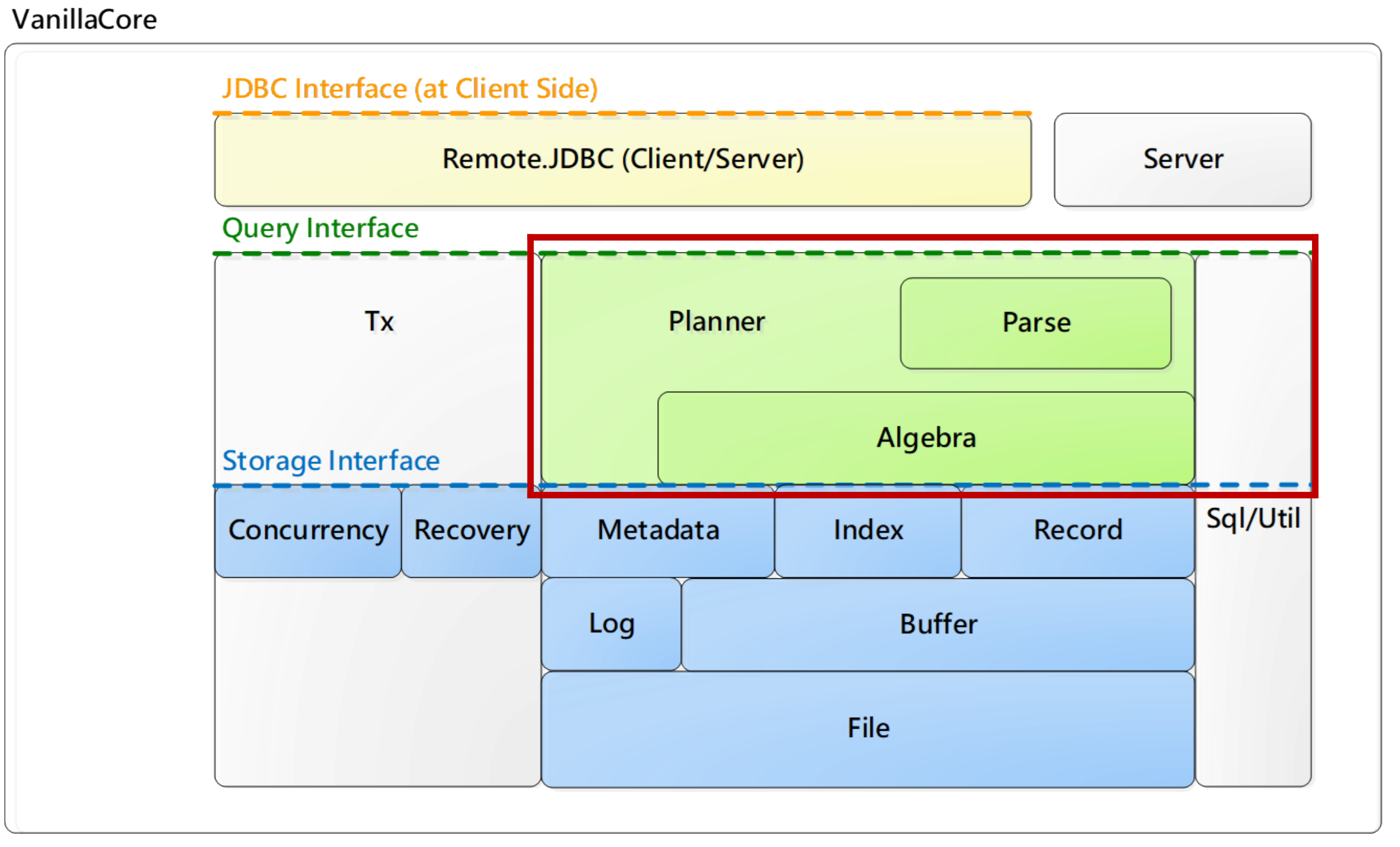
Query Processing

Shan-Hung Wu & DataLab
CS, NTHU

Query Engine



Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

Recap: Finding Major

- JDBC client

```java
Connection conn = null;
try {
	// Step 1: connect to database server
	Driver d = new JdbcDriver();
	conn = d.connect("jdbc:vanilladb://localhost", null);
	conn.setAutoCommit(false);
	conn.setReadOnly(true);

	// Step 2: execute the query
	Statement stmt = conn.createStatement();
	String qry = "SELECT s-name, d-name FROM departments, "
		+ "students WHERE major-id = d-id";
	ResultSet rs = stmt.executeQuery(qry);
	// Step 3: loop through the result set
	rs.beforeFirst();
	System.out.println("name\tmajor");
	System.out.println("-------\t-------");
	while (rs.next()) {
		String sName = rs.getString("s-name");
		String dName = rs.getString("d-name");
		System.out.println(sName + "\t" + dName);
	}
	rs.close();
} catch (SQLException e) {
	e.printStackTrace();
} finally {
	try {  // Step 4: close the connection
		if (conn != null) conn.close();
	} catch (SQLException e) {
		e.printStackTrace();
	}
}
```

- Native (server side)

```java
VanillaDb.init("studentdb");

// Step 1 correspondence
Transaction tx = VanillaDb.txMgr().transaction(
Connection.TRANSACTION_SERIALIZABLE, true);

// Step 2 correspondence
Planner planner = VanillaDb.newPlanner();
String query = "SELECT s-name, d-name FROM departments, "
	+ "students WHERE major-id = d-id";
Plan plan = planner.createQueryPlan(query, tx);
Scan scan = plan.open();

// Step 3 correspondence
System.out.println("name\tmajor");
System.out.println("-------\t-------");
while (scan.next()) {
	String sName = (String) scan.getVal("s-
	name").asJavaVal();
	String dName = (String) scan.getVal("d-
	name").asJavaVal();
	System.out.println(sName + "\t" + dName);
}
scan.close();

// Step 4 correspondence
tx.commit();
```

Query Evaluation: Input and Output

- Input:
  - A SQL command (string)
- Output for `SELECT`:
  - Scan (iterator of records) of the output table
  - By `planner.createQueryPlan().open()`
- Output for others commands (`CREATE`, `INSERT`, `UPDATE`, `DELETE`):
  - #records affected
  - By `planner.executeUpdate()`

What does a `Planner` do?

1. Parses the SQL command
2. Verifies the SQL command
3. Finds a good plan for the SQL command
4. a. Returns the plan (`createQueryPlan()`)
   b. Executes the plan by iterating through the corresponding scan and returns #records affected (`executeUpdate()`)

Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

What does a `Planner` do?

1. Parses the SQL command
2. Verifies the SQL command
3. Finds a good plan for the SQL command
4. a. Returns the plan (`createQueryPlan()`)
   b. Executes the plan by iterating through the scan and returns #records affected (`executeUpdate()`)

SQL Statement Processing

- Input:
  - A SQL statement
- Output:
  - Internal ***SQL data*** object that can be fed to the constructors of various plans/scans
- Two stages:
  - ***Parsing*** (syntax-based)
  - ***Verification*** (semantic-based)

Syntax vs. Semantics

- The ***syntax*** of a language is a set of rules that describes the strings that could possibly be meaningful statements
- Is this statement syntactically legal?

```
SELECT FROM TABLES t1 AND t2 WHERE b - 3
```

- No
  - SELECT clause must refer to some field
  - TABLES is not a keyword
  - AND should separate predicates not tables
  - b-3 is not a predicate

Syntax vs. Semantics

- Is this statement syntactically legal?

  `SELECT a FROM t1, t2 WHERE b = 3`
  - Yes, we can infer that this statement is a query
  - But is it actually meaningful?
- The ***semantics*** of a languages specifies the actual meaning of a syntactically correct string
- Whether it is semantically legal depends on
  - Is `a` a field name?
  - Are `t1`, `t2` the names of tables?
  - Is `b` the name of a numeric field?
- Semantic information is stored in the database's metadata (catalog)

Syntax vs. Semantics in VanillaCore

- `Parser` converts a SQL statement to SQL data based on the syntax
  - Exceptions are thrown upon syntax error
  - Outputs SQL data, e.g., `QueryData`, `InsertData`, `ModifyData`, `CreatTableData`, etc.
  - All defined in `query.parse` package
- `Verifier` examines the metadata to validate the semantics of SQL data
  - Defined in `query.planner` package

Outline

- Overview
- Scans and plans
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - **Lexer**, parser, and SQL data
  - Predicates
  - Verifier
- Query planning
  - Deterministic planners

Parsing SQL Commands

- `Parser` uses a ***parsing algorithm*** to convert a SQL string to SQL data
  - To be detailed later
- Uses a ***lexical analyzer*** (also called ***lexer*** or tokenizer) that splits the SQL string into tokens when reading

`SELECT` `a` `FROM` `t1` `,` `t2` `WHERE` `b` `=` `3`

Tokens

- Each token has a ***type*** and a ***value***
- VanillaCore lexical analyzer supports five token types:
  - Single-character ***delimiters***, such as the comma `,`
  - ***Numeric constants***, such as `123.6` (scientific notation is not supported)
  - ***String constants***, such as `'netdb'`
  - ***Keywords***, such as `SELECT`, `FROM`, and `WHERE`
  - ***Identifiers***, such as `t1`, `a`, and `b`
- E.g.,

```
SELECT a FROM t1, t2 WHERE b = 3
```

Type	Value
Keyword	SELECT
Identifier	a
Keyword	FROM
Identifier	t1
Delimiter	,
Identifier	t2
Keyword	WHERE
Identifier	b
Delimiter	=
Numeric Constant	3

Whitespace

- A SQL command is split at whitespace characters
  - E.g., spaces, tabs, new lines, etc.
- The only exception are those inside '...'

Stream-based API

- Reads a SQL string only ***once***
- `matchXXX`
  - Returns whether the next token is of the specified type
- `eatXXX`
  - Returns the value of the next token if the token is of the specified type
  - Otherwise throws `BadSyntaxException`

Lexer
- keywords : Collection<String> - tok : StreamTokenizer
+ Lexer(s : String) + matchDelim(delimiter : char) : boolean + matchNumericConstant() : boolean + matchStringConstant() : boolean + matchKeyword(keyword : String) : boolean + matchId() : boolean + eatDelim(delimiter : char) + eatNumericConstant() : double + eateStringConstant() : String + eatKeyword(keyword : String) + eatId() : String

Implementing the Lexical Analyzer

- Java SE offers 2 built-in tokenizers
- `java.util.StringTokenizer`
  - Supports only two kinds of token: delimiters and words
- **`java.io.StreamTokenizer`**
  - Has an extensive set of token types, including all five types used by VanillaCore
  - Wrapped by `Lexer` in VanillaDB

Lexer

```
public class Lexer {
    private Collection<String> keywords;
    private StreamTokenizer tok;

    public Lexer(String s) {
        initKeywords();
        tok = new StreamTokenizer(new StringReader(s));
        tok.wordChars('_', '_');
        tok.ordinaryChar('.');
        // ids and keywords are converted into lower case
        tok.lowerCaseMode(true); // TT_WORD
        nextToken();
    }

    public boolean matchDelim(char delimiter) {
        return delimiter == (char) tok.ttype;
    }

    public boolean matchNumericConstant() {
        return tok.ttype == StreamTokenizer.TT_NUMBER;
    }
```

Lexer

```
public boolean matchStringConstant() {
    return '\'' == (char) tok.ttype; // 'string'
}

public boolean matchKeyword(String keyword) {
    return tok.ttype == StreamTokenizer.TT_WORD
    && tok.sval.equals(keyword) && keywords.contains(tok.sval);
}

public double eatNumericConstant() {
    if (!matchNumericConstant())
        throw new BadSyntaxException();
    double d = tok.nval;
    nextToken();
    return d;
}

public void eatKeyword(String keyword) {
    if (!matchKeyword(keyword))
        throw new BadSyntaxException();
    nextToken();
}
```

Setting Up `StreamTokenizer`

- The constructor for `Lexer` sets up a stream tokenizer as follows:
  - `tok.ordinaryChar('.')` tells the tokenizer to interpret the period character as a delimiter
  - `tok.lowerCaseMode(true)` tells the tokenizer to convert all string tokens (but not quoted strings) to lower case

Outline

- Overview
- Scans and plans
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Query planning
  - Deterministic planners

Grammar

- A ***grammar*** is a set of rules that describe how tokens can be legally combined
  - We have already seen the supported SQL grammar by VanillaCore
- E.g.,

```
<Field>      := IdTok
<Constant>   := StrTok | NumericTok
<Expression> := <Field> | <Constant>
<Term>       := <Expression> = <Expression>
<Predicate>  := <Term> [ AND <Predicate> ]
```

  - Each grammar rule specifies the ***syntactic category*** and its ***content***

Grammar

- ***Syntactic category*** is the left side of a grammar rule, and it denotes a particular concept in the language
  - <Field> as field name
- ***The content*** of a category is the right side of a grammar rule, and it is the set of strings that satisfy the rule
  - `IdTok` matches any identifier token

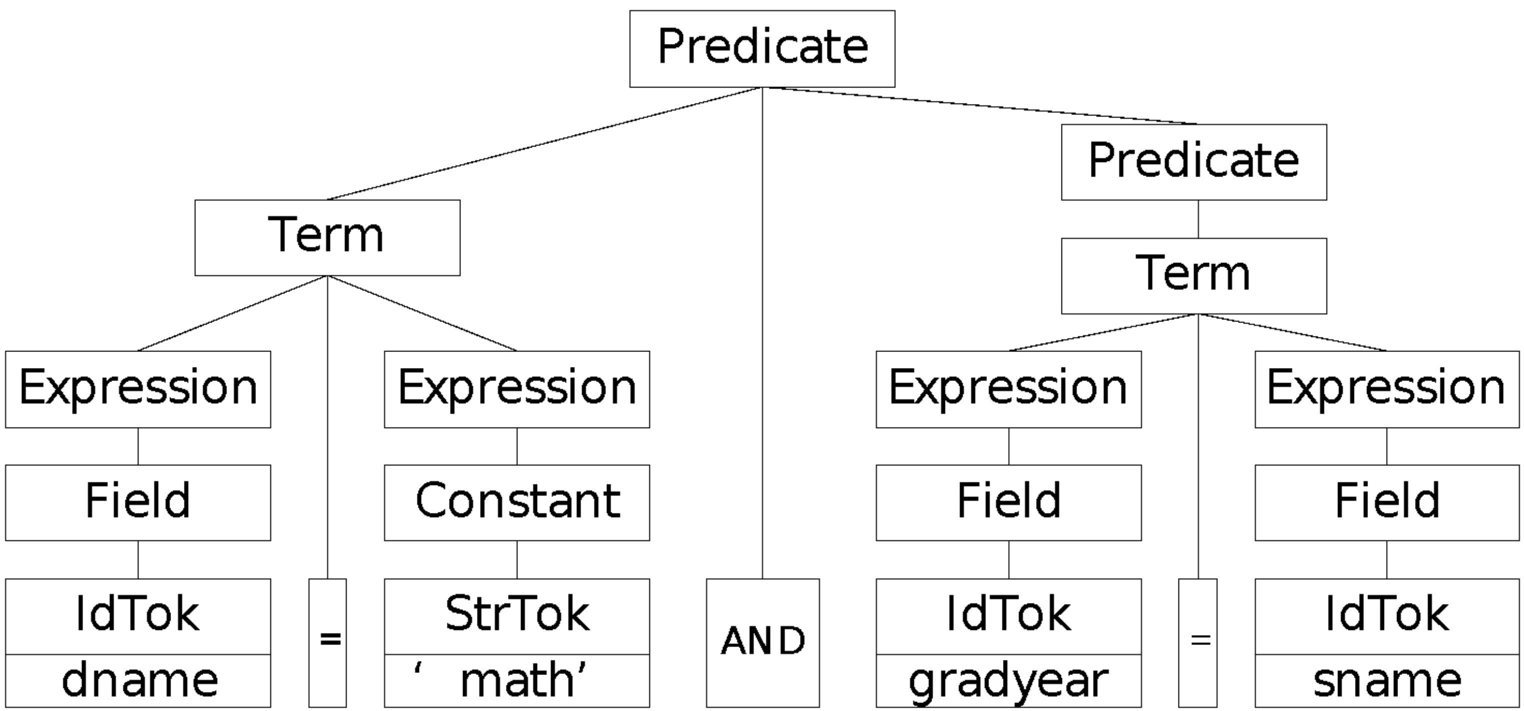
Parse Tree

- We can draw a ***parse tree*** to depict how a string belongs to a particular syntactic category
  - Syntactic categories as its internal nodes, and tokens as its leaf nodes
  - The children of a category node correspond to the application of a grammar rule
- Used by a ***parsing algorithm*** to verify if a given string is syntactically legal
  - An exception is fired if the tree cannot be constructed following the grammar

Parse Tree

- Parse tree for a predicate string:

`dname = 'math' AND gradyear = sname`



Parsing Algorithm

- The complexity of the ***parsing algorithm*** is usually in proportion to the complexity of supported grammar
- VanillaCore has simple SQL grammar, and so we will use the simplest parsing algorithm, known as ***recursive descent***

Recursive-Descent Parser

- A ***recursive-descent parser*** has a method for each grammar rule, and calls these methods recursively to traverse the parse tree ***in prefix order***

Recursive-Descent Parser

```
public class PredParser {
    private Lexer lex;

    public PredParser(String s) {
        lex = new Lexer(s);
    }

    public void field() {
        lex.eatId();
    }

    public Constant constant() {
        if (lex.matchStringConstant())
            return new VarcharConstant(lex.eatStringConstant());
        else
            return new DoubleConstant(lex.eatNumericConstant());
    }
```

```
<Field>
        := IdTok
<Constant>
        := StrTok | NumericTok
```

```java
    public Expression queryExpression() {
        return lex.matchId() ? new FieldNameExpression(id()) :
                new ConstantExpression(constant());
    }

    public Term term() {
        Expression lhs = queryExpression();
        Term.Operator op;
        if (lex.matchDelim('=')) {
            lex.eatDelim('=');
            op = OP_EQ;
        } else if (lex.matchDelim('>')) {
            lex.eatDelim('>');
            if (lex.matchDelim('=')) {
                lex.eatDelim('=');
                op = OP_GTE;
            } else
                op = OP_GT;
        } else ...
        Expression rhs = queryExpression();
        return new Term(lhs, op, rhs);
    }

    public Predicate predicate() {
        Predicate pred = new Predicate(term());
        while (lex.matchKeyword("and")) {
            lex.eatKeyword("and");
            pred.conjunctWith(term());
        }
        return pred;
    }
}
```

```
<Expression>
   := <Field> | <Constant>
<Term>
   := <Expression> = <Expression>
<Predicate>
   := <Term> [ AND <Predicate> ]
```

- Prefix traversal allows a SQL string to be read just once

SQL Data

- Parser returns SQL data
  - E.g., when the parsing the query statement (syntactic category `<Query>`), parser will returns a `QueryData` object
- All SQL data are defined in `query.parse` package

Parser and QueryData

Parser

- lex : Lexer

+ Parser(s : String)
+ updateCmd() : Object
+ query() : QueryData

- id() : String
- constant() : Constant
- queryExpression() : Expression
- term() : Term
- predicate() : Predicate
...
- create() : Object
- delete() : DeleteData
- insert() : InsertData
- modify() : ModifyData
- createTable() : CreateTableData
- createView() : CreateViewData
- createIndex() : CreateIndexData

QueryData

+ QueryData(projFields : Set<String>, tables : Set<String>, pred : Predicate, groupFields : Set<String>, aggFn : Set<AggregationFn>, sortFields : List<String>, sortDirs : List<Integer>)

+ projectFields() : Set<String>
+ tables() : Set<String>
+ pred() : Predicate
+ groupFields() : Set<String>
+ aggregationFn() : Set<String>
+ sortFields() : List<String>
+ sortDirs() : List<Integer>
+ toString() : String

Other SQL data

InsertData

+ InsertData(tblname : String, flds : List<String>, vals : List<Constant>)
+ tableName() : String
+ fields() : List<String>
+ val() : List<Constant>

CreateTableData

+ InsertData(tblname : String, sch : Schema)
+ tableName() : String
+ newSchema : Schema

Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

Predicate

```
<Field>        := IdTok
<Constant>     := StrTok | NumericTok
<Expression>   := <Field> | <Constant>
<Term>         := <Expression> = <Expression>
<Predicate>    := <Term> [ AND <Predicate> ]
```

- Classes defined in `sql.predicates` in VanillaCore
- For example,

Predicate

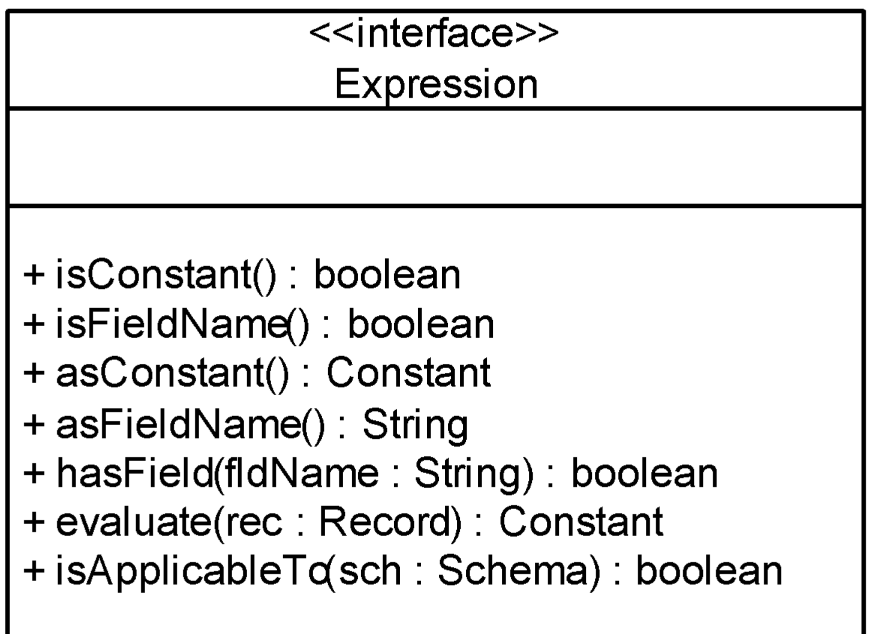
```
(gradyear > 2012 OR gradyear <= 2015) AND majorid = did
```

Expression

Term

Expression

- VanillaCore has three `Expression` implementations
  - `ConstanExpression`
  - `FieldNameExpression`
  - `BinaryArithmeticExpression`



Methods of `Expression`

- The method `evaluate(rec)` returns the value (of type `Constant`) of the expression with respect to the passed record
  - Used by, e.g., `SelectScan` during query evaluation
- The methods `isConstant`, `isFieldName`, `asConstant`, and `asFieldName` allow clients to get the contents of the expression, and are used by planner in analyzing a query
- The method `isApplicableTo` tells the planner whether the expression mentions fields only in the specified schema

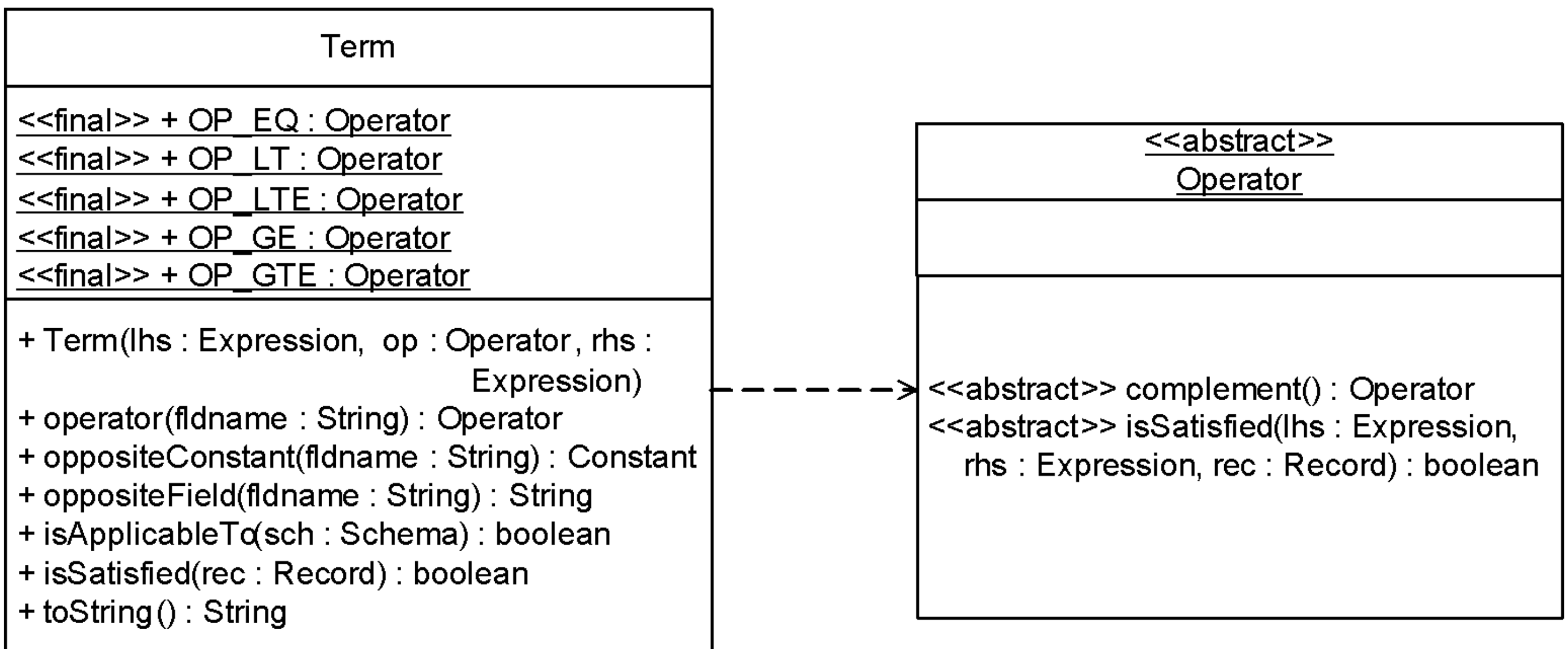
Methods of `Expression`

- `FieldNameExpression`

```
public class FieldNameExpression implements Expression {
    private String fldName;

    public FieldNameExpression(String fldName) {
    this.fldName = fldName;
    }
    ...

    public Constant evaluate(Record rec) {
    return rec.getVal(fldName);
    }

    public boolean isApplicableTo(Schema sch) {
    return sch.hasField(fldName);
    }
    ...
```

Term

- Term supports five operators
  - OP_EQ(=), OP_LT(<), OP_LTE(<=), OP_GE(>), and OP_GTE(>=)



Methods of `Term`

- The method `isSatisfied(rec)` returns true if given the specified record, the two expressions evaluate to matching values

`Term5: created = 2012/11/15`

	blog_id	ur l	created	author_id
X	33981	…	2009/10/31	729
O	33982	…	2012/11/15	730
X	41770	…	2012/10/20	729

```
public boolean isSatisfied(Record rec) {
    return op.isSatisfied(lhs, rhs, rec);
}
```

Operator in Term

- Implement the supported operators of term
- OP_LTE

```
public static final Operator OP_LTE = new Operator() {
    Operator complement() {
        return OP_GTE;
    }

    boolean isSatisfied(Expression lhs, Expression rhs, Record rec) {
        return lhs.evaluate(rec).compareTo(rhs.evaluate(rec)) <= 0;
    }

    public String toString() {
        return "<=";
    }
};
```

Methods of `Term`

- The method `oppositeConstant` returns a constant if this term is of the form "*F<OP>C*" where *F* is the specified field, *<OP>* is an operator, and *C* is some constant
- Examples:

```
Term1: majorid > 5
       // the opposite constant of majorid is 5

Term2: 2012 <= gradyear
       // the opposite constant of gradyear is 2012
```

Methods of `Term`

- The method `oppositeConstant` returns a constant if this term is of the form "*F<OP>C*" where *F* is the specified field, *<OP>* is an operator, and *C* is some constant

```
public Constant oppositeConstant(String fldName) {
    if (lhs.isFieldName() && lhs.asFieldName().equals(fldName)
            && rhs.isConstant())
        return rhs.asConstant();
    if (rhs.isFieldName() && rhs.asFieldName().equals(fldName)
            && lhs.isConstant())
        return lhs.asConstant();
    return null;
}
```

Methods of `Term`

- The method `oppositeField` returns a field name if this term is of the form "*F1<OP>F2*" where *F1* is the specified field, *<OP>* is an operator, and *F2* is another field
- Examples:

```
Term1: majorid > 5
       // the opposite field of “majorid” is null

Term3: since = gradyear
       // the opposite field of gradyear is since
       // the opposite field of since is gradyear
```

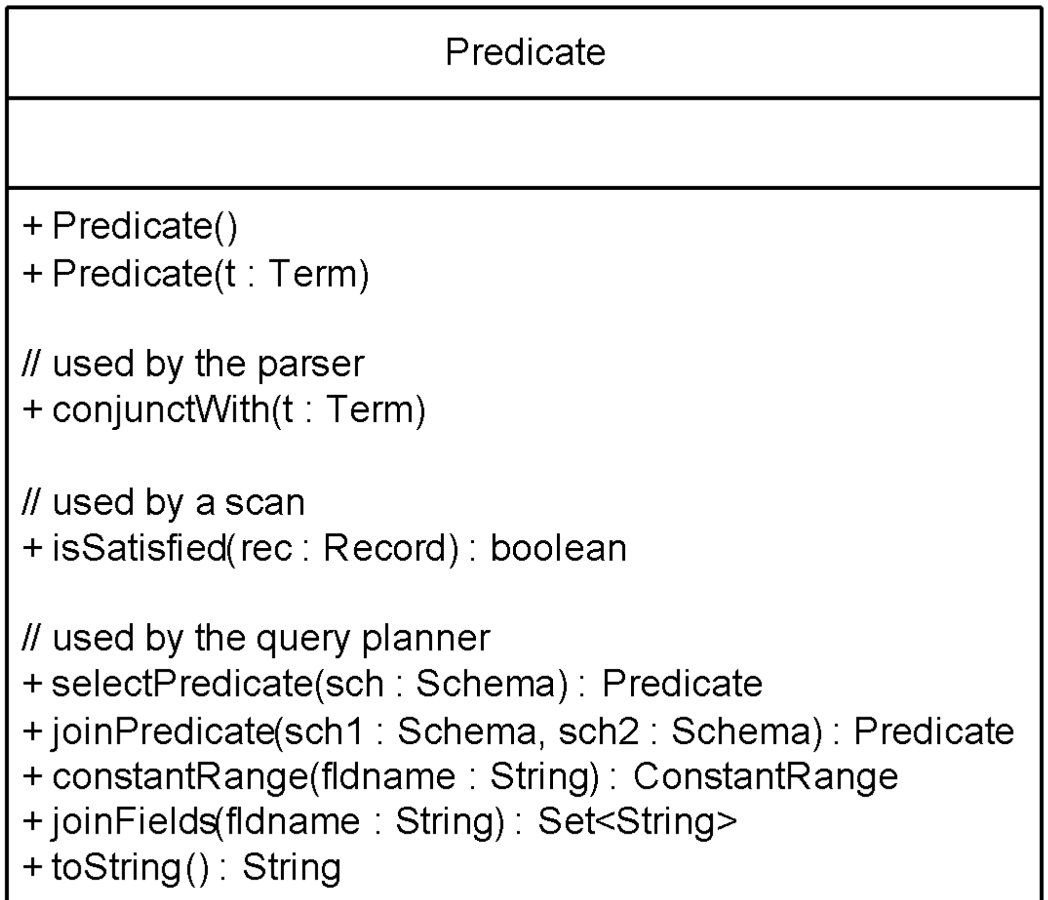
Methods of `Term`

- The method `isApplicableTo` tells the planner whether *both* expressions of this term apply to the specified schema
- Examples:

```
Table s with schema(sid, sname, majorid)
Table d with schema(did, dname)

Term1: majorid > 5
       // it is not applicable to d.schema
       // it is applicable to s.schema

Term4: majorid = did
       // it is not applicable to d.schema
       // it is not applicable to s.schema
```

Predicate

- A predicate in VanillaCore is a conjunct of terms, e.g., *term1 AND term2 AND ...*



Methods of `Predicate`

- The methods of `Predicate` address the needs of several parts of the database system:
  - A select scan evaluates a predicate by calling `isSatisfied`
  - The parser construct a predicate as it processes the WHERE clause, and it calls `conjoinWith` to conjoin another term
  - The rest of the methods help the query planner to analyze the scope of a predicate and to break it into smaller pieces

Methods of `Predicate`

- The method `selectPredicate` returns a sub-predicate that applies only to the specified schema
- Example:

```
Table s with schema(sid, sname, majorid)
Table d with schema(did, dname)

Predicate1:
      majorid = did AND majorid > 5 AND sid >= 100
      // the select predicate for table s: majorid > 5
                    AND sid >= 100
      // the select predicate for table d: null
```

Methods of `Predicate`

- The method `selectPredicate` returns a sub-predicate that applies only to the specified schema

```
public Predicate selectPredicate(Schema sch) {
   Predicate result = new Predicate();
   for (Term t : terms)
      if (t.isApplicableTo(sch))
         result.terms.add(t);
   if (result.terms.size() == 0)
      return null;
   else
      return result;
}
```

Methods of `Predicate`

- The method `joinPredicate` returns a sub-predicate that applies to the union of the two specified schemas, but not to either schema individually

```
Table s with schema(sid, sname, majorid)
Table d with schema(did, dname)

Predicate1:
      majorid = did AND majorid > 5 AND sid >= 100
      // the join predicate for tables s, d: majorid = did
```

Methods of `Predicate`

- The method `joinPredicate` returns a sub-predicate that applies to the union of the two specified schemas, but not to either schema separately

```
public Predicate joinPredicate(Schema sch1, Schema sch2) {
    Predicate result = new Predicate();
    Schema newsch = new Schema();
    newsch.addAll(sch1);
    newsch.addAll(sch2);
    for (Term t : terms)
        if (!t.isApplicableTo(sch1) && !t.isApplicableTo(sch2)
                && t.isApplicableTo(newsch))
            result.terms.add(t);
    return result.terms.size() == 0 ? null : result;
}
```

Methods of `Predicate`

- The method `constantRange` determines if the specified field is constrained by a constant range in this predicate. If so, the method returns that range

```
Predicate2: sid > 5 AND sid <= 100
      // the constant range of sid is (5, 100]
```

Methods of `Predicate`

- The method `joinFields` determines if there are terms of the form "*F1=F2*" or result in "*F1=F2*" via equal transitivity, where *F1* is the specified field and *F2* is another field. If so, the method returns the names of all join fields

```
Predicate3: sid = did AND did = tid
        // the join fields of sid are {did, tid}
```

Creating a Predicate in a Query Parser

```
// majorid <=30 AND majorid=did
Expression exp1 = new FieldNameExpression("majorid");
Expression exp2 = new ConstantExpression(
       new IntegerConstant(30));
Term t1 = new Term(exp1, OP_LTE, exp2);

Expression exp3 = new FieldNameExpression("majorid");
Expression exp4 = new FieldNameExpression("did");
Term t2 = new Term(exp3, OP_EQ, exp4);

Predicate pred = new Predicate(t1);
pred.conjunctWith(t2);
```

Outline

- Overview
- Scans and plans
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Query planning
  - Deterministic planners

Things that Parser Cannot Ensure

- The parser cannot enforce type compatibility, because it doesn't know the types of the identifiers it sees

```
dname = 'math' AND gradyear = sname
```

- The parser also cannot enforce compatible list size

```
INSERT INTO dept (did, dname) VALUES ('math')
```

Verification

- Before feeding the SQL data into the plans/scans, the planner asks the `Verifier` to verify the semantics correctness of the data

Verification

- The `Verifier` checks whether:
  - The mentioned tables and fields actually exist in the catalog
  - The mentioned fields are not ambiguous
  - The actions on fields are type-correct
  - All constants are of correct type and size to their corresponding fields

Verifying the INSERT Statement

```
public static void verifyInsertData(InsertData data, Transaction tx) {
      // examine table name
      TableInfo ti = VanillaDb.catalogMgr().getTableInfo(data.tableName(), tx);
      if (ti == null)
            throw new BadSemanticException("table " + data.tableName() + " does not exist");

      Schema sch = ti.schema();
      List<String> fields = data.fields();
      List<Constant> vals = data.vals();

      // examine whether values have the same size with fields
      if (fields.size() != vals.size())
            throw new BadSemanticException("#fields and #values does not match");

      // verify field existence and type
      for (int i = 0; i < fields.size(); i++) {
            String field = fields.get(i);
            Constant val = vals.get(i);
            // check field existence
            if (!sch.hasField(field))
                  throw new BadSemanticException("field " + field+ " does not exist");
            // check whether field matches value type
            if (!verifyConstantType(sch, field, val))
                  throw new BadSemanticException("field " + field
                              + " doesn't match corresponding value in type");
      }
}
```

Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Predicates
  - Lexer, parser, and SQL data
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

What does a `Planner` do?

1. Parses the SQL command
2. Verifies the SQL command
3. Finds a good ***plan*** for the SQL command
4. a. Returns the plan (`createQueryPlan()`)
   b. Executes the plan by iterating through the ***scan*** and returns #records affected (`executeUpdate()`)

What’s the difference between scans and plans?

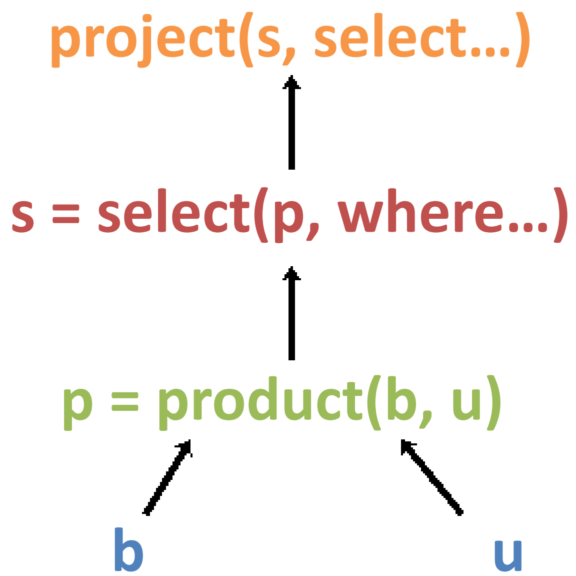
Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Predicates
  - Lexer, parser, and SQL data
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

SQL and Relational Algebra (1/2)

- Recall that a SQL command can be expressed as at-least one tree in relational algebra

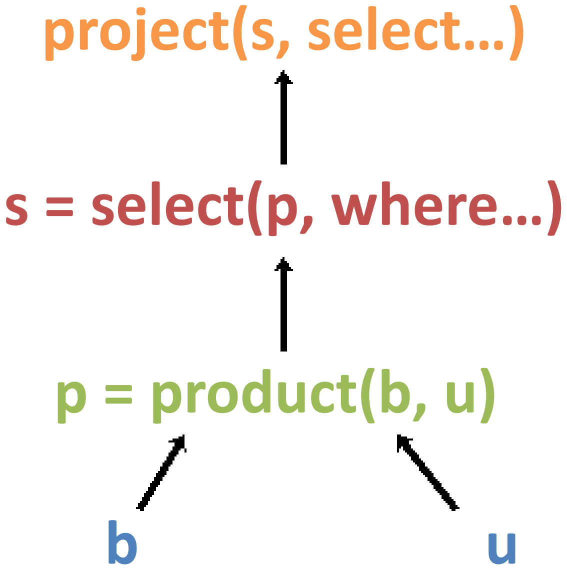
```
SELECT b.blog_id
    FROM blog_pages b, users u
    WHERE b.author_id=u.user_id
      AND u.name='Steven Sinofsky'
      AND b.created >= 2011/1/1;
```



Why this translation?

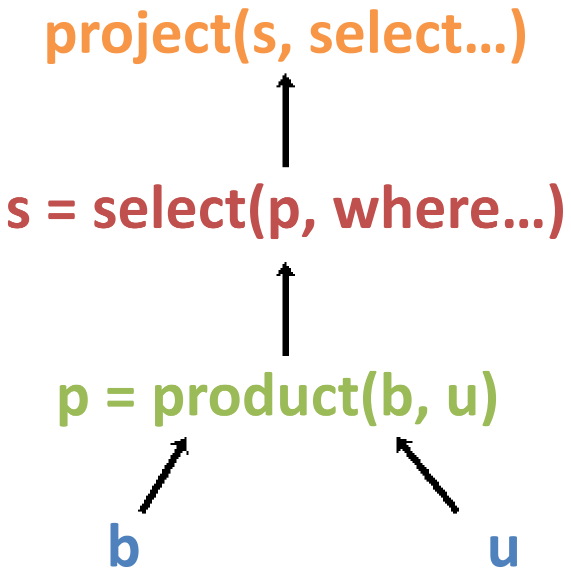
SQL and Relational Algebra (2/2)

- SQL is difficult to implement directly
  - A single SQL command can embody several tasks
- Relational algebra is relatively easy to implement
  - Each ***operator*** denotes a small, well-defined task



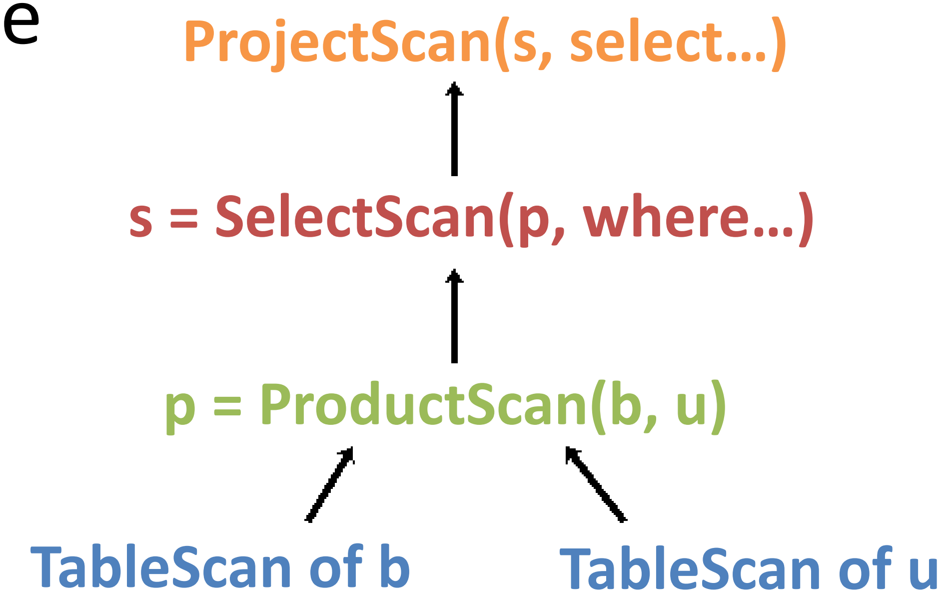
Operators

- Single-table operators
  - select, project, sort, rename, extend, groupby, etc.
- Two-table operators
  - product, join, semijoin, etc.
- Operands
  - Tables, views, output of other operators, predicates, etc.
- Output
  - Always a table
  - To be returned or used as a param of the next op



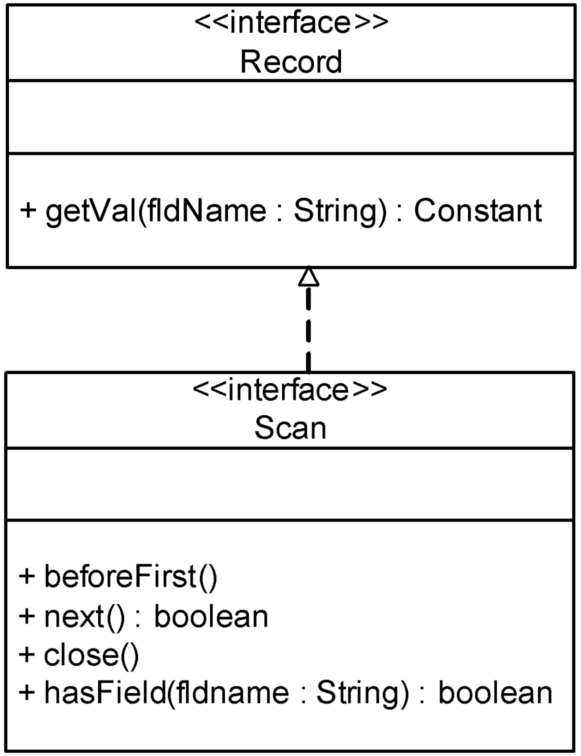
Scans

- A ***scan*** represents the output of an operator in a relational algebra tree
  - I.e., output of a subtree (***partial query***)
- All scans in VanillaCore implement the `Scan` interface
- In `query.algebra` package



The `Scan` Interface

- An iterator of output records of a partial query
- Not to confuse with `RecordFile`
  - A `RecordFile` is an iterator of records in a ***table file***
  - Storage-specific



Using a Scan

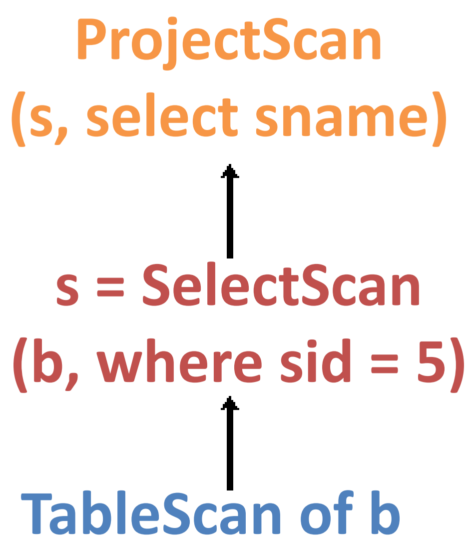
```
public static void printNameAndGradyear(Scan s) {
    s.beforeFirst();
    while (s.next()) {
        Constant sname = s.getVal("sname");
        Constant gradyear = s.getVal("gradyear");
        System.out.println(sname + "\t" + gradyear);
    }
    s.close();
}
```

Basic Scans

```
public SelectScan(Scan s, Predicate pred);

public ProjectScan(Scan s,
     Collection<String> fieldList);

public ProductScan(Scan s1, Scan s2);

public TableScan(TableInfo ti, Transaction tx);
```

Building a Scan Tree

```
VanillaDb.init("studentdb");
Transaction tx =
    VanillaDb.txMgr().transaction(
    Connection.TRANSACTION_SERIALIZABLE, true);
TableInfo ti =
    VanillaDb.catalogMgr().getTableInfo("b", tx);

Scan ts = new TableScan(ti, tx);
Predicate pred = new Predicate("..."); // sid = 5

Scan ss = new SelectScan(ts, pred);
Collection<String> projectFld =
    Arrays.asList("sname");
Scan ps = new ProjectScan(ss, projectFld);

ps.beforeFirst();
while (ps.next())
    System.out.println(ps.getVal("sname"));
ps.close();
```



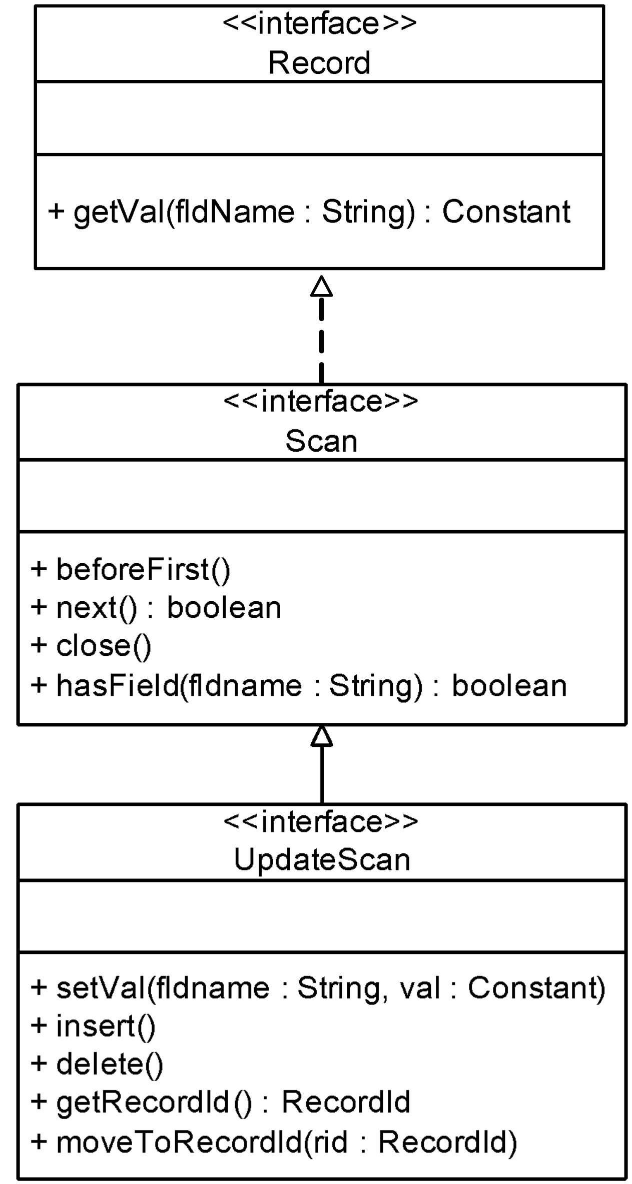
Updatable Scans

- A scan is read-only by default
- We need the `TableScan` and `SelectScan` to be ***updatable*** to support UPDATE and DELETE commands:

```
UPDATE student
        SET major-id = 10, grad-year = grad-year - 1
        WHERE major-id=20;

DELETE FROM student
        WHERE major-id=20;
```

UpdateScan

- Provides setters
- Allows random access
  - Useful to indices
- Implemented by `TableScan` and `SelectScan`
- Not every scan is updatable
  - A scan is updatable only if every record *r* in the scan has a corresponding record *r'* in underlying database table



Using Updatable Scans

- SQL command:

```
UPDATE enroll SET grade = 'A+'
              WHERE section-id = 53;
```

- Code:

```
VanillaDb.init("studentdb");
Transaction tx = VanillaDb.txMgr().newTransaction(
       Connection.TRANSACTION_SERIALIZABLE, false);
TableInfo ti = VanillaDb.catalogMgr().getTableInfo("enroll",
tx);

Scan ts = new TableScan(ti, tx);
Predicate pred = new Predicate(...); // section-id = 53
UpdateScan us = new SelectScan(ts, pred);
us.beforeFirst();
while (us.next())
    us.setVal("grade", new VarcharConstant("A+"));
us.close();
```

TableScan

```
public class TableScan implements UpdateScan {
	private RecordFile rf;
	private Schema schema;

	public TableScan(TableInfo ti, Transaction tx) {
		rf = ti.open(tx);
		schema = ti.schema();
	}

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

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

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

	public Constant getVal(String fldName) {
		return rf.getVal(fldName);
	}

	public boolean hasField(String fldName) {
		return schema.hasField(fldName);
	}

	public void setVal(String fldName, Constant val) {
		rf.setVal(fldName, val);
	}
	...
}
```

- Basically, tasks are delegated to a `RecordFile`

SelectScan

```
public class SelectScan implements UpdateScan {
    private Scan s;
    private Predicate pred;

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

    public boolean next() {
        while (s.next())
            // if current record satisfied the predicate
            if (pred.isSatisfied(s))
                return true;
        return false;
    }

    public void setVal(String fldname, Constant val) {
        UpdateScan us = (UpdateScan) s;
        us.setVal(fldname, val);
    }
    ...
}
```

ProductScan

```java
public class ProductScan implements Scan {
	private Scan s1, s2;
	private boolean isLhsEmpty;

	public ProductScan(Scan s1, Scan s2) {
		this.s1 = s1;
		this.s2 = s2;
		s1.beforeFirst();
		isLhsEmpty = !s1.next();
	}

	public boolean next() {
		if (isLhsEmpty)
			return false;
		if (s2.next())
			return true;
		else if (!(isLhsEmpty = !s1.next())) {
			s2.beforeFirst();
			return s2.next();
		} else
			return false;
	}

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

- Iterates through records following the ***nested loops***

ProjectScan

```
public class ProjectScan implements Scan {
    private Scan s;
    private Collection<String> fieldList;

    public ProjectScan(Scan s, Collection<String> fieldList) {
        this.s = s;
        this.fieldList = fieldList;
    }

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

    public Constant getVal(String fldName) {
        if (hasField(fldName))
            return s.getVal(fldName);
        else
            throw new RuntimeException("field " + fldName + " not found.");
    }
    ...
}
```

Example

project(s, select blog_id)

beforeFirst()

select(p, where name = ‘Picachu’
and author_id = user_id)

beforeFirst()

product(b, u)

beforeFirst()

beforeFirst()

```
SELECT blog_id FROM b, u
        WHERE name = "Picachu"
        AND author_id = user_id;
```

b

blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example

project(s, select blog_id)

next()

select(p, where name = ‘Picachu’ and author_id = user_id)

next()

✗

blog_id	url	created	author_id	user_id	name	balance
33981	…	2009/10/31	729	729	Steven Sinofsky	10,235

product(b, u)

next()

b

blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example

project(s, select blog_id)

next()

select(p, where name = 'Picachu'
and author_id = user_id)

next()

product(b, u)

next()

blog_id	url	created	author_id	user_id	name	balance
33981	…	2009/10/31	729	730	Picachu	NULL

b

blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example

project(s, select blog_id)

next()

select(p, where name = 'Picachu'
and author_id = user_id)

next()

product(b, u)

next()

next()

beforeFirst()

false

b

blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example

project(s, select blog_id)

next()

select(p, where name = ‘Picachu’ and author_id = user_id)

next()

product(b, u)

next()

blog_id	url	created	author_id	user_id	name	balance
33982	…	2012/11/15	730	729	Steven Sinofsky	10,235

b

blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example

blog_id
33982

project(s, select blog_id)

next()

blog_id	url	created	author_id	user_id	name	balance
33982	…	2012/11/15	730	730	Picachu	NULL

select(p, where name = ‘Picachu’ and author_id = user_id)

next()

blog_id	url	created	author_id	user_id	name	balance
33982	…	2012/11/15	730	730	Picachu	NULL

product(b, u)

next()

b

blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example

project(s, select…)

getVal()

blog_id
33982

select(p, where name = ‘Picachu’)

getVal()

blog_id	url	created	author_id	user_id	name	balance
33982	…	2012/11/15	730	730	Picachu	NULL

product(b, u)

getVal()

blog_id	url	created	author_id	user_id	name	balance
33981	…	2009/10/31	729	729	Steven Sinofsky	10,235
33981	…	2009/10/31	729	730	Picachu	NULL
33982	…	2012/11/15	730	729	Steven Sinofsky	10,235
33982	…	2012/11/15	730	730	Picachu	NULL
41770	…	2012/10/20	729	729	Steven Sinofsky	10,235
41770	…	2012/10/20	729	730	Picachu	NULL

b

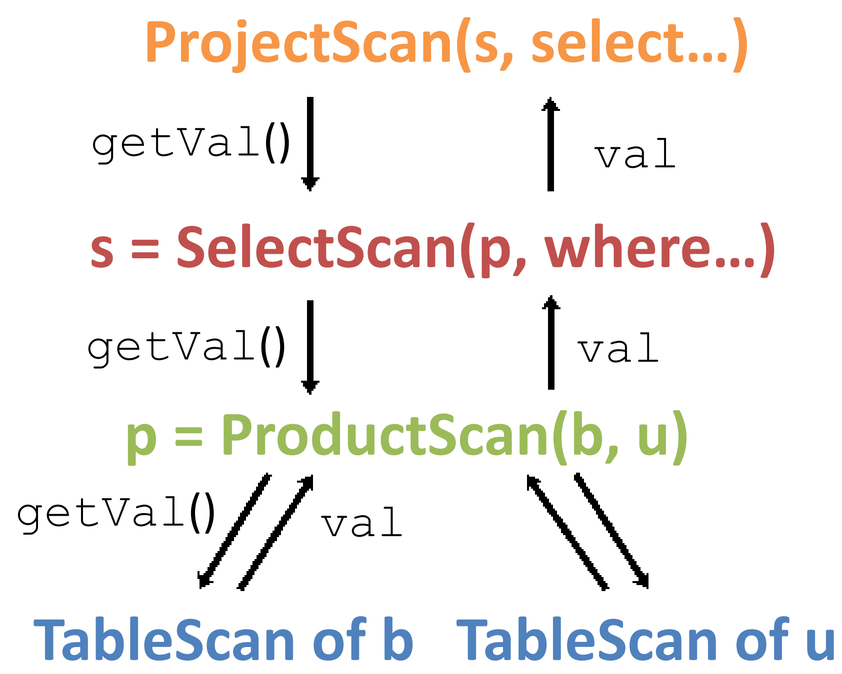
blog_id	url	created	author_id
33981	…	2009/10/31	729
33982	…	2012/11/15	730
41770	…	2012/10/20	729

u

user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Pipelined Scanning

- The above operators implement ***pipelined scanning***
  - Calling a method of a node results in recursively calling the same methods of child nodes on-the-fly
  - Records are computed one at a time as needed---no intermediate records are saved



Pipelined vs. Materialized

- Despite its simplicity, pipelined scanning is inefficient in some cases
  - E.g., when implementing `SortScan` (for `ORDER BY`)
  - Needs to iterate the entire child to find the next record
- Later, we will see ***materialized scanning*** in some scans
  - Intermediate records are materialized to a temp table (file)
  - E.g., the `SortScan` can use an external sorting algorithm to sort all records at once, save them, and return each record upon `next()` is called
- Pipelined or materialized?
  - Saving in scanning cost vs. materialization overhead

Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

Scan Tree for SQL Command?

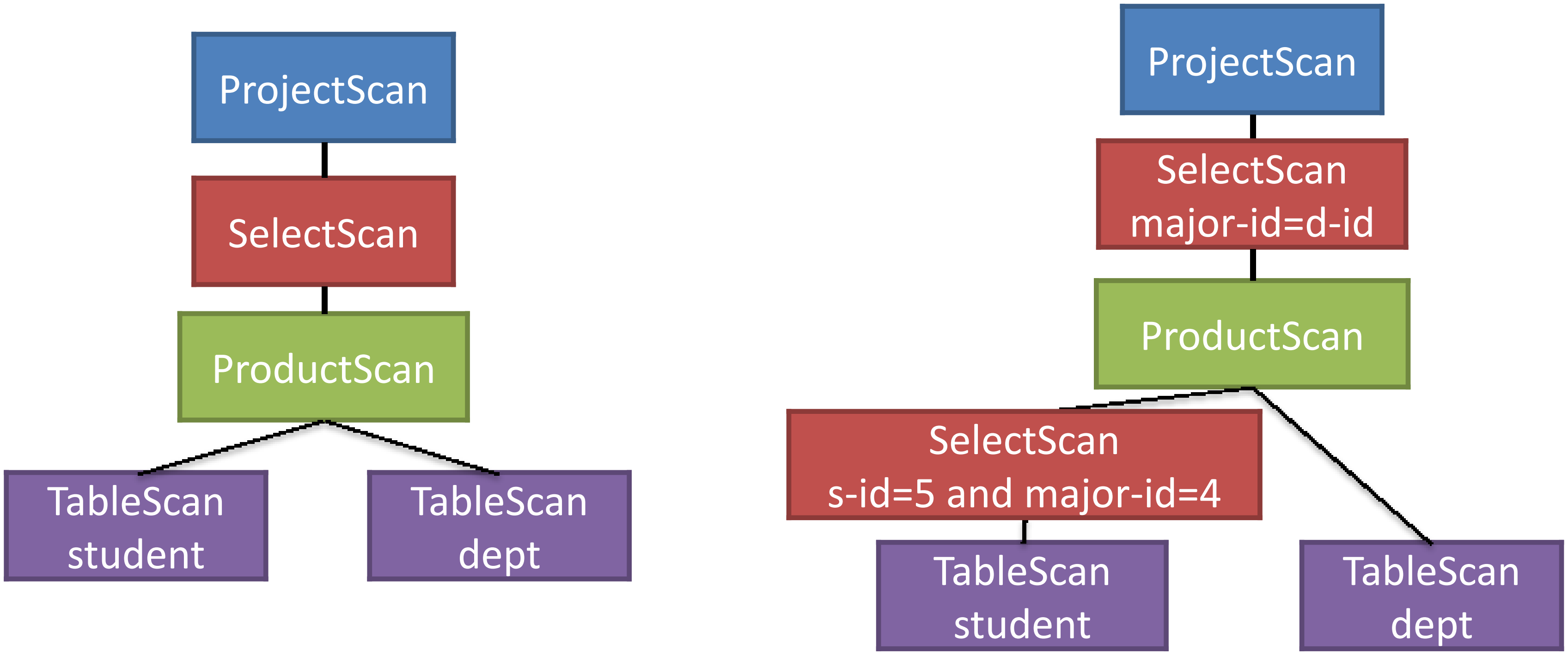
- Given the scans:

- Can you build a scan tree for this query:

```
SELECT sname FROM student, dept
        WHERE major-id = d-id
              AND s-id = 5 AND major-id = 4;
```

Which One is Better?

```
SELECT sname FROM student, dept
        WHERE major-id = d-id
              AND s-id = 5 AND major-id = 4;
```



Why Does It Matter?

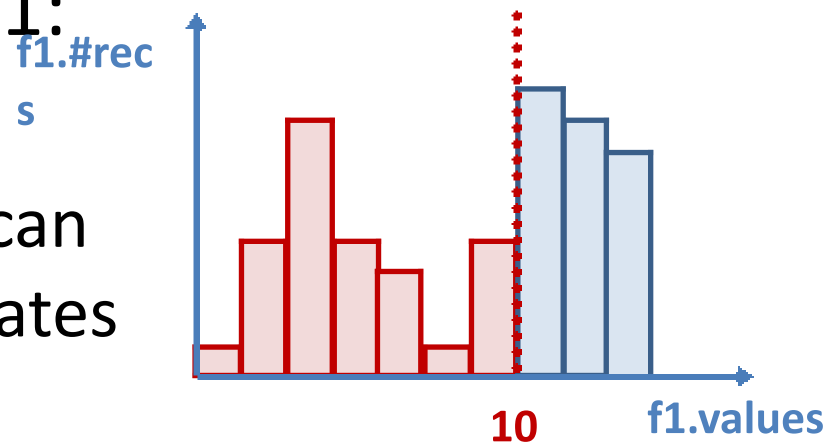
- A good scan tree can be faster than a bad one for orders of magnitude
- Consider the product scan at middle
  - Let *R*(student)=10000, *B*(student)=1000, *B*(dept)= 500, and *selectivity*(s-id=5&major-id=4)=0.01
  - Each block access requires 10ms
- Left: (1000+10000*500)*10ms = 13.9 hours
- Right: (1000+10000*0.01*500)*10ms = 8.4 mins
- We need a way to estimate the cost of a scan tree ***without actual scanning***
  - As we just did above

Which Cost to Estimate?

- CPU delay, memory delay, or I/O delay?
- The ***number of block accesses*** performed by a scan is usually the most important factor in determining running time of a query
- Usually needs other estimates, such as the ***number of output records*** and ***value histogram***

Estimating Block Access (1/2)

- E.g., SELECT(T1, WHERE f1<10)
- Statistics metadata for T1:
  - VH(T1, f1), R(T1), B(T1)
  - Updated by a full table scan every, say, 100 table updates



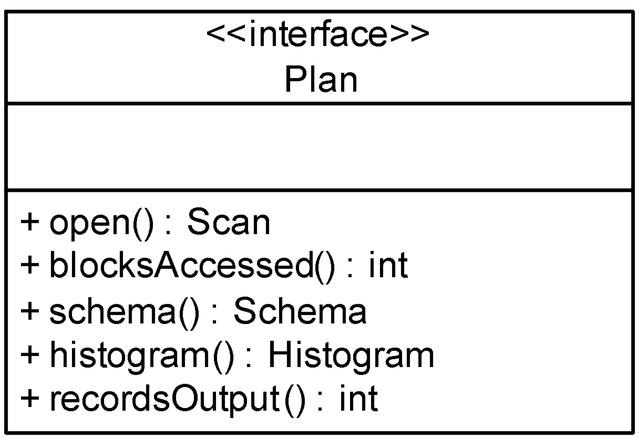
- #blocks accessed?
  - W/o index: B(T1)
  - W/ B-tree:
    - IndexSearchCost + VH(T1, f1).predHistogram(WHERE…).recordsOutput()

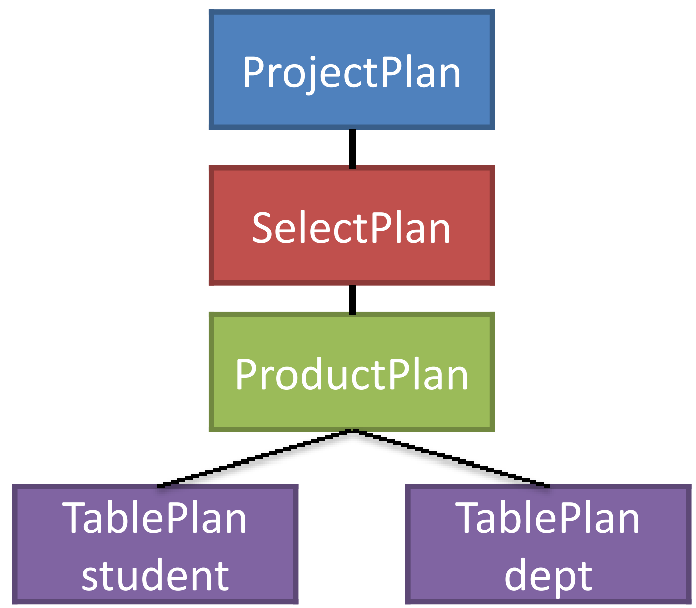
Estimating Block Access (2/2)

- Complications
  – Multiple fields in SELECT (e.g., f1=f2)
  – Multiple tables, etc.
- Topics of query optimization

The `Plan` Interface

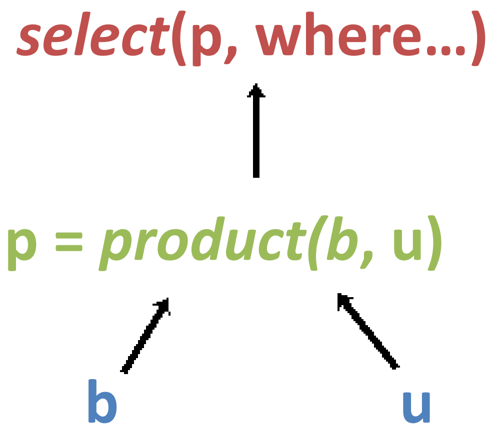
- A cost estimator for a ***partial query***
- Each plan instance corresponds to an operator in relational algebra
  - Also to a subtree





Using a Query Plan

```
VanillaDb.init("studentdb");
Transaction tx = VanillaDb.txMgr().transaction(
    Connection.TRANSACTION_SERIALIZABLE, true);

Plan pb = new TablePlan("b", tx);
Plan pu = new TablePlan("u", tx);
Plan pp = new ProductPlan(pb, pu);
Predicate pred = new Predicate(...);
Plan sp = new SelectPlan(pp, pred);

sp.blockAccessed(); // estimate #blocks accessed

// open corresponding scan only if sp has low
cost
Scan s = sp.open();
s.beforeFirst();
while (s.next())
s.getVal("bid");
s.close();
```



Plan before Scan

- A plan (tree) is a blueprint for evaluating a query
- Estimates cost by accessing statistics metadata only
  - No actual I/Os
  - Memory access only, ***very efficient***
- Once a good plan is decided, we then create a scan following the blueprint

Opening a Scan Tree

- The `open()` constructs a scan tree with the same structure as the current plan

```
public class TablePlan implements Plan {

        public Scan open() {
        return new TableScan(ti, tx);
        }
        ...
}
public class SelectPlan implements Plan {

        public SelectPlan(Plan p, Predicate pred) {
                this.p = p;
                this.pred = pred;
                ...
        }

        public Scan open() {
                Scan s = p.open();
                return new SelectScan(s, pred);
        }
        ...
}
public class ProductPlan implements Plan {

        public ProductPlan(Plan p1, Plan p2) {
                this.p1 = p1;
                this.p2 = p2;
                ...
        }

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

}
```

How to Find a Good Plan Tree?

- The planner can create multiple trees first, and then pick the one having the lowest cost
- Determining the best plan tree for a SQL command is call ***planning***

Outline

- Overview
- Parsing and Validating SQL commands
  - Syntax vs. Semantics
  - Lexer, parser, and SQL data
  - Predicates
  - Verifier
- Scans and plans
- Query planning
  - Deterministic planners

What does a `Planner` do?

1. Parses the SQL command
2. Verifies the SQL command
3. ***Finds a good plan*** for the SQL command
4. a. Returns the plan (`createQueryPlan()`)
   b. Executes the plan by iterating through the scan and returns #records affected (`executeUpdate()`)

Planning

- Input:
  - SQL data
- Output:
  - A good plan tree

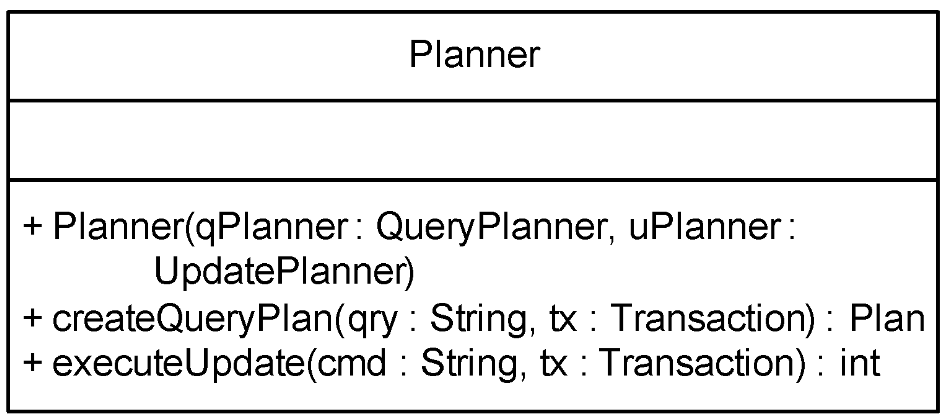
- Done by the ***planner***

Using the VanillaCore Planner

```
VanillaDb.init("studentdb");
Planner planner = VanillaDb.planner();
Transaction tx = VanillaDb.txMgr().transaction(
Connection.TRANSACTION_SERIALIZABLE, false);
// part 1: Process a query
String qry = "SELECT sname FROM student";
Plan p = planner.createQueryPlan(qry, tx);
Scan s = p.open();
s.beforeFirst();
while (s.next())
System.out.println(s.getVal("sname"));
s.close();

// part 2: Process an update command
String cmd = "DELETE FROM student WHERE majorid = 30";
int numRecs = planner.executeUpdate(cmd, tx);
System.out.println(numRecs + " students were deleted");
tx.commit();
```

Planner

- In VanillaCore, all planner implementations are placed in `query.planner` package
- A client can obtain a `Planner` object by calling `server.VanillaDb.planner()`



Query and Update Planners

- After verifying the parsed SQL data, the `Planner` delegates the planning tasks to
  - `QueryPlanner`
  - `UpdatePlanner`

  implementations
- Interfaces defined in `query.planner` package

Planner

```java
public class Planner {
    private QueryPlanner qPlanner;
    private UpdatePlanner uPlanner;

    public Planner(QueryPlanner qPlanner, UpdatePlanner uPlanner)
    {
        this.qPlanner = qPlanner;
        this.uPlanner = uPlanner;
    }

    public Plan createQueryPlan(String qry, Transaction tx) {
        Parser parser = new Parser(qry);
        QueryData data = parser.query();
        Verifier.verifyQueryData(data, tx);
        return qPlanner.createPlan(data, tx);
    }
```

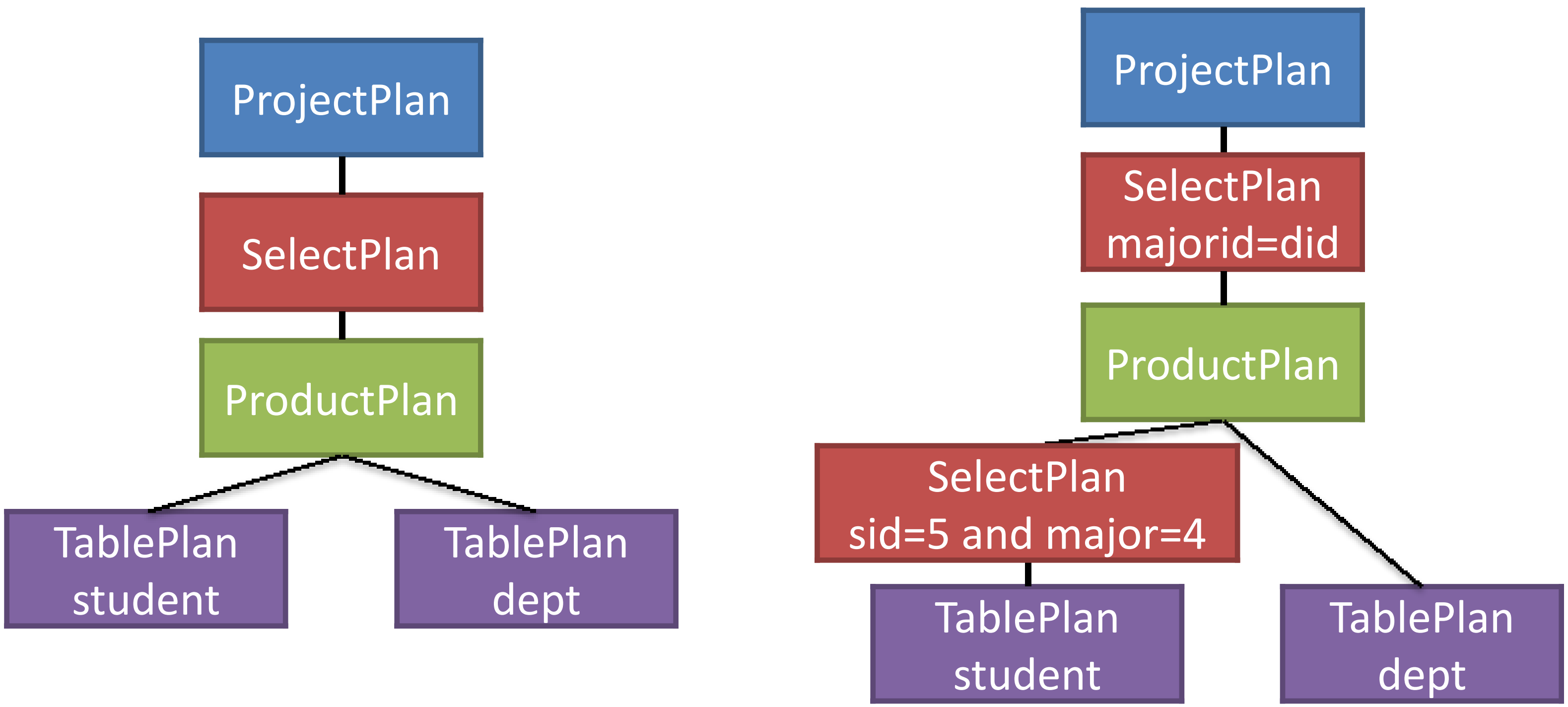
Planner

```
    public int executeUpdate(String cmd, Transaction tx) {
        if (tx.isReadOnly())
            throw new UnsupportedOperationException();
        Parser parser = new Parser(cmd);
        Object obj = parser.updateCommand();
        if (obj instanceof InsertData) {
            Verifier.verifyInsertData((InsertData) obj, tx);
            return uPlanner.executeInsert((InsertData) obj, tx);
        } else if (obj instanceof DeleteData) {
            Verifier.verifyDeleteData((DeleteData) obj, tx);
            return uPlanner.executeDelete((DeleteData) obj, tx);
        } else if (obj instanceof ModifyData) {
            Verifier.verifyModifyData((ModifyData) obj, tx);
            return uPlanner.executeModify((ModifyData) obj, tx);
        } else if (obj instanceof CreateTableData) {
            Verifier.verifyCreateTableData((CreateTableData) obj, tx);
            return uPlanner.executeCreateTable((CreateTableData) obj, tx);
        } else if (obj instanceof CreateViewData) {
            Verifier.verifyCreateViewData((CreateViewData) obj, tx);
            return uPlanner.executeCreateView((CreateViewData) obj, tx);
        } else if (obj instanceof CreateIndexData) {
            Verifier.verifyCreateIndexData((CreateIndexData) obj, tx);
            return uPlanner.executeCreateIndex((CreateIndexData) obj, tx);
        } else
            throw new UnsupportedOperationException();
    }
}
```

Query Planning

- Plan tree?

```
SELECT sname FROM student, dept
WHERE majorid = did
      AND sid = 5 AND majorid = 4
```

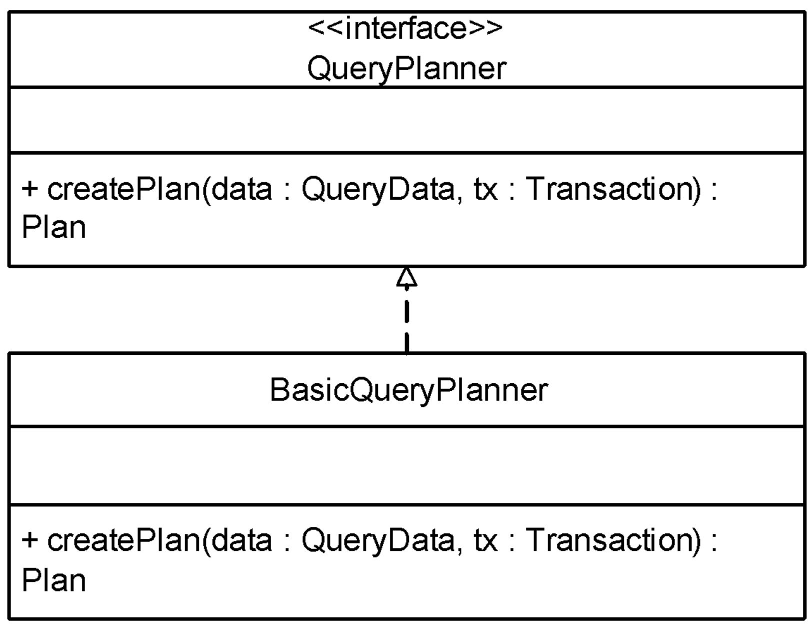


Deterministic Query Planning Algorithm

1. Construct a plan for each table *T* in the FROM clause
   a. If *T* is a table, then the plan is a table plan for *T*
   b. If *T* is a view, then the plan is the result of calling this algorithm recursively on the definition of *T*
2. Take the product of plans from Step 1 if needed
3. A Select on predicate in the WHERE clause if needed
4. Project on the fields in the SELECT clause

QueryPlanner

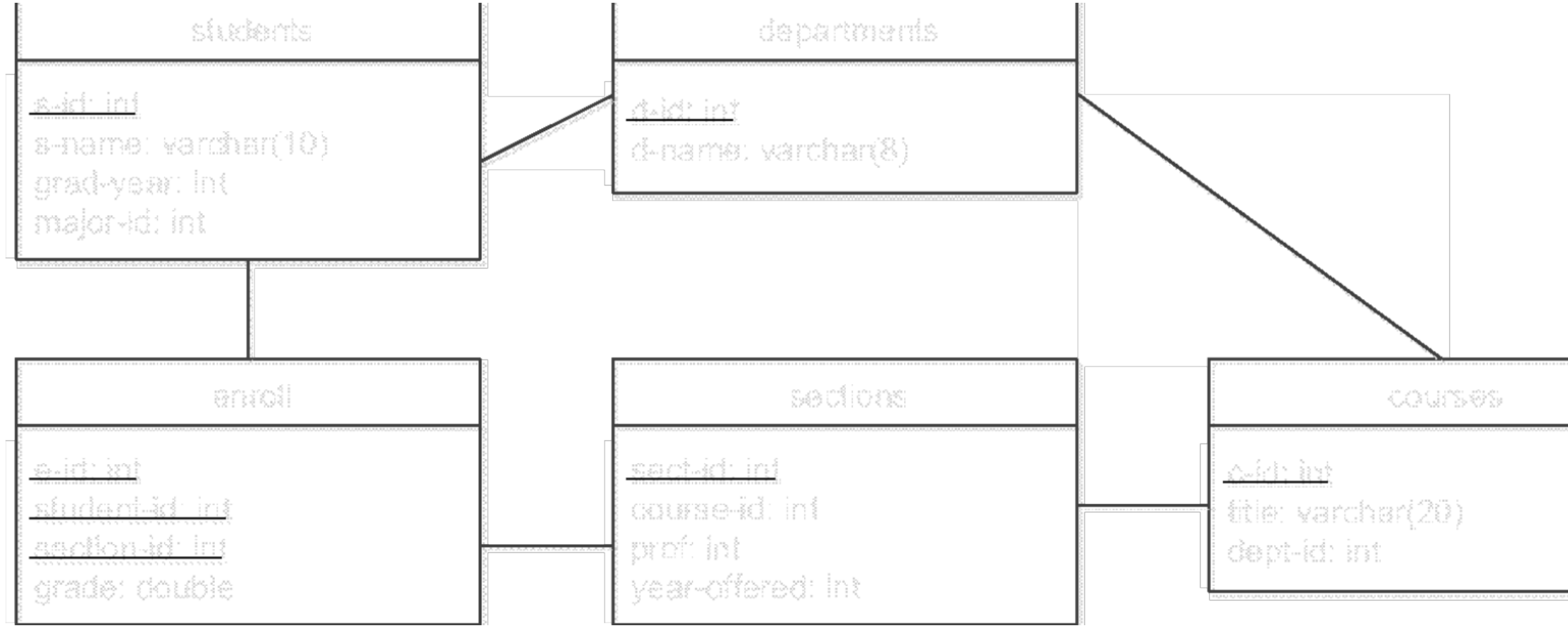
- The `BasicQueryPlanner` implements the deterministic planning algorithm
  - In `query.planner`



BasicQueryPlanner

- The simplified code:

```
public Plan createPlan(QueryData data, Transaction tx) {
	// Step 1: Create a plan for each mentioned table or view
	List<Plan> plans = new ArrayList<Plan>();
	for (String tblname : data.tables()) {
		String viewdef = VanillaDb.catalogMgr().getViewDef(tblname, tx);
		if (viewdef != null)
			plans.add(VanillaDb.planner().createQueryPlan(viewdef, tx));
		else
			plans.add(new TablePlan(tblname, tx));
	}
	// Step 2: Create the product of all table plans
	Plan p = plans.remove(0);
	for (Plan nextplan : plans)
		p = new ProductPlan(p, nextplan);

	// Step 3: Add a selection plan for the predicate
	p = new SelectPlan(p, data.pred());

	// Step 4: Project onto the specified fields
	p = new ProjectPlan(p, data.projectFields());
	return p;
}
```



Where to place GROUP BY, HAVING, and ORDER BY?

```
SELECT major-id, AVG(grade)
  FROM students, enroll
  WHERE s-id = student-id AND sec-id = ...
  GROUP BY major-id
  HAVING AVG(grade) >= 60
  ORDER BY AVG(grade) DESC;
```

Logical Planning Order (Bottom Up)

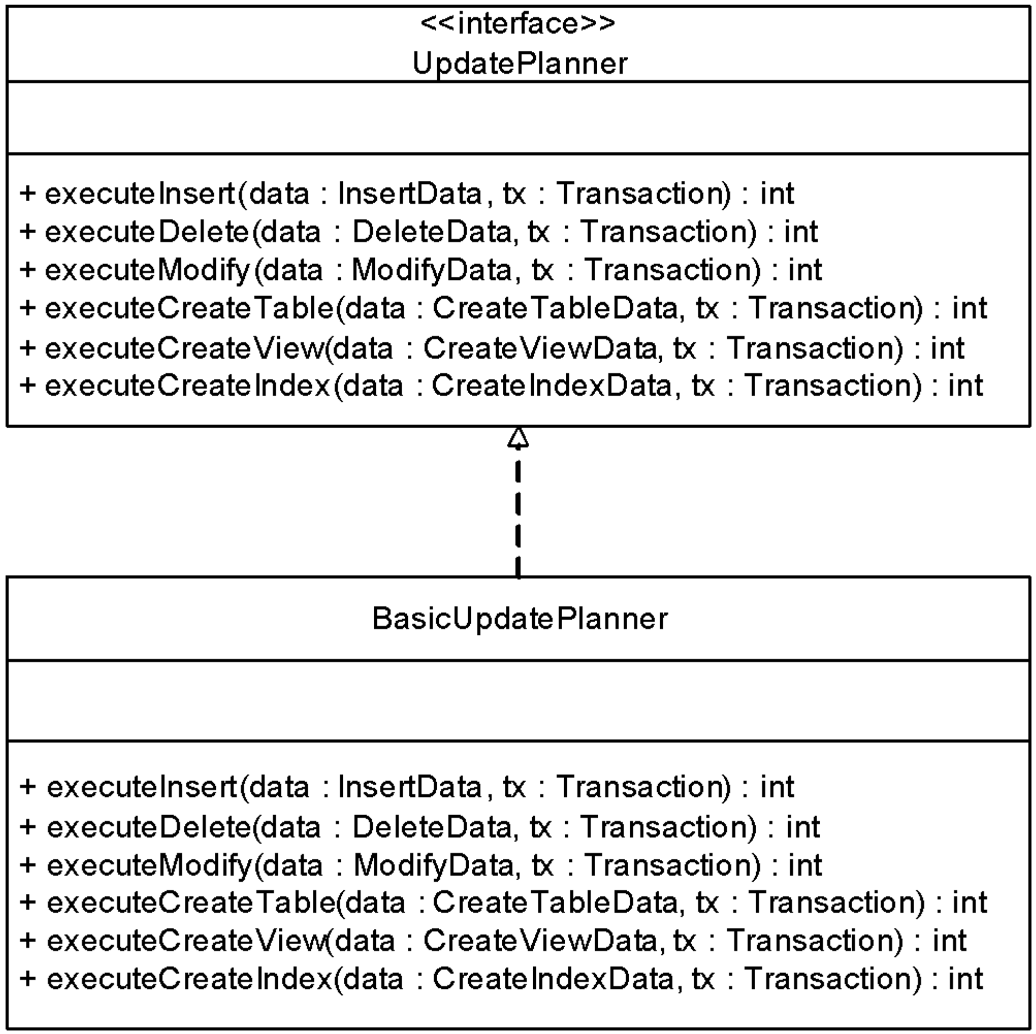
1. Table plans (FROM)
2. Product plan (FROM)
3. Select plan (WHERE)
4. ***Group-by plan (GROUP BY)***
5. Project (SELECT)
6. ***Having plan (HAVING)***
7. ***Sort plan (ORDER BY)***

- Fields mentioned in HAVING and ORDER BY clauses must appear in the project list

Update Planning

- DDLs and update commands are usually simpler than SELECTs
  - Single table
  - WHERE only, no GROUP BY, HAVING, SORT BY, etc.
- Deterministic planning algorithm is often sufficient
- `BasicUpdatePlanner` implements deterministic planning algorithm for updates

BasicUpdatePlanner



executeModify

- The modification statement are processed by the method `executeModify`

```
public int executeModify(ModifyData data, Transaction tx) {
    Plan p = new TablePlan(data.tableName(), tx);
    p = new SelectPlan(p, data.pred());
    UpdateScan us = (UpdateScan) p.open();
    us.beforeFirst();
    int count = 0;
    while (us.next()) {
        Collection<String> targetflds = data.targetFields();
        for (String fld : targetflds)
            us.setVal(fld, data.newValue(fld).evaluate(us));
        count++;
    }
    us.close();
    VanillaDb.statMgr().countRecordUpdates(data.tableName(), count);
    return count;
}
```

executeInsert

- The insertion statement are processed by the method `executeInsert`

```
public int executeInsert(InsertData data, Transaction tx) {
    Plan p = new TablePlan(data.tableName(), tx);
    UpdateScan us = (UpdateScan) p.open();
    us.insert();
    Iterator<Constant> iter = data.vals().iterator();
    for (String fldname : data.fields())
        us.setVal(fldname, iter.next());

    us.close();
    VanillaDb.statMgr().countRecordUpdates(data.tableName(), 1);
    return 1;
}
```

Methods for DDL Statements

```
public int executeCreateTable(CreateTableData data, Transaction tx) {
     VanillaDb.catalogMgr().createTable(data.tableName(), data.newSchema(), tx);
return 0;
}

public int executeCreateView(CreateViewData data, Transaction tx) {
     VanillaDb.catalogMgr().createView(data.viewName(), data.viewDef(), tx);
     return 0;
}

public int executeCreateIndex(CreateIndexData data, Transaction tx) {
     VanillaDb.catalogMgr().createIndex(data.indexName(), data.tableName(),
                    data.fieldName(), data.indexType(), tx);
     return 0;
}
```

References

- Ramakrishnan Gehrke., chapters 4, 12, 14 and 15, *Database management System*, 3ed
- Edward Sciore., chapters 17, 18 and 19, *Database Design and Implementation*
- Hellerstein, J. M., Stonebraker, M., and Hamilton, J., Architecture of a database system, *Foundations and Trends in Databases*, 1, 2, 2007