Fall 2013	**CSC 468:** DBMS Implementation	Alexander Dekhtyar

Project Stage 2 Part 1: Relational Algebra and DDL/DML layers

Due date: Wednesady, May 25, 11:59pm.

Overview

Stage 2

Stage 2 of the project sets up the query processing and SQL command implementation in FLOPPY. The key components of this stage are:

- Relational Algebra query execution layer.
- SQL DDL/DML command implementation layer.
- Implementation of volatile index-only data storage.
- FLOPPY server.

This document contains a quick overview of each of the parts of Stage 2, and the specification for the Relational Algebra and SQL DDL/DML processing layers.

FLOPPY Stage 2 Components

Relational Algebra query execution layer. FLOPPY shall support a subset of ANSI SQL with a few minor modifications called FLOPPY-SQL. The FLOPPY-SQL specification has already been released. The `SELECT` statement in FLOPPY-SQL is designed to support the following relational algebra operations[1]:

- *Selection* (*sigma*)
- *Projection* (π)
- *Product* ($\times$)
- *Join* ($\bowtie$) (note: limited version)

[1] We consider *limit*, a.k.a. $\lambda_k(R)$ to be a relational algebra operation. More on its semantics in the Relational Algebra component description.

- *Grouping and aggregation* (γ)
- *Sorting* (τ)
- *Limit* (λ)

SQL DDL/DML command implementation layer. This component shall implement individual SQL DDL/DML commands:

- `CREATE TABLE`
- `DROP TABLE`
- `CREATE INDEX`
- `DROP INDEX`
- `INSERT`

During Stage 3 of the project you will also implement `UPDATE` and `DELETE` commands (they require query compilation as they contain the execution of the `WHERE` clause). For the time being, you will implement a simple

```
DELETE FROM <TableName>;
```

command to be able to delete all tuples from a given table.

Implementation of volatile index-only data storage. FLOPPY has one and a half tricks up its sleeve when it comes to managing some data in volatile storage. The trick is storing data in volatile storage in index structures only. The remaining half-trick is splitting the primary key when doing this.

Tables selected for storage in index-only fashion will be stored in volatile storage as a B+-tree on the primary key of the table. Leaf nodes need to be modified to allow for storing the contents of entire tuples (alternatively, you can make other decisions for how to store the data).

Tables that have compound primary key can be stored in volatile storage format in a *split index*. A *split index* is a B+-tree index of a data table that indexes at the top level, the first attribute of a primary key, and has leaves of that B+-tree point to the roots of B+-trees indexing the remaining attributes of the primary key.

The implementation of this part of FLOPPY and the use case for it will be presented to you in a separate handout.

FLOPPY Server. FLOPPY will run as a server accepting `localhost` connections on a specific port. The connections shall deliver the FLOPPY-SQL commands to FLOPPY. FLOPPY server, in turn, shall return back the results of the queries and acknowledgements of updates.

Relational Algebra Implementation.

As mentioned above, FLOPPY shall implement the following subset of relational algebra operatiaons.

- *Selection* (σ)
- *Projection* (π)
- *Product* ($\times$)
- *Join* ($\bowtie$) (note: limited version)
- *Grouping and aggregation* (γ)
- *Sorting* (τ)
- *Limit* (λ)

Relations as sets. FLOPPY shall take the view that any relational table stored in it is a **set of records**. This means that FLOPPY shall tolerate no duplicate records in its data tables. An attempt to insert a duplicate record into any FLOPPY table shall gracefully refuse to insert a new record.

Projection operation can create duplicate records in the output. The default behavior of the `SELECT` statement is to not eliminate duplicates in the projection operation. FLOPPY-SQL `SELECT` statement does come with a `DISTINCT` keyword, which, when used, shall trigger *duplicate elimination*. While we do not list *duplicate elimination* explicitly in the list of operations above, please be aware that you may need to implement it. See more on this in the discussion of the *projection* operation.

Temporary tables. Each relational algebra operation will have as its one or two operands, relational tables. The relational tables can be either

- Database tables residing in persistent of volatile storage. These tables have been created with a `CREATE TABLE` statement and their contents persist beyond the execution of a single query.
- Temporary tables. A temporary table is created during the execution of a single `SELECT` statement consisting of multiple relational algebra operations to pass the results of one operation to the next operation in the query plan. A temporary table seizes to exist together with all its assets (both on disk and in the buffer) after the execution of a single query is concluded.

The management of temporary tables is the prerogative of the `eval()` function that you will implement on Stage 3 that will execute a given query plan. However, since the temporary tables are created to host the results of a relational algebra operation, their creation is part of the code base you are implementing now. For now, assume the following about the temporary files:

- Unique Identification. All temporary tables created to execute a single `SELECT` statement must have unique identification.
- Reuse. If a relational algebra operation can reuse a temporary table (e.g., sort in-place), you may do it, but it is not required.
- Identification. All temporary tables shall be accessible in the same way as regular database tables. They will be implemented as tinyFS files and therefore will have tinyFS `fileDescriptor`s. You may want to implement a function that given a tinyFS `FileDescriptor` checks if it is a temporary table or a regular table.

With these notes in mind, let us look at the individual Relational Algebra operations FLOPPY needs to implement.

Relational Algebra Operation outputs. While this might not be the fastest way around it, it is probably the simplest to implement. All relational algebra operation implementations will put the output of an operation into a *temporary table* and return the output table credentials *rather than the actual results of the operation*. This makes "hooking" relational algebra implementation functions together relatively straighforward.

Expressions. One of the inputs to *Selection* and *Join* operation implementation functions is an expression specifying the selection/join condition. In the specifications of the operations, we assume the existence of a data type `Expr` storing the information about the expression. We discuss the `Expr` data type below.

Selection

You shall implement at least two *Selection* operations:

- `int selectScan(fileDescriptor inTable, Expr condition, fileDescriptor * outTable)`: *Selection* operation via a table scan.
- `int selectIndex(fileDescriptor inTable, Exprt condition, fileDescriptor index, fileDescriptor * outTable)`: *Selection* operation via an index scan.

`int selectScan(fileDescriptor inTable, Expr condition, fileDescriptor * outTable)` . This function takes as input the following parameters:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable`	in	input table
`Expr condition`	in	selection condition
`fileDescriptor * outTable`	out	output temporary table

This function shall perform a **one-pass full-table scan**. As it proceeds to scan the tuples of the input table, the function shall evaluate the provided expression for each tuple, and for any tuple that satisfies the expression, put it into the output temporary table.

`int selectIndex(fileDescriptor inTable, Exprt condition, fileDescriptor index, fileDescriptor * outTable)` . This function takes as input the following parameters:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable`	in	input table
`Expr condition`	in	selection condition
`fileDescriptor index`	in	index to be used in the scan
`fileDescriptor * outTable`	out	output temporary table

This function shall perform an index-based selection operation. If the index provided to the function indexes one of the attributes used in the selection condition, the function scans the index, and for each tuple retrieved through the index scan checks the condition. If the selection condition does not "touch" the index, then the index scan falls back onto `selectScan()` (simply calls it with the appropriate input parameters).

Projection

As mentioned above, both *set projection* (with duplicate elimination) and *bag projection* (without duplicate elimination) may be needed, as `SELECT` statement in FLOPPY-SQL comes with the optional `DISTINCT` keyword. The two *Projection* functions are as follows:

- `int project(fileDescriptor inTable, attList * attributes, fileDescriptor * outTable)`: Projection without forced duplicate elimination.
- `int projectDistinct(fileDescriptor inTable, attList * attributes, fileDescriptor * outTable)`: Projection with forced duplicate elimination.

`attList.` Projection operations take as input a list of attributes to be projected. We use `attList` to refer to an object of this type. Create and maintain the appropriate data type in your codebase.

`int project(fileDescriptor inTable, attList * attributes, fileDescriptor * outTable)` : Projection without forced duplicate elimination. This function takes the following parameters:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable`	in	input table
`attList * attributes`	in	list of attributes
`fileDescriptor * outTable`	out	output temporary table

The function shall work in a straightforward manner. Scan the input table, for each tuple in the table, perform the projection operation by limiting the content of the output tuple to the attributes specified in the `attributes` parameter, output the tuple to the output temporary table.

`int projectDistinct(fileDescriptor inTable, attList * attributes, fileDescriptor * outTable).` Projection with forced duplicate elimination. This function is more intricate. It takes the same parameters as `int project()`. However, because it needs to perform duplicate elimination, this function can no longer proceed as a simple table scan.

You can perform this operation in one of two ways: either call `project()` and then implement a duplicate elimination operation on the output, or implement a multi-pass set projection operation (you are allowed to implement only one way of achieving this goal). The choice is left up to you.

Limitations. Compared to standard relational algebra *projection* operation, FLOPPY projection has a number of limitations. In particular:

1. The only elements of the `attributes` list that are allowed are
   - Attributes
   - Aggregations

   Please note, aggregations are only allowed when projection is executed upstream of a grouping/aggregation operation, in which case, the tuples in the temp table provided as input to the projection function will contain columns representing the appropriate aggregates. The projection operation needs to be able to understand how to find the necessary aggragate columns in the records of the temp table. The projection operation itself is not responsible for performing any computations.
2. Unsupported by FLOPPY are
   - Columns that are constructed as (arithmetic) expressions based on other columns. FLOPPY Version 1 includes no built-in functions, hence no built-in functions shall appear in the elements of projection lists.
   - "Constant" columns (i.e., columns that have a single constant value for all tuples)

Cartesian Product

To ensure that FLOPPY won't choke on queries of the form

```
SELECT *
FROM X, Y;
```

you need to implement a nested-blocks version of the *Cartesian Product* operation. The function definition is:

```
int product(fileDescriptor inTable1, fileDescriptor inTable2, fileDescriptor * outTable);
```

The parameters are:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable1`	in	first input table
`fileDescriptor inTable2`	in	second input table
`fileDescriptor * outTable`	out	output temporary table

The implementation is pretty straightforward. The hope is that this function is not called often, but it needs to be there in case someone issues a query that misses a join condition.

Join

The biggest limitations of FLOPPY (besides not implementing certain operations at all) come from the requirements on the *Join* operation. Specifically, we only allow for one type of joins in FLOPPY: *equijoins based on conjunction of equality join conditions*. That is the only join conditions that are allowed for a join between two tables, R and S are of the form:

$$R.A1 = S.B1 \wedge \ldots \wedge R.Ak = S.Bk$$

Having said that, because of the importance of efficiency of join for the overall success of a DBMS, you will implement multiple versions of *join* operation. Below is the list of *Join* functions for the relational algebra module of FLOPPY:

- `int joinOnePass(fileDescriptor inTable1, fileDescriptor inTable2, Expr condition, fileDescriptor * outTable)`: one-pass join. Can only be called if one of the tables fits in the buffer.
- `int joinMultiPass(fileDescriptor inTable1, fileDescriptor inTable2, Expr condition, fileDescriptor * outTable)`: multi-pass join. Can be either sort-based or hash-based[2]
- `int joinNestedLoops(fileDescriptor inTable1, fileDescriptor inTable2, Expr condition, fileDescriptor *outTable)`: block-nested loops join - FLOPPY's catch-all join implementation.

In addition, you may choose to implement a zig-zag join or another index-based join algorithm (this might be useful when dealing with joins that involve volatile index-only tables). However, we leave this off the current specification for now.

The parameters of all join functions are:

[2]In present form, FLOPPY has more built-in support for sort-based operations.

Parameter	Parameter Type	Meaning
`fileDescriptor inTable1`	in	first input table
`fileDescriptor inTable2`	in	second input table
`Expr condition`	in	join condition (see limitations above)
`fileDescriptor * outTable`	out	output temporary table

Algorithms for all join operations have been discussed in class. Some notes are below:

`int joinOnePass(fileDescriptor inTable1, fileDescriptor inTable2, Expr condition, fileDescript`
`* outTable).` For the one-pass join operation, make sure you select the smaller table to place in main memory, while using the larger table for the table scan.

`int joinMultiPass(fileDescriptor inTable1, fileDescriptor inTable2, Expr condition,`
`fileDescriptor * outTable).` As noted in the footnote, sort-based joins are probably easier to implement given the Stage 1 functionality (namely, FLOPPY is supposed to support sequential files, but we did not require support for hashed files). Because 2-pass, 3-pass, etc.. joins are essentially the same (differ from each other in the number of merge-sort steps), a single multi-pass algorithm implementation can cover both the 2-pass and the 3-and-above pass versions of the algorithm.

`int joinNestedLoops(fileDescriptor inTable1, fileDescriptor inTable2, Expr condition,`
`fileDescriptor *outTable).` Make sure you use the smaller table to break into blocks, while scanning the larger table.

Note. Because we only allow equijoins, one of the issues related to non nested-loops join operations - namely the possibility that a subset of the two tables larger than the buffer needs to produce a "cartesian product" as part of the join operation essentially goes away (equijoins are expected to have a reasonably high selectivity. While it is possible to have a lot of records in each of the table with the same value of the join attributes, the workloads for FLOPPY are not going to be like that). Because of this, you can always assume in your `joinMultiPass()` implementation that all tuples that need to be joined to each other can fit in main memory at any moment of time. This should save you some code.

Grouping and Aggregation

You will implement two functions for grouping and aggregation:

- `int groupOnePass(fileDescriptor inTable, attList * group, attList * aggregates,`
  `fileDescriptor * outTable)`: one-pass grouping/aggregation. Can only be called if the table fits in the buffer.

- `int groupMultiPass(fileDescriptor inTable, attList * group, attList * aggregates, fileDescriptor * outTable)`: multi-pass grouping/aggregation. Can be sort-based or hash-based.

The parameters for these functions are:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable`	in	input table
`attList * group`	in	sequence of grouping attributes (from `GROUP BY`)
`attList * aggregates`	in	Aggregates to compute
`fileDescriptor * outTable`	out	output temporary table

The `attList * group` parameter contains an ordered list of attributes that are used for grouping the table. These are the attributes found in the `GROUP BY` clause of the `SELECT` statement.

The `attList * aggregates` contains the list of aggregate quanities that need to be computed by the grouping operation. These come from two sources:

- The `SELECT` clause of the `SELECT` statement — these aggregates will be returned as part of the query output.
- The `HAVING` clause of the `SELECT` statement — these aggregates, if not found in the `SELECT` clause are needed only for the selection operation prescribed by the `HAVING` clause.

Aggregation-only runs. In some cases, the `attList * group` list may be null. This represents the situation when a FLOPPY-SQL query includes aggregation operations in the `SELECT` clause, but does not include `GROUP BY` clause. In this case, the grouping functions shall gracefully resort to computing overall aggregates over the input relation.

Each team can determine who it wants to represent aggregates in the `attList` element values.

Aggregates Supported. The following list of aggregate operations shall be supported:

- `COUNT(<Attribute>)`. Shall return the total number of values in the specified column **that are not NULL**. Can be applied to any column.
- `COUNT(*)`. Shall return the total number of tuples in the group.
- `SUM(<Attribute>`. Applicable only to numeric columns. Computes the sum of values in the column.
- `AVG(<Attribute>`. Applicable only to numeric columns. Computes the average value for the column.
- `MIN(<Attribute>`. For numeric attributes returns the smallest value in the column. For `DATETIME` values returns the earliest value in the column. For strings returns the smallest value in the column based on lexicographic sort order.

- `MAX(<Attribute>`. For numeric attributes returns the largest value in the column. For `DATETIME` returns the latest value in the column. For strings returns the largest value in the column based on lexicographic sort order.

Note. For simplicity we omit the `COUNT(DISCTINCT <Attribute>)` aggregate from the FLOPPY requirements. You can choose to implement it, and the appropriate SQL syntax to support this operation.

Sorting

To react properly to the `ORDER BY` clauses in the `SELECT` statements FLOPPY shall implement a sort operation.

You can implement just a single sorting function that uses merge-sort to sort the input table in the necessary number of passes. Because the *segment-sort* step (first pass) of the multi-pass sorting sorts a single segment that fits main memory, a faithful implementation can cover one-pass, two-pass and multi-pass sorts in a single function in a straightforward way.

The sort function shall look as follows:

```
    sortTable(fileDescriptor inTable, attList * attributes, fileDescriptor outTable);
\end{verbastim}

The parameters are:


\begin{tabular}{lll}
\hline
\textsf{Parameter} & \textsf{Parameter Type}& \textsf{Meaning} \\
\hline
\texttt{fileDescriptor inTable} & \textsf{in} &input table\\
\texttt{attList * attributes} & \textsf{in} & list of attributes on which to sort\\
\texttt{fileDescriptor * outTable} & \textsf{out} & output temporary table\\
\hline
\end{tabular}

\paragraph{Sort order.} You shall instrument the \texttt{attList} data type so that
it is possible to pass the \textit{sort order flag} for each attribute into the \texttt{sortTa
There are two sort orders: ascending (default) and descending. It is your choice how to engine
but it shall be possible to infer the sort order on each attribute from the passed pointer.

\subsubsection*{Limit}
```

```
For \textsf{FLOPPY} Relational Algebra we introduce an additional operation: $\lambda_k(R)$ - \

While all other relational algebra operations assume that their operand tables are sets, and th
the \textit{limit} operation assumes that the input table comes as specific serialization of th
a \textbf{sequence} of tuples.}

The semantics of the $\lambda_k(R)$ is straightforward: given a sequence of tuples as input, it
tuples, and ignores all remaining ones. This operation shall be implemented in \textsf{FLOPPY}
following function:

\begin{verbatim}
  int limitTable(fileDescriptor inTable, int k, fileDescriptor * outTable);
```

The parameters are:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable`	in	input table
`int k`	in	number of tuples to return
`fileDescriptor * outTable`	out	output temporary table

FLOPPY-SQL DDL/DML Module

The second key part of Stage 2 of the project is implementation of the FLOPPY-SQL DDL and DML commands. In Stage 2 we omit the implementations of `DELETE` and `UPDATE` commands - their `WHERE` clauses require passing these commands through the query optimizer/compiler to select an apprpriate physical query plan for them. Stage 2 shall concentrate on `CREATE TABLE`, `CREATE INDEX`, `DROP TABLE`, `DROP INDEX` and `INSERT` commands, with `DELETE FROM <Table>` stub also implemented to allow you to clean up tables.

CREATE TABLE

Implement the following function to create a new table:

```
int createTable(char * tableName, parseTree * command, fileDescriptor * FD);
```

The parameters of this function are:

- `char * tableName` (in parameter): the name of the table to create.
- `parseTree * command` (in parameter): the command to create the table.
- `fileDescriptor * FD` (out parameter): tinyFS descriptor of the data file.

parseTree data type. I use `parseTree` data type in this and some other functions as a stand-in for the result of parsing a FLOPPY-SQL command. This is **not** a query tree/relational algebra tree produced by the query compiler. Rather, this is simply the representation of the parsed input command.

For most other commands, it is easier to pre-parse the input parameters, but because `CREATE TABLE` is so vast and complex, I choose to pass the parsed command into this function directly, and have this function manage the recognition of what needs to be done.

The basic steps to implement are:

1. Parsing of the command. Traverse the parse tree and extract from it the following information:
   - list of attributes together with their types
   - primary key
   - list of foreign keys (if any)
   - any modifiers related to storing the table in volatile storage (`VOLATILE`, `INDEX ONLY`, `SPLIT`)
2. Check if table exists. If the table with the given name does not exist, proceed.
3. Create the table. Create the data file to store the table data. Modify all appropriate internal data structures to include the new table into the list of tables in the database.
4. Create indexes. Create the primary key index file and any foreign key index files as needed.
5. Special handling. If the table is to be stored as index-only (full index, or split-index format) in volatile storage, create the appropriate index file structures instead of the regular data file.

CREATE INDEX command

The `CREATE INDEX` command shall be supported by the following function:

```
int createIndex(fileDescriptor inTable, char * indexName, attList  * attributes, int indexTy
```

The paramters for this function are:

Parameter	Parameter Type	Meaning
`fileDescriptor inTable`	in	table to be indexed
`char * indexName`	in	number of tuples to return
`attList * attributes`	in	list of attributes to be indexed
`int indexType`	in	a flag indicating the type of the index
`fileDescriptor * indexFile`	out	file descriptor of the new index file

If you need, feel free to pass other parameters into this function.

In addition to supporting the `CREATE INDEX` statement, this function can also support automated index creation calls from the `CREATE TABLE` statement implementation.

Index Type. The `int indexType` attribute can have the following values:

Value	Index type
1	secondary index supporting a table stored in a data file
2	volatile primary index used to store a relational table (triggered by the `INDEX ONLY` keyword)
3	volatile split primary index (triggered by the `SPLIT` keyword)

Index-only data file implementation. Index-only data files can be created using the `createIndex()` function, but it may require some care. The `indexType` flag will pass the index type. The `attributes` parameter in this case shall be a full description of the relational table, including information about the primary key, which is what is used to index the tuples in the table[3] The Index-Only and Split Index Storage of Data handout covers the details of the implementation of index-only and split index storage.

DROP TABLE command

`DROP TABLE` command and its semantics is described in the FLOPPY-SQL handout. The command is to be implemented using the following function

```
int dropTable(fileDescriptor table);
```

The `fileDescriptor table` is an in parameter specifying the table to be deleted.

The effects of this function shall be as described in the semantics of the `DROP TABLE` command in the FLOPPY-SQL handout.

DROP INDEX command

`DROP INDEX` command and its semantics is described in the FLOPPY-SQL handout. The command is to be implemented using the following function

```
int dropTable(fileDescriptor table, char * index);
```

The `fileDescriptor table` is an in parameter specifying the table whose index is to be deleted, which the `char * index` is the name of the index to be deleted. (You can also have a version of this function that uses a `fileDescriptor index` input parameter if you feel this is more straigh-forward).

The effects of this function shall be as described in the semantics of the `DROP INDEX` command in the FLOPPY-SQL handout.

[3] You may need to recast the pointer in this case to a more appropriate type.

INSERT command

The FLOPPY-SQL `INSERT` command shall be implemented using the following function:

```
int insertRecord(fileDescriptor table, char * record);
```

Here, `table` is the table into which the insertion is to take place, and `char * record` is the in parameter specifying the contents of the record to be inserted (as a byte array).

Note: We rely on the record to be built from the input `INSERT` command by some other function. If you wanted to, you could replace `char * record` with `parseTree * command` and form the record inside the `insertRecord()` function.

This function has the limitations described in the FLOPPY-SQL handout. In particular, it only supports insertion of a **single tuple**, and the tuple must be fully specified in the `INSERT` command (all values in order of the attribute specifications in the CREATE TABLE schema for the table).

Good Luck!