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, as it provides a pluggable storage engine model. The completed storage engine is not intended for production use and does not implement all the features a production quality storage engine should. We implement only the basic functionality to demonstrate how to begin persistent memory programming using a well known database. The intent is to provide you with a more hands-on approach for persistent memory programming so you may enable persistent memory features and functionality within your own application. Our storage engine is left as an optional exercise for you to complete. Doing so would create a new persistent memory storage engine for MariaDB, MySQL, Percona Server, and other derivatives. You may also choose to modify an existing MySQL database storage engine to add persistent memory features, or perhaps choose a different database entirely.

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 still 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 can be found on GitHub at https://github.com/pmem/pmdk-examples/tree/master/pmem-mariadb.

S. Scargall, *Programming Persistent Memory*, https://doi.org/10.1007/978-1-4842-4932-1_13

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.

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.
- It has higher throughput than NAND storage devices.
- Real-time access to data allows ultrafast access to large datasets.
- Data persists in memory after a power interruption.

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 index, tables, and other files, the data is immediately accessible. This avoids lengthy startup times where the data is traditionally read from disk and paged in to memory before it can be accessed or processed.
- **Fraud detection:** Financial institutions and insurance companies can perform real-time data analytics on millions of records to detect fraudulent transactions.
- **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 rate 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 data ingest and 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 a 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. If the CDN cache is persisted, a node can restart with a warm cache and sync only the data it is missed while it was out of the cluster.

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

In our code, 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).

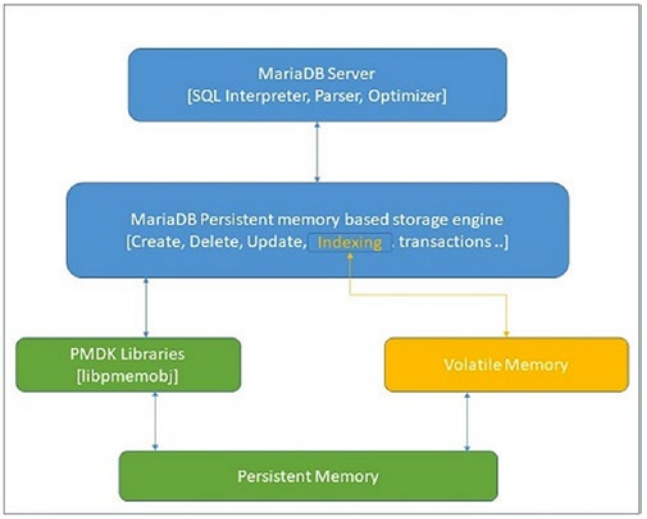

Figure 13-1. *MariaDB storage engine architecture diagram for persistent memory*

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 pool into a flexible object store.

Creating a Storage Engine Class

The implementation of the storage engine described here is single-threaded to support a single session, a single user, and single table requests. A multi-threaded implementation would detract from the focus of this chapter. Chapter 14 discussed concurrency in more detail. 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

```
117  static handler *pmdk_create_handler(handlerton *hton,
118                                      TABLE_SHARE *table,
119                                      MEM_ROOT *mem_root);
120
121  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 implementation, only one `handler` instance is created.

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

```
209  static int pmdk_init_func(void *p)
210  {
...
213    pmdk_hton= (handlerton *)p;
214    pmdk_hton->state=   SHOW_OPTION_YES;
215    pmdk_hton->create=  pmdk_create_handler;
```

```
216    pmdk_hton->flags=   HTON_CAN_RECREATE;
217    pmdk_hton->tablefile_extensions= ha_pmdk_exts;
218
219    pmdk_hton->commit= pmdk_commit;
220    pmdk_hton->rollback= pmdk_rollback;
...
223  }
```

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

```
71  struct row {
72    persistent_ptr<row> next;
73    uchar buf[];
74  };
```

Creating a Database Table

The `create()` method is used to create the table. This method creates all necessary files in persistent memory using `libpmemobj`. 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

```
1247  int ha_pmdk::create(const char *name, TABLE *table_arg,
1248                      HA_CREATE_INFO *create_info)
1249  {
1250
```

```
1251    char path[MAX_PATH_LEN];
1252    DBUG_ENTER("ha_pmdk::create");
1253    DBUG_PRINT("info", ("create"));
1254
1255    snprintf(path, MAX_PATH_LEN, "%s%s", name, PMEMOBJ_EXT);
1256    PMEMobjpool *pop = pmemobj_create(path, name,PMEMOBJ_MIN_POOL,
        S_IRWXU);
1257    if (pop == NULL) {
1258      DBUG_PRINT("info", ("failed : %s error number :
          %d",path,errCodeMap[errno]));
1259      DBUG_RETURN(errCodeMap[errno]);
1260    }
1261    DBUG_PRINT("info", ("Success"));
1262    pmemobj_close(pop);
1263
1264    DBUG_RETURN(0);
1265  }
```

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 data and index tables. 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

```
290  int ha_pmdk::open(const char *name, int mode, uint test_if_locked)
291  {
...
```

```
302    objtab = pmemobj_open(path, name);
303    if (objtab == NULL)
304      DBUG_RETURN(errCodeMap[errno]);
305
306    proot = pmemobj_root(objtab, sizeof (root));
307    // update the MAP when start occured
308    loadIndexTableFromPersistentMemory();
...
310  }
```

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

```
376  int ha_pmdk::close(void)
377  {
378    DBUG_ENTER("ha_pmdk::close");
379    DBUG_PRINT("info", ("close"));
380
381    pmemobj_close(objtab);
382    objtab = NULL;
383
384    DBUG_RETURN(0);
385  }
```

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

```
417  int ha_pmdk::write_row(uchar *buf)
418  {
...
421    int err = 0;
422
423    if (isPrimaryKey() == true)
424      DBUG_RETURN(HA_ERR_FOUND_DUPP_KEY);
425
426    persistent_ptr<row> row;
427    TX_BEGIN(objtab) {
428      row = pmemobj_tx_alloc(sizeof (row) + table->s->reclength, 0);
429      memcpy(row->buf, buf, table->s->reclength);
430      row->next = proot->rows;
431      proot->rows = row;
432    } TX_ONABORT {
433      DBUG_PRINT("info", ("write_row_abort errno :%d ",errno));
434      err = errno;
435    } TX_END
436    stats.records++;
437
438    for (Field **field = table->field; *field; field++) {
439      if ((*field)->key_start.to_ulonglong() >= 1) {
440        std::string convertedKey = IdentifyTypeAndConvertToString((*fie
ld)->ptr, (*field)->type(),(*field)->key_length(),1);
441        insertRowIntoIndexTable(*field, convertedKey, row);
442      }
443    }
444    DBUG_RETURN(err);
445  }
```

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

```
462  bool ha_pmdk::isPrimaryKey(void)
463  {
464    bool ret = false;
465    database *db = database::getInstance();
466    table_ *tab;
467    key *k;
468    for (unsigned int i= 0; i < table->s->keys; i++) {
469      KEY* key_info = &table->key_info[i];
470      if (memcmp("PRIMARY",key_info->name.str,sizeof("PRIMARY"))==0) {
471        Field *field = key_info->key_part->field;
472        std::string convertedKey = IdentifyTypeAndConvertToString
           (field->ptr, field->type(),field->key_length(),1);
473        if (db->getTable(table->s->table_name.str, &tab)) {
474          if (tab->getKeys(field->field_name.str, &k)) {
475            if (k->verifyKey(convertedKey)) {
476              ret = true;
477              break;
478            }
479          }
480        }
481      }
482    }
483    return ret;
484  }
```

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 new data.

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

```
506  int ha_pmdk::update_row(const uchar *old_data, const uchar *new_data)
507  {
...
540              if (k->verifyKey(key_str))
541                k->updateRow(key_str, field_str);
...
551    if (current)
552      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

```
1363  bool key::updateRow(const std::string oldStr, const std::string newStr)
1364  {
...
1366     persistent_ptr<row> row_;
1367     bool ret = false;
1368     rowItr matchingEleIt = getCurrent();
1369
1370     if (matchingEleIt->first == oldStr) {
1371       row_ = matchingEleIt->second;
1372       std::pair<const std::string, persistent_ptr<row> > r(newStr, row_);
1373       rows.erase(matchingEleIt);
1374       rows.insert(r);
1375       ret = true;
1376     }
1377     DBUG_RETURN(ret);
1378  }
```

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);
```

```
620           if (searchNode(prevNode->second)) {
621             if (prevNode->second) {
622               deleteRowFromAllIndexedColumns(prevNode->second);
623               deleteNodeFromSLL();
624             }
625           }
626         }
627       }
628     }
629     stats.records--;
630
631     DBUG_RETURN(0);
632   }
```

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

```
634   void ha_pmdk::deleteRowFromAllIndexedColumns(const persistent_ptr<row>
      &row)
635   {
...
643       if (db->getTable(table->s->table_name.str, &tab)) {
644         if (tab->getKeys(field->field_name.str, &k)) {
645           k->deleteRow(row);
646         }
...
```

The `deleteNodeFromSLL()` method 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

```
651 int ha_pmdk::deleteNodeFromSLL()
652 {
653   if (!prev) {
654     if (!current->next) { // When sll contains single node
655       TX_BEGIN(objtab) {
656         delete_persistent<row>(current);
657         proot->rows = nullptr;
658       } TX_END
659     } else { // When deleting the first node of sll
660       TX_BEGIN(objtab) {
661         delete_persistent<row>(current);
662         proot->rows = current->next;
663         current = nullptr;
664       } TX_END
665     }
666   } else {
667     if (!current->next) { // When deleting the last node of sll
668       prev->next = nullptr;
669     } else { // When deleting other nodes of sll
670       prev->next = current->next;
671     }
672     TX_BEGIN(objtab) {
673       delete_persistent<row>(current);
674       current = nullptr;
675     } TX_END
676   }
677   return 0;
678 }
```

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

```
869  int ha_pmdk::rnd_init(bool scan)
870  {
...
874    current=prev=NULL;
875    iter = proot->rows;
876    DBUG_RETURN(0);
877  }
```

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 indexed or not. These methods populate the buffer with data from the current object and updates 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

```
902  int ha_pmdk::rnd_next(uchar *buf)
903  {
...
910    memcpy(buf, iter->buf, table->s->reclength);
911    if (current != NULL) {
912      prev = current;
913    }
```

```
914     current = iter;
915     iter = iter->next;
916
917     DBUG_RETURN(0);
918   }
```

This concludes the basic functionality our persistent memory enabled storage engine set out to achieve. We encourage you to continue the development of this storage engine to introduce more features and functionality.

Summary

This chapter provided a walk-through 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.