13
Advanced Design Considerations

This section discusses the design considerations for more advanced application development issues.

- Read-Consistent Locators
  Oracle Database provides the same read consistency mechanisms for LOBs as for all other database reads and updates of scalar quantities.
- LOB Locators and Transaction Boundaries
  LOB locators can be used in both transactions as well as transaction IDs.
- LOBs in the Object Cache
  When you copy one object to another in the object cache with a LOB locator attribute, only the LOB locator is copied.
- Guidelines for Creating Terabyte sized LOBs
  To create terabyte LOBs in supported environments, use the following guidelines to make use of all available space in the tablespace for LOB storage.

13.1 Read-Consistent Locators

Oracle Database provides the same read consistency mechanisms for LOBs as for all other database reads and updates of scalar quantities.

Read consistency has some special applications to LOB locators that you must understand. The following sections discuss read consistency and include examples which should be looked at in relationship to each other.

- A Selected Locator Becomes a Read-Consistent Locator
  A read-consistent locator contains the snapshot environment as of the point in time of the `SELECT` operation.
- Example of Updating LOBs and Read-Consistency
  Read-consistent locators provide the same LOB value regardless of when the `SELECT` occurs. The following example demonstrates the relationship between read-consistency and `UPDATE` operation.
- Example of Updating LOBs Through Updated Locators
  Learn about updating LOBs through Locators in this section.
- Example of Updating a LOB Using SQL DML and DBMS_LOB
  Using the `print_media` table in the following example, a `CLOB` locator is created as `clob_selected`.
- Example of Using One Locator to Update the Same LOB Value
  You may avoid many pitfalls if you use only one locator to update a given LOB value. Learn about it in this section.
- Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable
  Learn about updating a LOB with a PL/SQL bind variable in this section.
- Example of Deleting a LOB Using Locator
  Learn about deleting a LOB with a PL/SQL bind variable in this section.

- Ensuring Read Consistency
  This script in this section can be used to ensure that hot backups can be taken of tables that have `NOLOGGING` or `FILESYSTEM_LIKE_LOGGING` LOBs and have a known recovery point without read inconsistencies.

> **See Also:**
>
> - *Oracle Database Concepts* for general information about read consistency

13.1.1 A Selected Locator Becomes a Read-Consistent Locator

A read-consistent locator contains the snapshot environment as of the point in time of the `SELECT` operation.

A selected locator, regardless of the existence of the `FOR UPDATE` clause, becomes a *read-consistent locator,* and remains a read-consistent locator until the LOB value is updated through that locator.

This has some complex implications. Suppose you have created a read-consistent locator (`L1`) by way of a `SELECT` operation. In reading the value of the persistent LOB through `L1`, note the following:

- The LOB is read as of the point in time of the `SELECT` statement even if the `SELECT` statement includes a `FOR UPDATE`.
- If the LOB value is updated through a different locator (`L2`) in the same transaction, then `L1` does not see the `L2` updates.
- `L1` does not see committed updates made to the LOB through another transaction.
- If the read-consistent locator `L1` is copied to another locator `L2` (for example, by a PL/SQL assignment of two locator variables — `L2:= L1`), then `L2` becomes a read-consistent locator along with `L1` and any data read is read as of the point in time of the `SELECT` for `L1`.

You can use the existence of multiple locators to access different transformations of the LOB value. However, in doing so, you must keep track of the different values accessed by different locators.

13.1.2 Example of Updating LOBs and Read-Consistency

Read-consistent locators provide the same LOB value regardless of when the `SELECT` occurs. The following example demonstrates the relationship between read-consistency and `UPDATE` operation.

Using the `print_media` table and PL/SQL, three `CLOB` instances are created as potential locators: `clob_selected`, `clob_update`, and `clob_copied`.

Observe these progressions in the code, from times `t1` through `t6`:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_selected`.
- In the second operation (at `t2`), the value in `ad_sourcetext` is associated with the locator `clob_updated`. Because there has been no change in the value of `ad_sourcetext` between `t1` and `t2`, both `clob_selected` and `clob_updated` are read-consistent locators that

effectively have the same value even though they reflect snapshots taken at different moments in time.

- The third operation (at `t3`) copies the value in `clob_selected` to `clob_copied`. At this juncture, all three locators see the same value. The example demonstrates this with a series of `DBMS_LOB.READ()` calls.
- At time `t4`, the program uses `DBMS_LOB.WRITE()` to alter the value in `clob_updated`, and a `DBMS_LOB.READ()` reveals a new value.
- However, a `DBMS_LOB.READ()` of the value through `clob_selected` (at `t5`) reveals that it is a read-consistent locator, continuing to refer to the same value as of the time of its `SELECT`.
- Likewise, a `DBMS_LOB.READ()` of the value through `clob_copied` (at `t6`) reveals that it is a read-consistent locator, continuing to refer to the same value as `clob_selected`.

Example 13-1

```
INSERT INTO print_media VALUES (2056, 20020, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);

BEGIN
  -- At time t1:
  SELECT ad_sourcetext INTO clob_selected
     FROM Print_media
     WHERE ad_id = 20020;

  -- At time t2:
  SELECT ad_sourcetext INTO clob_updated
     FROM Print_media
     WHERE ad_id = 20020
     FOR UPDATE;

  -- At time t3:
  clob_copied := clob_selected;
  -- After the assignment, both the clob_copied and the
  -- clob_selected have the same snapshot as of the point in time
  -- of the SELECT into clob_selected

  -- Reading from the clob_selected and the clob_copied does
  -- return the same LOB value. clob_updated also sees the same
  -- LOB value as of its select:
  read_amount := 10;
  read_offset := 1;
  DBMS_LOB.READ(clob_selected, read_amount, read_offset, buffer);
  DBMS_OUTPUT.PUT_LINE('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  DBMS_LOB.READ(clob_copied, read_amount, read_offset, buffer);
```

```
  DBMS_OUTPUT.PUT_LINE('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  DBMS_LOB.READ(clob_updated, read_amount, read_offset, buffer);
  DBMS_OUTPUT.PUT_LINE('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t4:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  DBMS_LOB.WRITE(clob_updated, write_amount, write_offset, buffer);

  read_amount := 10;
  DBMS_LOB.READ(clob_updated, read_amount, read_offset, buffer);
  DBMS_OUTPUT.PUT_LINE('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'

  -- At time t5:
  read_amount := 10;
  DBMS_LOB.READ(clob_selected, read_amount, read_offset, buffer);
  DBMS_OUTPUT.PUT_LINE('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t6:
  read_amount := 10;
  DBMS_LOB.READ(clob_copied, read_amount, read_offset, buffer);
  DBMS_OUTPUT.PUT_LINE('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

13.1.3 Example of Updating LOBs Through Updated Locators

Learn about updating LOBs through Locators in this section.

When you update the value of the persistent LOB through the LOB locator (`L1`), `L1` is updated to contain the current snapshot environment.

This snapshot is as of the time after the operation was completed on the LOB value through locator `L1`. `L1` is then termed an updated locator. This operation enables you to see your own changes to the LOB value on the next read through the same locator, `L1`.

> **Note:**
>
> The snapshot environment in the locator is *not* updated if the locator is used to merely read the LOB value. It is only updated when you modify the LOB value through the locator using the PL/SQL `DBMS_LOB` package or the OCI LOB APIs.

Any committed updates made by a different transaction are seen by L1 only if your transaction is a read-committed transaction and if you use L1 to update the LOB value after the other transaction committed.

> **Note:**
>
> When you update a persistent LOB value, the modification is always made to the most current LOB value.

Updating the value of the persistent LOB through any of the available methods, such as OCI LOB APIs or PL/SQL `DBMS_LOB` package, updates the LOB value *and then reselects* the locator that refers to the new LOB value.

> **Note:**
>
> Once you have selected out a LOB locator by whatever means, you can read from the locator but not write into it.
>
> Note that updating the LOB value through SQL is merely an `UPDATE` statement. It is up to you to do the reselect of the LOB locator or use the `RETURNING` clause in the `UPDATE` statement so that the locator can see the changes made by the `UPDATE` statement. Unless you reselect the LOB locator or use the `RETURNING` clause, you may think you are reading the latest value when this is not the case. For this reason you should avoid mixing SQL DML with `OCI` and `DBMS_LOB` piecewise operations.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference*

13.1.4 Example of Updating a LOB Using SQL DML and DBMS_LOB

Using the `print_media` table in the following example, a `CLOB` locator is created as `clob_selected`.

Note the following progressions in the example, from times `t1` through `t3`:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_selected`.
- In the second operation (at `t2`), the value in `ad_sourcetext` is modified through the `SQL UPDATE` statement, without affecting the `clob_selected` locator. The locator still sees the value of the LOB as of the point in time of the original `SELECT`. In other words, the locator does not see the update made using the SQL `UPDATE` statement. This is illustrated by the subsequent `DBMS_LOB.READ()` call.
- The third operation (at `t3`) re-selects the LOB value into the locator `clob_selected`. The locator is thus updated with the latest snapshot environment which allows the locator to see the change made by the previous SQL `UPDATE` statement. Therefore, in the next `DBMS_LOB.READ()`, an error is returned because the LOB value is empty, that is, it does not contain any data.

```
INSERT INTO Print_media VALUES (3247, 20010, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  buffer            VARCHAR2(20);

BEGIN

  -- At time t1:
  SELECT ad_sourcetext INTO clob_selected
  FROM Print_media
  WHERE ad_id = 20010;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t2:
  UPDATE Print_media SET ad_sourcetext = empty_clob()
      WHERE ad_id = 20010;
  -- although the most current LOB value is now empty,
  -- clob_selected still sees the LOB value as of the point
  -- in time of the SELECT

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  SELECT ad_sourcetext INTO clob_selected FROM Print_media WHERE
       ad_id = 20010;
  -- the SELECT allows clob_selected to see the most current
  -- LOB value

  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  -- ERROR: ORA-01403: no data found
END;
/
```

13.1.5 Example of Using One Locator to Update the Same LOB Value

You may avoid many pitfalls if you use only one locator to update a given LOB value. Learn about it in this section.

> **Note:**
>
> Avoid updating the same LOB with different locators.

In the following example, using table `print_media`, two `CLOB`s are created as potential locators: `clob_updated` and `clob_copied`.

Note these progressions in the example at times t1 through t5:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_updated`.
- The second operation (at time `t2`) copies the value in `clob_updated` to `clob_copied`. At this time, both locators see the same value. The example demonstrates this with a series of `DBMS_LOB.READ()` calls.
- At time `t3`, the program uses `DBMS_LOB.WRITE()` to alter the value in `clob_updated`, and a `DBMS_LOB.READ()` reveals a new value.
- However, a `DBMS_LOB.READ()` of the value through `clob_copied` (at time `t4`) reveals that it still sees the value of the LOB as of the point in time of the assignment from `clob_updated` (at `t2`).
- It is not until `clob_updated` is assigned to `clob_copied` (`t5`) that `clob_copied` sees the modification made by `clob_updated`.

```
INSERT INTO PRINT_MEDIA VALUES (2049, 20030, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var          INTEGER;
  clob_updated     CLOB;
  clob_copied      CLOB;
  read_amount      INTEGER;
  read_offset      INTEGER;
  write_amount     INTEGER;
  write_offset     INTEGER;
  buffer           VARCHAR2(20);
BEGIN

-- At time t1:
  SELECT ad_sourcetext INTO clob_updated FROM PRINT_MEDIA
      WHERE ad_id = 20030
      FOR UPDATE;

  -- At time t2:
  clob_copied := clob_updated;
  -- after the assign, clob_copied and clob_updated see the same
  -- LOB value

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  write_amount := 3;
  write_offset := 5;
```

```
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset,
        buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t5:
  clob_copied := clob_updated;

  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcdefg'
END;
/
```

13.1.6 Example of Updating a LOB with a PL/SQL (DBMS_LOB) Bind Variable

Learn about updating a LOB with a PL/SQL bind variable in this section.

When a LOB locator is used as the source to update another persistent LOB (as in a SQL `INSERT` or `UPDATE` statement, the `DBMS_LOB.COPY` routine, and so on), the snapshot environment in the source LOB locator determines the LOB value that is used as the source.

If the source locator (for example `L1`) is a read-consistent locator, then the LOB value as of the time of the `SELECT` of `L1` is used. If the source locator (for example `L2`) is an updated locator, then the LOB value associated with the `L2` snapshot environment at the time of the operation is used.

In the following example, three `CLOB`s are created as potential locators: `clob_selected`, clob_updated, and clob_copied.

Note these progressions in the example at times `t1` through `t5`:

- At the time of the first `SELECT INTO` (at `t1`), the value in `ad_sourcetext` is associated with the locator `clob_updated`.
- The second operation (at `t2`) copies the value in `clob_updated` to `clob_copied`. At this juncture, both locators see the same value.
- Then (at `t3`), the program uses `DBMS_LOB.WRITE()` to alter the value in `clob_updated`, and a `DBMS_LOB.READ()` reveals a new value.
- However, a `DBMS_LOB.READ()` of the value through `clob_copied` (at `t4`) reveals that `clob_copied` does not see the change made by `clob_updated`.
- Therefore (at `t5`), when `clob_copied` is used as the source for the value of the `INSERT` statement, the value associated with `clob_copied` (for example, without the new changes

made by `clob_updated`) is inserted. This is demonstrated by the subsequent `DBMS_LOB.READ()` of the value just inserted.

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20020, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_selected     CLOB;
  clob_updated      CLOB;
  clob_copied       CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);
BEGIN

  -- At time t1:
  SELECT ad_sourcetext INTO clob_updated FROM PRINT_MEDIA
      WHERE ad_id = 20020
      FOR UPDATE;

  read_amount := 10;
  read_offset := 1;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t2:
  clob_copied := clob_updated;


  -- At time t3:
  write_amount := 3;
  write_offset := 5;
  buffer := 'efg';
  dbms_lob.write(clob_updated, write_amount, write_offset, buffer);

  read_amount := 10;
  dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_updated value: ' || buffer);
  -- Produces the output 'abcdefg'
  -- note that clob_copied does not see the write made before
  -- clob_updated


  -- At time t4:
  read_amount := 10;
  dbms_lob.read(clob_copied, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_copied value: ' || buffer);
  -- Produces the output 'abcd'

  -- At time t5:
  -- the insert uses clob_copied view of the LOB value which does
  -- not include clob_updated changes
  INSERT INTO PRINT_MEDIA VALUES (2056, 20022, EMPTY_BLOB(),
    clob_copied, EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL)
    RETURNING ad_sourcetext INTO clob_selected;
```

```
  read_amount := 10;
  dbms_lob.read(clob_selected, read_amount, read_offset, buffer);
  dbms_output.put_line('clob_selected value: ' || buffer);
  -- Produces the output 'abcd'
END;
/
```

13.1.7 Example of Deleting a LOB Using Locator

Learn about deleting a LOB with a PL/SQL bind variable in this section.

The following example illustrates that LOB content through a locator selected at a given point of time is available even though the LOB is deleted in the same transaction.

In the following example, using table `print_media`, two CLOBs are created as potential locators:`clob_selected` and `clob_copied`.

Note these progressions in the example at times `t1` through `t3`:

- At the time of the first `SELECT INTO` (at `t1`), the value in`ad_sourcetext` for `ad_id` value 20020 is associated with the locator `clob_selected`. The value in `ad_sourcetext` for `ad_id` value 20021 is associated with the locator `clob_copied`.
- The second operation (at `t2`) deletes the row with `ad_id` value 20020. However, a `DBMS_LOB.READ()` of the value through `clob_selected` (at `t1`) reveals that it is a read-consistent locator, continuing to refer to the same value as of the time of its SELECT.
- The third operation (at `t3`), copies the LOB data read through `clob_selected` into the LOB `clob_copied`. `DBMS_LOB.READ()` of the value through `clob_selected` and `clob_copied` are now the same and refer to the same value as of the time of `SELECT` of `clob_selected`.

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20020, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

INSERT INTO PRINT_MEDIA VALUES (2057, 20021, EMPTY_BLOB(),
    'cdef', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

DECLARE
  clob_selected CLOB;
  clob_copied CLOB;
  buffer VARCHAR2(20);
  read_amount INTEGER := 20;
  read_offset INTEGER := 1;

BEGIN
  -- At time t1:
  SELECT ad_sourcetext INTO clob_selected
      FROM PRINT_MEDIA
      WHERE ad_id = 20020
      FOR UPDATE;

  SELECT ad_sourcetext INTO clob_copied
      FROM PRINT_MEDIA
      WHERE ad_id = 20021
      FOR UPDATE;

  dbms_lob.read(clob_selected, read_amount, read_offset,buffer);
  dbms_output.put_line(buffer);
  -- Produces the output 'abcd'

  dbms_lob.read(clob_copied, read_amount, read_offset,buffer);
```

```
  dbms_output.put_line(buffer);
  -- Produces the output 'cdef'

  -- At time t2: Delete the CLOB associated with clob_selected
  DELETE FROM PRINT_MEDIA WHERE ad_id = 20020;

  dbms_lob.read(clob_selected, read_amount, read_offset,buffer);
  dbms_output.put_line(buffer);
  -- Produces the output 'abcd'

  -- At time t3:
  -- Copy using clob_selected
  dbms_lob.copy(clob_copied, clob_selected, 4000, 1, 1);
  dbms_lob.read(clob_copied, read_amount, read_offset,buffer);
  dbms_output.put_line(buffer);
  -- Produces the output 'abcd'

END;
/
```

13.1.8 Ensuring Read Consistency

This script in this section can be used to ensure that hot backups can be taken of tables that have `NOLOGGING` or `FILESYSTEM_LIKE_LOGGING` LOBs and have a known recovery point without read inconsistencies.

```
ALTER DATABASE FORCE LOGGING;
SELECT CHECKPOINT_CHANGE# FROM V$DATABASE;  --Start SCN
```

SCN (System Change Number) is a stamp that defines a version of the database at the time that a transaction is committed.

Perform the backup.

Run the next script:

```
ALTER SYSTEM CHECKPOINT GLOBAL;
SELECT CHECKPOINT_CHANGE# FROM V$DATABASE;  --End SCN
ALTER DATABASE NO FORCE LOGGING;
```

Back up the archive logs generated by the database. At the minimum, archive logs between start SCN and end SCN (including both SCN points) must be backed up.

To restore to a point with no read inconsistency, restore to end SCN as your incomplete recovery point. If recovery is done to an SCN after end SCN, there can be read inconsistency in the `NOLOGGING` LOBs.

For SecureFiles, if a read inconsistency is found during media recovery, the database treats the inconsistent blocks as holes and fills `BLOB`s with `0`'s and `CLOB`s with fill characters.

13.2 LOB Locators and Transaction Boundaries

LOB locators can be used in both transactions as well as transaction IDs.

- About LOB Locators and Transaction Boundaries
  Learn about LOB locators and transaction boundaries in this section.

- Read and Write Operations on a LOB Using Locators
  You can always read LOB data using the locator irrespective of whether or not the locator contains a transaction ID. Learn about various aspects of it in this section.
- Selecting the Locator Outside of the Transaction Boundary
  This section has two scenarios that describe techniques for using locators in non-serializable transactions when the locator is selected outside of a transaction.
- Selecting the Locator Within a Transaction Boundary
  This section has two scenarios that describe techniques for using locators in non-serializable transactions when the locator is selected within a transaction.
- LOB Locators Cannot Span Transactions
  LOB locators that are used to write data cannot span transactions. However, the locator can be used to read the LOB value unless you are in a serializable transaction.
- Example of Locator Not Spanning a Transaction
  The example of locator not spanning a transaction uses the `print_media` table.

> **See Also:**
>
> Locator Interface for LOBs for more information about LOB locators

13.2.1 About LOB Locators and Transaction Boundaries

Learn about LOB locators and transaction boundaries in this section.

Note the following regarding LOB locators and transactions:

- Locators contain transaction IDs when:

  You Begin the Transaction, Then Select Locator: If you begin a transaction and subsequently select a locator, then the locator contains the transaction ID. Note that you can implicitly be in a transaction without explicitly beginning one. For example, `SELECT... FOR UPDATE` implicitly begins a transaction. In such a case, the locator contains a transaction ID.
- Locators Do Not Contain Transaction IDs When...
  - You are Outside the Transaction, Then Select Locator: By contrast, if you select a locator outside of a transaction, then the locator does not contain a transaction ID.
  - When Selected Prior to DML Statement Execution: A transaction ID is not assigned until the first DML statement executes. Therefore, locators that are selected prior to such a DML statement do not contain a transaction ID.

13.2.2 Read and Write Operations on a LOB Using Locators

You can always read LOB data using the locator irrespective of whether or not the locator contains a transaction ID. Learn about various aspects of it in this section.

- Cannot Write Using Locator:

  If the locator contains a transaction ID, then you cannot write to the LOB outside of that particular transaction.
- Can Write Using Locator:

If the locator *does not* contain a transaction ID, then you can write to the LOB after beginning a transaction either explicitly or implicitly.

- Cannot Read or Write Using Locator With Serializable Transactions:

  If the locator contains a transaction ID of an older transaction, and the current transaction is serializable, then you cannot read or write using that locator.

- Can Read, Not Write Using Locator With Non-Serializable Transactions:

  If the transaction is non-serializable, then you can read, but not write outside of that transaction.

The examples Selecting the Locator Outside of the Transaction Boundary, Selecting the Locator Within a Transaction Boundary, LOB Locators Cannot Span Transactions, and Example of Locator Not Spanning a Transaction show the relationship between locators and *non-serializable* transactions

13.2.3 Selecting the Locator Outside of the Transaction Boundary

This section has two scenarios that describe techniques for using locators in non-serializable transactions when the locator is selected outside of a transaction.

First Scenario:

1. Select the locator with no current transaction. At this point, the locator does not contain a transaction id.
2. Begin the transaction.
3. Use the locator to read data from the LOB.
4. Commit or rollback the transaction.
5. Use the locator to read data from the LOB.
6. Begin a transaction. The locator does not contain a transaction id.
7. Use the locator to write data to the LOB. This operation is valid because the locator did not contain a transaction id prior to the write. After this call, the locator contains a transaction id.

Second Scenario:

1. Select the locator with no current transaction. At this point, the locator does not contain a transaction id.
2. Begin the transaction. The locator does not contain a transaction id.
3. Use the locator to read data from the LOB. The locator does not contain a transaction id.
4. Use the locator to write data to the LOB. This operation is valid because the locator did not contain a transaction id prior to the write. After this call, the locator contains a transaction id. You can continue to read from or write to the LOB.
5. Commit or rollback the transaction. The locator continues to contain the transaction id.
6. Use the locator to read data from the LOB. This is a valid operation.
7. Begin a transaction. The locator contains the previous transaction id.
8. Use the locator to write data to the LOB. This write operation fails because the locator does not contain the transaction id that matches the current transaction.

13.2.4 Selecting the Locator Within a Transaction Boundary

This section has two scenarios that describe techniques for using locators in non-serializable transactions when the locator is selected within a transaction.

First Scenario:

1. Select the locator within a transaction. At this point, the locator contains the transaction id.
2. Begin the transaction. The locator contains the previous transaction id.
3. Use the locator to read data from the LOB. This operation is valid even though the transaction id in the locator does not match the current transaction.

> **See Also:**
>
> "Read-Consistent Locators" for more information about using the locator to read LOB data.

4. Use the locator to write data to the LOB. This operation fails because the transaction id in the locator does not match the current transaction.

Second Scenario:

1. Begin a transaction.
2. Select the locator. The locator contains the transaction id because it was selected within a transaction.
3. Use the locator to read from or write to the LOB. These operations are valid.
4. Commit or rollback the transaction. The locator continues to contain the transaction id.
5. Use the locator to read data from the LOB. This operation is valid even though there is a transaction id in the locator and the transaction was previously committed or rolled back.
6. Use the locator to write data to the LOB. This operation fails because the transaction id in the locator is for a transaction that was previously committed or rolled back.

13.2.5 LOB Locators Cannot Span Transactions

LOB locators that are used to write data cannot span transactions. However, the locator can be used to read the LOB value unless you are in a serializable transaction.

Modifying a persistent LOB value through the LOB locator using `DBMS_LOB`, OCI, or SQL `INSERT` or `UPDATE` statements changes the locator from a read-consistent locator to an updated locator.

The `INSERT` or `UPDATE` statement automatically starts a transaction and locks the row. Once this has occurred, the locator cannot be used outside the current transaction to modify the LOB value. In other words, LOB locators that are used to write data cannot span transactions. However, the locator can be used to read the LOB value unless you are in a serializable transaction.

In the following code example, a `CLOB` locator called `clob_updated` is created and following operations are performed:

- At the time of the first `SELECT INTO` (at t1), the value in `ad_sourcetext` is associated with the locator `clob_updated`.
- The second operation (at t2), uses the `DBMS_LOB.WRITE` function to alter the value in `clob_updated`, and a `DBMS_LOB.READ` reveals a new value.
- The `commit` statement (at t3) ends the current transaction.
- Therefore (at t4), the subsequent `DBMS_LOB.WRITE` operation fails because the `clob_updated` locator refers to a different (already committed) transaction. This is noted by the error returned. You must re-select the LOB locator before using it in further `DBMS_LOB` (and OCI) modify operations.

13.2.6 Example of Locator Not Spanning a Transaction

The example of locator not spanning a transaction uses the `print_media` table.

```
INSERT INTO PRINT_MEDIA VALUES (2056, 20010, EMPTY_BLOB(),
    'abcd', EMPTY_CLOB(), EMPTY_CLOB(), NULL, NULL, NULL, NULL);

COMMIT;

DECLARE
  num_var           INTEGER;
  clob_updated      CLOB;
  read_amount       INTEGER;
  read_offset       INTEGER;
  write_amount      INTEGER;
  write_offset      INTEGER;
  buffer            VARCHAR2(20);

BEGIN
          -- At time t1:
     SELECT      ad_sourcetext
     INTO        clob_updated
     FROM        PRINT_MEDIA
     WHERE       ad_id = 20010
     FOR UPDATE;
     read_amount := 10;
     read_offset := 1;
     dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcd'

     -- At time t2:
     write_amount := 3;
     write_offset := 5;
     buffer := 'efg';
     dbms_lob.write(clob_updated, write_amount, write_offset, buffer);
     read_amount := 10;
     dbms_lob.read(clob_updated, read_amount, read_offset, buffer);
     dbms_output.put_line('clob_updated value: ' || buffer);
     -- This produces the output 'abcdefg'

    -- At time t3:
    COMMIT;

     -- At time t4:
    dbms_lob.write(clob_updated , write_amount, write_offset, buffer);
    -- ERROR: ORA-22990: LOB locators cannot span transactions
```

```
END;
/
```

13.3 LOBs in the Object Cache

When you copy one object to another in the object cache with a LOB locator attribute, only the LOB locator is copied.

This means that the LOB attribute in these two different objects contain exactly the same locator that refers to *one and the same* LOB *value*. Only when you flush the target LOB, a separate physical copy of the LOB value is made, which is distinct from the source LOB value.

> **See Also:**
>
> Example of Updating LOBs and Read-Consistency for a description of what version of the LOB value is seen by each object if a write operation is performed through one of the locators.

Therefore, in cases where you want to modify the LOB that was the target of the copy, *you must flush the target object, refresh the target object, and then* write to the LOB through the locator attribute.

Consider the following object cache issues for LOB and BFILE attributes:

- Persistent LOB attributes: Creating an object in the object cache, sets the LOB attribute to empty.

  When you create an object in the object cache that contains a persistent LOB attribute, the LOB attribute is implicitly set to empty. You may not use this empty LOB locator to write data to the LOB. You must first flush the object, thereby inserting a row into the table and creating an empty LOB, that is, a LOB with zero (0) length. Once you refresh the object in the object cache, using the `OCI_PIN_LATEST` function, the real LOB locator is read into the attribute, and you can then call the OCI LOB APIs to write data to the LOB.

- `BFILE` attributes: Creating an object in the object cache, sets the `BFILE` attribute to `NULL`.

  When creating an object with a `BFILE` attribute, the `BFILE` is set to `NULL`. You must update it with a valid `DIRECTORY` object name and file name before reading from the `BFILE`.

13.4 Guidelines for Creating Terabyte sized LOBs

To create terabyte LOBs in supported environments, use the following guidelines to make use of all available space in the tablespace for LOB storage.

- Single Data File Size Restrictions:

  There are restrictions on the size of a single data file for each operating system. Hence, add more data files to the tablespace when the LOB grows larger than the maximum allowed file size of the operating system on which your Oracle Database runs.

- Set MAXEXTENTS to a Suitable Value or UNLIMITED:

  The `MAXEXTENTS` parameter limits the number of extents allowed for the LOB column. A large number of extents are created incrementally as the LOB size grows. Therefore, the parameter should be set to a value that is large enough to hold all the LOBs for the column. Alternatively, you could set it to `UNLIMITED`.

- Use a Large Extent Size:

  For every new extent created, Oracle generates undo information for the header and other metadata for the extent. If the number of extents is large, then the rollback segment can be saturated. To get around this, choose a large extent size, say 100 megabytes, to reduce the frequency of extent creation, or commit the transaction more often to reuse the space in the rollback segment.

- Creating a Tablespace and Table to Store Terabyte LOBs
  The following example illustrates how to create a tablespace and table to store terabyte LOBs.

13.4.1 Creating a Tablespace and Table to Store Terabyte LOBs

The following example illustrates how to create a tablespace and table to store terabyte LOBs.

```
CREATE TABLESPACE lobtbs1 DATAFILE '/your/own/data/directory/lobtbs_1.dat'
SIZE 2000M REUSE ONLINE NOLOGGING DEFAULT STORAGE (MAXEXTENTS UNLIMITED);
ALTER TABLESPACE lobtbs1 ADD DATAFILE
'/your/own/data/directory/lobtbs_2.dat' SIZE 2000M REUSE;

CREATE TABLE print_media_backup
  (product_id NUMBER(6),
   ad_id NUMBER(6),
   ad_composite BLOB,
   ad_sourcetext CLOB,
   ad_finaltext CLOB,
   ad_fltextn NCLOB,
   ad_textdocs_ntab textdoc_tab,
   ad_photo BLOB,
   ad_graphic BLOB,
   ad_header adheader_typ)
   NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab5
   LOB(ad_sourcetext) STORE AS (TABLESPACE lobtbs1 CHUNK 32768 PCTVERSION 0
                                NOCACHE NOLOGGING
                                STORAGE(INITIAL 1000M NEXT 1000M MAXEXTENTS
                                UNLIMITED));
```