# LightDB

> **Student Number:** S2053346
>
> This is a README file for LightDB, which is a simple interpreter for SQL statement.

## Some Data Structures

### 1. `Tuple`

**Description:** Tuple is the class designed for storing data that comes query. The values in a tuple is unchangeable. Tuple class has a final type List, which stores the values of tuple.

**Constructor:** A List type `args` should be passed into the constructor. The `args` contains the values of a tuple.

**Method:**

- `toString()`: Change tuple to a String.
- `toList()`: Create a List, and copy the value from the tuple and then return.
- `get(int index)`: Get the value in the tuple by index.
- `concate(Tuple<T>, tuple)`: Glue this tuple with another tuple, return a new tuple that glued together.
- `compareValues(Tuple tuple)`: compare values in two tuples. If all values are the same, return true, otherwise return false.

### 2. `TableInfo`

**Description:** `TableInfo` is designed for  storing the database table information. Contains table name (`tableName`), disk path of the table (`tablePath`), columns of the table (`columns`) and the flag that whether a table is a temporary table (`inMemory`). When `inMemory` is set true, it means that the table is not really in the disk, but comes from join operation or projection operation.

### 3. `Catalog`

**Description:** `Catalog` stores the information of the database and its tables. `Catalog` is designed as a singleton pattern, which shares globally. The following attributions are contained in `Catalog`:

```
String dbPath: the path of the database
Map<String, TableInfo> tables: a hashMap that contains all table information in
the database.
```

**Constructor:** A `String` type `dbPath` should be passed into the constructor. During the initialisation, tables in the database will be loaded automatically.

**Method:**

- `getIndex(String tableName, String columnName)`: Given `tableName` and `columnName`, find out the column index in the tuple and return.

- `getTable(String tableName)`: Given a `tableName`, find out the `tableInfo` instance and return.
- `dropInMemoryTable()`: Clear the temporary table info, this should be done after a SQL has executed.
- `getInstance(String dbPath)`: Get the singleton instance of `Catalog`.

---

# Join Logic

## 1. Extract Join Conditions

Class `ExpressionVisitor` is designed for extracting filter conditions of tables and join conditions of joint tables. `ExpressionVisitor` is a subclass of `ExpressionDeParser`. It extracts conditions by visiting where clause expression.

Class `ExpressionVisitor` maintains a `expMap` which is a `LinkedHashMap<String, ArrayList<Expression>>` . The `LinkedHashMap` keeps the order of the keys, which could ensure left deep join. The key of `expMap` is table name or joint table name. For a single table `S`, its key in `expMap` is `"S"`. For two tables `T1` and `T2`, if they join together, the joint table name will be `"T1 T2"`. Specially, all numeric conditions will be kept in the `List` that corresponds to the key `"numeric"`. `ExpressionVisitor` also has an attribute `tableOrder`, which is a `String` list, contains the left deep join table order extracted from `from` clause.

When `ExpressionVisito` instance is initialised, it will call `buildExpMap()` to construct a left-deep-join form `expMap`. If there are $k$ tables, there will be $2k-1$ table (or joint table) keys in `expMap`, and 1 `numeric` keys. The `expMap` is generated following the pseudocode shown below. After calling the `buildExpMap()`, all potential left-deep joint tables are created.

```
void buildExpMap():
   tableName = tableOrder[0]  // the first table name in the table order list
   expMap.put(tableName, emptyList)  // put the table name and empty list into
the expMap
   for 1:tableOrder.length:
       expMap.put(tableOrder[i], emptyList)  // the single table and empty list
       tableName = tableName + " " + tableOrder[i] // create potential joint
table
       expMap.put(tableName, emptyList)
```

Method `putExp(ComparisonOperator operator)` is to put the expression into the corresponded Expression List in `expMap`. The `operator` has left part and right part. If the left part and right part are both involves tables, then it is a join condition. Concatenate two table name as a join table name, traverse the `expMap` 's key set from begin to end (key set is in order), if find a key that contains the join table name (such as `"T1 T2 T3"` contains `"T2 T3"`), break the loop and add the expression into the corresponded list in `expMap`. Since `expMap` is a ordered `LinkedHashMap`, the situation that the expression for `T1 T2` is wrongly added into the list which corresponds to `"T1 T2 T3"` will not happen. If only one of left part and right part involves table, this is a single table filter condition, directly put the expression into the corresponding list. If the expression is numeric expression such as `1=2`, put it into the numeric list.

All `visit` methods that involve subclasses of `ComparisonOperator` are overridden. when these methods are called, `putExp` will be called too, and put the expression to the proper list in `expMap`.

The `getExpression` method will use this visitor to traverse `Expression` `exp`. After traverse the `exp`, subexpression will be put into the `expMap`. `getExpression` will integrate the expressions that correspond to one same table together, and put them into a `HashMap<String, Expression>` `hashMap` to return.

## 2. Join Operation

The join operation is rely on the join conditions. As described in the last section, if we want to get `T1` and `T2`s' joint condition expression, we can use the `getExpression` method in `ExpressionVisitor` and take out the expression in the return HashMap, where the corresponding key is `"T1 T2"`.

When the `JoinOperator` is opened, it will read one tuple `outerTuple` from his left child, which is the start point of outer loop. The inner loop is for reading tuple from the right child. While the `outerTuple` is not `null`, the inner loop will start. Inner loop first glue `outerTuple` and `innerTuple` together, and check whether it satisfies the join condition. If it does, return the glued tuple, otherwise, keep looping.  When the inner loop is finished, but the outer loop is not finished, reset the right child and read next `outerTuple`. The pseudocode is shown below.

```
// when the operator is opened...
outerTuple = leftChild.getNextTuple()
....

Tuple getNextTuple():
    while outerTuple is not null:
        innerTuple = rightChild.getNextTuple()
        while innerTuple is not null:
            tempTuple = glue(outerTuple, innerTuple)
            if checkCondition(tempTuple) is true:
                return tempTuple
        rightChild.reset()
        outerTuple = leftChild.getNextTuple()
    return null
```