12

Persistent LOBs: Advanced DDL

This chapter describes advanced LOB DDL features to make your application more scalable.

> **Note:**
>
> Unless otherwise stated, all features in this chapter apply to both SecureFile and Basicfile LOBs. However, Oracle strongly recommends SecureFiles for storing and managing LOBs.

- Creating a New LOB Column
  You can provide the LOB storage characteristics when creating a LOB column using the `CREATE TABLE` statement or the `ALTER TABLE ADD COLUMN` statement.
- Altering an Existing LOB Column
  You can use the `ALTER TABLE` statement to change the storage characteristics of a LOB column.
- Creating an Index on LOB Column
  The contents of a LOB are often specific to the application, so an index on the LOB column will usually deal with application logic. You can create a function-based or a domain index on a LOB column to improve the performance of queries accessing data stored in LOB columns. You cannot build a B-tree or bitmap index on a LOB column.
- LOBs in Partitioned Tables
  Partitioning can simplify the manageability of large database objects. This section discusses various aspects of LOBs in partitioned tables.
- LOBs in Index Organized Tables
  Index Organized Tables (IOTs) support LOB and BFILE columns.

12.1 Creating a New LOB Column

You can provide the LOB storage characteristics when creating a LOB column using the `CREATE TABLE` statement or the `ALTER TABLE ADD COLUMN` statement.

For most users, default values for these storage characteristics are sufficient. However, if you want to fine-tune LOB storage, then consider the guidelines discussed in this section.

When defining LOBs in a table, you can explicitly indicate the tablespace and storage characteristics for each persistent LOB column. It is common to use separate tablespaces for large LOBs. SecureFiles is the default storage for LOBs, so the `SECUREFILE` keyword is optional, but is shown for clarity in the following example. The example assumes that `TABLESPACE lobtbs1` is managed with ASSM, because SecureFile LOBs can only be created in tablespaces managed with Automatic Segment Space Management (ASSM).:

```
CREATE TABLE lobtab1 (n NUMBER, c CLOB)
      lob (c) STORE AS SECUREFILE sfsegname
      ( TABLESPACE lobtbs1
        ENABLE STORAGE IN ROW
```

```
  CACHE LOGGING
  RETENTION AUTO
  COMPRESS
  STORAGE (MAXEXTENTS 5)
);
```

To create a BasicFiles LOB, replace the `SECUREFILE` keyword with the `BASICFILE` keyword in the preceding example, and remove the `COMPRESS` keyword, which is specific to SecureFiles.

The data dictionary views `USER_LOBS`, `ALL_LOBS`, and `DBA_LOBS` provide information specific to a LOB column.

> **Note:**
>
> Oracle recommends Securefile LOBs for storing persistent LOBs, so this chapter focuses only on Securefile storage. All mentions of *LOBs* in the persistent LOB context is for Securefile LOBs, unless mentioned otherwise.

> **Note:**
>
> There are no tablespace or storage characteristics that you can specify for `BFILE`s as they are not stored in the database.

Assigning a LOB Data Segment Name

As shown in the previous example, specifying a name for the LOB data segment (`sfsegname` in the example) makes for a much more intuitive working environment. When querying the LOB data dictionary views `USER_LOBS`, `ALL_LOBS`, and `DBA_LOBS`, you see the LOB data segment that you chose instead of system-generated names.

- CREATE TABLE BNF
  The `CREATE TABLE` statement works with LOB storage using parameters that are specific to SecureFiles, BasicFiles LOB storage, or both.
- ENABLE or DISABLE STORAGE IN ROW
  LOB columns store locators that reference the location of the actual LOB value. This section describes how to enable or disable storage in a table row.
- CACHE, NOCACHE, and CACHE READS
  This section discusses the guidelines to follow while creating tables that contain LOBs.
- LOGGING and FILESYSTEM_LIKE_LOGGING
  You can apply the `LOGGING` parameter to LOBs in the same manner as you apply it for other table operations.
- The RETENTION Parameter
  The `RETENTION` parameter for SecureFile LOBs specifies how the database manages the old versions of the LOB data blocks.
- SecureFiles Compression, Deduplication, and Encryption
  In addition to the features supported by BasicFiles, SecureFiles LOB storage supports the following three features that are not available with the BasicFiles LOB storage option: compression, deduplication, and encryption.

- BasicFile Specific Parameters
  This section discusses the storage parameters specific to BasicFiles.
- Restriction on First Extent of a LOB Segment
  This section discusses the first extent requirements on SecureFiles and BasicFiles.
- Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs
  The table in this section summarizes the parameters of the `CREATE TABLE` statement that relate to Securefile LOB storage.

12.1.1 CREATE TABLE BNF

The `CREATE TABLE` statement works with LOB storage using parameters that are specific to SecureFiles, BasicFiles LOB storage, or both.

The following is the syntax for `CREATE TABLE` in Backus Naur (BNF) notation, parts of which have been simplified to keep the focus on LOB-specific parameters.

> **See Also:**
>
> - *Oracle Database SQL Language Reference*

Example 12-1 BNF for CREATE TABLE

```
CREATE ... TABLE [schema.]table ...;

<column_definition> ::= column [datatype]...

<datatype> ::= ... | BLOB | CLOB | NCLOB | BFILE | ...

<column_properties> ::= ... | LOB_storage_clause | ... |
LOB_partition_storage |...

<LOB_storage_clause> ::=
  LOB
  { (LOB_item [, LOB_item ]...)
      STORE AS [ SECUREFILE | BASICFILE ] (LOB_storage_parameters)
  | (LOB_item)
      STORE AS [ SECUREFILE | BASICFILE ]
        { LOB_segname (LOB_storage_parameters)
        | LOB_segname
        | (LOB_storage_parameters)
        }
  }
<LOB_storage_parameters> ::=
  { TABLESPACE tablespace
  | { LOB_parameters [ storage_clause ]
    }
  | storage_clause
  }
    [ TABLESPACE tablespace
    | { LOB_parameters [ storage_clause ]
      }
```

```
    ]...
<LOB_parameters> ::=
  [ ENABLE STORAGE IN ROW [{4000|8000}]
  | DISABLE STORAGE IN ROW
  | CHUNK integer
  | PCTVERSION integer
  | RETENTION [ { MAX | MIN integer | AUTO | NONE } ]
  | FREEPOOLS integer
  | LOB_deduplicate_clause
  | LOB_compression_clause
  | LOB_encryption_clause
  | { CACHE |  NOCACHE | CACHE READS } [ logging_clause ] } }
  ]
<LOB_retention_clause> ::=
  {RETENTION [ MAX | MIN integer | AUTO | NONE ]}
<LOB_deduplicate_clause> ::=
  { DEDUPLICATE
  | KEEP_DUPLICATES
  }
<LOB_compression_clause> ::=
  { COMPRESS [ HIGH | MEDIUM | LOW ]
  | NOCOMPRESS
  }
<LOB_encryption_clause> ::=
  { ENCRYPT [ USING 'encrypt_algorithm' ]
    [ IDENTIFIED BY password ]
  | DECRYPT
  }
<LOB_partition_storage> ::=
  {PARTITION partition
  { LOB_storage_clause | varray_col_properties }...
  [ (SUBPARTITION subpartition
  { LOB_partitioning_storage | varray_col_properties }...
  )
  ]
  }
<LOB_partitioning_storage> ::=
  {LOB (LOB_item) STORE AS [BASICFILE | SECUREFILE]
  [ LOB_segname [ ( TABLESPACE tablespace | TABLESPACE SET tablespace_set ) ]
  | ( TABLESPACE tablespace | TABLESPACE SET tablespace_set )
  ]
  }
```

12.1.2 ENABLE or DISABLE STORAGE IN ROW

LOB columns store locators that reference the location of the actual LOB value. This section describes how to enable or disable storage in a table row.

Actual LOB values are stored either in the table row (inline) or outside of the table row (out-of-line), depending on the column properties you specify when you create the table and the size of the LOB. The `ENABLE | DISABLE STORAGE IN ROW` clause is used to indicate whether the LOB should be stored inline or out-of-line. The default is `ENABLE STORAGE IN ROW` because it provides a performance benefit for small LOBs.

ENABLE STORAGE IN ROW

If `ENABLE STORAGE IN ROW` is set, the minimum inline size is 4000 and the maximum is 8000. This includes the control information and the LOB value. The default inline size for LOBs is 4000.

If the LOB is stored `IN ROW`,

- Exadata pushdown is enabled for LOBs, including when using securefile compression and encryption.
- In-Memory is enabled for LOBs without securefile compression and encryption.

LOBs larger than approximately 32k are stored out-of-line. However, the control information is still stored in the row, thus enabling us to read the out-of-line LOB data faster.

VARCHAR2(32K) and VARRAYs stored as LOBs do not support the increased inlining syntax.

DISABLE STORAGE IN ROW

In some cases, `DISABLE STORAGE IN ROW` is a better choice becase storing the LOB in the row increases the size of the row. This impacts performance if you are doing a lot of base table processing, such as full table scans, multi-row accesses (range scans), or many `UPDATE/SELECT` to columns other than the LOB columns.

12.1.3 CACHE, NOCACHE, and CACHE READS

This section discusses the guidelines to follow while creating tables that contain LOBs.

Use the cache options according to the guidelines in the following table:

Table 12-1 Using CACHE, NOCACHE, and CACHE READS Options

Cache Mode	Frequency of Read	Buffer Cache Behavior
`NOCACHE` (default)	Once or occasionally	LOB values are never brought into the buffer cache.
`CACHE READS`	Frequently	LOB values are brought into the buffer cache only during read operations and not during write operations.
`CACHE`	Read the LOB soon after write	LOB pages are placed in the buffer cache during both read and write operations. For storing semi-structured data consider turning on `CACHE` option.

> **Caution:**
>
> If your application frequently writes to LOBs, then using the `CACHE` option can potentially age other non-LOB pages out of the buffer cache prematurely.

12.1.4 LOGGING and FILESYSTEM_LIKE_LOGGING

You can apply the LOGGING parameter to LOBs in the same manner as you apply it for other table operations.

The default value of this parameter is LOGGING. For SecureFiles, the FILESYSTEM_LIKE_LOGGING parameter is equivalent to the NOLOGGING option.

If you set the LOGGING option, then Oracle Database determines the most efficient way to generate the REDO and UNDO logs for the change. Oracle recommends that you keep the LOGGING parameter turned on.

The FILESYSTEM_LIKE_LOGGING or the NOLOGGING option is useful for bulk loads and inserts. When loading data into the LOB, if you do not care about the REDO logs and can restart a failed load, then set the LOB data segment storage characteristics to FILESYSTEM_LIKE_LOGGING. This provides good performance for the initial load of data. Once you have completed loading data, Oracle recommends that you use the ALTER TABLE statement to modify the LOB storage characteristics for the LOB data segment for normal LOB operations. For example, set the cache option to CACHE or CACHE READS, along with the LOGGING option.

> **See Also:**
>
> Precedence of FORCE LOGGING Settings for more information about overriding the logging behavior at the database level

> **Note:**
>
> For BasicFiles, specifying the CACHE NOLOGGING option results in an error.

12.1.5 The RETENTION Parameter

The RETENTION parameter for SecureFile LOBs specifies how the database manages the old versions of the LOB data blocks.

Unlike other data types, the old versions of the LOB data blocks for SecureFile LOBs are stored in the LOB segment itself and are used to support consistent read operations. Without the corresponding old versions of the LOB data blocks, reading of a LOB at an earlier SCN may fail with ORA-1555. Set the RETENTION parameter as per the following guidelines:

Table 12-2 RETENTION parameter behavior

RETENTION Parameter value	Behavior
MAX	Allows the old versions of the LOB data blocks to fill the entire LOB segment. This minimizes the likelihood of an ORA-1555, if space usage is not a concern. With this setting, the old versions of the LOB data blocks may cause the LOB segment to grow. If you do not set the MAXSIZE attribute, then MAX behaves like AUTO.

Table 12-2 (Cont.) RETENTION parameter behavior

RETENTION Parameter value	Behavior
MIN	Limits the retention of old versions of the LOB data blocks to n seconds. With this setting, you must also specify the retention duration in number of seconds as n. The old versions of the LOB data blocks may also cause the LOB segment to grow.
AUTO	Oracle Database manages the space as efficiently as possible, weighing both time and space needs.
NONE	Set this value if no old version of the LOB data blocks is required for consistent read purposes. This is the most efficient setting in terms of space utilization.
not set (sets to DEFAULT)	Uses the UNDO_RETENTION setting can be set dynamically or manually. If the UNDO_RETENTION parameter is set to a positive value, then it is equivalent to setting the RETENTION parameter to MIN with the same value for retention duration. If the UNDO_RETENTION parameter is set to zero (0), then it is equivalent to setting the RETENTION parameter to NONE.

The SHRINK feature for SecureFile LOBs partially deletes old versions of the LOB data blocks to free extents, regardless of the RETENTION parameter setting. Therefore, it is recommended to have the SHRINK feature only when the RETENTION parameter is set to NONE.

The following SQL code snippet helps you determine the RETENTION parameter for a LOB segment.

```
SELECT RETENTION_TYPE, RETENTION_VALUE FROM USER_LOBs WHERE ...;
```

12.1.6 SecureFiles Compression, Deduplication, and Encryption

In addition to the features supported by BasicFiles, SecureFiles LOB storage supports the following three features that are not available with the BasicFiles LOB storage option: compression, deduplication, and encryption.

Oracle recommends that you enable compression, deduplication, and encryption through the CREATE TABLE statement.

> **Caution:**
>
> Enabling table or column level compression or encryption does not compress or encrypt the LOB data. To compress or encrypt the LOB data, use SecureFiles compression or encryption by specifying it in the LOB_storage_clause.

> **Note:**
>
> You can enable the compression, deduplication, and encryption features using the `ALTER TABLE` statement. However, if you enable these features using the `ALTER TABLE` statement, then all the data in the SecureFiles LOB storage is read, modified, and written. This can cause the database to lock the table during a potentially lengthy operation. There are online capabilities in the `ALTER TABLE` statement that can help you avoid this issue.

Topics

- Advanced LOB Compression
  Advanced LOB Compression transparently analyzes and compresses SecureFiles LOB data to save disk space and improve performance.
- Advanced LOB Deduplication
  Advanced LOB Deduplication enables Oracle Database to automatically detect duplicate LOB data within a LOB column or partition, and conserve space by storing only one copy of the data.
- SecureFiles Encryption
  In SecureFiles Encryption, the data is encrypted using Transparent Data Encryption (TDE), which allows the data to be stored securely, and still allows for random read and write access.

12.1.6.1 Advanced LOB Compression

Advanced LOB Compression transparently analyzes and compresses SecureFiles LOB data to save disk space and improve performance.

License Requirement: You must have a license for the Oracle Advanced Compression Option to implement Advanced LOB Compression.

Before you enable compression, use the `DBMS_COMPRESSION.GET_COMPRESSION_RATIO` function to estimate the space that you can save by enabling this feature for existing LOBs. This allows you to take an informed decision to enable compression.

Consider the following issues when using the `CREATE TABLE` statement with Advanced LOB Compression:

- Advanced LOB Compression is performed on the server and enables random reads and writes to LOB data. Compression utilities on the client, like `utl_compress`, cannot provide random access.
- Advanced LOB Compression does not enable table or index compression. Conversely, table and index compression do not enable Advanced LOB Compression.
- The `LOW`, `MEDIUM`, and `HIGH` options provide varying degrees of compression. The higher the compression, the higher the latency incurred. The `HIGH` setting incurs more work, but compresses the data better. The default is `MEDIUM`.

  The `LOW` compression option uses an extremely lightweight compression algorithm that removes the majority of the CPU cost that is typical with file compression. Compressed SecureFiles LOBs at the `LOW` level provide a very efficient choice for SecureFiles LOB storage. SecureFiles LOBs compressed at `LOW` generally consume less CPU time and less storage than BasicFiles LOBs, and typically help the application run faster because of a reduction in disk I/O.

- Compression can be specified at the partition level. The `CREATE TABLE lob_storage_clause` enables specification of compression for partitioned tables on a per-partition basis.
- The `DBMS_LOB.SETOPTIONS` procedure can enable and disable compression on individual SecureFiles LOBs.
- Advanced LOB compression may convert an out-of-line LOB, to an inline LOB, by moving the data from a LOB segment into the table segment (inlined in column).

The following examples demonstrate how to issue `CREATE TABLE` statements for specific compression scenarios:

Example 12-2 Creating a SecureFiles LOB Column with LOW Compression

```
CREATE TABLE t1 (a CLOB)
    LOB(a) STORE AS SECUREFILE(
    COMPRESS LOW
    CACHE
    NOLOGGING
  );
```

Example 12-3 Creating a SecureFiles LOB Column with MEDIUM (default) Compression

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         COMPRESS
         CACHE
         NOLOGGING
    );
```

Example 12-4 Creating a SecureFiles LOB Column with HIGH Compression

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         COMPRESS HIGH
         CACHE
    );
```

Example 12-5 Creating a SecureFiles LOB Column with Disabled Compression

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         NOCOMPRESS
         CACHE
    );
```

Example 12-6 Creating a SecureFiles LOB Column with Compression on One Partition

```
CREATE TABLE t1 ( REGION VARCHAR2(20), a BLOB)
     LOB(a) STORE AS SECUREFILE (
        CACHE
     )
     PARTITION BY LIST (REGION) (
          PARTITION p1 VALUES ('x', 'y')
               LOB(a) STORE AS SECUREFILE (
                   COMPRESS
                ),
          PARTITION p2 VALUES (DEFAULT)
     );
```

12.1.6.2 Advanced LOB Deduplication

Advanced LOB Deduplication enables Oracle Database to automatically detect duplicate LOB data within a LOB column or partition, and conserve space by storing only one copy of the data.

License Requirement: You must have a license for the Oracle Advanced Compression Option to implement Advanced LOB Deduplication.

Consider these issues when using `CREATE TABLE` and Advanced LOB Deduplication.

- Identical LOBs are good candidates for deduplication. Copy operations can avoid data duplication by enabling deduplication.
- Duplicate detection happens within a LOB segment. Duplicate detection does not span partitions or subpartitions for partitioned and subpartitioned LOB columns.
- Deduplication can be specified at a partition level. The `CREATE TABLE lob_storage_clause` enables specification for partitioned tables on a per-partition basis.
- The `DBMS_LOB.SETOPTIONS` procedure can enable or disable deduplication on individual LOBs.

Sample Commands

The following examples demonstrate how to issue `CREATE TABLE` statements for specific deduplication scenarios:

Example 12-7 Creating a SecureFiles LOB Column with Deduplication

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
        DEDUPLICATE
        CACHE
    );
```

Example 12-8 Creating a SecureFiles LOB Column with Disabled Deduplication

```
CREATE TABLE t1 ( a CLOB)
    LOB(a) STORE AS SECUREFILE (
         KEEP_DUPLICATES
         CACHE
    );
```

Example 12-9 Creating a SecureFiles LOB Column with Deduplication on One Partition

```
CREATE TABLE t1 ( REGION VARCHAR2(20), a BLOB)
     LOB(a) STORE AS SECUREFILE (
           CACHE
)
PARTITION BY LIST (REGION) (
     PARTITION p1 VALUES ('x', 'y')
          LOB(a) STORE AS SECUREFILE (
               DEDUPLICATE
          ),
     PARTITION p2 VALUES (DEFAULT)
);
```

Example 12-10 Creating a SecureFiles LOB column with Deduplication Disabled on One Partition

```
CREATE TABLE t1 ( REGION VARCHAR2(20), ID NUMBER, a BLOB)
     LOB(a) STORE AS SECUREFILE (
           DEDUPLICATE
           CACHE
)
PARTITION BY RANGE (REGION)
      SUBPARTITION BY HASH(ID) SUBPARTITIONS 2 (
        PARTITION p1 VALUES LESS THAN (51)
           lob(a) STORE AS a_t2_p1
           (SUBPARTITION t2_p1_s1 lob(a) STORE AS a_t2_p1_s1,
            SUBPARTITION t2_p1_s2 lob(a) STORE AS a_t2_p1_s2),
        PARTITION p2 VALUES LESS THAN (MAXVALUE)
           lob(a) STORE AS a_t2_p2 ( KEEP_DUPLICATES )
           (SUBPARTITION t2_p2_s1 lob(a) STORE AS a_t2_p2_s1,
            SUBPARTITION t2_p2_s2 lob(a) STORE AS a_t2_p2_s2)
      );
```

12.1.6.3 SecureFiles Encryption

In SecureFiles Encryption, the data is encrypted using Transparent Data Encryption (TDE), which allows the data to be stored securely, and still allows for random read and write access.

License Requirement: You must have a license for the Oracle Advanced Security Option to implement SecureFiles Encryption.

Consider the following issues when using `CREATE TABLE` statement with SecureFiles Encryption:

- Securefile Encryption encrypts the data stored in the SecureFile LOB column, irrespective of whether the data is stored in-row or out-of-line in the LOB segment. Note that table or column level encryption will not encrypt the data stored out-of-line in the LOB segment.
- SecureFile Encryption relies on a wallet, or Hardware Security Model (HSM), to hold the encryption key. The wallet setup is the same as that described for Transparent Data Encryption (TDE) and Tablespace Encryption, so complete that before using SecureFile encryption.

> **See Also:**
>
> "*Oracle Database Advanced Security Guide* for information about creating and using Oracle wallet with TDE.

- The *`encrypt_algorithm`* indicates the name of the encryption algorithm. Valid algorithms are: `AES192` (default), `AES128`, and `AES256`.
- The column encryption key is derived from `PASSWORD`, if specified.
- The default for LOB encryption is `SALT`. `NO SALT` is not supported.
- SecureFile Encryption is only supported at the table level on a per-column basis, and not at the per-partition level. Hence all partitions within a LOB column are encrypted.
- `DECRYPT` keeps the LOBs in clear text.
- Key management controls the ability to encrypt or decrypt.

- TDE is not supported by the traditional `import` and `export` utilities or by transportable-tablespace-based `export`. Use the Data Pump `expdb` and `impdb` utilities with encrypted columns instead.

The following examples demonstrate how to issue `CREATE TABLE` statements for specific encryption scenarios:

Example 12-11 Creating a SecureFiles LOB Column with a Specific Encryption Algorithm

```
CREATE TABLE t1 ( a CLOB ENCRYPT USING 'AES128')
    LOB(a) STORE AS SECUREFILE (
         CACHE
    );
```

Example 12-12 Creating a SecureFiles LOB column with encryption for all partitions

```
CREATE TABLE t1 ( REGION VARCHAR2(20), a BLOB)
  LOB(a) STORE AS SECUREFILE (
  ENCRYPT USING 'AES128'
  NOCACHE
  FILESYSTEM_LIKE_LOGGING
)
PARTITION BY LIST (REGION) (
PARTITION p1 VALUES ('x', 'y'),
PARTITION p2 VALUES (DEFAULT)
);
```

Example 12-13 Creating a SecureFiles LOB Column with Encryption Based on a Password Key

```
CREATE TABLE t1 ( a CLOB ENCRYPT IDENTIFIED BY foo)
    LOB(a) STORE AS SECUREFILE (
        CACHE
    );
```

The following example has the same result because the encryption option can be set in the `LOB_encryption_clause` section of the statement:

```
CREATE TABLE t1 (a CLOB)
    LOB(a) STORE AS SECUREFILE (
        CACHE
        ENCRYPT
        IDENTIFIED BY foo
    );
```

Example 12-14 Creating a SecureFiles LOB Column with Disabled Encryption

```
CREATE TABLE t1 ( a CLOB )
    LOB(a) STORE AS SECUREFILE (
        CACHE DECRYPT
    );
```

12.1.7 BasicFile Specific Parameters

This section discusses the storage parameters specific to BasicFiles.

The following storage parameters are specific to BasicFiles:

> **Caution:**
>
> Oracle strongly recommends that you use SecureFile LOBs for all your LOB needs.

PCTVERSION

When a BasicFiles LOB is modified, a new version of the BasicFiles LOB page is produced in order to support consistent read operations of prior versions of the BasicFiles LOB value. The `PCTVERSION` parameter is the percentage of all used BasicFiles LOB data space that can be occupied by old versions of BasicFiles LOB data pages. As soon as old versions of BasicFiles LOB data pages start to occupy more than the `PCTVERSION` amount of used BasicFiles LOB space, Oracle Database tries to reclaim the old versions and reuse them. The `PCTVERSION` parameter has the following preset values:

- Default: 10%
- Minimum: 0
- Maximum: 100

If your application requires several BasicFiles LOB updates that are concurrent with heavy reads of BasicFiles LOB columns, then consider using a higher value for the `PCTVERSION` parameter, such as 20%. If persistent BasicFiles LOB instances in your application are created and written just once and are primarily read-only afterward, then updates are infrequent. In this case, consider using a lower value for the `PCTVERSION` parameter, such as 5% or lower. If existing BasicFiles LOBs are known to be read-only, then you can safely set the `PCTVERSION` parameter to 0% because there will never be any pages needed for old versions of data.

> **Note:**
>
> The `PCTVERSION` parameter and the `RETENTION` parameter are mutually exclusive for BasicFiles LOBs, that is, you can specify either the `PCTVERSION` parameter or the `RETENTION` parameter, but not both.

CHUNK

A chunk is one or more Oracle blocks. You can specify the chunk size for the BasicFiles LOB when creating the table that contains the LOB. This corresponds to the data size used by Oracle Database when accessing or modifying the LOB value. Part of the chunk is used to store system-related information and the rest stores the LOB value. The APIs that you use to retrieve the chunk size, return the amount of space used in the LOB chunk to store the LOB value. You can use the following APIs to retrieve the chunk size:

- The `DBMS_LOB.GETCHUNKSIZE` procedure in PL/SQL
- The `OCILobGetChunkSize()` function in OCI

Once you specify the value of the `CHUNK` parameter (when the LOB column is created), you cannot change it without moving the LOB. You can set the `CHUNK` parameter to the data size most frequently accessed or written. It is more efficient to access LOBs in big chunks. If you explicitly specify storage characteristics for the LOB, then make sure that you set the `INITIAL` parameter and the `NEXT` parameter for the LOB data segment storage to a size that is larger than the `CHUNK` size.

For SecureFiles, the `CHUNK` size is an advisory size and is provided for backward compatibility purposes.

FREEPOOLS

Specifies the number of `FREELIST` groups for BasicFiles LOBs, if the database is in automatic undo mode. Under Release 12*c* compatibility, this parameter is ignored when SecureFiles LOBs are created.

FREELISTS or FREELIST GROUPS

Specifies the number of process freelists or freelist groups, respectively, allocated to the segment; `NULL` for partitioned tables. Under Release 12*c* compatibility, these parameters are ignored when SecureFiles LOBs are created.

12.1.8 Restriction on First Extent of a LOB Segment

This section discusses the first extent requirements on SecureFiles and BasicFiles.

First Extent of a SecureFile LOB Segment

A SecureFile LOB segment can only be created in Locally Managed Tablespace with Automatic Segment Space Management (ASSM). The number of blocks required in the first extent depends on the release. Before 21c, the first extent requires at least 16 blocks. After 21c, the number is 32 if the compatible parameter is greater than or equal to 20.1.0.0.0. Segments created in the previous release will continue to work in the new release. However, they will not be automatically upgraded.

The actual size of the first extent depends on the database block_size. If the tablespace is configured to use uniform extent, the extent must be bigger than the aforementioned number. For example, with `block_size = 8k`, the uniform extent size must be at least 128K pre-21c, or 256K on 21c with compatible parameter set. If the tablespace is configured to use uniform extent that is less than this number, the LOB segment creation will fail.

First Extent of a BasicFile LOB Segment

A BasicFile LOB segment can be created in Dictionary Managed or Locally Managed Tablespaces. The segment requires at least 3 blocks in the first extent. This translates into different extent sizes based on the database block_size. If the tablespace is configured to use uniform extent that contains fewer than 3 blocks, the LOB segment creation will fail.

12.1.9 Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs

The table in this section summarizes the parameters of the `CREATE TABLE` statement that relate to Securefile LOB storage.

Table 12-3 Parameters of CREATE TABLE Statement Related to LOBs

Parameter	Description
SECUREFILE	Specifies SecureFiles LOBs storage. Starting with Oracle Database 12*c*, the SecureFiles LOB storage type, specified by the parameter SECUREFILE, is the default. A SecureFiles LOB can only be created in a tablespace managed with Automatic Segment Space Management (ASSM).
BASICFILE	Specifies BasicFiles LOB storage, the original architecture for LOBs. You must explicitly specify the parameter BASICFILE to use the BasicFiles LOB storage type. For BasicFiles LOBs, specifying any of the SecureFiles LOB options results in an error.
RETENTION	Specifies the retention policy for storing old versions of LOB data to support consistent read. Possible values are: MAX, MIN, AUTO and NONE.
MAXSIZE	Specifies the upper limit of storage space that a LOB may use. The default size is 4000, but this can go up to 8000. If this amount of space is consumed, new LOB data blocks are taken from the pool of old versions of LOB data blocks as needed, regardless of time requirements.
CACHE, NOCACHE, CACHE READS	Specifies when the LOB data in brought into the buffer cache. • NOCACHE: Never brought into buffer cache. • CACHE READS: Only during reads. • CACHE: During reads and writes. The default is NOCACHE.
LOGGING, NOLOGGING, or FILESYSTEM_LIKE_LOGGING	Specifies whether to generate REDO and UNDO for changes to the LOB: • LOGGING: Generate REDO and UNDO for the change • FILESYSTEM_LIKE_LOGGING/NOLOGGING: Log only the metadata. The default is LOGGING.
COMPRESS or NOCOMPRESS	The COMPRESS option turns on Advanced LOB Compression, and NOCOMPRESS turns it off. The default is NOCOMPRESS.
DEDUPLICATE or KEEP_DUPLICATES	The DEDUPLICATE option enables Advanced LOB Deduplication; it specifies that SecureFiles LOB data that is identical in two or more rows in a LOB column, partition or subpartition must share the same data blocks. The database combines SecureFiles LOBs with identical content into a single copy, reducing storage and simplifying storage management. The opposite of this option is KEEP_DUPLICATES. The default is KEEP_DUPLICATES.
ENCRYPT or DECRYPT	The ENCRYPT option turns on SecureFiles Encryption, and encrypts all SecureFiles LOB data using Oracle Transparent Data Encryption (TDE). The DECRYPT options turns off SecureFiles Encryption. The default is DECRYPT.

12.2 Altering an Existing LOB Column

You can use the `ALTER TABLE` statement to change the storage characteristics of a LOB column.

- ALTER TABLE BNF
  This section has the syntax for `ALTER TABLE` in Backus Naur (BNF) notation, parts of which have been simplified to keep the focus on LOB-specific parameters.
- ALTER TABLE MODIFY vs ALTER TABLE MOVE LOB
  This section compares the storage characteristics while using `ALTER TABLE MODIFY` and `ALTER TABLE MOVE LOB`.
- ALTER TABLE SecureFiles LOB Features
  This section discusses the features of SecureFile LOBs that work with the `ALTER TABLE` statement.

12.2.1 ALTER TABLE BNF

This section has the syntax for `ALTER TABLE` in Backus Naur (BNF) notation, parts of which have been simplified to keep the focus on LOB-specific parameters.

> **See Also:**
>
> ALTER TABLE for more information on usage of `ALTER TABLE` statement.

```
ALTER TABLE [ schema.]table ... [ ... | column_clauses | ... |
move_table_clause] ...;

<column_clauses> ::= ... | [ modify_LOB_storage_clause |
rename_lob_storage_clause ] ...

<modify_LOB_storage_clause> ::= MODIFY LOB (LOB_item)
( modify_LOB_parameters )
<modify_LOB_parameters> ::=
{ storage_clause
  | PCTVERSION integer
  | FREEPOOLS integer
  | REBUILD FREEPOOLS
  | LOB_retention_clause
  | LOB_deduplicate_clause
  | LOB_compression_clause
  | { ENCRYPT encryption_spec | DECRYPT }
  | { CACHE
  | { NOCACHE | CACHE READS } [ logging_clause ]
  | allocate_extent_clause
  | shrink_clause
  | deallocate_unused_clause
}. . .

<rename_lob_storage_clause> ::= RENAME LOB(LOB_item) <LOB_RENAME_PARAMETERS>
<LOB_RENAME_PARAMETERS> ::= [ PARTITION | SUBPARTITION | ] <OLD_SEGMENT_NAME>
```

```
TO <NEW_SEGMENT_NAME>

<move_table_clause> ::= MOVE ...[ ... | LOB_storage_clause | ...] ...

<LOB_storage_clause> ::=
  LOB
  { (LOB_item [, LOB_item ]...)
      STORE AS [ SECUREFILE | BASICFILE ] (LOB_storage_parameters)
  | (LOB_item)
      STORE AS [ SECUREFILE | BASICFILE ]
        { LOB_segname (LOB_storage_parameters)
        | LOB_segname
        | (LOB_storage_parameters)
        }
  }

<LOB_storage_parameters> ::=
  { TABLESPACE tablespace
  | { LOB_parameters [ storage_clause ]
    }
  | storage_clause
  }
    [ TABLESPACE tablespace
    | { LOB_parameters [ storage_clause ]
      }
    ]...

<LOB_parameters> ::=
  [ ENABLE STORAGE IN ROW [{4000|8000}]
  | DISABLE STORAGE IN ROW
  | CHUNK integer
  | PCTVERSION integer
  | RETENTION [ { MAX | MIN integer | AUTO | NONE } ]
  | FREEPOOLS integer
  | LOB_deduplicate_clause
  | LOB_compression_clause
  | LOB_encryption_clause
  | { CACHE |  NOCACHE | CACHE READS } [ logging_clause ] } }
  ]

<LOB_retention_clause> ::=
  {RETENTION [ MAX | MIN integer | AUTO | NONE ]}

<LOB_deduplicate_clause> ::=
  { DEDUPLICATE
  | KEEP_DUPLICATES
  }

<LOB_compression_clause> ::=
  { COMPRESS [ HIGH | MEDIUM | LOW ]
  | NOCOMPRESS
  }

<LOB_encryption_clause> ::=
  { ENCRYPT [ USING 'encrypt_algorithm' ]
    [ IDENTIFIED BY password ]
```

```
| DECRYPT
}
```

12.2.2 ALTER TABLE MODIFY vs ALTER TABLE MOVE LOB

This section compares the storage characteristics while using `ALTER TABLE MODIFY` and `ALTER TABLE MOVE LOB`.

There are two kinds of changes to existing storage characteristics:

1. Some changes to storage characteristics merely apply to the way the data is accessed and do not require moving the entire existing LOB data. For such changes, use the `ALTER TABLE MODIFY LOB` syntax, which uses the `modify_LOB_storage_clause` from the `ALTER TABLE` BNF. Examples of changes that do not require moving the entire existing LOB data are: `RETENTION`, `PCTVERSION`, `CACHE`, `NOCACHELOGGING`, `NOLOGGING`, or `STORAGE` settings, shrinking the space used by the LOB data, and deallocating unused segments.

   > **See Also:**
   >
   > ALTER TABLE

2. Some changes to storage characteristics require changes to the way the data is stored, hence requiring movement of the entire existing LOB data. For such changes use the `ALTER TABLE MOVE LOB` syntax instead of the `ALTER TABLE MODIFY LOB` syntax because the former performs parallel operations on SecureFiles LOBs columns, making it a resource-efficient approach. The `ALTER TABLE MOVE LOB` syntax can process any arbitrary LOB storage clause represented by the `LOB_storage_clause` in the `ALTER TABLE` BNF, and will move the LOB data to a new location.
   Examples of changes that require moving the entire existing LOB data are: `TABLESPACE`, `ENABLE/DISABLE STORAGE IN ROW`, `CHUNK`, `COMPRESSION`, `DEDUPLICATION` and `ENCRYPTION` settings.

   As an alternative to `ALTER TABLE MOVE LOB`, you can use online redefinition to enable one or more of these features. As with `ALTER TABLE`, online redefinition of SecureFiles LOB columns can be executed in parallel.

> **See Also:**
>
> - ALTER TABLE for more information about `ALTER TABLE` statement.
> - DBMS_REDEFINITION for more information about `DBMS_REDEFINITION` package.

12.2.3 ALTER TABLE SecureFiles LOB Features

This section discusses the features of SecureFile LOBs that work with the `ALTER TABLE` statement.

- ALTER TABLE with Advanced LOB Compression
  When used with the `ALTER TABLE` statement, advanced LOB compression syntax alters the compression mode of the LOB column. The examples in this section demonstrate how to issue `ALTER TABLE` statements for specific compression scenarios.

- ALTER TABLE with Advanced LOB Deduplication
  When used with the `ALTER TABLE` statement, advanced LOB deduplication syntax alters the deduplication mode of the LOB column.
- ALTER TABLE with SecureFiles Encryption
  The examples in this section demonstrate how to issue ALTER TABLE statements for to enable SecureFiles encryption.

12.2.3.1 ALTER TABLE with Advanced LOB Compression

When used with the `ALTER TABLE` statement, advanced LOB compression syntax alters the compression mode of the LOB column. The examples in this section demonstrate how to issue `ALTER TABLE` statements for specific compression scenarios.

Example: Altering a SecureFiles LOB Column to Enable LOW Compression

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(COMPRESS LOW)
```

Example: Altering a SecureFiles LOB Column to Disable Compression

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(NOCOMPRESS)
```

Example: Altering a SecureFiles LOB Column to Enable HIGH Compression

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(COMPRESS HIGH);
```

Example: Altering a SecureFiles LOB Column to Enable Compression on One partition

```
ALTER TABLE t1 MOVE PARTITION p1 LOB(a) STORE AS SECUREFILE(COMPRESS HIGH);
```

12.2.3.2 ALTER TABLE with Advanced LOB Deduplication

When used with the `ALTER TABLE` statement, advanced LOB deduplication syntax alters the deduplication mode of the LOB column.

Before you enable deduplication, you can use the `GET_LOB_DEDUPLICATION_RATIO` function to estimate the space that you can save by enabling this feature for an existing LOB. You can also use this function to estimate the space that you can save by enabling deduplication, before migrating a BasicFiles LOB to SecureFiles LOB. This enables you to take an informed decision to enable deduplication. See GET_LOB_DEDUPLICATION_RATIO Function in *PL/SQL Packages and Types Reference*.

Disclaimer: The deduplication ratio is an approximate value, which is calculated based on the sampled rows in the LOB column. The actual space that you save when you enable deduplication for the complete table may be different.

The examples in this section demonstrate how to issue `ALTER TABLE` statements for specific deduplication scenarios.

Example: Altering a SecureFiles LOB Column to Disable Deduplication

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(KEEP_DUPLICATES);
```

Example: Altering a SecureFiles LOB Column to Enable Deduplication

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(DEDUPLICATE);
```

Example: Altering a SecureFiles LOB Column to Enable Deduplication on One Partition

```
ALTER TABLE t1 MOVE PARTITION p1 LOB(a) STORE AS SECUREFILE(DEDUPLICATE);
```

12.2.3.3 ALTER TABLE with SecureFiles Encryption

The examples in this section demonstrate how to issue ALTER TABLE statements for to enable SecureFiles encryption.

Consider the following points when using the `ALTER TABLE` statement with SecureFiles Encryption:

- The `ALTER TABLE` statement enables and disables SecureFiles Encryption. Using the `REKEY` option with the `ALTER TABLE` statement also enables you to encrypt LOB columns with a new key or algorithm.
- The `DECRYPT` option converts encrypted columns to clear text form.

> **See Also:**
>
> 'CREATE TABLE' Usage Notes for SecureFiles Encryption

Following examples demonstrate how to issue `ALTER TABLE` statements for specific encryption scenarios:

Example: Altering a SecureFiles LOB Column by Encrypting Based on `AES256` encryption

```
ALTER TABLE t1 MOVE LOB(a) STORE AS SECUREFILE(ENCRYPT USING 'AES256');
```

Example: Altering a SecureFiles LOB Column by Encrypting Based on a Password Key

```
ALTER TABLE t1 MOVE LOB(a)
    STORE AS SECUREFILE(ENCRYPT USING 'AES256' IDENTIFIED BY foo);
```

Example: Altering a SecureFiles LOB Column by Regenerating the Encryption key

```
ALTER TABLE t1 REKEY USING 'AES256';
```

12.3 Creating an Index on LOB Column

The contents of a LOB are often specific to the application, so an index on the LOB column will usually deal with application logic. You can create a function-based or a domain index on a LOB column to improve the performance of queries accessing data stored in LOB columns. You cannot build a B-tree or bitmap index on a LOB column.

Function-based and domain indexes are automatically updated when a DML operation is performed on the LOB column, or when a LOB is updated using an API like DBMS_LOB.

You can use the LOB Open/Close API to defer index maintenance to after a bunch of write operations. Opening a LOB in read-write mode defers any index maintenance on the LOB

column until you close the LOB. This is useful when you do not want the database to perform index maintenance every time you write to the LOB. This technique can improve the performance of your application if you are doing several write operations on the LOB while it is open. Any index on the LOB column is not valid until you explicitly close the LOB.

- Function-Based Indexing on LOB Columns
  A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.
- Domain Indexing on LOB Columns
  Indexes created by using Extensible Indexing interfaces are known as Domain indexes.

> **See Also:**
>
> Before You Begin

12.3.1 Function-Based Indexing on LOB Columns

A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

> **See Also:**
>
> When to Use Function-Based Indexes

The following example demonstrates the creation of a function-based index on a LOB column using a SQL function:

```
-- Function-Based Index using a SQL function
CREATE INDEX ad_sourcetext_idx_sql ON
print_media(to_char(substr(ad_sourcetext,1,10)));
```

The following example demonstrates the creation of a function-based index on a LOB column using a PL/SQL function:

```
-- Function-Based Index using a PL/SQL function
-- LOB can be an input but cannot be the return type of hte function
CREATE OR REPLACE FUNCTION Ret1st2Char(CLobInput CLOB) RETURN CHAR
DETERMINISTIC IS
       First2Char         CHAR(2) ;
       NoOfChar           INTEGER ;
BEGIN
        NoOfChar := 2 ;
        DBMS_LOB.Read(CLobInput, NoOfChar, 1, First2Char) ;
        RETURN First2Char ;
END ;
/
```

```
CREATE INDEX ad_sourcetext_idx_plsql on
print_media(Ret1st2Char(ad_sourcetext));
```

12.3.2 Domain Indexing on LOB Columns

Indexes created by using Extensible Indexing interfaces are known as Domain indexes.

The database provides extensible indexing interfaces, a feature which enables you to define new index types as required. This is based on the concept of cooperative indexing where a data cartridge and the database build and maintain indexes for data types such as text and spatial.

The cartridge is responsible for defining the index structure, maintaining the index content during load and update operations, and searching the index during query processing. The index structure can be stored in Oracle as heap-organized, or an index-organized table, or externally as an operating system file.

To support this structure, the database provides an indextype. The purpose of an indextype is to enable efficient search and retrieval functions for complex domains such as text, spatial, and image by means of a data cartridge. An indextype is analogous to the sorted or bit-mapped index types that are built-in within the Oracle Server. The difference is that an indextype is implemented by the data cartridge developer, whereas the Oracle kernel implements built-in indexes. Once a new indextype has been implemented by a data cartridge developer, end users of the data cartridge can use it just as they would built-in index types.

When the database system handles the physical storage of domain indexes, data cartridges:

- Define the format and content of an index. This enables cartridges to define an index structure that can accommodate a complex data object. For instance, an inverted index for text documents or a quad-tree for spatial features.
- Build, delete, and update a domain index. The cartridge handles building and maintaining the index structures.
- Access and interpret the content of an index. This capability enables the data cartridge to become an integral component of query processing. That is, the content-related clauses for database queries are handled by the data cartridge.

By supporting domain indexes, the database significantly reduces the effort needed to develop high-performance solutions that access complex data types such as LOBs.

- Extensible Optimizer
  Extensible Optmizer enables collection of statistics on user-defined functions and domain indexes.
- Text Indexes on LOB Columns
  If the contents of your LOB column correspond to that of a document type, users are allowed to index such a column using Oracle Text indexes.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide*

12.3.2.1 Extensible Optimizer

Extensible Optmizer enables collection of statistics on user-defined functions and domain indexes.

The SQL optimizer cannot collect statistics over LOB columns nor can it estimate the cost and selectivity of predicates involving LOB columns. Instead, the Extensible Optimizer functionality allows authors of user-defined functions and domain indexes to create statistics collection, selectivity, and cost functions. This information is used by the optimizer in choosing a query plan. The cost-based optimizer is thus extended to use the user-supplied information.

The Extensible Indexing interfaces enable you to define new operators, indextypes, and domain indexes. For such user-defined operators and domain indexes, the Extensible Optimizer interfaces allows users to control the three main components used by the optimizer to select an execution plan: statistics, selectivity, and cost. This allows the cartridge developer to tune the Extensible Optimizer for efficient execution of queries involving predicates or indexes over complex data types such as LOBs.

> **See Also:**
>
> Extensible Optimizer

12.3.2.2 Text Indexes on LOB Columns

If the contents of your LOB column correspond to that of a document type, users are allowed to index such a column using Oracle Text indexes.

For example, consider the following table `DOCUMENT_TABLE` storing text-based documents on a CLOB column:

```
CREATE TABLE document_table (
    docno NUMBER,
    document CLOB);
```

You can index the contents of the DOCUMENT column with one of the Oracle Text indexing options to speed up text-based queries. The following example will create a SEARCH index used for text-search queries over the DOCUMENT column.

```
CREATE INDEX document_index ON document_table (document) INDEXTYPE IS
CTXSYS.CONTEXT;

CREATE SEARCH INDEX document_index ON document_table (document);
```

> **Note:**
>
> You can create an Oracle Text index on other formats as well. Examples of other formats include PDF, JSON, or XML.

> **See Also:**
>
> Creating Oracle Text Indexes

12.4 LOBs in Partitioned Tables

Partitioning can simplify the manageability of large database objects. This section discusses various aspects of LOBs in partitioned tables.

Very large tables and indexes can be decomposed into smaller and more manageable pieces called partitions, which are entirely transparent to an application. You can partition tables that contain LOB columns. All partitioning schemes supported by Oracle are fully supported on LOBs.

> **See Also:**
>
> Partitions_ Views_ and Other Schema Objects
> Partitioning for All Databases

LOBs can take advantage of all of the benefits of partitioning including the following:

- LOB segments can be spread between several tablespaces to balance I/O load and to make backup and recovery more manageable.
- LOBs in a partitioned table become easier to maintain.
- LOBs can be partitioned into logical groups to speed up operations on LOBs that are accessed as a group.

The following section describes some of the ways you can manipulate LOBs in partitioned tables.

- Partitioning a Table Containing LOB Columns
  All partitioning schemes supported by Oracle are fully supported on LOBs. This section discusses the partitioning of tables with LOB columns.
- Default LOB Storage Attributes
  This section discusses the default LOB storage attributes.
- Partition Maintenance Operation
  This section discusses maintenance operations on partitioned tables with LOB columns.
- Creating an Index on a Table Containing Partitioned LOB Columns
  To improve the performance of queries, you can create local or global indexes on partitioned LOB columns.

12.4.1 Partitioning a Table Containing LOB Columns

All partitioning schemes supported by Oracle are fully supported on LOBs. This section discusses the partitioning of tables with LOB columns.

You can partition a table containing LOB columns using any of the following techniques:

- When the table is created using the `PARTITION BY ...` clause of the `CREATE TABLE` statement.

- Adding a partition to an existing table using the `ALTER TABLE ... ADD PARTITION` clause.

The data dictionary views `USER_LOB_PARTITIONS`, `ALL_LOB_PARTITIONS` and `DBA_LOB_PARTITIONS` provide partition specific information for a LOB column.

Different partitions can have different inline sizes. This is useful if you want to `EXCHANGE` a new table into a partition of an existing table. If a partition level inline size is not specified, the column's table level default is used. In the case of composite partitioning, the sub-partition-level inline size takes precedence over the partition-level inline size, which takes precedence over the table-level inline size. During new partition creation, if inline size values are not specified, the table level defaults are used. Inline size values are never `NULL`.

Example 12-15 A partitioned table with LOB columns:

```
CREATE TABLE print_media
    ( product_id        NUMBER(6),
      ad_id             NUMBER(6),
      ad_sourcetext     CLOB)
    LOB (ad_sourcetext)  STORE AS SECUREFILE (TABLESPACE tbs_2)
    PARTITION BY RANGE(product_id)
    (PARTITION P1 VALUES LESS THAN (1000)
         LOB (ad_sourcetext) STORE AS BASICFILE (TABLESPACE tbs_1),
     PARTITION P2 VALUES LESS THAN (2000)
         LOB (ad_sourcetext) STORE AS (TABLESPACE tbs_2 COMPRESS HIGH),
     PARTITION P3 VALUES LESS THAN (3000));
```

> **See Also:**
>
> Summary of CREATE TABLE LOB Storage Parameters for Securefile LOBs

Example 12-16 A partitioned table with different inline sizes for LOB columns

```
CREATE TABLE print_media
    ( product_id        NUMBER(6),
      ad_content        BLOB,
      ad_sourcetext     CLOB)
      LOB (ad_content) STORE AS SECUREFILE (ENABLE STORAGE IN ROW)
      LOB (ad_sourcetext) STORE AS SECUREFILE (ENABLE STORAGE IN ROW 8000)
TABLESPACE tbs_1
    PARTITION BY RANGE(product_id)
    (PARTITION P1 VALUES LESS THAN (1000)
        LOB (ad_sourcetext) STORE AS SECUREFILE (ENABLE STORAGE IN ROW 4000),
    PARTITION P2 VALUES LESS THAN (2000)
        LOB (ad_sourcetext) STORE AS (ENABLE STORAGE IN ROW 8000 COMPRESS
HIGH),
    PARTITION P3 VALUES LESS THAN (3000)
        LOB (ad_content) STORE AS (DISABLE STORAGE IN ROW));
```

12.4.2 Default LOB Storage Attributes

This section discusses the default LOB storage attributes.

In the above example, the default storage attribute for LOB column `ad_sourcetext` is mentioned as "`STORE AS SECUREFILE (TABLESPACE tbs_2)`". This means that if no LOB storage clause is provided for any partition, this default will be used. In this example, partition `P3` uses tablespace `tbs_2` since no LOB storage is specified. Similarly, `SECUREFILE` is the default storage and is used by partitions `P2` and `P3`, but partition `P1` overrides it to specify BasicFile storage.

The dictionary views `USER_PART_LOBS`, `ALL_PART_LOBS` and `DBA_PART_LOBS` provide information on default LOB storage options for a LOB column in a table.

The table level default LOB storage attribute can be changed, as shown in the example below:

```
ALTER TABLE print_media MODIFY DEFAULT ATTRIBUTES  LOB (ad_sourcetext)
    (TABLESPACE  tbs_1);
```

The change in the default attribute will not affect the existing partitions. Any new partitions created without LOB storage clause will inherit the default values for that column.

12.4.3 Partition Maintenance Operation

This section discusses maintenance operations on partitioned tables with LOB columns.

All partitioning maintenance operations are supported with LOB columns. Here are some examples:

Example 12-17 Adding Partition containing LOBs

```
ALTER TABLE print_media ADD PARTITION P4 VALUES LESS THAN (4000)
         LOB (ad_sourcetext) STORE AS SECUREFILE(TABLESPACE tbs_2);
```

Example 12-18 Modifying Partition Containing LOBs

```
ALTER TABLE print_media MODIFY PARTITION P3 LOB(ad_sourcetext)
     (RETENTION AUTO);
```

Example 12-19 Moving Partition Containing LOBs

```
ALTER TABLE print_media MOVE PARTITION P1 LOB(ad_sourcetext)
    STORE AS (TABLESPACE tbs_3 COMPRESS LOW);
```

The example above moves a LOB partition into a different tablespace, which can be useful if the tablespace is no longer large enough to hold the partition. Move partition can also be used to perform other operations that require moving the LOB data, such as performing a `COMPRESS` operation on the LOB, or changing the `ENABLE / DISABLE STORAGE IN ROW` option.

Example 12-20 Splitting Partitions Containing LOBs

You can split a partition containing LOBs into two using the `ALTER TABLE ... SPLIT PARTITION` clause. Doing so permits you to place one or both new partitions in a new tablespace. For example:

```
ALTER TABLE print_media SPLIT PARTITION  P1 AT(500) into
(PARTITION P1A LOB(ad_sourcetext) STORE AS (TABLESPACE tbs_1),
PARTITION P1B LOB(ad_sourcetext) STORE AS (TABLESPACE tbs_2)) UPDATE INDEXES;
```

Example 12-21 Merging Partitions Containing LOBs

Merging partitions is useful for reclaiming unused partition space. For example:

```
ALTER TABLE print_media MERGE PARTITIONS P1A, P1B INTO PARTITION P1;
```

Example 12-22 Exchange Partition containing LOB column with non-partitioned table

Exchanging partitions with a table that has partitioned LOB columns using the `ALTER TABLE ... EXCHANGE PARTITION` clause. Exchange partition is a powerful tool to change new data / partitions to a newer storage format without the costly operation of migrating old data. You can exchange partition with LOB data having different storage option, e.g. partition p1 of BasicFile data in Example 11-15 can be exchanged with non-partitioned table with LOB column stored in SecureFile Compressed form:

```
CREATE TABLE print_media_nonpart
    ( product_id NUMBER(6),
      ad_id NUMBER(6),
      ad_sourcetext CLOB)
      LOB (ad_sourcetext) STORE AS SECUREFILE (COMPRESS HIGH);

ALTER TABLE print_media  EXCHANGE PARTITION p1 WITH TABLE print_media_nonpart;
```

12.4.4 Creating an Index on a Table Containing Partitioned LOB Columns

To improve the performance of queries, you can create local or global indexes on partitioned LOB columns.

Only function-based and domain indexes are supported on LOB columns. Other types of indexes, such as unique indexes are not supported with LOBs.

For example:

```
CREATE INDEX ad_sourcetext_idx_sql on print_media
(to_char(substr(ad_sourcetext,1,10)))
      GLOBAL;


CREATE INDEX ad_sourcetext_idx_sql on print_media
(to_char(substr(ad_sourcetext,1,10)))
      LOCAL;
```

12.5 LOBs in Index Organized Tables

Index Organized Tables (IOTs) support LOB and BFILE columns.

For the most part, SQL DDL, DML, and piecewise operations on LOBs in IOTs produce the same results as those for normal tables. The only exception is the default semantics of LOBs during creation. The main differences are:

- ***Tablespace Mapping***: By default, or unless specified otherwise, the LOB data and index segments are created in the tablespace in which the primary key index segments of the index organized table are created.
- ***Inline as Compared to Out-of-Line Storage***: By default, all LOBs in an index organized table created without an overflow segment are stored out of line. In other words, if an index organized table is created without an overflow segment, then the LOBs in this table have their default storage attributes as `DISABLE STORAGE IN ROW`. If you forcibly try to specify an `ENABLE STORAGE IN ROW` clause for such LOBs, then SQL raises an error.

  On the other hand, if an overflow segment has been specified, then LOBs in index organized tables exactly mimic their semantics in conventional tables.

Example of Index Organized Table (IOT) with LOB Columns

Consider the following example:

```
CREATE TABLE iotlob_tab (c1 INTEGER PRIMARY KEY, c2 BLOB, c3 CLOB, c4
VARCHAR2(20))
  ORGANIZATION INDEX
    TABLESPACE iot_ts
    PCTFREE 10 PCTUSED 10 INITRANS 1 MAXTRANS 1 STORAGE (INITIAL 4K)
    PCTTHRESHOLD 50 INCLUDING c2
  OVERFLOW
    TABLESPACE ioto_ts
    PCTFREE 10 PCTUSED 10 INITRANS 1 MAXTRANS 1 STORAGE (INITIAL 8K) LOB (c2)
    STORE AS lobseg (TABLESPACE lob_ts DISABLE STORAGE IN ROW
                     CHUNK 16384 PCTVERSION 10 CACHE STORAGE (INITIAL 2M)
                     INDEX lobidx_c1 (TABLESPACE lobidx_ts STORAGE (INITIAL 4K)));
```

Executing these statements results in the creation of an index organized table `iotlob_tab` with the following elements:

- A primary key index segment in the tablespace `iot_ts`,
- An overflow data segment in tablespace `ioto_ts`
- Columns starting from column `C3` being explicitly stored in the overflow data segment
- `BLOB` (column `C2`) data segments in the tablespace `lob_ts`
- `BLOB` (column `C2`) index segments in the tablespace `lobidx_ts`
- `CLOB` (column `C3`) data segments in the tablespace `iot_ts`
- `CLOB` (column `C3`) index segments in the tablespace `iot_ts`
- `CLOB` (column `C3`) stored in line by virtue of the IOT having an overflow segment
- `BLOB` (column `C2`) explicitly forced to be stored out of line

> **Note:**
>
> If no overflow had been specified, then both C2 and C3 would have been stored out of line by default.

LOBs in Partitioned Index-Organized Tables

LOB columns and attributes can be stored in partitioned index-organized tables.

Index-organized tables can have LOBs stored as follows; however, partition maintenance operations, such as `MOVE`, `SPLIT`, and `MERGE` are not supported with:

- VARRAY data types stored as LOB data types.
- Abstract data types with LOB attributes.
- Nested tables with LOB types.

Restrictions on Index Organized Tables with LOB Columns

The ALTER TABLE MOVE operation cannot be performed on an index organized table with a LOB column in parallel. Instead, use the `NOPARALLEL` clause to move the LOB column for such tables. For example:

```
ALTER TABLE t1 MOVE LOB(a) STORE AS (<tablespace users>) NOPARALLEL;
```