14

External Tables Concepts

The external tables feature is a complement to existing SQL*Loader functionality. It enables you to access data in external sources as if it were in a table in the database.

- How Are External Tables Created?
  External tables are created using the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.
- CREATE_EXTERNAL_PART_TABLE Procedure
  This procedure creates an external partitioned table on files in the Cloud. This procedure enables you to run queries on external data in Oracle Autonomous Database, using the `ORACLE_BIGDATA` driver.
- CREATE_EXTERNAL_TABLE Procedure
  This procedure creates an external table on files in the Cloud or from files in a directory. This enables you to run queries on external data from Oracle Database.
- Location of Data Files and Output Files
  Data files and output files must be located on the server. You must have a directory object that specifies the location from which to read and write files.
- Access Parameters for External Tables
  To modify the default behavior of the access driver for external tables, specify access parameters.
- Data Type Conversion During External Table Use
  If source and target data types do not match, then conversion errors can occur when Oracle Database reads from external tables, and when it writes to external tables.
- Vectors in External Tables
  External tables can be created with columns of type `VECTOR`, allowing vector embeddings represented in text or binary format stored in external files to be rendered as the `VECTOR` data type in Oracle Database.

14.1 How Are External Tables Created?

External tables are created using the SQL `CREATE TABLE...ORGANIZATION EXTERNAL` statement.

Note that SQL*Loader may be the better choice in data loading situations that require additional indexing of the staging table. See "Behavior Differences Between SQL*Loader and External Tables" for more information about how load behavior differs between SQL*Loader and external tables.

Starting with Oracle Database 23ai, you can load the source file name as a field in a data file for both external tables and SQL*Loader.

As of Oracle Database 12c Release 2 (12.2.0.1), you can partition data contained in external tables, which allows you to take advantage of the same performance improvements provided when you partition tables stored in a database (for example, partition pruning).

> **Note:**
>
> External tables can be used as inline external tables in SQL statements, thus eliminating the need to create an external table as a persistent database object in the data dictionary. For additional information, see *Oracle Database SQL Language Reference*.

When you create an external table, you specify the following attributes:

- `TYPE` — specifies the type of external table. Each type of external table is supported by its own access driver.
  - `ORACLE_LOADER` — this is the default access driver. It loads data from external tables to internal tables. The data must come from text data files. (The `ORACLE_LOADER` access driver cannot perform unloads; that is, it cannot move data from an internal table to an external table.)
  - `ORACLE_DATAPUMP` — this access driver can perform both loads and unloads. The data must come from binary dump files. Loads to internal tables from external tables are done by fetching from the binary dump files. Unloads from internal tables to external tables are done by populating the binary dump files of the external table. The `ORACLE_DATAPUMP` access driver can write dump files only as part of creating an external table with the SQL `CREATE TABLE AS SELECT` statement. After the dump file is created, it can be read any number of times, but it cannot be modified (that is, no DML operations can be performed).
  - `ORACLE_BIGDATA` — this access driver enables you to access data stored in object stores as if that data was stored in tables in an Oracle Database.
- `DEFAULT DIRECTORY` — specifies the default directory to use for all input and output files that do not explicitly name a directory object. The location is specified with a directory object, not a directory path. You must create the directory object before you create the external table; otherwise, an error is generated. See Location of Data Files and Output Files for more information.
- `ACCESS PARAMETERS` — describe the external data source and implement the type of external table that was specified. Each type of external table has its own access driver that provides access parameters unique to that type of external table. Access parameters are optional. See Access Parameters.
- `LOCATION` — specifies the data files for the external table.

  For `ORACLE_LOADER`, `ORACLE_BIGDATA`, and `ORACLE_DATAPUMP`, the files are named in the form *`directory:file`*. The *`directory`* portion is optional. If it is missing, then the default directory is used as the directory for the file. If you are using the `ORACLE_LOADER` access driver, then you can use wildcards in the file name: an asterisk (*) signifies multiple characters, a question mark (?) signifies a single character.

The following examples briefly show the use of attributes for each of the access drivers.

Example 14-1 Specifying Attributes for the ORACLE_LOADER Access Driver

The following example uses the `ORACLE_LOADER` access driver to show the use of each of these attributes (it assumes that the default directory `def_dir1` already exists):

```
SQL> CREATE TABLE emp_load
  2    (employee_number      CHAR(5),
```

```
  3     employee_dob         CHAR(20),
  4     employee_last_name   CHAR(20),
  5     employee_first_name  CHAR(15),
  6     employee_middle_name CHAR(15),
  7     employee_hire_date   DATE)
  8  ORGANIZATION EXTERNAL
  9    (TYPE ORACLE_LOADER
 10     DEFAULT DIRECTORY def_dir1
 11     ACCESS PARAMETERS
 12       (RECORDS DELIMITED BY NEWLINE
 13        FIELDS (employee_number      CHAR(2),
 14                employee_dob         CHAR(20),
 15                employee_last_name   CHAR(18),
 16                employee_first_name  CHAR(11),
 17                employee_middle_name CHAR(11),
 18                employee_hire_date   CHAR(10) date_format DATE mask "mm/dd/
yyyy"
 19               )
 20       )
 21     LOCATION ('info.dat')
 22    );

Table created.
```

The information you provide through the access driver ensures that data from the data source is processed so that it matches the definition of the external table. The fields listed after `CREATE TABLE emp_load` are actually defining the metadata for the data in the `info.dat` source file.

Example 14-2 Specifying Attributes for the ORACLE_DATAPUMP Access Driver

This example creates an external table named `inventories_xt` and populates the dump file for the external table with the data from table `inventories` in the `oe` sample schema.

```
SQL> CREATE TABLE inventories_xt
2 ORGANIZATION EXTERNAL
3 (
4 TYPE ORACLE_DATAPUMP
5 DEFAULT DIRECTORY def_dir1
6 LOCATION ('inv_xt.dmp')
7 )
8 AS SELECT * FROM inventories;
Table created.
```

Example 14-3 Specifying Attributes for the ORACLE_BIGDATA Access Driver

```
CREATE TABLE tab_from_csv
     (
      c0 number,
      c1 varchar2(20)
     )
     ORGANIZATION external
     (
      TYPE oracle_bigdata
      DEFAULT DIRECTORY data_pump_dir
      ACCESS PARAMETERS
```

```
    (
     com.oracle.bigdata.fileformat=csv
    )
    location
    (
     'data.csv'
    )
   )REJECT LIMIT 1
   ;
```

Related Topics

- Behavior Differences Between SQL*Loader and External Tables
  Oracle recommends that you review the differences between loading data with external tables, using the `ORACLE_LOADER` access driver, and loading data with SQL*Loader conventional and direct path loads.
- *Oracle Database Administrator's Guide* Managing External Tables

14.2 CREATE_EXTERNAL_PART_TABLE Procedure

This procedure creates an external partitioned table on files in the Cloud. This procedure enables you to run queries on external data in Oracle Autonomous Database, using the `ORACLE_BIGDATA` driver.

Use Case

Starting with Oracle Database 19c, when you are using the `ORACLE_BIGDATA` driver with object stores, you are now able to select column values from a path in external tables. This feature enables you to query and load files in object storage that are partitioned, which represent the partition columns for the table.

Syntax

```
DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE (
    table_name           IN VARCHAR2,
    credential_name      IN VARCHAR2,
    partitioning_clause  IN CLOB,
    column_list          IN CLOB,
    field_list           IN CLOB DEFAULT,
    format               IN CLOB DEFAULT);
```

Parameters

Parameter	Description
`table_name`	The name of the external table. For example: `'mysales'`
`credential_name`	The name of the credential to access the Cloud Object Storage. When resource principal is enabled, you can use `'OCI$RESOURCE_PRINCIPAL'` as the `credential_name`
`partitioning_clause`	Specifies the complete partitioning clause, including the location information for individual partitions. If you use the `partitioning_clause` parameter, then the `file_url_list` parameter is not allowed.

Parameter	Description
file_uri_list	There are two options for the file_uri_list parameter: • A comma-delimited list of individual file URIs without wildcards. • A single file URI with wildcards. The wildcards can only be after the last slash "/". If you use the parameter file_url_list, then the partitioning_clause parameter is not allowed. The specification should be the root folder in a nested path, where are multiple files within a folder structure that have the same schema. For example: https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-string/b/mybucket/0/sales/month=jan2022.csv https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-string/b/mybucket/0/sales/month=feb2022.csv In this case, the root folder for the sales table is /0/sales
column_list	Comma-delimited list of column names and data types for the external table. This parameter has the following requirements, depending on the type of the data files specified with the file_url_list parameter: • The column_list parameter is required with unstructured files. Using unstructured files, for example with CSV text files, the column_list parameter must specify all the column names and data types inside the data file as well as the partition columns derived from the object name. • The column_list parameter is optional with structured files. For example, with Avro, ORC, or Parquet data files, the column_list is not required. When the column_list is not included, the format parameter partition_columns option must include specifications for both column names (name) and data types (type). For example: 'product varchar2(100), units number, country varchar2(100), year number, month varchar2(2)',
field_list	Identifies the fields in the source files and their data types. The default value is NULL, meaning the fields and their data types are determined by the column_list parameter. This argument's syntax is the same as the field_list clause in regular Oracle Database external tables. The field_list is not required for structured files, such as Apache Parquet files..

Parameter	Description
format	The format option partition_columns specifies the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE column names and data types of partition columns when the partition columns are derived from the file path, depending on the type of data file, structured or unstructured: • When the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE includes the column_list parameter and the data files are unstructured, such as with CSV text files, partition_columns does not include the data type. For example, use a format such as the following for this type of partition_columns specification: '"partition_columns":["state","zipcode"]' The data type is not required because it is specified in the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE column_list parameter. • When the DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE does not include the column_list parameter and the data files are structured, such as Avro, ORC, or Parquet files, the partition_columns option includes both the column name, name sub-clause, and the data type, type sub-clause. For example, the following shows a partition_columns specification: '"partition_columns":[{"name":"country", "type":"varchar2(10)"}, {"name":"year", "type":"number"}, {"name":"month", "type":"varchar2(10)"}]' If the data files are unstructured and the type sub-clause is specified with partition_columns, the type sub-clause is ignored. For object names that are not based on hive format, the order of the partition_columns specified columns must match the order as they appear in the object name in the file path specified in the file_url_list parameter.

Usage Notes

- You cannot call this procedure with both partitioning_clause and file_url_list parameters.
- Specifying the column_list parameter is optional with structured data files, including Avro, Parquet, or ORC data files. If column_list is not specified, the format parameter partition_columns option must include both name and type.
- The column_list parameter is required with unstructured data files, such as CSV text files.
- The procedure DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE supports external partitioned files in the supported cloud object storage services, including:
  - Oracle Cloud Infrastructure Object Storage
  - Azure Blob Storage

  - Amazon S3-Compatible, including: Oracle Cloud Infrastructure Object Storage, Google Cloud Storage, and Wasabi Hot Cloud storage.
  - GitHub Repository

- When you call `DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE` with the `file_url_list` parameter, the types for columns specified in the Cloud Object Store file name must be one of the following types:

  ```
  VARCHAR2(n)
  NUMBER(n)
  NUMBER(p,s)
  NUMBER
  DATE
  TIMESTAMP(9)
  ```

- The default record delimiter is `detected newline`. With `detected newline`, `DBMS_CLOUD` tries to automatically find the correct newline character to use as the record delimiter. `DBMS_CLOUD` first searches for the Windows newline character `\r\n`. If it finds the Windows newline character, this is used as the record delimiter for all files in the procedure. If a Windows newline character is not found, `DBMS_CLOUD` searches for the UNIX/Linux newline character `\n`, and if it finds one it uses `\n` as the record delimiter for all files in the procedure. If the source files use a combination of different record delimiters, you may encounter an error such as, "`KUP-04020: found record longer than buffer size supported`". In this case, you need to either modify the source files to use the same record delimiter or only specify the source files that use the same record delimiter.
- The external partitioned tables that you create with `DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE` include two invisible columns, `file$path` and `file$name`. These columns help identify which file a record is coming from.
  - `file$path`: Specifies the file path text up to the beginning of the object name.
  - `file$name`: Specifies the object name, including all the text that follows the bucket name.

Examples

Example using the `partitioning_clause` parameter:

```
BEGIN
   DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
      table_name =>'PET1',
      credential_name =>'OBJ_STORE_CRED',
      format => json_object('delimiter' value ',', 'recorddelimiter' value
'newline', 'characterset' value 'us7ascii'),
      column_list => 'col1 number, col2 number, col3 number',
      partitioning_clause => 'partition by range (col1)
                              (partition p1 values less than (1000) location
                                  ( ''&base_URL//file_11.txt'')
                               ,
                               partition p2 values less than (2000) location
                                  ( ''&base_URL/file_21.txt'')
                               ,
                               partition p3 values less than (3000)
location
                                  ( ''&base_URL/file_31.txt'')
                               )'
```

```
        );
    END;
/
```

Example using the `file_uri_list` and `column_list` parameters with unstructured data files:

```
BEGIN
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
   table_name => 'MYSALES',
   credential_name => 'DEF_CRED_NAME',
   file_uri_list     => 'https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-
string/b/bucketname/o/*.csv',
   column_list       => 'product varchar2(100), units number, country varchar2(100), year
number, month varchar2(2)',
   field_list        => 'product, units', --[Because country, year and month are not in the
file, they are not listed in the field list]
   format            => '{"type":"csv", "partition_columns":["country","year","month"]}');
END;
/
```

Example using the `file_uri_list` without the `column_list` parameter with structured data files:

```
BEGIN
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
  table_name => 'MYSALES',
  credential_name => 'DEF_CRED_NAME',
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
    table_name      => 'MYSALES',
    credential_name => 'DEF_CRED_NAME',
    file_uri_list   => 'https://objectstorage.us-phoenix-1.oraclecloud.com/n/namespace-
string/b/bucketname/o/*.parquet',
    format          =>
        json_object('type' value 'parquet', 'schema' value 'first',
                    'partition_columns' value
                          json_array(
                                json_object('name' value 'country', 'type' value
'varchar2(100)'),
                                json_object('name' value 'year', 'type' value 'number'),
                                json_object('name' value 'month', 'type' value 'varchar2(2)')
                          )
         )
    );
END;
/
```

Example with a partitioned Apache Parquet source. You can run this example, because the data is public.

In this case, data is organized into months. The resource principal was enabled, as shown below. However, because this is a public data source, it is not required.

> **Note:**
>
> The list of columns is not required, because it is derived from the Parquet source. You do need to specify the data type for month, because there is no column list.

```
BEGIN
  DBMS_CLOUD.CREATE_EXTERNAL_PART_TABLE(
   credential_name => 'OCI$RESOURCE_PRINCIPAL',
   table_name => 'sales',
   file_uri_list => 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/
c4u04/b/moviestream_gold/o/custsales/*.parquet',
   format => '{"type":"parquet","partition_columns":
[{name:"month","type":"varchar2(20)"}]}'
   );
END;
/
mgubar: Finally, here is the generated ddl:
CREATE TABLE sales
   ( "DAY_ID" TIMESTAMP (6),
      "GENRE_ID" NUMBER(19,0),
      "MOVIE_ID" NUMBER(19,0),
      "CUST_ID" NUMBER(19,0),
      "APP" VARCHAR2(4000 BYTE),
      "DEVICE" VARCHAR2(4000 BYTE),
      "OS" VARCHAR2(4000 BYTE),
      "PAYMENT_METHOD" VARCHAR2(4000 BYTE),
      "LIST_PRICE" BINARY_DOUBLE,
      "DISCOUNT_TYPE" VARCHAR2(4000 BYTE),
      "DISCOUNT_PERCENT" BINARY_DOUBLE,
      "ACTUAL_PRICE" BINARY_DOUBLE,
      "MONTH" VARCHAR2(20 BYTE)
   )
   ORGANIZATION EXTERNAL
    ( TYPE ORACLE_BIGDATA
      DEFAULT DIRECTORY "DATA_PUMP_DIR"
      ACCESS PARAMETERS
       ( com.oracle.bigdata.fileformat=parquet
com.oracle.bigdata.filename.columns=["month"]
com.oracle.bigdata.file_uri_list="https://objectstorage.us-
ashburn-1.oraclecloud.com/n/c4u04/b/moviestream_gold/o/custsales/*.parquet"
com.oracle.bigdata.credential.schema="ADMIN"
com.oracle.bigdata.credential.name="OCI$RESOURCE_PRINCIPAL"
com.oracle.bigdata.trimspaces=notrim
  )
    )
   REJECT LIMIT 0
  PARTITION BY LIST ("MONTH")
 (PARTITION "P1" VALUES (('2019-01'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-01/*.parquet'
       ),
 PARTITION "P2" VALUES (('2019-02'))
      LOCATION
```

```
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-02/*.parquet'
       ),
 PARTITION "P3" VALUES (('2019-03'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-03/*.parquet'
       ),
 PARTITION "P4" VALUES (('2019-04'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2019-04/*.parquet'
       ),
 ...
 PARTITION "P24" VALUES (('2020-12'))
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_gold/o/custsales/month=2020-12/*.parquet'
       ))
  PARALLEL ;
```

Example of not requiring a field list. Parquet is a structured file. Because the file is Parquet, the field list is derived from the structured file.

```
CREATE TABLE ADMIN.EXT_CUSTSALES
   ( DAY_ID TIMESTAMP (6),
      GENRE_ID NUMBER(19,0),
      MOVIE_ID NUMBER(19,0),
      CUST_ID NUMBER(19,0),
      APP VARCHAR2(4000 BYTE),
      DEVICE VARCHAR2(4000 BYTE),
      OS VARCHAR2(4000 BYTE),
      PAYMENT_METHOD VARCHAR2(4000 BYTE),
      LIST_PRICE BINARY_DOUBLE,
      DISCOUNT_TYPE VARCHAR2(4000 BYTE),
      DISCOUNT_PERCENT BINARY_DOUBLE,
      ACTUAL_PRICE BINARY_DOUBLE
   ) DEFAULT COLLATION USING_NLS_COMP
   ORGANIZATION EXTERNAL
    ( TYPE ORACLE_BIGDATA
      DEFAULT DIRECTORY DATA_PUMP_DIR
      ACCESS PARAMETERS
       ( com.oracle.bigdata.fileformat=parquet
com.oracle.bigdata.trimspaces=notrim
  )
      LOCATION
       ( 'https://objectstorage.us-ashburn-1.oraclecloud.com/n/c4u04/b/
moviestream_landing/o/sales_sample/*.parquet'
       )
    )
   REJECT LIMIT UNLIMITED
  PARALLEL ;
```

Related Topics

- ORACLE_LOADER Access Driver field_list under field_definitions Clause
- DBMS_CLOUD Package Format Format Options in *Oracle Database PL/SQL Packages and Types Reference*

14.3 CREATE_EXTERNAL_TABLE Procedure

This procedure creates an external table on files in the Cloud or from files in a directory. This enables you to run queries on external data from Oracle Database.

Use Case

Starting with Oracle Database 19c, when you are using the `ORACLE_BIGDATA` driver with object stores, you are now able to select column values from a path in external tables.

Syntax

```
DBMS_CLOUD.CREATE_EXTERNAL_TABLE (
    table_name        IN VARCHAR2,
    credential_name   IN VARCHAR2,
    file_uri_list     IN CLOB,
    column_list       IN CLOB,
    field_list        IN CLOB DEFAULT,
    format            IN CLOB DEFAULT);
```

Parameters

Parameter	Description
`table_name`	The name of the external table.
`credential_name`	The name of the credential to access the Cloud Object Storage. This parameter is not used when you specify a directory with `file_uri_list`.
`file_uri_list`	Comma-delimited list of source file URIs. You can use wildcards in the file names in your URIs. The character "*" can be used as the wildcard for multiple characters, the character "?" can be used as the wildcard for a single character. The format of the URIs depend on the Cloud Object Storage service you are using. For details see: DBMS_CLOUD URI Formats
`column_list`	Comma-delimited list of column names and data types for the external table.
`field_list`	Identifies the fields in the source files and their data types. The default value is NULL meaning the fields and their data types are determined by the `column_list` parameter. This argument's syntax is the same as the `field_list` clause in regular Oracle external tables. For more information about `field_list` see: ORACLE_LOADER Access Driver field_list under field_definitions Clause.

Parameter	Description
format	The options describing the format of the source files. For the list of the options and how to specify the values see: DBMS_CLOUD Package Format Options in *Oracle Database PL/SQL Packages and Types Reference* For Avro or Parquet format files, see: DBMS_CLOUD CREATE_EXTERNAL_TABLE Procedure for Avro or Parquet Files in *Oracle Database PL/SQL Packages and Types Reference*

Usage Notes

The procedure DBMS_CLOUD.CREATE_EXTERNAL_TABLE supports external partitioned files in the supported cloud object storage services, including:

- Oracle Cloud Infrastructure Object Storage
- Azure Blob Storage
- Amazon S3

The credential is a table level property; therefore, the external files must be on the same object store.

See DBMS_CLOUD URI Formats in *Oracle Database PL/SQL Packages and Types Reference*

Example

```
BEGIN
   DBMS_CLOUD.CREATE_EXTERNAL_TABLE(
      table_name =>'WEATHER_REPORT_DOUBLE_DATE',
      credential_name =>'OBJ_STORE_CRED',
      file_uri_list =>'&base_URL/
Charlotte_NC_Weather_History_Double_Dates.csv',
      format => json_object('type' value 'csv', 'skipheaders' value '1'),
      field_list => 'REPORT_DATE DATE''mm/dd/yy'',
                    REPORT_DATE_COPY DATE ''yyyy-mm-dd'',
                    ACTUAL_MEAN_TEMP,
                    ACTUAL_MIN_TEMP,
                    ACTUAL_MAX_TEMP,
                    AVERAGE_MIN_TEMP,
                    AVERAGE_MAX_TEMP,
                    AVERAGE_PRECIPITATION',
      column_list => 'REPORT_DATE DATE,
                    REPORT_DATE_COPY DATE,
                    ACTUAL_MEAN_TEMP NUMBER,
                    ACTUAL_MIN_TEMP NUMBER,
                    ACTUAL_MAX_TEMP NUMBER,
                    AVERAGE_MIN_TEMP NUMBER,
                    AVERAGE_MAX_TEMP NUMBER,
                    AVERAGE_PRECIPITATION NUMBER');
   END;
/

SELECT * FROM WEATHER_REPORT_DOUBLE_DATE where
   actual_mean_temp > 69 and actual_mean_temp < 74
```

Related Topics

- DBMS_CLOUD in *Oracle Database PL/SQL Packages and Types Reference*

14.4 Location of Data Files and Output Files

Data files and output files must be located on the server. You must have a directory object that specifies the location from which to read and write files.

> **Note:**
>
> The information in this section about directory objects does not apply to data files for the `ORACLE_HDFS` access driver or `ORACLE_HIVE` access driver. With the `ORACLE_HDFS` driver, the location of data is specified with a list of URIs for a directory or for a file, and there is no directory object associated with a URI. The `ORACLE_HIVE` driver does not specify a data source location; it reads the Hive metastore table to get that information, so no directory object is needed.

The access driver runs inside the Oracle Database server. This behavior is different from SQL*Loader, which is a client program that sends the data to be loaded over to the server. This difference has the following implications:

- The server requires access to files that the access driver can load.
- The server must create and write the output files created by the access driver: the log file, bad file, discard file, and also any dump files created by the `ORACLE_DATAPUMP` access driver.

To specify the location from which to read and write files, the access driver requires that you use a **directory object**. A directory object maps a name to a directory name on the file system. For example, the following statement creates a directory object named `ext_tab_dir` that is mapped to a directory located at `/usr/apps/datafiles`.

```
CREATE DIRECTORY ext_tab_dir AS '/usr/apps/datafiles';
```

DBAs or any user can create directory objects with the `CREATE ANY DIRECTORY` privilege.

> **Note:**
>
> To use external tables in an Oracle Real Applications Cluster (Oracle RAC) configuration, you must ensure that the directory object path is on a cluster-wide file system.

After a directory is created, the user creating the directory object must grant `READ` and `WRITE` privileges on the directory to other users. These privileges must be explicitly granted, rather than assigned by using roles. For example, to allow the server to read files on behalf of user `scott` in the directory named by `ext_tab_dir`, the user who created the directory object must execute the following command:

```
GRANT READ ON DIRECTORY ext_tab_dir TO scott;
```

The Oracle Database SYS user is the only user that can own directory objects, but the SYS user can grant other Oracle Database users the privilege to create directory objects.READ or WRITE permission to a directory object means that only Oracle Database reads or writes that file on your behalf. You are not given direct access to those files outside of Oracle Database, unless you have the appropriate operating system privileges. Similarly, Oracle Database requires permission from the operating system to read and write files in the directories.

14.5 Access Parameters for External Tables

To modify the default behavior of the access driver for external tables, specify access parameters.

When you create an external table of a particular type, you can specify access parameters to modify the default behavior of the access driver. Each access driver has its own syntax for access parameters. Oracle provides the following access drivers for use with external tables: ORACLE_LOADER , ORACLE_DATAPUMP, ORACLE_HDFS, and ORACLE_HIVE.

> **Note:**
>
> These access parameters are collectively referred to as the opaque_format_spec in the SQL CREATE TABLE...ORGANIZATION EXTERNAL statement. The ACCESS parameter clause allows SQL comments.

Related Topics

- The ORACLE_LOADER Access Driver
- The ORACLE_DATAPUMP Access Driver
- CREATE TABLE in*Oracle Database SQL Language Reference*

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about specifying opaque_format_spec when using the SQL CREATE TABLE statement

14.6 Data Type Conversion During External Table Use

If source and target data types do not match, then conversion errors can occur when Oracle Database reads from external tables, and when it writes to external tables.

Conversion Errors When Reading External Tables

When you select rows from an external table, the access driver performs any transformations necessary to make the data from the data source match the data type of the corresponding column in the external table. Depending on the data and the types of transformations required, the transformation can encounter errors.

To illustrate the types of data conversion problems that can occur when reading from an external table, suppose you create the following external table, KV_TAB_XT, with two columns: KEY, whose data type is VARCHAR2(4), and VAL, whose data type is NUMBER.

```
SQL> CREATE TABLE KV_TAB_XT (KEY VARCHAR2(4), VAL NUMBER)
2 ORGANIZATION EXTERNAL
3 (DEFAULT DIRECTORY DEF_DIR1 LOCATION ('key_val.csv'));
```

The external table KV_TAB_XT uses default values for the access parameters. The following is therefore true:

- Records are delimited by new lines.
- The data file and the database have the same character set.
- The fields in the data file have the same name and are in the same order as the columns in the external table.
- The data type of the field is CHAR(255).
- Data for each field is terminated by a comma.

The records in the data file for the KV_TAB_XT external table should have the following:

- A string, up to 4 bytes long. If the string is empty, then the value for the field is NULL.
- A terminating comma.
- A string of numeric characters. If the string is empty, then the value for this field is NULL.
- An optional terminating comma.

When the access driver reads a record from the data file, it verifies that the length of the value of the KEY field in the data file is less than or equal to 4, and it attempts to convert the value of the VAL field in the data file to an Oracle Database number.

If the length of the value of the KEY field is greater than 4, or if there is a non-numeric character in the value for VAL, then the ORACLE_LOADER access driver rejects the row. The result is that a copy of the row is written to the bad file, and an error message is written to the log file.

All access drivers must handle conversion from the data type of fields in the source for the external table and the data type for the columns of the external tables. The following are some examples of the types of conversions and checks that access drivers perform:

- Convert character data from character set used by the source data to the character set used by the database.
- Convert from character data to numeric data.
- Convert from numeric data to character data.
- Convert from character data to a date or timestamp.
- Convert from a date or timestamp to character data.
- Convert from character data to an interval data type.
- Convert from an interval data type to a character data.
- Verify that the length of data value for a character column does not exceed the length limits of that column.

When the access driver encounters an error doing the required conversion or verification, it can decide how to handle the error. When the ORACLE_LOADER and ORACLE_DATAPUMP access drivers encounter errors, they reject the record, and write an error message to the log file. In

that event it is as if that record were not in the data source. When the ORACLE_HDFS and ORACLE_HIVE access drivers encounter errors, the value of the field in which the error is encountered is set to NULL. This action is consistent with the behavior of how Hive handles errors in Hadoop.

Even after the access driver has converted the data from the data source to match the data type of the external table columns, the SQL statement that is accessing the external table could require additional data type conversions. If any of these additional conversions encounter an error, then the entire statement fails. (The exception to this is if you use the DML error logging feature in the SQL statement to handle these errors.) These conversions are the same as any that typically can be required when running a SQL statement. For example, suppose you change the definition of the KV_TAB_XT external table to only have columns with character data types, and then you run an INSERT statement to load data from the external table into another table that has a NUMBER data type for column VAL:

```
SQL> CREATE TABLE KV_TAB_XT (KEY VARCHAR2(20), VAL VARCHAR2(20))
2 ORGANIZATION EXTERNAL
3 (DEFAULT DIRECTORY DEF_DIR1 LOCATION ('key_val.csv'));
4 CREATE TABLE KV_TAB (KEY VARCHAR2(4), VAL NUMBER);
5 INSERT INTO KV_TAB SELECT * FROM KV_TAB_XT;
```

In this example, the access driver will not reject a record if the data for VAL contains a non-numeric character, because the data type of VAL in the external table is now VARCHAR2 (instead of NUMBER). However, SQL processing now must handle the conversion from character data type in KV_TAB_XT to number data type in KV_TAB. If there is a non-numeric character in the value for VAL in the external table, then SQL raises a conversion error, and rolls back any rows that were inserted. To avoid conversion errors in SQL Oracle recommends that you make the data types of the columns in the external table match the data types expected by other tables or functions that will be using the values of those columns.

Conversion Errors When Writing to External Tables

The ORACLE_DATAPUMP access driver allows you to use a CREATE TABLE AS SELECT statement to unload data into an external table. Data conversion occurs if the data type of a column in the SELECT expression does not match the data type of the column in the external table. If SQL encounters an error while converting the data type, then SQL stops the statement, and the data file will not be readable.

To avoid problems with conversion errors that cause the operation to fail, the data type of the column in the external table should match the data type of the column in the source table or expression used to write to the external table. This is not always possible, because external tables do not support all data types. In these cases, the unsupported data types in the source table must be converted into a data type that the external table can support. The following CREATE TABLE statement shows an example of this conversion:

```
CREATE TABLE LONG_TAB_XT (LONG_COL CLOB)
ORGANIZATION EXTERNAL...SELECT TO_LOB(LONG_COL) FROM LONG_TAB;
```

The source table named LONG_TAB has a LONG column. Because of that, the corresponding column in the external table being created, LONG_TAB_XT, must be a CLOB, and the SELECT subquery that is used to populate the external table must use the TO_LOB operator to load the column.

> **Note:**
>
> All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

14.7 Vectors in External Tables

External tables can be created with columns of type `VECTOR`, allowing vector embeddings represented in text or binary format stored in external files to be rendered as the `VECTOR` data type in Oracle Database.

The ability to store `VECTOR` data in external tables can be beneficial when considering the vast amount of data that is involved in AI workflows. Using external tables provides a convenient option to store vector embeddings and contextual information outside of the database while still using the database as a semantic search engine.

The following types of external files support `VECTOR` columns in external tables:

- CSV
- Parquet
- Avro
- ORC
- Dmp

External tables with `VECTOR` columns can be accessed by using the drivers `ORACLE_LOADER`, `ORACLE_DATAPUMP`, and `ORACLE_BIGDATA`. Once the chosen driver has loaded the file data into the database, you can interact with the external table rows in any SQL operation (supported by your preferred SQL interface), such as joins, SQL functions, aggregation, and so on.

Vectors of any dimension format and storage format are supported. If a vector is `SPARSE`, the data must be provided as text as opposed to an array or list format.

Columns of type `VECTOR` can be included in both explicitly created external tables as well as inline external tables created as part of a `SELECT` statement. The benefit of this approach is that there is no need to predefine a static table to access the vectors in the external table before loading them into the database. The external tables mapping is persisted only while the external table is in use by the SQL query. For an example, see Querying an Inline External Table, which shows how the external table mappings are created as part of the SQL query operation. Once the query has completed, the external table mapping is discarded from the database.

Additionally, the `row_limiting_clause` can be used in `SELECT` statements that reference external tables. Internal and external tables can be referenced in the same query. You can use the `CREATE TABLE AS SELECT` statement to create an internal table by selecting from an external table with `VECTOR` column(s). Similarly, you can use the `INSERT INTO SELECT` statement to insert values into an internal table from an external table called in the `SELECT` subquery.

> **Note:**
>
> - External tables are not currently supported in multi-vector similarity searches.
> - HNSW and IVF indexes cannot currently be created on `VECTOR` columns stored in external tables.

Vector embeddings in external tables can be accessed for use in similarity searches in the same way as you would use an internal table, as in the following query:

```
SELECT id, embedding
FROM external_table
ORDER BY VECTOR_DISTANCE(embedding, '[1,1]', COSINE)
FETCH APPROX FIRST 3 ROWS ONLY WITH TARGET ACCURACY 90;
```

The following examples demonstrate the syntax used to create external tables with `VECTOR` columns depending on the access driver:

- Using `ORACLE_LOADER`:

```
CREATE TABLE ext_vec_tab1(
  v1 VECTOR,
  v2 VECTOR
)
  ORGANIZATION EXTERNAL
  (
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY my_dir
    ACCESS PARAMETERS
    (
      RECORDS DELIMITED BY NEWLINE
      FIELDS TERMINATED BY ":"
      MISSING FIELD VALUES ARE NULL
    )
    LOCATION('my_ext_vec_embeddings.csv')
  )
REJECT LIMIT UNLIMITED;
```

- Using `ORACLE_DATAPUMP`:

```
-- First create the table with the loader
CREATE TABLE dp_ext_tab(
  country_code         VARCHAR2(5),
  country_name         VARCHAR2(50),
  country_language     VARCHAR2(50),
  country_vector       VECTOR(*,*)
)
  ORGANIZATION EXTERNAL
  (
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY my_dir
    ACCESS PARAMETERS
    (
```

```
      RECORDS DELIMITED BY NEWLINE
      FIELDS TERMINATED BY ":"
      MISSING FIELD VALUES ARE NULL
      (
        country_code         CHAR(5),
        country_name         CHAR(50),
        country_language     CHAR(50),
        country_vector       CHAR(10000)
      )
    )
    LOCATION ('ext_vectorcountries.dat')
    )
  PARALLEL 5
  REJECT LIMIT UNLIMITED;

-- Then generate the dmp file
CREATE TABLE ext_export_table
ORGANIZATION EXTERNAL
(
  TYPE ORACLE_DATAPUMP
  DEFAULT DIRECTORY my_dir
  LOCATION ('ext.dmp')
)
  AS SELECT * FROM dp_ext_tab;

-- Finally, create an external table with the datapump driver
CREATE TABLE dp_ext_tab_final
(
  country_code         VARCHAR2(5),
  country_name         VARCHAR2(50),
  country_language     VARCHAR2(50),
  country_vector       VECTOR(3, INT8)
)
  ORGANIZATION EXTERNAL
  (
    TYPE ORACLE_DATAPUMP
    DEFAULT DIRECTORY my_dir
    LOCATION ('ext.dmp')
  )
  PARALLEL 5
  REJECT LIMIT UNLIMITED;
```

- Using `ORACLE_BIGDATA`:

```
CREATE TABLE bd_ext_tab
(
  COL1 vector(5,INT8),
  COL2 vector(5,INT8),
  COL3 vector(5,INT8),
  COL4 vector(5,INT8)
)
  ORGANIZATION external
  (
    TYPE ORACLE_BIGDATA
    DEFAULT DIRECTORY my_dir
    ACCESS PARAMETERS
```

```
  (
    com.oracle.bigdata.credential.name\=OCI_CRED
    com.oracle.bigdata.credential.schema\=PDB_ADMIN
    com.oracle.bigdata.fileformat=parquet
    com.oracle.bigdata.debug=true
  )
  LOCATION ( 'https://swiftobjectstorage.<region>.oraclecloud.com/v1/
<namespace>/<filepath>/basic_vec_data.parquet' )
  )
REJECT LIMIT UNLIMITED PARALLEL 2;
```

See Also:

- *Oracle Database Utilities* for more information about external tables