MySQL 深潜 - 一文详解 MySQL Data Dictionary

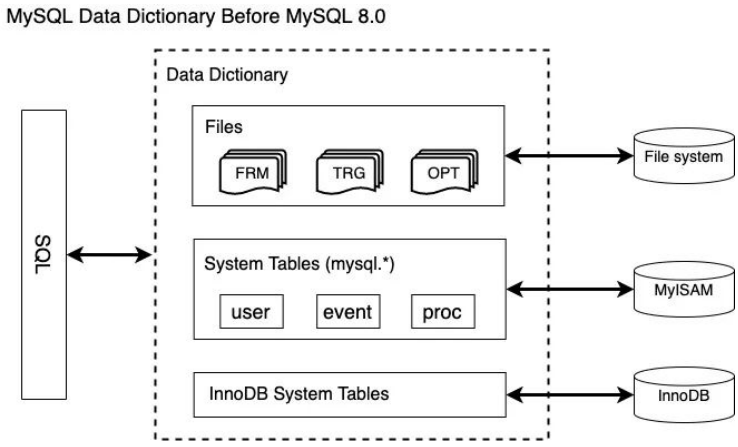
Original 泊歌 阿里开发者 2021-08-23 16:58

收录于合集

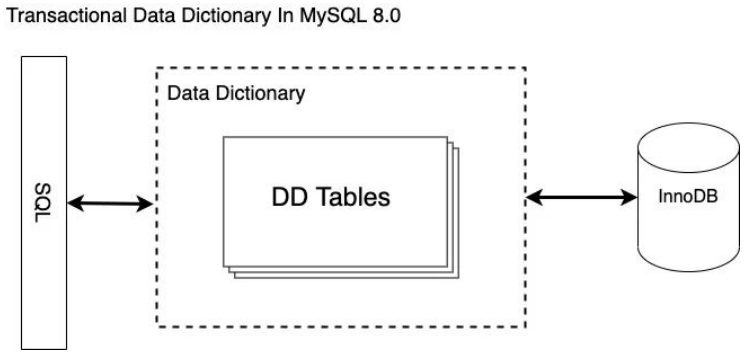
#C++ 2 #mysql 5 #PolarDB 2

一 背景

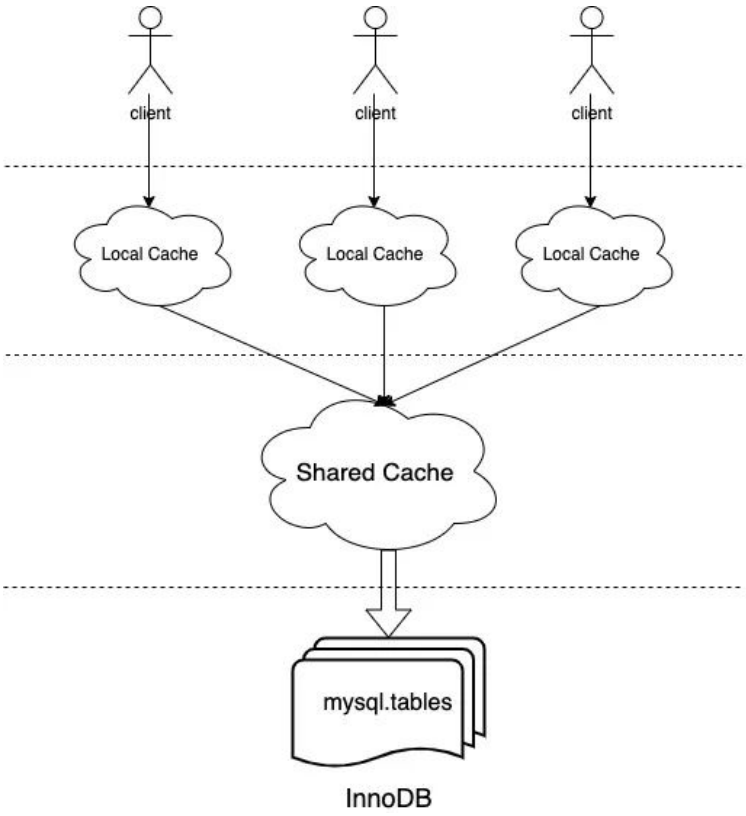
在 MySQL 8.0 之前，Server 层和存储引擎（比如 InnoDB）会各自保留一份元数据（schema name, table definition 等），不仅在信息存储上有着重复冗余，而且可能存在两者之间存储的元数据不同步的现象。不同存储引擎之间（比如 InnoDB 和 MyISAM）有着不同的元数据存储形式和位置(.FRM, .PAR, .OPT, .TRN and .TRG files)，造成了元数据无法统一管理。此外，将元数据存放在不支持事务的表和文件中，使得 DDL 变更不会是原子的，crash recovery 也会成为一个问题。



为了解决上述问题，MySQL 在 8.0 中引入了 data dictionary 来进行 Server 层和不同引擎间统一的元数据管理，这些元数据都存储在 InnoDB 引擎的表中，自然的支持原子性，且 Server 层和引擎层共享一份元数据，不再存在不同步的问题。



二 整体架构



data dictionary 提供了统一的 client API 供 Server 层和引擎层使用，包含对元数据访问的 acquire() / drop() / store() / update() 基本操作。底层实现了对 InnoDB 引擎存放的数据字典表的读写操作，包含开表(open table)、构造主键、主键查找等过程。client 和底层存储之间通过两级缓存来加速对元数据对象的内存访问，两级缓存都是基于 hash map 实现的，一层缓存是 local 的，由每个 client（每个线程对应一个 client）独享；二级缓存是 share 的，为所有线程共享的全局缓存。下面我将对 data

dictionary 的数据结构和实现架构做重点介绍，也会分享一个支持原子的 DDL 在 data dictionary 层面的实现过程。

三 metadata 在内存和引擎层面的表示

data dictionary (简称DD)中的数据结构是完全按照多态、接口/实现的形式来组织的，接口通过纯虚类来实现（比如表示一个表的 Table），其实现类（Table_impl）为接口类的名字加 _impl 后缀。下面以 Table_impl 为例介绍一个表的元数据对象在 DD cache 中的表示。

1 Table_impl

Table_impl 类中包含一个表相关的元数据属性定义，比如下列最基本引擎类型、comment、分区类型、分区表达式等。

```
class Table_impl : public Abstract_table_impl, virtual public Table {
  // Fields.

  Object_id m_se_private_id;

  String_type m_engine;
  String_type m_comment;

  // - Partitioning related fields.

  enum_partition_type m_partition_type;
  String_type m_partition_expression;
  String_type m_partition_expression_utf8;
  enum_default_partitioning m_default_partitioning;

  // References to tightly-coupled objects.

  Index_collection m_indexes;
  Foreign_key_collection m_foreign_keys;
  Foreign_key_parent_collection m_foreign_key_parents;
  Partition_collection m_partitions;
```

```
  Partition_leaf_vector m_leaf_partitions;
  Trigger_collection m_triggers;
  Check_constraint_collection m_check_constraints;
};
```

Table_impl 也是代码实现中 client 最常访问的内存结构，开发者想要增加新的属性，直接在这个类中添加和初始化即可，但是仅仅如此不会自动将该属性持久化到存储引擎中。除了上述简单属性之外，还包括与一个表相关的复杂属性，比如列信息、索引信息、分区信息等，这些复杂属性都是存在其他的 DD 表中，在内存 cache 中也都会集成到 Table_impl 对象里。

从Abstract_table_impl继承来的 Collection m_columns 就表示表的所有列集合，集合中的每一个对象 Column_impl 表示该列的元信息，包括数值类型、是否为 NULL、是否自增、默认值等。同时也包含指向 Abstract_table_impl 的指针，将该列与其对应的表联系起来。

```
class Column_impl : public Entity_object_impl, public Column {
  // Fields.

  enum_column_types m_type;

  bool m_is_nullable;
  bool m_is_zerofill;
  bool m_is_unsigned;
  bool m_is_auto_increment;
  bool m_is_virtual;

  bool m_default_value_null;
  String_type m_default_value;

  // References to tightly-coupled objects.

  Abstract_table_impl *m_table;
};
```

此外 Table_impl 中也包含所有分区的元信息集合 Collection m_partitions，存放每个分区的 id、引擎、选项、范围值、父子分区等。

```
class Partition_impl : public Entity_object_impl, public Partition {
  // Fields.

  Object_id m_parent_partition_id;
  uint m_number;
  Object_id m_se_private_id;

  String_type m_description_utf8;
  String_type m_engine;
  String_type m_comment;
  Properties_impl m_options;
  Properties_impl m_se_private_data;

  // References to tightly-coupled objects.

  Table_impl *m_table;

  const Partition *m_parent;

  Partition_values m_values;
  Partition_indexes m_indexes;
  Table::Partition_collection m_sub_partitions;
};
```

因此获取到一个表的 Table_impl，我们就可以获取到与这个表相关联的所有元信息。

2 Table_impl 是如何持久化存储和访问的

DD cache 中的元信息都是在 DD tables 中读取和存储的，每个表存放一类元信息的基本属性字段，比如 tables、columns、indexes等，他们之间通过主外键关联连接起来，组成 Table_impl 的全部元信息。DD tables 存放在 mysql 的表空间中，在 release 版本对用户隐藏，只能通过 INFORMATION

SCHEMA 的部分视图查看；在 debug 版本可通过设置 SET debug='+d,skip_dd_table_access_check' 直接访问查看。比如：

```
root@localhost:test 8.0.18-debug> SHOW CREATE TABLE mysql.tables\G
***************************<strong> 1. row </strong>***********************
       Table: tables
Create Table: CREATE TABLE `tables` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `schema_id` bigint(20) unsigned NOT NULL,
  `name` varchar(64) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
  `type` enum('BASE TABLE','VIEW','SYSTEM VIEW') COLLATE utf8_bin NOT NUI
  `engine` varchar(64) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NUI
  `mysql_version_id` int(10) unsigned NOT NULL,
  `row_format` enum('Fixed','Dynamic','Compressed','Redundant','Compact',
  `collation_id` bigint(20) unsigned DEFAULT NULL,
  `comment` varchar(2048) COLLATE utf8_bin NOT NULL,
  `hidden` enum('Visible','System','SE','DDL') COLLATE utf8_bin NOT NULL,
  `options` mediumtext COLLATE utf8_bin,
  `se_private_data` mediumtext COLLATE utf8_bin,
  `se_private_id` bigint(20) unsigned DEFAULT NULL,
  `tablespace_id` bigint(20) unsigned DEFAULT NULL,
  `partition_type` enum('HASH','KEY_51','KEY_55','LINEAR_HASH','LINEAR_KE
  `partition_expression` varchar(2048) COLLATE utf8_bin DEFAULT NULL,
  `partition_expression_utf8` varchar(2048) COLLATE utf8_bin DEFAULT NULI
  `default_partitioning` enum('NO','YES','NUMBER') COLLATE utf8_bin DEFAU
  `subpartition_type` enum('HASH','KEY_51','KEY_55','LINEAR_HASH','LINEAI
  `subpartition_expression` varchar(2048) COLLATE utf8_bin DEFAULT NULL,
  `subpartition_expression_utf8` varchar(2048) COLLATE utf8_bin DEFAULT N
  `default_subpartitioning` enum('NO','YES','NUMBER') COLLATE utf8_bin DE
  `created` timestamp NOT NULL,
  `last_altered` timestamp NOT NULL,
  `view_definition` longblob,
  `view_definition_utf8` longtext COLLATE utf8_bin,
  `view_check_option` enum('NONE','LOCAL','CASCADED') COLLATE utf8_bin DE
  `view_is_updatable` enum('NO','YES') COLLATE utf8_bin DEFAULT NULL,
  `view_algorithm` enum('UNDEFINED','TEMPTABLE','MERGE') COLLATE utf8_bin
```

```
  `view_security_type` enum('DEFAULT','INVOKER','DEFINER') COLLATE utf8_b
  `view_definer` varchar(288) COLLATE utf8_bin DEFAULT NULL,
  `view_client_collation_id` bigint(20) unsigned DEFAULT NULL,
  `view_connection_collation_id` bigint(20) unsigned DEFAULT NULL,
  `view_column_names` longtext COLLATE utf8_bin,
  `last_checked_for_upgrade_version_id` int(10) unsigned NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `schema_id` (`schema_id`,`name`),
  UNIQUE KEY `engine` (`engine`,`se_private_id`),
  KEY `engine_2` (`engine`),
  KEY `collation_id` (`collation_id`),
  KEY `tablespace_id` (`tablespace_id`),
  KEY `type` (`type`),
  KEY `view_client_collation_id` (`view_client_collation_id`),
  KEY `view_connection_collation_id` (`view_connection_collation_id`),
  CONSTRAINT `tables_ibfk_1` FOREIGN KEY (`schema_id`) REFERENCES `schema
  CONSTRAINT `tables_ibfk_2` FOREIGN KEY (`collation_id`) REFERENCES `col
  CONSTRAINT `tables_ibfk_3` FOREIGN KEY (`tablespace_id`) REFERENCES `ta
  CONSTRAINT `tables_ibfk_4` FOREIGN KEY (`view_client_collation_id`) REF
  CONSTRAINT `tables_ibfk_5` FOREIGN KEY (`view_connection_collation_id`)
) /*!50100 TABLESPACE `mysql` */ ENGINE=InnoDB AUTO_INCREMENT=549 DEFAULT
1 row in set (0.00 sec)
```

通过以上 mysql.tables 的表定义可以获得存储引擎中实际存储的元信息字段。DD tables 包括 tables 、 schemata 、 columns 、 column_type_elements 、 indexes 、 index_column_usage 、 foreign_keys 、 foreign_key_column_usage 、 table_partitions 、 table_partition_values 、 index_partitions、triggers、check_constraints、view_table_usage、view_routine_usage 等。

Storage_adapter 是访问持久存储引擎的处理类，包括 get() / drop() / store() 等接口。当初次获取一个表的元信息时，会调用 Storage_adapter::get() 接口，处理过程如下：

```
Storage_adapter::get()
  // 根据访问对象类型，将依赖的 DD tables 加入到 open table list 中
  |--Open_dictionary_tables_ctx::register_tables<T>()
```

```
    |--Table_impl::register_tables()
  |--Open_dictionary_tables_ctx::open_tables() // 调用 Server 层接口打开所有
  |--Raw_table::find_record() // 直接调用 handler 接口根据传入的 key（比如表名）
    |--handler::ha_index_read_idx_map() // index read
  // 从读取到的 record 中解析出对应属性，调用 field[field_no]->val_xx() 函数
  |--Table_impl::restore_attributes()
    // 通过调用 restore_children() 函数从与该对象关联的其他 DD 表中根据主外键读取完
    |--Table_impl::restore_children()
  |--返回完整的 DD cache 对象
```

上述在获取列和属性的对应关系时，根据的是 Tables 对象的枚举类型下标，按顺序包含了该类型 DD 表中的所有列，与上述表定义是一一对应的。因此如果我们需要新增 DD 表中存储的列时，也需要往下面枚举类型定义中加入对应的列，并且在 Table_impl::restore_attributes() / Table_impl::store_attributes() 函数中添加对新增列的读取和存储操作。

```
class Tables : public Entity_object_table_impl {
  enum enum_fields {
    FIELD_ID,
    FIELD_SCHEMA_ID,
    FIELD_NAME,
    FIELD_TYPE,
    FIELD_ENGINE,
    FIELD_MYSQL_VERSION_ID,
    FIELD_ROW_FORMAT,
    FIELD_COLLATION_ID,
    FIELD_COMMENT,
    FIELD_HIDDEN,
    FIELD_OPTIONS,
    FIELD_SE_PRIVATE_DATA,
    FIELD_SE_PRIVATE_ID,
    FIELD_TABLESPACE_ID,
    FIELD_PARTITION_TYPE,
    FIELD_PARTITION_EXPRESSION,
    FIELD_PARTITION_EXPRESSION_UTF8,
```

```
      FIELD_DEFAULT_PARTITIONING,
      FIELD_SUBPARTITION_TYPE,
      FIELD_SUBPARTITION_EXPRESSION,
      FIELD_SUBPARTITION_EXPRESSION_UTF8,
      FIELD_DEFAULT_SUBPARTITIONING,
      FIELD_CREATED,
      FIELD_LAST_ALTERED,
      FIELD_VIEW_DEFINITION,
      FIELD_VIEW_DEFINITION_UTF8,
      FIELD_VIEW_CHECK_OPTION,
      FIELD_VIEW_IS_UPDATABLE,
      FIELD_VIEW_ALGORITHM,
      FIELD_VIEW_SECURITY_TYPE,
      FIELD_VIEW_DEFINER,
      FIELD_VIEW_CLIENT_COLLATION_ID,
      FIELD_VIEW_CONNECTION_COLLATION_ID,
      FIELD_VIEW_COLUMN_NAMES,
      FIELD_LAST_CHECKED_FOR_UPGRADE_VERSION_ID,
      NUMBER_OF_FIELDS  // Always keep this entry at the end of the enum
    };
};

```

四 多级缓存

为了避免每次对元数据对象的访问都需要去持久存储中读取多个表的数据，使生成的元数据内存对象能够复用，data dictionary 实现了两级缓存的架构，第一级是 client local 独享的，核心数据结构为 Local_multi_map，用于加速在当前线程中对于相同对象的重复访问，同时在当前线程涉及对 DD 对象的修改（DDL）时管理 committed、uncommitted、dropped 几种状态的对象。第二级就是比较常见的多线程共享的缓存，核心数据结构为 Shared_multi_map，包含着所有线程都可以访问到其中的对象，所以会做并发控制的处理。

两级缓存的底层实现很统一，都是基于 hash map 的，目前的实现是 std::map。Local_multi_map 和 Shared_multi_map都是派生于 Multi_map_base。

```
template <typename T>
class Multi_map_base {
 private:
  Element_map<const T *, Cache_element<T>> m_rev_map;  // Reverse element

  Element_map<typename T::Id_key, Cache_element<T>>
      m_id_map;  // Id map instance.
  Element_map<typename T::Name_key, Cache_element<T>>
      m_name_map;  // Name map instance.
  Element_map<typename T::Aux_key, Cache_element<T>>
      m_aux_map;  // Aux map instance.
};

template <typename K, typename E>
class Element_map {
 public:
  typedef std::map<K, E *, std::less<K>,
                   Malloc_allocator<std::pair<const K, E *>>>
      Element_map_type;  // Real map type.

 private:
  Element_map_type m_map;  // The real map instance.
  std::set<K, std::less<K>,
           Malloc_allocator<K>>
      m_missed;  // Cache misses being handled.
};
```

之所以叫 Multi_map_base，是因为其中包含了多个 hash map，适合用户根据不同类型的 key 来获取缓存对象，比如 id、name、DD cache 本身等。Element_map 就是对 std::map 的一个封装，key 为前述几种类型之一，value 为 DD cache 对象指针的一个封装 Cache_element，封装了对象本身和引用计数。

Multi_map_base 对象实现了丰富的 m_map() 模板函数，可以很方便的根据 key 的类型不同选择到对应的 hash map。

Shared_multi_map 与 Local_multi_map 的不同在于，Shared_multi_map 还引入了一组 latch 与 condition variable 用于并发访问中的线程同步与 cache miss 的处理。同时对 Cache_element 对象做了内存管理和复用的相关能力。

1 局部缓存

一级缓存位于每个 Dictionary_client （每个 client 与线程 THD 一一对应）内部，由不同状态（committed、uncommitted、dropped）的 Object_registry 组成。每个 Object_registry 由不同元数据类型的 Local_multi_map 组成，用于管理不同类型的对象（比如表、schema、字符集、统计数据、Event 等）缓存。

```
class Dictionary_client {
  Object_registry m_registry_committed;    // Registry of committed objec
  Object_registry m_registry_uncommitted;  // Registry of uncommitted obj
  Object_registry m_registry_dropped;      // Registry of dropped objects
  THD *m_thd;                          // Thread context, needed for cache
  Auto_releaser m_default_releaser;  // Default auto releaser.
  Auto_releaser *m_current_releaser;  // Current auto releaser.
};

class Object_registry {
  std::unique_ptr<Local_multi_map<Abstract_table>> m_abstract_table_map;
  std::unique_ptr<Local_multi_map<Charset>> m_charset_map;
  std::unique_ptr<Local_multi_map<Collation>> m_collation_map;
  std::unique_ptr<Local_multi_map<Column_statistics>> m_column_statistics
  std::unique_ptr<Local_multi_map<Event>> m_event_map;
  std::unique_ptr<Local_multi_map<Resource_group>> m_resource_group_map;
  std::unique_ptr<Local_multi_map<Routine>> m_routine_map;
  std::unique_ptr<Local_multi_map<Schema>> m_schema_map;
  std::unique_ptr<Local_multi_map<Spatial_reference_system>>
      m_spatial_reference_system_map;
```

```
  std::unique_ptr<Local_multi_map<Tablespace>> m_tablespace_map;
};

template <typename T>
class Local_multi_map : public Multi_map_base<T> {};
```

其中 committed 状态的 registry 就是我们访问数据库中已经存在的对象时，将其 DD cache object 存放在局部缓存中的位置。uncommitted 和 dropped 状态的存在，主要用于当前连接执行的是一条 DDL 语句，在执行过程中会将要 drop 的旧表对应的 DD object 存放在 dropped 的 registry 中，将还未提交的新表定义对应的 DD object 存放在 uncommitted 的 registry 中，用于执行状态的区分。

2 共享缓存

共享缓存是 Server 全局唯一的，使用单例 Shared_dictionary_cache 来实现。与上述局部缓存中 Object_registry 相似，Shared_dictionary_cache 也需要包含针对各种类型对象的缓存。与 Multi_map_base 实现根据 key 类型自动选取对应 hash map 的模版函数相似，Object_registry 和 Shared_dictionary_cache 也都实现了根据访问对象的类型选择对应缓存的 m_map() 函数，能够很大程度上简化函数调用。

```
class Shared_dictionary_cache {
  Shared_multi_map<Abstract_table> m_abstract_table_map;
  Shared_multi_map<Charset> m_charset_map;
  Shared_multi_map<Collation> m_collation_map;
  Shared_multi_map<Column_statistics> m_column_stat_map;
  Shared_multi_map<Event> m_event_map;
  Shared_multi_map<Resource_group> m_resource_group_map;
  Shared_multi_map<Routine> m_routine_map;
  Shared_multi_map<Schema> m_schema_map;
  Shared_multi_map<Spatial_reference_system> m_spatial_reference_system_r
  Shared_multi_map<Tablespace> m_tablespace_map;
};

template <typename T>
```

```
class Shared_multi_map : public Multi_map_base<T> {
 private:
  static const size_t initial_capacity = 256;

  mysql_mutex_t m_lock;          // Single mutex to lock the map.
  mysql_cond_t m_miss_handled;  // Broadcast a miss being handled.

  Free_list<Cache_element<T>> m_free_list;  // Free list.
  std::vector<Cache_element<T> *>
      m_element_pool;  // Pool of allocated elements.
  size_t m_capacity;   // Total capacity, i.e., if the
                       // number of elements exceeds this
                       // limit, shrink the free list.
}
```

与局部缓存可以无锁访问 hash map 不同，共享缓存在获取 / 释放 DD cache object 时都需要加锁来完成引用计数的调整和防止访问过程中被 destroy 掉。

3 缓存获取过程

用户通过 client 调用元数据对象获取函数，传入元数据的 name 字符串，然后构建出对应的 name key，通过 key 去缓存中获取元数据对象。获取的整体过程就是一级局部缓存 -> 二级共享缓存 -> 存储引擎。

```
// Get a dictionary object.
template <typename K, typename T>
bool Dictionary_client::acquire(const K &key, const T **object,
                                bool *local_committed,
                                bool *local_uncommitted) {
  // Lookup in registry of uncommitted objects
  T *uncommitted_object = nullptr;
  bool dropped = false;
  acquire_uncommitted(key, &uncommitted_object, &dropped);

```

```
    ...

    // Lookup in the registry of committed objects.
    Cache_element<T> *element = NULL;
    m_registry_committed.get(key, &element);

    ...

    // Get the object from the shared cache.
    if (Shared_dictionary_cache::instance()->get(m_thd, key, &element)) {
      DBUG_ASSERT(m_thd->is_system_thread() || m_thd->killed ||
                  m_thd->is_error());
      return true;
    }
  }
```

在一级局部缓存中获取时，会优先去 uncommitted 和 dropped 的 registry 获取，因为这两者是最新的修改，同时判断获取对象是否已经被 dropped。之后再会去 committed 的 registry 获取，如果获取到就直接返回，反之则去二级共享缓存中尝试获取。

Cache miss

共享缓存的获取过程在 Shared_multi_map::get() 中实现。就是加锁后直接的 hash map 查找，如果存在则给引用计数递增后返回；如果不存在，就会进入到 cache miss 的处理过程，调用上面介绍的存储引擎的接口 Storage_adapter::get() 从 DD tables 中读取，创建出来后依次加入共享缓存和局部缓存 committed registry 中。

```
// Get a wrapper element from the map handling the given key type.
template <typename T>
template <typename K>
bool Shared_multi_map<T>::get(const K &key, Cache_element<T> **element) {
  Autolocker lock(this);
  *element = use_if_present(key);
  if (*element) return false;
```

```
8
9     // Is the element already missed?
10    if (m_map<K>()->is_missed(key)) {
11      while (m_map<K>()->is_missed(key))
12        mysql_cond_wait(&m_miss_handled, &m_lock);
13
14      *element = use_if_present(key);
15
16      // Here, we return only if element is non-null. An absent element
17      // does not mean that the object does not exist, it might have been
18      // evicted after the thread handling the first cache miss added
19      // it to the cache, before this waiting thread was alerted. Thus,
20      // we need to handle this situation as a cache miss if the element
21      // is absent.
22      if (*element) return false;
23    }
24
25    // Mark the key as being missed.
26    m_map<K>()->set_missed(key);
27    return true;
28  }
```

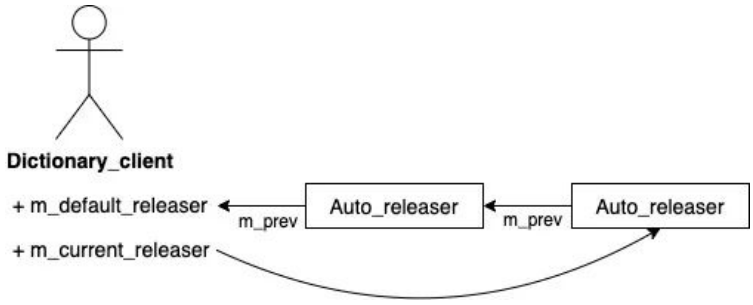
由于开表访问 DD tables，构建 DD cache object 的过程相对耗时，不会一直给 Shared_multi_map 加锁，因此需要对并发访问的 client 做并发控制。DD 的实现方法是第一个访问的 client 会将 cache miss 的 key 加入到 Shared_multi_map的 m_missed 集合中，这个集合包含着现在所有正在读取元数据的对象 key 值。之后访问的 client 看到目标 key 值在 m_missed 集合中就会进入等待。

当第一个 client 获取到完整的 DD cache object，加入到共享缓存之后，移除 m_missed 集合中对应的 key，并通过广播的方式通知之前等待的线程重新在共享缓存中获取。

五 Auto_releaser

Auto_releaser 是一个 RAII 类，基本上在使用 client 访问 DD cache 前都会做一个封装，保证在整个 Auto_releaser 对象存在的作用域内，所获取到的 DD cache 对象都会在局部缓存中存在不释放。

Auto_releaser 包含需要 release 的对象 registry，通过 auto_release() 函数收集着当前 client 从共享缓存中获取到的 DD cache 对象，在超出其作用域进行析构时自动 release 对象，从局部缓存 committed 的 registry 中移除对象，并且在共享缓存中的引用计数递减。



在嵌套函数调用过程中，可能在每一层都会有自己的 Auto_releaser，他们之间通过一个简单的链表指针连接起来。在函数返回时将本层需要 release 的对象 release 掉，需要返回给上层使用的 DD cache 对象交给上层的 Auto_releaser 来负责。通过 transfer_release() 可以在不同层次的 Auto_releaser 对象间转移需要 release 的对象，可以灵活的指定不再需要 DD cache 对象的层次。

六 应用举例：inplace DDL 过程中对 DD 的操作

在 MySQL inplace DDL 执行过程中，会获取当前表定义的 DD cache 对象，然后根据实际的 DDL 操作内容构造出新对应的 DD 对象。然后依次调用 client 的接口完成对当前表定义的删除和新表定义的存储。

```
{
  if (thd->dd_client()->drop(table_def)) goto cleanup2;
  table_def = nullptr;

  DEBUG_SYNC_C("alter_table_after_dd_client_drop");

  // Reset check constraint's mode.
  reset_check_constraints_alter_mode(altered_table_def);

  if ((db_type->flags & HTON_SUPPORTS_ATOMIC_DDL)) {
    /*
      For engines supporting atomic DDL we have delayed storing new
```

```
      table definition in the data-dictionary so far in order to avoid
      conflicts between old and new definitions on foreign key names.
      Since the old table definition is gone we can safely store new
      definition now.
    */
    if (thd->dd_client()->store(altered_table_def)) goto cleanup2;
  }
}

...

/*
  If the SE failed to commit the transaction, we must rollback the
  modified dictionary objects to make sure the DD cache, the DD
  tables and the state in the SE stay in sync.
*/
if (res)
  thd->dd_client()->rollback_modified_objects();
else
  thd->dd_client()->commit_modified_objects();
```

在 drop() 过程中，会将当前表定义的 DD cache 对象对应的数据从存储引擎中删除，然后从共享缓存中移除（这要求当前对象的引用计数仅为1，即只有当前线程使用），之后加入到 dropped 局部缓存中。

在 store() 过程中，会将新的表定义写入存储引擎，并且将对应的 DD cache 对象加入 uncommitted 缓存中。

在事务提交或者回滚后，client 将局部缓存中的 dropped 和 uncommitted registry 清除。由于 InnoDB 引擎支持事务，持久存储层面的数据会通过存储引擎的接口提交或回滚，不需要 client 额外操作。

在这个过程中，由于 MDL(metadata lock) 的存在，不会有其他的线程尝试访问正在变更对象的 DD object，所以可以安全的对 Shared_dictionary_cache 进行操作。当 DDL 操作结束（提交或回滚），

释放 EXCLUSIVE 锁之后，新的线程就可以重新从存储引擎上加载新的表定义。

七 总结

MySQL data dictionary 解决了背景所述旧架构中的诸多问题，使元数据的访问更加安全，存储和管理成本更低。架构实现非常的精巧，通过大量的模版类实现使得代码能够最大程度上被复用。多层缓存的实现也能显著提升访问效率。通过 client 简洁的接口，让 Server 层和存储层能在任何地方方便的访问元数据。

八 关于我们

PolarDB 是阿里巴巴自主研发的云原生分布式关系型数据库，于2020年进入Gartner全球数据库Leader象限，并获得了2020年中国电子学会颁发的科技进步一等奖。PolarDB 基于云原生分布式数据库架构，提供大规模在线事务处理能力，兼具对复杂查询的并行处理能力，在云原生分布式数据库领域整体达到了国际领先水平，并且得到了广泛的市场认可。在阿里巴巴集团内部的最佳实践中，PolarDB还全面支撑了2020年天猫双十一，并刷新了数据库处理峰值记录，高达1.4亿TPS。欢迎有志之士加入我们，简历请投递到hope.lb@alibaba-inc.com，期待与您共同打造世界一流的下一代云原生分布式关系型数据库。

参考

[1] MySQL8.0DataDictionary:BackgroundandMotivation
http://mysqlserverteam.com/mysql-8-0-data-dictionary-background-and-motivation/

[2] MySQL 8.0: Data Dictionary Architecture and Design
http://mysqlserverteam.com/mysql-8-0-data-dictionary-architecture-and-design/

[3] Source code mysql / mysql-server 8.0.18
https://github.com/mysql/mysql-server/tree/mysql-8.0.18