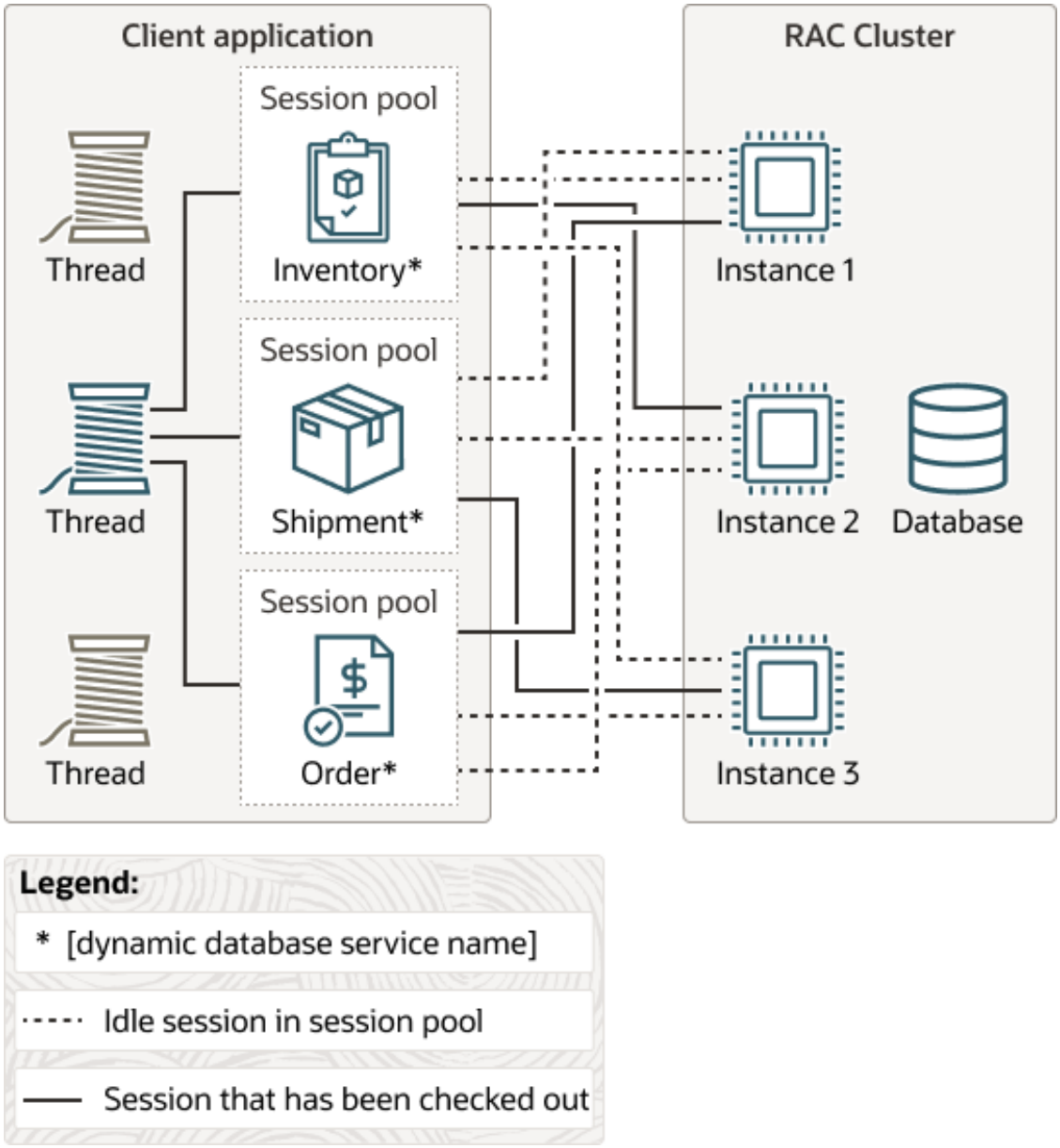
31.3.1 Example Use Cases

Applications Using Dynamic Database Services

You can create multiple services on a RAC cluster and specify their attributes, such as service-level thresholds and priority. Sessions are connected to a specific service. With the Sessionless Transaction feature, the application can wrap the changes that are done using different services into a single Sessionless transaction. In the following Figure 31-5, a thread in a client application uses multiple sessions that are connected to different services, which make transactional changes.

> **See Also:**
>
> Overview of Automatic Workload Management with Dynamic Database Services in RAC Administration and Deployment Guide for more information about Dynamic Database Services

Figure 31-5 Dynamic Database Services with Sessionless Transactions



Microservices

Microservices follow the stateless paradigm of application deployment. If several microservices have to be logically combined under a transaction, you have two choices:

- Deploy external TMs like MicroTx to manage the individual microservices as branches of a global transaction.
- Auto-commit each microservice request. Upon error, an application needs to call compensatory microservices to undo the failed task, which is complex to implement for applications because the recovery logic needs to be handled by the application.

Figure 31-6 and Figure 31-7 respectively demonstrate how to use XA transactions and Sessionless Transactions to combine work from multiple microservices into a single

transaction. Sessionless Transactions offers a simpler solution and removes the transaction manager as the middle tier.

Figure 31-6 Microservices with XA Transactions

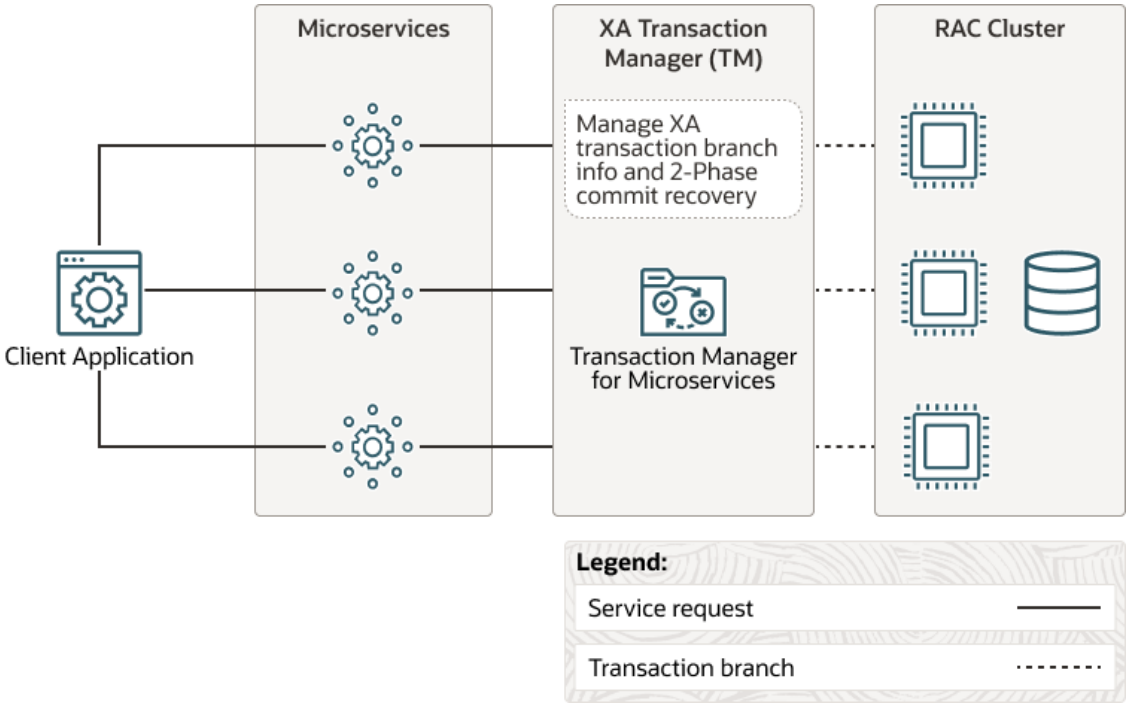
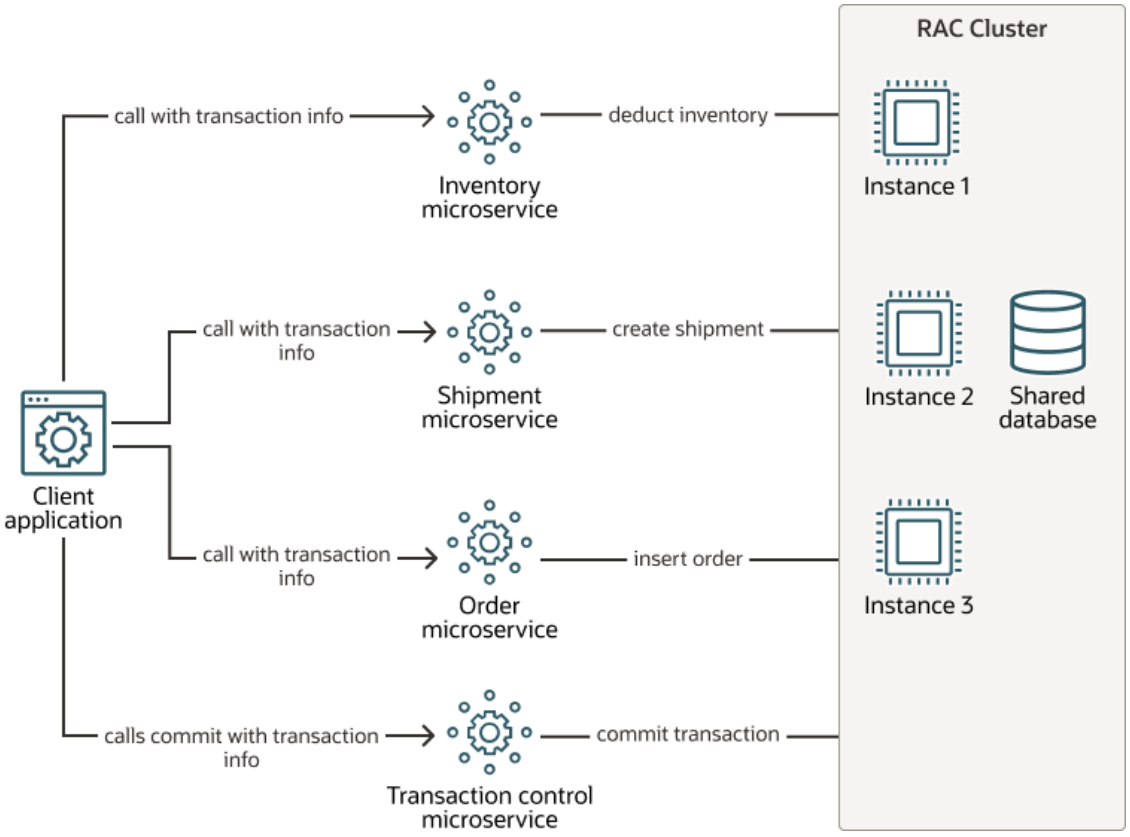


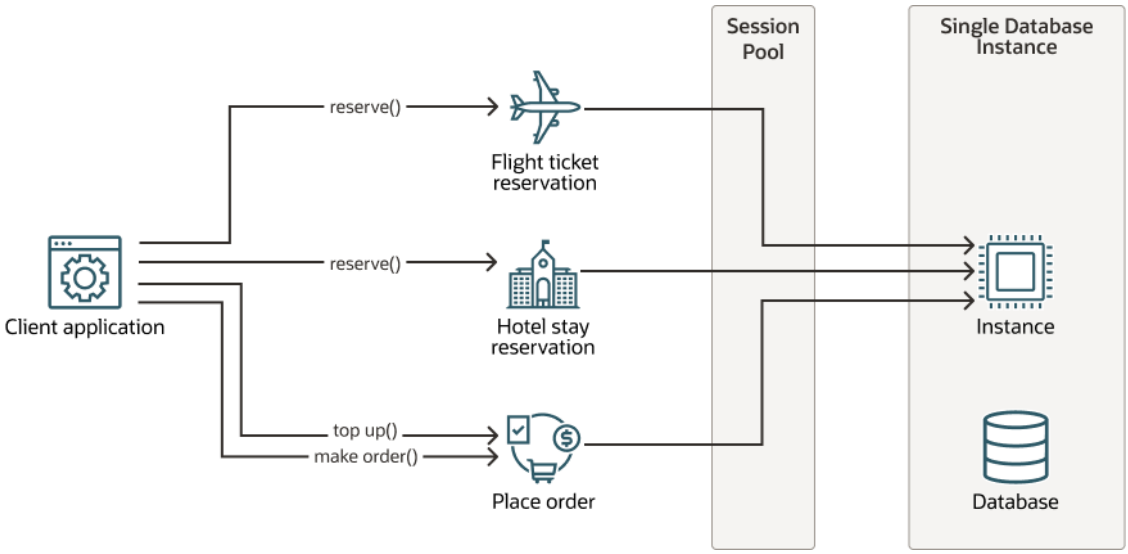
Figure 31-7 Microservices with a Single Sessionless Transaction



Interactive Applications

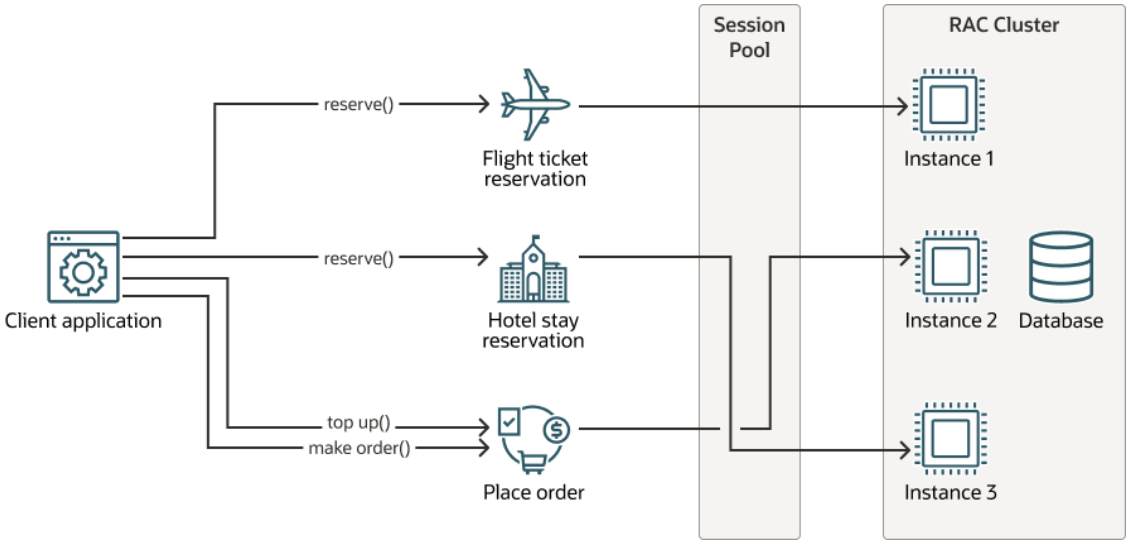
An interactive application as an application with think time and computation logic can take advantage of Sessionless Transactions. For example, a travel reservation service may let the customers browse available hotel vacancies and flight tickets. When browsing, the customers may hold the hotel or flight reservations that they are interested in for a certain duration. Once the itinerary is finalized, the customers are asked to pay for the bookings from their account balance. Sessionless transactions use the database sessions only during the actual operation and can release the database sessions during the customer's browsing and input time. As such, the same set of database sessions can serve more concurrent requests. Figure 31-8 and Figure 31-9 depict such use cases on a single-instance deployment and a RAC deployment respectively.

Figure 31-8 Interactive Applications with Sessionless Transactions on a Single-instance Deployment



The interactive application can run on a RAC deployment to enjoy the availability and load balancing provided by RAC.

Figure 31-9 Interactive Applications with Sessionless Transactions on a RAC Deployment



31.4 Benefits of Sessionless Transactions

The following are the benefits of using Sessionless Transactions in your applications.

Efficient Use of Session and Connection Resources

Sessionless Transactions allows applications to multiplex sessions and transactions without the complexity of the eXtended Architecture (XA).

> **See Also:**
>
> Developing Applications with Oracle XA for information about Oracle XA

Database-managed Recovery

Oracle Database is responsible for transaction recovery and commit operations for Sessionless transactions. The client application does not require transaction management or an external transaction manager. Transactions also never go in-doubt.

Reduced Network Round Trips between Application Client and (Oracle) Database Server

Using Sessionless Transactions, applications would simply send the commit command and the server manages the commit protocol internally for the transactions. The number of round trips between the client application and the database instances is reduced substantially.

Transaction Can Be Resumed across RAC Database Instances

Unlike the XA protocol, a Sessionless transaction can be started and suspended on one RAC database instance and resumed on another RAC database instance. An application does not need to know about which RAC database instance the transaction uses.

In Oracle XA, to suspend and resume an XA transaction on different RAC database instances, the external transaction managers must have the knowledge of each RAC database instance, create at least one branch per instance, which is used by the XA transaction, and manage the 2PC of these branches. The XA driver cannot suspend an XA transaction on one instance and resume it on another. The users are forced to create branches on a new instance to continue the work under a given XA transaction.

Reduced 2PC Protocol Latency

Sessionless Transactions coordinates commit operations internally, and hence applications can benefit from reduced commit (2PC Protocol) latency because fewer client-server round trips are needed for finalizing transactions.

If an XA transaction is spread across multiple branches, the external transaction manager must send 2PC messages to all those branches.

External Transaction Managers Are Not Required

With Sessionless Transactions, there is no need for an external transaction manager when communicating with one database.

31.5 Using Sessionless Transactions

The following sections explain how to use Sessionless transactions in your applications.

- Understanding Active Sessionless Transactions
- Understanding the Lifecycle of Sessionless Transactions
- Understanding Server Round Trips and Pre-call and Post-call Functions
- Prerequisites for Using Sessionless Transactions
- Setting a Global Transaction ID
- Starting a New Sessionless Transaction

31.5.1 Understanding Active Sessionless Transactions

A Sessionless transaction that is currently associated with a session is said to be an active Sessionless transaction. A Sessionless transaction is active from the time it is newly started or resumed to the time the transaction is suspended, committed, or rolled back.

31.5.2 Understanding the Lifecycle of Sessionless Transactions

You can start a Sessionless transaction on the server (using the PL/SQL functions and procedures in the `DBMS_TRANSACTION` package) or on the client (using client APIs, such as Oracle Call Interface (OCI) or Oracle JDBC). The set of APIs to start, resume, and suspend Sessionless transactions on the server is not interoperable with that on the client during the duration in which the Sessionless transaction is active.

A unique transaction identifier called GTRID (Global Transaction ID, which is also referred to as Transaction ID) identifies each Sessionless transaction. When starting a Sessionless transaction, you can specify the desired GTRID that identifies the transaction. If you do not provide a GTRID, the database server (if you start the Sessionless transaction on the database server) or the client driver (if you start the Sessionless transaction on the client) generates an identifier that is used as the transaction's GTRID. Subsequent units of work submitted to the Oracle Database server are in the transaction context. Over the course of time, the transaction could be suspended and resumed between units of work, multiple times. The transaction ends when the transaction is committed or rolled back.

Unlike local transactions where the row lock waiting period is infinite, in Sessionless Transactions, row lock waiting period is bound by the `DISTRIBUTED_LOCK_TIMEOUT` database parameter. After the timeout, the statement that is waiting on a row lock receives an error, and is rolled back.

31.5.3 Understanding Server Round Trips and Pre-call and Post-call Functions

Server Round Trip

A server round trip is the trip from the client to the server and then back to the client. The user call that incurs the server round trip is the main call. Each server round trip has a main call and may have pre-call functions and post-call functions attached to the main call.

Pre-call and Post-call Functions

For certain user calls, client libraries do not immediately incur a server round trip. Instead, client libraries store the information of the calls and attach the calls on the next server round trip. If the attached call is run before the main call of the server round trip, such a call is calling

a pre-call function, and if the attached call is run after the main call of the server round trip, such a call is calling a post-call function.

Pre-call Functions

An example of a pre-call function is as follows:

```
int main()
{
  ...
  OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);  /*
a pre-call that starts a Sessionless transaction */
  OCIStmtExecute(...) /* a call that incurs a round-trip to server. This is
the "server round trip." */
  ...
}
```

In the preceding example, the first OCI call to start a Sessionless transaction (`OCITransStart`) is a pre-call function. This pre-call returns, after the client library records the call information, without incurring a server round trip. The `OCIStmtExecute` call incurs the server round trip (and therefore is the round trip's main call), on which the `OCITransStart` pre-call function attaches itself. To handle this server round trip, the database server first runs the pre-call function (`OCITransStart`) and then runs the main call (`OCIStmtExecute`).

If any pre-call function raises an error, the main call and the possible post-call functions are not invoked.

Post-call Functions

Post-call functions are attached to the next server round trip and are run after the successful running of the main call.

An example of a post-call function is as follows:

```
int main()
{
   ...
  OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS |
OCI_TRANS_POST_CALL));  /* a post-call which suspends a Sessionless
transaction */
  OCIStmtExecute(...) /* a call that incurs a round-trip to server. This is
the "main call". */
  ...
}
```

In the preceding example, the first OCI call to suspend the Sessionless transaction (`OCITransDetach`) is a post-call function. It is returned immediately after the client library records this call, but the call is not sent to the server yet. `OCIStmtExecute` incurs the server round trip (and therefore is the round trip's main call), on which the post-call function `OCITransDetach` attaches itself. To handle this server round trip, the database server first runs the main call (`OCIStmtExecute`) and then runs the post-call function (`OCITransDetach` with `OCI_TRANS_POST_CALL`).

If the main call raises an error, the post-call functions are not invoked.

31.5.4 Prerequisites for Using Sessionless Transactions

Prerequisites for Using Sessionless Transactions on the Server

There are no prerequisites for using Sessionless Transactions on the server.

Prerequisites for Using Sessionless Transactions on the OCI Client

To use the start, suspend, and resume Sessionless Transactions APIs on the OCI client, you must allocate a transaction handle and set it as the `OCI_ATTR_TRANS` attribute of the service context handle.

```
OCISvcCtx *svchp = NULL;

/* get a service context handle from the session pool. */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0,
              NULL, NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

/* create a transaction handle */
OCITrans    *txnhp  = (OCITrans *)0; /* Sessionless transaction */
OCIHandleAlloc(envhp, (void **)&txnhp, OCI_HTYPE_TRANS, 0, 0);

/* set txn handle as service context handle's OCI_ATTR_TRANS attribute */
OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);
```

31.5.5 Setting a Global Transaction ID

Every Sessionless transaction is identified by a unique transaction identifier called Global Transaction ID (GTRID). You can either provide your desired GTRID or let Oracle generate a GTRID to identify the Sessionless transaction to be started. In either case, you need to set the GTRID field properly so that the client driver or the database server can perform the right action.

Sessionless APIs are provided both on the client side and on the server side, enabling you to maximize the flexibility to independently upgrade to the database server and client versions.

> **Note:**
>
> For auto-generated GTRIDs, the client library generates the GTRID for a Sessionless transaction that is started on the client and the database server generates the GTRID for a Sessionless transaction that is started on the database server.

Setting a GTRID on the Database Server

A GTRID is set on the server as a part of starting a Sessionless transaction on the server.

> **See Also:**
>
> Starting a New Sessionless Transaction

Setting GTRID on the OCI Client

> **Note:**
>
> This should be done before starting the transaction.

If you prefer to identify the new Sessionless transaction with a specific GTRID, create an XID structure, set the GTRID in the XID's `data` member, and set the XID's `gtrid_length` member to the size of the GTRID (in bytes). Then, call `OCIAttrSet` to set transaction handle's `OCI_ATTR_XID` attribute to the pointer of the XID structure, as shown in the following code sample. The client library records the GTRID and stores it in the transaction handle.

Example 31-1 Setting a Specific GTRID

```
/* Get current transaction handle */
OCITrans *current_txnhp;
OCIAttrGet(svchp, OCI_HTYPE_SVCCTX, &current_txnhp, NULL,
           errhp);

/* set GTRID in XID struct */
XID         res_xid;
strcpy(res_xid.data, "client_user_gtrid_1");
res_xid.gtrid_length = strlen("client_user_gtrid_1");

/* set XID as txn handle's attribute */
OCIAttrSet(current_txnhp, OCI_HTYPE_TRANS, &res_xid, sizeof(XID),
OCI_ATTR_XID, errhp);
```

If you prefer to use a generated UUID as the GTRID identifying the new Sessionless transaction, set the transaction handle's `OCI_ATTR_XID` attribute to a `NULL` pointer, as shown in the following code sample. When starting the transaction, a UUID is generated by the client driver and used as GTRID.

If a generated UUID is used as the GTRID, before the transaction gets suspended, you must retrieve the XID from the transaction handle so that you can use it again to resume or commit the transaction.

> **See Also:**
>
> Retrieving a Global Transaction ID

Example 31-2 Setting a Generated GTRID

```
/* Get current transaction handle */
OCITrans *current_txnhp;
```

```
OCIAttrGet(svchp, OCI_HTYPE_SVCCTX, &current_txnhp, NULL,
           errhp);

/* set XID as txn handle's attribute */
OCIAttrSet(current_txnhp, OCI_HTYPE_TRANS, NULL, sizeof(XID), OCI_ATTR_XID,
errhp);
```

You cannot set an XID (`OCI_ATTR_XID`) when:

- There is an active Sessionless transaction that was started or resumed on the database server. In this case, `OCI_ATTR_TRANS` attribute of the service context handle is a read-only transaction handle managed by the OCI library (returns an error 26210).
- An active Sessionless transaction was started with client library generated GTRID, and such GTRID has not been retrieved yet (returns an error 26204).
- A pre-call function to start a new Sessionless transaction or resume an existing Sessionless transaction is recorded but is not yet sent to the database server (returns an error 26216).

> **See Also:**
>
> OCIAttrSet() and OCIAttrGet() in *Oracle Call Interface Developer's Guide*

31.5.6 Starting a New Sessionless Transaction

You can start a Sessionless transaction on the server or on the client. If an active Sessionless transaction exists when the request for starting a new transaction is made, such an active transaction is suspended, regardless of whether the new transaction request succeeds or not.

When you start a new Sessionless transaction, you can specify a unique GTRID (or use a generated GTRID) and specify a time-out value to use if the transaction is suspended.

> **See Also:**
>
> Time-out of a Suspended Sessionless Transaction for information about time-out of suspended Sessionless transactions

Starting a New Sessionless Transaction on the Database Server

To start a new Sessionless transaction on the server, run the `DBMS_TRANSACTION.START_TRANSACTION` PL/SQL function. Set the flag to `TRANSACTION_NEW` to specify that the request must start a new transaction.

You can start a Sessionless transaction on the server using a user-specified GTRID or a generated GTRID.

To start a Sessionless transaction on the database server using a generated UUID as GTRID, simply set the RAW-type parameter `XID` of the `DBMS_TRANSACTION.START_TRANSACTION` PL/SQL function to `NULL`. The function invokes the GUID generator to create a 16-byte random identifier, which is used as the GTRID. Alternatively, you can use the existing `SYS_GUID()` API to obtain a globally unique identifier and pass it in as a Sessionless transaction's GTRID.

See Also:

- `DBMS_TRANSACTION` package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function
- SYS_GUID in *SQL Language Reference Guide*

Note:

Programs that use an older client library, which does not support Sessionless transactions can still use Sessionless transactions if the Sessionless transaction is started on a server that supports Sessionless transactions.

Example 31-3 Starting a New Sessionless Transaction on the Server with a User-Specified GTRID

The following example starts a new Sessionless transaction on the server with a specific GTRID:

```
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( UTL_RAW.CAST_TO_RAW('user_specified_gtrid')
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );
  DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
user_specified_gtrid


PL/SQL procedure successfully completed.
```

The following retrieves the transaction ID of the newly created Sessionless transaction.

```
BEGIN
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN

DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION
_ID()));
  ELSE
```

```
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;
 END;
 /
```

The output is:

```
user_specified_gtrid

PL/SQL procedure successfully completed.
```

Example 31-4 Starting a New Sessionless Transaction on the Server with a Generated GTRID

The following example starts a new Sessionless transaction on the server using a generated GTRID.

```
-- Start a new Sessionless transaction with a GTRID generated by the
-- START_TRANSACTION function.
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( NULL
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );
  DBMS_OUTPUT.PUT_LINE(gtrid);
END;
/
```

The output is:

```
9AF035CCB2024FA6BF17C8CDBF7B6D6C

PL/SQL procedure successfully completed.
```

The following retrieves the transaction ID of the newly created Sessionless transaction.

```
BEGIN
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN

DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION
_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;
 END;
 /
```

The output is:

```
9AF035CCB2024FA6BF17C8CDBF7B6D6C

PL/SQL procedure successfully completed.
```

Client Library Behavior when Sessionless Transaction Starts on the Server Side

For the client library that supports Sessionless transactions, once a Sessionless transaction starts on the server, the server informs the client library about when a Sessionless transaction has started, resumed, suspended, or ended. In OCI's case, when the information is received, the service context handle's `OCI_ATTR_TRANS` attribute is changed to a pointer to a transaction handle managed by the OCI client library. This managed transaction handle is read-only and is available only when the corresponding Sessionless transaction is active. Users can retrieve GTRID from this handle (See Retrieving GTRID on the Database OCI client section). Once the Sessionless transaction is no longer active, the `OCI_ATTR_TRANS` attribute is set back to the value that was set before the Sessionless transaction became active.

Starting a New Sessionless Transaction on the OCI Client

To start a new Sessionless transaction on the client side, run the `OCITransStart` function with the (`OCI_TRANS_SESSIONLESS` | `OCI_TRANS_NEW`) flag and the timeout settings. The `OCI_TRANS_SESSIONLESS` flag distinguishes this request from the requests to start an XA branch. When `OCITransStart` is used with the `OCI_TRANS_SESSIONLESS` flag, it is always a pre-call function.

When you start a new Sessionless transaction, you can specify a unique GTRID (or use a generated GTRID) and specify a time-out value to use if the transaction is suspended.

> **See Also:**
>
> Setting a Global Transaction ID

Example 31-5 Starting a New Sessionless Transaction on the OCI Client

```
OCISvcCtx *svchp;
OCIError *errhp;
...
ub4 timeout = 60;

/* start the transaction. This will be attached on the subsequent server
round trip.
    */
OCITransStart(svchp, errhp, timeout, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
```

timeout: how long (in seconds) after the suspension of this Sessionless transaction should the server roll back this transaction. This avoids a transaction from holding on to database resources (such as row locks) indefinitely.

> **Note:**
>
> The timeout value must be a positive number.

> **See Also:**
>
> - OCITransStart() in *Oracle Call Interface Developer's Guide*
> - Time-out of a Suspended Sessionless Transaction

31.5.6.1 Running SQL Statements within Sessionless Transactions

After starting a new Sessionless transaction or resuming an existing one, all the SQL statements issued are in the context of the Sessionless transaction.

> **Note:**
>
> If a SQL statement causes any lock contention when a Sessionless transaction is active (for example, updating a row whose row lock is held by another transaction), the transaction waits for the lock for up to `distributed_lock_timeout` seconds, and then returns an ORA-2049 error. If an error is returned, the statement is rolled back, but other work in the transaction remains intact.

> **See Also:**
>
> - Starting a New Sessionless Transaction
> - Resuming a Suspended Sessionless Transaction

31.5.7 Retrieving a Global Transaction ID

You can retrieve the GTRID of a Sessionless transaction that is currently in the active state.

Retrieving GTRID on the Database Server

To retrieve the GTRID of an active Sessionless transaction on the server, use the `DBMS_TRANSACTION.GET_TRANSACTION_ID` PL/SQL function.

Example 31-6 Retrieving GTRID on the Database Server

The following code samples start a Sessionless transaction and use the `GET_TRANSACTION_ID()` function to retrieve the Sessionless transaction's GTRID.

```
SET SERVEROUTPUT ON

DECLARE
```

```
  gtrid VARCHAR2(128);
BEGIN
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( UTL_RAW.CAST_TO_RAW('user_specified_gtrid')
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );
  DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
user_specified_gtrid

PL/SQL procedure successfully completed.
```

The following retrieves the GTRID.

```
BEGIN
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN

DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION
_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;
END;
/
```

The output is:

```
user_specified_gtrid

PL/SQL procedure successfully completed.
```

> **See Also:**
>
> `DBMS_TRANSACTION` package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `GET_TRANSACTION_ID` function

Retrieving a GTRID on the OCI Client

To retrieve a GTRID on the OCI client, call the `OCIAttrGet` function to retrieve `OCI_ATTR_XID` attribute of the transaction handle, which is a pointer to an XID structure. Then, read the XID structure's `gtrid_length` member to learn about the size of the GTRID in bytes and the `data` member to get the actual content of the GTRID.

> **Note:**
>
> - If the GTRID was generated by the client library, users must retrieve the transaction's GTRID before suspending the transaction. Users must have the GTRID to resume and commit the transaction.
> - The pointer obtained from `OCIAttrGet` is read-only. Users should set GTRID using the method specified in Setting GTRID on the OCI Client.

Example 31-7 Retrieving GTRID on the OCI Client

```
OCITrans *current_txnhp;
XID      *current_xidp;
char      gtrid_buffer[65];

/* get transaction handle from service context handle */
OCIAttrGet(svchp, OCI_HTYPE_SVCCTX, &current_txnhp, NULL, OCI_ATTR_TRANS,
errhp);

/* get XID from transaction handle */
OCIAttrGet(current_txnhp, OCI_HTYPE_TRANS, &current_xidp, NULL, OCI_ATTR_XID,
errhp));

/* get GTRID from XID */
memcpy(gtrid_buffer, current_xidp->data, current_xidp->gtrid_length);
```

> **See Also:**
>
> OCIAttrGet() in *Oracle Call Interface Developer's Guide*

31.5.8 Suspending a Sessionless Transaction

You can suspend an active Sessionless transaction using the same method (on the database server or on the client) that the Sessionless transaction is last started or resumed. The suspend function or procedure can only be applied to Sessionless transactions. If no transaction is in the session, the suspend is a no-op; if a non-sessionless transaction (for example, a local transaction or an XA transaction) is in the session, calling the suspend function raises an ORA-26202 error, but leaves the transaction in the session intact.

Suspending an Active Sessionless Transaction on the Database Server

Run the `DBMS_TRANSACTION.SUSPEND_TRANSACTION` PL/SQL procedure to suspend a Sessionless transaction that is active in a particular session. This procedure does not have any parameters.

> **See Also:**
>
> DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `SUSPEND_TRANSACTION` procedure

Example 31-8 Suspending an Active Sessionless Transaction on the Server

The following is an example of suspending an active Sessionless transaction on the server:

```
SET SERVEROUTPUT ON

DECLARE
  gtrid VARCHAR2(128);
BEGIN
  -- start Sessionless transaction
  gtrid := DBMS_TRANSACTION.START_TRANSACTION
  ( UTL_RAW.CAST_TO_RAW('user_specified_gtrid')
  , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
  , 20
  , DBMS_TRANSACTION.TRANSACTION_NEW
  );

  -- confirm the transaction is started
  IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN
    DBMS_OUTPUT.PUT_LINE('After start: ' ||
UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION_ID()));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
  END IF;

  -- suspend the transaction
  DBMS_TRANSACTION.SUSPEND_TRANSACTION;
  -- confirm the transaction is suspended
  DBMS_OUTPUT.PUT_LINE('After suspend: ' ||
DBMS_TRANSACTION.GET_TRANSACTION_ID());
END;
/
```

The output is:

```
After start: user_specified_gtrid
After suspend:

PL/SQL procedure successfully completed.
```

Suspending an Active Sessionless Transaction on the OCI Client

To suspend an active Sessionless transaction on the OCI client, call `OCITransDetach` with an appropriate flag depending on how you would like to suspend the transaction. You can specify flag bits to indicate your required option. When the flag is set to (`OCI_TRANS_SESSIONLESS | OCI_DEFAULT`), the suspend is run immediately. When the flag is set to (`OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL`), the suspend a pre-call function (attached

to the next server round trip and run before the main call). When the flag is set to (OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL), the suspend is a post-call function (attached to the next server round trip and run after the main call).

> **Note:**
>
> If the pre-call suspend (OCI_TRANS_PRE_CALL is specified) raises an error, the main call is not run. If the main call raises an error, the post-call suspend (OCI_TRANS_POST_CALL is specified) is not run.

Example 31-9 Immediate Suspend

To suspend the active Sessionless transaction immediately, call the OCITransDetach function with the flag (OCI_TRANS_SESSIONLESS | OCI_DEFAULT), as in the following example.

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_DEFAULT);
```

Example 31-10 Pre-call Suspend (Suspend before the Next Server Round Trip)

To issue a pre-call suspend (suspend on the next server round trip, before the main call), call the OCITransDetach function with the OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL flag. Being a pre-call function, the suspend code is run *before* the main call.

In the following example, the transaction is actually suspended before the OCIPing() call.

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL);
OCIPing(svchp, errhp, OCI_DEFAULT);
```

The run sequence of immediate suspend and pre-call suspend are the same, but pre-call suspend is attached to the next server round trip, thus reducing network latency. For example, if you would make a query after suspending a Sessionless transaction, using pre-call suspend can save a server round trip and reduce latency.

Example 31-11 Post-call Suspend (Suspend after the Next Server Round Trip)

To issue a post-call suspend (suspend on the next server round trip, after the main call), call the OCITransDetach function with the OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL flag.

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL);
OCIPing(svchp, errhp, OCI_DEFAULT);
```

Doing this is useful if you know about the last round trip to the server (for example, the last DML of this unit of work).

Example 31-12 Post-call Suspend

In the following example, the transaction identified by the "my_Sessionless_txn3" GTRID is suspended automatically after the successful running of OCIStmtExecute().

```
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0,
               NULL, NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);
OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);
```

```
memcpy(xidp->data, "my_Sessionless_txn3", strlen("my_Sessionless_txn3"));
xidp->gtrid_length = strlen("my_Sessionless_txn3");
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);

// Ask a post-call suspend
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_POST_CALL);

// Do the DML
OCIStmtPrepare(…); /* insert into mytab1(c1, c2) values (1, 1); */
OCIStmtBindByPos(…);
OCIStmtExecute(…);   /* Server suspends the transaction after this DML is
executed. */
```

See Also:

- OCITransDetach() and other OCI library functions in *Oracle Call Interface Developer's Guide*
- Resuming a Suspended Sessionless Transaction

31.5.8.1 Time-out of a Suspended Sessionless Transaction

The time-out value of a Sessionless transaction is set when starting a new Sessionless transaction.

Every time the Sessionless transaction is suspended, the timer starts to tick. If a Sessionless transaction is resumed within the time-out period, the timer is reset. If a Sessionless transaction is not resumed within the time-out period, it is rolled back. This is because every transaction in Oracle Database potentially holds resource locks (such as row locks). These resources are held for the duration of the transaction, so if the transaction remains active indefinitely, it will cause outages.

Note:

A zero time-out value is not allowed. You must always specify a positive time-out value.

Example 31-13 Timing Out a Sessionless Transaction

This is an example of a Sessionless transaction that has timed out.

The following starts a Sessionless transaction.

```
SET ECHO ON
SET SERVEROUTPUT ON

DROP TABLE IF EXISTS mytab1;
CREATE TABLE mytab1(c1 NUMBER, c2 NUMBER);
```

```
-- start a Sessionless transaction with timeout=20 secs
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table.

```
INSERT INTO mytab1(c1, c2) VALUES (1, 1);
```

The output is:

```
1 row created.
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
```

The output is:

```
PL/SQL procedure successfully completed.
```

Trying to resume the transaction fails with error 26218 because the transaction that started has been rolled back due to the timeout.

```
-- idle for 25 seconds
!sleep 25

DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );
```

```
END;
/
```

The output is:

```
ERROR at line 1:
ORA-26218: sessionless transaction with GTRID
6D795F73657373696F6E6C6573735F74696D656F75745F6578 does not exist.
```

Trying to start a new transaction with the same GTRID succeeds because the previous transaction was rolled back.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

31.5.9 Resuming a Suspended Sessionless Transaction

You can resume a Sessionless transaction that is in a suspended state.

Resuming a suspended Sessionless transaction is similar to starting a Sessionless transaction. The differences between the two are:

- When resuming a Sessionless transaction on the server, in the `DBMS_TRANSACTION.START_TRANSACTION` function, the `TRANSACTION_RESUME` flag replaces the `TRANSACTION_NEW` flag. Similarly, when resuming a Sessionless transaction on the client library, in the `OCITransStart` function, the `OCI_TRANS_RESUME` flag replaces the `OCI_TRANS_NEW` flag.
- The timeout parameter used in resuming a transaction refers to the resume timeout as opposed to the transaction timeout, which is specified during the start of a new Sessionless transaction.

> **See Also:**
>
> DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* and Resuming a Sessionless Transaction on the OCI Client for more information about the timeout parameter as used in the `START_TRANSACTION` function and the `OCITransStart` function respectively

If an active Sessionless transaction exists when the resume request is made, such active transaction is suspended before the resume request is run, regardless of whether the resume request succeeds or not.

Resuming a Sessionless Transaction on the Database Server

To resume a suspended Sessionless transaction on the server, run the `DBMS_TRANSACTION.START_TRANSACTION` PL/SQL function. Set the flag to `TRANSACTION_RESUME`.

> **See Also:**
>
> DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function

Example 31-14 Resuming a Sessionless Transaction on the Database Server

```
CREATE PROCEDURE PRINT_GTRID IS
BEGIN
    IF DBMS_TRANSACTION.GET_TRANSACTION_TYPE() =
DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS THEN
      DBMS_OUTPUT.PUT_LINE('Sessionless transaction in session: ' ||
UTL_RAW.CAST_TO_VARCHAR2(DBMS_TRANSACTION.GET_TRANSACTION_ID()));
    ELSE
      DBMS_OUTPUT.PUT_LINE('Not a Sessionless transaction');
    END IF;
END PRINT_GTRID;
/
```

The output is:

```
Procedure created.
```

The following starts a Sessionless transaction and prints the GTRID.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
    PRINT_GTRID;
    DBMS_OUTPUT.PUT_LINE('gtrid: ' || gtrid);
    DBMS_OUTPUT.PUT_LINE('cast gtrid: ' || UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
Sessionless transaction in session: my_sessionless_timeout_ex
gtrid: 6D795F73657373696F6E6C6573735F74696D656F75745F6578
cast gtrid: my_sessionless_timeout_ex

PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table and fetches the updates.

```
INSERT INTO mytab1(c1, c2) values (1, 1);
SELECT * FROM mytab1;
```

The output is:

```
1 row created.

        C1 C2
---------- ----------
         1 1
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
SELECT * FROM mytab1;
```

The output is:

```
PL/SQL procedure successfully completed.

no rows selected
```

The following resumes the transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_timeout_ex')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );

    PRINT_GTRID;
    DBMS_OUTPUT.PUT_LINE('gtrid: ' || gtrid);
    DBMS_OUTPUT.PUT_LINE('cast gtrid: ' || UTL_RAW.CAST_TO_VARCHAR2(gtrid));
END;
/
```

The output is:

```
Sessionless transaction in session: my_sessionless_timeout_ex
gtrid: 6D795F73657373696F6E6C6573735F74696D656F75745F6578
cast gtrid: my_sessionless_timeout_ex

PL/SQL procedure successfully completed.
```

The following fetches the updates made to the table.

```
SELECT * FROM mytab1;
```

The output is:

```
        C1 C2
---------- ----------
         1 1
```

The following rolls back the transaction.

```
ROLLBACK;
```

The output is:

```
Rollback complete.
```

The following drops the `PRINT_GTRID` procedure.

```
DROP PROCEDURE PRINT_GTRID;
```

The output is:

```
Procedure dropped.
```

Resuming a Sessionless Transaction on the OCI Client

Before resuming a Sessionless transaction on the client, the service context handle must have a transaction handle as attribute, and the `OCI_ATTR_XID` attribute must be set in the transaction handle. Calling the `OCITransStart` function with the `OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME` flag and timeout resumes an existing Sessionless transaction. Similar to starting a new Sessionless transaction on the OCI client, the resume call is a pre-call function.

Example 31-15 Resuming a Sessionless Transaction on the OCI Client

```
ub4 timeout = 20;
OCITransStart(svchp, errhp, timeout, OCI_TRANS_SESSIONLESS |
OCI_TRANS_RESUME);
```

timeout: The timeout here refers to the resume timeout, which is different from the transaction timeout. The resume time-out value can be a 0 or a positive number. Within an instance, a

Sessionless transaction can only be associated with one session at any given time. Thus, a Sessionless transaction cannot be resumed if it is associated with another session on the same instance. When this situation arises, the resume timeout value specifies how long (in seconds) this session waits for the other session to suspend the transaction, so that it can resume the transaction. If the other session does not suspend the transaction before the wait times out, an ORA-25351 error is raised. If the resume timeout value is 0, the error is immediately raised.

See Also:

- OCITransStart() in *Oracle Call Interface Developer's Guide*
- Time-out of a Suspended Sessionless Transaction

31.5.10 Finalizing a Sessionless Transaction

This section explains how you can commit or roll back a Sessionless transaction.

Topics:

- Committing a Sessionless Transaction
- Rolling Back a Sessionless Transaction

31.5.10.1 Committing a Sessionless Transaction

A Sessionless transaction can be ended by all the usual means that end a local transaction. In terms of committing a Sessionless transaction, the SQL statement `COMMIT`, `OCITransCommit`, and `OCIStmtExecute` in `OCI_COMMIT_ON_SUCCESS` mode can be used.

See Also:

Commit or Roll Back Operations and OCIStmtExecute() in *Oracle Call Interface Developer's Guide* for information about committing transactions and `OCI_COMMIT_ON_SUCCESS`.

Example 31-16 Committing an Active Sessionless Transaction

In the following example, an anonymous PL/SQL block starts a Sessionless transaction identified by the "my_sessionless_commit_ex_1" GTRID, issues a DML statement, and then commits the transaction.

The following starts a Sessionless transaction.

```
DROP TABLE IF EXISTS mytab1;
CREATE TABLE mytab1(c1 NUMBER, c2 NUMBER);

-- start a Sessionless transaction and print the GTRID
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
```

```
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_commit_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table.

```
INSERT INTO mytab1(c1, c2) values (1, 1);
SELECT * FROM mytab1;
```

The output is:

```
1 row created.

        C1 C2
---------- ----------
         1 1
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
SELECT * FROM mytab1;
```

The output is:

```
PL/SQL procedure successfully completed.
no rows selected
```

The following resumes the transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_commit_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following commits the transaction.

```
COMMIT;
```

The output is:

```
Commit complete.
```

Example 31-17 Commit with `OCITransCommit()`

Similarly, to commit an active Sessionless transaction in a session, you can run the `OCITransCommit()` API from the client.

The following example starts a Sessionless transaction identified by the "my_Sessionless_txn5" GTRID, issues a DML statement, and then commits the transaction.

```
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0,
               NULL, NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);
OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

memcpy(xidp->data, "my_Sessionless_txn5", strlen("my_Sessionless_txn5"));
xidp->gtrid_length = strlen("my_Sessionless_txn5");
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);

OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCITransCommit(svchp, errhp, OCI_DEFAULT);
```

Example 31-18 Commit with `OCISessionRelease()`

In keeping with the existing behavior, the `OCISessionRelease` function can also commit an active Sessionless transaction. In the following example, the function is used to commit a Sessionless transaction having the "my_Sessionless_txn6" GTRID.

```
/* start Sessionless transaction */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0, NULL,
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

memcpy(xidp->data, "my_Sessionless_txn6", strlen("my_Sessionless_txn6"));
xidp->gtrid_length = strlen("my_Sessionless_txn6");
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
/* done start Sessionless transaction */
```

```
OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCISessionRelease(…); /* Commits the Sessionless transaction */
```

Example 31-19 Commit with `OCI_COMMIT_ON_SUCCESS`

The following example uses `OCI_COMMIT_ON_SUCCESS` to commit a Sessionless transaction.

```
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIBind *bndhp[2];
char *dml = "INSERT INTO mytab1(c1, c2) VALUES (:1, :2)";
int number[2];

...

OCIStmtPrepare2(svchp, &stmthp, errhp, dml, strlen(dml), 0, 0, OCI_NTV_SYNTAX,
                OCI_DEFAULT);
OCIBindByPos(stmthp, &bndhp[0], errhp, 1, &number[0], sizeof(int), SQLT_INT,
0, 0, 0,
             0, 0, OCI_DEFAULT);
OCIBindByPos(stmthp, &bndhp[1], errhp, 2, &number[1], sizeof(int), SQLT_INT,
0, 0, 0,
             0, 0, OCI_DEFAULT);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_COMMIT_ON_SUCCESS);
OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT);
```

See Also:

- DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function
- OCITransCommit(), OCISessionRelease(), and OCIStmtExecute() in *Oracle Call Interface Developer's Guide*

31.5.10.2 Rolling Back a Sessionless Transaction

Sessionless transaction can be rolled back by all the usual means that end a local transaction, such as the SQL statements `ROLLBACK`, `OCITransRollback()` and `OCIRequestEnd()`.

See Also:

OCIRequestEnd() - TAC-23c and OCITransRollback() in *Oracle Call Interface Developer's Guide* for information about `OCIRequestEnd()` and `OCITransRollback()` that enable you to roll back an open transaction

Example 31-20 Rollback Using PL/SQL

In the following example, an anonymous PL/SQL block starts a Sessionless transaction identified by the "my_sessionless_rollback_ex_1" GTRID, issues a DML statement, and rolls back the transaction.

The following code starts the Sessionless transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_rollback_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_NEW
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following inserts values into the `mytab1` table and fetches them from the table.

```
INSERT INTO mytab1(c1, c2) values (1, 1);
SELECT * FROM mytab1;
```

The output is:

```
1 row created.

        C1 C2
---------- ----------
         1 1
```

The following suspends the transaction.

```
EXEC DBMS_TRANSACTION.SUSPEND_TRANSACTION;
```

The output is:

```
PL/SQL procedure successfully completed.
```

Trying to fetch from the table shows no result.

```
no rows selected
```

Now, let us resume the transaction.

```
DECLARE
    gtrid VARCHAR2(128);
BEGIN
    gtrid := DBMS_TRANSACTION.START_TRANSACTION
    ( UTL_RAW.CAST_TO_RAW('my_sessionless_rollback_ex_1')
    , DBMS_TRANSACTION.TRANSACTION_TYPE_SESSIONLESS
    , 20
    , DBMS_TRANSACTION.TRANSACTION_RESUME
    );
END;
/
```

The output is:

```
PL/SQL procedure successfully completed.
```

The following fetches the transaction details from the table.

```
SELECT * FROM mytab1;
```

The output is:

```
        C1 C2
---------- ----------
         1 1
```

The following rolls back the transaction.

```
ROLLBACK;
```

The output is:

```
Rollback complete.
```

The following verifies that the roll back was indeed successful.

```
SELECT * FROM mytab1;
```

The output is:

```
no rows selected
```

Example 31-21 `OCITransRollback()`

Similarly, to roll back an active Sessionless transaction in a session, run the `OCITransRollback()` API from the client.

The following example starts a Sessionless transaction identified by the "my_Sessionless_txn8" GTRID, issues a DML statement, and then rolls back the transaction.

```
/* start Sessionless transaction */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0, NULL,
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

char gtrid = "my_Sessionless_txn8";
memcpy(xidp->data, gtrid, strlen(gtrid));
xidp->gtrid_length = strlen(gtrid);
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
/* done start Sessionless transaction */

OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCITransRollback(svchp, errhp, OCI_DEFAULT);
```

Example 31-22 Rollback with `OCIRequestEnd()`

In keeping with the existing behavior, the `OCIRequestEnd()` function can also roll back an open Sessionless transaction.

In the following example, the transaction identified by the "my_Sessionless_txn9" GTRID is rolled back after the successful execution of `OCIRequestEnd()`.

```
/* start Sessionless transaction */
OCISessionGet (envhp, errhp, &svchp, authInfop, poolName, poolNameLen, NULL,
0, NULL,
               NULL, NULL, OCI_SESSGET_SPOOL | OCI_SESSGET_PURITY_NEW);

OCIAttrSet (svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp);

char gtrid = "my_Sessionless_txn9";
memcpy(xidp->data, gtrid, strlen(gtrid));
xidp->gtrid_length = strlen(gtrid);
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, &xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
/* done start Sessionless transaction */

OCIRequestBegin(…);
OCIStmtPrepare(…);  // Insert into mytab1(c1, c2) values (1, 1);
OCIStmtExecute(…);

OCIRequestEnd(…); /* Rolled back the Sessionless transaction */
```

> **See Also:**
>
> - DBMS_TRANSACTION package in *Oracle Database PL/SQL Packages and Types Reference* for more information about the `START_TRANSACTION` function
> - OCITransRollback(), OCIRequestEnd() - TAC-23c, and OCIStmtExecute() in *Oracle Call Interface Developer's Guide*

31.5.11 Example: Sessionless Transactions with OCI API

The following is an end-to-end example of Sessionless Transactions using the OCI API.

> **Note:**
>
> This example requires the `scott.dept` table from the sample schema.

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define USERNAME_LEN 256
#define PASSWORD_LEN 1025
#define DBNAME_LEN   1025

OCIEnv      *envhp  = (OCIEnv *)0;
OCIError    *errhp  = (OCIError *)0;
OCIAuthInfo *authp  = (OCIAuthInfo *)0;
OCISPool    *poolhp = (OCISPool *)0;
OCISvcCtx   *svchp  = (OCISvcCtx *)0;
OCITrans    *txnhp  = (OCITrans *)0;
OCIStmt     *stmthp = (OCIStmt *)0;
OCIBind     *bndhp[3];
OCIDefine   *defhp[3];

char    *username;
char    *password;
char    *dbname;
OraText *pool_name;
ub4      pool_name_len;

void insert_dept(int deptno, const char *dname, const char *loc);
void query_dept(const char *description);
void query_transaction_type(const char *description);
void chk_err(sword ret, OCIError *errhp, sb2 linenum);
#define CHK_ERR(cmd) chk_err(cmd, errhp, __LINE__)

int main(int argc, char **argv)
```

```
{
  int nread;
  size_t usernameLen = USERNAME_LEN;
  size_t passwordLen = PASSWORD_LEN;
  size_t dbnameLen = DBNAME_LEN;

  unsigned int   i;
  unsigned char *gtrid;
  unsigned int   gtridlen = 0;
  unsigned char  gtridbuf[64];

  /* get username, password, dbname (connect string) */
  username = malloc(USERNAME_LEN * sizeof(char));
  password = malloc(PASSWORD_LEN * sizeof(char));
  dbname = malloc(DBNAME_LEN * sizeof(char));
  memset(username, 0, USERNAME_LEN);
  memset(password, 0, PASSWORD_LEN);
  memset(dbname, 0, DBNAME_LEN);

  printf("enter username:");
  if ((nread = (int)getline((char **)&username, &usernameLen, stdin)) == -1)
  {
    printf("ERROR: Failed to get username\n");
    return -1;
  }
  if(strlen(username) >= 1)
    username[strlen(username)-1] = '\0';
  printf("enter password:");
  if ((nread = (int)getline((char **)&password, &passwordLen, stdin)) == -1)
  {
    printf("ERROR: Failed to get password\n");
    return -1;
  }
  if(strlen(password) >= 1)
    password[strlen(password)-1] = '\0';
  printf("enter dbname:");
  if ((nread = (int)getline((char **)&dbname, &dbnameLen, stdin)) == -1)
  {
    printf("ERROR: Failed to get dbname\n");
    return -1;
  }
  if(strlen(dbname) >= 1)
    dbname[strlen(dbname)-1] = '\0';
  printf("\n");

  /* set up */
  {
    int ret;
    /* Create Environment */
    if ((ret = OCIEnvCreate((OCIEnv **)&envhp, (ub4)OCI_DEFAULT | OCI_OBJECT,
                            (dvoid *)0, (dvoid * (*)(dvoid *, size_t))0,
                            (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                            (void (*)(dvoid *, dvoid *))0, (size_t)0,
                            (dvoid **)0)) != OCI_SUCCESS)
    {
      printf("failed to create OCIEnv. ret = %d\n", ret);
```

```
    return 1;
  }

  /* Allocate error handle */
  CHK_ERR(OCIHandleAlloc(envhp, (void **)&errhp, OCI_HTYPE_ERROR, 0, 0));

  /* create a session pool and get a session from it */
  CHK_ERR(OCIHandleAlloc(envhp, (void **)&poolhp, OCI_HTYPE_SPOOL, 0, 0));
  CHK_ERR(OCISessionPoolCreate(
    envhp, errhp, poolhp, &pool_name, &pool_name_len, (OraText *)dbname,
    strlen(dbname), 4, 10, 1, (OraText *)username, strlen(username),
    (OraText *)password, strlen((char *)password), OCI_DEFAULT));
  CHK_ERR(OCISessionGet(envhp, errhp, &svchp, NULL, pool_name,
                        pool_name_len, 0, 0, 0, 0, 0, OCI_SESSGET_SPOOL));

  /* allocate a transaction handle and set is as OCI_ATTR_TRANS */
  CHK_ERR(OCIHandleAlloc(envhp, (void **)&txnhp, OCI_HTYPE_TRANS, 0, 0));
  CHK_ERR(
    OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp));
}

/* Query dept table before changes. */
query_dept("before begin");

/* Start a new Sessionless transaction with a generated GTRID. Insert a row
 * in dept table, then query the table. We should see the inserted row. */
CHK_ERR(
  OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp));
CHK_ERR(
  OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW));
insert_dept(50, "DEVELOPMENT1", "SEATTLE");
query_dept("in sessionless txn, after insert");

/* Get the generated GTRID so we can suspend and resume transaction. */
CHK_ERR(OCIAttrGet(txnhp, OCI_HTYPE_TRANS, &gtrid, &gtridlen,
                   OCI_ATTR_TRANS_NAME, errhp));
memcpy(gtridbuf, gtrid, gtridlen);
printf("Generated GTRID in hexadecimal is: ");
for (i = 0; i < gtridlen; ++i)
  printf("%02X", (unsigned int)gtridbuf[i]);
printf("\n");

/* Suspend Sessionless transaction and query dept table. We should not see
 * the inserted row. */
CHK_ERR(OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_DEFAULT));
query_dept("outside sessionless txn");

/* Set OCI_ATTR_TRANS to NULL. */
CHK_ERR(OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, NULL, 0, OCI_ATTR_TRANS,
errhp));

/* release the session here and get another session connected to the
 * same database. This sessions can be a different session and can connect
 * to a different instance in RAC configuration. */
CHK_ERR(OCISessionRelease(svchp, errhp, 0, 0, OCI_DEFAULT));
CHK_ERR(OCISessionGet(envhp, errhp, &svchp, NULL, pool_name,
```

```
                          pool_name_len, 0, 0, 0, 0, 0, OCI_SESSGET_SPOOL));

  /* resume the Sessionless transaction and query dept. We can see the
previous
   * change. */
  CHK_ERR(OCIAttrSet(txnhp, OCI_HTYPE_TRANS, gtrid, gtridlen,
                     OCI_ATTR_TRANS_NAME, errhp));
  CHK_ERR(
    OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, txnhp, 0, OCI_ATTR_TRANS, errhp));
  CHK_ERR(
    OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS |
OCI_TRANS_RESUME));
  query_dept("resumed sessionless txn");

  /* insert an additional row, and commit the Sessionless transaction. */
  insert_dept(51, "DEVELOPMENT2", "SAN FRANCISCO");
  CHK_ERR(OCITransCommit(svchp, errhp, OCI_DEFAULT));

  /* after commit, we can confirm no transaction is active in the session.
when
   * query the table, we can see both rows got inserted. */
  query_transaction_type("after commit");
  query_dept("after commit");

  /* tear down */
  {
    if (txnhp)
      CHK_ERR(OCIHandleFree(txnhp, OCI_HTYPE_TRANS));
    if (svchp)
    {
      CHK_ERR(OCISessionRelease(svchp, errhp, 0, 0, OCI_DEFAULT));
      svchp = NULL;
    }
    CHK_ERR(OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
    if (poolhp)
      OCIHandleFree((void *)poolhp, OCI_HTYPE_SPOOL);
  }

  free(dbname);
  free(password);
  free(username);
  return 0;
}

void insert_dept(int deptno, const char *dname, const char *loc)
{
  OraText dml[] = "INSERT INTO dept(deptno, dname, loc) values (:1, :2, :3)";
  CHK_ERR(OCIStmtPrepare2(svchp, &stmthp, errhp, dml, strlen((char *)dml), 0,
                          0, OCI_NTV_SYNTAX, OCI_DEFAULT));
  CHK_ERR(OCIBindByPos(stmthp, &bndhp[0], errhp, 1, &deptno, sizeof(deptno),
                       SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIBindByPos(stmthp, &bndhp[1], errhp, 2, (void *)dname,
                       strlen(dname) + 1, SQLT_STR, 0, 0, 0, 0, 0,
                       OCI_DEFAULT));
  CHK_ERR(OCIBindByPos(stmthp, &bndhp[1], errhp, 3, (void *)loc,
                       strlen(loc) + 1, SQLT_STR, 0, 0, 0, 0, 0,
```

```
OCI_DEFAULT));
  CHK_ERR(OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT));
}

void query_dept(const char *description)
{
  OraText query[] = "SELECT deptno, dname, loc FROM dept ORDER BY deptno";
  int     deptno;
  OraText dname[20];
  OraText loc[20];
  int     i = 1;
  int     ret;
  printf("DEPT table [%s]\n", description);
  CHK_ERR(OCIStmtPrepare2(svchp, &stmthp, errhp, query, strlen((char *)query),
                          0, 0, OCI_NTV_SYNTAX, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[0], errhp, 1, &deptno, sizeof(deptno),
                         SQLT_INT, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (void *)dname, 20,
                         SQLT_STR, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[2], errhp, 3, (void *)loc, 20,
                         SQLT_STR, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIStmtExecute(svchp, stmthp, errhp, 0, 0, 0, 0, OCI_DEFAULT));
  while ((ret = OCIStmtFetch2((OCIStmt *)stmthp, (OCIError *)errhp, (ub4)1,
                       (ub4)OCI_FETCH_NEXT, (sb4)0, (ub4)OCI_DEFAULT)) == 0)
  {
    printf("Row %d: (%d, %s, %s)\n", i++, deptno, dname, loc);
  }
  if (ret && ret != OCI_NO_DATA)
    CHK_ERR(ret);
  printf("\n");
  CHK_ERR(OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT));
}

void query_transaction_type(const char *description)
{
  OraText query[] =
    "SELECT NVL(TO_CHAR(DBMS_TRANSACTION.GET_TRANSACTION_TYPE), 'NULL') from "
    "dual";
  char txn_type[10];
  int  ret;
  printf("Query transaction type [%s]: ", description);
  CHK_ERR(OCIStmtPrepare2(svchp, &stmthp, errhp, query, strlen((char *)query),
                          0, 0, OCI_NTV_SYNTAX, OCI_DEFAULT));
  CHK_ERR(OCIDefineByPos(stmthp, &defhp[0], errhp, 1, &txn_type,
                         sizeof(txn_type), SQLT_STR, 0, 0, 0, OCI_DEFAULT));
  CHK_ERR(OCIStmtExecute(svchp, stmthp, errhp, 0, 0, 0, 0, OCI_DEFAULT));
  while (
    (ret = OCIStmtFetch2((OCIStmt *)stmthp, (OCIError *)errhp, (ub4)1,
                         (ub4)OCI_FETCH_NEXT, (sb4)0, (ub4)OCI_DEFAULT)) == 0)
    printf("%s\n", txn_type);

  if (ret && ret != OCI_NO_DATA)
    CHK_ERR(ret);
  printf("\n");
  CHK_ERR(OCIStmtRelease(stmthp, errhp, 0, 0, OCI_DEFAULT));
```

```
}

void chk_err(sword ret, OCIError *errhp, sb2 linenum)
{
  OraText msgbuf[1024];
  sb4     errcode;
  memset((void *)msgbuf, '\0', 1024);
  if (ret == OCI_ERROR || ret == OCI_SUCCESS_WITH_INFO)
    OCIErrorGet((dvoid *)errhp, (ub4)1, (OraText *)NULL, &errcode, msgbuf,
                (ub4)sizeof(msgbuf), (ub4)OCI_HTYPE_ERROR);

  if (ret == OCI_SUCCESS)
    return;

  printf("-------------------------------------------------------------\n");
  printf(":Line # %d: ", linenum);
  switch (ret)
  {
  case OCI_ERROR:
  case OCI_SUCCESS_WITH_INFO:
    printf(ret == OCI_SUCCESS_WITH_INFO ? "OCI_SUCCESS_WITH_INFO\n"
                                        : "OCI_ERROR\n");
    printf("%s\n", msgbuf);
    break;

  case OCI_NEED_DATA:
    printf("OCI_NEED_DATA\n");
    break;

  case OCI_NO_DATA:
    printf("OCI_NO_DATA\n");
    break;

  case OCI_INVALID_HANDLE:
    printf("OCI_INVALID_HANDLE\n");
    exit(1);

  case OCI_STILL_EXECUTING:
    printf("OCI_STILL_EXECUTING\n");
    break;

  case OCI_CONTINUE:
    printf("OCI_CONTINUE\n");
    break;

  default:
    printf("Error Code %d\n", ret);
    break;
  }
}
```

The output is as follows. Note that your output may have a different username, password, dbname (TNS name), and generated GTRID.

```
enter username:scott
enter password:tiger
enter dbname:cdb1_pdb1

DEPT table [before begin]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)

DEPT table [in sessionless txn, after insert]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
Row 5: (50, DEVELOPMENT1, SEATTLE)

Generated GTRID in hexadecimal is: 688330D5C1764F96BFF8267F58E5709E
DEPT table [outside sessionless txn]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)

DEPT table [resumed sessionless txn]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
Row 5: (50, DEVELOPMENT1, SEATTLE)

Query transaction type [after commit]: NULL

DEPT table [after commit]
Row 1: (10, ACCOUNTING, NEW YORK)
Row 2: (20, RESEARCH, DALLAS)
Row 3: (30, SALES, CHICAGO)
Row 4: (40, OPERATIONS, BOSTON)
Row 5: (50, DEVELOPMENT1, SEATTLE)
Row 6: (51, DEVELOPMENT2, SAN FRANCISCO)
```

31.5.12 Rules and Guidelines for Using Sessionless Transactions

These rules and guidelines apply to the OCI client API.

The Sessionless Transactions feature enforces the following rules and guidelines when using Sessionless transactions with the OCI client API.

Restrictions on Current Transaction Handle

When a Sessionless transaction is active, you are not allowed to change the `OCI_ATTR_TRANS` attribute of service context handle or free the transaction handle.

Rules for OCI Call Functions

The OCI call functions are distinguished as follows:

- Main call - this function runs immediately
- Pre-call - this function runs attached to the next server round trip, before the call
- Post-call - this function runs attached to the next server round trip, after the call

Last Pre-call Function Call Wins

For a single round trip, you can run a maximum of one pre-call function (which can be either start, resume, or suspend), and one post-call suspend. Before a server round trip, if there are multiple calls to start, resume, or pre-call suspend before the next server round trip is made, only the last call is sent to the database server.

Following is an example where only a pre-call resume [`OCITransStart(OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME)`] and a post-call suspend [`OCITransDetach(OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL)`] calls are sent to the server.

```
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW);
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_PRE_CALL); /*
this overrides the above OCITransStart call */

memcpy(xidp->data, "GTRID-2", strlen("GTRID-2"));
OCIAttrSet(txnhp, OCI_HTYPE_TRANS, xidp, sizeof(XID), OCI_ATTR_XID, errhp);
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME); /*
this overrides the above OCITransDetach(PRE_CALL) call */

OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL); /*
does NOT override anything since it's a post-call */
OCIStmtPrepare2(svchp, &stmthp, errhp, dml, strlen(dml), 0, 0,
OCI_NTV_SYNTAX, OCI_DEFAULT);
OCIBindByPos(…);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_DEFAULT);
                                    /* This is a server round trip that
attaches the pre-call resume and post-call suspend */
```

Start Transaction with Pre-call Suspend is Disallowed if a Post-call Suspend is Recorded

On the client, when post-call suspend is called and has not been attached to the next server round trip, you cannot request to start a new Sessionless transaction, resume an existing Sessionless transaction, or request to pre-call suspend a Sessionless transaction before the next server round trip. Make a server round trip (for example, call `OCIPing()`) before doing so. Following is an example:

```
OCITransDetach(svchp, errhp, OCI_TRANS_SESSIONLESS | OCI_TRANS_POST_CALL); /*
a post-call suspend */
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW); /*
```

```
this is prohibited and will throw 26215 error */


OCIPing(svchp, errhp, OCI_DEFAULT); /* This incurs a server round trip that
attaches post-call suspend */
OCITransStart(svchp, errhp, 60, OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW); /* no
error thrown*/
```

Request Boundaries Behavior

Calls that define request boundaries are `OCIRequestBegin`, `OCIRequestEnd`, and `OCISessionRelease`. Sessionless transactions that are active in a session with respect to these request boundaries currently follow the behavior that is applicable to local transactions in a session.

31.5.13 Error Messages and Notifications

The following table outlines the errors that you might see while using Sessionless transactions, the description of when these errors are displayed, and the next steps you can take to resolve these errors. However, this is not an exhaustive list of all possible errors.

Table 31-1 Errors Summary

Error Codes	Cause	Resolution
ORA-26202	Suspending a transaction that is not a Sessionless transaction (but a local transaction or an XA transaction branch)	Ensure that the transaction being suspended is a Sessionless transaction.
ORA-26204	The active Sessionless transaction was started using a GTRID generated by the client library, and the GTRID has not been retrieved by the user. Actions that suspend the active transaction make users unable to access the transaction forever and are disallowed.	Retrieve the GTRID before doing the action. To learn how to retrieve a GTRID, see the section "Retrieving a GTRID on the OCI client" in the Retrieving a Global Transaction ID section of this document.
ORA-26207	Starting or suspending a Sessionless transaction when the server does not support the Sessionless Transactions feature	Ensure that you are using Oracle Database 23ai, 23.6, and later versions.
ORA-26210	The `OCI_ATTR_TRANS` attribute cannot be altered when a Sessionless transaction that is started or resumed on the database server, is active. In such a case, a read-only transaction handle is created and set as service context handle's `OCI_ATTR_TRANS` attribute. Once this Sessionless transaction is no longer active, the original `OCI_ATTR_TRANS` attribute is placed back and can be altered.	Suspend or end the active Sessionless transaction that was started or resumed on the database server.

Table 31-1 (Cont.) Errors Summary

Error Codes	Cause	Resolution
ORA-26211	The method (on the database server or on the client) used to start the active Sessionless transaction is different from the method that is used for the current start or suspend call.	Use the same method that started the active Sessionless transaction to suspend the active transaction first.
ORA-26213	When calling start transaction functions, you did not specify (using the appropriate flag) that the call is meant to start a new Sessionless transaction or to resume an existing one.	In the flag, ensure that you specify whether you intend to start a new Sessionless transaction or resume an existing one. For `OCITransStart`, use the right flag: `OCI_TRANS_SESSIONLESS | OCI_TRANS_NEW` or `OCI_TRANS_SESSIONLESS | OCI_TRANS_RESUME`.
ORA-26214	When suspending the Sessionless transaction on the client, you did not specify whether the call is to be made immediately, attached to the server round trip, and run before the main call (pre-call suspend), or attached to the next server round trip, and run after the main call (post-call suspend).	When suspending a Sessionless transaction, in addition to `OCI_TRANS_SESSIONLESS`, ensure that either `OCI_DEFAULT`, `OCI_TRANS_PRE_CALL`, or `OCI_TRANS_POST_CALL` is specified in the `flag` parameter.
ORA-26215	A suspend is recorded to be run after the next server round trip. Before the suspend is done, starting a new or resuming an existing Sessionless transaction is not allowed. Also, a suspend that runs before the next server round trip is not allowed.	Incur a server round trip.
ORA-26216	A start new or resume call is recorded but not run yet. Therefore, `OCI_ATTR_XID` and `OCI_ATTR_TRANS_NAME` cannot be changed.	Incur a server round trip.
ORA-26217	Attempting to start a new transaction identified by a GTRID, but on the database server a Sessionless transaction identified by such a GTRID already exists	Use a unique GTRID to start a new Sessionless transaction.
ORA-26218	Attempting to resume an existing transaction identified by a GTRID, but on the database server such a Sessionless transaction does not exist	Ensure that the Sessionless transaction identified by the GTRID for the transaction you want to resume, actually exists.
ORA-26219	Attempting to change the `OCI_ATTR_TRANS` attribute of service context handle when a Sessionless transaction is active	`OCI_ATTR_TRANS` can be set only when no Sessionless transaction is active.

Table 31-1 (Cont.) Errors Summary

Error Codes	Cause	Resolution
ORA-26220	Attempting to free the transaction handle when a Sessionless transaction is active	Free the transaction handle only when no Sessionless transaction is active.

31.6 Sessionless Transactions and Oracle Coordinated Distributed Transaction Interoperability

Sessionless Transactions allows access to remote database objects using database links, either directly or indirectly. An Oracle Coordinated Distributed transaction starts when database links are used to access tables on a remote Oracle Database. If Sessionless transactions access remote database objects, Oracle Database seamlessly manages multi-level (Sessionless and Oracle-coordinated transactions) hierarchical distributed transactions during the 2PC protocol operations.

31.7 Restrictions for Sessionless Transactions

Promoting a Sessionless Transaction to an XA Transaction Is Not Supported

A local transaction can be promoted to an XA transaction. When that happens, the original local transaction becomes a transaction branch of the XA transaction. However, a Sessionless transaction cannot be promoted to an XA transaction in Oracle Database 23ai, Version 23.6. *This restriction will be addressed in the near future.*

Rollback to Savepoint Is Not Supported

Rollback to savepoint is not supported with Sessionless Transactions in Oracle Database 23ai, Version 23.6 under multi-instance RAC configuration. Under such configurations, if a Sessionless transaction savepoint is created in one session and the transaction is later resumed in another session, the savepoint cannot be used. When a rollback to savepoint is called, an ORA-1086 error is raised.

Session States Are Not Carried Across Sessions by Sessionless Transactions

When resuming a Sessionless transaction in a session that is different from the session the transaction was last suspended from, the session states (such as, all parameters set by `ALTER SESSION`, NLS settings, Temp LOB states, PL/SQL states, and any UGA state) are not carried over to the new session. Thus, if you suspend a Sessionless transaction in one session and resume it in another session, you should set up the session states properly. You must ensure that you re-establish the same session states to retrieve the same results (such as NLS settings).

Insert Direct Load, Online Direct Load, and Parallel DML Are Serialized

As of Oracle Database 23ai, Version 23.6, when Insert Direct Load (IDL) or Online Direct Load (ODL) operation is done in a Sessionless transaction, the operation falls back to the traditional insertion method. Likewise, when a Parallel DML (PDML) statement is issued in a Sessionless transaction, such DML runs in a serial manner. However, some of these restrictions will be removed in the upcoming releases.

Database Link Is Not Supported

Database Links (DBLink) are not compatible with Sessionless transactions as of today. This restriction is being addressed.