COL362/COL632 Assignment 3

B$^+$Tree and Index Scan

Due: 22nd March 2024 7:00pm

Modified on: 16th March 2024

Goal

The goal of this assignment is to get familiar with

- Physical storage of database files on disk
- Indexing databases files, particularly using B+Trees
- Implementing a new operator using Apache Calcite, and using it to evaluate simple queries

You have been provided with a starter code for this assignment. The code is available here (You should be logged in with your IITD credentials to access it). In this assignment, you will further develop the provided code.

Background

Apache Calcite. We will be using Apache Calcite, an open-source extensible data management framework. It contains a SQL parser, an API for building expressions in relational algebra, and a query optimizer. It is used in many popular big data systems, such as Apache Flink, Apache Hive, Apache Drill, Apache Kylin, and is integral to data systems technology stack across several companies. See companies and projects that are powered by Calcite. You are strongly encouraged to check out more information and get yourself familiar with Apache Calcite here. We will be using Calcite in more detail in Assignments 4 and 5.

Memory management. Managing data storage is a critical part of any data management system. The system must manage the memory efficiently, and must also provide a way to index the data for faster access.

In this assignment, you will develop an index and an efficient access control method to access data efficiently.

Tasks

Broadly, You have to solve two tasks in this assignment.

Task 1. Implement a B+ Tree

In this task, you will implement B+Trees. The B+Tree is a data structure used to store and retrieve data in a database. It is a balanced tree, and is particularly useful for efficiently accessing records, and efficiently performing insertions, deletions, and updates on files stored on disks.

You job is to implement a B+Tree that supports (point) insert and search operations. Feel free to flex your programming and hacking skills by also implementing deletions (see bonus section below). The B+Tree you develop, will be used to index CSV files. There are some example files provided to you in the starter code.

Task 2. Implement IndexScan in Calcite

After you have completed task 1, we'd love to see your B+Tree in action, i.e, given simple `SELECT-FROM-WHERE` query, we'd like to evaluate the query and return the rows in the CSV file which satisfy the filter condition in the `WHERE` clause.

The code provided to you already does most of the ground work for you. In particular, it

- Parses the query and convert it into a relational algebra expression
- Converts the relational algebra expression into a logical plan
- Converts the logical plan into a physical plan
- Executes the physical plan and return the result

The only missing piece the above steps, is the PIndexScan operator that you have to implement. This operator will be used to evaluate queries which have a WHERE clause and the column in the WHERE clause is indexed.

Note: For this assignment, we will have simple SFW queries. The queries will always of the type -

"`SELECT * FROM $table_name WHERE $col_name OP $value`"

where the `OP` can be `EQUALS, LESS_THAN, GREATER_THAN, GREATER_THAN_OR_EQUAL, LESS_THAN_OR_EQUAL`.

DB362

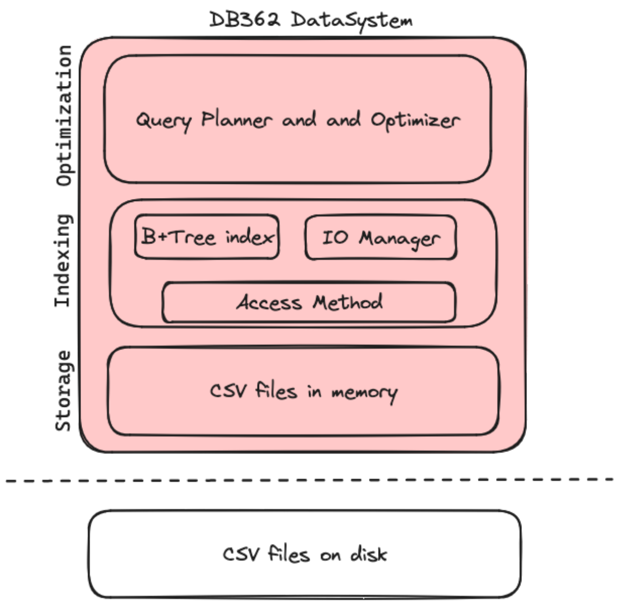
`DB362` is a data system that we've developed, and need your help to further develop it. The system is being designed to efficiently and effectively query data stored in CSV files. `DB362` is designed as an in-memory system, i.e., it first loads all files from disk to memory, and then provides the data management framework to efficiently query CSV files–one can easily query a CSV file as "`select ... from CSVfile where ...`". The following figure shows the high level overview the "three layers" of the system.

Storage

The storage layer is already developed! `DB362` system organizes CSV files in memory in a similar way a database organizes the database files on disk. In the code provided to you, the classes `DB`, `File`, and `Block` contain important parts of the code for storage layer (You should NOT modify these). The system organizes the in-memory database (of CSV files) as a collection of `File` objects, each of which is a collection of `Block` objects, and each `Block` contains records (rows of a CSV file.)

The following explains how the `DB362` system organizes data in memory.

The structure of each block is as follows



s

Figure 1: The `DB362` data system.

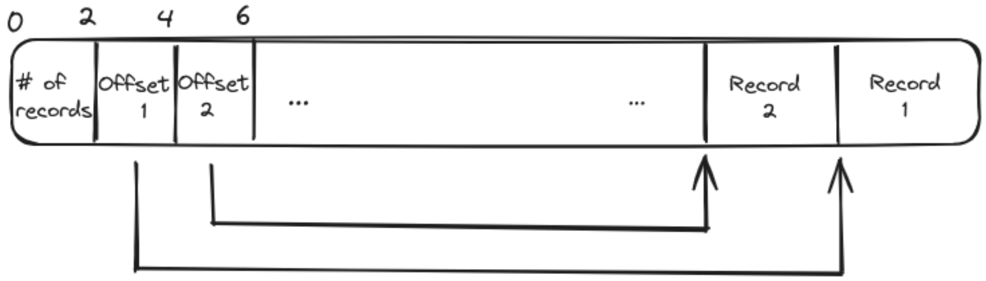


Figure 2: Block Structure for Records

A block is of always of fixed size of 4KB. When used to store rows of a CSV file, we follow the convention as shown in the figure. The first 2 bytes represent the number of rows, say n. The next $2n$ bytes store the offset of each row in this block (2 bytes for each offset). We start filling the rows from the right hand side.

The structure of each record is as follows

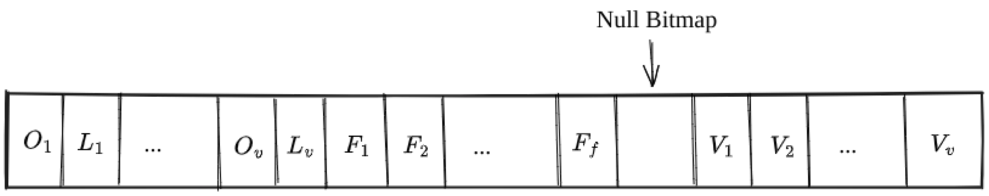


Figure 3: Record Structure

Here, O_i, L_i are the offset and length for the ith variable-length field. v is the number of variable length fields and f is the number of fixed length fields. Each O_i is 2 bytes each, same for L_i. For more details, refer to the lecture notes.

Note that in each file that is used to store rows of a CSV file, the first block is always the metadata block. In this case, we use the block to store the schema of the CSV file. The metadata block looks like -

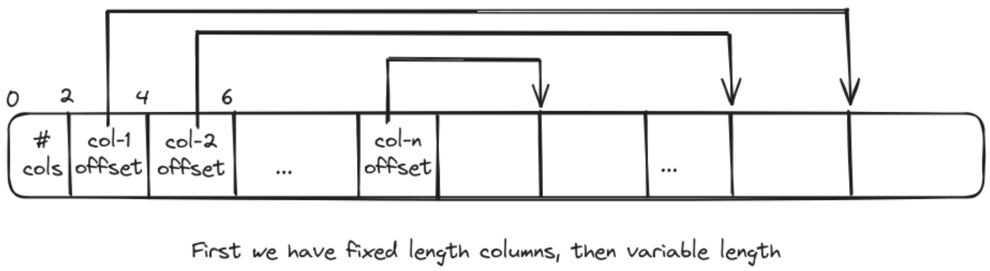


Figure 4: Schema Block Structure

For each column, we store the datatype of the column, and the column name. This looks like -

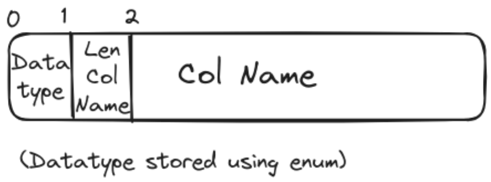


Figure 5: Each column data in schema

Note: Since we store everything as bytes, take care of the little endian and big endian encoding at different places.

Index

For this assignment, you will develop the Indexing layer, in particular a B+Tree index, and an access method using the B+Tree (the two tasks above). A skeleton code is already been provided to you. The B+Tree is implemented using the class `Tree`. The `Tree` class contains the `InternalNode` and `LeafNode` classes.

Note that the Tree class extends the AbstractFile Class. This is because, we also store the B+ Tree as a file on the (simulated) in-memory disk. The InternalNode and LeafNode classes also extend the AbstractBlock class.

Remember that - since the InternalNode and LeafNode are blocks in the BPlusTreeIndexFile file - the identifier (ID) of a node is the index of the block in the file.

Some methods are already implemented in these classes. **You should NOT modify them**. You should implement the insert and search methods.

The first block in the Tree file will be the metadata block. In this block, the first 2 bytes are reserved for the order of the block, and the next 2 bytes are reserved for the index of the root node.

The structure of InternalNode and LeafNode are different. Each InternalNode looks as follows -

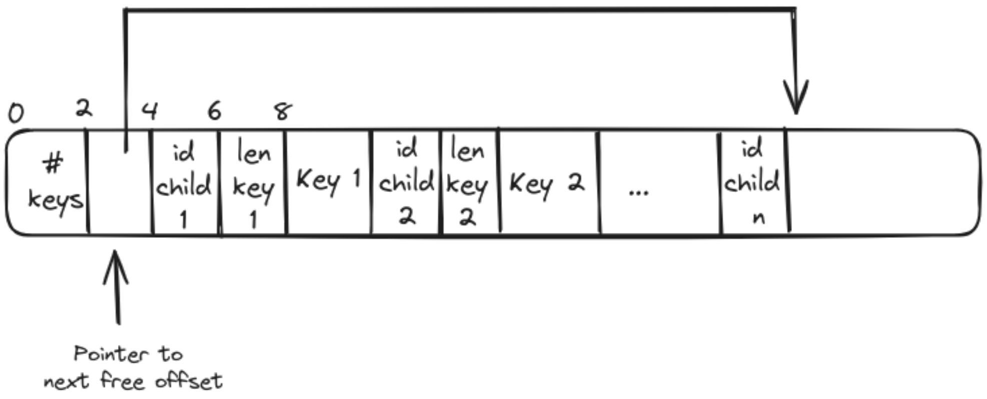


Figure 6: Internal Node

Whereas, each LeafNode looks as follows -

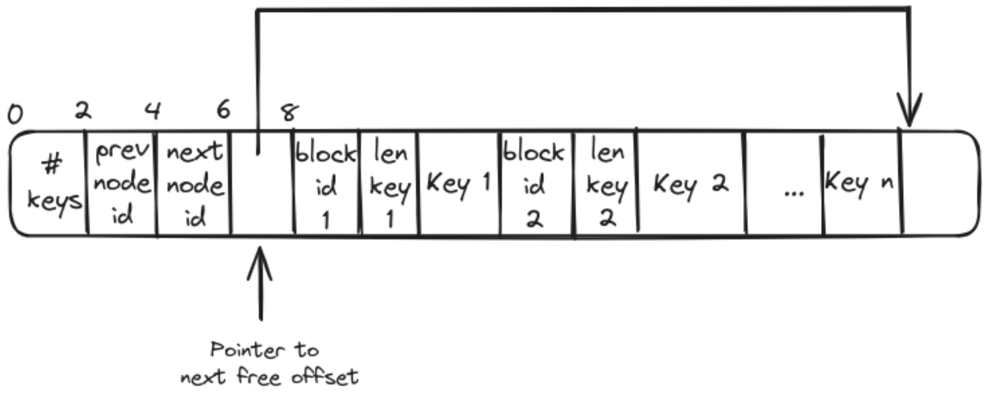


Figure 7: Leaf Node

Here, you need to implement the search and insert function in BPlusTreeIndexFile.java. The search and insert functions in LeafNode.java and InternalNode.java can be used as helper functions and will NOT be tested separately.

The search function in BPlusTreeIndexFile should return the leftmost ID of leafnode that contains the key.

StorageManager

The StorageManager class helps in managing the database. It provides methods to load csv files an in-memory storage, and to serialize and deserialize the rows into series of bytes.

The StorageManager class also provides methods to create indexes on columns of tables. **You will need to implement the create_index method.**

Calcite

As a part of the Apache Calcite framework, the starter code has 3 directories in the optimizer package -

- rel - This directory contains the classes for the relational algebra expressions. For this assignment, we only have classes PRel and PIndexScan. PRel is the base class for all the relational algebra expressions. PIndexScan is the class for the IndexScan operator.
- convention - PConvention is the physical convention for the relational algebra expressions.
- rules - This directory contains the rules for converting the logical plan into a physical plan. We have already implemented the rule for converting a logical plan (which matches a SFW query) into a physical plan. Once converted, we have a physical IndexScan operator in the plan.

As an index method, you will only need to implement the evaluate function in the PIndexScan class. You will not need to modify any other part of the code in the Calcite package.

What to submit?

`DB362` system requires Java 8. Ensure that you have java version 8 before proceeding with developing the assignment. Further, you would also require Gradle version 4.5. Then proceed as follows:

- Clone the project from `https://git.iitd.ac.in/dbsys/assignment_3.git`
  - create directory `path/to/assignment_3/`
  - cd into the newly created directory by `cd path/to/assignment_3/`
  - run `git clone https://git.iitd.ac.in/dbsys/assignment_3.git .` to clone the project on your local machine
- Import the project into your favorite editor. We strongly recommend using Intellj
- Develop the system further. **You should only work on the following files**
  - InternalNode.java
  - LeafNode.java
  - ~~Tree.java~~ BPlusTreeIndexFile.java
  - PIndexScan.java
  - StorageManager.java
  - TreeNode.java
- Test that your code works
  - You can add your own test cases in the files placed in "in/ac/iitd/src/test/java" directory.
  - To add new test cases, follow similar syntax as already included ones. (Should include a "@Test" annotation before the test function)
  - To run the test cases, run the command `./gradlew test` in the `/path/to/assignment_3` directory. You can also use `./gradlew test --info` to get detailed output on your console. (You can also setup these run commands in Intellij IDE).
- Submit your contribution
  - `cd` into `path/to/assignment_3/`
  - create a patch `git diff [COMMITID] > [ENTRYNO].patch`
  - Submit the `.patch` file on Moodle

Please follow the instructions strictly as given here and as comments in the code. Do not rename any files, modify function signatures, or include any other file unless asked for.

When submitting your patch:

- replace `[ENTRYNO]` with your entry number.
- replace `[COMMITID]` with 0fd98a56ee9534b9360ce87929612669d2cfac4b

Ensure that your patch doesn't contain any files other than those you can make changes in. Thus, if you create any new files for test cases, you should remove them from your patch.

Bonus Part

As a part of this assignment, you are free to also implement the delete functionality for your B+ Tree implementations. Note that this part is optional and will carry some bonus marks. The signature for the delete function is provided in ~~Tree.java~~ BPlusTreeIndexFile.java and StorageManager.java. As before, you should NOT modify this signature. If you choose to not implement this function, please do not remove the function signature from the ~~Tree.java~~ BPlusTreeIndexFile.java file - just leave the implementation empty.