Arrow Database Compression

Final Project Presentation

By Abhijith, Arvind, Paulina, and Vilas

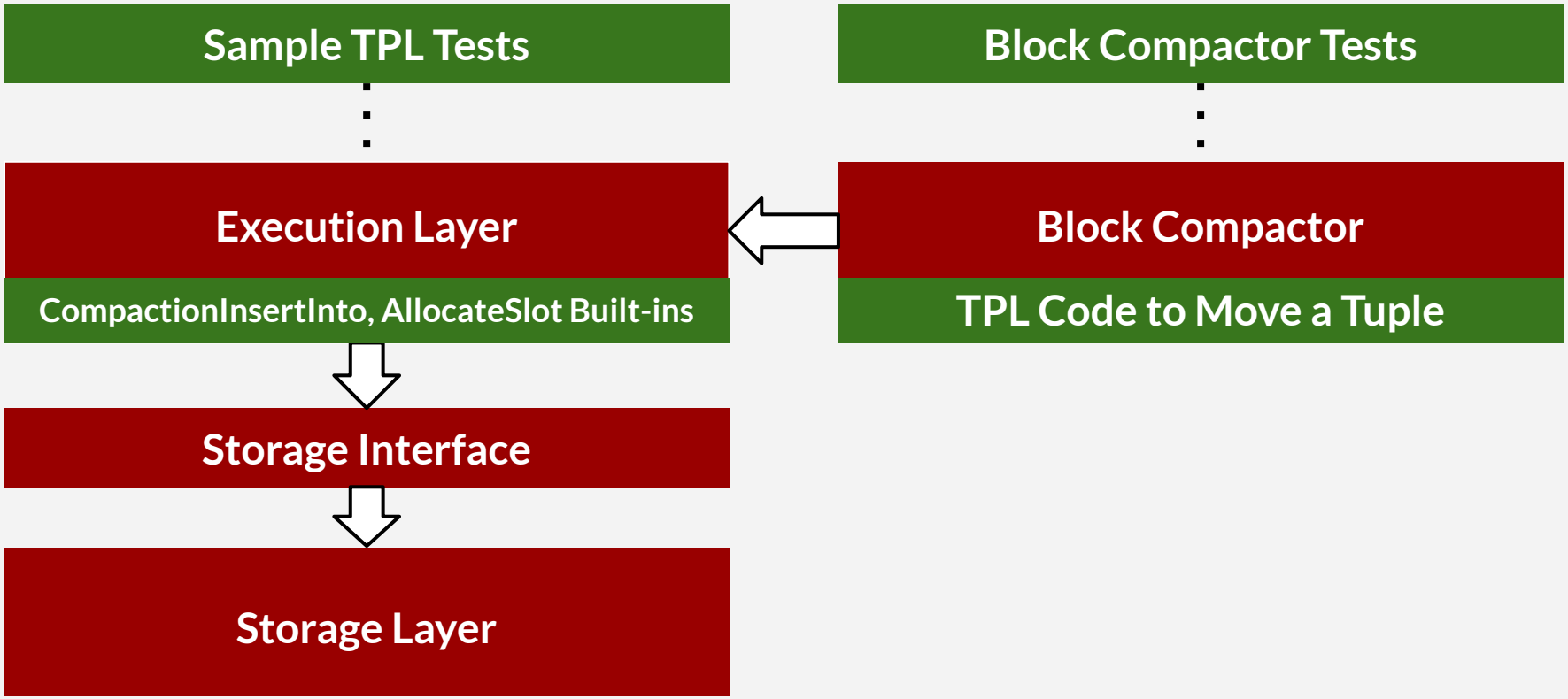
May 4th, 2020

Goals

Add support for:

- **75%** - Compacting blocks into Arrow format with updating indexes
- **100%** - Processing scan queries on compressed Arrow data
- **125%** - Identifying cold blocks to compact into Arrow

Progress


Sample TPL Tests
Block Compactor Tests
Execution Layer
Block Compactor
CompactionInsertInto, AllocateSlot Built-ins
TPL Code to Move a Tuple
Storage Interface
Storage Layer

Testing Correctness

Sample TPL Tests

Success:

- Verified CompactionInsertInto built-in

Difficulties:

- Reproducing the scenario required by “MoveTuple” in TPL required a new built-in (AllocateSlot)

Block Compactor Tests

Success:

- Linked database, table, catalog, block compactor etc. (to avoid direct interaction with data table)

Difficulties:

- “MoveTuple” built-in not compiling within the block_compactor.h file

Code Quality

- **Hacks:**

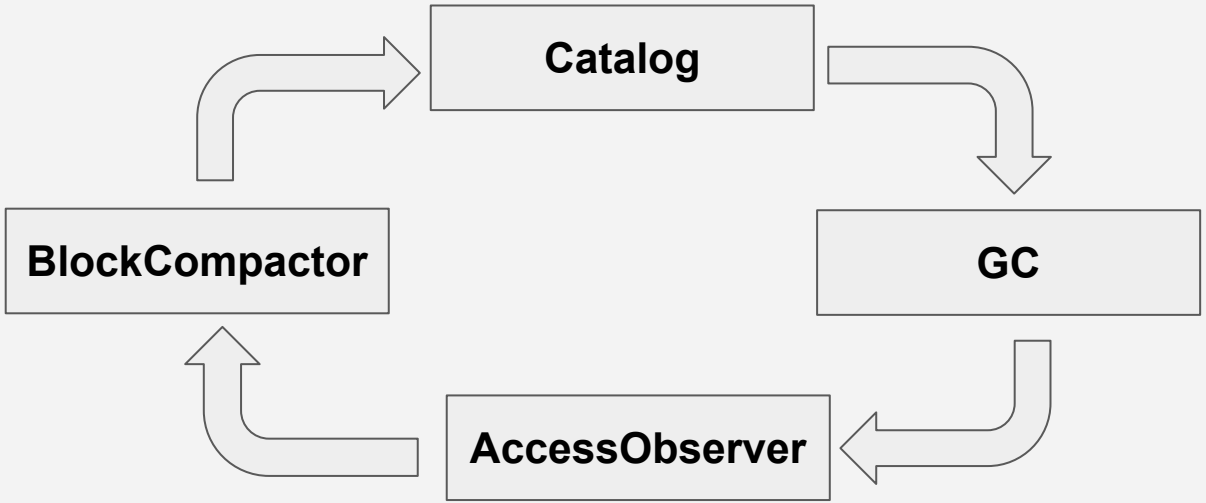
  Currently passing table name, col_ids, and an execution context to the block_compactor in the test. Need to use the catalog.

- **Incomplete code:**

  Wrote TPL code that moves a tuple from original slot to a newly allocated slot. Need to use codegen for generating that TPL code.

Design Difficulties (Dependencies)

- Execution Context inside the BlockCompactor introduces a Circular Dependency



Design Difficulties (Threading)

Block Compaction running as a background thread

Options:

- Standard C++ threads
- Thread pool using TBB
- C++ coroutines

What We've Learned

- Preventing direct interactions with the data table and importance of access and moving data via the execution layer

- Importance of using an internal transaction for performing compaction

- Writing and understanding TPL code, CodeGen, Translators

Learning TPL

TPL arguments: using class variables vs. using function parameters

terrier/src/include/execution/sql/storage_interface.h, .cpp

```
  /**
   * The redo record.
   */
  storage::RedoRecord *table_redo_{nullptr};
```

```
storage::TupleSlot StorageInterface::TableInsert() {
  exec_ctx_->RowsAffected()++;
  return table_->Insert(exec_ctx_->GetTxn(), table_redo_);
}
```

terrier/sample_tpl/insert.tpl

Getter
Setter
Insert

```
// Insert into empty_table
var insert_pr = @getTablePR(&inserter)
@prSetInt(insert_pr, 0, colA)
var insert_slot = @tableInsert(&inserter)
```

Learning TPL

TPL arguments: using class variables vs. using function parameters

terrier/src/include/execution/sql/storage_interface.h, .cpp

```
  /**
   * The redo record.
   */
  storage::RedoRecord *table_redo_{nullptr};
```

```
bool StorageInterface::TableUpdate(storage::TupleSlot table_tuple_slot) {
  exec_ctx_->RowsAffected()++;
  table_redo_->SetTupleSlot(table_tuple_slot);
  return table_->Update(exec_ctx_->GetTxn(), table_redo_);
}
```

Getter
Setters
Update

terrier/sample_tpl/update-2.tpl

```
// Update
var update_pr = @getTablePR(&updater)
@prSetInt(&update_pr, 0, cola)
@prSetInt(&update_pr, 1, @intToSql(500))
if (!@tableUpdate(&updater, &slot)) {
```

Our Sample TPL

```
22    for (@indexIteratorScanAscending(&index, 0, 0); @indexIteratorAdvance(&index);) {
23      // Materialize the current match.
24      var table_pr = @indexIteratorGetTablePR(&index)
25      var colA = @prGetInt(table_pr, 0)
26      var slot = @indexIteratorGetSlot(&index)
27
28      // Insert into empty_table
29      var insert_pr = @getTablePR(&inserter)
30      @prSetInt(insert_pr, 0, colA)
31      var insert_slot = @tableAllocateSlot(&inserter)
32      // Insert into the newly allocated slot
33      @tableCompactionInsertInto(&inserter, &insert_slot)
34
35      // Insert into index
36      var index_pr = @getIndexPRBind(&inserter, "index_empty")
37      @prSetInt(index_pr, 0, colA)
38      if (!@indexInsert(&inserter)) {
39        @indexIteratorFree(&index)
40        @storageInterfaceFree(&inserter)
41        return 37
42      }
43      count = count + 1
44    }
45    @indexIteratorFree(&index)
46    @storageInterfaceFree(&inserter)
47    return count
```

terrier/sample_tpl/compaction-insert-into.tpl

Future Work

- **75%** - Compacting blocks into Arrow format with updating indexes
  - Create a compaction_translator in compiler → internal folder
  - Add code for generating compaction TPL in the compaction_translator

Future Work

- **100%** - Processing scan queries on compressed Arrow data

  Add (accessible to the execution layer):

  - An indicator variable for whether a tuple's block is compacted (or access to a block's temperature)
  - A shared pointer to the dictionary of a tuple's block
  - Decompressing and materializing scanned tuples → directly work on compressed data
  - Comparators for compressed data using block dictionaries
  - Benchmarking performance and identifying optimization opportunities

Future Work

- **125%** - Identifying cold blocks to compact into Arrow
  - Utilize access statistics that may already be collected
  - Analyze performance of compaction (% successful compactions, memory usage):
    - Across workloads with different scan characteristics
    - Across a range of frequencies of when compaction occurs
    - Across different tuple types (number of columns, column types)
  - Analyze cause of non-successful compactions to identify design and optimization opportunities

Future Work

- **Extra Thoughts**
  - Currently block compaction queue only has one block. Add testing and implementation of multi-block queues.
  - Currently one internal transaction is processing the whole block compaction queue, so if even one block is accessed, the transaction rolls back. Consider other designs for the queue and performance constraints.
  - Test to make sure that any changes do not block other transactions.

Thank you to…

Andy

Prashanth

Matt

Tianyu

Terrier