CSC 468 DBMS Organization
Spring 2016
Project, Stage 2, Part 2

FLOPPY SQL

This document specifies the version of SQL that FLOPPY must support. We provide the full description of the FLOPPY-SQL syntax. In most cases, FLOPPY-SQL statements maintain the semantics of ANSI SQL statements. Where we introduce modifications, we explain their semantics in detail.

1. Data Definition Language.

FLOPPY-SQL DDL consists of the following statements:

```
1. CREATE TABLE
2. DROP TABLE
3. CREATE INDEX
4. DROP INDEX
```

1. **`CREATE TABLE`** Statement. The CREATE TABLE statement in FLOPPY-SQL has the following syntax (it is a slightly modified version of the statement described to you in a previous handout):

```
CREATE TABLE <TableName> [VOLATILE[, INDEX ONLY[, SPLIT]]](
      [<AttributeName> <AttributeType>,]+
      PRIMARY KEY (<AttributeName>,...,<AttributeName>)[,
      FOREIGN KEY (<AttrtibuteName,...,<AttributeName>)
                                    REFERENCES <Table>]*
 );
```

- There are only two types of constraints: Primary Key constraints and Foreign Key constraints. Both have to be specified after the list of attributes. Primary Key constraint is mandatory, one or more Foreign Key constraints may also be specified.

- There is only one database. All tables in the database must have unique names.
- Tables referenced in Foreign Key constraints must already exist.

Semantics.

CREATE TABLE command results in a table with a given name created in the FLOPPY database. In addition, for all tables stored in persistent storage (no VOLATILE keyword used) a B+-tree index is created on the Primary Key of the table and a B+-tree index is created for every Foreign Key.

For tables created in the volatile storage see specifications below.

Additional Functionality.

- `VOLATILE` keyword. If the CREATE TABLE statement contains the `VOLATILE` keyword, the table is created in the volatile storage according to the rules identified in Stage 1 of the project. In this keyword is specified and no other keywords are present, only the data table is created in the data storage. Volatile storage tables are not indexed unless other keywords are specified, or a CREATE INDEX command is run.

- `INDEX ONLY` keyword. This keyword is only available when `VOLATILE` keyword is specified. If this keyword is used, the table will be stored in volatile storage as B+tree index structure. See **additional Stage 2 documentation** for the specification of index-structure-only tables.

- `SPLIT` keyword. This keyword is only available when `VOLATILE` and `INDEX ONLY` keywords are specified. This keyword only has effect when the Primary Key of the table is compound (i.e., has more than one attribute). If this keyword is used and the Primary Key is compound, then the B+-tree used to store the data file in the volatile storage is split hierarchically - the "outer" B+-tree indexes the first attribute of the Primary key, while the inner B+-trees index the second (and all other) attributes of the Primary Key.

  There are only two layers to a SPLIT index - the first layer for the first Primary Key attribute, and the second - for all other attributes combined. (Usually, this option is reserved for storing tables with two-attribute primary keys, where the first attribute does not have too many values, but the second does). See **additional Stage 2 documentation** for the specification of split indexes.

2. DROP TABLE statement. The DROP TABLE statement is straightforward:

```
DROP TABLE <TableName>;
```

Semantics. The result of this command is the removal of the table with a given name from the database and the deletion of all data and index files supporting the table. For simplicity, dropping a table **always removes** Foreign Key constraints referencing the table from other tables, but otherwise requires no other action with respect to the foreign key constraints (that is, B+-tree indexes on these constraints in other tables continue to be maintained).

3. CREATE INDEX statement. The CREATE INDEX statement has the following syntax:

```
CREATE INDEX <IndexName>
ON <TableName> (<AttributeName>[,...,<AttributeName>]);
```

Here, `<IndexName>` is the name of the index. It shall be unique for a given table, but different tables can have indexes with the same name.

`<TableName>` is the name of the table for which the index is to be created. Each index must have at least one attribute specified, and may have more than one (although most of our use cases will concentrate on single-column user-defined indexes).

Semantics.

The result of this command is the creation of a B+-tree index indexing the specified attribute/attributes in the given table. If the table is not empty, the index structure reflecting the current state of the table is built.

For tables stored in persistent storage, the index is built in persistent storage as well. For tables stored in volatile storage, the index is built in volatile storage. Volatile storage tables can have secondary indexes **only if they were constructed** without the use of `INDEX ONLY` keyword. If a `CREATE INDEX` issued for an `INDEX ONLY` volatile storage table, it shall not create an index.

3.DROP INDEX statement. The DROP INDEX statement has the following syntax:

```
DROP INDEX <IndexName>  ON <TableName>;
```

Semantics.

The effect of this statement is to drop the user-created index structure on a given table. Only indexes created via the CREATE INDEX command can be dropped - none of the automatically created and maintained indexes can be dropped. If the index with a given name exists on the specified table, then the index is removed, and the data file storing it is deleted from tinyFS.

2. Data Manipulation Language

The FLOPPY-SQL DML consists of three statements:

1. INSERT
2. DELETE
3. UPDATE

The syntax for each of the statements is a simplified version of ANSI SQL.

NOTE. Handling NULL values. Because INSERT command is restricted (for the sake of simplicity) to inserting only full tuples, FLOPPY-DB DML processor must handle properly the NULL values. By convention, the value NULL when passed as a value of an attribute shall be treated as s null value. Internal representation of NULL values is left to individual teams, but the FLOPPY parser shall properly identify NULL values and not confuse them with any other type of value (e.g., a string "NULL").

1. INSERT statement. The INSERT statement has the following syntax:

```
INSERT INTO <TableName>
VALUES (<V1>,...,<Vn>);
```

Here, the total number of values in the `(<V1>,...,<Vn>)` list must match ***exactly*** the number of attributes in the CREATE TABLE statement creating the given table. The order of values is assumed to be ***exactly the same*** as the order of attributes in the CREATE TABLE statement for the table. If the number of values is different from the number of attributes, or there is a type mismatch between the type of an attribute and the type of the value in its place, finish the INSERT command with an error.

Semantics. A successful execution of the INSERT statement results in the following effects:

- The given tuple is inserted into the specified table.
- The primary key index is updated.
- All foreign key indexes are updated.
- All user-defined indexes are updated.

A tuple whose primary key already exists in the table cannot be inserted. In such cases, the result of the INSERT command shall be an error.

2. `DELETE` statement. The DELETE statement has the following syntax:

```
DELETE FROM <TableName>
WHERE <Condition>;
```

Note. See description of the SELECT statement for the full information about the syntax and the semantics of `<Condition>`.

Semantics.

This operation shall be executed as follows:

- Perform a table scan of the specified table.
- For each tuple, check whether the `<Condition>` is satisfied on it.
- If the condition is satisfied, delete the tuple from the data table, and remove it from the index structures.

From the execution point of view, this command is a variant of the *selection* operation, with the difference being the treatment of the discovered tuples. DELETE command removes them from the data table, whereas *selection* operation returns them without affecting the contents of the table.

3. `UPDATE` statement. The UPDATE statement has the following syntax:

```
UPDATE <TableName>
SET <AttributeName> = <Expression>
WHERE <Condition>;
```

For the sake of simplicity, each UPDATE statement performs only one update to the tuple (i.e., updates only one attribute).

See description of the SELECT statement for the full information about the syntax and the semantics of `<Condition>` and `<Expression>`.

Semantics.

The UPDATE operation works as follows:

- Perform a table scan of the specified table.
- For each tuple, check whether the `<Condition>` is satisfied on it.
- If the condition is satisfied, perform the update specified in the `SET` keyword, i.e., replace the old value of the attribute specified in the SET keyword with the one to which `<Expression>` evaluates.

3. SELECT Statement

The FLOPPY-SQL SELECT statement is a restricted variant of the ANSI SQL SELECT. The syntax is as follows:

```
SELECT [DISTINCT] <SelectItem>[,<SelectItem>]*
FROM <TableSpec>[,<TableSpec>]*
[ WHERE <Condition>]
[ GROUP BY <Attribute>[,<Attribute>]*
[ HAVING <Condition>] ]
[ ORDER BY <OrderTerm>[,<OrderTerm>]]
[ LIMIT <Number>]
;
```

The FLOPPY-SQL SELECT statement describes a combination of the following operations of Relational Algrebra:

- *Projection*
- *Selection*
- *Join*
- *Product (hopefully not!)*
- *Grouping and Aggregation*
- *Sorting*
- *Limiting*

The following syntactic limitations apply.

1. `<SelectItem>`.

To simplify processing, a single `<SelectItem>` can be one of the following:

- An attribute name, specified in an unambiguous way in one of the following formats:
  - A simple attribute name
  - `TableName.attrbuteName`
  - `Alias.attributeName`
- An aggregate function of a single attribute name. The following aggregates are supported:
  - `COUNT(<AttributeName>)`
  - `AVERAGE(<AttributeName>)`
  - `MAX(<AttributeName>)`
  - `MIN(<AttributeName>)`
  - `SUM(<AttributeName>)`
- A "*" wildcard ***as the only <SelectItem>*** in the body of the SELECT clause
- A `COUNT(*)` aggregate.

FLOPPY-SQL does not (out of the box) support built-in functions and arithmetic expressions in the SELECT clause.

2. `<TableSpec>`.

The most restricted version of FLOPPY-SQL supports only two types of table specifications:

- Table name
- Table name followed by a table alias.

There is no support for a nested SELECT statement in the FROM clause in the most restricted version of FLOPPY-SQL.

3. <Condition>

FLOPPY-SQL conditions have the following syntax:

```
<Condition> ::= <AtomicCondition> |
               <Condition> AND <Condition> |
              NOT <Condition> |
              (<Condition>)
<AtomicCondition> ::= <Expression> <Comparison> <Expression> |
                      True | False

<Comparison> ::=  < | > | <= | >= |
                  = | != | <>

<Expression> ::=  <AtomicExpression> |
                <Expression> <ArithmeticOp> <Expression> |
                (<Expression>)

<ArithmeticOp> ::=  + | - | * | / | MOD

<AtomicExpression> ::= <AttributeName> |
                       <Constant> |
                     <AggregateOp>(<AttributeName>)|
                     COUNT(*)

<AggregateOp> :: = COUNT | AVERAGE | MIN | MAX | SUM
```

NOTE: The grammar above describes the overall syntax of the SELECT conditions. Not all conditions are appropriate for all places where conditions might occur. E.g., conditions containing aggregations may only appear in the HAVING clause of the SELECT statement, and are not allowed in the WHERE clauses of any of the operations that have them.

Additionally, there is no nested querying in the most restrictive version of FLOPPY-SQL.

SEMANTICS.

The semantics of the SELECT statement follows the semantics of the ANSI SQL select statement discussed in class with the addition of the *limit* operation as the last operation to be performed (after the final projection). Please refer to you in-class notes and the lecture notes for the full semantics of the SELECT statement.

A brief overview is below. A FLOPPY-SQL SELECT statement is translated as follows:

1. The FROM clause is processed to identify the cartesian product of the tables to participate in the query.
2. The WHERE clause (if exists) is processed to identify any selection conditions on the tables. Join conditions are also identified and the appropriate *Product* operations are replaced with *Join* operations. **As a limitation of FLOPPY-SQL we only allow equi-joins as join operations.**
3. The GROUP BY clause (if exists) is processed to identify the *grouping* attributes)
4. The HAVING clause (if exists) is processed to identify the post-grouping *selection* conditions.
5. The ORDER BY clause (if exists) is processed to identify the parameters of the *sort* operation.
6. The SELECT clause is processed to identify the result of the operation (the *porjection)*.
7. The LIMIT clause (if exists) is processed to identify the parameter of the *limit* operation.