模块二实验报告

负责人：常潍麟　日本语言文学　3210100056
电子邮箱：xishanqiming@gmail.com
开发时间：2023 年 5 月 25 日 — 31日

一．实验概述

在 MiniSQL 的设计中，Record Manager 负责管理数据表中所有的记录，它能够支持记录的插入、删除与查找操作，并对外提供相应的接口。与记录相关的概念有以下几个：

- 列（Column）：在 src/include/record/column.h 中被定义，用于定义和表示数据表中的某一个字段，即包含了这个字段的字段名、字段类型、是否唯一等等；
- 模式（Schema）：在 src/include/record/schema.h 中被定义，用于表示一个数据表或是一个索引的结构。一个 Schema 由一个或多个的 Column 构成；
- 域（Field）：在 src/include/record/field.h 中被定义，它对应于一条记录中某一个字段的数据信息，如存储数据的数据类型，是否是空，存储数据的值等等；
- 行（Row）：在 src/include/record/row.h 中被定义，与元组的概念等价，用于存储记录或索引键，一个 Row 由一个或多个 Field 构成。
- 此外，与数据类型相关的定义和实现位于 src/include/record/types.h 中。

其中，列和行的概念与表中的列和行的概念不完全一致，可以理解为是便于对数据进行组织和操作的一种基本的结构。

在模块二中，主要包括了两个部分，下面将对这两部分的实现进行分别阐述：

1. 记录与模式的实现
2. 通过堆表管理记录

二．记录与模式的实现

在实现通过堆表来管理记录之前，需要先将数据进行序列化和反序列化操作。为了能够持久化存储 Row、Field、Schema 和 Column 对象，需要将这些对象序列化成字节流（char*）以写入数据页中。与之相对，同样需要能够提供一种反序列化的方法，从磁盘的数据页中 char*类型的字节流中反序列化出我们需要的对象。

总而言之，序列化和反序列化是将数据库系统中的对象（如：记录、索引、目录等）进行内外存格式转化的过程。序列化是将内存中的逻辑数据（即对象）通过一定的方式，转换成便于在文件中存储的物理数据；反序列化则是从存储的物理数据中恢复出逻辑数据。但两者的目的都是为了实现数据的持久化。

为了确保数据能够被正确存储，在 Row、Schema 和 Column 对象中都引入了 ” 魔数 ” MAGIC_NUM，它在序列化时被写入到字节流的头部并在反序列化中被读出，以验证在反序列化时生成的对象是否一致。

本节实现了 Row、Schema 和 Column 对象的序列化，反序列化和 GetSerializedSize 方法，我选取了类中较为重要的数据（值得被持久化存储的数据）进行序列化操作。但实际上，在测试代码中只会验证序列化前后的对象是否匹配，并对上层是完全透明的。

其中，SerializeTo 和 DeserializeFrom 函数的返回值为 uint32_t 类型，它表示在序列化和反序列化过程中 buf 指针向前推进了多少个字节。此外，在序列化和反序列化中使用了一些宏（如：ALLOC，ALLOC_P），这些宏被定义在 include/common/macros.h 中，可根据实际需要使用。

在 Row 类型对象进行序列化时，对于空值的 Field，可以通过空位图的方式标记；对于 Row 类型对象的反序列化，在反序列化每一个 Field 时，反序列化出来的 Field 的内存都由该 Row 对象维护。对于 Column 和 Schema 类型对象的反序列化，分配后新生成的对象于参数 column 和 schema 中返回。

1. Row 类

(1)Row::SerializeTo 函数

```
uint32_t Row::SerializeTo(char *buf, Schema *schema) const {}
```

- 功能：将 Row 对象序列化到缓冲区 buf 中。首先检查传入的 schema 是否为空并且字段的数量是否和列数匹配。然后将字段数量和 null 字段数量写入缓冲区。对于每个为 null 的字段，将其索引会写入到缓冲区。最后将非 null 字段会被序列化并写入缓冲区。
- 输入：buf（一个字符指针，用于存储序列化的结果），schema（一个 Schema 类的指针，用于定义字段的结构）。
- 输出：返回序列化后最后写入的偏移量。

(2)Row::DeserializeFrom 函数

```
uint32_t Row::DeserializeFrom(char *buf, Schema *schema) {}
```

- 功能：从缓冲区 buf 中反序列化 Row 对象。首先检查传入的 schema 是否为空，然后从缓冲区读取字段数量和 null 字段数量。接着构建 bitmap 来标记哪些字段为 null。最后，非 null 字段会从缓冲区中反序列化。
- 输入：buf（一个字符指针，用于存储序列化的结果），schema（一个 Schema 类的指针，用于定义字段的结构）。
- 输出：返回读取结束的偏移量。

(3)Row::GetSerializedSize 函数

```
uint32_t Row::GetSerializedSize(Schema *schema) const {}
```

- 功能：计算 Row 对象序列化后的大小。函数会检查 schema 是否为空并且字段的数量是否和 schema 的列数匹配。然后，计算非 null 字段的序列化大小。如果 fields 为空，返回 0
- 输入：schema（一个 Schema 类的指针，用于定义字段的结构）。
- 输出：返回 Row 对象序列化后的大小。

(4)Row::GetKeyFromRow 函数

```
void Row::GetKeyFromRow(const Schema *schema, const Schema *key_schema, Row &key_row) {}
```

- 功能：从当前行中提取出 key_schema 定义的字段，然后将这些字段设置到 key_row 中。
- 输入：schema（一个 Schema 类的指针，定义了原始数据的结构）；key_schema（一个 Schema 类的指针，定义了 key 的结构）；key_row（一个 Row 对象的引用，用于保存结果）。

2. Column 类

(1)Column 类的构造函数

在 Column 类中针对实际需求的不同，设置了三种构造函数：

- Column(std::string column_name, TypeId type, uint32_t index, bool nullable, bool unique)：用于非字符型(kTypeChar)列的构造。接收列名称、类型、索引位置、是否可为空、是否唯一等信息。根据类型计算列的长度。
- Column(std::string column_name, TypeId type, uint32_t length, uint32_t index, bool nullable, bool unique)：用于字符型列的构造。除了接收基础信息外，还需要

接收一个字符型列的长度。

- Column(const Column *other)：拷贝构造函数，根据已存在的 Column 对象创建一个新的 Column 对象。

(2)Column::SerializeTo 函数

```
uint32_t Column::SerializeTo(char *buf) const {}
```

- 功能：将 Column 对象序列化到给定的缓冲区中，返回写入缓冲区的字节数。序列化的顺序是：魔数、列名称长度、列名称、列类型、列长度、列索引、是否可空、是否唯一。
- 输入：buf（一个字符指针，用于存储序列化的结果），schema（一个 Schema 类的指针，用于定义字段的结构）。
- 输出：返回序列化后最后写入的偏移量。

(3)Column::DeserializeFrom 函数

```
uint32_t Column::DeserializeFrom(char *buf, Column *&column) {}
```

- 功能：反序列化函数。从缓冲区中读取数据，并生成一个新的 Column 对象。反序列化的顺序与序列化的顺序相同。在反序列化时，会检查魔数是否正确，如果不正确，表示缓冲区中的数据并非一个有效的 Column 对象。
- 输入：buf（一个字符指针，用于存储序列化的结果），schema（一个 Schema 类的指针，用于定义字段的结构）。
- 输出：返回读取结束的偏移量。

(4)Column::GetSerializedSize 函数

```
uint32_t Column::GetSerializedSize() const {}
```

- 功能：获取 Column 对象序列化后的大小。计算的元素与序列化的顺序相同。
- 输出：返回 Column 对象序列化后的大小。

3. Schema 类

(1)Schema::SerializeTo 函数

```
uint32_t Schema::SerializeTo(char *buf) const {}
```

- 功能：将 Schema 对象序列化到字符缓冲区 buf 中，返回序列化后的总字节数。首先将 SCHEMA_MAGIC_NUM 写入 buf 中，然后更新偏移量 ofs。接着，将 columns_的大小写入 buf 中，再次更新偏移量。最后，遍历 columns_，对每一个 Column 对象进行序列化并写入 buf 中，每次序列化后更新偏移量。
- 输入：buf（一个字符指针，用于存储序列化的结果），schema（一个 Schema 类的指针，用于定义字段的结构）。
- 输出：返回序列化后最后写入的偏移量。

(2)Row::DeserializeFrom 函数

```
uint32_t Schema::DeserializeFrom(char *buf, Schema *&schema) {}
```

- 功能：实现了从字符缓冲区 buf 中反序列化出 Schema 对象，并将其赋值给传入的指针 schema，最后返回反序列化后的总字节数。首先从 buf 中读出 SCHEMA_MAGIC_NUM，检查其是否正确。然后读出 columns_的大小，遍历 buf 中的数据，反序列化出每个 Column 对象，并将其添加到 columns 向量中。最后生成一个新的 Schema 对象并返回从缓冲区中读取的字节数。
- 输入：char *buf：字符缓冲区，用于存储序列化后的 Schema 对象。Schema *&schema：一个 Schema 指针的引用，用于存储反序列化出的 Schema 对象。
- 输出：返回反序列化后的总字节数。

(3)Schema::GetSerializedSize 函数

```
uint32_t Schema::GetSerializedSize() const {}
```

- 功能：获取 Schema 对象序列化后的大小。遍历 columns_，获取每个 Column 对象序列化后的字节数并累加到 size 中。最后，返回序列化后的总字节数，包括 SCHEMA_MAGIC_NUM 和 columns_长度两个 uint32_t 类型的字节数。
- 输出：返回 Schema 对象序列化后的大小。

4. 类的设计和实现细节

- 在 Row 类中，使用成员变量 fields_，用于保存当前行的所有字段。fields_nums 记录当前行的字段总数，null_nums 记录空字段数量。heap_ 用于分配内存。
- 在 Column 类中，主要用于表示数据库表中的一列。每个 Column 对象有一个列名，列类型，长度，是否允许空值，是否唯一等属性。此外还提供对象的序列化和反序列化方法，允许对象以二进制格式存储和读取，在需要时从磁盘上加载。
- 在 Schema 类中，主要是考虑是用于操作记录模式，包括序列化和反序列化 Schema 对象以及获取 Schema 对象序列化后的字节数。Schema 对象中包含一组

Column 对象，由成员变量 columns_（一个 Column 对象的 vector）存储。

5. 可改进的地方

在序列化和反序列化代码中，直接使用了 memcpy 来读写数据，但是不能处理字节序的问题。在不同 CPU 架构（X64，X86）的机器上，字节序可能不同，这可能会导致序列化和反序列化出现不一致的问题。

在设计中，字段是以 Field 对象的形式保存的，这种方式可能会增加内存使用和处理的复杂性。建议直接保存字段的值，而不是 Field 对象。

更重要的是，当前的代码在处理错误时主要依赖 ASSERT 和 LOG(ERROR)，在实际的生产环境中，需要更强大和灵活的错误处理机制。例如目前在代码中，如果反序列化出来的 SCHEMA_MAGIC_NUM 不正确，我们直接用 ASSERT 进行错误处理，导致程序直接退出。应该提供异常处理机制，如抛出具体异常，然后在调用处进行捕获和处理。

三．通过堆表管理记录

对于数据表中的每一行记录，都有一个唯一标识符 RowId 与之对应。RowId 具有逻辑和物理双重意义。在物理意义上，RowId 是一个 64 位整数，用于唯一标识每行记录；在逻辑意义上，它的高 32 位存储的是该 RowId 对应记录所在数据页的 page_id，低 32 位存储的是该 RowId 在 page_id 对应的数据页中对应的是第几条记录。

RowId 的作用有两个：一是在索引中，叶结点中存储的键值对是索引键 Key 到 RowId 的映射，通过索引键 Key，沿着索引查找，就能够得到该索引键对应记录的 RowId，也就能够在堆表中定位到该记录；二是在堆表中，借助 RowId 中存储的逻辑信息 page_id 和 slot_num，可以快速地定位到其对应的记录在物理文件中的位置。

堆表 TableHeap 是将记录以无序的堆进行组织的数据结构，不同数据页（TablePage） 之间通过双向链表连接。堆表中的记录通过 RowId 进行定位。RowId 记录了该行记录所在的 page_id 和 slot_num，其中 slot_num 用于定位记录在这个数据页中的下标位置。

堆表中的每个数据页都由表头（Table Page Header）、空闲空间（Free Space）和已经插入的数据（Inserted Tuples）三部分组成。表头在页中从左往右扩展，记录了 PrevPageId、NextPageId、FreeSpacePointer 及每条记录在 TablePage 中的偏移和

长度；插入的记录在页中从右向左扩展，插入时将 FreeSpacePointer 的位置向左移动。

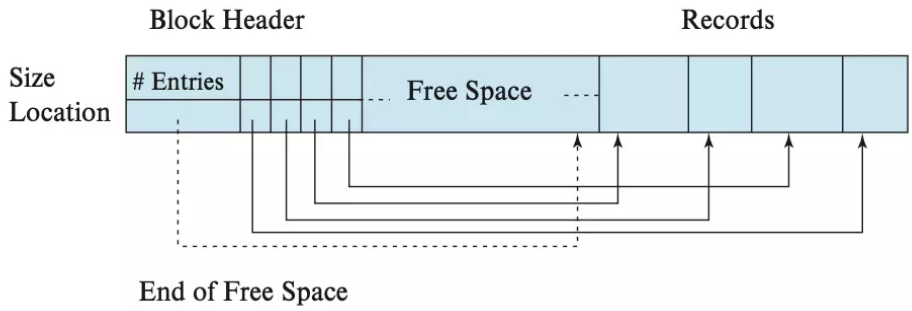


Figure 13.6 Slotted-page structure.

当向堆表中插入一条记录时，可以沿着 TablePage 构成的链表依次查找，直到找到第一个能够容纳该记录的 TablePage（First Fit 策略）。当需要删除指定 RowId 对应的记录时，可以通过打上 Delete Mask 来标记记录被删除，在之后某个时间段再从物理意义上真正删除该记录。

对更新操作，需要分两种情况考虑。如果 TablePage 能容纳下更新后的数据，则可直接在数据页中进行更新。如果 TablePage 不能够容纳下更新后的数据，则需要分配新页。
同时，在堆表中还需要实现迭代器 TableIterator，以便上层模块遍历堆表的所有记录。

综上，在本节中实现了堆表的插入、删除、查询以及堆表记录迭代器的相关的功能。

- TableHeap:InsertTuple(&row, *txn): 向堆表中插入一条记录，插入记录后生成的 RowId 需要通过 row 对象返回（即 row.rid_）；
- TableHeap:UpdateTuple(&new_row, &rid, *txn)：将 RowId 为 rid 的记录 old_row 替换成新的记录 new_row，并将 new_row 的 RowId 通过 new_row.rid_ 返回；
- TableHeap:ApplyDelete(&rid, *txn)：从物理意义上删除这条记录；
- TableHeap:GetTuple(*row, *txn)：获取 RowId 为 row->rid_的记录；
- TableHeap:FreeHeap()：销毁整个 TableHeap 并释放这些数据页；
- TableHeap::Begin()：获取堆表的首迭代器；
- TableHeap::End()：获取堆表的尾迭代器；
- TableIterator 类中的成员操作符
  - TableIterator::operator++()：移动到下一条记录，通过++iter 调用；
  - TableIterator::operator++(int)：移动到下一条记录，通过 iter++调用；

1. TableHeap 类

(1)TableHeap::InsertTuple 函数

```
bool TableHeap::InsertTuple(Row &row, Transaction *txn) {}
```

- 功能：将一条记录插入到堆表中。首先会检查要插入的记录的序列化大小是否超过了页面允许的最大记录大小。如果超过则插入失败并返回 false。然后获取堆表的第一个页面在该页面上尝试插入记录。如果插入成功则返回 true。如果当前页面没有足够的空间来插入新的记录，则获取下一页并在该页面上尝试插入记录。如果没有下一页，就创建新的页面并在新的页面上插入记录。
- 输入：row: 表示要插入的记录的对象，它是一个引用，因此插入成功后，row 的 rid_ 会被设置为新插入的记录的 RowId。txn: 表示正在执行的事务的对象。
- 输出：如果插入成功，则返回 true。否则返回 false。

(2)TableHeap::MarkDelete 函数

```
bool TableHeap::MarkDelete(const RowId &rid, Transaction *txn) {}
```

- 功能：首先获取包含指定记录的页，然后在该页上标记指定的记录为已删除状态。
- 输入：rid: 需要标记为已删除状态记录的 RowId。txn: 表示正在执行的事务的对象。
- 输出：如果成功标记，则返回 true。否则返回 false。

(3)TableHeap::UpdateTuple 函数

```
bool TableHeap::UpdateTuple(const Row &row, const RowId &rid, Transaction
 *txn) {}
```

- 功能：更新一个已存在的记录。首先根据指定的 RowId 来获取包含该记录的页。然后在该页上尝试更新指定的记录。如果更新成功，则函数返回 true。如果更新失败（比如，新的记录的大小大于旧的记录的大小，而当前页面没有足够的空间来存储新的记录），则会首先删除旧的记录，然后在合适的页面上插入新的记录。
- 输入：row: 包含新数据的记录对象。rid: 需要被更新的记录的 RowId。txn: 表示正在执行的事务的对象。
- 输出：如果更新成功，则返回 true。否则返回 false。

(4)TableHeap::ApplyDelete 函数

```
void TableHeap::ApplyDelete(const RowId &rid, Transaction *txn) {}
```

- 功能：首先获取包含指定记录的页，然后从物理意义上删除一个已标记为已删除状态的记录。

- 输入：rid: 需要被删除的记录的 RowId。txn: 表示正在执行的事务的对象。

(5)TableHeap::RollbackDelete 函数

```
void TableHeap::RollbackDelete(const RowId &rid, Transaction *txn) {}
```

- 功能：首先获取包含指定记录的页，然后在该页上回滚一个已标记为已删除状态的记录的删除操作，即将该记录的状态改回到未删除状态。
- 输入：rid: 需要被回滚删除操作的记录的 RowId。txn: 表示正在执行的事务的对象。

(6)TableHeap::GetTuple 函数

```
bool TableHeap::GetTuple(Row *row, Transaction *txn)  {}
```

- 功能：根据指定的 RowId 获取包含该记录的页。然后在该页获取指定记录的数据。
- 输入：row: 一个记录对象的指针，该对象的 rid_ 成员变量被设置为需要获取的记录的 RowId。txn: 表示正在执行的事务的对象。
- 输出：如果成功获取记录，则返回 true，并将获取的记录的数据保存到输入的记录对象中。否则返回 false。

(7)TableHeap::DeleteTable 函数

```
void TableHeap::DeleteTable(page_id_t page_id) {}
```

- 功能：将整个表删除，首先获取指定页面 ID 的页面，然后如果存在下一个页面，递归调用 DeleteTable 函数删除下一页，最后删除当前页面。
- 输入：page_id: 需要被删除的表的第一页的页面 ID。

(8)TableHeap::Begin 函数

```
TableIterator TableHeap::Begin(Transaction *txn) {}
```

- 功能：获取堆表的首迭代器。首先获取堆表的第一页。然后在该页上获取第一个记录的 RowId，并以该 RowId 作为迭代器的当前位置。
- 输入：txn: 表示正在执行的事务的对象。
- 输出：返回一个 TableIterator 对象，表示堆表的首迭代器。

(9)TableHeap::End 函数

```
TableIterator TableHeap::End() {}
```

- 功能：获取堆表的尾迭代器。直接返回一个包含无效 RowId 的 TableIterator 对象作为堆表的尾迭代器。
- 输出：返回一个 TableIterator 对象，表示堆表的尾迭代器。

2. TableIterator 类

TableIterator 类的主要目的是为 TableHeap 数据结构提供一个迭代器，使上层可以方便地遍历 TableHeap 中的元素。

(1)TableIterator 类的构造函数

```
TableIterator::TableIterator(TableHeap *table_heap, RowId rid, Transaction
* txn)
    : table_heap(table_heap), row(new Row(rid)), txn(txn) {
  if (rid.GetPageId() != INVALID_PAGE_ID)
    this->table_heap->GetTuple(row, txn);}
```

- 功能：初始化一个 TableIterator 对象。如果 RowId 是有效的，就会调用 table_heap 的 GetTuple 方法来获取元组。否则不进行任何操作。
- 输入：传入一个已有的 TableHeap 对象、RowId 对象和 Transactio 对象。

(2)TableIterator 类的拷贝构造函数

```
TableIterator::TableIterator(const TableIterator &other)
    : table_heap(other.table_heap), row(other.row ? new Row(*other.row) :
nullptr), txn(other.txn) {}
```

- 功能：创建一个新的 TableIterator 对象，并复制输入对象 other 的所有状态。
- 输入：传入一个已有的 TableHeap 对象。

(3)TableIterator 类的析构函数

```
TableIterator::~TableIterator() {}
```

- 功能：释放动态分配的 Row。

(4)TableIterator 类的重载运算符函数

- operator=(const TableIterator &other)：将一个 TableIterator 对象的状态复制到当前对象。
- operator==(const TableIterator &itr) const：比较两个 TableIterator 对象是否相等。如果两个对象的 RowId 相同，那么就返回真。

- operator!=(const TableIterator &itr) const：比较两个 TableIterator 对象是否不等。通过调用 operator==来实现。
- operator*()：返回当前 Row 的引用。在解引用之前，先检查当前迭代器是否处于有效状态。
- operator->()：返回当前 Row 的指针。在返回指针之前，先检查当前迭代器是否处于有效状态。
- operator++()：前置自增操作符，使迭代器指向下一个元素。如果 table_heap 或 row 为 nullptr，会抛出异常。该函数会首先获取包含当前行的页面，然后在当前页面或下一页面中查找下一行。如果找到下一行，就将 RowId 更新为下一行的 RowId。如果在当前页面找不到下一行并且没有下一页，就会抛出异常。最后，如果新的 RowId 指向的行存在于表堆中，就获取该行。
- operator++(int)：后置自增操作符，使迭代器指向下一个元素，并返回迭代器自增前的状态。

3. 可改进的地方

目前的代码在 row 中是通过 new 手动分配的，同时在析构函数和其他函数中通过 delete 释放。如果使用智能指针，则可以自动管理内存，降低出错的可能性。

同时，在 operator++中，若抛出异常，则当前页面的锁会一直保持，造成死锁。应在抛出异常前解锁并取消固定页面。

四. 单元测试

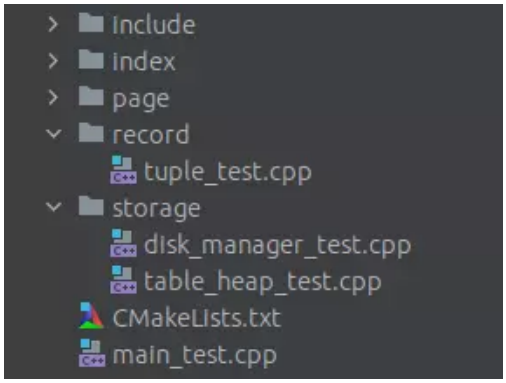
本模块需要进行两项测试，分别为 tuple_test 和 table_heap_test。

在 tuple_test 测试中包含了两个单元测试，分别对 Field 类和 Row 类进行了测试。这两个类是数据库系统中关于表数据存储的基本结构，通过数据读写和删除等基本操作进行了验证，确保数据存储和管理的正确性。

其中 FieldSerializeDeserializeTest 对 Field 类的序列化和反序列化进行了测试。该测试先通过 SerializeTo 函数将不同类型（整型、浮点型、字符型）的字段序列化到 buffer 中，然后再通过 DeserializeFrom 函数反序列化回 Field 对象。通过与原始字段进行比较，验证反序列化的正确性。同时，也对字段的比较函数进行了测试。

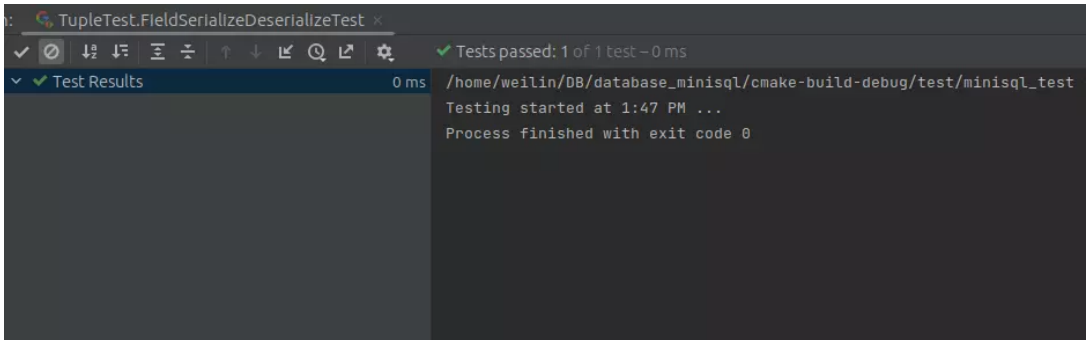
而 RowTest 对 Row 类进行了测试。首先，创建了一个带有三个字段（整型、字符型、浮点型）的 Row 对象，然后将其插入到 TablePage 中。然后，通过从 TablePage 中获取的 RowId 来读取并验证行的内容，确认读取的行与插入的行是相同的。最后，对该行进行了删除操作的测试，包括标记删除和应用删除。

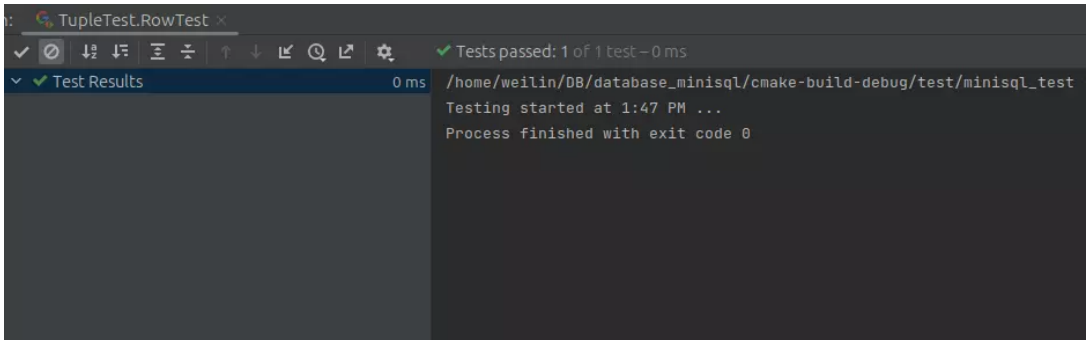
而在 table_heap_test 中测试了 TableHeap 的创建、元组的插入以及元组的获取功能，并通过对比插入和获取的记录来验证 TableHeap 的正确性。



1. tuple_test.cpp

该测试文件包含两个子测试，分别对 Field 和 Row 的序列化和反序列化进行测试。



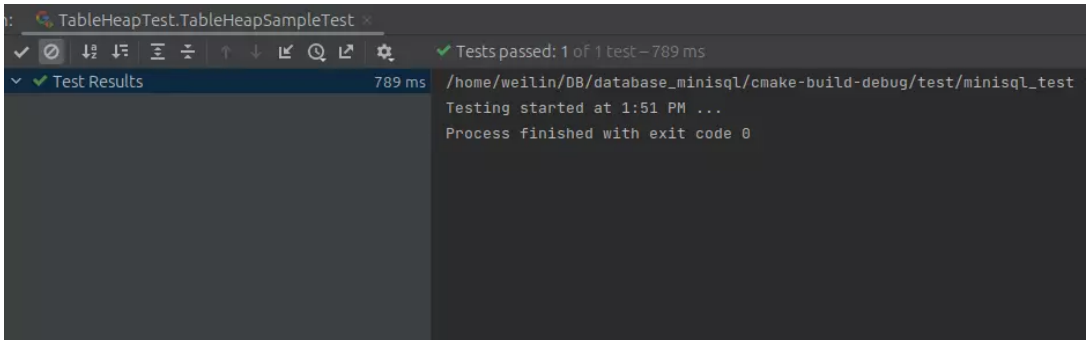


2. table_heap_test.cpp

该测试检验了 TableHeap 的创建、元组的插入以及元组的获取功能。在该测试中，通过控制插入的行数，能够对数据库的性能进行简单的分析。

当设置插入行数为 1 万行时，插入速度很快；但是在插入行数较多时，性能会出现显著下降，此时bufferpool已经全部占满并且数据也全部被固定，无法再分配新的数据页，

导致插入失败。此时可以修改默认缓冲池大小（如设置为 8192 KB）解决此问题。



此时可使用 Perf 工具进行性能热点分析，发现插入元组占据进程 95%以上的资源。