CS411 Project Track 2 Final Report

Team No. 8 Team Name: Quartet
Team Member:
Zhoutao Pei zpei3 | Yuetong Liu liu298 | Shuomeng Guang sguang2 | Zhuang Zuo zhuangz3

1. The Architecture of SimpleDB System

The original and extended SimpleDB system are shown in Fig 1. The gray blocks belongs to the original system, the blue ones are revised classes, and the yellow ones are extended new functions.

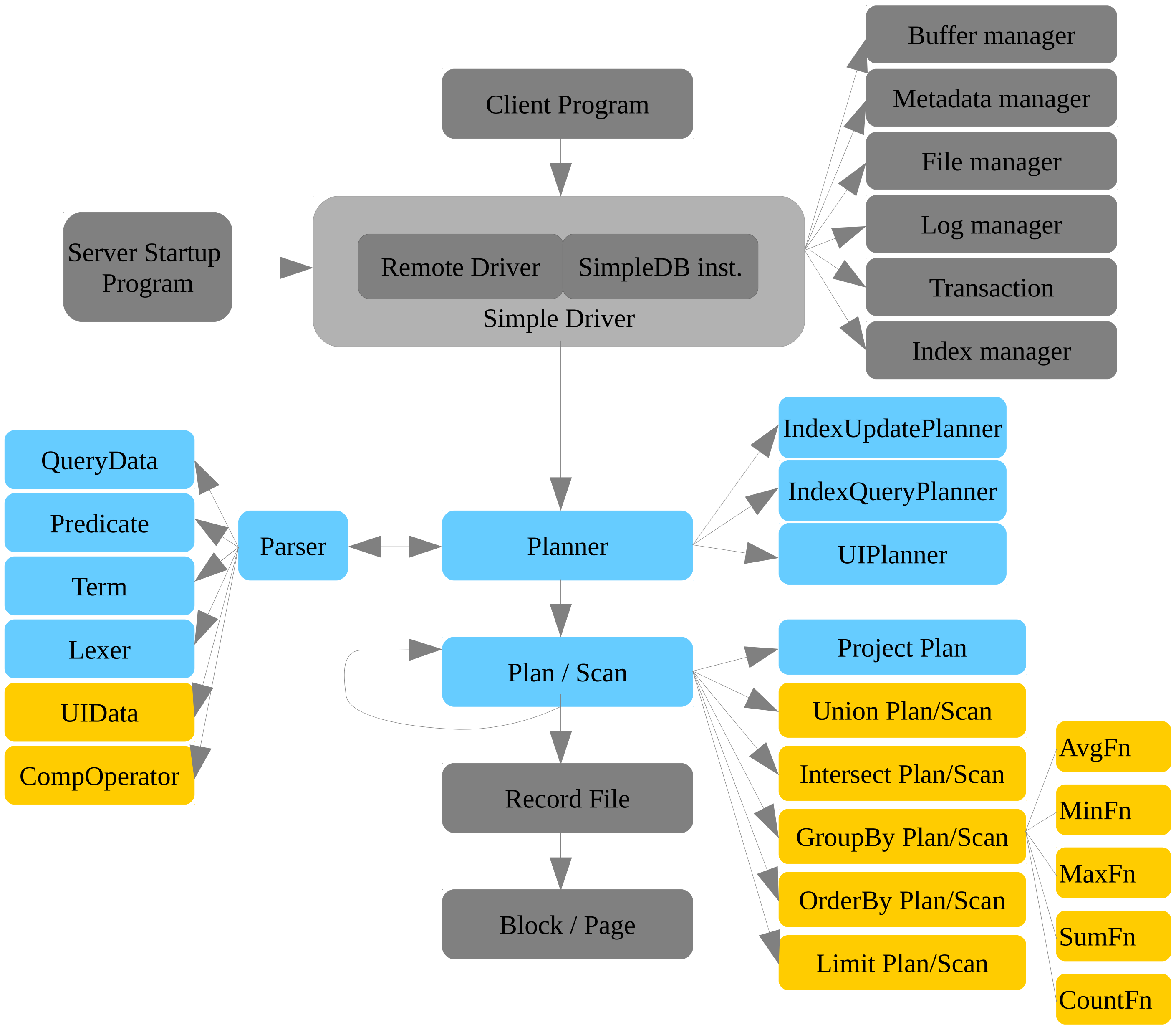

Fig 1. The Architecture of SimpleDB System

2. Function Implementation

2.1 Preparation

To test the new functions of our SimpleDB system, you need to compile it and create a database using the following code:

```
cd SimpleDB_2.10
javac -cp . simpledb/*/*.java simpledb/*/*/*.java            #compile SimpleDB
javac -cp . studentClient/simpledb/*.java                    #compile client code
java simpledb.server.Startup studentdb
jar cf simpledb.jar simpledb/*/*.class simpledb/*/*/*.class
cp simpledb.jar studentClient/simpledb/
cd studentClient/simpledb
java -cp simpledb.jar:. CreateStudentDB            #create a database and add some tables
java -cp simpledb.jar:. AddTable      # add some larger tables (to test the efficiency of index)
# This might take about 1 minute
# To run AddTable, you need 3 txt files, “direct.txt”, “director.txt” and “movie.txt”, which
#  are in folder  studentClient/simpledb/
java -cp simpledb.jar:. SQLInterpreter       # now you can run queries in console
```

2.2 Comparison operator

The original SimpleDB only support predicate in the form “ A = B ”. We added more comparison operators, including “<”, “>”, “<=”, “>=” and “<>”. These operators can be used to compare both integers and strings.

Test case:

select majorid, gradyear, sname, sid from student where gradyear >= 2005

Query result:

```
majorid  gradyear    sname   sid
-----------------------------------
10       2005         max     3
20       2005         sue     4
```

Test case:

select majorid, gradyear, sname, sid from student where sname < 'max'

Query result:

```
majorid  gradyear    sname   sid
-----------------------------------
 10      2004         joe     1
 20      2004         amy     2
 30      2003         bob     5
 20      2001         kim    6
 30      2004         art     7
 10      2004         lee     9
```

Files modified or added:

Parser.java
 revised: method term()

Term.java

revised: constructor, method isSatisfied()
added: member variable Oper
CompOperator.java : new class (under query package)

2.3 Select all

We added support for "*" to allow selecting all fields from the tables. To implement this feature, we revised Parser, so that it will return an empty field list if it sees "*", and ProjectPlan will extract all the fields from schema if the field list is empty.

Test case:

*select * from student, dept where majorid = did and sid < 3*

Query result:

```
majorid  gradyear    sname   dname     did   sid
-------------------------------------------------------
 10      2004            joe      compsci    10     1
 20      2004            amy     math        20     2
```

Files modified or added:

ProjectPlan.java
revised: constructor
Parser.java
revised: method query()
QueryData.java
revised: method toString()
StringConstant.java
revised: method toString()

2.4 Union and intersect

Since union and intersect operation will combine two query results, we wrote a UnionPlanner instead of modify any existing planner. We also wrote new plan and scan. The basic idea for implementing union is to maintain a set in UnionScan to remember the value it has seen, and for intersect, this set will read in records from one table first and than look for same records in the other table. Which table to read in first depends on which table has less records. The function we implemented are set union and set intersect, so you won't see duplicates in the outputs.

Test case:

*select * from student where sid < 2 union select * from student where sid > 8*

Query result:

```
  majorid  gradyear    sname   sid
-----------------------------------
    10       2004           joe     1
    10       2004           lee     9
```

Test case:

*select * from student where sid > 2 intersect select * from student where sid < 8*

Query result:

```
majorid  gradyear   sname   sid
----------------------------------
  10       2005       max      3
  20       2005       sue      4
  30       2003       bob      5
  20       2001       kim      6
  30       2004       art      7
```

Files modified or added:

Planner.java
    revised: constructor, method createQueryPlan()
    added: member variable uiplanner,
SimpleDB.java
    revised: method planner()
Lexer.java
    revised: keywords
Parser.java
    added method unionIntersect()
UIData.java: new class, under parse package
UIPlanner.java: new class, under planner package
UnionPlan.java: new class, under query package
UnionScan.java: new class, under query package
IntersectPlan.java: new class, under query package
IntersectScan.java: new class, under query package

2.5 Complex predicate

The original SimpleDB only supports multiple term connected by keyword “and”. For each record, it would check each of the term, if any one of them is not satisfied, then the record is skipped.

In our extended SimpleDB system, the predicate can be more complex. We added support for “or” and also parenthesis to separate terms in query predicate.

To implement complex predicate, we added three arraylists in predicate to save the information for left, right parenthesis and logical operators. Basically, we used two stacks to evaluate if a predicate is true or not, Fig 2 shows a simple example:

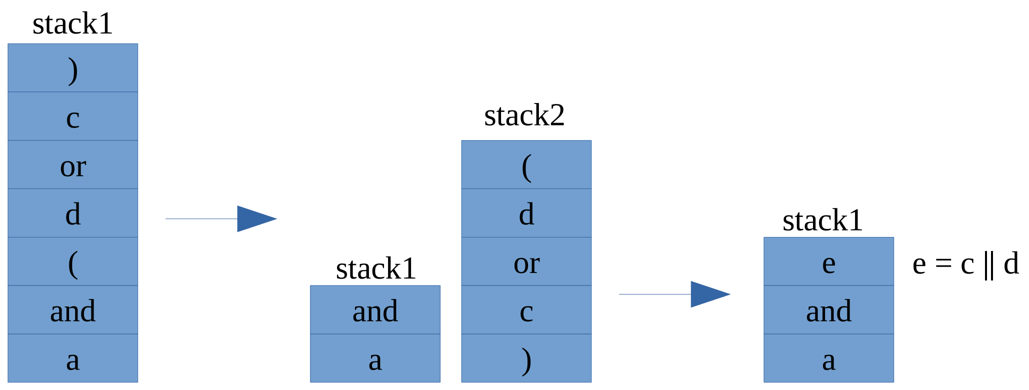

Fig 2. Use 2 stacks to evaluate a complex predicate

We also implemented "in" operator by converting "A in (a, b, c)" into a series of "or" terms "A = a or A = b or A=c).

Test case:

*select * from student, enroll where ((sid=studentid) and ((gradyear in (2004,2005)) or (grade='A')))*

Query result:

```
studentid   eid  majorid  gradyear    sname  grade  sectionid   sid
---------------------------------------------------------------------
    1        14     10      2004       joe     A       13     1
    1        24     10      2004       joe     C       43     1
    2        34     20      2004       amy     B+      43     2
    4        44     20      2005       sue     B       33     4
    4        54     20      2005       sue     A       53     4
    6        64     20      2001       kim      A      53     6
```

Files modified or added:

Lexer.java
- revised: keyword

Parser.java
- revised: method term(), predicate()

Predicate.java
- revised: method isSatisfied()
- added: member variables leftBrac, rightBrac, logicalOper, isComplex, method setComp() simpleIsSatisfied(), complexIsSatisfied()

2.6 Group by

We utilized the GroupByPlan and GroupByScan under "materialize" folder and add three more aggregation function "avg", "min" and "sum" apart from "count" and "max".

Test case:

select count(sid), majorid, gradyear from student group by majorid, gradyear

Query result:

```
countofsid  majorid  gradyear
---------------------------------
    2           10       2004
    1           10       2005
    2           20       2001
    1           20       2004
 ………..
```

Files modified or added:

Parser.java
- revised: method query(), selectList()
- added: method groupList()

QueryData.java
- revised: method toString()

added: member variable groupFields, constructor, method aggFns(), groupFields(),
BasicQueryPlanner.java
    revised: method createPlan()
Lexer.java
    revised: keyword
AvgFn.java - new class, under materialize package
MinFn.java - new class, under materialize package
SumFn.java - new class, under materialize package

2.7 Order by

We implemented "order by" operator by using materialization. The method open in OrderPlan will preprocess the input records, split them into up to 2 sorted temporary tables. Then they are passed into OrderScan for merging. Each call to method next in OrderScan will simply retrieve the next record from the sorted temporary tables.

In our extended SimpleDB, records can be sorted by ascending or descending order. This is implemented by maintaining a list of booleans in OrderPlan to indicate the sorting order for each column.

Test case:

*select * from student,dept where majorid=did order by gradyear desc, dname*

Query result:

```
majorid  gradyear    sname   dname     did   sid
--------------------------------------------------------
   10       2005         max     compsci    10     3
   20      2005          sue     math        20     4
   10      2004          lee     compsci    10     9
   10      2004          joe     compsci    10     1
   ……...
```

Files modified or added:

Lexer.java
    revised: keyword
Parser.java
    revised: method query()
    added: method sortList()
QueryData.java:
    revised: constructor
    added: member variable sortFields, method sortFields(), desc()
BasicQueryPlanner
    revised: method createPlan()
OrderPlan.java - new class, under materialize package
OrderScan.java - new class, under materialize package
OrderRecordComparator.java - new class, under materialize package

2.8 Limit

The "limit" function is implemented by adding LimitPlan and LimitScan, which only output a certain number of records.

The most complicated query flow now becomes:

TablePlan → ProductPlan → SelectPlan → GroupByPlan → OrderPlan → ProjectPlan → LimitPlan

Here we give a complex test case:

select sectionid, count(sid) from student, enroll where sid=studentid and (gradyear in (2004,2005) or sid > 5) group by sectionid order by sectionid desc limit 2

Query result:

```
countofsid  sectionid
-----------------------
   2           53
   2           43
```

Files modified or added:

Parser.java
    revised: method query()
QueryData.java
    added: member variable, constructor, method toString(), add limit()
Lexer.java
    revised: keyword
BasicQueryPlanner.java
    revised: method createPlan()
LimitPlan.java - new class, under query package
LimitScan.java - new class, under query package

2.9 Index selection, index join and query optimization

"Create index" has been implemented in the original SimpleDB. We revised IndexInfo so that it can use BTree index. To utilize index in query, a new IndexQueryPlanner was created.

To compare the difference between query with index and without index, here we use table director, direct and movie, which are created by program AddTable in section 2.1. The relationship between these tables are shown below:

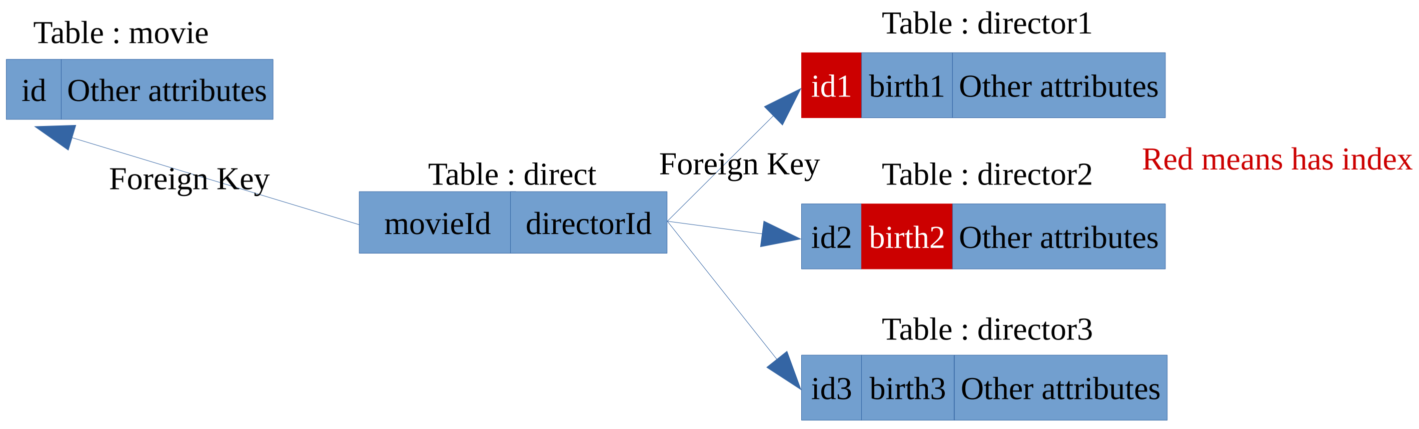

Fig 3. Relationship between tables used for testing index query

Table director1, director2 and director3 has exactly the same content, however, director1 has index for id1, and director2 has index for birth, director3 has no index. We revised SQLInterpreter so that it can show running time for each query.

In IndexQueryPlanner, we didn't follow the sequence of query plans in BasicQueryPlanner. Instead, we did some query optimization. If a term is in the form "Column A = some constant", SelectPlan will be executed directly after TablePlan, instead of ProductPlan and SelectPlan in BasicQueryPlanner. If Column A contains index, IndexSelectPlan will be executed.

Test case:

Query1:

select name2, title from director2, direct, movie where id2 = directorId and id = movieId and birth2 = 1964 limit 5

Query2(be careful, this might take 1 minute to run):

select name3, title from director3, direct, movie where id3 = directorId and id = movieId and birth3 = 1964 limit 5

Query result:

```
                 name2                    title
-------------------------------------------------------------
            EytanFox                TheBubble
            PeterFlinth             ArnTheKnightTemplar
            StephanElliott          EyeoftheBeholder
            StephanElliott          EasyVirtue
           GuillermodelToro        PacificRim
Time Elapsed for Query1: 5152832051
Time Elapsed for Query2: 57829743627
```

The query results are the same, but the running time for query 2 is significantly longer than query1.

In IndexQueryPlanner, if more than two tables are joined, the table with index on joining field will be joined first, and IndexJoinPlan will be performed, instead of ProductPlan and SelectPlan.

Test case:

Query1:

select name1, title from director1, direct, movie where id1 = directorId and id = movieId and year = 1955 limit 5

*Query2(**do not** try this, this will take more than 10 mins):*

select name3, title from director3, direct, movie where id3 = directorId and id = movieId and year = 1955 limit 5

Query result:

```
                          title              name1
-------------------------------------------------------------
                         Ordet        CarlTheodorDreyer
Time Elapsed for Query1 : 2278075587
Time Elapsed for Query2 : 996650645315
```

Again, the query results will be the same, but the running time for query 2 is significantly longer than query1.

Files modified or added:

- Planner.java
  - revised: method executeUpdate()
  - added: method hasIdx(),
- Planner.java
  - revised: method createQueryPlan()
  - added: method predHasIdx()
- IndexQueryPlanner.java - new class, under index.planner package
- IndexInfo.java
  - revised: method open(), blockAccessed()
- Predicate.java
  - added: method size()

3. Lessons Learned

Firstly, none of us have written code in Java before, so this project is quite challenging. At first, we don’t even know how to compile the code and run test cases. Through learning Java and getting familiar with the code, we start from the easy part, discuss about the structure, and test with small changes. The learning curve is sharp but worthwhile.

More important, since the database contains multiple units, it is key to figure out the relation between each units. An overview of the architecture is very important.