19
DBFS SecureFiles Store

There are certain procedures for setting up and using a DBFS SecureFiles Store.

- Setting Up a SecureFiles Store
  This section shows how to set up a SecureFiles Store.
- Using a DBFS SecureFiles Store File System
  The DBFS Content API provides methods to access and manage a SecureFiles Store file system.
- About DBFS SecureFiles Store Package, DBMS_DBFS_SFS
  The SecureFiles Store provider is a default implementation of the DBFS Content API (and is a standard example of a store provider that conforms to the Provider SPI) .
- Database File System (DBFS)— POSIX File Locking
  Starting from Oracle Database 12*c* Release 2(12.2), Oracle supports the Database File system `POSIX` File locking feature.

19.1 Setting Up a SecureFiles Store

This section shows how to set up a SecureFiles Store.

- About Managing Permissions
  You must be a non-SYS database user for all operational access to the Content API and stores.
- Creating or Setting Permissions
  You must grant the `DBFS_ROLE` role to any user that needs to use the DBFS content API.
- Accessing SecureFiles Store
  You should never directly access tables that hold data for a SecureFiles Store file systems.
- Reinitializing SecureFiles Store File Systems
  You can truncate and re-initialize tables associated with an SecureFiles Store.
- Comparison of SecureFiles LOBs to BasicFiles LOBs
  SecureFiles LOBs are only available in Oracle Database 11*g* Release 1 and higher. They are not available in earlier releases.

19.1.1 About Managing Permissions

You must be a non-SYS database user for all operational access to the Content API and stores.

Do not use `SYS` or `SYSTEM` users or `SYSDBA` or `SYSOPER` system privileges. For better security and separation of duty, only allow specific trusted users to access DBFS Content API.

You must grant each user the `DBFS_ROLE` role. Otherwise, the user is not authorized to use the DBFS Content API. A user with suitable administrative privileges (or SYSDBA) can grant the role to additional users as needed.

The `CREATEFILESYSTEM` procedure auto-commits before and after its execution (like a DDL). The method `CREATESTORE` is a wrapper around `CREATEFILESYSTEM`.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for `DBMS_DBFS_SFS` syntax details

19.1.2 Creating or Setting Permissions

You must grant the `DBFS_ROLE` role to any user that needs to use the DBFS content API.

1. Create or determine DBFS Content API target users.

   This example uses this user and password: `sfs_demo/password`

   At minimum, this database user must have the `CREATE SESSION`, `CREATE RESOURCE`, and `CREATE VIEW` privileges.

2. Grant the `DBFS_ROLE` role to the user.

```
CONNECT / as sysdba
GRANT dbfs_role TO sfs_demo;
```

This sets up the DBFS Content API for any database user who has the `DBFS_ROLE` role.

19.1.3 Accessing SecureFiles Store

You should never directly access tables that hold data for a SecureFiles Store file systems.

This is the correct way to access the file systems.

- For procedural operations: Use the DBFS Content API (`DBMS_DBFS_CONTENT` methods).
- For SQL operations: Use the resource and property views (`DBFS_CONTENT` and `DBFS_CONTENT_PROPERTIES`).

19.1.4 Reinitializing SecureFiles Store File Systems

You can truncate and re-initialize tables associated with an SecureFiles Store.

- Use the procedure `INITFS()`.

  The procedure executes like a DDL, auto-committing before and after its execution.

The following example uses file system `FS1` and table `SFS_DEMO.T1`, which is associated with the SecureFiles Store `store_name`.

```
CONNECT sfs_demo;
Enter password: password
EXEC DBMS_DBFS_SFS.INITFS(store_name => 'FS1');
```

19.1.5 Comparison of SecureFiles LOBs to BasicFiles LOBs

SecureFiles LOBs are only available in Oracle Database 11*g* Release 1 and higher. They are not available in earlier releases.

You must use BasicFiles LOB storage for LOB storage in tablespaces that are not managed with Automatic Segment Space Management (ASSM).

Compatibility must be at least 11.1.0.0 to use SecureFiles LOBs.

Additionally, you need to specify the following in `DBMS_DBFS_SFS.CREATEFILESYSTEM`:

- To use SecureFiles LOBs (the default), specify `use_bf => false`.
- To use BasicFiles LOBs, specify `use_bf => true`.

19.2 Using a DBFS SecureFiles Store File System

The DBFS Content API provides methods to access and manage a SecureFiles Store file system.

- DBFS Content API Working Example
  You can create new file and directory elements to populate a SecureFiles Store file system.
- Dropping SecureFiles Store File Systems
  You can use the `unmountStore` method to drop SecureFiles Store file systems.

19.2.1 DBFS Content API Working Example

You can create new file and directory elements to populate a SecureFiles Store file system.

If you have executed the steps in "Setting Up a SecureFiles Store", set the DBFS Content API permissions, created at least one SecureFiles Store reference file system, and mounted it under the mount point `/mnt1`, then you can create a new file and directory elements as demonstrated in Example 19-1.

Example 19-1 Working with DBFS Content API

```
CONNECT tjones
Enter password: <password>

DECLARE
   ret INTEGER;
   b   BLOB;
   str VARCHAR2(1000)  := '' || chr(10) ||
    '#include <stdio.h>' || chr(10) ||
    '' || chr(10) ||
    'int main(int argc, char** argv)' || chr(10) ||
    '{' || chr(10) ||
    '    (void) printf("hello world\n");' || chr(10) ||
    '    RETURN 0;' || chr(10) ||
    '}' || chr(10) ||
    '';
   properties      DBMS_DBFS_CONTENT.PROPERTIES_T;

BEGIN
   properties('posix:mode') := DBMS_DBFS_CONTENT.propNumber(16777);
                                          --  drwxr-xr-x  --
   properties('posix:uid')  := DBMS_DBFS_CONTENT.propNumber(0);
   properties('posix:gid')  := DBMS_DBFS_CONTENT.propNumber(0);
   DBMS_DBFS_CONTENT.createDirectory(
            '/mnt1/FS1',
            properties);
```

```
   properties('posix:mode') := DBMS_DBFS_CONTENT.propNumber(33188);
                                                --  -rw-r--r--  --
   DBMS_DBFS_CONTENT.createFile(
            '/mnt1/FS1/hello.c',
            properties,
            b);

    DBMS_LOB.writeappend(b, length(str), utl_raw.cast_to_raw(str));
    COMMIT;
END;
/
SHOW ERRORS;

-- verify newly created directory and file

SELECT pathname, pathtype, length(filedata),
      utl_raw.cast_to_varchar2(filedata)
      FROM dbfs_content
         WHERE pathname LIKE '/mnt1/FS1%'
         ORDER BY pathname;
```

The file system can be populated and accessed from PL/SQL with `DBMS_DBFS_CONTENT`. The file system can be accessed read-only from SQL using the `dbfs_content` and `dbfs_content_properties` views.

The file system can also be populated and accessed using regular file system APIs and UNIX utilities when mounted using FUSE, or by the standalone `dbfs_client` tool (in environments where `FUSE` is either unavailable or not set up).

> **See Also:**
>
> DBFS Client Access Methods

19.2.2 Dropping SecureFiles Store File Systems

You can use the `unmountStore` method to drop SecureFiles Store file systems.

This method removes all stores referring to the file system from the metadata tables, and drops the underlying file system table. The procedure executes like a DDL, auto-committing before and after its execution.

1. Unmount the store.

```
CONNECT sfs_demo/<password>

DECLARE
  BEGIN
    DBMS_DBFS_CONTENT.UNMOUNTSTORE(
      store_name    => 'FS1',
      store_mount   => 'mntl';
    );
    COMMIT;
END;
/
```

where:

- `store_name` is `FS1`, a case-sensitive unique username.
- `store_mount` is the mount point.

2. Unregister the stores.

```
CONNECT sfs_demo/<password>
EXEC DBMS_DBFS_CONTENT.UNREGISTERSTORE(store_name => 'FS1');
COMMIT;
```

where `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.

3. Drop the store.

```
CONNECT sfs_demo/<password>;
EXEC DBMS_DBFS_SFS.DROPFILESYSTEM(store_name => 'FS1');
COMMIT;
```

where `store_name` is SecureFiles Store `FS1`, which uses table `SFS_DEMO.T1`.

19.3 About DBFS SecureFiles Store Package, DBMS_DBFS_SFS

The SecureFiles Store provider is a default implementation of the DBFS Content API (and is a standard example of a store provider that conforms to the Provider SPI) .

To use the `DBMS_DBFS_SFS` package, you must be granted the `DBFS_ROLE` role.

The SecureFiles Store provider is a default implementation of the DBFS Content API (and is a standard example of a store provider that conforms to the Provider SPI) . This enables existing applications to easily add PL/SQL provider implementations and provide access through the DBFS Content API without changing their schemas or their business logic.

See Also:

- See *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_DBFS_SFS` package.
- Creating Your Own DBFS Store and *Oracle Database PL/SQL Packages and Types Reference* for more information about the Provider SPI defined in `DBMS_DBFS_CONTENT_SPI`.
- Introduction to Large Objects and SecureFiles for advanced features of SecureFiles LOBs.

19.4 Database File System (DBFS)— POSIX File Locking

Starting from Oracle Database 12*c* Release 2(12.2), Oracle supports the Database File system `POSIX` File locking feature.

The DBFS provides file locking support for the following types of applications:

- `POSIX` applications using `DBFS_CLIENT` (in mount mode) as a front-end interface to DBFS.

> **See Also:**
>
> DBFS Client Access Methods

- Applications using `PL/SQL` as an interface to DBFS.

> **Note:**
>
> Oracle supports only Full-file locks in DBFS. Full-file lock implies locking the entire file from byte zero offset to the end of file.

- About Advisory Locking
  Advisory locking is a file locking mechanism that locks the file for a single process.
- About Mandatory Locking
  Mandatory locking is a file locking mechanism that takes support from participating processes.
- File Locking Support
  Enabling the file locking mechanism helps applications to block files for various file system operations.
- Compatibility and Migration Factors of Database Filesystem—File Locking
  The Database Filesystem File Locking feature does not impact the compatibility of `DBFS` and `SFS` store provider with `RDBMS`.
- Examples of Database File System—File Locking
  These examples illustrate the advisory locking and the locking functions available on `UNIX` based systems.
- DBFS Locking Behavior
  This section describes the DBFS locking behavior.
- Scheduling File Locks
  DBFS File Locking feature supports lock scheduling.

19.4.1 About Advisory Locking

Advisory locking is a file locking mechanism that locks the file for a single process.

File locking mechanism cannot independently enforce any form of locking and requires support from the participating processes. For example, if a process `P1` has a `write` lock on file `F1`, the locking API or the operating system does not perform any action to prevent any other process `P2` from issuing a `read` or `write` system call on the file `F1`. This behavior of file locking mechanism is also applicable to other file system operations. The processes that are involved (in file locking mechanism) must follow a lock or unlock protocol provided in a suitable API form by the user-level library. File locking semantics are guaranteed to work as per POSIX standards.

19.4.2 About Mandatory Locking

Mandatory locking is a file locking mechanism that takes support from participating processes.

Mandatory locking is an enforced locking scheme that does not rely on the participating processes to cooperate and/or follow the locking API. For example, if a process `P1` has taken a

write lock on file F1 and if a different process P2 attempts to issue a read/write system call (or any other file system operation) on file F1 , the request is blocked because the concerned file is exclusively locked by process P1.

19.4.3 File Locking Support

Enabling the file locking mechanism helps applications to block files for various file system operations.

The fcntl(), lockf(), and flock() system calls in UNIX and LINUX provide file locking support. These system calls enable applications to use the file locking facility through dbfs_client-FUSE callback interface. File Locks provided by fcntl() are widely known as POSIX file locks and the file locks provided by flock() are known as BSD file locks. The semantics and behavior of POSIX and BSD file locks differ from each other. The locks placed on the same file through fcntl() and flock() are orthogonal to each other. The semantics of file locking functionality designed and implemented in DBFS is similar to POSIX file locks. In DBFS, semantics of file locks placed through flock() system call will be similar to POSIX file locks (such as fcntl()) and not BSD file locks. lockf() is a library call that is implemented as a wrapper over fcntl() system call on most of the UNIX systems, and hence, it provides POSIX file locking semantics. In DBFS, file locks placed through fcntl(), flock(), and lockf() system-calls provide same kind of behavior and semantics of POSIX file locks.

> **Note:**
>
> BSD file locking semantics are not supported.

19.4.4 Compatibility and Migration Factors of Database Filesystem—File Locking

The Database Filesystem File Locking feature does not impact the compatibility of DBFS and SFS store provider with RDBMS.

DBFS_CLIENT is a standalone OCI Client and uses OCI calls and DBMS_FUSE API.

> **Note:**
>
> This feature will be compatible with OraSDK/RSF .

19.4.5 Examples of Database File System—File Locking

These examples illustrate the advisory locking and the locking functions available on UNIX based systems.

The following example uses two running processes — Process A and Process B.

Example 19-2 No locking

```
Process A opens file:

file_desc = open("/path/to/file", O_RDONLY);
/* Reads data into bufffers */
read(fd, buf1, sizeof(buf));
read(fd, buf2, sizeof(buf));
close(file_desc);
```

Subjected to OS scheduling, `process B` can enter any time and issue a `write` system call affecting the integrity of file data.

Example 19-3 Advisory locking used but process B does not follow the protocol

```
Process A opens file:

file_desc = open("/path/to/file", O_RDONLY);
ret = AcquireLock(file_desc, RD_LOCK);
if(ret)
{
    read(fd, buf1, sizeof(buf));
    read(fd, buf2, sizeof(buf));
    ReleaseLock(file_desc);
}
close(file_desc);
```

Subjected to OS scheduling, `process B` can come in any time and still issue a `write` system call ignoring that `process A` already holds a `read` lock.

```
Process B opens file:

file_desc1 = open("/path/to/file", O_WRONLY);
write(file_desc1, buf, sizeof(buf));
close(file_desc1);
```

The above code is executed and leads to inconsistent data in the file.

Example 19-4 Advisory locking used and processes are following the protocol

```
Process A opens file:

file_desc = open("/path/to/file", O_RDONLY);
ret = AcquireLock(file_desc, RD_LOCK);
if(ret)
{
    read(fd, buf1, sizeof(buf));
    read(fd, buf2, sizeof(buf));
    ReleaseLock(file_desc);
}
close(file_desc);
```

```
Process B opens file:

file_desc1 = open("/path/to/file", O_WRONLY);
ret = AcquireLock(file_desc1, WR_LOCK);
/* The above call will take care of checking the existence of a lock */
if(ret)
{
    write(file_desc1, buf, sizeof(buf));
    ReleaseLock(file_desc1);
} close(file_desc1);
```

`Process B` follows the lock API and this API makes sure that the process does not `write` to the file without acquiring a lock.

19.4.6 DBFS Locking Behavior

This section describes the DBFS locking behavior.

The `DBFS` File Locking feature exhibits the following behaviors:

- File locks in `DBFS` are implemented with idempotent functions. If a process issues "N" `read` or `write` lock calls on the same file, only the first call will have an effect, and the subsequent "N-1" calls will be treated as redundant and returns No Operation (NOOP).
- File can be unlocked exactly once. If a process issues "N" `unlock` calls on the same file, only the first call will have an effect, and the subsequent "N-1" calls will be treated as redundant and returns NOOP.
- Lock conversion is supported only from `read` to `write`. If a process `P` holds a `read` lock on file `F` (and `P` is the only process holding the `read` lock), then a `write` lock request by `P` on file `F` will convert the `read` lock to `exclusive`/`write` lock.

19.4.7 Scheduling File Locks

DBFS File Locking feature supports lock scheduling.

This facility is implemented purely on the DBFS client side. Lock request scheduling is required when client application uses blocking call semantics in their `fcntl()`, `lockf()`, and `flock()` calls.

There are two types of scheduling:

- Greedy Scheduling
- Fair Scheduling

Oracle provides the following command line option to switch the scheduling behavior.

```
Mount -o lock_sched_option = lock_sched_option Value;
```

Table 19-1 `lock_sched_option` Value Description

Value	Description
1	Sets the scheduling type to Greedy Scheduling. (Default)
2	Sets the scheduling type to Fair Scheduling.

> **Note:**
>
> Lock Request Scheduling works only on per `DBFS_CLIENT` mount basis. For example, lock requests are not scheduled across multiple mounts of the same file system.

- Greedy Scheduling
  In this scheduling technique, the file lock requests does not follow any guaranteed order.
- Fair Scheduling
  The fair scheduling technique is implemented using a queuing mechanism on per file basis.

19.4.7.1 Greedy Scheduling

In this scheduling technique, the file lock requests does not follow any guaranteed order.

> **Note:**
>
> This is the default scheduling option provided by `DBFS_CLIENT`.

If a file `F` is `read` locked by process `P1`, and if processes `P2` and `P3` submit blocking `write` lock requests on file `F`, the processes `P2` and `P3` will be blocked (using a form of spin lock) and made to wait for its turn to acquire the lock. During the wait, if a process `P4` submits a `read` lock request (blocking call or a non-blocking call) on file `F`, `P4` will be granted the `read` lock even if there are two processes (`P2` and `P3`) waiting to acquire the `write` lock. Once both `P1` and `P4` release their respective `read` locks, one of `P2` and `P3` will succeed in acquiring the lock. But, the order in which processes `P2` and `P3` acquire the lock is not determined. It is possible that process `P2` would have requested first, but the process `P3`'s request might get unblocked and acquire the lock and the process `P2` must wait for `P3` to release the lock.

19.4.7.2 Fair Scheduling

The fair scheduling technique is implemented using a queuing mechanism on per file basis.

For example, if a file `F` is read locked by process `P1`, and processes `P2` and `P3` submit blocking write lock requests on file `F`, these two processes will be blocked (using a form of spin lock) and will wait to acquire the lock. The requests will be queued in the order received by the DBFS client. If a process `P4` submits a read lock request (blocking call or a non-blocking call) on file `F`, this request will be queued even though a read lock can be granted to this process.

DBFS Client ensures that after `P1` releases its read lock, the order in which lock requests are honored is P2->P3 -> P4.

This implies that `P2` will be the first one to get the lock. Once `P2` releases its lock, `P3` will get the lock and so on.