50.043 Database Systems and Big Data

Lab 2 - Spring 2024

Group Members

Name	Student ID
Atul Parida	1006184
Chandrasekar Akash	1006228
Jignesh Motwani	1006178

Summary of Code & Design Decisions

Exercise 1: Predicate, JoinPredicate, Filter and Join

Predicate and *JoinPredicate* is implemented to compare fields of tuples. *Filter* is an operator that filters tuples based on a given predicate. *Join* implements the relational join operation. The code implemented for the join operation uses a ***simple nested loop*** comparing the tuples one by one.

Exercise 2: IntegerAggregator, StringAggregator and Aggregate

The *IntegerAggregator*, *StringAggregator*, and *Aggregate* Java files implement the functionality to compute aggregate values over a set of tuples in a database management system, supporting operations like *MIN*, *MAX*, *SUM*, *AVG*, and *COUNT*. The *IntegerAggregator* and *StringAggregator* classes specifically handle aggregation for integer and string fields respectively, maintaining counts and sums where necessary, especially for averaging. They differentiate behavior based on the presence of a group-by field, accumulating results in a hash map. The *Aggregate* operator class manages these aggregators, initializing them based on the type of the aggregation field and the presence of a group-by field, orchestrating the process of feeding tuples into the aggregator, and providing an iterator to access the aggregated results. These designs reflect a modular approach to handling SQL-like aggregation queries, emphasizing separation of concerns and the encapsulation of aggregation logic within specialized classes.

Exercise 3: HeapPage and HeapFile

In implementing *insertTuple* and *deleteTuple* across both *HeapPage.java* and *HeapFile.java*, the design prioritises efficient data manipulation within a database's heap file structure, embodying principles of space management and transactional integrity. For *insertTuple*, *HeapFile.java* scans through existing pages to find available space, leveraging *HeapPage.java*'s capacity to insert a tuple into the found space, and creating a new page if necessary, thereby ensuring data is compactly stored and dynamically scalable with data insertion demands. Concurrently, *deleteTuple* locates the specific tuple within pages, utilizing *HeapPage.java* to remove the tuple and update the slot status, illustrating a design that supports data mutability and integrity. Both methods highlight a layered architecture where *HeapFile* manages page-level operations, directing specific tuple manipulations to *HeapPage*, which directly manipulates byte arrays representing page data. This architecture not only encapsulates file and page operations within their respective classes but also ensures that modifications are transaction-safe, leveraging the buffer pool's locking mechanism to maintain data consistency and support concurrent database operations.

Exercise 4: Insert and Delete

The *Insert* class inserts tuples read from a child operator into a specified table. It utilizes a transaction ID, a child operator, and a table ID. The class ensures that the *TupleDesc* of the child matches the table's *TupleDesc* before proceeding. It opens and closes the child operator appropriately, rewinding it as necessary. The core functionality lies in the *fetchNext* method, which iterates through the child's tuples, inserts them into the table using the *BufferPool*, and returns a one-field tuple containing the number of inserted records. The class follows typical operator conventions with methods for getting and setting children.

The *Delete* class is responsible for removing tuples from a table. It reads tuples from its child operator and deletes them from the table they belong to. The class utilizes a transaction ID and a child operator for its operation. Key methods include opening, closing, and rewinding the operator, as well as fetching next tuples for deletion. Deletions are processed through the buffer pool via *Database.getBufferPool()* method. The class follows typical operator conventions with methods for getting and setting children.

Exercise 5: BufferPool

The *BufferPool* class manages the reading and writing of pages into memory from disk in a thread-safe manner. It caches up to a specified number of pages and handles page retrieval, eviction, locking, and flushing. Key methods include *getPage*, *insertTuple*, *deleteTuple*, *flushAllPages*, *discardPage*, and *evictPage*. The class maintains a *ConcurrentHashMap* to store pages, ensuring efficient and concurrent access.

Incomplete Code Elements

All code elements due for Lab 1 and 2 have been completed.

The implementations of the remaining code elements will be done in the coming lab submissions.