1. The POSTGRES Storage Manager Switch

The POSTGRES abstraction that hides device-specific characteristics from the data manager is called the *storage manager switch*. The storage manager switch is similar to the *cdevsw* and *bdevsw* interfaces of UNIX and the user-supplied backing store interface in Mach. This section describes the POSTGRES storage manager switch in detail.

Hiding device characteristics from the data manager required that the abstract operations required on the data store be identified. For example, the interface requires routines for opening relations and instantiating particular 8kByte relation blocks. Every device is managed by a *device manager*, which knows how to implement the abstract operations on the particular device. The *storage manager* is the module that chooses the correct device manager to service a particular request.

Once the interfaces required of the device manager were defined, an appropriate structure (the *storage manager switch table*) was compiled into POSTGRES. This structure contains an entry for all existing device managers. New device types are added by editing a source file, adding appropriate entries to the storage manager switch table, and recompiling POSTGRES.

Whenever a device operation is required, the data manager calls the storage manager to carry it out. Every relation is tagged in POSTGRES with the device manager on which it appears. The storage manager identifies the appropriate device and routine in the storage manager switch table, and calls the device manager to execute the routine on behalf of the data manager.

Data may be located on a particular device manager at the granularity of a relation. Physical relations may not be horizontally or vertically partitioned, but the POSTGRES rules system can be used to create a partitioned logical relation.

1.1. Isolation of Device Differences

New device managers are written to manage new types of devices. For example, main memory, magnetic disk, and the Sony optical disk jukebox are all managed by different device managers. To isolate the database system from the underlying devices, device managers are required to accept and return data in the format used internally by the data manager. This means, among other things, that data pages from user relations must be accepted and returned in units of 8kBytes.

Nevertheless, there is no requirement that a device manager use the internal representation when writing a page to persistent storage. Data pages may be compressed, broken into pieces, or concatenated into larger units by a device manager, as long as the original page can be reconstructed on demand.

In addition, device managers may implement caches managed according to an appropriate policy. This means that the new POSTGRES architecture supports a multi-level cache. A main-memory cache of recently-used blocks is maintained in shared memory by the buffer manager. The buffer manager calls the storage manager to migrate pages between shared memory and persistent storage. A device manager may implement a separate cache for its own use, subject to whatever policy it likes. For example, the Sony jukebox device manager maintains a cache of *extents* (contiguous collections of user data blocks) on magnetic disk. If there is locality of reference in accesses to a Sony jukebox relation, then some requests may be satisfied without accessing the jukebox device at all. This cache will be described in more detail in section 4.2.

1.2. The Storage Manager Switch Interface

This section defines the interface presented by the storage manager switch.

The storage manager switch data structure, `f_smgr`, contains

```
typedef struct f_smgr {
    int         (*smgr_init)();         /* may be NULL */
    int         (*smgr_shutdown)();     /* may be NULL */
    int         (*smgr_create)();
    int         (*smgr_unlink)();
    int         (*smgr_extend)();
    int         (*smgr_open)();
    int         (*smgr_close)();
    int         (*smgr_read)();
    int         (*smgr_write)();
    int         (*smgr_flush)();
    int         (*smgr_blindwrt)();
    int         (*smgr_nblocks)();
    int         (*smgr_commit)();       /* may be NULL */
    int         (*smgr_abort)();        /* may be NULL */
    int         (*smgr_cost)();
} f_smgr;
```

This structure defines the interface routines that all device managers must define. When a new device type is added to POSTGRES, this structure is filled in with function pointers appropriate to the device, and POSTGRES is recompiled. The storage manager calls these routines in response to requests from the data manager. In some cases, the switch entry for an interface routine may be `NULL`. This means that the corresponding operation is not defined for the device in question. Unless otherwise stated in the sections that follow, all interface routines return an integer status code indicating success or failure.

The following table summarizes the responsibilities of these interface routines.

Routine	*Purpose*
smgr_init()	Called at startup to allow initialization.
smgr_shutdown()	Called at shutdown to allow graceful exit.
smgr_create(*r*)	Create relation *r*.
smgr_unlink(*r*)	Destroy relation *r*.
smgr_extend(*r*, *buf*)	Add a new block to the end of relation *r* and fill it with data from *buf*.
smgr_open(*r*)	Open relation *r*. The relation descriptor includes a pointer to the state for the open object.
smgr_close(*r*)	Close relation *r*.
smgr_read(*r*, *block*, *buf*)	Read block *block* of relation *r* into the buffer pointed to by *buf*.
smgr_write(*r*, *block*, *buf*)	Write *buf* as block *block* of relation *r*.
smgr_flush(*r*, *block*, *buf*)	Write *buf* synchronously as block *block* of relation *r*.
smgr_blindwrt(n_d, n_r, i_d, i_r, *blk*, *buf*)	Do a "blind write" of buffer *buf* as block *blk* of the relation named n_r in the database named n_d.
smgr_nblocks(*r*)	Return the number of blocks in relation *r*.
smgr_commit()	Force all dirty data to stable storage.
smgr_abort()	Changes made during this transaction may be discarded.

The rest of this section gives more detailed descriptions.

1.2.1. smgr_init

The routine

```
int
smgr_init()
```

is called when POSTGRES starts up, and should initialize any private data used by a device manager. In addition, the first call to this routine should initialize any shared data. Typically, this is done by noticing whether shared data exists, and creating and initializing it if it does not. If a particular device manager requires no initialization, it may define this routine to be NULL.

1.2.2. smgr_shutdown

The routine

```
int
smgr_shutdown()
```

is analogous to `smgr_init()`, but is called when POSTGRES exits cleanly. Any necessary cleanup may be done here. If no work is required, this routine may be NULL.

1.2.3. smgr_create

The routine to create new relations on a particular device manager is

```
int
smgr_create(reln)
    Relation reln;
```

The `reln` argument points to a POSTGRES relation descriptor for the relation to be created. The device manager should initialize whatever storage structure is required for the new relation. For example, the magnetic disk device manager creates an empty file with the name of the relation.

1.2.4. smgr_unlink

The routine

```
int
smgr_unlink(reln)
    Relation reln;
```

is called by the storage manager to destroy the relation described by `reln`. Device managers may take whatever action is appropriate. For example, the magnetic disk device manager removes the relation's associated file and archive, and reclaims the disk blocks they occupy. Since the Sony write-once drive cannot reclaim blocks once they are used, the Sony jukebox device manager simply marks the relation as destroyed and returns.

1.2.5. smgr_extend

When a relation must be extended with a new data block, the data manager calls

```
int
smgr_extend(reln, buffer)
    Relation reln;
    char *buffer;
```

to do the work. The buffer is an 8kByte buffer in the format used by the data manager internally. The data manager assumes that 8kByte data blocks in relations are numbered starting from one. Thus `smgr_extend()` must logically allocate a new block for the relation pointed to by `reln` and store some representation of `buffer` there.

1.2.6. smgr_open

The routine

```
int
smgr_open(reln)
    Relation reln;
```

should open the relation pointed to by `reln` and return a file descriptor for it. The notion of file descriptors is a holdover from the pre-storage manager switch days, and is no longer necessary. If file descriptors make no sense for a given device, then the associated device manager may return any non-negative value. A negative return value indicates an error.

1.2.7. smgr_close

When the data manager is finished with a relation, it calls

```
int
smgr_close(reln)
    Relation reln;
```

The device manager may release resources associated with the relation pointed to by `reln`.

1.2.8. smgr_read

To instantiate an 8kByte data block from a relation, the data manager calls

```
int
smgr_read(reln, blocknum, buffer)
    Relation reln;
    BlockNumber blocknum;
    char *buffer;
```

As stated above, 8kByte blocks in a relation are logically numbered from one. The storage manager must locate the block of interest and load its contents into `buffer`.

1.2.9. smgr_write

The routine

```
int
smgr_write(reln, blocknum, buffer)
    Relation reln;
    BlockNumber blocknum;
    char *buffer;
```

writes some representation of `buffer` as block number `blocknum` of the relation pointed to by `reln`. This write may be asynchronous; the device manager need not guarantee that the buffer is flushed through to stable storage. Synchronous writes are handled using the `smgr_flush` routine, described below. As long as this routine guarantees that the buffer has been copied somewhere safe for eventual writing, it may return successfully.

The buffer is an 8kByte POSTGRES page in the format used by the data manager. It may be stored in any form, but must be reconstructed exactly when the `smgr_read` routine is called on it.

1.2.10. smgr_flush

This routine synchronously writes a block to stable storage. The POSTGRES data manager almost never needs to do synchronous writes of single blocks. In some cases, however, like the write of the transaction status file that marks a transaction committed, a single block must be written synchronously. In this case, the data manager calls the device manager's `smgr_flush` routine. The interface for this routine is

```
int
smgr_flush(reln, blocknum, buffer)
    Relation reln;
    BlockNumber blocknum;
    char *buffer;
```

The behavior of `smgr_flush` is almost exactly the same as that of `smgr_write`, except that `smgr_flush` must not return until `buffer` is safely written to stable storage.

1.2.11. smgr_blindwrt

Under normal circumstances, POSTGRES moves dirty pages from memory to stable storage by calling the `smgr_write` routine for the appropriate storage manager. In order to call `smgr_write`, the buffer manager must construct and pass a relation descriptor.

In certain cases, POSTGRES is unable to construct the relation descriptor for a dirty buffer that must be evicted from the shared buffer cache. This can happen, for example, when the relation descriptor has just been created in another process, and has not yet been committed to the database. In such a case, POSTGRES must write the buffer without constructing a relation descriptor.

The buffer manager can detect this case, and handles it using a strategy called "blind writes." When a blind write must be done, the buffer manager determines the name and object id of the database, the name and object id of the relation, and the block number of the buffer to be written. All of this information is stored with the buffer in the buffer cache. The device manager is responsible for determining, from this information alone, where the buffer must be written. This means, in general, that every device manager must use some subset of these values as a unique key for finding the proper location to write a data block.

In general, the blind write routine is slower than the ordinary write routine, since it must assemble enough information to synchronously open the relation and write the data.

The interface for this routine is

```
int
smgr_blindwrt(dbname, relname, dbid, relid, blkno,
              buffer)
    Name dbname;
    Name relname;
    ObjectId dbid;
    ObjectId relid;
    BlockNumber blkno;
    char *buffer;
```

It is expected that most device managers will write data much more efficiently using `smgr_write` than `smgr_blindwrt`.

1.2.12. smgr_nblocks

The routine

```
int
```

```
smgr_nblocks(reln)
    Relation reln;
```

should return the number of blocks stored for the relation pointed to by `reln`. The number of blocks is used during query planning and execution.

Since this is a potentially expensive operation, the storage manager maintains a shared-memory cache of relation sizes recently computed by device managers.

1.2.13. smgr_commit

When a transaction finishes, the data manager calls the `smgr_commit` routine for every device manager. This routine must flush any pending asynchronous writes for this transaction to disk, so that all relation data is on persistent storage before the transaction is marked committed. In addition, the device manager can release resources or update its state appropriately.

This routine may be NULL, if there is no work to do for a device manager at transaction commit. If it is supplied, its interface is

```
int
smgr_commit()
```

There is one important issue regarding transaction commit that should also be mentioned. Much research has been done in database systems to support *distribution*, or databases that span multiple computers in a network. Although POSTGRES is not a distributed database system, it does support *data distribution* by means of the storage manager switch and device managers.

Data distribution means that different tables, or parts of tables, may reside on storage devices connected to different computers on a network. POSTGRES does all data manipulation at a single site, but the data it works with may be distributed. The data manager will call the device manager, which can operate on a local device, or use the network to satisfy requests on a remote device. In fact, the Sony jukebox device manager works correctly whether it runs on the host computer to which the jukebox is connected, or on some other computer on the Internet.

At transaction commit time, the data manager informs each device manager that all its data must be on stable storage. Each device manager flushes any pending writes synchronously, and returns. Once all device managers have forced data to stable storage, the data manager synchronously records that the transaction has committed by writing a ten-byte record to the *transaction status file* on one of the storage devices. If the system crashes before the commit record is stored in the transaction status file, POSTGRES will consider the transaction to be aborted when the transaction's changes are encountered later.

Thus the `smgr_commit` interface supports data distribution without using a traditional multi-phase commit protocol. In particular, device managers are not notified whether a transaction actually commits, once they have forced data to stable storage.

1.2.14. smgr_abort

If a transaction aborts, the data manager calls `smgr_abort` in place of `smgr_commit`. In this case, it does not matter if changes made to user relations during the aborted transaction are reflected on stable storage. Device managers may deschedule pending writes, as long as the writes include no data from other transactions.

This routine may be NULL, if there is no work to do for a device manager at transaction abort.

1.3. Architecture Summary

The POSTGRES storage manager architecture assigns a particular device manager to manage each device, or class of devices, available to the database system. A well-defined set of interface routines makes relation accesses device-transparent.

The architecture uses a *storage manager switch* to record the interface routines for all the device managers known to the system.

So far, three device managers exist: magnetic disk, a Sony WORM optical disk jukebox, and volatile main memory.