Josephine Lee
Dec 2024

For my project, I modified GoDB to use a column-oriented physical layout. I used the “early materialization” design: making complete tuples while reading columns off disk and processing them with the existing query operators. Using a column-oriented physical layout helps speed up query processing in GoDB, and allows for data compression for further potential processing optimization.

To do this, I made a new DBFile interface that supports column-oriented files, making methods like insertTuple, deleteTuple, LoadFromCSV, flushPage, readPage, Iterator, etc. The iterator takes in a list of columns to read and output the tuples that have records from those columns, using BufferPool.GetPage. Similar to how there are heap page and heap file go files, I made one go file for column pages (column_store_page.go) and one go file for the column file (column_store_file.go).

Here are some key features of my implementation. I will explain the two methods that took me the most time: inserting tuples (insertTuple) and the iterator (Iterator) in the column file (column_store_file.go):

- The Iterator function in the columnStoreFile returns an iterator function to retrieve tuples across all columns of the file. It first constructs a list of all column indices (columns) and calls the IteratorCol helper function, which handles iterating across the specified columns. Within IteratorCol, another helper function, initColumnPagesAndIterators, initializes pages and their respective iterators for the specified columns using the buffer pool (bufPool) to retrieve pages. The final iterator function processes tuples page by page, combining data from each column using another helper function, joinTuples.
- The insertTuple function in columnStoreFile inserts a tuple into the columnar file, ensuring it fits into the structure across all columns. It begins by iterating over existing pages in the primary column (column j = 0), attempting to insert the tuple into a page using the tryInsertIntoPage helper function. This function uses the buffer pool (bufPool) to retrieve the relevant page, tries to insert the tuple, assigns a record ID, and then inserts corresponding column values into other column pages using the insertIntoColumnPage helper.
    - If all existing pages are full, insertTuple calls the createNewPagesAndInsert helper to create new pages for all columns. This function locks the file, appends new pages to each column, and inserts the tuple into the primary column first, followed by other columns.

These are the test cases (column_store_test.go):

- Tests for column file methods: TestcolumnStoreFileCreateAndInsert, TestcolumnStoreFileDelete, TestcolumnStoreFilePageKey, TestcolumnStoreFileSize, TestcolumnStoreFileDirtyBit
- Tests for column page methods: TestColumnPageInsert, TestColumnPageDelete, TestColumnPageInsertTuple, TestColumnPageDeleteTuple, TestColumnPageDirty, TestColumnPageSerialization
- Tests for GoDB operations: TestIntFilterCol, TestStringFilterCol, TestJoinCol, TestProjectCol
- Performance tests: TestLoadCSVPerformance50, TestLoadCSVPerformance500, TestLoadCSVPerformance2000, TestLoadCSVPerformance10000. These evaluate the performance gain of using column store.

I feel that these are sufficient to demonstrate that my implementation works as expected because the test cases show that my implementation of column store can handle the same basic use cases, operations, and queries that the row-oriented implementation can.

To evaluate the performance gain, I made CSV tests (one CSV has 50 rows, one has 500 rows, one has 2000 rows, and one has 10,000 rows) where I iterated through the CSV and reported how much time it took vs. the original heap file/heap page methods. This measures how much faster the column-oriented file iteration is compared to the row-oriented (heap) iteration. At the start of the test cases, using os.Remove and making a new buffer pool each time ensures that the cache is controlled. I also reset the file pointer for the heap store test within each performance test.

I ran these performance tests 10 times. Here are the results. There is close to a 10x improvement for the two larger CSVs.

Test name	**Average time (microseconds) for column**	**Average time (microseconds) for heap**
TestLoadCSVPerformance50	9	18
TestLoadCSVPerformance500	29	102
TestLoadCSVPerformance2000	309	449
TestLoadCSVPerformance10000	3865	35516
TestLoadCSVPerformance20000	8074	76559

Inserting tuples and the iterator in the column file required the most work and debugging; getting an accurate picture of what the column file looked like at each step required thought and debugging work. Given more time, I would extend my implementation by using a late materialization design. This would result in less data flowing through memory which would be a great improvement.

I have zipped all the code with this writeup (let me know if there are any issues).