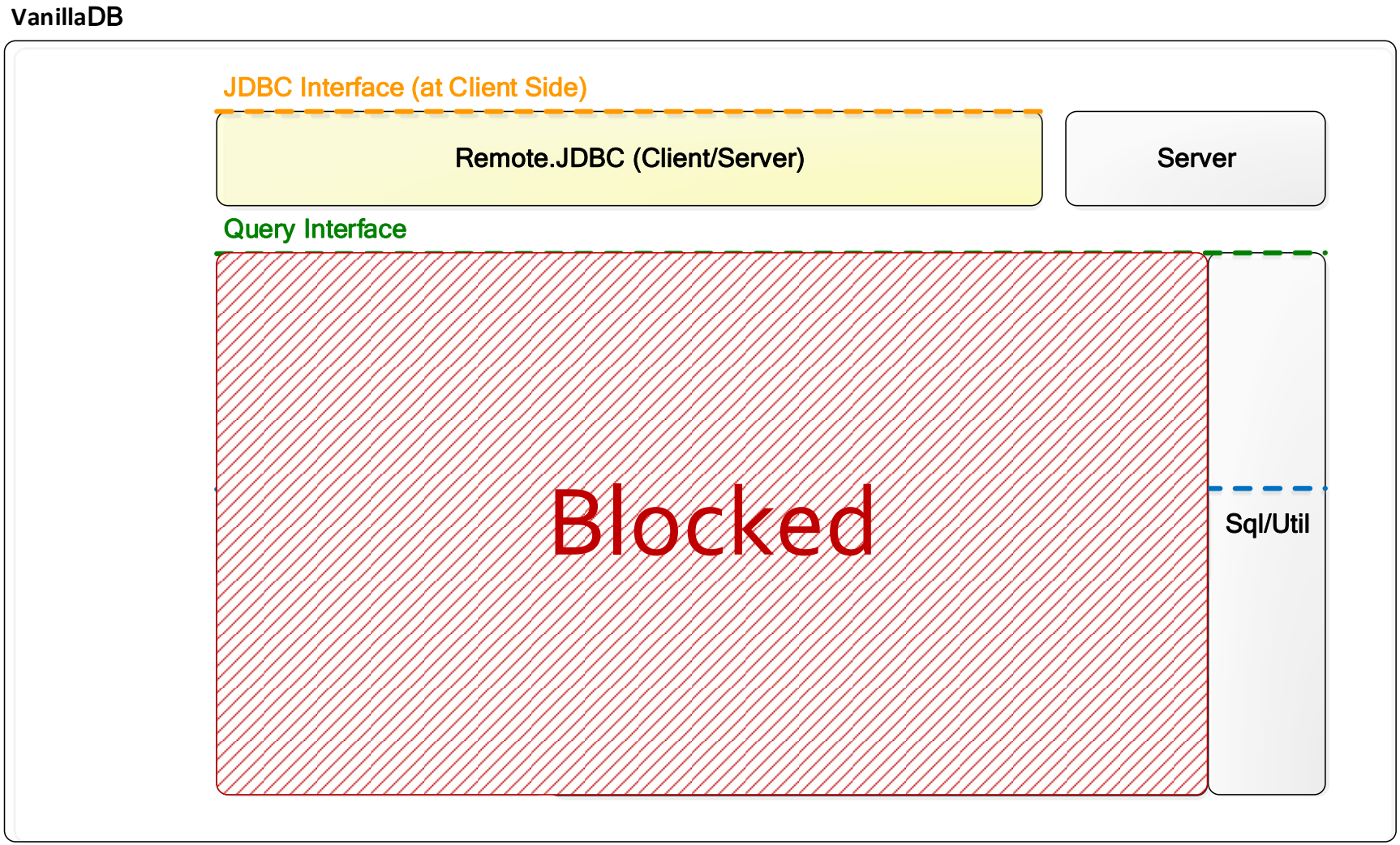
Assignment 3

Cloud Databases

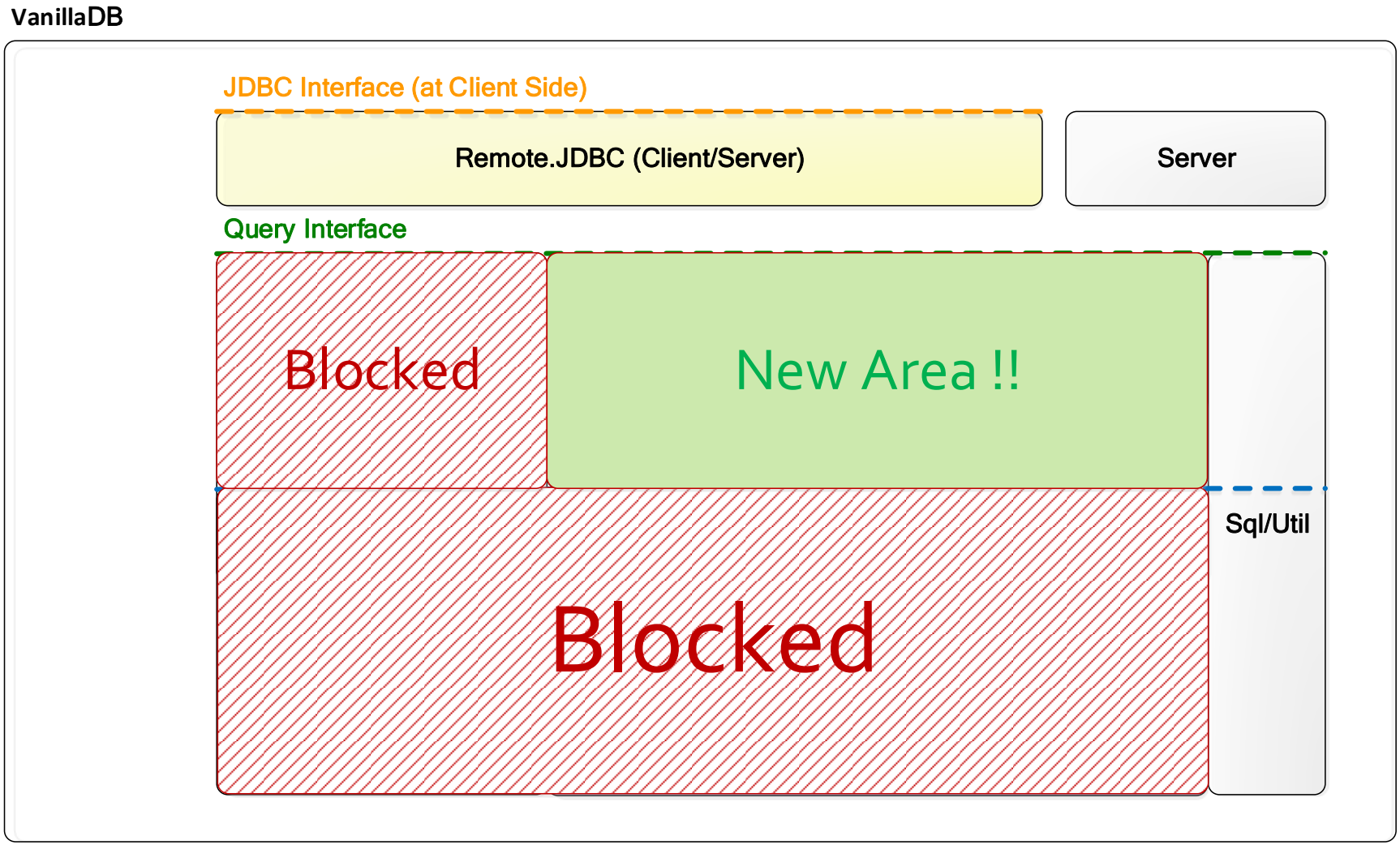
DataLab

CS, NTHU

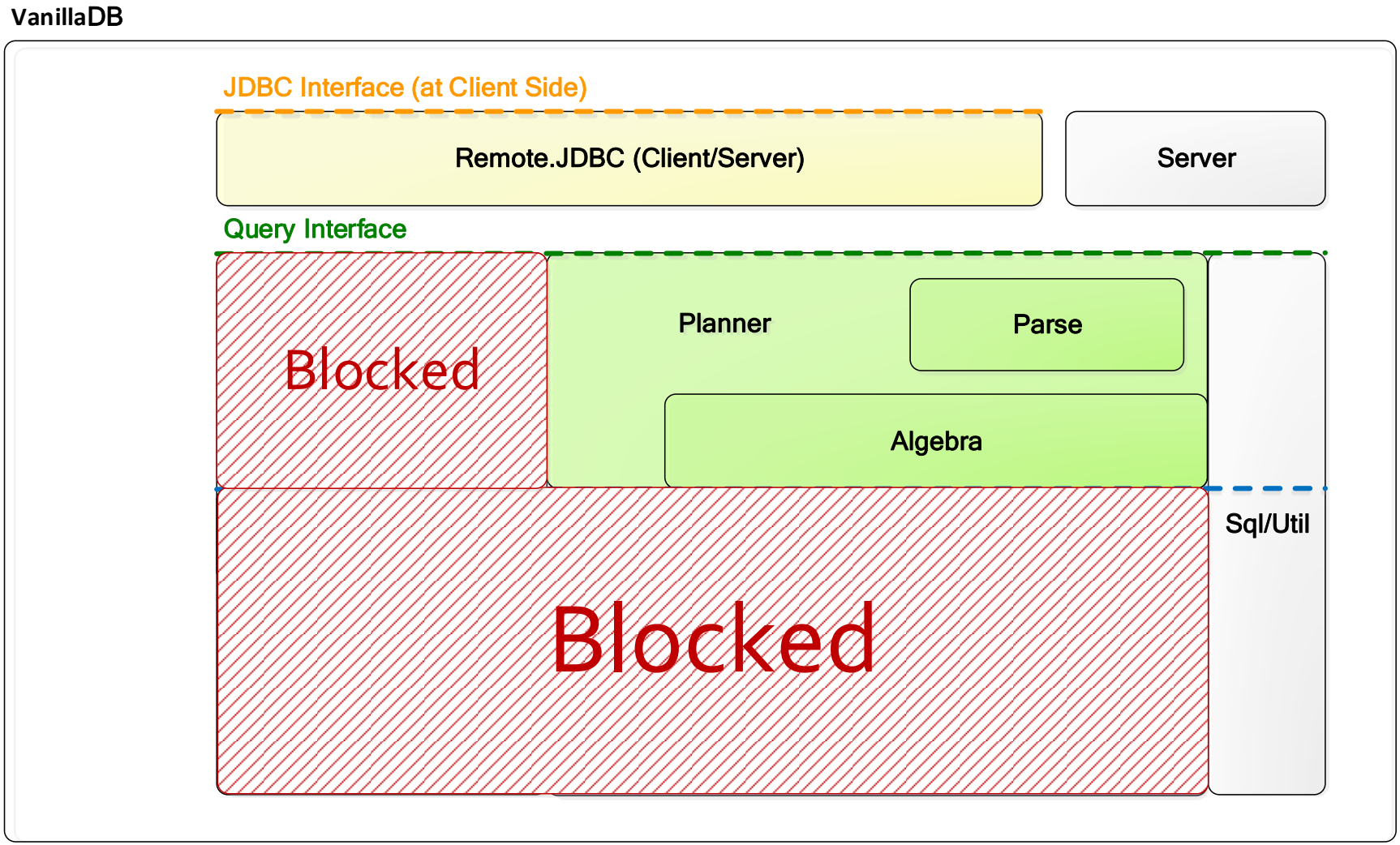
Last Time


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
Sql/Util

This Time


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
New Area !!
Sql/Util
Blocked

This Time


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
Planner
Parse
Algebra
Sql/Util
Blocked

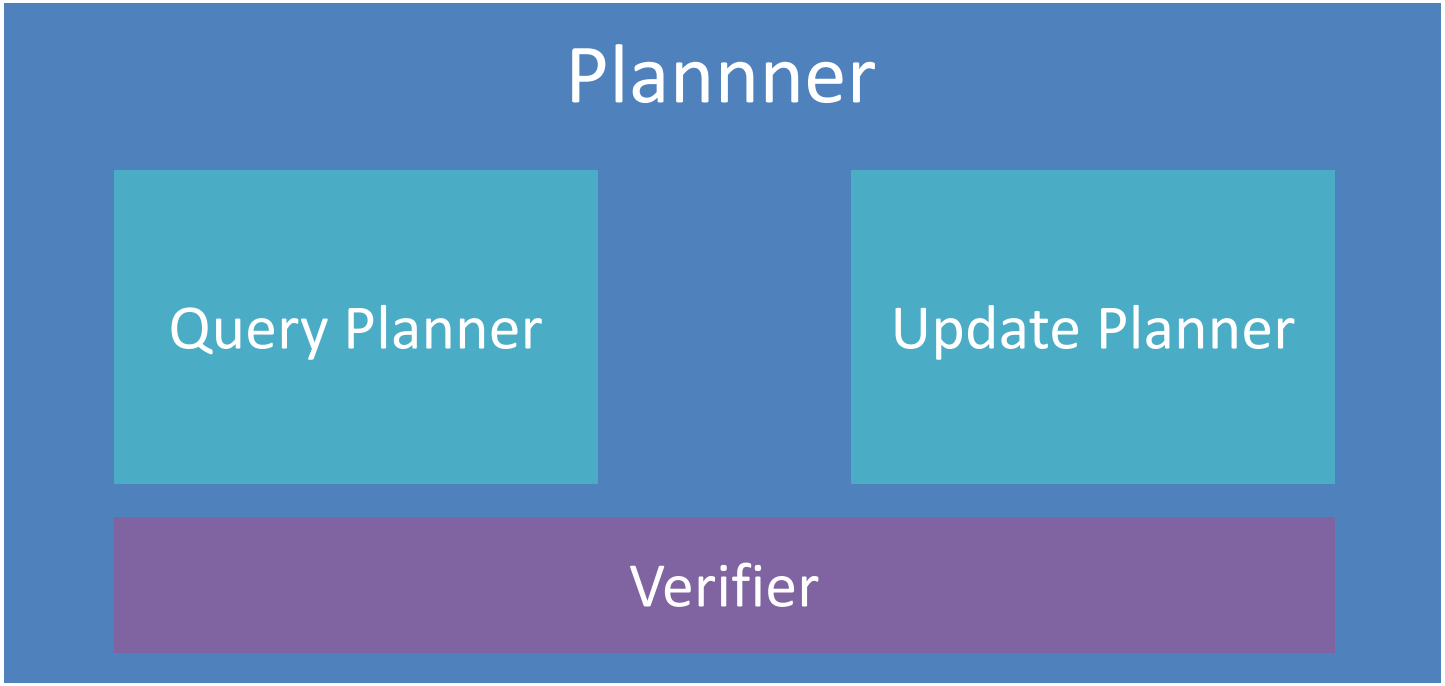
Outline

- Query package
  - Parse package
  - Algebra package
  - Planner package

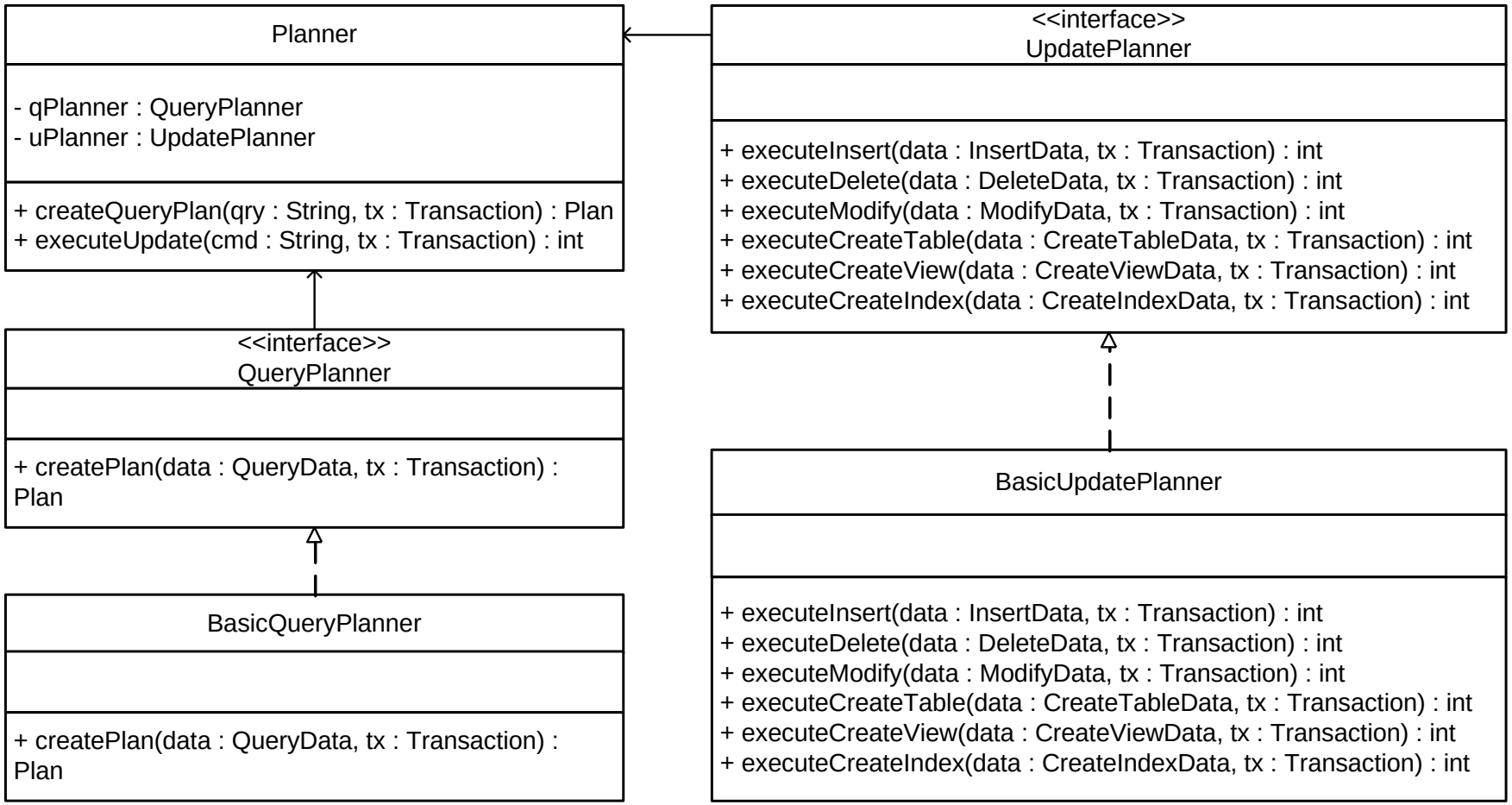
Planner

- The one puts all these together
  1. Accepts a query
  2. Creates a parser to parse the query
  3. Verifies all parameters are reasonable
  4. Generates a plan tree according to the query

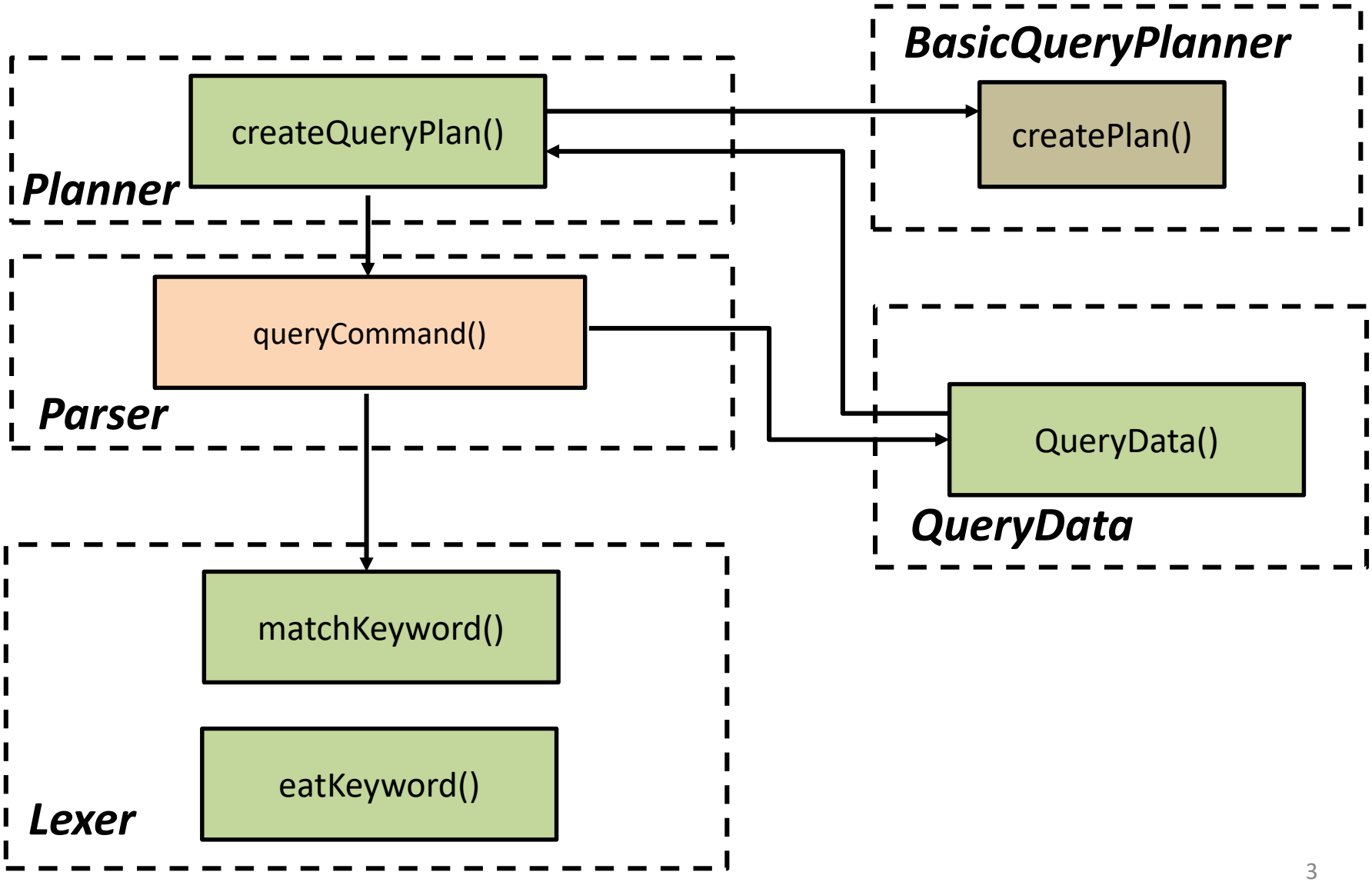
`planner` Pcakage



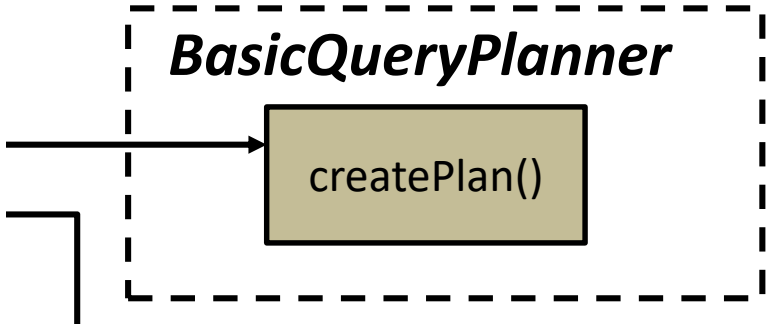
Basic Implementation



Overview


BasicQueryPlanner
createQueryPlan()
createPlan()
Planner
queryCommand()
Parser
QueryData()
QueryData
matchKeyword()
eatKeyword()
Lexer

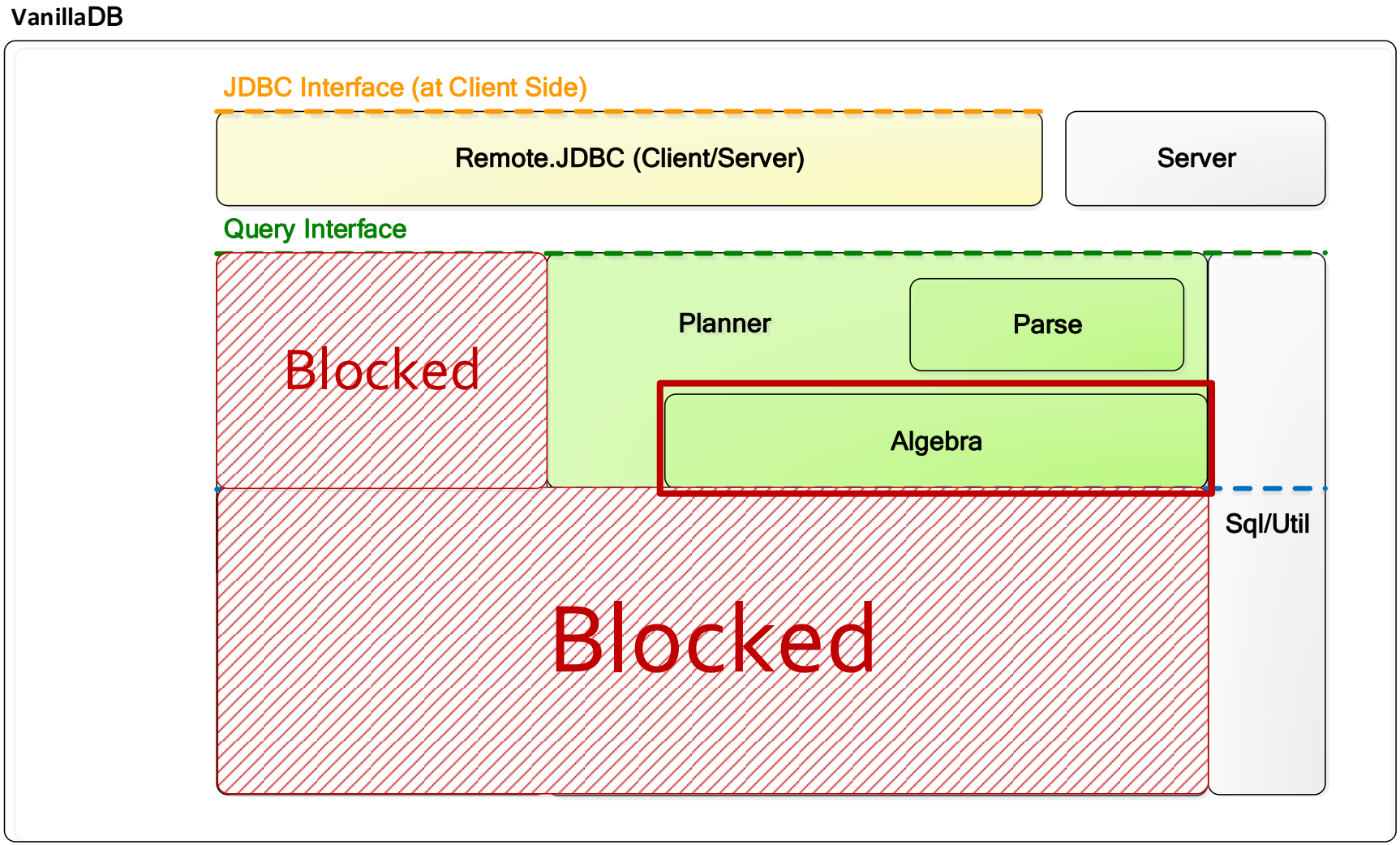
BasicQueryPlanner



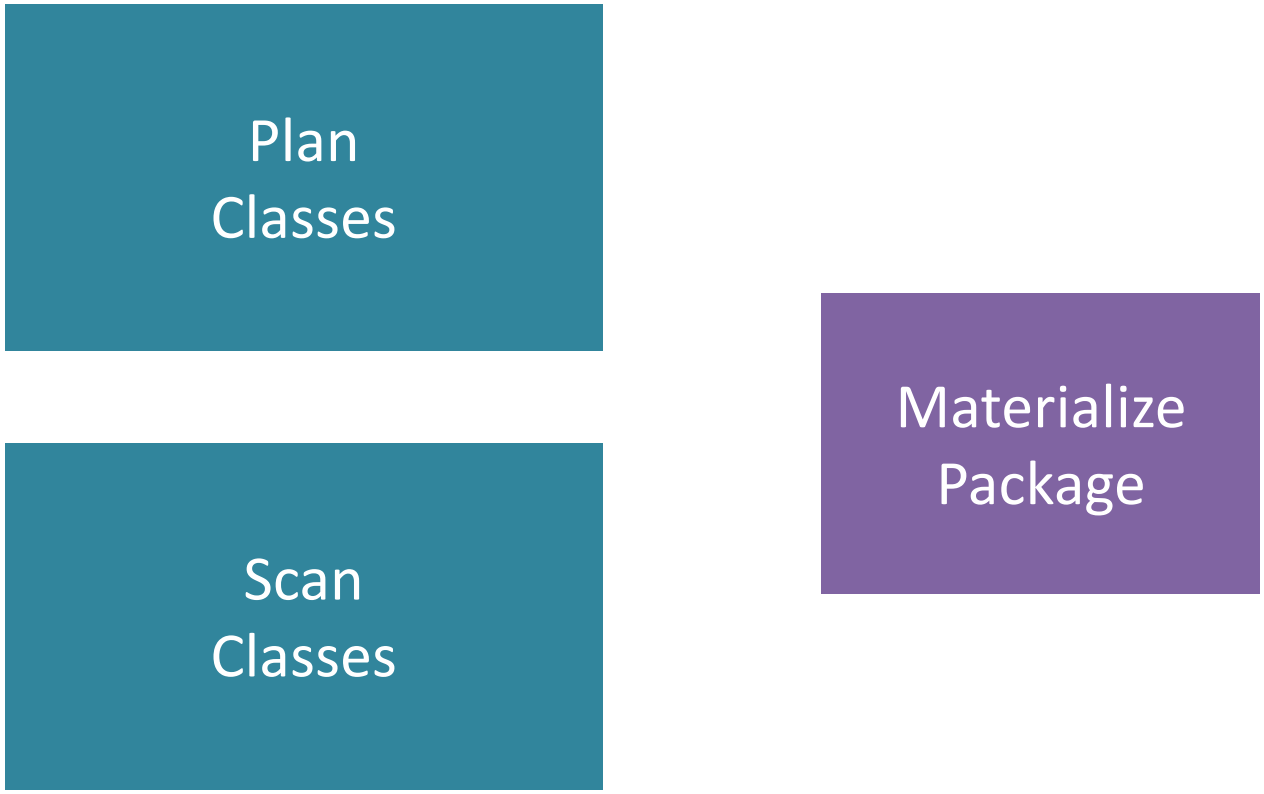
BasicQueryPlanner

```
public class BasicQueryPlanner implements QueryPlanner {

    /**
     * Creates a query plan as follows. It first takes the product of all tables
     * and views; it then selects on the predicate; and finally it projects on
     * the field list.
     */
    @Override
    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);

        return p;
    }
}
```

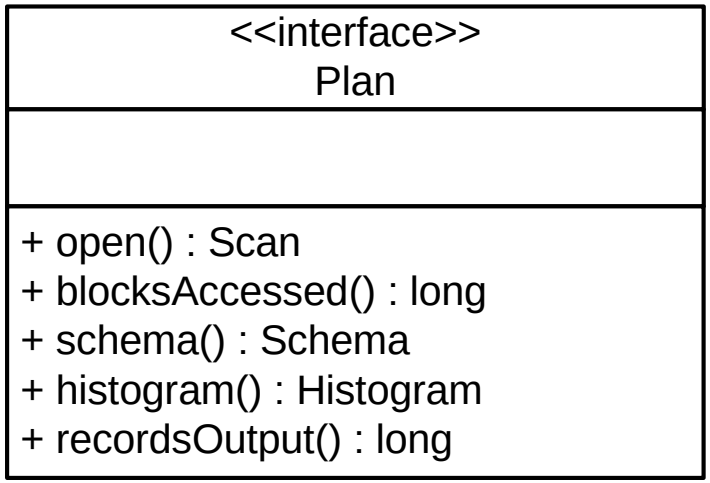
Where Are We ?

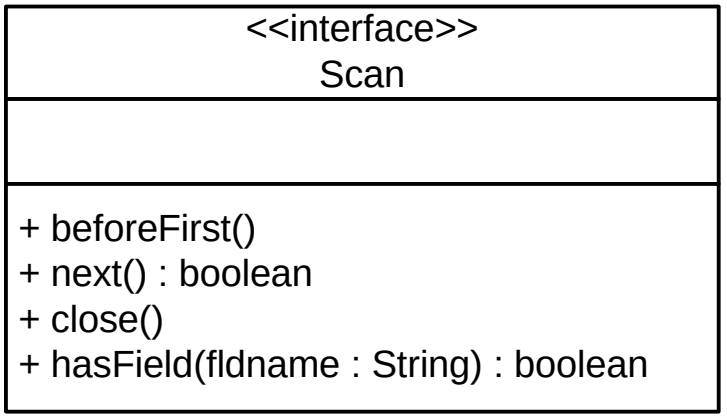

VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
Planner
Parse
Algebra
Sql/Util
Blocked

`algebra` Package



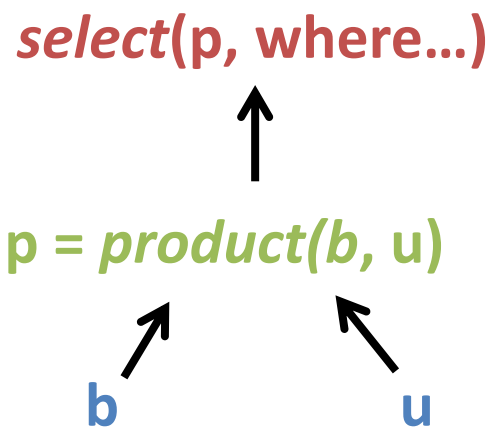
Plan & Scan


<<interface>>
Plan
+ open() : Scan
+ blocksAccessed() : long
+ schema() : Schema
+ histogram() : Histogram
+ recordsOutput() : long


<<interface>>
Scan
+ beforeFirst()
+ next() : boolean
+ close()
+ hasField(fldname : String) : boolean

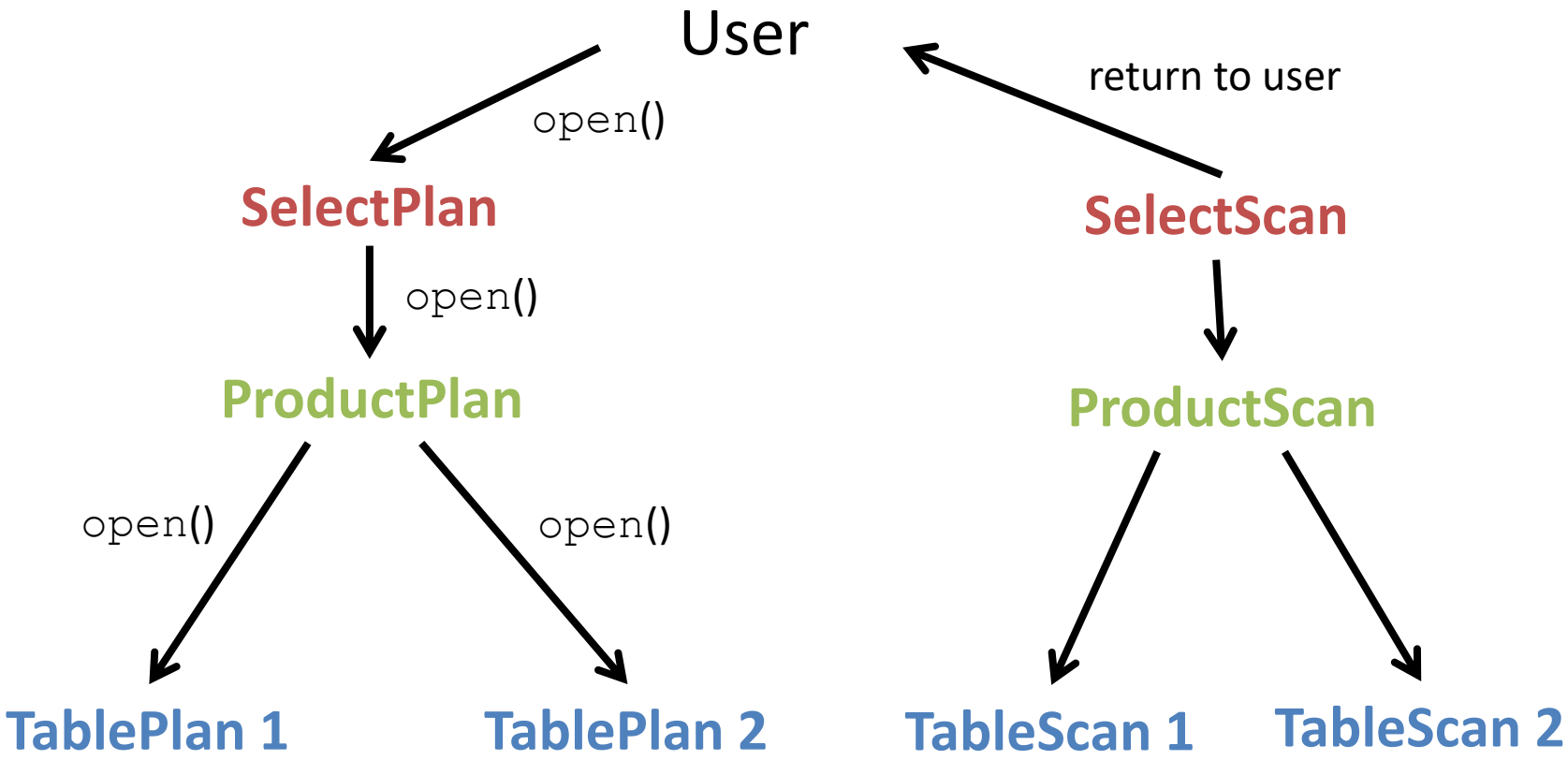
Using a Query Plan

```
VanillaDb.init("studentdb");
Transaction tx = VanillaDb.txMgr().newTransaction(
    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();
```



What Happened When We Called `open()`?

open()


User
open()
return to user
SelectPlan
SelectScan
open()
ProductPlan
ProductScan
open()
open()
TablePlan 1
TablePlan 2
TableScan 1
TableScan 2

How Do Scans Work ?

Example

project(s, select blog_id)

↓ `beforeFirst()`

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

↓ `beforeFirst()`

product(b, u)

`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)

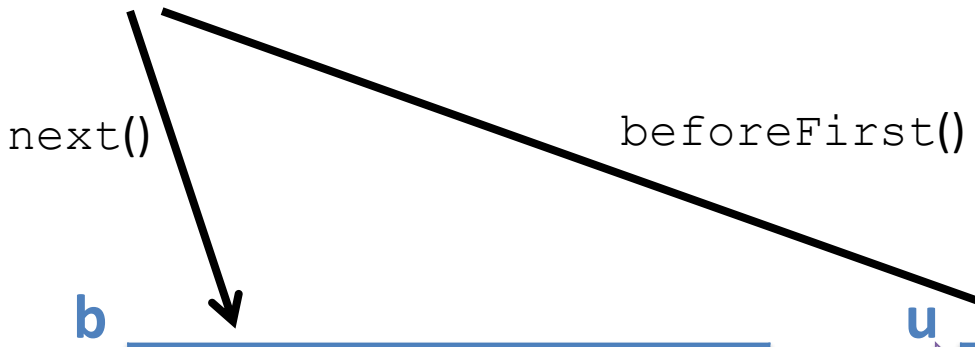
↓ beforeFirst()

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

↓ beforeFirst()

product(b, u)

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



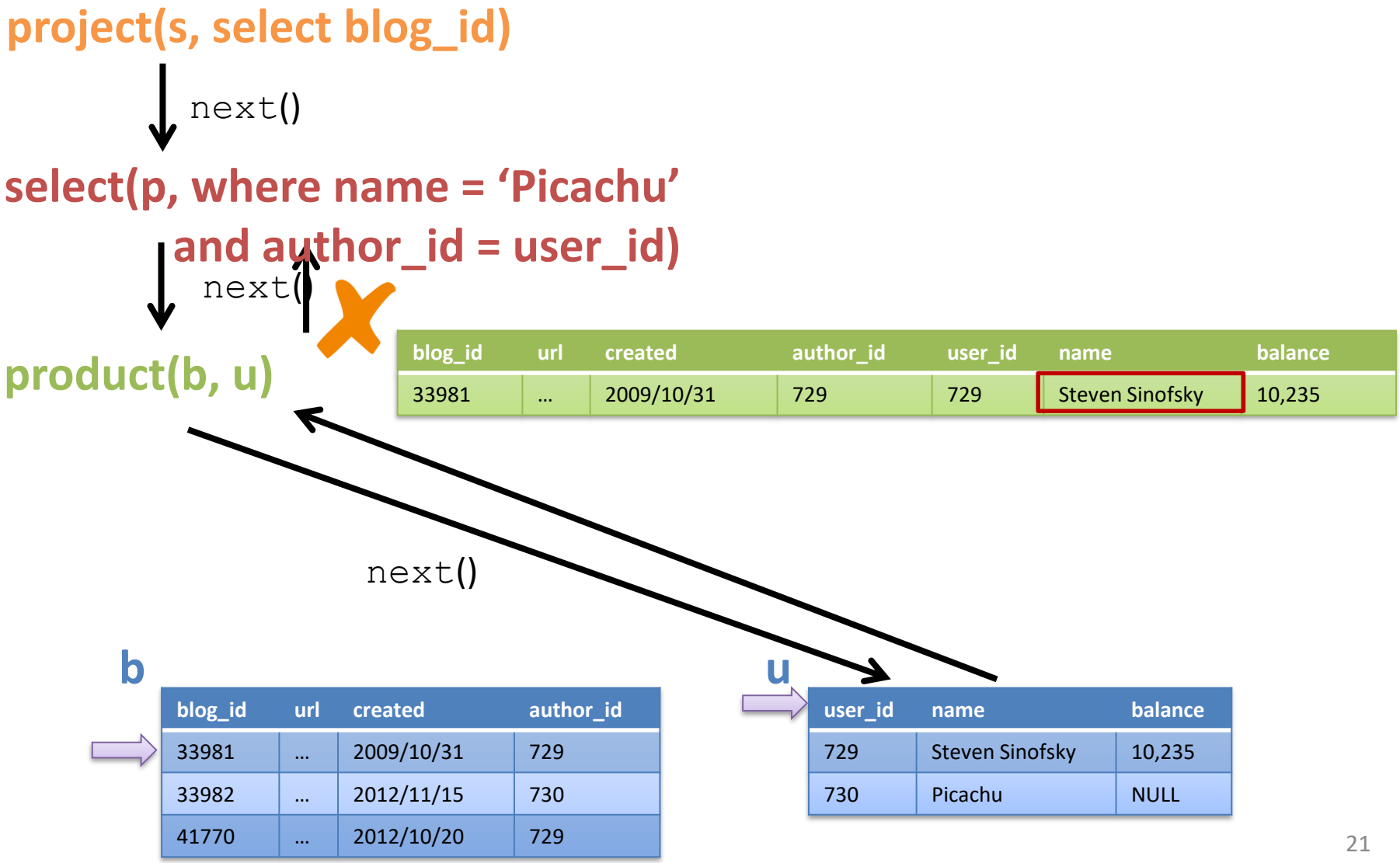
b

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

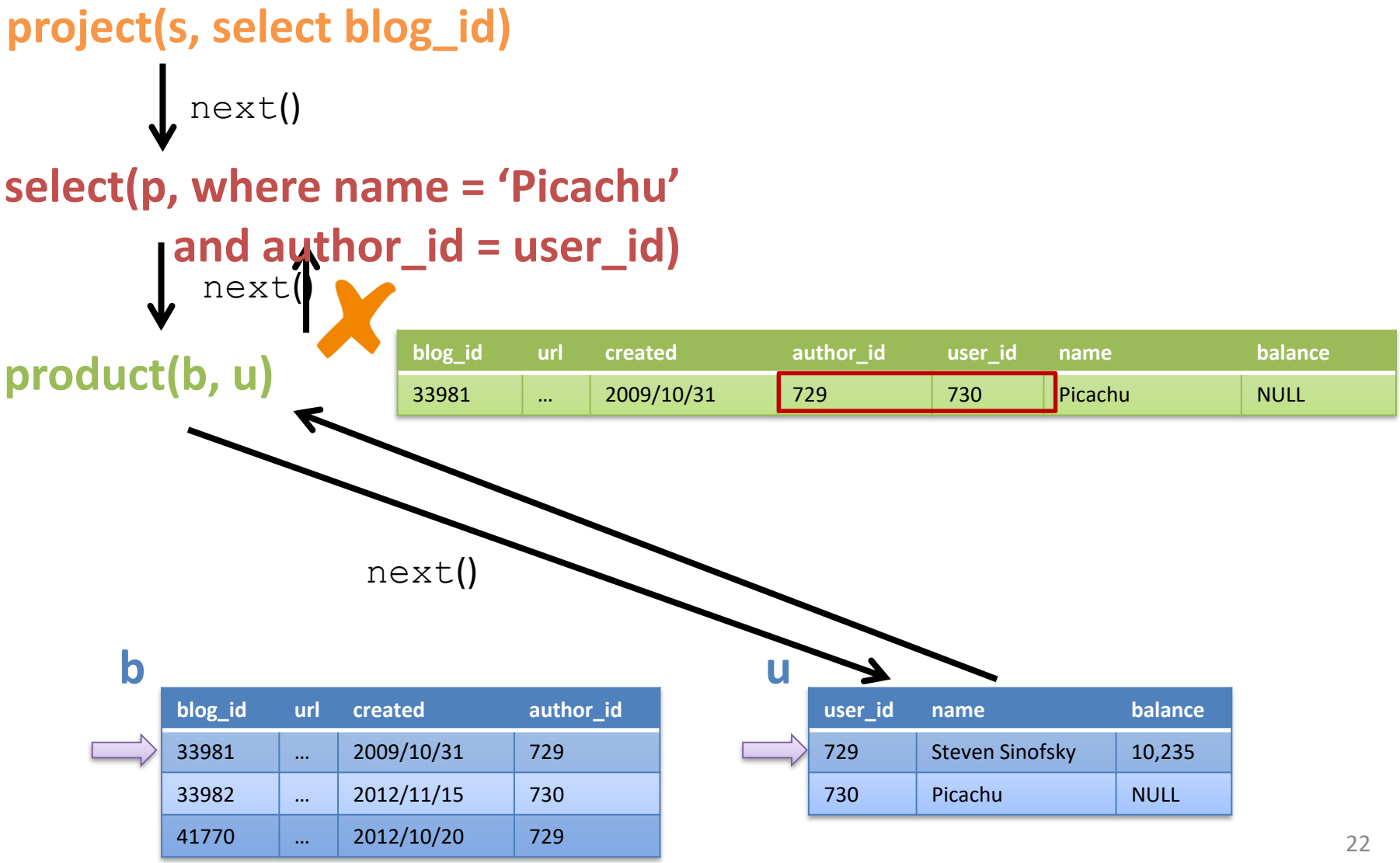


blog_id	url	created	author_id	user_id	name	balance
33981	…	2009/10/31	729	729	Steven Sinofsky	10,235

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

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

Example

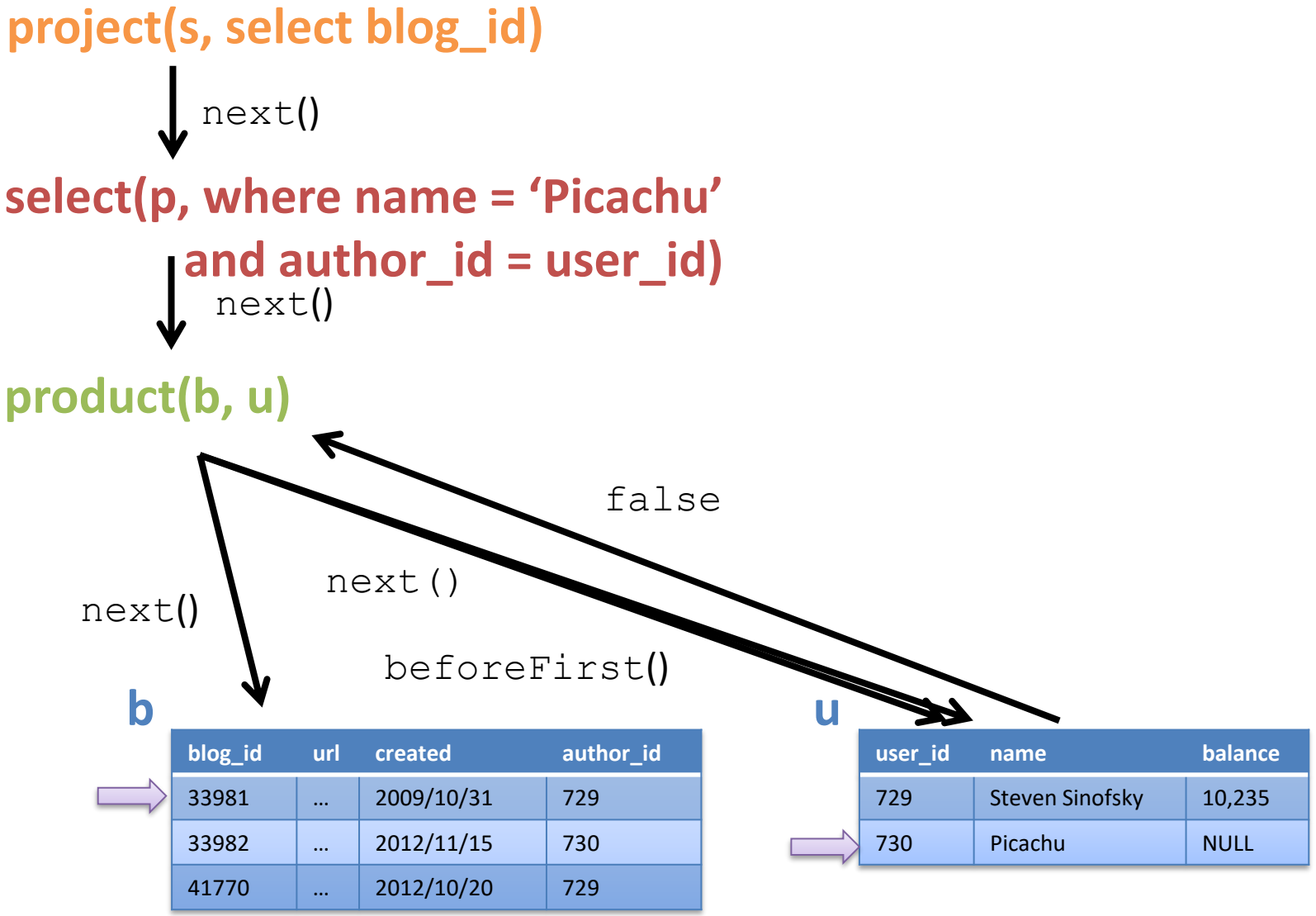


blog_id	url	created	author_id	user_id	name	balance
33981	…	2009/10/31	729	730	Picachu	NULL

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

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

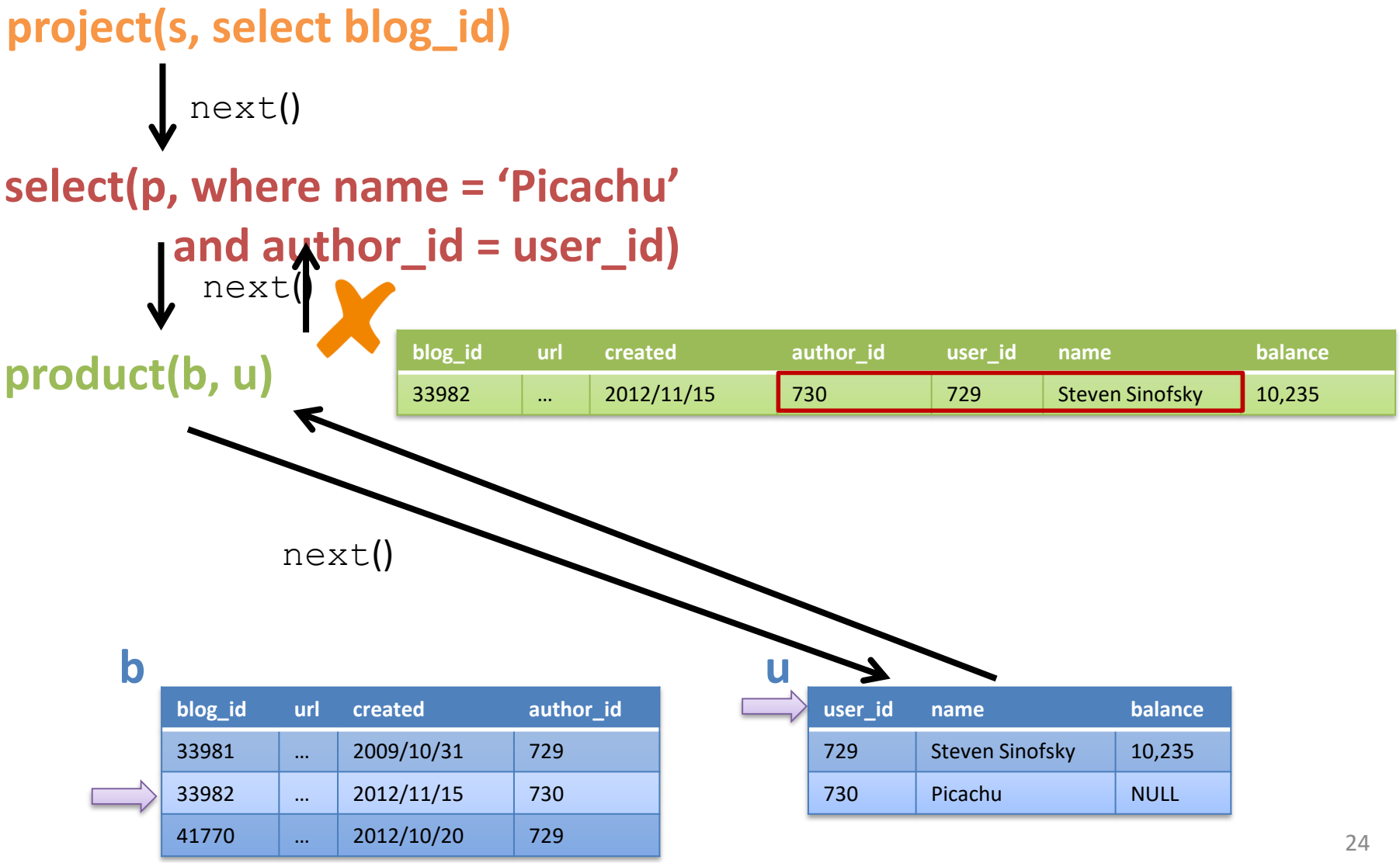
Example



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

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

Example



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

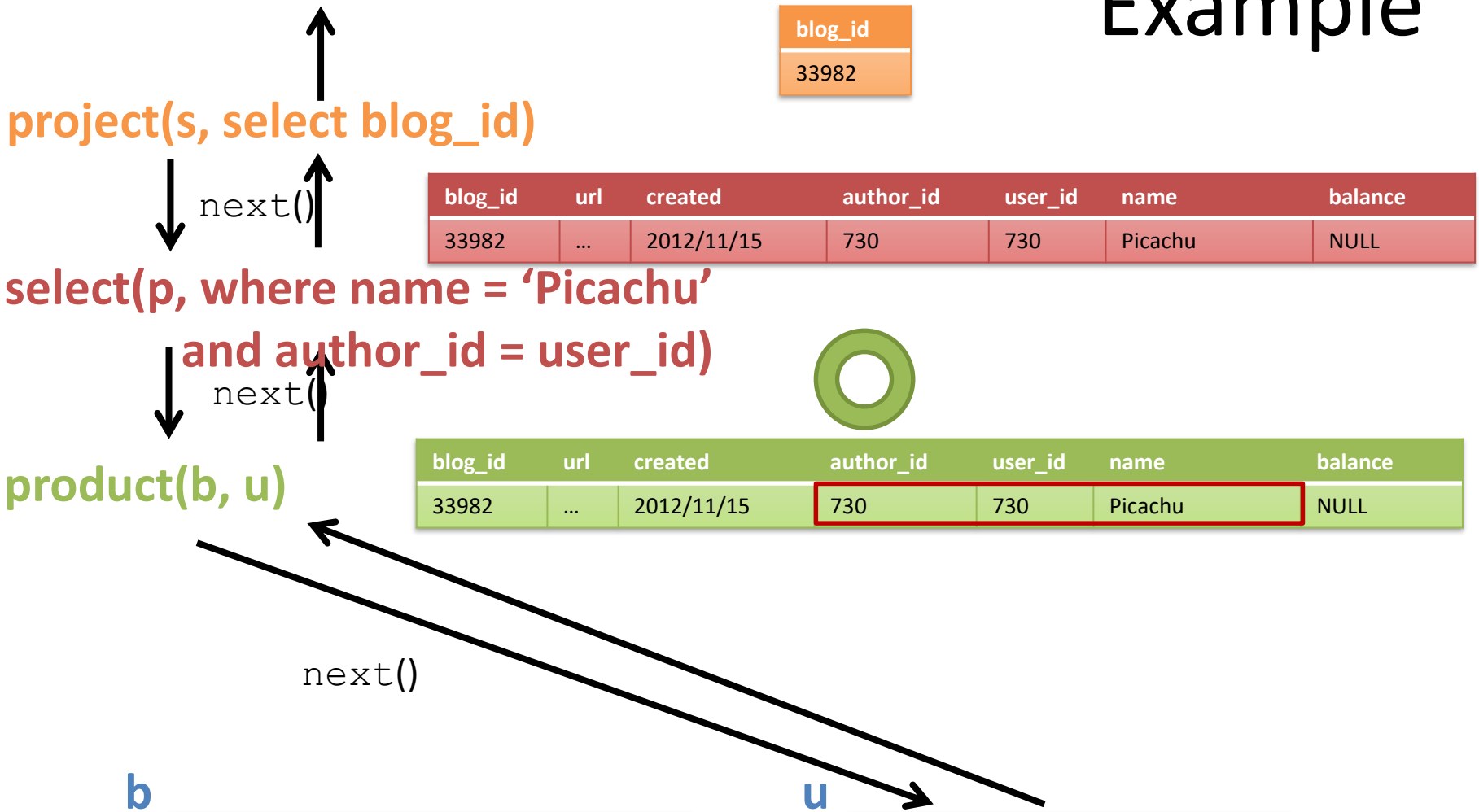
b

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

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

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

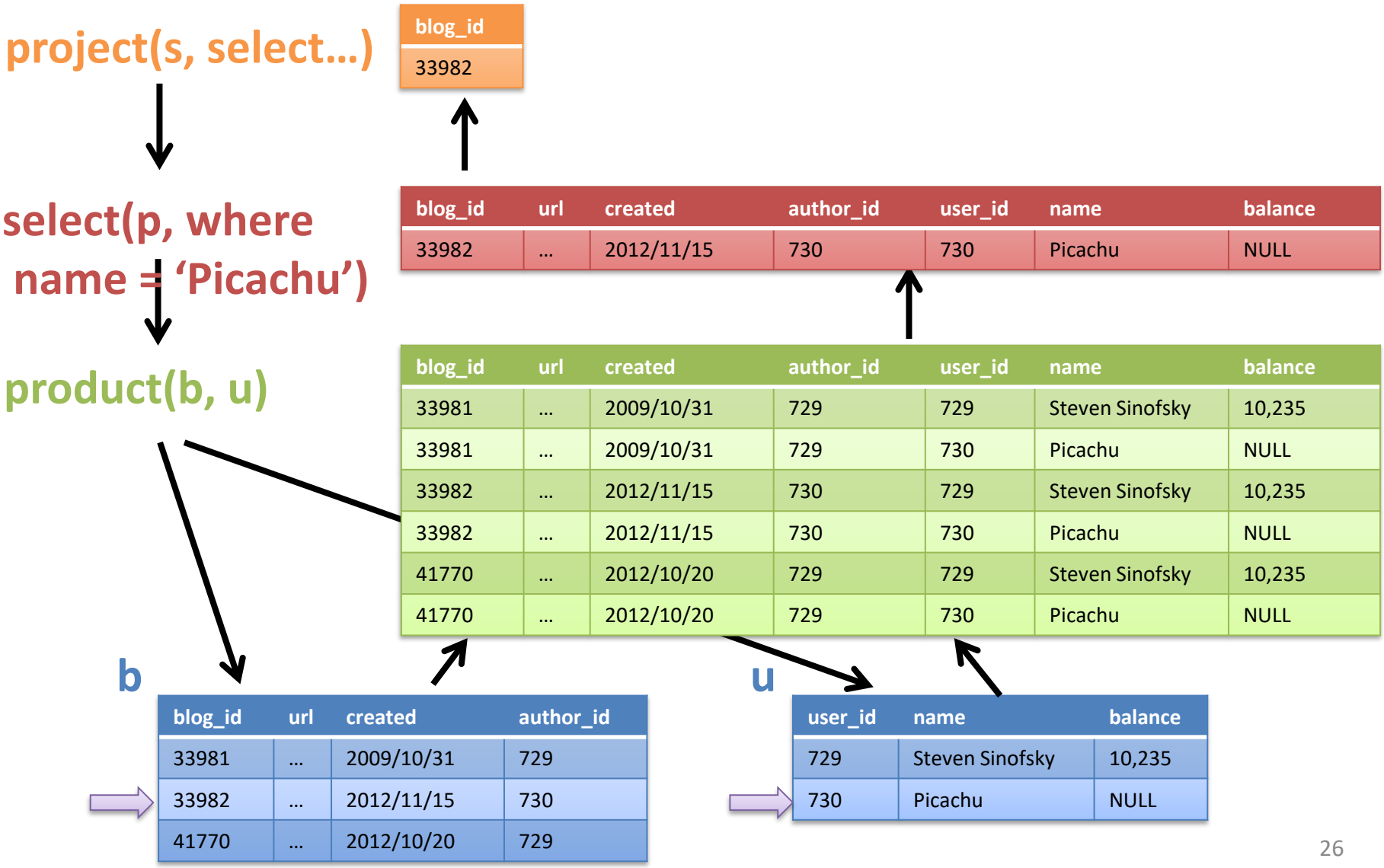
b

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

u

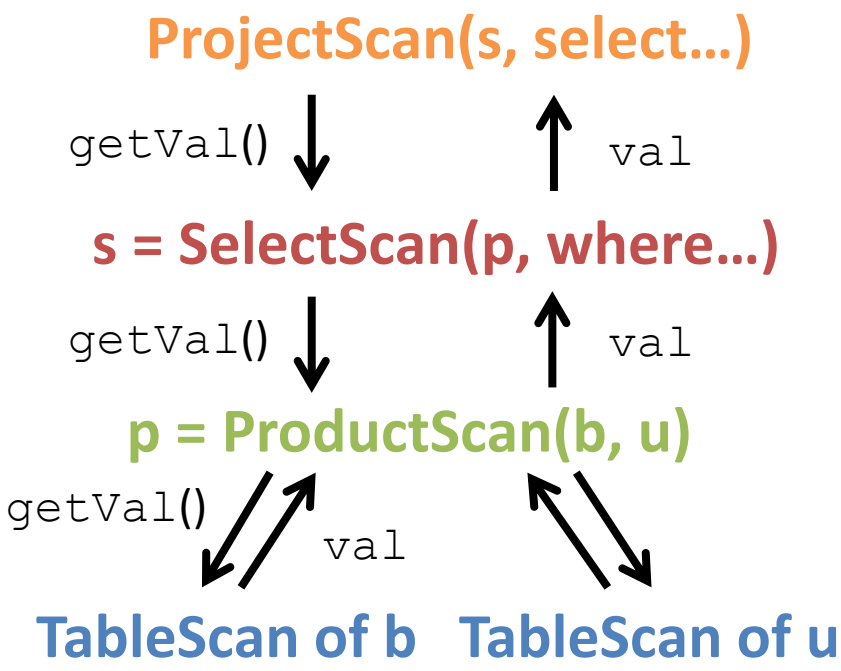
user_id	name	balance
729	Steven Sinofsky	10,235
730	Picachu	NULL

Example


project(s, select…)
select(p, where name = ‘Picachu’)
product(b, u)
blog_id
33982
blog_id | url | created | author_id | user_id | name | balance
33982 | … | 2012/11/15 | 730 | 730 | Picachu | NULL
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

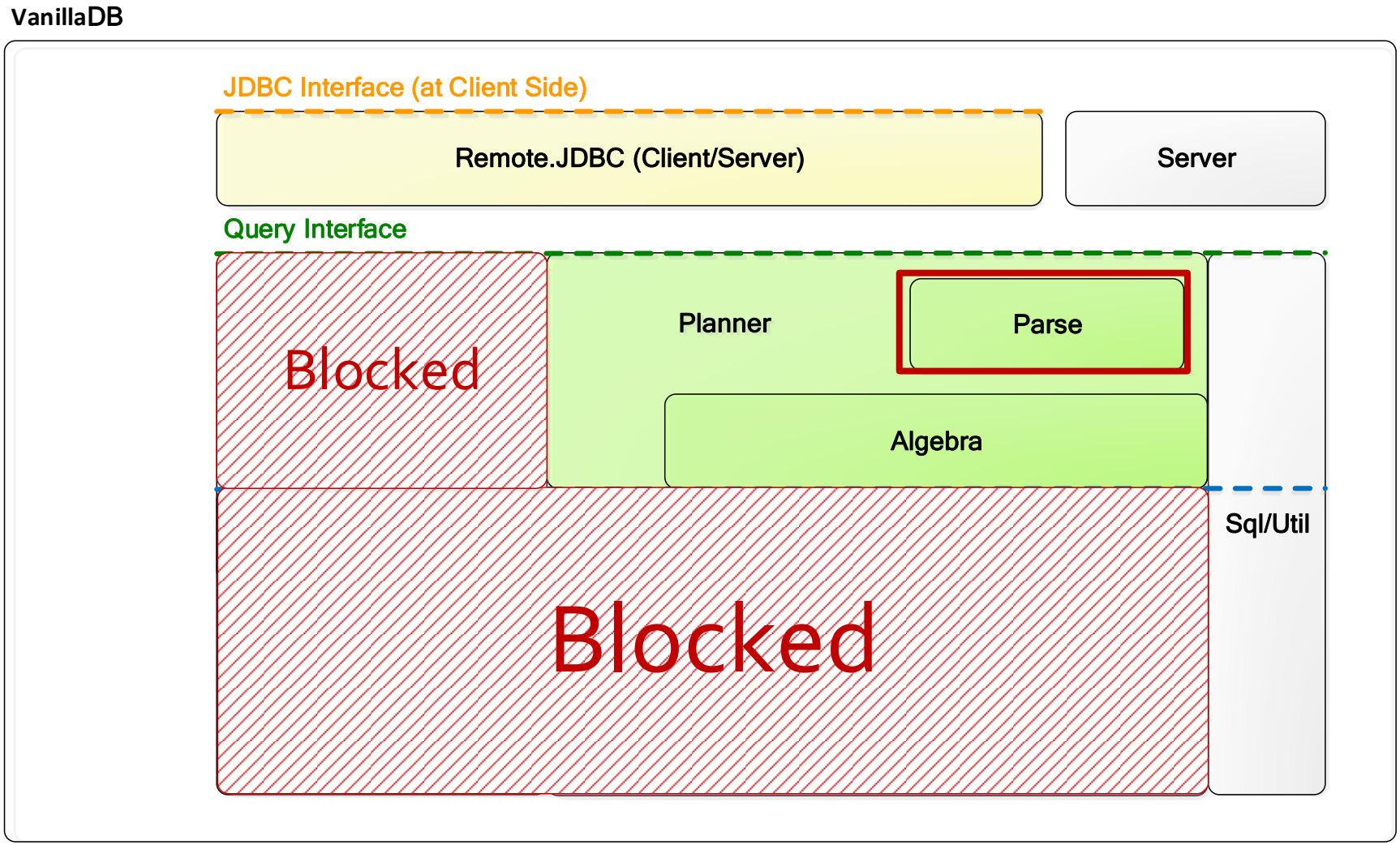
- The above operators implement ***pipelined scanning***
  - Calling a method of a node results in recursively calling the same methods of child nodes on-the-fly
  - Records are computed one at a time as needed --- no intermediate records are saved



Pipelined vs. Materialized

- Despite its simplicity, pipelined scanning is inefficient in some cases
  - E.g., when implementing `SortScan` (for `ORDER BY`)
  - It needs to iterate all children to find the next record
- For such cases, we use ***materialized scanning***
  - 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

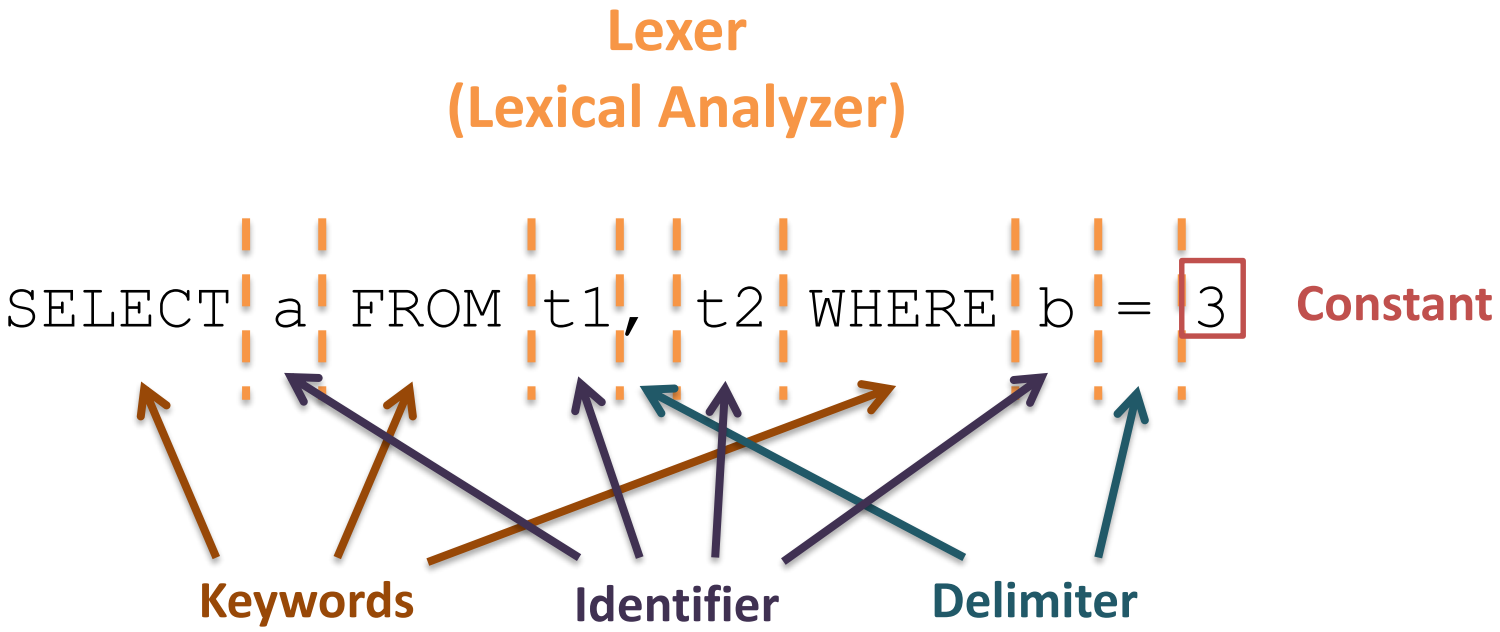
Where Are We ?


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
Planner
Parse
Algebra
Sql/Util
Blocked

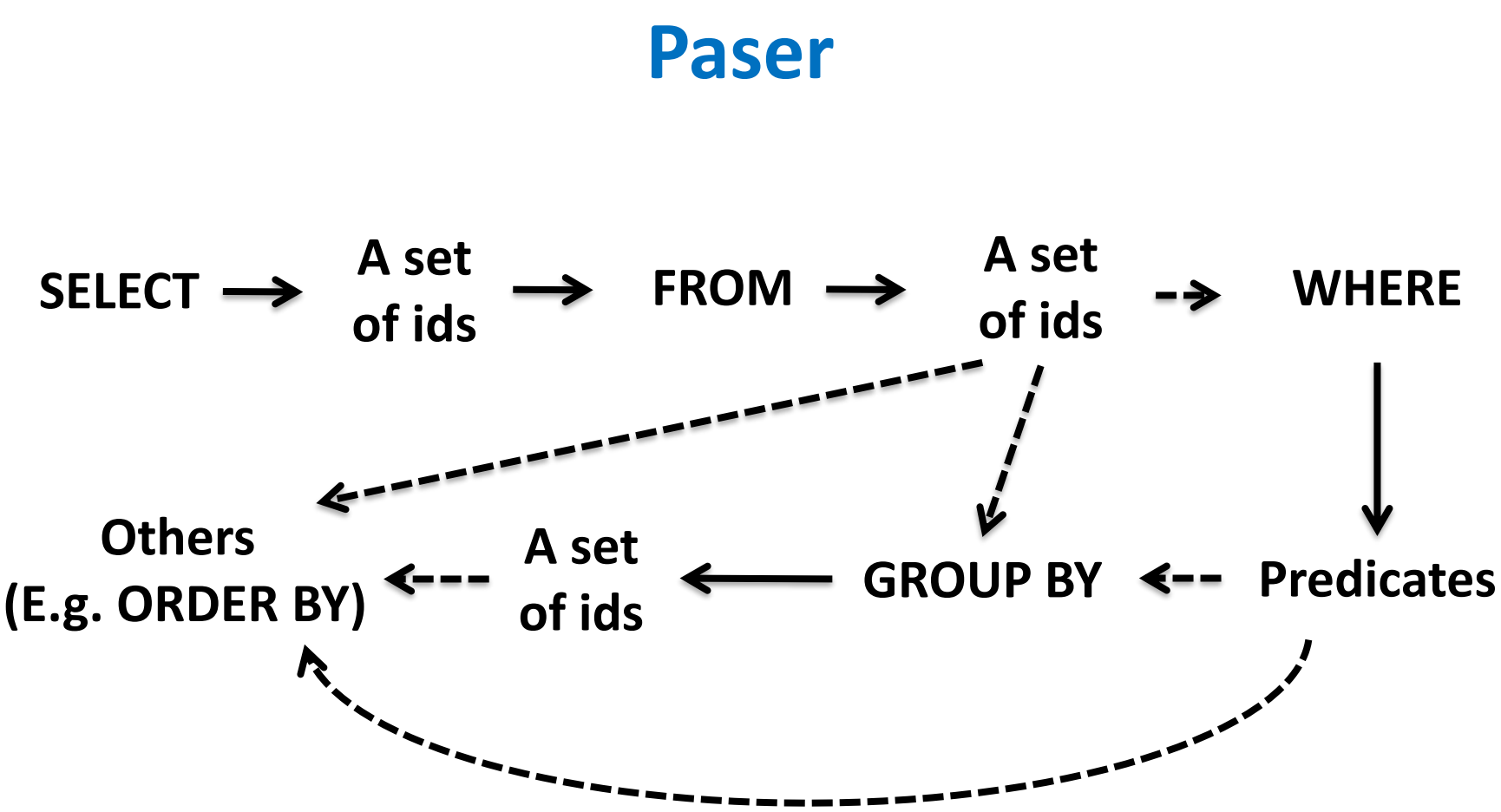
Two Steps of Parsing A SQL

- Lexer
  - Tokenizing
  - Identifying keywords, IDs, values, delimiters
- Parser
  - Checking syntax
  - Identifying the action and the parameters

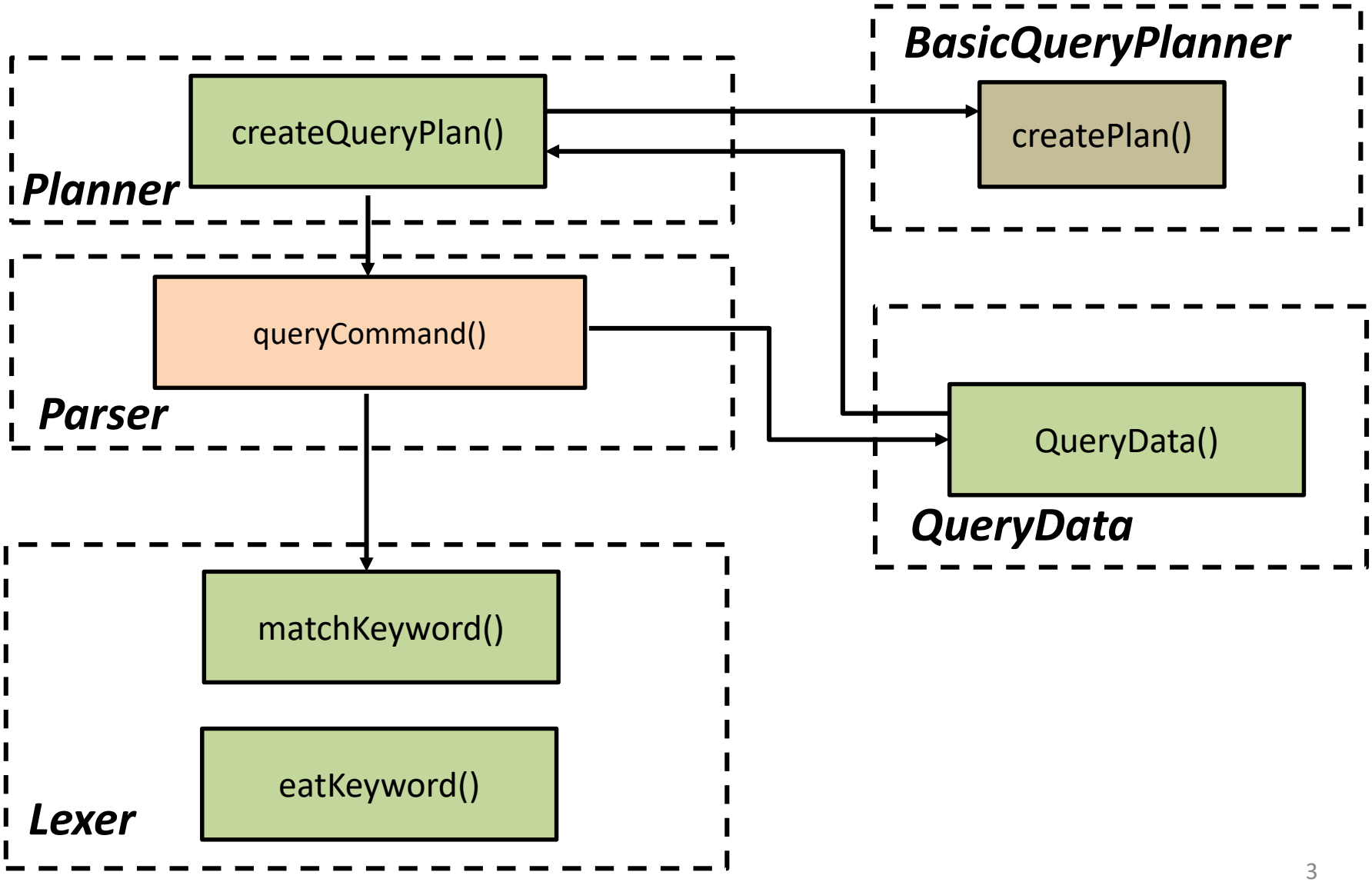
How VanillaCore Parses A SQL (1/2)



How VanillaCore Parses A SQL (2/2)



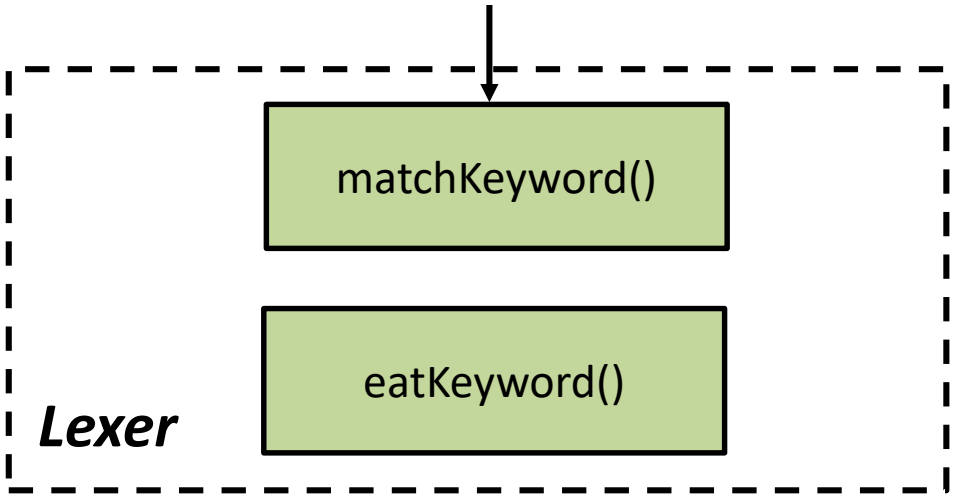
Overview


BasicQueryPlanner
createQueryPlan()
createPlan()
Planner
queryCommand()
Parser
QueryData()
QueryData
matchKeyword()
eatKeyword()
Lexer

Modified/Added Classes

- Parse
  - *Lexer*
  - *Parser*
  - *queryData*
- Algebra
  - *ExplaiunPlan*、*ExplainScan*
  - *TablePlan*、*ProductPlan*、*SelectPlan*、 *SortPlan*、 *GroupByPlan*、*ProjectPlan*
- Planner
  - *BasicQueryPlanner*
- An example of Experiment Results

Lexer



Parse

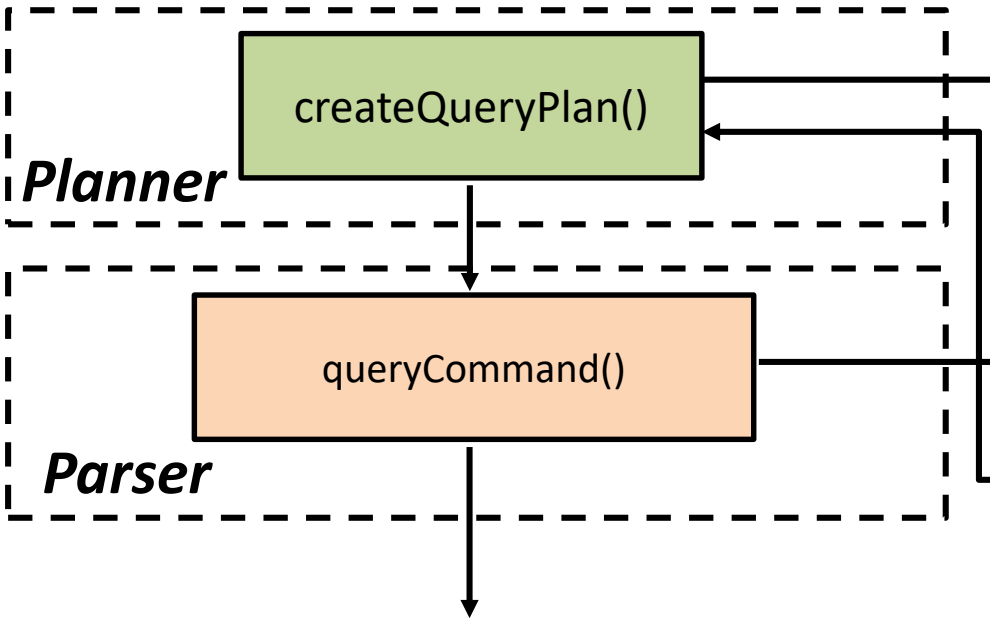
- Lexer

```
private void initKeywords() {
    keywords = Arrays.asList("select", "from", "where", "and", "insert",
            "into", "values", "delete", "drop", "update", "set", "create", "table",
            "int", "double", "varchar", "view", "as", "index", "on",
            "long", "order", "by", "asc", "desc", "sum", "count", "avg",
            "min", "max", "distinct", "group", "add", "sub", "mul", "div",
            "using", "hash", "btree");
}
```

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

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

Parser

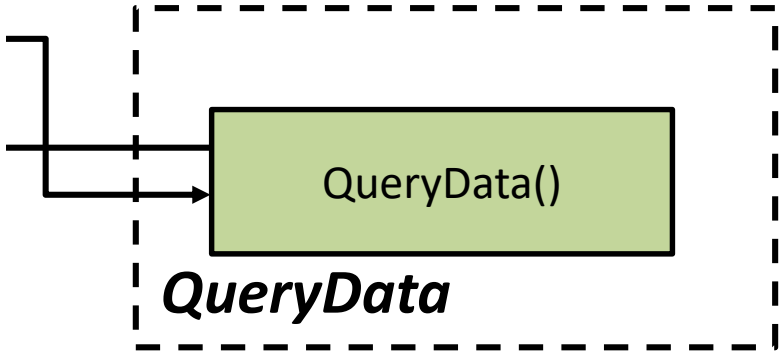


Parse

- Parser

```
public QueryData queryCommand() {

    lex.eatKeyword("select");
    ProjectList projs = projectList();
    lex.eatKeyword("from");
    Set<String> tables = idSet();
    Predicate pred = new Predicate();
    if (lex.matchKeyword("where")) {
        lex.eatKeyword("where");
        pred = predicate();
    }
```

QueryData

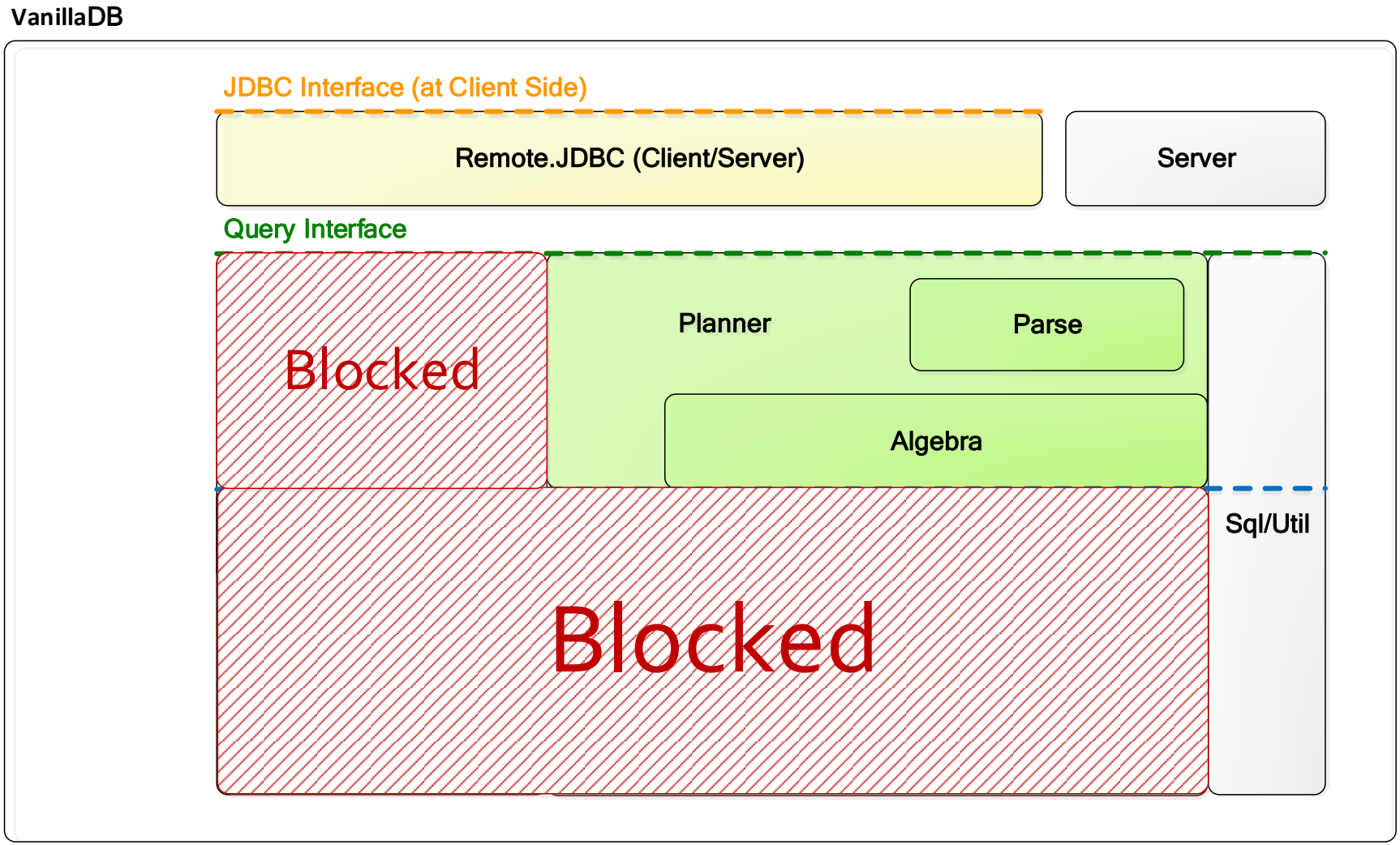


Parse

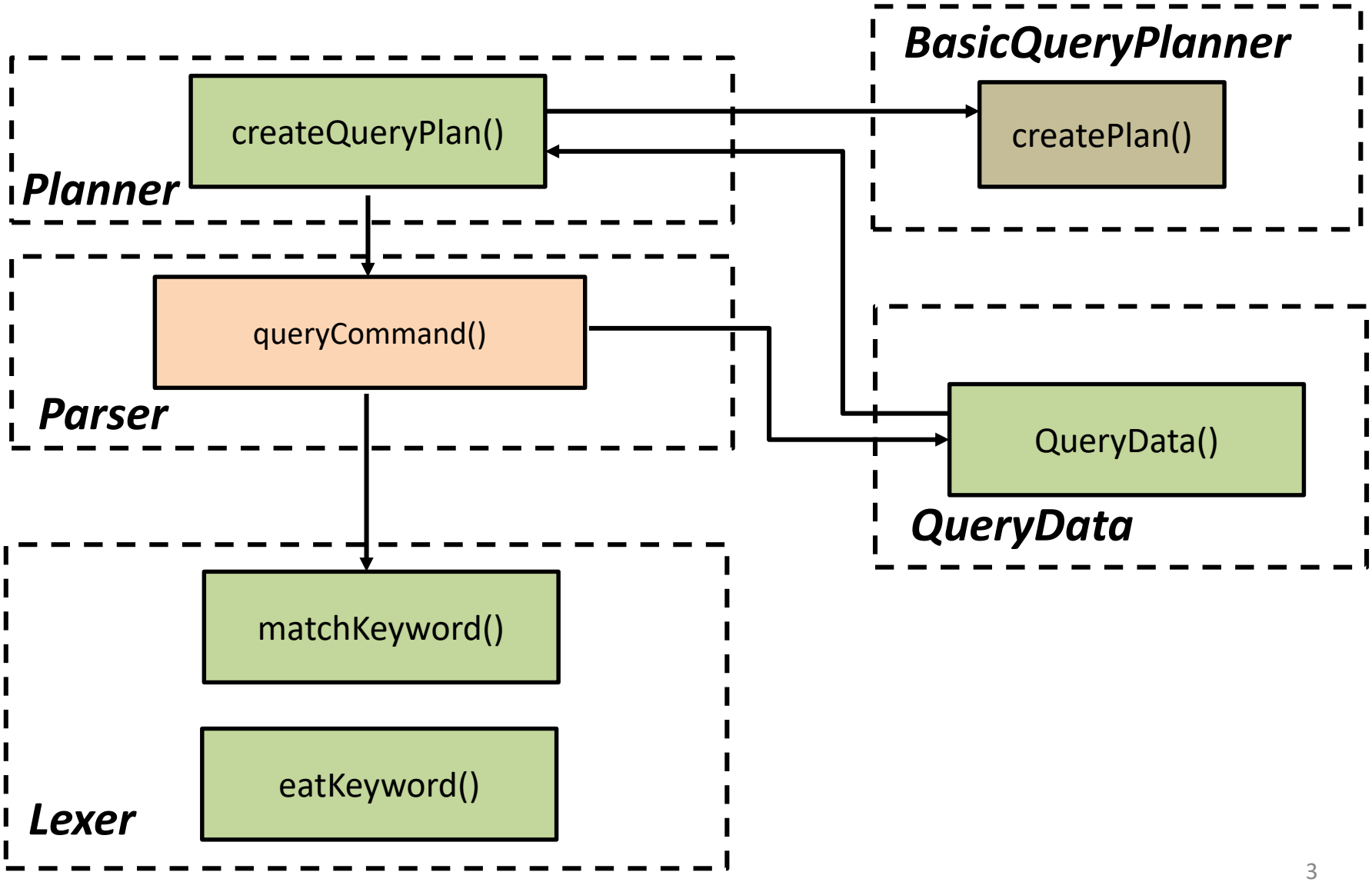
- Parser
  - Parser returns SQL data
  - In method “queryCommand()”

```
public QueryData(Set<String> projFields, Set<String> tables, Predicate pred,
        Set<String> groupFields, Set<AggregationFn> aggFn, List<String> sortFields, List<Integer> sortDirs) {

    this.projFields = projFields;
    this.tables = tables;
    this.pred = pred;
    this.groupFields = groupFields;
    this.aggFn = aggFn;
    this.sortFields = sortFields;
    this.sortDirs = sortDirs;
}
```

This Time


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
Planner
Parse
Algebra
Sql/Util
Blocked

Overview


BasicQueryPlanner
createQueryPlan()
createPlan()
Planner
queryCommand()
Parser
QueryData()
QueryData
matchKeyword()
eatKeyword()
Lexer