CS422 Principles of Database Systems
Introduction to Query Processing

Chengyu Sun
California State University, Los Angeles

DBMS

Client

- psql
- pgAdmin
- phpPgAdmin
- user applications
- ...

Server

SQL

Results

Client

Server

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 )

```sql
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

```plaintext
SQL
Lexer
Tokens
Parser

ResultSet
Execution
Query Plan
Planner
```
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.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>select</td>
</tr>
<tr>
<td>identifier</td>
<td>sname</td>
</tr>
<tr>
<td>keyword</td>
<td>from</td>
</tr>
<tr>
<td>identifier</td>
<td>students</td>
</tr>
<tr>
<td>keyword</td>
<td>where</td>
</tr>
<tr>
<td>identifier</td>
<td>id</td>
</tr>
<tr>
<td>delimiter</td>
<td>=</td>
</tr>
<tr>
<td>intconstant</td>
<td>1</td>
</tr>
</tbody>
</table>

SimpleDB Token Types

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

SimpleDB Lexer Implementation

Java StreamTokenizer  
- Number  
- Word  
- Quoted String  
- Single-character Token

SimpleDB Lexer

- Number  
- Word  
- Quoted String  
- Single-character Token

- Keywords
- Identifiers

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

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

Syntax

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

```
<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 )
```

Recursive Definition in Grammar

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

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

Using Grammar

- Which of the following are valid SimpleDB SQL statements?
  ```sql
  create table students (id integer, name varchar(10))
  select * from students;
  ```
From Grammar to Code ...

```java
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

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

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

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Amy</td>
</tr>
<tr>
<td>sid=1</td>
<td></td>
</tr>
</tbody>
</table>

Projection

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Amy</td>
</tr>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
</tr>
<tr>
<td></td>
<td>Amy</td>
</tr>
</tbody>
</table>
Product

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>dept</th>
<th>did</th>
<th>dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
<td>10</td>
<td>10</td>
<td>CS</td>
</tr>
<tr>
<td>2</td>
<td>Amy</td>
<td>20</td>
<td>20</td>
<td>Math</td>
</tr>
</tbody>
</table>

Scan

A scan is an interface to a RA operation implementation

```java
public interface Scan {
    public boolean next(); // move to the next result
    public int getInt( String fieldName );
    public String getString( String fieldName );
}
```

Scan Example: TableScan

```java
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

```java
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;
```

Result

```
ProjectScan(name)
SelectScan(sid=1)
TableScan(students)
```

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

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

Readings

- Textbook Chapter 17, 18, 19