SC3020 Database System Principles

Project 1

Group Members:

Name
Jadhav Chaitanya Dhananjay
Sethi Aryan
Gupta Tushar Sandeep
Run Yu

Link to repository:
https://github.com/TushG29/SC3020_DatabaseSystemPrinciplesProject1

Record Component

As we were presented with the task of storing a relatively large data set, we explored various means by which we could reduce the size of a Record stored on a disk block. A smaller Record structure enables more nodes to be stored on a disk block, minimizing the potential wastage of memory space in our unspanned storage system. Having more records per block also reduces the number of disk I/O reads for ranged queries in our contiguous storage solution.

The tradeoff for optimizing Record storage is that more computational work is required to convert data stored on a disk to a usable structure. Our team, however, believes that the benefits outweigh the costs in this case. With that in mind, the following are the steps we took to optimize the storage of a Record:

1. **Storing FG_PCT_home, FT_PCT_home, and FG3_PCT_home as float**

As we only need to store values up to three decimal places, we use the smallest data type available in C++ for doing so.

2. **Storing the number of days between 01/01/200 and GAME_DATE as int**

Storing GAME_DATE as a string or date object in C++ would take up a lot of memory. We decided therefore to write a helper function dateToOffset() available in the Record class to return the number of days from 01/01/2000 to the GAME_DATE value read from each record.

```cpp
/**
 * @brief Function that converts date string found in data to an integer offset from 01/01/2000 for storage
 *
 * @param dateString Input string for conversion
 * @return int Date offset from 01/01/2000
 */
int Record::dateToOffset(const string &dateString)
{
    // Set starting date 01/01/2000
    chrono::system_clock::time_point startingDate = chrono::system_clock::from_time_t(946684800); // UNIX timestamp for January 1, 2000

    // Parse input date string with the format given in the data file
    tm tmDate = {};
    istringstream dateStream(dateString);
    dateStream >> get_time(&tmDate, "%d/%m/%Y");
    chrono::system_clock::time_point inputDate = chrono::system_clock::from_time_t(mktime(&tmDate));

    // Calculate the date offset from starting date
    days dateOffset = chrono::duration_cast<days>(inputDate - startingDate);
    int daysSinceEpoch = dateOffset.count();
    return daysSinceEpoch;
}
```

Consequently, we provide a helper function offsetToDate() that calculates and returns the original string value stored in games.txt. This function can be used when displaying information about records or for further computation.

```cpp
/**
 * @brief Function to convert an offset to a date string for displaying information
 *
 * @param offsetInDays Integer value stored in database
 * @return string Date string used for display
 */
string Record::offsetToDate(int offsetInDays)
{
    // Set starting date 01/01/2000
    chrono::system_clock::time_point startingDate = chrono::system_clock::from_time_t(946684800); // UNIX timestamp for January 1, 2000

    // Calculate date from offset in database
    chrono::system_clock::time_point calculatedDate = startingDate + days(offsetInDays);

    // Convert date to a string with same format as input file
    time_t calculatedTime = chrono::system_clock::to_time_t(calculatedDate);
    tm tmDate = *localtime(&calculatedTime);
    ostringstream dateStream;
    dateStream << put_time(&tmDate, "%d/%m/%Y");
    return dateStream.str();
}
```

3. **Storing the offset of TEAM_ID as uint8_t**

Further analysis of our data revealed that the range of values for TEAM_ID is between 1610612737 and 1610612766. By calculating the offset of the TEAM_ID from the minimum value, we can save three bytes of memory per record by storing the offset as a uint8_t (1 byte) instead of an integer (4 bytes).

The function teamIDToOffset() in the Record class helps to perform this computation. The value 1610612736 is used for subtraction to ensure that teams with TEAM_ID 161061237 have an offset of 1.

```cpp
/**
 * @brief Function that converts team id to a offset from the lowest team ID found in the dataset for storage
 *
 * @param teamID read from file
 * @return uint8_t
 */
uint8_t Record::teamIDToOffset(int fileTeamID)
{
    int difference = fileTeamID - 1610612736;
    uint8_t offset = difference;
    return (offset);
}
```

Similarly, the Record class also provides a function to convert the offset stored in the database to a readable TEAM_ID for displaying information.

```cpp
/**
 * @brief Function that converts team id offset to integer value for displaying
 *
 * @param offset read from database
 * @return int
 */
int Record::offsetToTeamID(uint8_t offset)
{
    int difference = static_cast<int>(offset);
    return (1610612736 + difference);
}
```

4. **Storing PTS_home, AST_home and REB_home as uint8_t**

Similar analysis on these columns revealed that the range of values of these columns in games.txt correspond to the range of the uint8_t data type. This enables us to save 9 bytes of memory space in total across these three fields.

Other variables stored in a Record include:

1. **blockAddress** - base address of the database block that a Record is stored in. This is assigned when a Record is pushed to the database.
2. **offset** - offset of the record from blockAddress. Similarly assigned when a Record is pushed to the database.
3. **homeTeamWins** - corresponds to the value of HOME_TEAM_WINS read from games.txt

The resulting variables, data types, size and byte offset of the Record structure can be found in Table 1 below. The original size of the Record is 35 bytes. The C++ compiler adds 5 bytes for each defined Record structure to maintain memory padding.

Variable	Data Type	Size	Offset
fgPct	float	4 bytes	0
ftPct	float	4 bytes	4
fg3Pct	float	4 bytes	8
gameDate	int	4 bytes	12
blockAddress	uchar *	8 bytes	16
offset	int	4 bytes	24
recordID	unsigned short int	2 bytes	28
teamID	uint8_t	1 byte	30

pts	uint8_t	1 byte	31
ast	uint8_t	1 byte	32
reb	uint8_t	1 byte	33
homeTeamWins	bool	1 byte	34
C++ Compiler Padding	-	5 bytes	-
Total Record Size:		40 bytes	

The following Record constructor is used when reading values from the database:

```
Record::Record(float fgPercentage, float ftPercentage, float fg3Percentage, int date, uchar* blockAddress, int offset,
    unsigned short int recordID, uint8_t teamID, uint8_t points, uint8_t assists, uint8_t rebounds, bool homeTeamWins)
    : recordID(recordID), gameDate(date), teamID(teamID), pts(points), ast(assists), reb(rebounds),
      fgPct(fgPercentage), ftPct(ftPercentage), fg3Pct(fg3Percentage), homeTeamWins(homeTeamWins), offset(offset),
      blockAddress(blockAddress){};
```

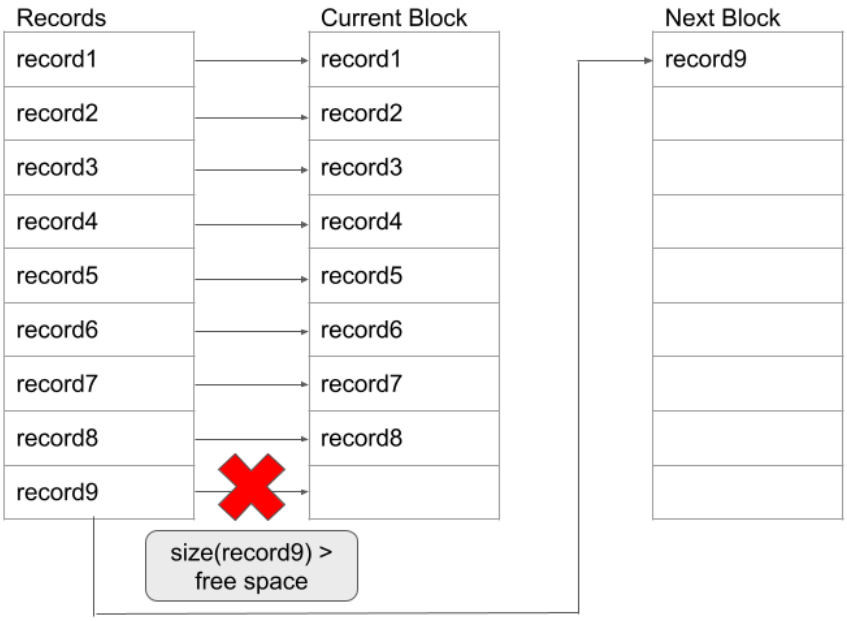
The following Record constructor is used while reading values from games.txt, using the helper functions defined earlier to instantiate the object.

```
Record::Record(unsigned short int recordID, string gameDateStr, uint8_t teamID, uint8_t pts, uint8_t reb, uint8_t ast,
float fgPct, float ftPct, float fg3Pct, int homeTeamWins, uchar* blockAddress, unsigned short int offset)
{
    this->fgPct = fgPct;
    this->ftPct = ftPct;
    this->fg3Pct = fg3Pct;
    int gameDateOffset = dateToOffset(gameDateStr);
    this->gameDate = gameDateOffset;
    this->blockAddress = blockAddress;
    this->recordID = recordID;
    this->offset = offset;
    this->teamID = teamID;
    this->pts = pts;
    this->reb = reb;
    this->ast = ast;
    bool homeTeamWinsBoolean = winsToBool(homeTeamWins);
    this->homeTeamWins = homeTeamWinsBoolean;
}
```

Storage Component Design

Unspanned Record System:

For the purposes of this project, we chose to implement an **unspanned** record storage system. Therefore, our database allocates a new block for storing a record if there is insufficient space in the current block. This is illustrated in Figure 1 below.



Traditionally, the unspanned record system comes with the downside of memory wastage in each block. However, as we have already taken measures to optimize the storage of a Record, the downsides are not as exacerbated in our current implementation.

We focus instead on the benefits of unspanned implementation - each record will always take only one disk I/O to read from the block. By eliminating the case where a record is spanned between two blocks, we reduce the operational complexity and ensure a more consistent level of performance. Given that disk I/O is the bottleneck for most systems and a historical dataset such as this would mostly be used for querying, unspanned storage is a good choice for our needs.

Contiguous Data Storage

In order to speed up insertion, deletion and retrieval queries in the experiments for this project, we sorted Records based on ascending order of FG_PCT_home before inserting them into the database. Maintaining a contiguous set of data also enables us to perform bulk loading when constructing the B+ tree on our Records.

```
Storage::Storage(uint diskCapacity, uint blockSize)
{
    this->diskCapacity = diskCapacity;
    this->blockSize = blockSize;
    this->currentBlock = 0;
    this->currentBlockSize = 0;
    this->availableBlocks = diskCapacity / blockSize;
    this->baseAddress = static_cast<uchar *>(malloc(size: diskCapacity));
    this->databaseCursor = baseAddress;
    this->blockRecords[0] = 0;
    this->recordsStored = 0;
}
```

Storage Class Constructor:

a) The Storage class constructor allocates memory for the database based on diskCapacity using malloc.
b) It also sets the initial values for several variables like the number of available blocks.

Class Attributes:

1. **diskCapacity**: disk capacity of database in bytes.
2. **blockSize**: size of a block in bytes (initialized to 400 for this project).
3. **currentBlock**: internal variable that keeps track of the current block that records are being inserted into.
4. **currentBlockSize**: internal variable that keeps track of the size of the current block that is being inserted into. Used during insertion to implement our unspanned storage system.
5. **availableBlocks**: initialized to the floor division of diskCapacity and blockSize. Used during insertion to check if there is space for new records.
6. **blockRecords**: unordered_map structure that tracks the number of records stored in

```
bool Storage::deleteBlock(int blockID)
{
    uchar *cursor = baseAddress;
    // Navigate to the first memory address of a block
    cursor += (blockID * blockSize);
    uchar zeroBytes[blockSize];             // Create an array of 400 null bytes
    memset(b: zeroBytes, c: 0, len: sizeof(zeroBytes)); // Initialize it to zeros
    if (memcpy(dst: cursor, src: zeroBytes, n: sizeof(zeroBytes)))
    {
        this->recordsStored -= (currentBlockSize / sizeof(Record));
        this->availableBlocks++;
        return true; // Copy to the memory pointed by cursor
    }
    return false;
}
```

deleteBlock Function:

a) This function is used to delete a specific block within the database
b) It creates an array of 400 bytes, initializing all of them to zero.
c) It calculates the memory location of the block to delete and uses ‘memset’ and ‘memcpy’ to erase the data.
d) Updates recordsStored and availableBlocks accordingly.

```
bool Storage::deleteRecord(int blockID, int offset)
{
    uchar *cursor = baseAddress;
    cursor += (blockID * blockSize) + offset;
    uchar zeroBytes[sizeof(Record)];         // Create an array of 400 null bytes
    memset(b: zeroBytes, c: 0, len: sizeof(zeroBytes)); // Initialize it to zeros
    if (memcpy(dst: cursor, src: zeroBytes, n: sizeof(zeroBytes))){
        this->recordsStored--;
        //TODO: logic for checking if there is only one node left in the block
        //TODO: logic for updating the blockRecords map
        return true; // Copy to the memory pointed by cursor
    }
    return false;
}
```

deleteRecord Function:

a) This function is used to delete a specific record within a block.
b) It creates an array of 400 bytes, initializing all of them to zero.
c) It calculates the memory location of the record to delete and uses memset and memcpy to erase the data.
d) Updates recordsStored.

```
bool Storage::allocateRecord(Record record)
{
    // Error case: no blocks are available
    if (availableBlocks == 0)
    {
        return false;
    }
    // Try to find an available block and get the address of it
    uchar *blockAddress = findAvailableBlock(size: sizeof(Record));
    if (!blockAddress)
    {
        cerr << "Failed to find an available block" << endl;
        return false;
    }
    // Update record header
    record.setBlockAddress(this->getBlockAddress(blockID: this->currentBlock));
    int offset = currentBlockSize;
    record.setOffset(offset);
    record.print();
    // Copy the record data to the memory address
    memcpy(dst: blockAddress, src: &record, n: sizeof(Record));
    // cout << "Current block size " << currentBlockSize << endl;
    currentBlockSize += sizeof(Record);
    // Update blockRecords value to keep track
    auto find: iterator = blockRecords.find(k: currentBlock);
    if (find != blockRecords.end())
    {
        find->second += 1;
    }
    recordsStored++;
    // Move the cursor forward by the amount of memory allocated
    databaseCursor = (blockAddress + sizeof(Record));
    return true;
}
```

allocateRecord function:

a) This function allocates memory to store a ‘Record’ object in the database
b) It checks for available blocks and memory space to allocate the record.
c) Moreover, it updates the record's block address, offset, etc and copies the record data into the allocated memory.
d) It returns true if allocation is successful, and false if there is an error in allocation.

```
uchar *Storage::findAvailableBlock(int recordSize)
{
    // Error case: no blocks are available
    if (currentBlockSize + recordSize > blockSize && availableBlocks == 0)
    {
        return nullptr;
    }
    // Case 1: The new record can fit in an existing block
    if ((currentBlockSize + recordSize) <= blockSize)
    {
        return databaseCursor;
    }
    // Case 2: A new block has to be allocated for the new record
    if (currentBlockSize + recordSize > blockSize && availableBlocks > 0)
    {
        // Unspanned implementation, go past the remaining fields
        databaseCursor += (blockSize - currentBlockSize);
        // Change internal variables to acknowledge the new block
        availableBlocks--;
        currentBlock++;
        currentBlockSize = 0;
        blockRecords[currentBlock] = 0;
        return databaseCursor;
    }
}
```

findAvailableBlock function:

a) It is a helper function that is used to find the memory address of the block with capacity for a record, or create a new block
b) If there are no available blocks and the new record cannot fit in the current block, a null pointer will be returned to indicate that no block is available.
c) If there are available blocks, our method checks if the new record can fit in the current block. If the new record can fit in the current block, our method returns the memory address of the current block.
d) If the new record cannot fit in the current block, our method allocates a new block for the new record. Our method then updates the 'databaseCursor' pointer to point to the next available memory address in the new block, and updates the `availableBlocks`, `currentBlock`, and `currentBlockSize` attributes to reflect the new block allocation. Our method then returns the memory address of the new block.

```
uchar *Storage::readBlock(int blockID)
{
    // Calculate starting memory address of block from base address of database and block size
    uchar *blockCursor = baseAddress + (blockID * blockSize);
    uchar *copy = new uchar[400];
    memcpy(dst: copy, src: blockCursor, n: blockSize);
    return copy;
}
```

readBlock function:

a) It is a function that copies a block to “RAM” and returns the memory of the copied area.
b) It returns the memory address of the copied data.

```
void Storage::printBlockRecords()
{
    for (const auto &pair: const value_type & : blockRecords)
    {
        cout << "Block ID: " << pair.first << ", Num Records: " << pair.second << endl;
    }
}
```

printBlockRecords function:

a) It is a function that prints the contents of the blockRecords attribute in the database

```
vector<Record> Storage::readAllRecords()
{
    vector<Record> records;
    for (int blockID = 0; blockID <= currentBlock; ++blockID)
    {
        vector<Record> blockRecords = readRecordsFromBlock(blockID);
        records.insert(position: records.end(), first: blockRecords.begin(), last: blockRecords.end());
    }
    return records;
}
```

readAllRecords function:

a) It is a function that reads all records in the database object and returns them as a vector.

```cpp
vector<Record> Storage::readRecordsFromBlock(int blockID)
{
    vector<Record> records;
    uchar nullBytes[sizeof(Record)];
    memset(b: nullBytes, c: 0, len: sizeof(Record));
    uchar *blockCursor = baseAddress + (blockID * blockSize); // starting memory address of the particular block
    int numRecordsInBlock = recordsInBlock(blockID);
    if (numRecordsInBlock == 0)
    {
        return records;
    }
    for (int i = 0; i < numRecordsInBlock; i++)
    {
        int offset = i * sizeof(Record);
        if (memcmp(s1: blockCursor + offset + sizeof(Record), s2: nullBytes, n: sizeof(Record)) == 0)
        {
            continue;
        }
        else
        {
            float fgPct = *reinterpret_cast<float *>(blockCursor + offset + offsetof(Record, fgPct));
            float ftPct = *reinterpret_cast<float *>(blockCursor + offset + offsetof(Record, ftPct));
            float fg3Pct = *reinterpret_cast<float *>(blockCursor + offset + offsetof(Record, fg3Pct));
            int gameDate = *reinterpret_cast<int *>(blockCursor + offset + offsetof(Record, gameDate));
            uchar *blockAddress = *reinterpret_cast<uchar **>(blockCursor + offset + offsetof(Record, blockAddress));
            int recordOffset = *reinterpret_cast<int *>(blockCursor + offset + offsetof(Record, offset));
            unsigned short int recordID = *reinterpret_cast<unsigned short int *>(blockCursor + offset + offsetof(Record, recordID));
            uint8_t teamID = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, teamID));
            uint8_t pts = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, pts));
            uint8_t ast = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, ast));
            uint8_t reb = *reinterpret_cast<uint8_t *>(blockCursor + offset + offsetof(Record, reb));
            bool homeTeamWins = *reinterpret_cast<bool *>(blockCursor + offset + offsetof(Record, homeTeamWins));
            Record record = Record(fgPercentage: fgPct, ftPercentage: ftPct, fg3Percentage: fg3Pct, date: gameDate, blockAddress,
            offset: recordOffset, recordID, teamID, points: pts, assists: ast, rebounds: reb, homeTeamWins);      You, 1 second ago • Uncommitted changes
            records.push_back(x: record);
        }
    }
    return records;
}
```

readRecordsFromBlock function:

a) It is a function that reads records from a specific block, identified by the ‘blockID’.
b) It calculates the memory address of the start of the specified block based on the “baseAddress” and “blockSize”
c) If there are no records in the specified block, an empty records vector is returned early.
d) It returns the ‘records’ vector that contains all the records of the block.

```cpp
int Storage::recordsInBlock(int blockID)
{
    auto find: iterator = blockRecords.find(k: blockID);
    if (find != blockRecords.end())
    {
        return find->second;
    }
    return 0;
}
```

recordsInBlock function:

a) It is a function that returns the number of records stored in a block by looking up the blockRecords map
b) It returns the number of records as an integer value.

```
void Storage::setRecordsInBlock(int blockID, int value)
{
    auto find: iterator = blockRecords.find(k: blockID);
    if (find != blockRecords.end())
    {
        find->second = value;
    }
    else
    {
        blockRecords[blockID] = value;
    }
}
```

setRecordsInBlock function:

a) It is a function that sets the number of records stored in a specific block in the blockRecords map.
b) If the block already has an entry in the blockRecords map, it updates the value associated with it, else if there's no existing entry for the block it creates a new entry with the provided value.

```cpp
vector<Record> Storage::readRecordsfromAddresses(vector<Address> addresses)
{
    // Initialize a map to store the indexes we want to read for each blockID
    int blockAccessCount = 0;
    map<int, vector<int>> indexMap;
    for (int i = 0; i < addresses.size(); i++)
    {
        int blockID = getBlockID(blockAddress: addresses[i].blockAddress);
        int index = addresses[i].offset / sizeof(Record);
        if (indexMap.find(k: blockID) != indexMap.end())
        {
            // Push index to existing vector if the key already exists
            indexMap[blockID].push_back(x: index);
        }
        else
        {
            // Create and push a vector with index value if the key does not exist
            indexMap[blockID] = vector<int>{index};
            // indexMap.insert(blockID, vector<int>{index});
        }
    }
    // Value to return, vector of Records
    vector<Record> results;
    // Iterate through all keys of hashmap (block ID we want to read)
    for (const auto &pair: const value_type & : indexMap)
    {
        // Perform a unit reading of each block
        vector<Record> recordsFromBlock = readRecordsFromBlock(blockID: pair.first);
        blockAccessCount++;
        // TODO: Make sure to write in the report how we're reading each block only once to optimise the "I/O operations"
        // Get the indexes we want to read from each block
        vector<int> indexes = pair.second;
        for (int index : indexes)
        {
            // Push each index into the results
            results.push_back(x: recordsFromBlock.at(n: index));
        }
    }
    cout << "Number of Data Blocks Accessed: " << blockAccessCount << endl;
    return results;
}
```

readRecordsfromAddresses function:

a) This function reads records from specific memory addresses based on the list of the addresses provided in the “addresses” vector. It optimizes reading by minimizing the number of block accesses.
b) It initializes a map to store the indexes we want to read for each BlockID.
c) The code uses a for loop to iterate through the elements in the addresses vector.
d) After processing all block IDs and indexes, the function prints the number of data blocks accessed (blockAccessCount) to the console.
e) Finally, it returns the results vector, which contains the records that were read based on the provided addresses.

```cpp
vector<Record> Storage::readRecordsfromNestedAddresses(vector<vector<Address>> addresses)
{
    int blockAccessCount = 0;
    map<int, vector<int>> indexMap;
    for (int i = 0; i < addresses.size(); i++)
    {
        vector<Address> cursor = addresses[i];
        for (int j = 0; j < cursor.size(); j++)
        {
            int blockID = getBlockID(blockAddress: cursor[j].blockAddress);
            int index = cursor[j].offset / sizeof(Record);
            if (indexMap.find(k: blockID) != indexMap.end())
            {
                // Push index to existing vector if the key already exists
                indexMap[blockID].push_back(x: index);
            }
            else
            {
                // Create and push a vector with index value if the key does not exist
                indexMap[blockID] = vector<int>{index};
                // indexMap.insert(blockID, vector<int>{index});
            }
        }
    }
    // Value to return, vector of Records
    vector<Record> results;
    // Iterate through all keys of hashmap (block ID we want to read)
    for (const auto &pair: const value_type & : indexMap)
    {
        // Perform a unit reading of each block
        vector<Record> recordsFromBlock = readRecordsFromBlock(blockID: pair.first);
        blockAccessCount++;
        // TODO: Make sure to write in the report how we're reading each block only once to optimise the "I/O operations"
        // Get the indexes we want to read from each block
        vector<int> indexes = pair.second;
        for (int index : indexes)
        {
            // Push each index into the results
            results.push_back(x: recordsFromBlock.at(n: index));
        }
    }
    cout << "Number of Data Blocks Accessed: " << blockAccessCount << endl;
    return results;
}
```

readRecordsfromNestedAddresses function:

a) This function reads records from a nested vector of addresses where each inner vector represents a set of addresses.
b) It initializes a map to store the indexes we want to read for each BlockID.
c) The code uses nested for loops to iterate through the elements in the addresses vector of vectors.
d) The outer loop (i) iterates over the outer vector, and the inner loop (j) iterates over each inner vector (cursor) contained within the outer vector.
e) After processing all block IDs and indexes, the function prints the number of data blocks accessed (blockAccessCount) to the console.
f) Finally, it returns the results vector, which contains the records that were read based on nested addresses.

```
int Storage::removeRecordsfromNestedAddresses(vector<vector<Address>> addresses)
{
    int blockAccessCount = 0;
    map<int, vector<int>> indexMap;
    for (int i = 0; i < addresses.size(); i++)
    {
        vector<Address> cursor = addresses[i];
        for (int j = 0; j < cursor.size(); j++)
        {
            int blockID = getBlockID(blockAddress: cursor[j].blockAddress);
            int index = cursor[j].offset / sizeof(Record);
            if (indexMap.find(k: blockID) != indexMap.end())
            {
                // Push index to existing vector if the key already exists
                indexMap[blockID].push_back(x: index);
            }
            else
            {
                // Create and push a vector with index value if the key does not exist
                indexMap[blockID] = vector<int>{index};
                // indexMap.insert(blockID, vector<int>{index});
            }
        }
    }
    cout << "Printing blocks and indexes to be deleted" << endl;
    for (const auto &pair: const value_type & : indexMap)
    {
        cout << "Block Number: " << pair.first << ", Indexes : ";
        vector<int> indexes = pair.second;
        // Check if the length is the full block, then just delete the whole block
        if (indexes.size() == (blockSize / sizeof(Record)))
        {
            if (deleteBlock(blockID: pair.first))
            {
                cout << "Block ID " << pair.first << "sucessfully deleted" << endl;
            }
        }
        for (int index : indexes)
        {
            // Push each index into the results
            cout << index << " ";
        }
        cout << endl;
    }
    return 1;
}
```

removeRecordsfromNestedAddresses function:

a) This function is used to remove records from the database based on a vector of nested addresses.
b) It initializes a map to store the block IDs and indexes of the records to be deleted.
c) It iterates over each vector of Address objects in the input vector. For each Address object, it calculates the block ID and index of the corresponding record, then checks if the block ID already exists in the map.
d) For each block ID in the map, the method checks if the vector of indexes is the full block size.
e) If the vector of indexes is the full block size, the method deletes the entire block.

Achieving One-Pass Deletion and Insertion

In addition, we also use some optimizaton when reading records from a disk by storing a map of the blockID and offsets to read for all the records we want to read. By doing pre-processing to ensure that we know all the offsets from one blockID, we avoid having to read the block multiple times from disk if we do not have enough memory space. The same is done for deletion so we can achieve a one pass deletion for each block and all the records inside it.

B+ Tree Design

In our database management system, we have implemented a B+ Tree structure which provides efficient indexing and retrieval capabilities for large datasets like our NBA games database. This meets the need for a robust and efficient data structure to manage and retrieve games by FG_PCT_home (Field Goal Percentage).

Implementation Details:

Node Structure

We followed the basic B+ tree structure where we have two different types of nodes, internal and leaf nodes. The leaf nodes store actual addresses of the records whereas the internal nodes store keys and pointers to the child nodes.

```
class Node{
    private:
        float *keys;
        uint8_t numKeys;
        uint8_t maxKeys;
        vector<vector<Address>> children;
        bool isLeaf;
        friend class BPlusTree;
```

To uniquely identify each record in the storage, we created a new structure *Address* which contained the blockAddress and the offset for every record. This helped us in accessing the records from the tree efficiently.

```
struct Address
{
  void *blockAddress;
  int offset;

//Constructor
  Address(void* blockAddress, int offset) : blockAddress(blockAddress), offset(offset) {}
};
```

The parameter ‘n’ or maximum keys allowed in a node is an important parameter. We have specified it as *maxKeys*. The parameter significantly influences the structure and performance of the tree. If the parameter is too small, the tree may become excessively tall, leading to slower searches. Conversely, if the parameter is too large, it may result in wasted memory space and slower reactions. Our logic behind finding the best appropriate size is to use the number of records to be stored in the tree at runtime to dynamically compute the r

Address

The Address structure is used to keep track of the pointers stored in the nodes of a B+ tree. It consists of the following variables:

- **void *blockAddress**: Address of the first byte of the block
- **int offset**: Offset from the record from the block address

Node

Node class is then created for the nodes in B+ tree:

The Node class consists of following attributes:

- **float *keys**: Array of keys
- **uint8_t numKeys**: Number of keys stored in a node
- **uint8_t maxKeys**: Maximum number of keys that can be stored in the node
- **vector<vector<Address>> children**: Vector of children
- **bool isLeaf**: Boolean to check if node is leaf
- **BPlusTree**: Friend class to allow BPlusTree to access private members of the node

In addition, the Node class consists of following functions to construct and destruct Node objects, and to get and set Node attributes:

To address the issue of duplicate keys, we pre-processed our records to generate a vector of all the *Addresses* sharing the same key. This vector has further been utilized to optimize the number of insertion calls made on the B+ tree.

```
    Node(int maxKeys, bool isLeaf);
    ~Node();

    float getKey(int index){
        return keys[index];
    };
    int getNumKeys(){
        return static_cast<int>(numKeys);
    };
    int getMaxKeys(){
        return static_cast<int>(maxKeys);
    };
    vector<Address> getChildren(int index){
        return children[index];
    };
    int getIsLeaf(){
        return isLeaf;
    };
    void setKey(int index, float key){
        keys[index] = key;
    };
    void setNumKeys(int numKeys){
        this->numKeys = numKeys;
    };
    void setChildren(int index, vector<Address> children){
        this->children[index] = children;
    };
    void setIsLeaf(bool isLeaf){
        this->isLeaf = isLeaf;
    };
```

Dealing with duplicate keys

To deal with duplicate keys, we did pre-processing again and created a vector for the addresses of all records with the same key. This is again used to optimize the number of insertion calls we make on the b+ tree - which will now be only the set of unique values that we have. Most of the games with the same key are also stored on the same block, which reduces the number of disk I/O when reading the records.

BPlusTree Structure

```
class BPlusTree{
    public:
        Node* rootNode;
        Node* index;
        int keysStored;
        int maxKeys;
        int levels;
        int nodesStored;
        int nodeSize;
        int blockSize;
```

Our BPlusTree class consists of following attributes:

- **Node* rootNode**: Memory address of the root node of the tree
- **Node* index**: Index node of the tree
- **int keysStored**: Number of keys stored in the tree
- **int maxKeys**: Maximum number of keys that can be stored in a node
- **int levels**: Number of levels in the tree
- **int nodesStored**: Number of nodes stored in the tree
- **int nodeSize**: Size of a node
- **int blockSize**: Size of a block

B+ Tree Search

Our B+ tree search functions first checks if the root node of the tree is null. If the root node is null, the function prints an error message and returns an empty vector. Else, it traverses the tree to find the appropriate leaf node for the key. Our search functions check if the current node is a leaf node. If the current node is not a leaf node, it updates the cursor to point to the appropriate child node and continues traversing until the leaf node is found. Once found, our search functions iterate over all the keys in the node to find the key that matches the input parameter. If the key is found, our search functions return the corresponding vector of Address objects.

We implemented 2 search functions for our B+ Tree:

```
> vector<Address> BPlusTree::searchKey(float key){···
```

This code is meant for performing a search operation on B+ Tree to find a specific key. It was used for Experiment 3.

```
> vector<vector<Address>> BPlusTree::searchRange(float low, float high){···
```

This code is meant for performing a search operation on B+ tree to find a range of keys. It was used for Experiment 4.

B+ Tree Insertion

```
int BPlusTree::insert(float key, const vector<Address> value)
```

This method is used to insert a new key-value pair into the B+ tree. It takes a float key and a vector of Address objects as input. It first checks if the root node is null. If the root node is null, it creates a new root node and sets the key and child pointer values accordingly. If the root node already exists, it traverses the tree to find the appropriate leaf node to insert the new key-value pair into the leaf node and updates the parent nodes accordingly. If the leaf node is full, it splits the node into two nodes and updates the parent nodes accordingly.

```
int BPlusTree::insertInternal(float key, Node *parent, Node *child)
```

This helper method is used by insert() method to insert a new key-value pair into an internal node of the B+ tree. It takes a float key, a Node object representing the parent node, and a Node object representing the child node as input. It first checks if the parent node is null. If the parent node is null, it creates a new root node and sets the key and child pointer values accordingly. If the parent node already exists, it inserts the new key-value pair into the parent node and updates the child pointers accordingly. If the parent node is full, it splits the node into two nodes and updates the parent nodes accordingly.

Since our records are sorted for fgPct, we are doing bulk loading in our B+ tree which helps to ensure that all the leaf nodes are as full as possible.

B+ Tree Deletion

```
int BPlusTree::deleteNode(float key)
```

This method is used to delete a range of key-value pairs from the B+ tree. It takes a float key as input. It first checks if the root node is null. If root is null, it will return. Else, it will search for the leaf node where the key to delete should be. If the key to delete is not found in the leaf node, it will return the key that is not in the tree. Else, it will remove the key and associated value from the leaf node.

After deletion, it will check if the leaf node has enough key. If the leaf node has fewer keys than required, redistribution of the leaf node will be tried. If redistribution is not possible, it will attempt to merge the leaf node with the left sibling. If the left sibling is full, it will merge with the right sibling. Afterwards, it will update the parent nodes and propagate changes up the tree.

We are not rearranging records after deletion because we will have to rebuild the Address values of the b+ tree which will be computationally expensive.

Experiment Results

Experiment 1

```
Experiment 1
Number of Records: 26651
Size of a Record: 40 bytes
Number of Records in a Block : 10
Number of Blocks for Storing Data : 2666
----------------------------------------
```

Number of records	26651
Size of a record	40
Number of records stored in a block	10
Number of blocks for storing the data	2666

Experiment 2

```
----------------------------------------
Experiment 2
Parameter 'n' of B+ Tree: 16
Number of nodes of B+ Tree: 25
Number of levels of B+ Tree: 3
Content of Root Node:
| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
----------------------------------------
```

Parameter n of the B+ tree	16
Number of nodes of the B+ tree	25
Number of levels of the B+ tree	3
Content of the root node (only the keys)	| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x |

Experiment 3

```
Experiment 3
Index Nodes Accessed while Searching B+ Key: 3
Number of Data Blocks Accessed for Reading Records from Database: 86
Number of Records with FG_PCT 0.5: 848
Linear Search for Records with FG_PCT 0.5
Average of FG3_PCT_HOME for Records Returned: 0.391201
Number of Records with FG_PCT 0.5 with Linear Search: 848
Number of Data Blocks accessed for Query with Linear Search: 2060
Runtime of Search and Retrieval: 806625 nanoseconds
Runtime of Linear Search of Data Blocks: 7213208 nanoseconds
```

Number of index nodes the process accesses	3
Number of data blocks the process accesses	86
Average of “FG3_PCT_home” of the records that are returned	0.391201
Running time of the retrieval process	806625 nanoseconds
Number of data blocks that would be accessed by a brute-force linear scan method	2600

Our linear search stops when we encounter the first record that is greater than 0.5, because we know that the data is structured in ascending order. So we can stop the search when we find the first key that violates the upper bound.

Experiment 4

```
----------------------------------------------
Experiment 4
Number of Index Nodes Accessed in B+ Tree Search: 3
Number of Data Blocks Accessed: 24
Average of FG3_PCT_HOME for Records Returned: 0.502329
Number of Records with FG_PCT between 0.6 and 1 (inclusive) with Linear Search: 237
Number of Data Blocks accessed for Ranged Query with Linear Search: 2666
Runtime of Search and Retrieval: 265625 nanoseconds
Runtime of Linear Search of Data Blocks: 7313458 nanoseconds
----------------------------------------------
```

Number of index nodes the process accesses	3
Number of data blocks the process accesses	25
Average of “FG3_PCT_home” of the records that are returned	0.502555
Running time of the retrieval process	314083 nanoseconds
Number of data blocks that would be accessed by a brute-force linear scan method	2666
Running time for brute-force linear scan	7313458 nanoseconds

We had to iterate through all the nodes as we have no way of knowing the minimum value in a simple linear search. So we had to start from the 0th index and go till the end.

Experiment 5

```
----------------------------------------
Experiment 5
Number of nodes of the updated B+ tree: 21
Number of levels of the updated B+ tree: 3
Content of the Root Node:
| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
Runtime of Linear Seach Query for Keys Between 0 and 0.35: 9275583 nanoseconds
Number of Data Blocks Accessed for Linear Search: 2666
Runtime of B+ Tree Query for Keys Between 0 and 0.35: 120208 nanoseconds
Runtime of B+ Tree Deletion of Keys Between 0 and 0.35: 1005000 nanoseconds
Runtime of Database Record Deletion of Keys Between 0 and 0.35: 2191625 nanoseconds
```

Number of nodes of the updated B+ tree	21
Number of levels of the updated B+ tree	3
Content of the root node of the updated B+ tree	| 0.425 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
Running time of the process	2191625 nanoseconds
Number of data blocks that would be accessed by a brute-force linear scan method	2666
Running time for brute-force linear scan	9275583 nanoseconds