CSEN 604 (Databases 2)
Dr Fatma Meawad
Caroline Sabty
Spring 2013

Planner and Optimization chapter 19, 24

Team 4

You should read chapters 19, 24 and understand how simpleDB implements the planner and query optimization. In chapter 24 skip sections 24.4 and 24.6.2.

It is optional for you to read parts of the parser chapter 18, in order to understand better the changes you will apply. Also, to know more information about the operator you will implement see its explanation in Chapter 4.

In addition you should implement the following required modifications.

SimpleDB doesn't support nested queries. You should modify the ***lexical analyzer***, ***parser*** and ***planner*** (and possibly other classes) to allow nested queries. The changes of the **lexical analyzer** and ***parser*** are given to you bellow, the implementation of the *semijoin* and *antijoin* operators. For more information about the operator you will implement see Chapter 4.

You may need to solve or adjust other errors that may occur while applying these changes. Your task is to implement the changes of the **planner.**

After applying all the above required changes, write a JDBC program to test your code.

Query Package:

You need to create a new class called SemijoinPlan

```
package simpledb.query;

import simpledb.record.Schema;

//The only suspect method is recordsOutput.
```

```
//Our “equal distribution” assumption means that every record will match some record from p2,
//which is not realistic for the cases when semijoin is used.

public class SemijoinPlan {
	private Plan p1, p2;
	private Predicate pred;

	public SemijoinPlan(Plan p1, Plan p2, Predicate pred) {
		this.p1 = p1;
		this.p2 = p2;
		this.pred = pred;
	}

	public Scan open() {
		Scan s1 = p1.open();
		Scan s2 = p2.open();
		return new SemijoinScan(s1, s2, pred);
	}

	public int blocksAccessed() {
		return p1.blocksAccessed()
		    + (p1.recordsOutput() * p2.blocksAccessed());
	}

	public int recordsOutput() {
		// assumes that every record matches
		return p1.recordsOutput();
	}

	public int distinctValues(String fldname) {
		return Math.min(p1.distinctValues(fldname), recordsOutput());
	}

	public Schema schema() {
		return p1.schema();
	}
}
```

You need to create another new class called SemijoinScan

```
package simpledb.query;
```

```
//The crucial method is next.  For each record in s1,
//the code searches for a matching s2 record.
//if one is found, the method returns true; otherwise, the next s1 record is considered.

public class SemijoinScan implements Scan {
	private Scan s1, s2;
	private Predicate pred;

	public SemijoinScan(Scan s1, Scan s2, Predicate pred) {
		this.s1 = s1;
		this.s2 = s2;
		this.pred = pred;
	}

	public void beforeFirst() {
		s1.beforeFirst();
	}

	public boolean next() {
		while (s1.next()) {
			s2.beforeFirst();
			while (s2.next()) {
				Scan s = new CombineScan(s1, s2);
				if (pred.isSatisfied(s)) {
					s2.close();
					return true;
				}
			}
			s2.close();
		}
		return false;
	}

	public void close() {
		s1.close();
	}

	public Constant getVal(String fldname) {
		return s1.getVal(fldname);
	}

	public int getInt(String fldname) {
		return s1.getInt(fldname);
	}
```

```
	public String getString(String fldname) {
		return s1.getString(fldname);
	}

	public boolean hasField(String fldname) {
		return s1.hasField(fldname);
	}

	public boolean wasNull() {
		return s1.wasNull();
	}
}
```

```
package simpledb.query;
```

You need to create another new class called CombineScan

```
//A technical difficulty occurs in testing for matching records,
//namely that the predicate's isSatisfied method only works for a single scan,
//not two scans. Thus we create a new class CombineScan, whose job is to simply
//turn s1 and s2 into a single scan by implementing the get methods.

	public class CombineScan implements Scan {
		private Scan s1, s2;

		public CombineScan(Scan s1, Scan s2) {
			this.s1 = s1;
			this.s2 = s2;
		}

		public void beforeFirst() {
		}

		public void close() {
		}

		public boolean next() {
			return false;
		}

		public int getInt(String fldname) {
			if (s1.hasField(fldname))
				return s1.getInt(fldname);
```

```
                else
                        return s2.getInt(fldname);
        }

        public String getString(String fldname) {
                if (s1.hasField(fldname))
                        return s1.getString(fldname);
                else
                        return s2.getString(fldname);
        }

        public Constant getVal(String fldname) {
                if (s1.hasField(fldname))
                        return s1.getVal(fldname);
                else
                        return s2.getVal(fldname);
        }

        public boolean hasField(String fldname) {
                return s1.hasField(fldname) || s2.hasField(fldname);
        }

        public boolean wasNull() {
                return false;
        }
}
```

You need to create another new class called *AntijoinPlan*

```
package simpledb.query;

import simpledb.record.Schema;

//This code is very similar to that of semijoin.
//The main difference is in the scan's next method.
//The method examines each record of s1, looking for matching s2 records.
//If no match is found it returns true; otherwise, it considers the next s1 record.

public class AntijoinPlan {
        private Plan p1, p2;
        private Predicate pred;

        public AntijoinPlan(Plan p1, Plan p2, Predicate pred) {
```

```
            this.p1 = p1;
            this.p2 = p2;
            this.pred = pred;
        }

        public Scan open() {
            Scan s1 = p1.open();
            Scan s2 = p2.open();
            return new AntijoinScan(s1, s2, pred);
        }

        public int blocksAccessed() {
            return p1.blocksAccessed() + (p1.recordsOutput() * p2.blocksAccessed());
        }

        public int recordsOutput() {
            // assumes that every record matches
            return 0;
        }

        public int distinctValues(String fldname) {
            return p1.distinctValues(fldname);
        }

        public Schema schema() {
            return p1.schema();
        }
}
```

You need to create another new class called *AntijoinScan*

```
package simpledb.query;

//Note that it uses the CombineScan class, just like SemijoinScan.


public class AntijoinScan implements Scan {
        private Scan s1, s2;
        private Predicate pred;

        public AntijoinScan(Scan s1, Scan s2, Predicate pred) {
            this.s1 = s1;
            this.s2 = s2;
            this.pred = pred;
```

```
}

public void beforeFirst() {
        s1.beforeFirst();
}

public boolean next() {
        while (s1.next()) {
                boolean found = false;
                s2.beforeFirst();
                while (s2.next()) {
                        Scan s = new CombineScan(s1, s2);
                        if (pred.isSatisfied(s)) {
                                found = true;
                                break;
                        }
                }
                s2.close();
                if (!found)
                        return true;
        }
        return false;
}

public void close() {
        s1.close();
}


public Constant getVal(String fldname) {
        return s1.getVal(fldname);
}

public int getInt(String fldname) {
        return s1.getInt(fldname);
}

public String getString(String fldname) {
        return s1.getString(fldname);
}

public boolean hasField(String fldname) {
        return s1.hasField(fldname);
}
```

```
	public boolean wasNull() {
		return s1.wasNull();
	}
}
```

Parser Package:

Lexer class:

You need to add “not” and “in” as keywords.

Parser class:

```
public Term term() {
		if (lex.matchId())
			return fieldFirstTerm();
		else
			return constFirstTerm();
	}

  public Term fieldFirstTerm() {
		return ffTerm(field());
	}


public Term fieldFirstTerm() {
		return ffTerm(field());
	}

public Term ffTerm(String fldname) {
		if(lex.matchDelim('='))
		{
			lex.eatDelim('=');
			Expression exp1 = new FieldNameExpression(fldname);
			Expression exp2 = expression();
			return new BasicTerm(exp1, exp2);
		}
		else {
			boolean negated = lex.matchKeyword("not");
			if (negated)
```

```
                    lex.eatKeyword("not");
                lex.eatKeyword("in");
                lex.eatDelim('(');
                QueryData qd = query();
                lex.eatDelim(')');
                return new NestedTerm(fldname, qd, negated, fieldDefs());
            }
        }

    public Term constFirstTerm() {
            Expression lhs = expression();  // will be a constant exp
            lex.eatDelim('=');
            Expression rhs = expression();
            return new BasicTerm(lhs, rhs);
        }
```

Since there are two ways to create a term, there are two constructors. Therefore you might as well make *Term* be an interface, with each kind of term being its own implementing class. The above code assumes that there are now two *Term* classes: ***BasicTerm*** and ***NestedTerm***. The code for ***BasicTerm*** is the **same** as what we had been calling ***Term***. Therefore you just need to write code for ***NestedTerm***.

However, the system won't use the nested term as a term. Instead, the planner will extract the nested terms from the predicate and implement them as semijoins and antijoins, as explained in Chapter 4. Thus you will just need the methods ***getQueryData***, ***getNegated***, and ***getNestedPred***. The first two methods return the values passed into the constructor. The third method constructs a predicate from the outer field name and the field in the select clause of the nested query.

Query Package:

NestedTerm class:

```
package simpledb.query;

import com.sun.org.apache.bcel.internal.generic.FDIV;

import simpledb.parse.QueryData;
import simpledb.record.Schema;
import simpledb.tx.Transaction;
```

```
public class NestedTerm implements Term {
        private QueryData nestedQuery;
        private boolean negated;
        private Predicate nestedPred;
        private String fldname;

        public NestedTerm(String fldname, QueryData nestedQuery,
                boolean negated) {
                  this.nestedQuery = nestedQuery;
                  this.negated = negated;
                  this.fldname = fldname;
                  String nestedfldname = qd.fields().iterator().next();
                  Expression exp1 = new FieldNameExpression(fldname);
                  Expression exp2 = new FieldNameExpression(nestedfldname);
                  nestedPred = new Predicate(new BasicTerm(exp1, exp2));
        }

        public QueryData getQueryData() {
                  return nestedQuery;
        }

        public Predicate getNestedPred() {
                  return nestedPred.toString();
        }

        public boolean getNegated() {
                  return negated;
        }

    public int reductionFactor(Plan p) {
                  return 1;  // some default
        }

        public Constant equatesWithConstant(String fldname) {
                  return null;
        }

        public String equatesWithField(String fldname) {
                  return null;
        }

        public boolean appliesTo(Schema sch) {
                  return sch.fields().contains(fldname);
```

```
	}

	public boolean isSatisfied(Scan s, Transaction tx) {
		return false;
	}

	public String toString() {
		String op = negated ? "NOT IN" : "IN";
		return fldname + " " + op + "(" + nestedQuery + ")";
	}

	@Override
	public boolean isSatisfied(Scan s) {
		// TODO Auto-generated method stub
		return false;
	}
}
```

You also need to modify the class ***Predicate*** so that the planner can extract the nested terms. You should therefore add the following methods:

Predicate class:

```
	public Collection<NestedTerm> nestedTerms() {
		Collection<NestedTerm> result =
				new ArrayList<NestedTerm>();
		for (Term t : terms)
			if (t instanceof NestedTerm)
				result.add((NestedTerm) t);
		return result;
	}

	public Predicate simplePred() {
		Predicate result = new Predicate();
		for (Term t : terms)
			if (!(t instanceof NestedTerm))
				result.conjoinWith(new Predicate(t));
		return result;
	}
```

After applying all the previous changes and having in ***Predicate*** *class* two new methods: a method ***simplePred*** that returned the non-nested terms of the predicate; and a method ***nestedTems*** that returns the nested terms. You should modify method ***createSelectPlan*** in

class ***BasicQueryPlanner*** by adding two steps the first step after Step 2 (which processes the product operators) and the second step before Step 3 (which processes the select operator).

```
public Plan createSelectPlan(SelectQueryData data, Transaction tx);
```

After applying all the above required changes, write a JDBC program to test your code.