Query Processing & Optimization

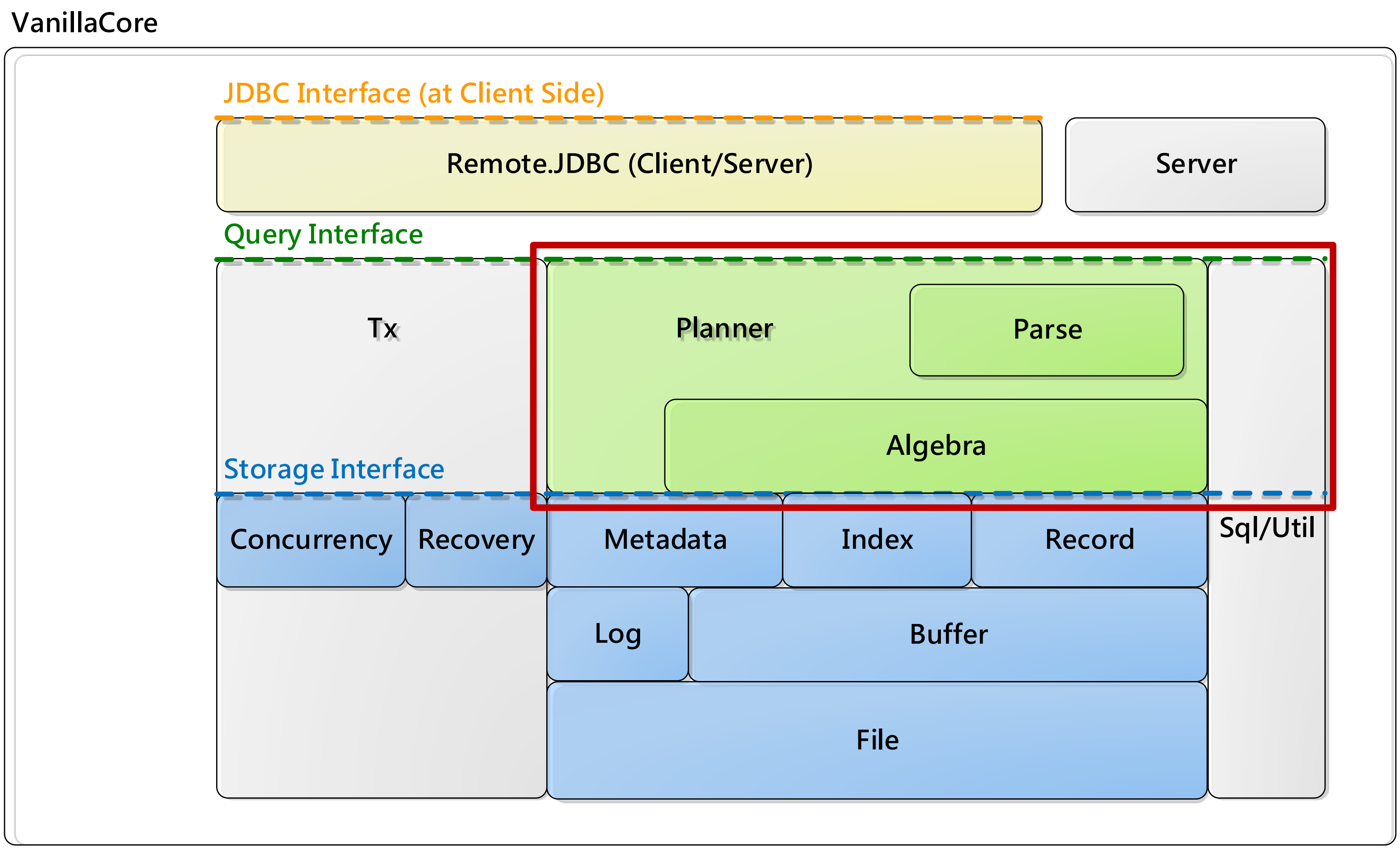
DB/AI Bootcamp

2018 Summer

DataLab, CS, NTHU

Query Processing

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

What does a DB do when SQL coming?

1. Parses the SQL command
2. Verifies the SQL command
3. Finds a good plan for the SQL command
4. Executes the plan

What does a DB do when SQL coming?

1. Parses the SQL command
2. Verifies the SQL command
3. Finds a good plan for the SQL command
4. Executes the plan

Outline

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

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

$\boxed{\texttt{SELECT}}\ \boxed{\texttt{a}}\ \boxed{\texttt{FROM}}\ \boxed{\texttt{t1}}\boxed{\texttt{,}}\ \boxed{\texttt{t2}}\ \boxed{\texttt{WHERE}}\ \boxed{\texttt{b}}\ \boxed{\texttt{=}}\ \boxed{\texttt{3}}$

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

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
- E.g.,

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

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,

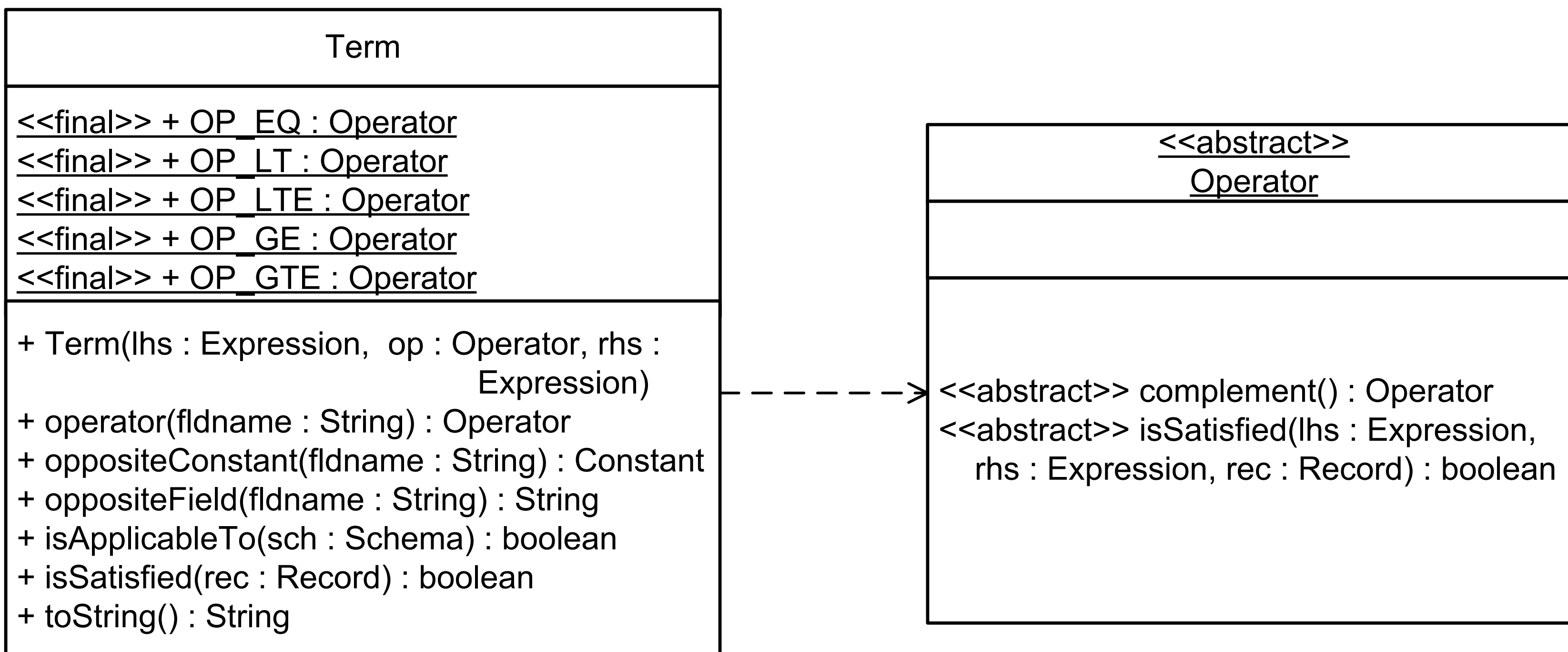
Predicate

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

Expression Term

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	url	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);
}
```

Predicate

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

Predicate

+ Predicate()
+ Predicate(t : Term)

// used by the parser
+ conjunctWith(t : Term)

// used by a scan
+ isSatisfied(rec : Record) : boolean

// used by the query planner
+ selectPredicate(sch : Schema) : Predicate
+ joinPredicate(sch1 : Schema, sch2 : Schema) : Predicate
+ constantRange(fldname : String) : ConstantRange
+ joinFields(fldname : String) : Set<String>
+ toString() : String

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

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 DB do when SQL coming?

1. Parses the SQL command
2. Verifies the SQL command
3. Finds a good ***plan*** for the SQL command
4. Executes the plan

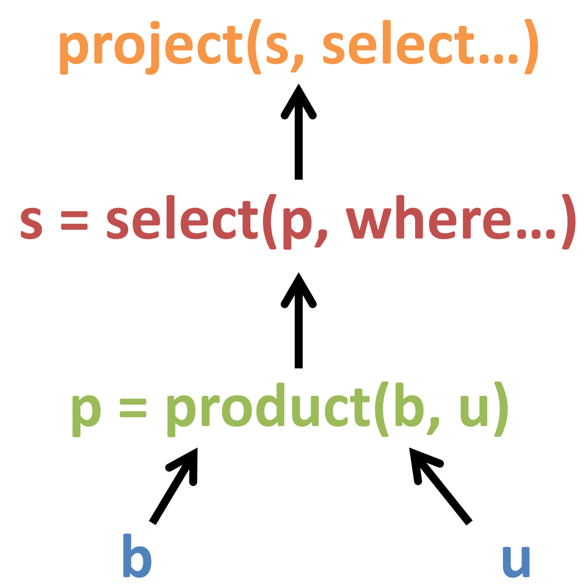
Outline

- 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)

- A SQL command will be expressed as at-least one tree in relational algebra

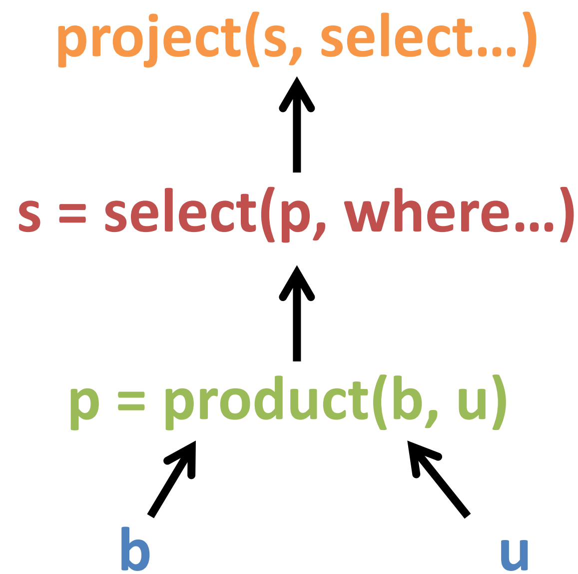
```
SELECT b.blog_id
    FROM blog_pages b, users u
    WHERE b.author_id=u.user_id
      AND u.name='Steven Sinofsky'
      AND b.created >= 2011/1/1;
```



Why this translation?

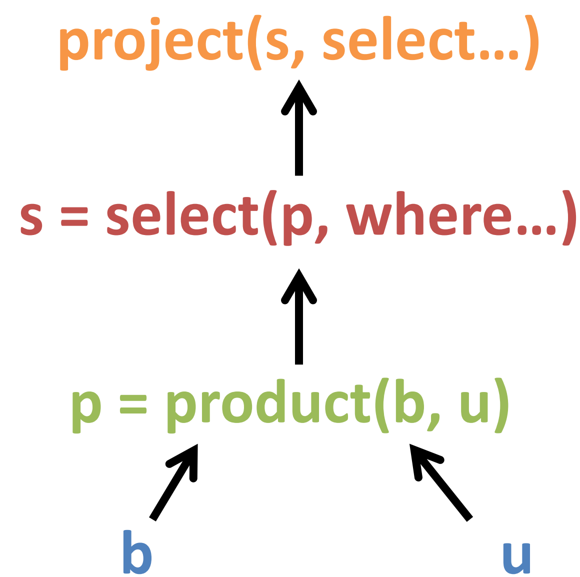
SQL and Relational Algebra (2/2)

- 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



Algebra Tree

- A SQL command will 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;
```

project(s, select…)

s = select(p, where…)

p = product(b, u)

b

u

Scans

- A ***scan*** represents an operator in a relational algebra tree

ProjectScan(s, select…)

↑

s = SelectScan(p, where…)

↑

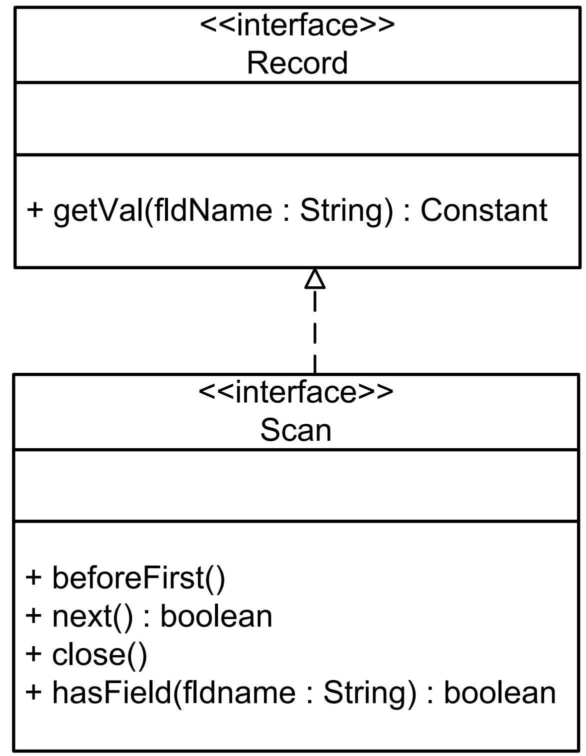
p = ProductScan(b, u)

↗ ↖

TableScan of b   TableScan of u

The `Scan` Interface

- An iterator of output records of a partial query



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();
```

ProjectScan (s, select sname)

s = SelectScan (b, where sid = 5)

TableScan of b

TableScan

- Basically, tasks are delegated to a `RecordFile`

```
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);
	}
	...
}
```

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);
    }
    ...
}
```

ProjectScan

```java
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.");
    }
    ...
}
```

ProductScan

```
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***

Example

project(s, select blog_id)

beforeFirst()

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

beforeFirst()

product(b, u)

beforeFirst()

next()

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()

product(b, u)

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

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)

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

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()

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()

next()

product(b, u)

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)

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

getVal()

b

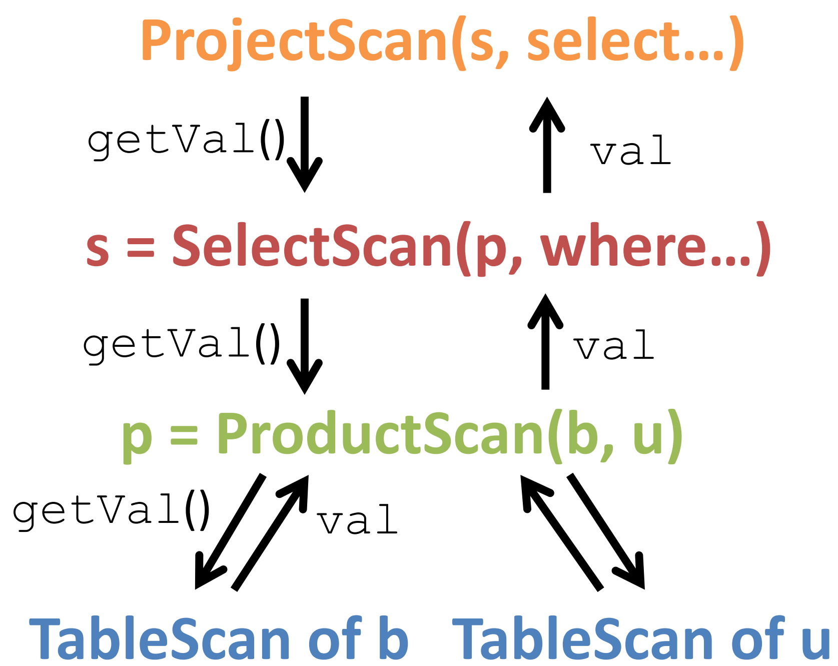
blog_id	url	created	author_id
33981	...	2009/10/31	729
33982	...	2012/11/15	730
41770	...	2012/10/20	729

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



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?

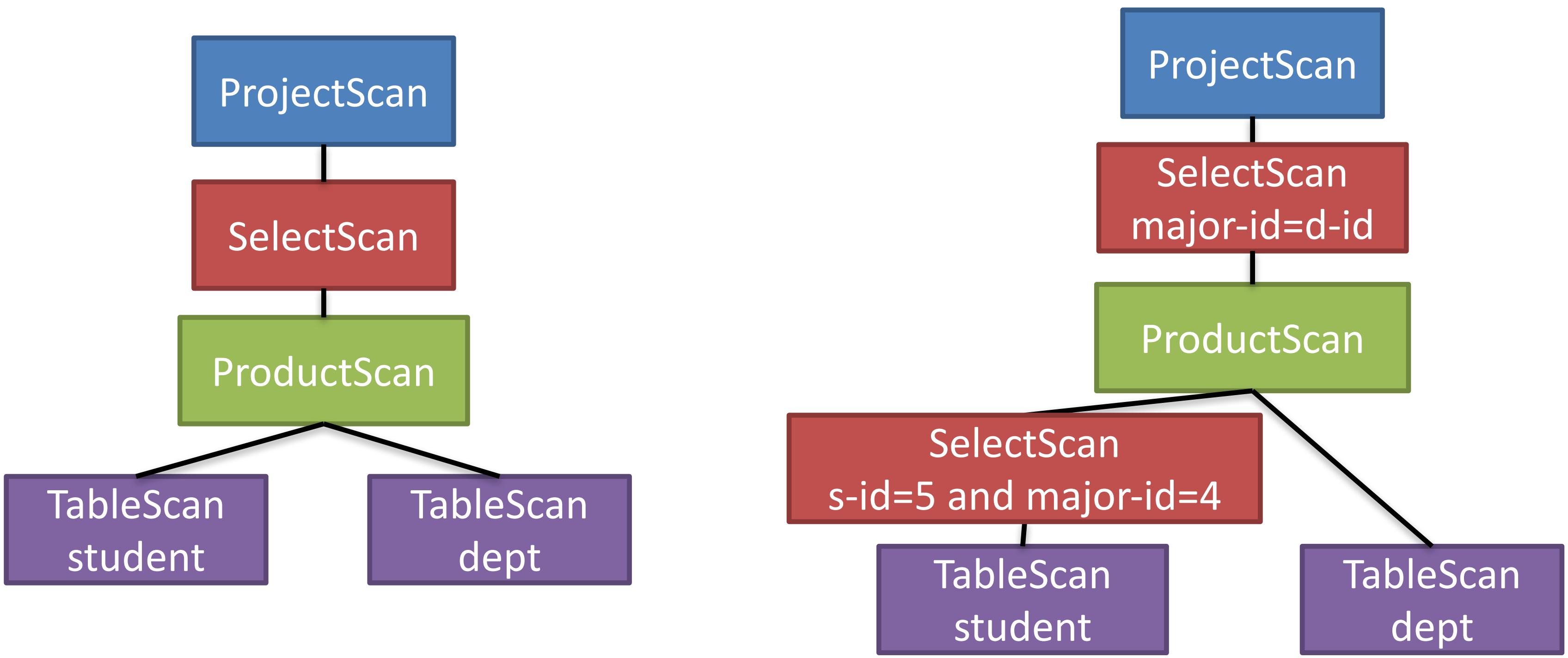
- Given the scans:

- 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?

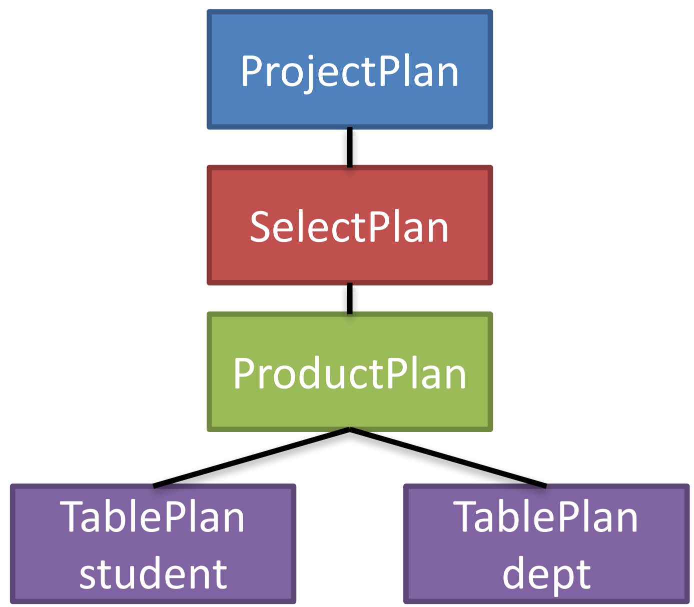
- A good scan tree can be faster than a bad one for orders of magnitude
- Consider the product scan at middle
  - Let *R*(student)=10000, *B*(student)=1000, *B*(dept)= 500, and *selectivity*(s-id=5&major-id=4)=0.01
  - Each block access requires 10ms
- Left: (1000+10000*500)*10ms = 13.9 hours
- Right: (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

The `Plan` Interface

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

<<interface>>
Plan

+ open() : Scan
+ blocksAccessed() : int
+ schema() : Schema
+ histogram() : Histogram
+ recordsOutput() : int



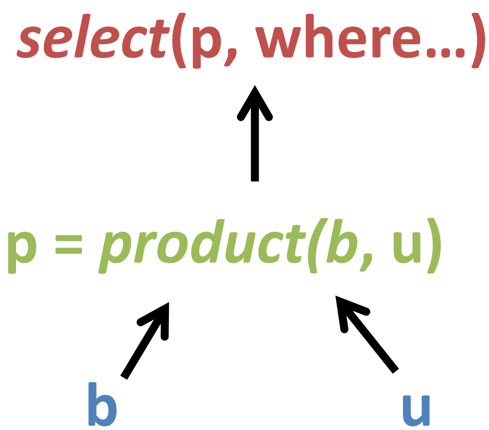
Using a Query Plan

```
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

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 DB do when SQL coming?

1. Parses the SQL command
2. Verifies the SQL command
3. ***Finds a good plan*** for the SQL command
4. Executes the plan

Planning

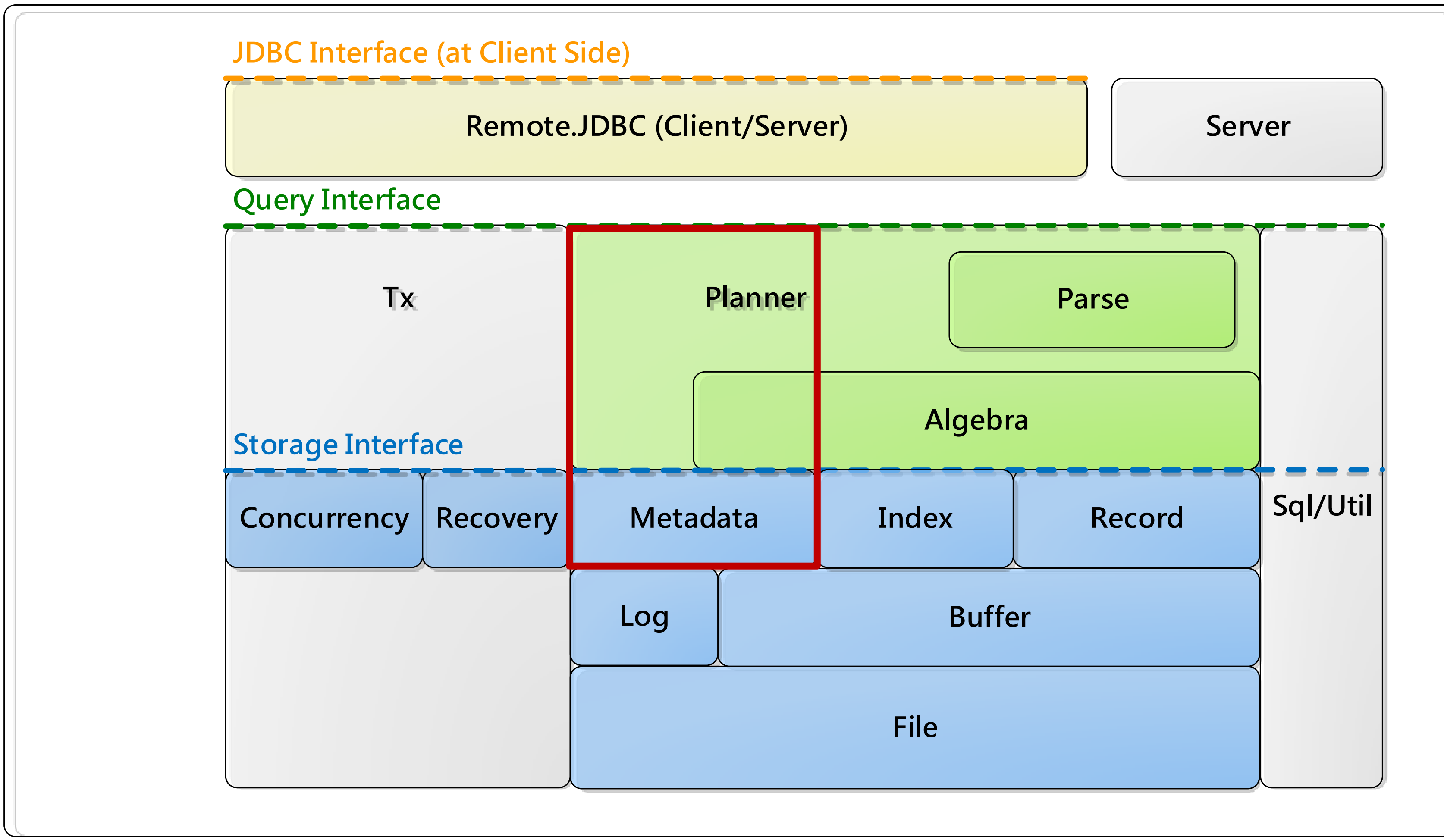
- Input:
  - SQL data
- Output:
  - A good plan tree

- Done by the ***planner***
- How?

Query Optimization

Where Are We?

VanillaCore



Outline

- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

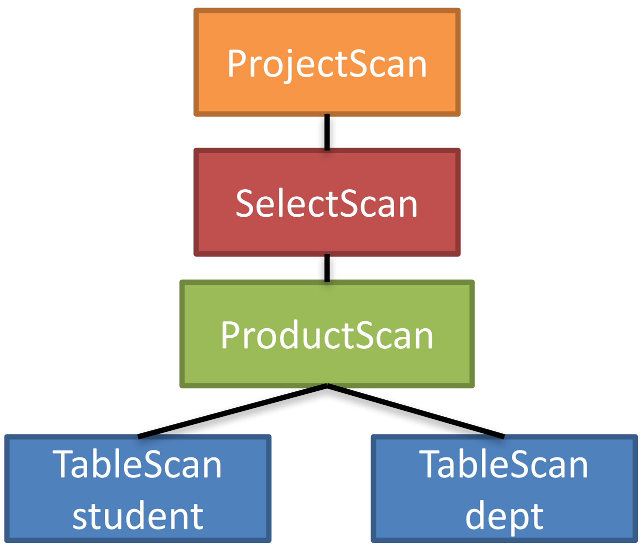
Outline

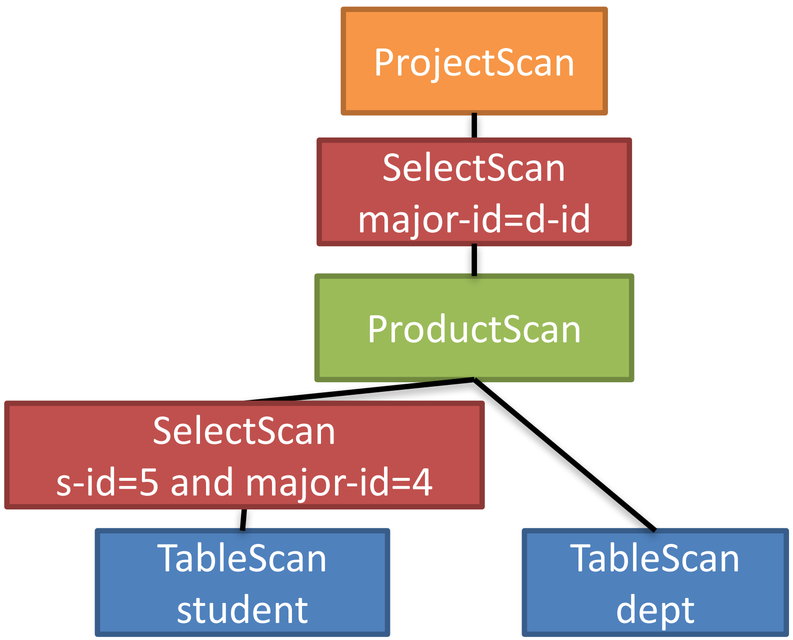
- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

SQL and Relational Algebra

- A SQL command can be expressed as multiple trees in relational algebra

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





Query Optimization

- A good scan tree can be faster than a bad one for orders of magnitude
- Query optimizer (planner):
  1. Generate candidate plan trees
  2. Estimate cost of each corresponding scan tree (not opened yet)
  3. Pick and open the "best" one to execute query

Query Optimization

- A good scan tree can be faster than a bad one for orders of magnitude
- Query optimizer (planner):
  1. Generate candidate plan trees
  2. Estimate cost of each corresponding scan tree (not opened yet)
  3. Pick and open the “best” one to execute query

Outline

- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Push Select Down
  - Join Order Problem
  - Heuristic Planner in VanillaCore
- Selinger-Style Query Optimizer

Metric for Cost

- Cost of query is generally measured as total elapsed time for answering query
  – Typically, I/O delay is the most important factor

Cost Estimation

- For each plan p, we estimate **B(p)**
  - #blocks accessed by the corresponding scan

- Usually, estimating B(p) requires more knowledge:
  - **R(p)**: #records output
  - **V(p,f)**: #distinct values for field f
  - Search cost (#blocks) of index, if used

Estimating B(p)

p	B(p)
TablePlan	Actual #blocks cached by StatMgr (via periodic table scanning)
ProjectPlan(c)	B(c)
SelectPlan(c)	B(c)
IndexSelectPlan(t)	IndexSearchCost(R(t), R(p)) + R(p)
ProductPlan(c1, c2)	B(c1) + (R(c1) * B(c2))
IndexJoinPlan(c1, t2)	B(c1) + (R(c1) * IndexSearchCost(R(t2), 1)) + R(p)

Estimating R(p) and Index Search Cost

- Index search cost:
  - `HashIndex.searchCost()`
  - `BTreeIndex.searchCost()`

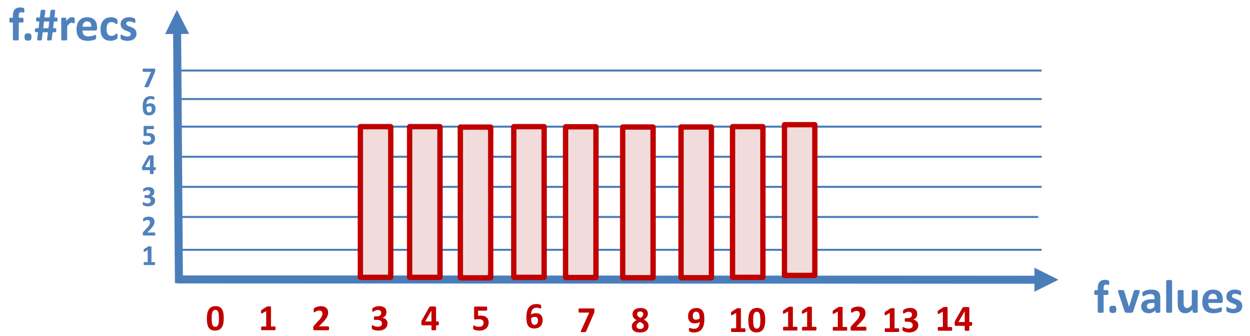
- Estimating R(p) is called ***cardinality estimation***

Outline

- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

Naïve Approach

- Uniform assumption
  - All values in field appear with the same probability



- Few statistics are enough:

R(c)	**#records** in child plan c
V(c, f)	**#distinct values** in field f in c
Max(c, f)	**Max value** in field f in c
Min(c, f)	**Min value** in field f in c

p = Select(c, f=x)

- R(p)?

R(c)	**#records** in child plan c
V(c, f)	**#distinct values** in field f in c
Max(c, f)	**Max value** in field f in c
Min(c, f)	**Min value** in field f in c

- Selectivity(f=x): $\frac{1}{V(c,f)}$
- R(p): Selectivity(f=x) * R(c)

p = Select(c, f>x)

- R(p)?

R(c)	**#records** in child plan c
V(c, f)	**#distinct values** in field f in c
Max(c, f)	**Max value** in field f in c
Min(c, f)	**Min value** in field f in c

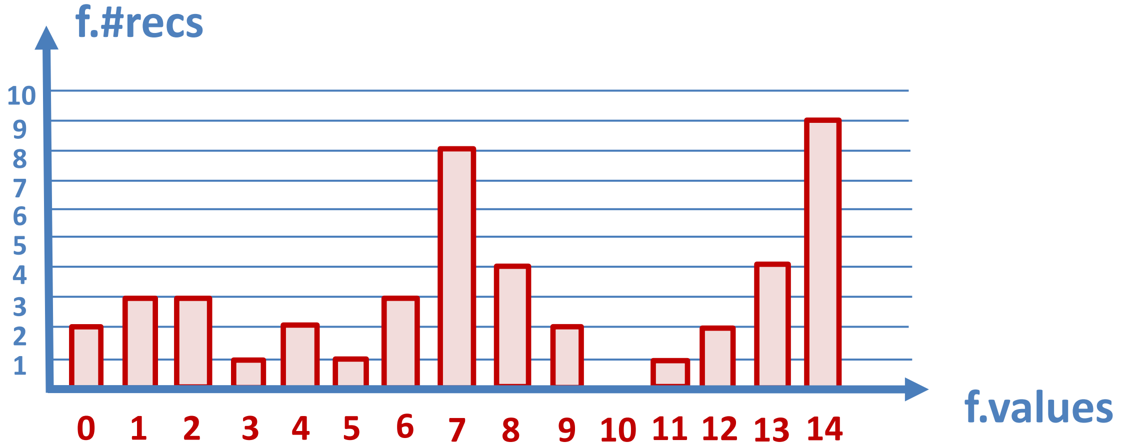
- Selectivity(f>x): $\frac{Max(c,f)-x}{Max(c,f)-Min(c,f)}$
- R(p): Selectivity(f>x) * R(c)

Outline

- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

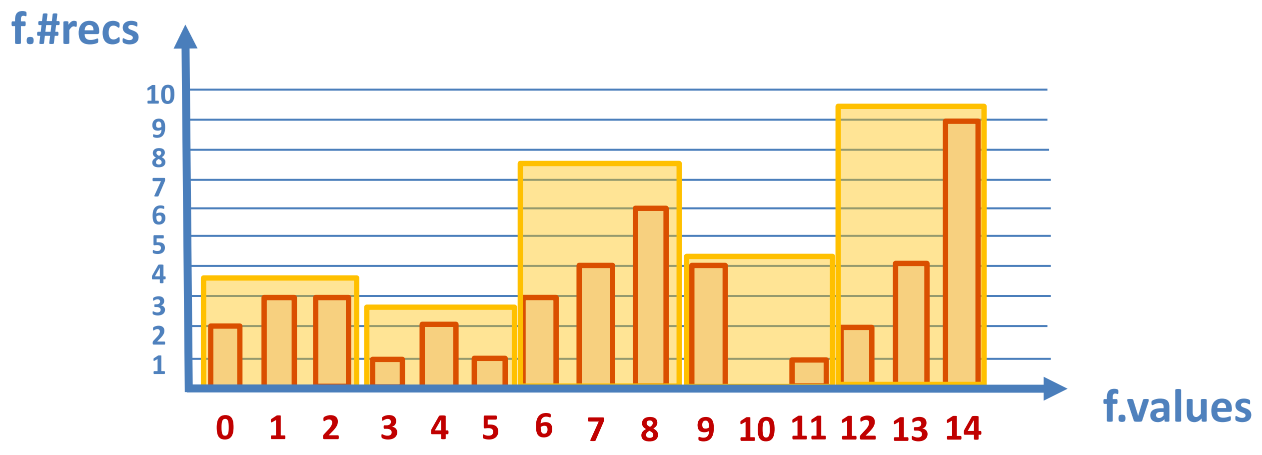
Naïve Estimation is Inaccurate

- In the real world, values in a field are seldom uniform distributed
- p = Select(c, f=14)
- Estimated R(p) = $\frac{1}{15}$ * R(c) = 3
- Actually, R(p) = 9



Histogram

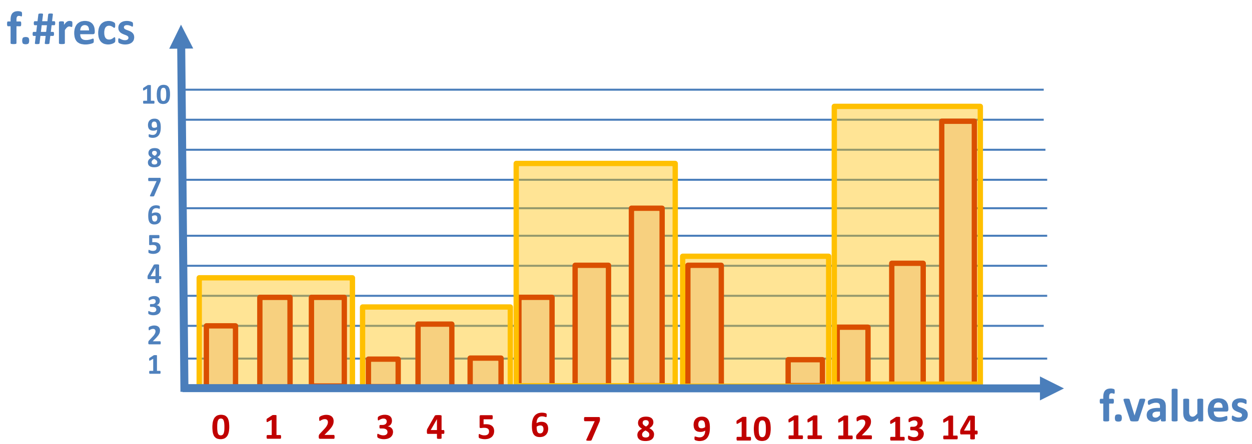
- Approximates value distribution in every field
- Partitions field values into a set of ***buckets***



- More #buckets, more accurate approximation
  - Tradeoff between accurate and storage cost

Buckets

- Each bucket collects statistics of a value range
  - Assuming ***uniform distribution*** within the bucket



- R(p, f, b): #records
- V(p, f, b): #distinct values
- Range(p, f, b): value range

Cardinality Estimation

- Not matter what p is, we have

$$R(p) = \sum_{b \in p.hist.buckets(f)} R(p, f, b)$$

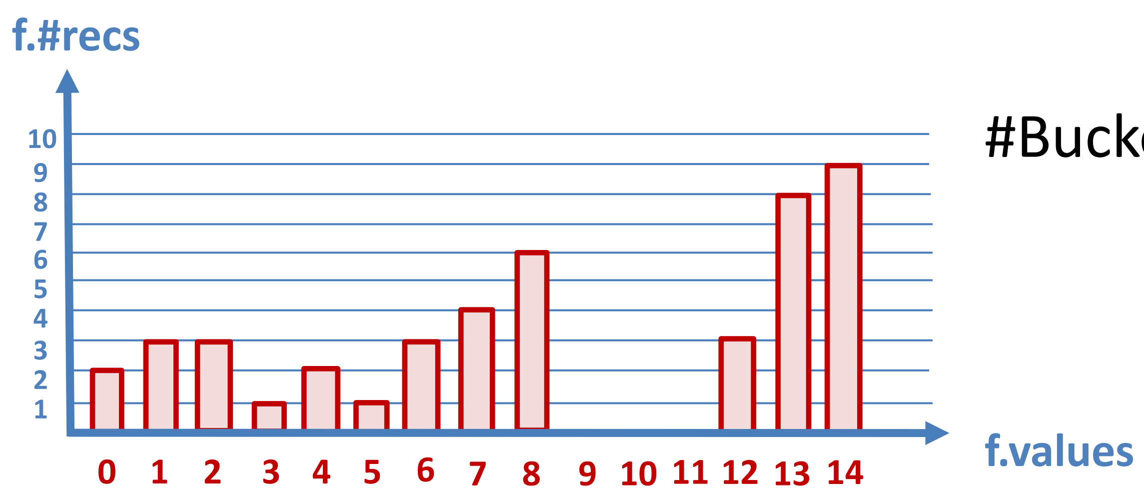
for any f

Outline

- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

“Raw” Histogram of a Table

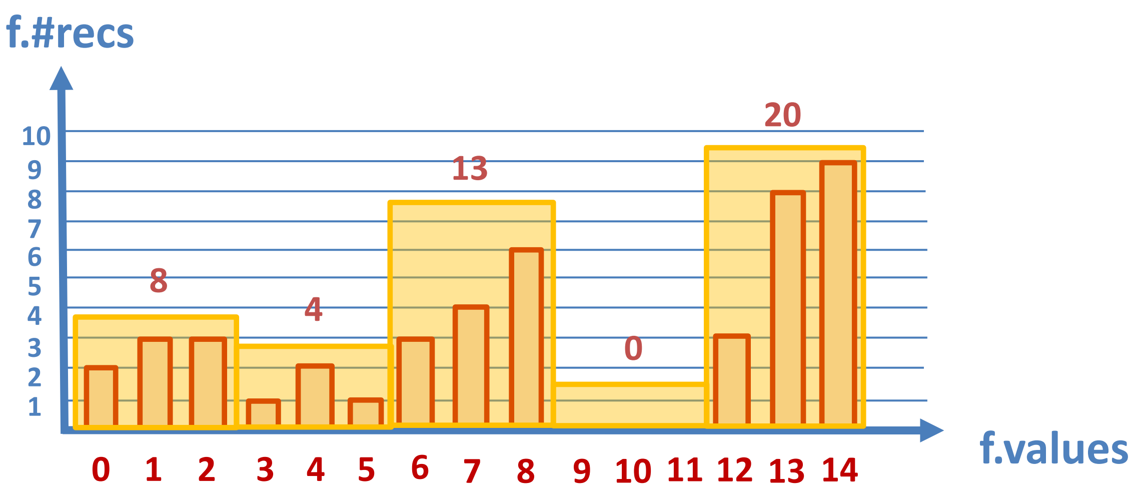
- Data structure that approximates value distribution
- Partitions field values into a set of ***buckets***
- Each bucket collects statistics of a value range
  - Assuming ***uniform distribution*** of records within the bucket
- Given raw values and #buckets, how to decide bucket ranges?



#Buckets = 5

Equi-Width Histogram

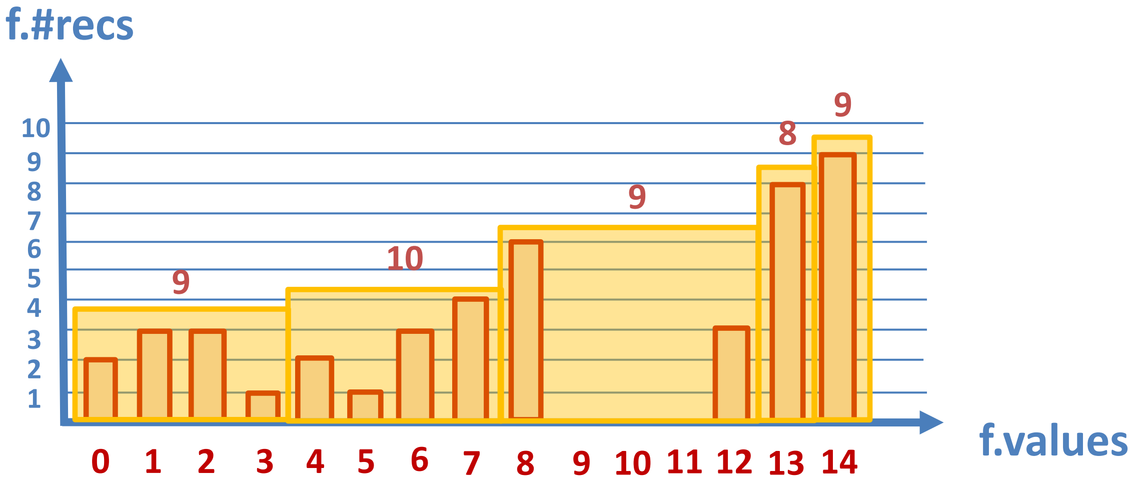
- Partition strategy: all buckets have the same range
- |Range(b)| = $\frac{Max(p,f) - Min(p,f) + 1}{\#Buckets}$



- Problem: some buckets may be wasted

Equi-Depth Histogram

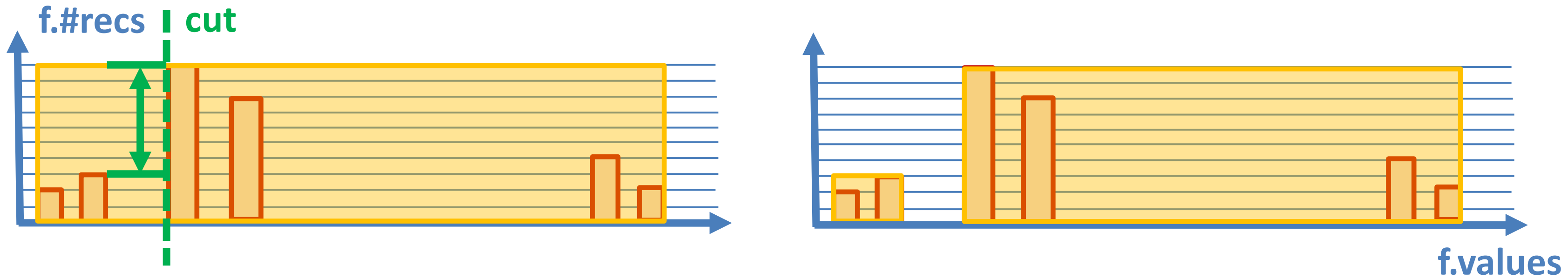
- Partition strategy: all buckets have the same #recs
- Depth = $\dfrac{R(p)}{\#Buckets}$



- Problem: records/values in a bucket may ***not*** be uniformly distributed

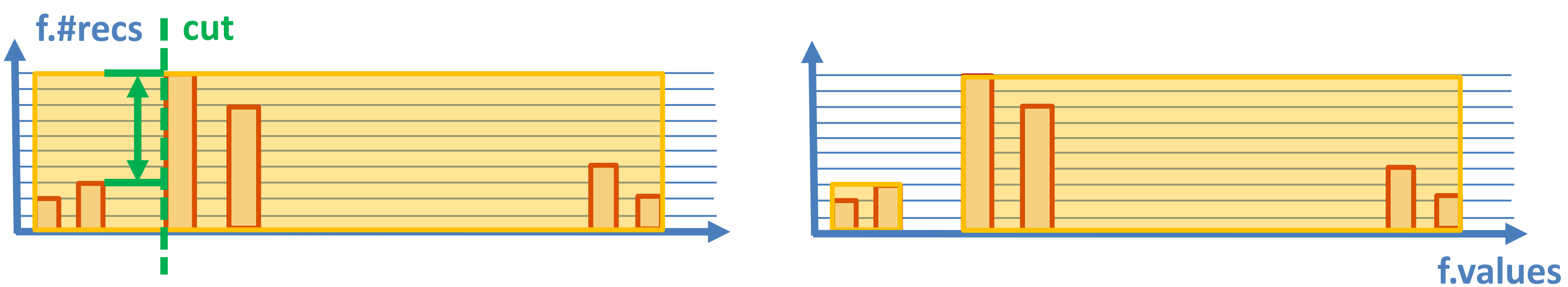
Max-Diff Histogram

- Partition strategy: split buckets at values with max. diff in #rec (MaxDiff(F)) or area (MaxDiff(A)):
  1. #recs: uniform #records in each bucket

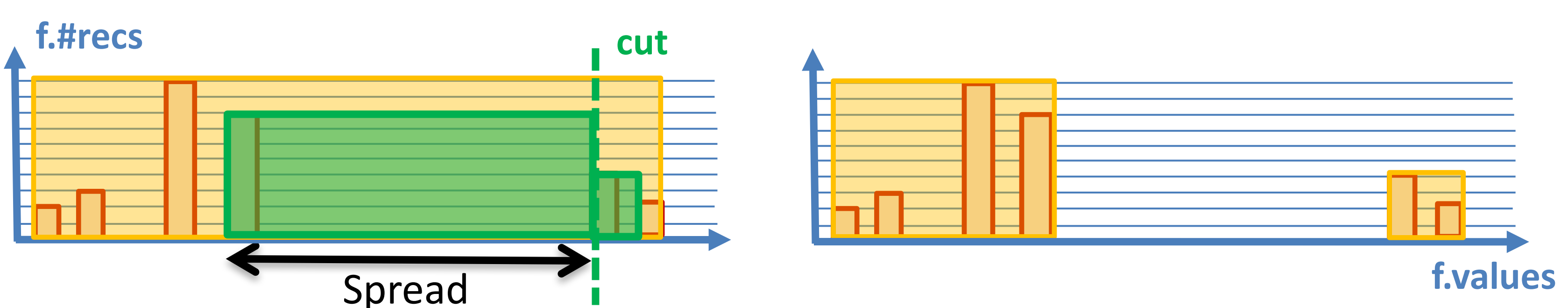


Max-Diff Histogram

- Partition strategy: split buckets at values with max. diff in #rec (MaxDiff(F)) or area (MaxDiff(A)):
  1. #recs: uniform #records in each bucket

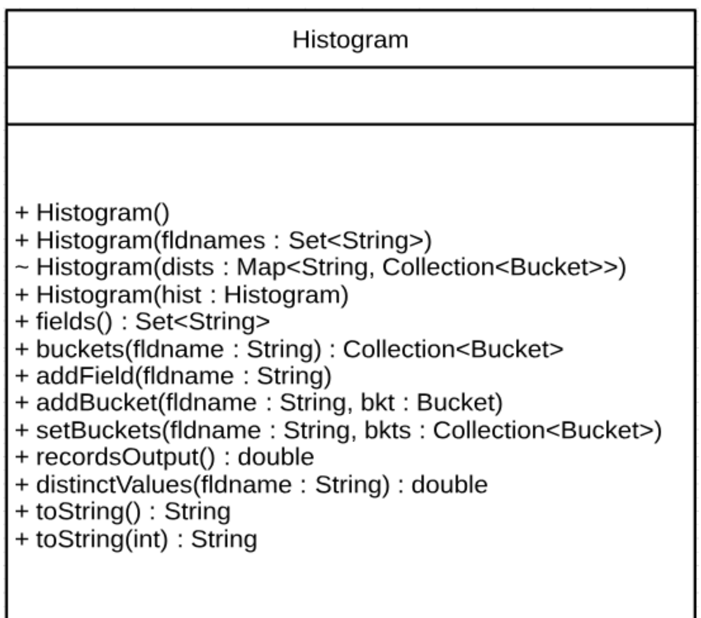


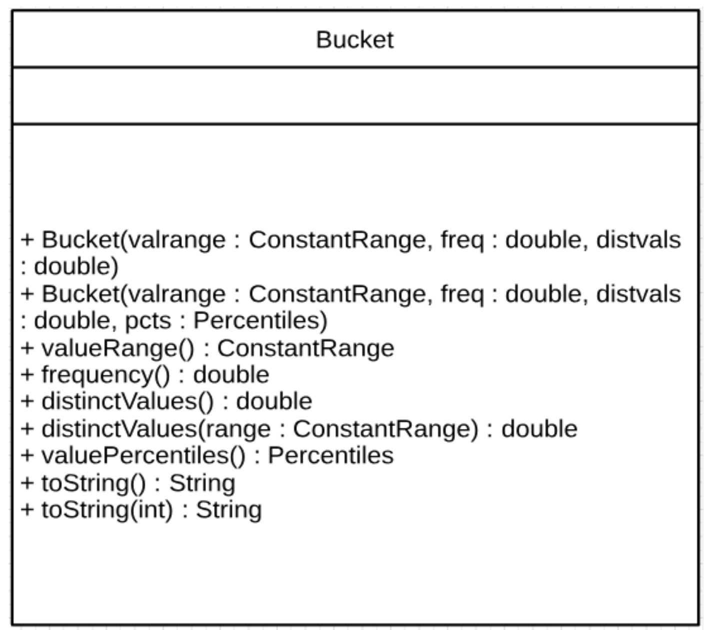
  2. Area: uniform #records ***and values*** in each bucket



Histogram in VanillaCore

- “Raw” histograms are statistics metadata
  - org.vanilladb.core.storage.metadata.statistics
- Accessed (by TablePlan) via StatMgr.getTableStatInfo()





Building Histogram (1/2)

- When system starts up:
- StatMgr:
  - Scans table and calls SampledHistogramBuilder.sample()
  - When done, calls SampledHistogramBuilder.newMaxDiffHistogram()
- Histogram types:
  - MaxDiff(A) : when field value is numeric
  - MaxDiff(F) : otherwise

Building Histogram (2/2)

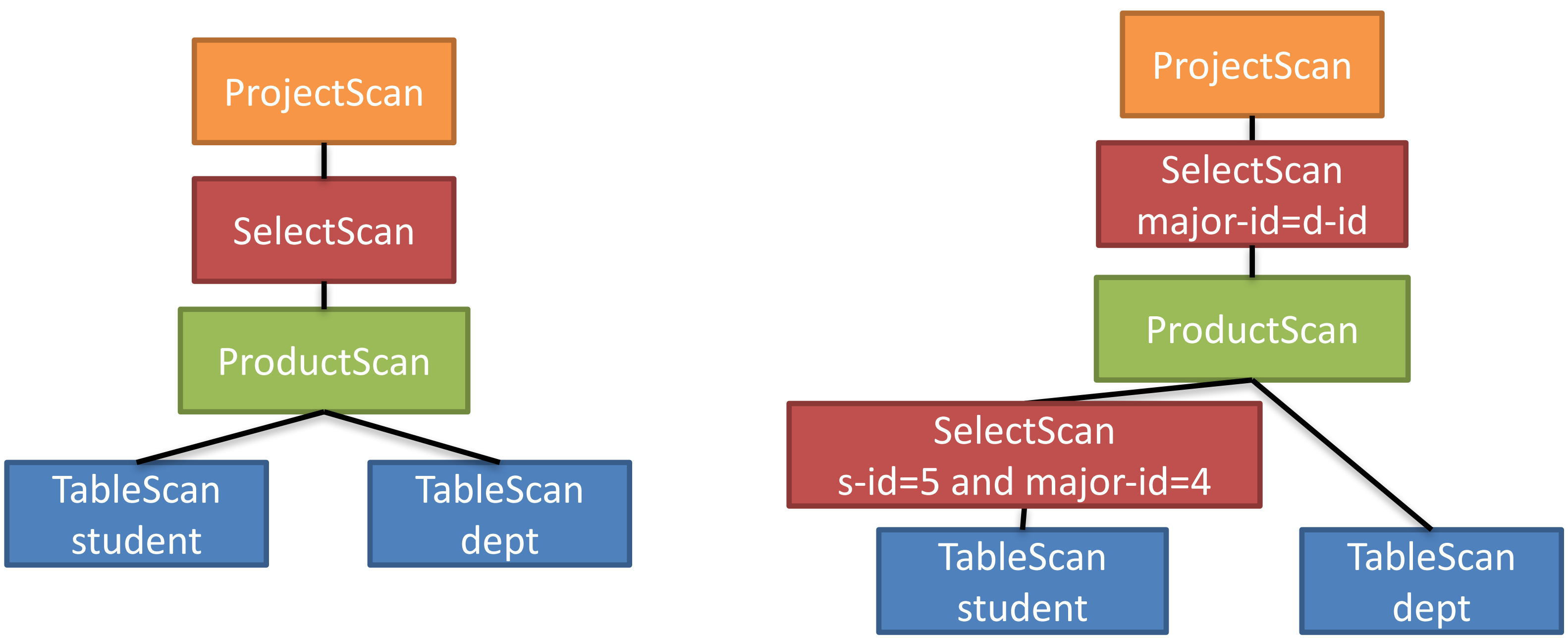
- At runtime:
- StatMgr tacks #recs updated for each table
  - QueryPlanner calls StatMgr.countRecordUpdates() after executing modify/insert/delete queries
- Rebuilds histogram ***in background*** when StatMgr.getTableStatInfo() is called
  - If #recs updated > threshold (e.g., 100)
- StatisticsRefreshTask:
  - Scans table and calls SampledHistogramBuilder.sample()
  - When done, calls SampledHistogramBuilder.newMaxDiffHistogram()

Outline

- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

Query Optimization

- Query optimizer:
  1. Generate candidate plan trees
  2. Estimate cost of each corresponding scan tree
  3. Pick and open the "best" one to execute query

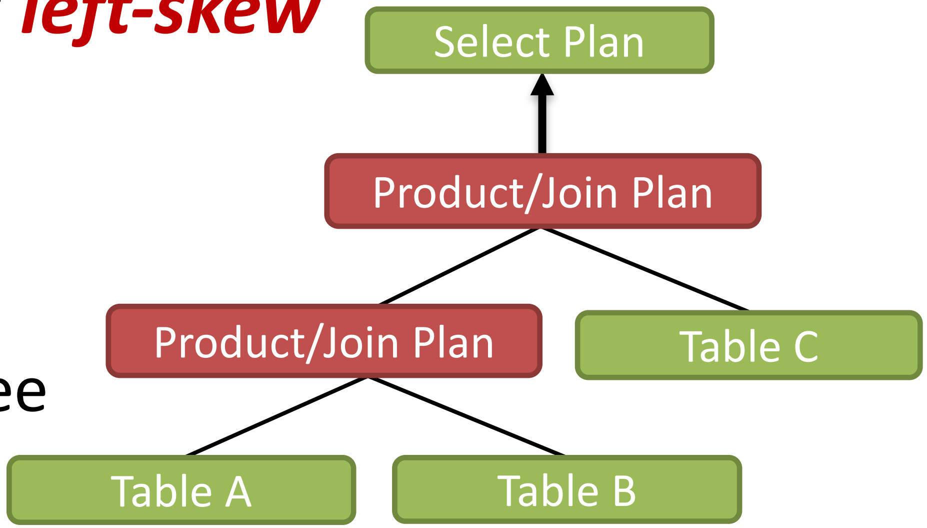


In Reality…

- Generating all candidate plan trees are too costly
  - #trees with n products/joins = Catalan number:

$$\frac{1}{n+1}\binom{2n}{n}$$

- Compromise: consider ***left-skew*** candidate trees only
- Query planner's goal
  - Avoiding bad trees
  - Not finding the best tree



Why Left-Skew Trees Only?

- Tend to be better than plans of other shapes
- Because many join algorithms scan right child c2 multiple times
- Normally, we don't want c2 to be a complex subtree

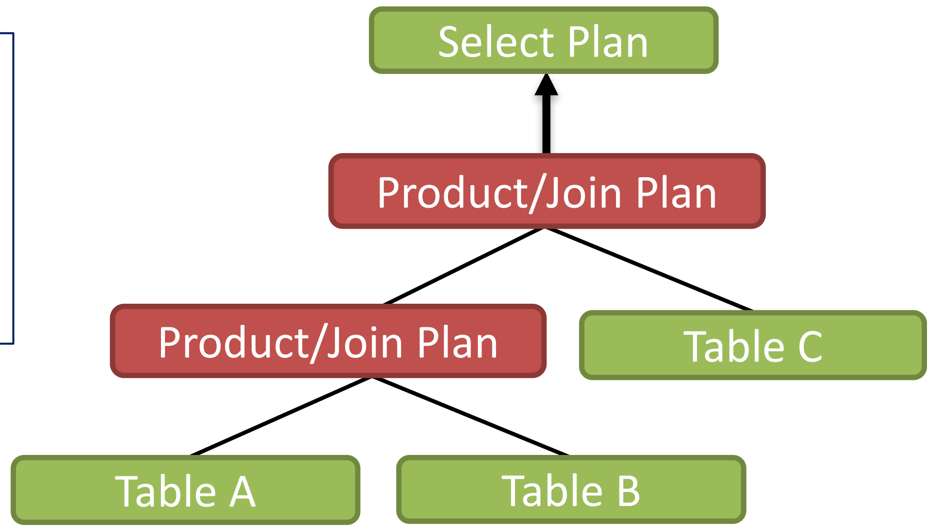
BasicQueryPlanner

```java
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.newPlanner().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: Add a group-by plan if specified
		if (data.groupFields() != null) {
			p = new GroupByPlan(p, data.groupFields(), data.aggregationFn(), tx);
	}
	// Step 5: Project onto the specified fields
	p = new ProjectPlan(p, data.projectFields());
	// Step 6: Add a sort plan if specified
	if (data.sortFields() != null)
		p = new SortPlan(p, data.sortFields(), data.sortDirections(), tx);
	// Step 7: Add a explain plan if the query is explain statement
	if (data.isExplain())
		p = new ExplainPlan(p);
	return p;
}
```

- Product/join order follows what's written in SQL

Cost & Bottlenecks

```
SELECT   A.c1, B.c2, C.c3
FROM     A, B, C
WHERE    A.aid = C.aid
AND      B.bid = C.bid
AND      A.c2 = xxx
```

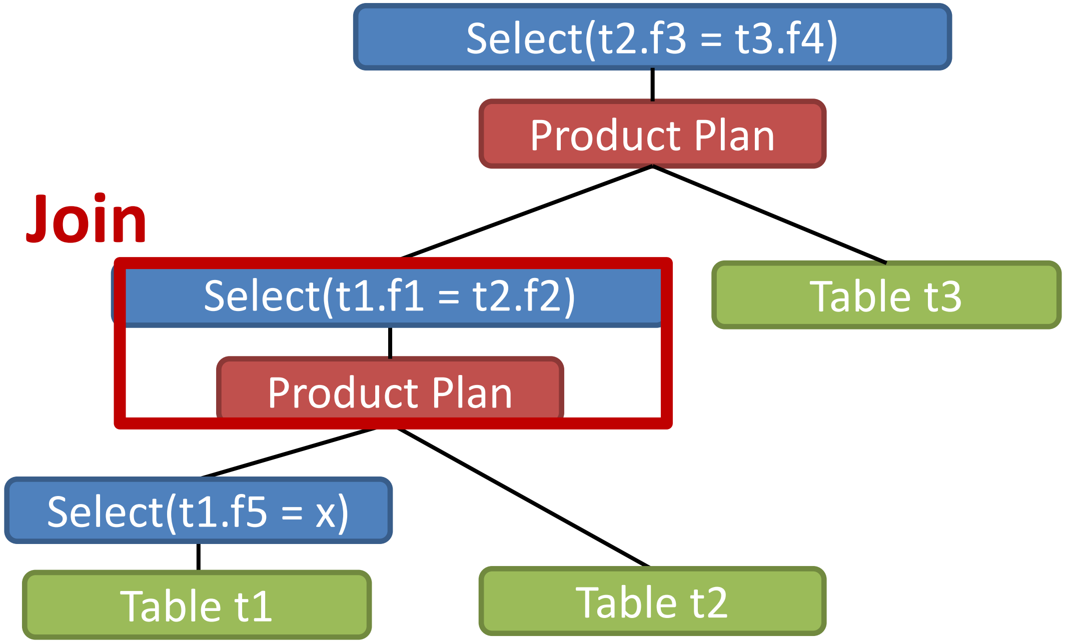


- B(root) dominated by ***#recs*** of product/join ops
  - B(Product(c1, c2)) = B(c1) + (**R(c1)** * B(c2))
  - B(IndexJoin(c1, c2)) = B(c1) + (**R(c1)** * SearchCost(...)) + ...
  - B(Select(c)) = B(c)

Pushing Select Ops Down

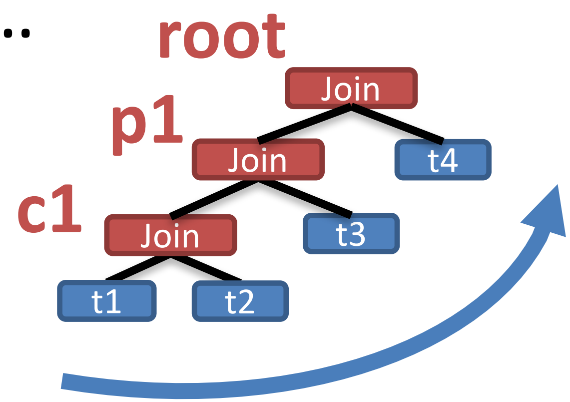
- Execute Select ops as early as possible
- ↓R(c1) and ↓R(c2) of each product/join op

```
SELECT   *
FROM     t1, t2, t3
WHERE    t1.f1 = t2.f2
AND      t2.f3 = t3.f4
AND      t1.f5 = x
```



Greedy Join Ordering

- B(root) = **B(p1)** + (**R(p1)** * …) + …
  - ↓ B(root) implies ↓(p1)
- **B(p1)** = **B(c1)** + (**R(c1)** * …) + …
  - ↓ B(root) also implies ↓(c1)
- **…**
- B(root) $\propto$ R(p1) + R(c1) + …



- ***Greedy Join ordering***: repeatedly add table to the “trunk” that result in lowest R(trunk)

HeuristicPlanner in VanillaCore

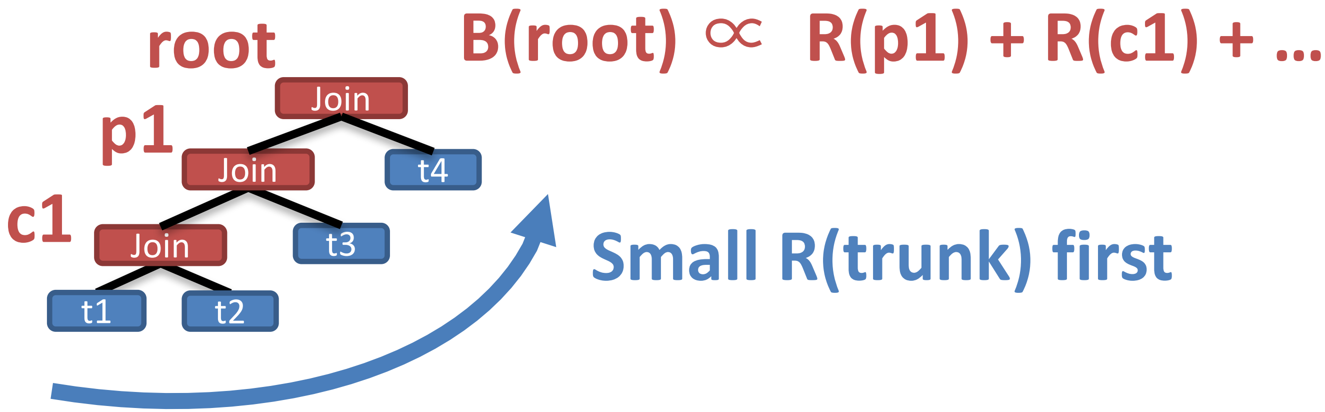
```
public Plan createPlan(QueryData data, Transaction tx) {
      // Step 1: Create a TablePlanner object for each mentioned table/view
      int id = 0;
      for (String tbl : data.tables()) {
            String viewdef = VanillaDb.catalogMgr().getViewDef(tbl, tx);
            if (viewdef != null)
                  views.add(VanillaDb.newPlanner().createQueryPlan(viewdef, tx));
            else {
                  TablePlanner tp = new TablePlanner(tbl, data.pred(), tx, id);
                  tablePlanners.add(tp);
            }
            id += 1;
      }
      // Step 2: Choose the lowest-size plan to begin the trunk of join
      Plan trunk = getLowestSelectPlan();
      // Step 3: Repeatedly add a plan to the join trunk
      while (!tablePlanners.isEmpty() || !views.isEmpty()) {
            Plan p = getLowestJoinPlan(trunk);
            if (p != null)
                  trunk = p;
            else
                  // no applicable join
                  trunk = getLowestProductPlan(trunk);
      }
      // Step 4: Add a group by plan if specified
      // Step 5. Project on the field names
      // Step 6: Add a sort plan if specified
      // Step 7: Add a explain plan if the query is explain statement
}
```

Feasible Select ops applied

Outline

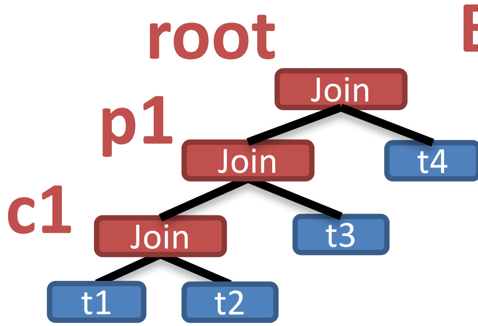
- Overview
- Cost Estimation
  - Cardinality Estimation
  - Histogram-based Estimation
  - Types of Histograms
- Heuristic Query Optimizer
  - Basic Planner
  - Pushing Select Down
  - Join Ordering
  - Heuristic Query Planner in VanillaCore
- Selinger-Style Query Optimizer

Why not HeuristicPlanner?



- Assumption: ↓R(c1) implies ↓R(p1))
- May ***not*** be true: match rate matters
- Exhaustively searching the best join order?
  - Cost: **O(n!)** for n joins (e.g., 8! = 40320)

Selinger-Style Optimizer



B(root) $\propto$ R(p1) + R(c1) + …

C(root) = R(root) + R(p1) + R(c1) + …

- Consider the best trees after 1, 2, 3, … joins
- Observation: if C(**t3 $\bowtie$ t1 $\bowtie$ t2**) <= C(**t2 $\bowtie$ t3 $\bowtie$ t1**), then C(**t3 $\bowtie$ t1 $\bowtie$ t2** $\bowtie$ t4) <= C(**t2 $\bowtie$ t3 $\bowtie$ t1** $\bowtie$ t4)
- We can use ***dynamic programming*** to avoid repeating computations

Selinger Optimizer Example (1/3)

- Here are 3 relations to join: A, B, C
- Step 1:
  - compute the cost (R) of each relation's cheapest plan

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =
    - Compare {A}B   Cost: {A} + {A, B}
    - Compare {B}A   Cost: {B} + {B, A}

Same # of Record

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =

    Choose the one with least cost
    - Compare {A}B Cost: {A} + {A, B}
    - Compare {B}A Cost: {B} + {B, A}

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =
    - Compare {A}B   Cost: {A} + {A, B}
    - Compare {B}A   Cost: {B} + {B, A}

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =
    - Compare {A}B Cost: {A} + {A, B} = 159
    - Compare {B}A Cost: {B} + {B, A}

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =
    - Compare {A}B   Cost: {A} + {A, B} = 159
    - Compare {B}A   Cost: {B} + {B, A} = 179

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =
    - Compare {A}B   Cost: {A} + {A, B} = 159
    - Compare {B}A   Cost: {B} + {B, A} = 179

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost
{A, B}	AB	159

Selinger Optimizer Example (2/3)

- Here are 3 relations to join – A, B, C
- Step 2
  - Compute the cost of 2-way join by estimating all permutation using the record just cached
  - Ex. {A, B} =
    - Compare {A}B Cost: {A} + (A, B) = 159
    - Compare {B}A Cost: {B} + (B, A) = 179

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C   Cost: {A, B} + (A, B, C)
    - Compare {B, C}A   Cost: {B, C} + (B, C, A)
    - Compare {C, A}B   Cost: {C, A} + (C, A, B)

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C Cost: {A, B} + (A, B, C) = 259
    - Compare {B, C}A Cost: {B, C} + (B, C, A)
    - Compare {C, A}B Cost: {C, A} + (C, A, B)

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C Cost: {A, B} + (A, B, C) = 259
    - Compare {B, C}A Cost: {B, C} + (B, C, A)
    - Compare {C, A}B Cost: {C, A} + (C, A, B)

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C   Cost: {A, B} + (A, B, C) = 259
    - Compare {B, C}A   Cost: {B, C} + (B, C, A) = 111
    - Compare {C, A}B   Cost: {C, A} + (C, A, B)

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C   Cost: {A, B} + (A, B, C) = 259
    - Compare {B, C}A   Cost: {B, C} + (B, C, A) = 111
    - Compare {C, A}B   Cost: {C, A} + (C, A, B)

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C Cost: {A, B} + (A, B, C) = 259
    - Compare {B, C}A Cost: {B, C} + (B, C, A) = 111
    - Compare {C, A}B Cost: {C, A} + (C, A, B) = 100

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost

Selinger Optimizer Example (3/3)

- Here are 3 relations to join – A, B, C
- Step 3
  - Compute the cost of 3-way join by estimating all left-deep tree permutation using the step2's record
  - Ex. {A, B, C} =
    - Compare {A, B}C   Cost: {A, B} + (A, B, C) = 259
    - Compare {B, C}A   Cost: {B, C} + (B, C, A) = 111
    - Compare {C, A}B   Cost: {C, A} + (C, A, B) = 100

2-Set	Best Plan	Cost
{A, B}	AB	159
{A, C}	CA	98
{B, C}	CB	77

3-Set	Best Plan	Cost
{A, B, C}	CBA	100

Compare with Heuristic Planner

- Heuristic Planner
  - Greedy Search

1-Set	Best Plan	R
{A}	Index Select Plan	10
{B}	Table Plan	30
{C}	Select Plan	20

- Selinger Optimizer
  - All combination

3-Set	Best Plan	Cost
{A, B, C}	CBA	100

```
private Plan getAllCombination(Plan viewTrunk) {
      long finalKey = 0;

      // for layer = 1, use select down strategy to construct
      for (TablePlanner tp: tablePlanners) {
            Plan bestPlan = null;
            if (viewTrunk != null) {
                  bestPlan = tp.makeJoinPlan(viewTrunk);
                  if (bestPlan == null)
                  bestPlan = tp.makeProductPlan(viewTrunk);
            }
            else
                 bestPlan = tp.makeSelectPlan();

            AccessPath ap = new AccessPath(tp, bestPlan);
            lookupTbl.put(ap.getAPId(), ap);

            // compute final access path id
            finalKey += ap.getAPId();
      }


      .
      .
      .

}
```

```java
// construct all combination layer by layer
for (int layer = 2; layer <= tablePlanners.size(); layer++) {
	Set<Long> keySet = new HashSet<Long>(lookupTbl.keySet());

	for (TablePlanner rightOne: tablePlanners) {
		for (Long key: keySet) {
			AccessPath leftTrunk = lookupTbl.get(key);

			// cannot join with table which (layer-1) combination already included
			if (leftTrunk.isUsed(rightOne.getId()))
				continue;

			// do join
			Plan bestPlan = rightOne.makeJoinPlan(leftTrunk.getPlan());
			if (bestPlan == null)
				bestPlan = rightOne.makeProductPlan(leftTrunk.getPlan());

			AccessPath candidate = new AccessPath(leftTrunk, rightOne, bestPlan);
			AccessPath ap = lookupTbl.get(candidate.getAPId());

			// there is no access path contains this combination
			if (ap == null) {
				lookupTbl.put(candidate.getAPId(), candidate);
			}
			// check whether new access path is better than previous
			else {
				if (candidate.getCost() < ap.getCost())
					lookupTbl.put(candidate.getAPId(), candidate);
			}
		}
	}
	// remove the elements belong to layer-1
	// because in the next layer we only need this layer's combination
	for (Long key: keySet)
		lookupTbl.remove(key);
}

return lookupTbl.get(finalKey).getPlan();
```

– Iterate all table planners to join with all existing (layer-1) combination to construct this layer

```java
public class AccessPath {
      private Plan p;
      private AccessPathId apId;
      private long cost = 0;
      private ArrayList<Integer> tblUsed = new ArrayList<Integer>();

      public class AccessPathId {
            long id;

            AccessPathId(TablePlanner tp) {
                  this.id = (long) Math.pow(2,tp.getId());
            }

            AccessPathId(AccessPath ap, TablePlanner tp) {
                  this.id = ap.getAPId()+(long) Math.pow(2,tp.getId());
            }
            public long getID() {
                  return id;
            }
      }

      public AccessPath (TablePlanner newTp, Plan p) {
            this.p = p;
            this.tblUsed.add(newTp.getId());
            this.apId = new AccessPathId(newTp);
            this.cost = p.recordsOutput();
      }
      public AccessPath (AccessPath preAp, TablePlanner newTp, Plan p) {
            this.p = p;
            this.tblUsed.addAll(preAp.getTblUsed());
            this.tblUsed.add(newTp.getId());
            this.apId = new AccessPathId(preAp, newTp);

            // approximate cost = previous cost + new cost
            this.cost = preAp.getCost() + p.recordsOutput();
      }
}
```

- Using **sum of pow(2, tp.id)** to represent the combination of tables in this access path
- Using **pow(2, tp.id)** to avoid problems with different combinations but with the same apID
- Then we can use apID as the key of the lookup table

```java
public class AccessPath {
      private Plan p;
      private AccessPathId apId;
      private long cost = 0;
      private ArrayList<Integer> tblUsed = new ArrayList<Integer>();

      public class AccessPathId {
            long id;

            AccessPathId(TablePlanner tp) {
                  this.id = (long) Math.pow(2,tp.getId());
            }

            AccessPathId(AccessPath ap, TablePlanner tp) {
                  this.id = ap.getAPId()+(long) Math.pow(2,tp.getId());
            }
            public long getID() {
                  return id;
            }
      }


      public AccessPath (TablePlanner newTp, Plan p) {
            this.p = p;
            this.tblUsed.add(newTp.getId());
            this.apId = new AccessPathId(newTp);
            this.cost = p.recordsOutput();
      }
      public AccessPath (AccessPath preAp, TablePlanner newTp, Plan p) {
            this.p = p;
            this.tblUsed.addAll(preAp.getTblUsed());
            this.tblUsed.add(newTp.getId());
            this.apId = new AccessPathId(preAp, newTp);

            // approximate cost = previous cost + new cost
            this.cost = preAp.getCost() + p.recordsOutput();
      }
}
```

- Using **sum of pow(2, tp.id)** to represent the combination of tables in this access path
- Using **pow(2, tp.id)** to avoid problems with different combinations but with the same apID
- Then we can use apID as the key of the lookup table

- Approximate B(root) using R(p1) + R(c1)…

Reference

- https://db.inf.uni-tuebingen.de/staticfiles/teaching/ws1011/db2/db2-selectivity.pdf
- https://www.cise.ufl.edu/~adobra/approxqp/histograms2
- https://pdfs.semanticscholar.org/b024/0a44105fa0a0967d96d109aac9f021902ebb.pdf