CHAPTER 13

Enabling Persistence Using a Real-World Application

This chapter turns the theory from Chapter 4 (and other chapters) into practice. We show how an application can take advantage of persistent memory by building a persistent memory-aware database storage engine. We use MariaDB* (https://mariadb.org/), a popular open source database, to reflect the real-life struggles of developers.

We assume that you are familiar with the preceding chapters that covered the fundamentals of the persistent memory programming model and Persistent Memory Development Kit (PMDK). In this chapter, we implement our storage engine using C++ and `libpmemobj-cpp` from Chapter 8. If you are not a C++ developer you will find this information helpful because the fundamentals apply to other languages and applications.

The complete source code for the persistent memory-aware database storage engine we develop here can be found on GitHub at https://github.com/pmem/pmdk-examples/tree/master/pmem-mariadb.

The Database Example

A tremendous number of existing applications can be categorized in many ways. For the purpose of this chapter, we explore applications from the common components perspective, including an interface, a business layer, and a store. The interface interacts with the user, the business layer is a tier where the application's logic is implemented, and the store is where data is kept and processed by the application's processes.

With so many applications available today, choosing one to include in this book that would satisfy all or most of our requirements was difficult. We chose to use a database as an example because a unified way of accessing data is a common denominator for many applications.

Different Persistent Memory Enablement Approaches

The main advantages of persistent memory include:

- It provides access latencies that are lower than flash SSDs.
- Increased throughput when compared to flash storage.
- Less expensive than DRAM.
- It is cacheable. This is a huge advantage over PCIe interconnect, which cannot be cached in the CPU.
- Real-time access to data allows ultrafast access to large datasets.
- Data persists in memory after a power interruption, like flash.

Persistent memory can be used in a variety of ways to deliver lower latency for many applications:

- **In-Memory databases.** In-memory databases can leverage persistent memory's larger capacities and significantly reduce restart times. Once the database memory-maps the files, the data is immediately accessible.
- **Fraud detection.** Financial institutions and insurance companies can more quickly perform data analytics on millions of records to detect fraudulent transactions, preventing financial losses and impact on the brand name.
- **Cyber threat analysis.** Companies can quickly detect and defend against increasing cyber threats.

- **Web-scale personalization.** Companies can tailor online user experiences by returning relevant content and advertisements, resulting in higher user click-through and more e-commerce revenue opportunities.
- **Financial trading.** Financial trading applications can rapidly process and execute financial transactions, allowing them to gain a competitive advantage and create a higher revenue opportunity.
- **Internet of Things (IoT).** Faster processing of huge datasets in real-time reduces time to value.
- **Content delivery networks (CDN).** A CDN is a highly-distributed network of edge servers strategically placed across the globe with the purpose of rapidly delivering digital content to users. With persistent memory capacity, each CDN node can cache more data and reduce the total number of servers, while networks can reliably deliver low latency data to their clients.

Developing a Persistent Memory-Aware MariaDB* Storage Engine

The storage engine developed here is not production quality and does not implement all the functionality expected by most database administrators. To demonstrate the concepts described earlier, we kept the example simple, implementing table `create()`, `open()`, and `close()` operations and `INSERT`, `UPDATE`, `DELETE`, and `SELECT` SQL operations. Because the storage engine capabilities are quite limited without indexing, we include a simple indexing system using volatile memory to provide faster access to the data residing in persistent memory.

Although MariaDB has many storage engines to which we could add persistent memory, we are building a new storage engine from scratch in this chapter. To learn more about the MariaDB storage engine API and how storage engines work, we suggest reading the MariaDB "Storage Engine Development" documentation (https://mariadb.com/kb/en/library/storage-engines-storage-engine-development/). Since MariaDB is based on MySQL*, you can also refer to the MySQL "Writing a Custom Storage Engine" documentation (https://dev.mysql.com/doc/internals/en/custom-engine.html) to find all the information for creating an engine from scratch.

Understanding the Storage Layer

MariaDB provides a pluggable architecture for storage engines that makes it easier to develop and deploy new storage engines for MySQL. A pluggable storage engine architecture also makes it possible to create new storage engines and add them to a running MySQL server without recompiling the server itself. The storage engines manage data storage and index management for MariaDB. The MariaDB server communicates with the storage engines through a well-defined API.

In our code example, we implement a prototype of a pluggable persistent memory-enabled storage engine for MariaDB using the `libpmemobj` library from the Persistent Memory Development Kit (PMDK).

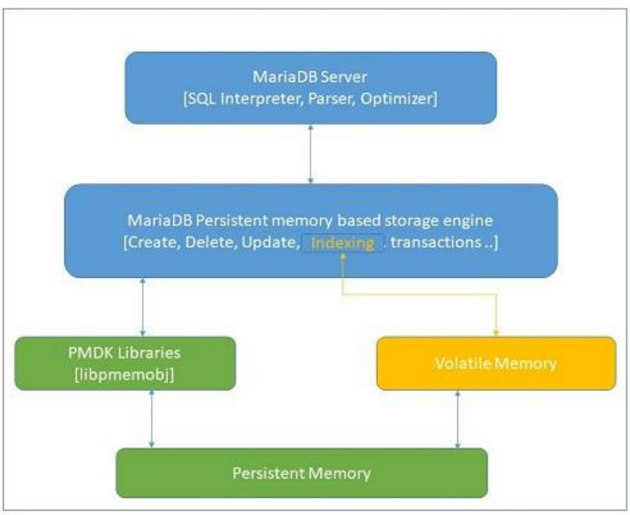

Figure 13-1. MariaDB storage engine architecture diagram for persistent memory.*

Figure 13-1 shows how the storage engine communicates with `libpmemobj` to manage the data stored in persistent memory. The library is used to turn a persistent memory file into a flexible object store.

Creating a Storage Engine Class

The implementation of the storage engine described here is a single-threaded application that supports a single session, a single user, and single table requests. The MariaDB server communicates with storage engines through a well-defined handler interface that includes a `handlerton`, which is a singleton `handler` that is connected to a table handler. The `handlerton` defines the storage engine and contains pointers to the methods that apply to the persistent memory storage engine.

The first method the storage engine needs to support is to enable the call for a new handler instance, shown in Listing 13-1.

Listing 13-1. ha_pmdk.cc - Creating a new handler instance.

```
static handler *pmdk_create_handler(handlerton *hton,
                                     TABLE_SHARE *table,
                                     MEM_ROOT *mem_root);

handlerton *pmdk_hton;
```

When a `handler` instance is created, the MariaDB server sends commands to the `handler` to perform data storage and retrieve tasks such as opening a table, manipulating rows, managing indexes, and transactions. When a `handler` is instantiated, the first required operation is the opening of a table. Since the storage engine is a single user and single-threaded application, only one `handler` instance is created in this implementation.

Various `handler` methods are also implemented; they apply to the storage engine as a whole, as opposed to methods like `create()` and `open()` that work on a per-table basis. Some examples of such methods include transaction methods to handle commits and rollbacks, shown in Listing 13-2.

Listing 13-2. ha_pmdk.cc - Handler methods including transactions, rollback, etc.

```
static int pmdk_init_func(void *p)
{
...
  pmdk_hton= (handlerton *)p;
  pmdk_hton->state=   SHOW_OPTION_YES;
```

```
  pmdk_hton->create=  pmdk_create_handler;
  pmdk_hton->flags=   HTON_CAN_RECREATE;
  pmdk_hton->tablefile_extensions= ha_pmdk_exts;

  pmdk_hton->commit= pmdk_commit;
  pmdk_hton->rollback= pmdk_rollback;
...
}
```

The abstract methods defined in the handler class are implemented to work with persistent memory. An internal representation of the objects in persistent memory is created using a single linked list (SLL). This internal representation is very helpful to iterate through the records to improve performance.

To perform a variety of operations and gain faster and easier access to data, we used the simple `row` structure shown in Listing 13-3 to hold the pointer to persistent memory and the associated field value in the buffer.

Listing 13-3. ha_pmdk.h - A simple data structure to store data in a single linked list.

```
struct row {
  persistent_ptr<row> next;
  uchar buf[];
};
```

Creating a Database Table

The `create()` abstract method is used to create the table. This method creates all necessary files in persistent memory using `libpmemobj` APIs. As shown in Listing 13-4, we create a new `pmemobj` type pool for each table using the `pmemobj_create()` method; this method creates a transactional object store with the given total `poolsize`. The table is created in the form of an .obj extension.

Listing 13-4. Creating a table method.

```
int ha_pmdk::create(const char *name, TABLE *table_arg,
                       HA_CREATE_INFO *create_info)
{

  char path[MAX_PATH_LEN];
```

```
  DBUG_ENTER("ha_pmdk::create");
  DBUG_PRINT("info", ("create"));

  snprintf(path, MAX_PATH_LEN, "%s%s", name,
PMEMOBJ_EXT);
  PMEMobjpool *pop = pmemobj_create(path,
name,PMEMOBJ_MIN_POOL, S_IRWXU);
  if (pop == NULL) {
    DBUG_PRINT("info", ("failed : %s error number :
%d",path,errCodeMap[errno]));
    DBUG_RETURN(errCodeMap[errno]);
  }
  DBUG_PRINT("info", ("Success"));
  pmemobj_close(pop);

  DBUG_RETURN(0);
}
```

Opening a Database Table

Before any read or write operations are performed on a table, the MariaDB server calls the `open()` method to open the table data and index table. This method opens all the named tables associated with the persistent memory storage engine at the time the storage engine starts. A new table class variable, `objtab`, was added to hold the `PMEMobjpool`. The names for the tables to be opened are provided by the MariaDB server. The index container in volatile memory is populated using the `open()` function call at the time of server start using the `loadIndexTableFromPersistentMemory()` function.

The `pmemobj_open()` function from `libpmemobj` is used to open an existing object store memory pool (see Listing 13-5). The table is also opened at the time of a table creation if any read/ write action is triggered.

Listing 13-5. ha_pmdk.cc - Opening a database table.

```
int ha_pmdk::open(const char *name, int mode, uint
test_if_locked)
{
...
  objtab = pmemobj_open(path, name);
```

```
  if (objtab == NULL)
    DBUG_RETURN(errCodeMap[errno]);

  proot = pmemobj_root(objtab, sizeof (root));
  // update the MAP when start occured
  loadIndexTableFromPersistentMemory();
...
}
```

Once the storage engine is up and running, we can begin to insert data into it. But we first must implement the `INSERT`, `UPDATE`, `DELETE`, and `SELECT` operations.

Closing a Database Table

When the server is finished working with a table, it calls the `closeTable()` method to close the file using `pmemobj_close()` and release any other resources (see Listing 13-6). The `pmemobj_close()` function closes the memory pool indicated by `objtab` and deletes the memory pool handle.

Listing 13-6. ha_pmdk.cc - Closing a database table.

```
int ha_pmdk::close(void)
{
  DBUG_ENTER("ha_pmdk::close");
  DBUG_PRINT("info", ("close"));

  pmemobj_close(objtab);
  objtab = NULL;

  DBUG_RETURN(0);
}
```

INSERT Operation

The `INSERT` operation is implemented in the `write_row()` method, shown in Listing 13-7. During an `INSERT`, the row objects are maintained in a singly linked list. If the table is indexed, the index table container in volatile memory is updated with the new row objects after the persistent operation completes successfully. `write_row()` is an important method because, in addition to the allocation of persistent pool storage to the rows, it is used to populate the indexing containers. `pmemobj_tx_alloc()` is used for inserts. `write_row()` transactionally allocates a new object of a given size and `type_num`.

Listing 13-7. ha_pmdk.cc - Closing a database table.

```
int ha_pmdk::write_row(uchar *buf)
{
...
  int err = 0;

  if (isPrimaryKey() == true)
    DBUG_RETURN(HA_ERR_FOUND_DUPP_KEY);

  persistent_ptr<row> row;
  TX_BEGIN(objtab) {
        row = pmemobj_tx_alloc(sizeof (row) + table->s->reclength, 0);
    memcpy(row->buf, buf, table->s->reclength);
    row->next = proot->rows;
    proot->rows = row;
  } TX_ONABORT {
         DBUG_PRINT("info", ("write_row_abort errno :%d ",errno));
    err = errno;
  } TX_END
  stats.records++;

  for (Field **field = table->field; *field; field++) {
    if ((*field)->key_start.to_ulonglong() >= 1) {
              std::string convertedKey = IdentifyTypeAndConvertToString((*field)->ptr, (*field)->type(),(*field)->key_length(),1);
      insertRowIntoIndexTable(*field, convertedKey, row);
    }
  }
  DBUG_RETURN(err);
}
```

In every `INSERT` operation, the field values are checked for a preexisting duplicate. The primary key field in the table is checked using the `isPrimaryKey()` function (Line 423). If the key is a duplicate, the error `HA_ERR_FOUND_DUPP_KEY` is returned. The `isPrimaryKey()` is implemented in Listing 13-8.

Listing 13-8. ha_pmdk.cc – Checking for duplicate primary keys.

```
bool ha_pmdk::isPrimaryKey(void)
{
  bool ret = false;
  database *db = database::getInstance();
  table_ *tab;
  key *k;
  for (unsigned int i= 0; i < table->s->keys; i++) {
    KEY* key_info = &table->key_info[i];
    if (memcmp("PRIMARY",key_info->name.str,sizeof("PRIMARY"))==0) {
      Field *field = key_info->key_part->field;
      std::string convertedKey = IdentifyTypeAndConvertToString(field->ptr, field->type(),field->key_length(),1);
      if (db->getTable(table->s->table_name.str, &tab)) {
        if (tab->getKeys(field->field_name.str, &k)) {
          if (k->verifyKey(convertedKey)) {
            ret = true;
            break;
          }
        }
      }
    }
  }
  return ret;
}
```

UPDATE Operation

The server executes `UPDATE` statements by performing a `rnd_init()` or `index_init()` table scan until it locates a row matching the key value in the `WHERE` clause of the `UPDATE` statement before calling the `update_row()` method. If the table is an indexed table, the index container is also updated after this operation is successful. In the `update_row()` method defined in Listing 13-9, the `old_data` field will have the previous row record in it, while `new_data` will have the newest data in it.

Listing 13-9. ha_pmdk.cc – Updating existing row data.

```
int ha_pmdk::update_row(const uchar *old_data, const uchar *new_data)
{
...
            if (k->verifyKey(key_str))
              k->updateRow(key_str, field_str);
...
  if (current)
    memcpy(current->buf, new_data, table->s->reclength);
...
```

The index table is also updated using the `updateRow()` method shown in Listing 13-10.

Listing 13-10. ha_pmdk.cc – Updating existing row data.

```
bool key::updateRow(const std::string oldStr, const std::string newStr)
{
...
  persistent_ptr<row> row_;
  bool ret = false;
  rowItr matchingEleIt = getCurrent();

  if (matchingEleIt->first == oldStr) {
    row_ = matchingEleIt->second;
    std::pair<const std::string, persistent_ptr<row> > r(newStr, row_);
    rows.erase(matchingEleIt);
    rows.insert(r);
    ret = true;
  }
  DBUG_RETURN(ret);
}
```

DELETE Operation

The `DELETE` operation is implemented using the `delete_row()` method. Three different scenarios should be considered:

- Deleting an indexed value from the indexed table
- Deleting a non-indexed value from the indexed table
- Deleting a field from the non-indexed table

For each scenario, different functions are called. When the operation is successful, the entry is removed from both the index (if the table is an indexed table) and persistent memory. Listing 13-11 shows the logic to implement the three scenarios.

Listing 13-11. ha_pmdk.cc – Updating existing row data.

```
int ha_pmdk::delete_row(const uchar *buf)
{
...
  // Delete the field from non indexed table
  if (active_index == 64 && table->s->keys ==0 ) {
    if (current)
      deleteNodeFromSLL();
  } else if (active_index == 64 && table->s->keys !=0 ) { // Delete non indexed column field from indexed table
    if (current) {
      deleteRowFromAllIndexedColumns(current);
      deleteNodeFromSLL();
    }
  } else { // Delete indexed column field from indexed table
  database *db = database::getInstance();
  table_ *tab;
  key *k;
  KEY_PART_INFO *key_part = table->key_info[active_index].key_part;
  if (db->getTable(table->s->table_name.str, &tab)) {
      if (tab->getKeys(key_part->field->field_name.str, &k)) {
```

```
        rowItr currNode = k->getCurrent();
        rowItr prevNode = std::prev(currNode);
        if (searchNode(prevNode->second)) {
          if (prevNode->second) {
            deleteRowFromAllIndexedColumns(prevNode->second);
            deleteNodeFromSLL();
          }
        }
      }
    }
  }
  stats.records--;

  DBUG_RETURN(0);
}
```

Listing 13-12 shows how the `deleteRowFromAllIndexedColumns()` function deletes the value from the index containers using the `deleteRow()` method.

Listing 13-12. ha_pmdk.cc – Deletes an entry from the index containers.

```
void ha_pmdk::deleteRowFromAllIndexedColumns(const persistent_ptr<row> &row)
{
...
    if (db->getTable(table->s->table_name.str, &tab)) {
      if (tab->getKeys(field->field_name.str, &k)) {
        k->deleteRow(row);
      }
...
```

Listing `deleteNodeFromSLL()` deletes the object from the linked list residing on persistent memory using `libpmemobj` transactions, as shown in Listing 13-13.

Listing 13-13. ha_pmdk.cc – Deletes an entry from the linked list using transactions.

```
int ha_pmdk::deleteNodeFromSLL()
{
  if (!prev) {
    if (!current->next) { // When sll contains single node
```

```
      TX_BEGIN(objtab) {
        delete_persistent<row>(current);
        proot->rows = nullptr;
      } TX_END
    } else { // When deleting the first node of sll
      TX_BEGIN(objtab) {
        delete_persistent<row>(current);
        proot->rows = current->next;
        current = nullptr;
      } TX_END
    }
  } else {
    if (!current->next) { // When deleting the last node of sll
      prev->next = nullptr;
    } else { // When deleting other nodes of sll
      prev->next = current->next;
    }
    TX_BEGIN(objtab) {
      delete_persistent<row>(current);
      current = nullptr;
    } TX_END
  }
  return 0;
}
```

SELECT Operation

`SELECT` is an important operation that is required by several methods. Many methods that are implemented for the `SELECT` operation are also called from other methods. The `rnd_init()` method is used to prepare for a table scan for non-indexed tables, resetting counters and pointers to the start of the table. If the table is an indexed table, the MariaDB Server calls the `index_init()` method. As shown in Listing 13-14, the pointers are initialized.

Listing 13-14. ha_pmdk.cc – rnd_init() is called when the system wants the storage engine to do a table scan.

```
int ha_pmdk::rnd_init(bool scan)
{
...
```

```
  current=prev=NULL;
  iter = proot->rows;
  DBUG_RETURN(0);
}
```

When the table is initialized, the MariaDB server calls the `rnd_next()`, `index_first()` or `index_read_map()` method, depending on whether the table is a non-indexed or indexed. These methods populate the buffer with data from the current object and point the iterator to the next value. The methods are called once for every row to be scanned.

Listing 13-15 shows how the buffer passed to the function is populated with the contents of the table row in the internal MariaDB format. If there are no more objects to read, the return value must be HA_ERR_END_OF_FILE.

Listing 13-15. ha_pmdk.cc – rnd_init() is called when the system wants the storage engine to do a table scan.

```
int ha_pmdk::rnd_next(uchar *buf)
{
...
  memcpy(buf, iter->buf, table->s->reclength);
  if (current != NULL) {
    prev = current;
  }
  current = iter;
  iter = iter->next;

  DBUG_RETURN(0);
}
```

Summary

This chapter provides a walkthrough using `libpmemobj` from the PMDK to create a persistent memory-aware storage engine for the popular open source MariaDB database. Using persistent memory in an application can provide continuity in the event of an unplanned system shutdown along with improved performance gained by storing your data close to the CPU where you can access it at the speed of the memory bus. While database engines commonly use in-memory caches for performance, which take time to warm up, persistent memory offers an immediately warm cache upon application startup.