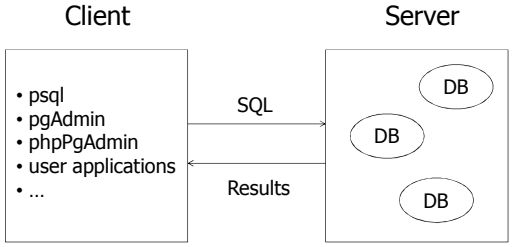
CS422 Principles of Database Systems

Introduction to Query Processing

Chengyu Sun
California State University, Los Angeles

DBMS



SimpleDB

- Developed by Edward Sciore
- A simplified DBMS for educational purpose
- Source code - http://csns.calstatela.edu/wiki/content/cysun/course_materials/cs422/simpledb

SimpleDB Basics

- Server
  - Server class: `simpledb.server.SimpleDB`
  - Startup program: `simpledb.server.Startup`
- Clients
  - `simpledb.client.SQLInterpreter`
  - Some other programs

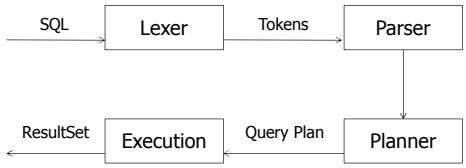
Query Processing

- Departments(dId, dName)
- Students(sId, sName, majorId)

```
select sName, dName
     from Students, Departments
     where majorId = dId and sId = 1;
```

What happens in a DBMS server when we run a query?

Query Processing in SimpleDB

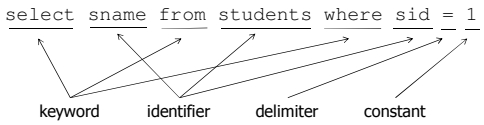


Query Parsing

- Analyze the query string and convert it into some *data structure* that can be used for query execution

Lexical Analysis

- Split the input string into a series of tokens



Token

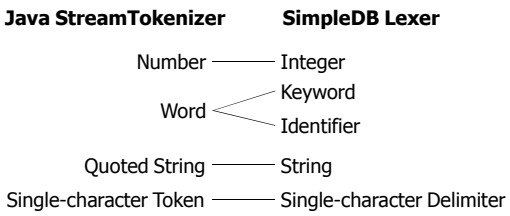
- <type, value>

Type	Value
keyword	select
identifier	sname
keyword	from
identifier	students
keyword	where
identifier	id
delimiter	=
intconstant	1

SimpleDB Token Types

- Single-character delimiter
- Integer constants
- String constants
- Keywords
- Identifiers

SimpleDB Lexer Implementation



- Example: StreamTokenizerTest

Lexer API …

- The API used by the *parser*
- Iterate through the tokens
  - Check the current token – "Match"
    - `matchKeyword()`, `matchId()`, `matchIntConstant()` …
  - Consume the current token – "Eat"
    - `eatKeyword()`, `eatId()`, `eatIntConstant()` …

… Lexer API

```
select sname from students where sid = 1
```

current token

```
lexer.matchKeyword("select");
lexer.eatKeyword("select");
```

```
select sname from students where sid = 1
```

current token

Syntax

- A set of rules that describes the strings that could *possibly* be meaningful statements
- Example: a syntactically wrong statement

```
select from a and b where c – 3;
```

Part of SimpleDB Grammar …

<Field> := IdTok
<Constant> := StrTok **|** IntTok
<Expression> := <Field> **|** <Constant>
<Term> := <Expression> = <Expression>
<Predicate> := <Term> **[** AND <Predicate> **]**

Backus-Naur Form (BNF)

… Part of SimpleDB Grammar

<Query> := SELECT <SelectList> FROM <TableList>
[WHERE <Predicate>]
<SelectList> := <Field> [, <SelectList>]
<TableList> := IdTok [, <TableList>]

<CreateTable> := CREATE TABLE IdTok (<FieldDefs>)
<FieldDefs> := <FieldDef> [, <FieldDefs>]
<FieldDef> := IdTok <TypeDef>
<TypeDef> := INT | VARCHAR (IntTok)

Full SimpleDB Grammar in Textbook Figure 18-4

Recursive Definition in Grammar

<SelectList> := <Field> [, <SelectList>]

```
select a, b, c from t where x = 10;
```

<SelectList> ??

Using Grammar

- Which of the following are valid SimpleDB SQL statements??

create table students (id integer, name varchar(10))
select * from students;

From Grammar to Code …

```
public QueryData query()
{
   lex.eatKeyword( "select" );
   Collection<String> fields = selectList();
   lex.eatKeyword( "from" );
   Collection<String> tables = tableList();
   Predicate pred = new Predicate();
   if( lex.matchKeyword("where") )
   {
      lex.eatKeyword("where");
      pred = predicate();
   }
   return new QueryData( fields, tables, pred );
}
```

… From Grammar to Code

```
public Collection<String> selectList()
{
   Collection<String> L = new ArrayList<String>();
   L.add( field() );
   if( lex.matchDelim(',') )
   {
      lex.eatDelim(',');
      L.addAll( selectList() );
   }
   return L;
}

public String field() { return lex.eatId(); }
```

After Parsing

```
select sName, dName
        from Students, Departments
        where majorId = dId and sId = 1;
```

`QueryData` in SimpleDB:

`fields`	{ "sName", "dName" }
`tables`	{ "Students", "Departments" }
`pred`	{ {"majorId", "dId"}, {"sId", 1}

More in the simpledb.parse package.

Query Planning

- Break a query into *individual operations*, and organize them into certain order, i.e. a query plan.

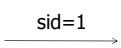
Relational Algebra Operations

- Selection, projection, product
- Join
- Rename
- Set operations: union, intersection, difference
- Extended Relation Algebra operations
  - Duplicate elimination
  - Sorting
  - Extended projection, outer join
  - Aggregation and grouping

Selection

Input

sid	sname
1	Joe
2	Amy



Output

sid	sname
1	Joe

Projection

Input

sid	sname
1	Joe
2	Amy

sname →

Output

sname
Joe
Amy

Product

Input

sid	sname	dept
1	Joe	10
2	Amy	20

did	dname
10	CS
20	Math

⇩

Output

sid	sname	dept	did	dname
1	Joe	10	10	CS
1	Joe	10	20	Math
2	Amy	20	10	CS
2	Amy	20	20	Math

Scan

- A scan is an interface to a RA operation implementation

```
public interface Scan {

    public boolean next(); // move to the next result
    public int getInt( String fieldName );
    public String getString( String fieldName );

}
```

Scan Example: TableScan

```
public TableScan( TableInfo ti, Transaction tx )
{ recordFile = new RecordFile( ti, tx ); }

public boolean next()
{  return recordFile.next(); }

public int getInt( String fieldName )
{  return recordFile.getInt( fieldName ); }

public int getString( String fieldName )
{  return recordFile.getString( fieldName ); }
```

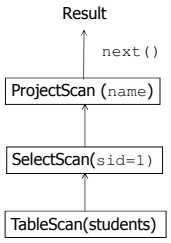
Scan Example: SelectScan

```
public SelectScan( Scan s, Predicate pred )
{
    this.s = s;
    this.pred = pred;
}

public boolean next()
{
    while( s.next() )
        if( pred.isSatisfied(s) ) return true;
    return false;
}
```

Query Execution

select name from students where id = 1;

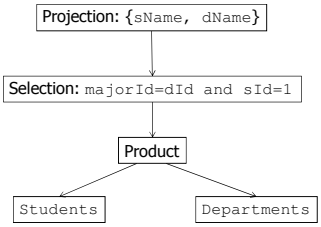


About Implementations of RA Operations

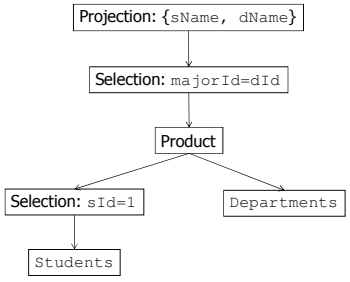
- Each RA operation can be implemented and optimized independently from others
- A RA operation may have multiple implementations
  - E.g. *table scan* vs. *index scan* for selection
- The efficiency of an implementation depends on the characteristics of the data

A Query Plan

select sName, dName from Students, Departments
where majorId = dId and sId = 1;



A Better Query Plan – Query Optimization



Readings

- Textbook Chapter 17, 18, 19