Ahmed Haitham Ibrahim

German University in Cairo

Tuesday 16th, March 2023

CSEN604: Database II - MET Project 1

**Database Engine Project Report**

***Introduction***

The purpose of this report is to provide a comprehensive overview of the database engine project. This project aims to develop a robust and efficient database engine that incorporates various packages and objects to handle data storage, retrieval, and manipulation. The project consists of three main packages: "Exception," "Objects," and "DBApp."

**The project description is** [**here**](file:///E:\Semster%206\2023-MET-DBII-Project\MET-2023-DatabaseII-MiniProject-1.pdf)**.**

***File 1: meta***

Metadata Storage: The meta file provides mechanisms for storing metadata information associated with serialized files. This can include details such as the file's name, size, creation date, last modification date, and any additional custom attributes specific to the application.

Metadata Retrieval: The file offers methods and utilities to retrieve metadata information for serialized files. This can involve querying the metadata storage based on various criteria, such as file name, file type, or custom attributes, to retrieve the relevant metadata.

Metadata Updates: The meta file enables the updating of metadata information as needed. This could involve modifying existing metadata attributes, adding new attributes, or removing outdated or unused metadata entries.

Search and Indexing: The file may include functionalities for searching and indexing metadata information. This allows for efficient retrieval of serialized files based on specific metadata criteria, enabling faster and targeted data retrieval operations.

Serialization and Deserialization: The meta file may also handle the serialization and deserialization of objects to and from serialized files. This can involve converting objects into a serialized format and storing them as files, as well as reading serialized files and reconstructing the objects.

Integration with Storage Systems: The meta file might provide integration capabilities with various storage systems, such as databases, file systems, or cloud storage services. This enables seamless interaction between the metadata management functionality and the underlying storage infrastructure.

By encapsulating metadata management within the meta package, the system can effectively organize and retrieve information about serialized files, enhancing data management capabilities and facilitating efficient data access and manipulation operations.

***File 2: Config***

holds a parameters as key=value pairs, of static values.

***Package 1: Exceptions***

The "Exception" package is responsible for handling exceptional scenarios within the database engine. It contains the following class:

a. DBAppException: This class is designed to handle specific exceptions that may occur during the execution of the database engine. It provides detailed error messages and logging capabilities to assist in debugging and troubleshooting.

***Package 2: Objects***

The "Objects" package encompasses various classes representing fundamental components of the database engine. These classes are designed to facilitate efficient data storage, indexing, and querying. The classes within this package include:

a. Page: The Page class represents a storage unit for data in the database engine. It manages the allocation, deallocation, and organization of data within the pages.

b. Table: The Table class defines the structure of a database table and provides methods for creating, modifying, and querying table data. It handles operations such as insertion, deletion, and updates efficiently.

***Package 3: Octree***

a. Node: The Node class implements the indexing structure used for faster data retrieval. It supports operations like searching, insertion, and deletion of records based on index keys.

b. Point: The Point class represents a data point or a record within the database. It encapsulates the data attributes and provides methods for accessing and modifying them.

c. Octree: The Octree class is a specialized data structure used for spatial indexing. It partitions the data space into octants to efficiently search for spatial objects.

***Package 4: src***

The "src" package serves as the core logic of the database engine. It integrates the Exception and Objects packages to provide the complete functionality of the database engine. The DBApp class incorporates methods for data manipulation, query processing, and transaction management.

***Project File Structure***

The project is organized using a standard file structure, as outlined below:

**/exception:**

DBAppException.java

**/objects:**

Page.java

Table.java

Node.java

Point.java

Octree.java

**/src:**

DBApp.java

**Class Page**

***Page():*** Constructor for the Page class. Initializes a new page object and sets up the maximum number of rows for the page by reading the value from the DBApp.config file.

***boolean isFull():*** Checks if the page is full (i.e., the number of tuples in the page has reached the maximum allowed).Returns true if the page is full; otherwise, returns false.

***int getMaxRows():*** Returns the maximum number of rows allowed in the page.

***Vector<Hashtable<String, Object>> getTuples():***Returns the vector of tuples stored in the page.

***compareToObject() :*** The **compareToObject** method compares two objects **Value** and **pageValue** and returns the result of the comparison as an integer. The method handles different data types, including **Integer**, **Double**, **String**, and **Date.**

***int binaryOnTuples() :***Performs a binary search on the tuples within the page to find the correct insertion position for a new tuple. Uses the primary key value from the input table and the specified primary key attribute name to determine the insertion position.Throws a DBAppException if the new tuple's primary key already exists in the page.

***void addRow():***Adds a new row (tuple) to the page while maintaining the order based on the primary key. ***C***alls binaryOnTuples to find the appropriate insertion position for the new tuple.Inserts the new tuple at the calculated position.

***void serializePage():*** Serializes the current page, object and writes it to a file. The serialized file is named based on the tableName and index parameters. Throws an IOException if there is an error during file serialization.

These methods provide essential functionality for managing the tuples within a page, including checking if the page is full, comparing objects for ordering, performing binary searches, adding new tuples while maintaining order, and serializing the page to disk.

**Class Table**

***public Table():*** *Constructor for the Table class, Initializes a new table object with the specified tableName and Initializes an empty list of indices (Indices).*

***public String getTableName():*** *Returns the name of the table.*

***public void setTableName():*** *Sets the name of the table to the specified tableName.*

***public void serializeTable():*** *Serializes the current table object and writes it to a file, The serialized file is named based on the tableName. Throws an IOException if there is an error during file serialization.*

***public void addPages():*** *Adds a new page to the table, Creates a new Page object, Increments the noOfPages counter.*

*Serializes the newly created page to disk using the serializePage method from the Page class.*

*Serializes the updated table object using the serializeTable method.*

***public String getStrClusteringKey():*** *Returns the name of the clustering key for the table.*

***void deletePage() :****Deletes a page from the table, Decrements the noOfPages counter. Serializes the updated table object using the serializeTable method. These methods provide functionality for managing the properties and pages of a table, including getting and setting the table name, serializing the table to disk, adding new pages to the table, getting the clustering key, and deleting pages from the table.*

**Class: Point**

*The Point class represents a point in three-dimensional space. It is designed to store the coordinates of a point and an associated reference value. The class implements the Serializable interface, allowing objects of this class to be serialized and deserialized.*

***x (Object):*** *Stores the x-coordinate of the point.*

***y (Object):*** *Stores the y-coordinate of the point.*

***z (Object):*** *Stores the z-coordinate of the point.*

***ref (int):*** *Stores the reference value associated with the point.*

***Point(Object x, Object y, Object z, int ref):*** Constructs a Point object with the specified coordinates and reference value. *.*

**Class: Node**

***Node(Object x, Object y, Object z, int PageIndex):***Constructor for the Node class.

Initializes a new node object and sets up the maximum number of points/entries for the node by reading the value from the DBApp.config file. Adds a new point to the Points list with the specified x, y, z coordinates, and PageIndex. Initializes an empty list of duplicates (Duplicates).

***ArrayList<Point> Points:*** Represents the list of points/entries stored in the node.

***ArrayList<Point> Duplicates:*** Represents the list of duplicate points/entries in the node.

***int maxPoints:*** Represents the maximum number of points/entries allowed in the node.

These methods and properties provide functionality for managing the points/entries in a node of an octree structure. The Node class allows for the addition of points, retrieval of points, and handling duplicates. The maximum number of points allowed in a node is determined by a configuration value read from the DBApp.config file.

**Class: Octree**

**Node root:** Represents the root node of the octree.

**Octree []childNodes:** Represents an array of child octrees.

Each child octree corresponds to a specific octant within the parent octree.

**boolean parent:** Indicates whether the current octree is a parent octree (has child octrees) or a leaf octree (no child octrees).

***Object minX, public Object minY, public Object minZ, public Object maxX, public Object maxY, public Object maxZ:***

Represents the minimum and maximum values for the x, y, and z dimensions of the octree.

These values define the boundaries of the octree's spatial region.

public String col1, public String col2, public String col3:

Represents the names of the three columns associated with the octree.

These columns store the x, y, and z coordinates of the points/entries in the octree.

**Octree(Node p):** Constructor for the Octree class, Initializes a new octree object with the specified p as the root node.

**Octree():** Default constructor for the Octree class, Initializes a new empty octree object.

These methods and properties provide functionality for creating and managing an octree data structure. The octree can have child octrees corresponding to specific octants, and it maintains information about the spatial boundaries and column names associated with the octree.

**Class: DBApp**

**Helper Methods**

**Method:** ***performRangeQuery***

**Serves: *selectFromTable***

**Description:**

The **performRangeQuery** method performs a range query on an Octree data structure to retrieve a list of integers based on the specified operator and point p. This method supports the ">" and "<" operators.

**Method Name:** ***performQueryWithIndex***

**Serves: *selectFromTable***

**Description:**

The **performQueryWithIndex** method performs a complex query operation on a given table **strTableName** using a combination of SQL terms and operators. It retrieves the tuples that satisfy the specified conditions and returns the result as a vector of **Hashtable** objects.

**Parameters:**

* **pageIndices (ArrayList<Integer>):** A list of page indices to restrict the query to specific pages.
* **strTableName (String):** The name of the table to perform the query on.
* **arrSQLTerms (SQLTerm[]):** An array of SQLTerm objects representing the search conditions for the query.
* **strarrOperators (String[]):** An array of string operators corresponding to the SQL terms, specifying the logical operations to apply between the terms.

**Method Name:** ***executeComplexQuery***

**Serves: *selectFromTable***

**Description:** The **executeComplexQuery** method is responsible for executing a complex query on a specified table. It takes a table name, an array of SQLTerm objects, and an array of operators as parameters.

**Parameters:**

* **strTableName (String):** The name of the table on which the complex query is executed.
* **arrSQLTerms (SQLTerm[]):** An array of SQLTerm objects representing the search conditions.
* **strarrOperators (String[]):** An array of operators specifying the logical operations between the search conditions.

**Method Name:*****allOperatorsAreAND***

**Serves: *selectFromTable***

**Description:** The **allOperatorsAreAND** method checks whether all operators in the given array are "AND." It takes an array of operators as a parameter. Returns **true** if all operators in the array are "AND," **false** otherwise.

**Method Name:** ***doOperation***

**Serves: *selectFromTable***

**Description:** The **doOperation** method performs a comparison between **tupleValue** and **SQLTermValue** using the specified **operator**. It takes three parameters: **tupleValue** represents the value of the tuple, **operator** specifies the type of comparison to perform, and **SQLTermValue** is the value to compare against the tuple value. Returns **true** if the comparison between **tupleValue** and **SQLTermValue** using the specified **operator** is true; otherwise, **false**.

**Parameters:**

* **tupleValue** (Object): The value of the tuple for comparison.
* **operator** (String): The operator to perform the comparison.
* **SQLTermValue** (Object): The value to compare against the tuple value.

**Method Name:** ***sub***

**Serves: *selectFromTable***

**Description:** The **sub** method calculates the set subtraction of two vectors of hashtables, **a** and **b**. It takes two parameters: **a** represents the first vector, and **b** represents the second vector. Returns A vector of hashtables that contains the elements from **a** that are not present in **b**.

**Parameters:**

* **a (Vector<Hashtable<String, Object>>):** The first vector of hashtables.
* **b** **(Vector<Hashtable<String, Object>>):** The second vector of hashtables.

**Method Name: *intersection***

**Serves: *selectFromTable***

**Description:** The **intersection** method calculates the intersection of two vectors of hashtables, **a** and **b**. It takes two parameters: **a** represents the first vector, and **b** represents the second vector. Returns A vector of hashtables that contains the common elements between **a** and **b**.

**Method Name:** ***union***

**Serves: *selectFromTable***

**Description:** The **union** method calculates the union of two vectors of hashtables, **a** and **b**. It takes two parameters: **a** represents the first vector, and **b** represents the second vector. Returns A vector of hashtables that contains all elements from both **a** and **b**, removing any duplicates.

**Method Name:** ***getTuplesFromIndex***

**Serves:**

**Description:** The **getTuplesFromIndex** method retrieves tuples from the specified page indices based on the provided search criteria. It takes multiple parameters: **pageIndices** represents the list of page indices to retrieve tuples from, **strTableName** represents the name of the table, **key** represents the column name to perform the comparison on, **operator** represents the comparison operator to apply, and **value** represents the value to compare against. Returns A vector of hashtables containing the tuples that satisfy the given search criteria.

**Parameters:**

* **pageIndices** (ArrayList<Integer>): A list of page indices to retrieve tuples from.
* **strTableName** (String): The name of the table.
* **key** (String): The column name to perform the comparison on.
* **operator** (String): The comparison operator to apply.
* **value** (Object): The value to compare against.

**Method Name: *getTuples***

**Serves:**

**Description:** The **getTuples** method retrieves tuples from the specified table based on the provided search criteria. It takes multiple parameters: **strTableName** represents the name of the table, **key** represents the column name to perform the comparison on, **operator** represents the comparison operator to apply, and **value** represents the value to compare against. Returns A vector of hashtables containing the tuples that satisfy the given search criteria.

**Parameters:**

* **strTableName** (String): The name of the table.
* **key** (String): The column name to perform the comparison on.
* **operator** (String): The comparison operator to apply.
* **value** (Object): The value to compare against.

**Method Name:** ***getPagesByRange***

**Serves:**

**Description:** The **getPagesByRange** method retrieves the page references that satisfy a range query based on the provided search condition. It takes three parameters: **sqlTerm**, the SQLTerm object representing the search condition; **tree**, the Octree object representing the index structure; and **strTableName**, the name of the table. Returns An ArrayList containing the page references that satisfy the range query.

**Parameters:**

* **sqlTerm** (SQLTerm): The SQLTerm object representing the search condition.
* **tree** (Octree): The Octree object representing the index structure.
* **strTableName** (String): The name of the table.

**Method Name:** ***getIndexName***

**Serves:**

**Description:**

The **getIndexName** method retrieves the name of the index associated with the specified column names in a table. Returns The name of the index associated with the specified column names.

**Parameters:**

* **colNames** (ArrayList<String>): The ArrayList containing the column names.
* **strTableName** (String): The name of the table.

**Method Name:** ***isClusterIndexed***

**Serves:**

**Description:** The **isClusterIndexed** method checks whether the specified clustering key is indexed in the given table. It takes two parameters: **strClusteringKey**, the name of the clustering key, and **strTableName**, the name of the table.

**Parameters:**

* **strClusteringKey** (String): The name of the clustering key.
* **strTableName** (String): The name of the table.

**Method Name:** ***handleOverflow***

**Serves:** ***insertIntoTable***

**Description:** The **handleOverflow** method is responsible for handling the overflow of tuples in a page within the specified table. It takes three parameters: **pageIndex**, the index of the page where the overflow occurred, **table**, the table object representing the table where the overflow occurred, and **primaryKey**, the name of the primary key. The **handleOverflow** method handles the overflow of tuples in a page within the specified table by moving the last tuple to the next page and updating the relevant index trees.

**Method Name:** ***compareToObject***

**Serves:**  All

**Description:** The **compareToObject** method compares two objects **Value** and **pageValue** and returns the result of the comparison as an integer. The method handles different data types, including **Integer**, **Double**, **String**, and **Date.**

**Method Name:** ***Writecsv***

**Serves: *createTable***

**Description:** The **Writecsv** method is responsible for writing the metadata of a table to the metadata CSV file. It checks if the table name is already used, and if not, it appends the metadata information to the CSV file.

**Method Name:** ***loopCSV***

**Serves:** All

**Description:** The **loopCSV** method is responsible for reading and parsing the metadata CSV file to retrieve the table names and their corresponding metadata.

**Method Name:** ***ReadTable***

**Serves:** All

**Description:** The **ReadTable** method is responsible for deserializing and retrieving a table object from a serialized file.

**Method Name:** ***WriteTable***

**Serves:** All

**Description:** The **WriteTable** method is responsible for serializing and storing a table object into a file.

**Method Name:** ***ReadPage***

**Serves:** All

**Description:** The **ReadPage** method is responsible for deserializing and retrieving a page object from a serialized file.

**Method Name:** ***WritePage***

**Serves:** All

**Description:** The **WritePage** method is responsible for serializing and storing a page object into a file.

**Method Name:** ***checkStringType***

**Serves:** All

**Description:** The **checkStringType** method is used to determine the data type of a given string value, primarily for identifying the data type of the clustering key value.

**Method Name:** ***checkDateFormat***

**Serves:** All

**Description:** The **checkDateFormat** method is used to validate whether a given date string is in the correct format.

**Method Name:** ***getPrimKeyType***

**Serves:** All

**Description:** The **getPrimKeyType** method is used to retrieve the data type of the primary key of a given table.

**Method Name:** ***getPrimKey***

**Serves:** All

**Description:** The **getPrimKey** method is used to retrieve the name of the primary key of a given table.

**Method Name:** ***checkColumns***

**Serves:** All

**Description:** The **checkColumns** method is used to validate if the columns specified in the **htblColNameValue** hashtable exist in the table.

**Method Name:** ***insertIntoIndex***

**Serves:** **createIndex**

**Description:** The **insertIntoIndex** method is used to insert the values from a table into an octree index.

**Method Name:** ***convertToDataType***

**Serves:** All

**Description:** The **convertToDataType** method converts a string value to its corresponding object type based on its data type.

**Method Name:** ***determineOctreeIndex***

**Serves:** **parseOctree**

**Description**: This method determines the index of the octree cell based on the given coordinates (x, y, z) and the midpoints (midX, midY, midZ) of the octree cell. It evaluates the relative positions of the coordinates and midpoints to determine the corresponding index, which ranges from 0 to 7. The method returns the computed index.

**Method Name:** ***findMedianValue***

**Serves:** All

**Description**: The **findMedianValue** method calculates the median value between the provided minimum (min) and maximum (max) values. It handles various data types such as strings, dates, integers, and doubles. For string values, it performs a lexicographic comparison and computes the middle string. For date values, it calculates the average date. For integer and double values, it computes the average of the two values. The method returns the resulting median value.

**Method Name:** ***findMinMaxValues***

**Serves:** **parseOctree**

**Description**: The **findMinMaxValues** method determines the minimum and maximum values for each dimension (X, Y, Z) of the octree cell. It takes an octree object and an index (i) as input. Based on the provided octree and index, the method computes and returns an array containing the minimum and maximum values for each dimension. The calculated values represent the boundaries of the corresponding octree cell.

**Method Name:** ***checkDup***

**Serves:** **parseOctree**

**Description**: This method checks whether a given **Point** object **p** already exists in a specified **Node** **r**.

**Method Name:** ***findIndexNul*l**

**Serves:** **parseOctree**

**Description**: The **findIndexNull** method determines the indices of child nodes in an Octree based on the provided coordinate values and midpoints.

**Method Name:** ***searchOctree***

**Serves:** All

**Description**: This method performs a search operation within an Octree to find references to **Point** objects that match a given **Point** **p**.

**Method Name:** ***deleteOctree***

**Serves:** **deleteFromTable/updateTable**

**Description**: This method is used to delete a specific point from an Octree data structure. It recursively searches for the point in the Octree and removes it if found. The method handles both leaf nodes and parent nodes in the Octree structure.

**Method Name:** ***updatePointRef***

**Serves:** **HandleOverFlow**

**Description**: The **updatePointRef** method is a helper method used to update a specific point in an Octree data structure. It recursively searches for the point in the Octree and updates its reference value if found. The method handles both leaf nodes and parent nodes in the Octree structure.

**Method Name:** ***parseOctree***

**Serves:** **InsertIntoTable/updateTable**

**Description**: The **parseOctree** method is a helper method used to parse and insert a point into an Octree data structure. The method handles both leaf nodes and parent nodes in the Octree structure.

**Method Name:** ***WriteIndex***

**Serves:** All

**Description**: The **WriteIndex** method is responsible for serializing an Octree object and writing it to a file. The serialized object represents the index data structure for a specific table.

**Method Name:** ***ReadIndex***

**Serves:** All

**Description**: The **ReadIndex** method is responsible for deserializing an Octree object from a file.

**Method Name:** ***CheckConditions***

**Serves:** InsertIntoTable/updateTable/deleteFromTable

**Description**: The **CheckConditions** method is a helper method that validates and checks various conditions for a given table and the provided values. It ensures that the provided values meet the specified requirements for each column.

**Method Name:** **init**

**Description**: this does whatever initialization needs to be.

**Required Methods**

**Method Name:** **createTable**

**Parameters:**

* **strTableName (String):** The name of the table.
* **strClusteringKeyColumn (String):** The name of the clustering key column.
* **htblColNameType (Hashtable<String, String>):** A hashtable containing the column names as keys and their corresponding data types as values.
* **htblColNameMin (Hashtable<String, String>):** A hashtable containing the column names as keys and their corresponding minimum values as values.
* **htblColNameMax (Hashtable<String, String>):** A hashtable containing the column names as keys and their corresponding maximum values as values.

**Method Flow**:

1. **Check** if the sizes of the minimum value hashtable and the maximum value hashtable are equal to the size of the column type hashtable. If not, throw a **DBAppException**.
2. **Write** the table information to the CSV file, including column names, types, clustering key column, minimum values, and maximum values.
3. **Create** an instance of the Table class with the specified table name.
4. **Write** the table object to the file.
5. If an exception occurs during the file writing process, print the stack trace and handle the exception.

**Method Name**: **createIndex**

**Parameters:**

* **strTableName (String):** The name of the table.
* **strarrColName (String[]):** An array containing the names of the columns to create an index on.

**Method Flow**:

1. **Check** if the length of the column name array is exactly 3. If not, throw a **DBAppException**.
2. **Create** the index name by concatenating the column names with "Index".
3. **Initialize** variables for the minimum and maximum values of each dimension.
4. **Retrieve** the list of table names from the CSV file.
5. **Iterate** over the table names.
   * **Check** if the current table name matches the specified table name and if the column name exists in the table.
   * **Throw** a **DBAppException** if an index already exists on the column.
   * **Update** the index and attribute information in the CSV file.
   * **Update** the minimum and maximum values based on the current column.
6. **Check** if all three columns were found in the table. If not, **undo** any partial changes made to the CSV file and throw a **DBAppException**.
7. **Update** the CSV file with the modified information.
8. **Create** an instance of the Octree class.
9. **Set** the minimum and maximum values and column names for the Octree.
10. **Write** the index tree to a file.
11. **Read** the table from the file.
12. **Add** the index name to the table's list of indices.
13. **Insert** the existing records into the index tree.
14. **Write** the updated table to the file.

**Method Name:** **insertIntoTable**

**Parameters:**

* **strTableName (String):** The name of the table.
* **htblColNameValue (Hashtable<String, Object>):** The hashtable containing column names as keys and corresponding values to be inserted.

**Method Flow:**

1. **Retrieve** a list of all table names in the database.
2. **Deserialize** the table with the given name.
3. **Check** if the provided column names exist in the table.
4. **Check** the conditions for inserting the row.
5. **Retrieve** the primary key for the table.
6. **Initialize** a Hashtable to store the row to be inserted.
7. **Check** the conditions for inserting the row.
8. **Read** the table from the file.
9. If the table has no pages:
   * **Add** a new page and serialize it.
   * **Read** the page.
   * For each index in the table:
     + **Read** the index tree.
     + **Parse** the octree with the provided column values.
     + **Write** the index tree.
   * **Add** the row to the page.
   * **Write** the page.
   * **Return** from the method.
10. If the table has pages:
    * **Retrieve** the value of the primary key.
    * **Set** a boolean variable 'done' to false.
    * **Iterate** over the pages in the table until 'done' is true.
      + **Read** the page.
      + **Check** if the value of the primary key falls within the range of the page.
      + For each index in the table:
        - **Read** the index tree.
        - **Parse** the octree with the provided column values and page number.
        - **Write** the index tree.
      + **Add** the row to the page.
      + **Write** the page.
      + **Set** 'done' to true if the page becomes full.
      + If there is an overflow:
        - **Handle** the overflow by calling the 'handleOverFlow' method.
    * If 'done' is still false:
      + **Read** the first page.
      + **Add** the row to the page.
      + **Write** the page.
      + **Retrieve** the last page.
      + **Add** the row to the page.
      + **Write** the page.
      + For each index in the table:
        - **Read** the index tree.
        - **Parse** the octree with the provided column values and last page number.
        - **Write** the index tree.
      + If there is an overflow:
        - **Handle** the overflow by calling the 'handleOverFlow' method.
11. **Return** from the method.

**Method Name**: **deleteFromTable**

**Description**: Deletes rows from a table based on the provided column values.

**Parameters**:

* **strTableName** (String): The name of the table to delete from.
* **htblColNameValue** (Hashtable<String, Object>): A Hashtable containing the column names and their corresponding values for the rows to be deleted.

**Method Flow**:

1. **Retrieve** a list of all table names in the database.
2. **Deserialize** the table with the given name.
3. **Validate** if the provided column names exist in the table.
4. **Check** if the provided column values match the specified conditions for deletion.
5. **Retrieve** the primary key for the table.
6. **Initialize** an ArrayList to store the page numbers where matching rows are found.
7. **Create** a Point object to store the values for the index tree search.
8. If the table has indices:
   * **Deserialize** the index tree for each index.
   * **Check** if the provided column values contain the indexed columns.
   * **Perform** an index tree search to find the page numbers that match the provided column values.
   * **Iterate** over the obtained page numbers and process each page.
   * **Iterate** over the tuples in the page.
   * **Check** if the column values match the provided values for deletion.
   * If a match is found:
     + **Delete** the tuple from the table.
     + **Remove** the tuple from the index tree.
     + **Update** and save the corresponding page.
     + If the page becomes empty, **delete** it from the table and **remove** the associated file.
9. If the table has a primary key and the provided column values contain the primary key:
   * **Perform** a binary search on the table to find the matching tuples.
   * If a match is found:
     + **Delete** the tuple from the table.
     + **Remove** the tuple from the index tree (if applicable).
     + **Update** and save the corresponding page.
     + If the page becomes empty, **delete** it from the table and **remove** the associated file.
10. If the table does not have an index or the provided column values do not contain any indexed columns:
    * **Perform** a linear search on the table.
    * For each page and tuple, **check** if the column values match the provided values for deletion.
    * If a match is found:
      + **Delete** the tuple from the table.
      + **Remove** the tuple from the index tree (if applicable).
      + **Update** and save the corresponding page.
      + If the page becomes empty, **delete** it from the table and **remove** the associated file.
11. End the method.

**Method Name:** **updateTable**

**Parameters:**

* **strTableName** (String): The name of the table to update.
* **strClusteringKeyValue** (String): The value of the clustering key for the row to update.
* **htblColNameValue** (Hashtable<String, Object>): A hashtable containing the column names and their corresponding updated values.

**Method Flow:**

1. **Retrieve** the list of table names from the database.
2. **Get** the primary key of the specified table.
3. **Check** if the columns in the update values exist in the table.
4. **Convert** the clustering key value to the appropriate data type based on the primary key's data type.
5. If the primary key data type is "java.util.Date", **check** if the clustering key value is in the correct date format.
6. **Perform** additional checks and validations on the specified table, clustering key, and update values.
7. **Create** new empty hashtables for the new row and old row.
8. **Set** the initial page reference to 0.
9. **Read** the table file from disk by deserializing it.
10. If the table is empty (noOfPages = 0), **throw** a DBAppException indicating that the table is empty.
11. **Declare** variables for page index, found flag, and an ArrayList to store page references.
12. **Check** if the clustering key is indexed.
13. If indexed, **perform** an octree search to find the row with the specified clustering key value.
14. For each page reference found, **iterate** through the tuples in the page. a. **Use** binary search to find the tuple with the matching clustering key value. b. If a match is found, **update** the old row information and delete the old point from the octree. c. **Update** the tuple with the new values from the htblColNameValue hashtable. d. **Update** the new row information and update the octree with the new point. e. **Write** the updated index, page, and table files back to disk. f. **Break** the loop as the update is complete.
15. If not indexed, **iterate** through each page in the table. a. **Check** if the clustering key value falls within the range of the current page. b. If within range, **perform** binary search to find the tuple with the matching clustering key value. c. If a match is found, **update** the old row information and update the tuple with the new values. d. **Write** the updated page back to disk. e. **Break** the loop as the update is complete.
16. **Iterate** through each index in the table and update the corresponding octree based on the updated values.
17. If no matching row is found, **throw** a DBAppException indicating that the primary key was not found.
18. **Clean up** variables and perform garbage collection.

**Method Name: *selectFromTable***

**Description:**

The **selectFromTable** method performs a select operation on a table based on the provided search conditions. It takes two parameters: **arrSQLTerms**, an array of SQLTerm objects representing the search conditions, and **strarrOperators**, an array of strings representing the operators between the search conditions.

**Parameters:**

* **arrSQLTerms (SQLTerm[]):** An array of SQLTerm objects representing the search conditions for the query.
* **strarrOperators (String[]):** An array of string operators corresponding to the SQL terms, specifying the logical operations to apply between the terms.

**Method Flow:**

1. Check if the number of search terms minus 1 is equal to the number of operators. If not, throw a DBAppException with the message "compilation error".
2. Get the table name from the first SQLTerm in **arrSQLTerms** and store it in **SingleTableName**.
3. Create an empty vector called **result** to store the selected tuples.
4. Create an ArrayList called **colNames** to store the column names.
5. Initialize the variable **i** to 0.
6. Enter a while loop that iterates until **i** is less than the length of **arrSQLTerms**.
7. Retrieve the current SQLTerm from **arrSQLTerms** at index **i** and store it in the variable **condition**.
8. Invoke the **loopCSV** method to retrieve a list of tables.
9. Set a boolean variable **valid** to false.
10. Iterate over each table name in the **tablesName** list.
    * Check if the current table name, column name, and value type match the values in the **tablesName** list.
    * If they match, set **valid** to true and break the loop.
11. Check if **valid** is true.
    * If true, add the column name from **condition** to the **colNames** list.
    * If false, throw a DBAppException with the message "error while validating!!".
12. Increment **i** by 1.
13. Invoke the **getIndexName** method to retrieve the index name based on the **colNames** list and **SingleTableName**. Store the result in the variable **Index**.
14. Check if **Index** is equal to "linear Search" or if not all operators in **strarrOperators** are "AND".
    * If true, invoke the **executeComplexQuery** method to perform a complex query and store the result in the **result** vector.
    * If false, retrieve the Octree object for the index using the **ReadIndex** method and store it in the variable **tree**.
      + Create an empty ArrayList called **reff** to store the page references.
      + Iterate over each SQLTerm in **arrSQLTerms**.
        - Invoke the **getPagesByRange** method to retrieve the page references based on the SQLTerm, the **tree**, and **SingleTableName**.
        - Add the page references to the **reff** ArrayList if they are not already present.
      + Invoke the **performQueryWithIndex** method to perform the query using the page references, **SingleTableName**, **arrSQLTerms**, and **strarrOperators**, and store the result in the **result** vector.
15. Return an iterator to iterate over the selected tuples.

**Possible Future Enhancements:**

*There are several potential avenues for future enhancements and improvements to the database engine project. Some suggestions include:*

-Implementation of additional indexing methods, such as B-trees or hash-based indexing, to further optimize query performance.

-Integration of advanced query optimization techniques to improve the execution plans and reduce query processing time.

-Incorporation of security features like authentication and encryption to ensure data privacy and protection.

-Development of a user-friendly interface or API for seamless integration with different application environments.