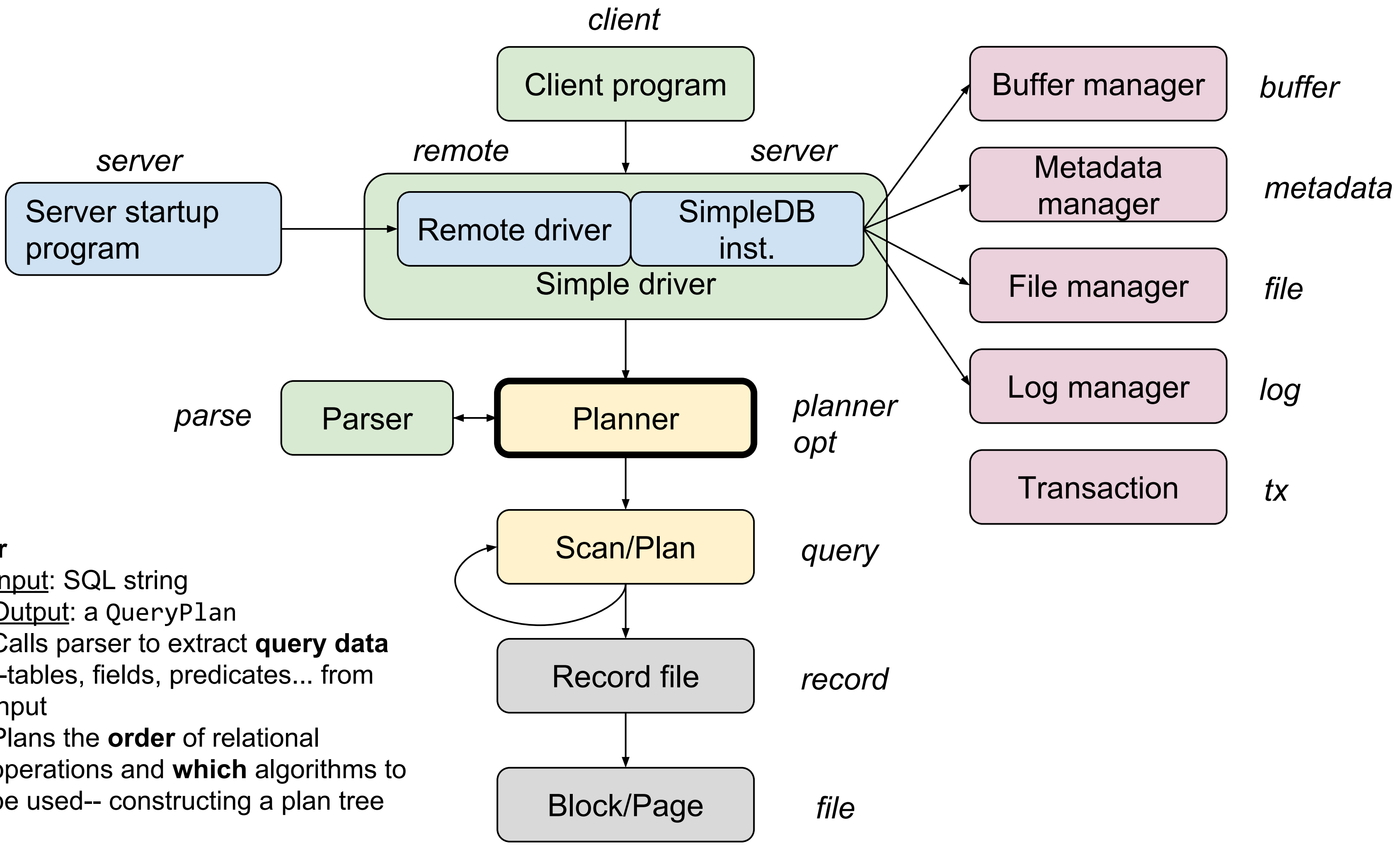
Tutorial on `SimpleDB`

Jia Wang
Oct 11, 2016



Planner

- Input: SQL string
- Output: a `QueryPlan`
- Calls parser to extract **query data** --tables, fields, predicates... from input
- Plans the **order** of relational operations and **which** algorithms to be used-- constructing a plan tree

Running example

- Two tables
  - Students table: (SID, SName, MajorId, Age)
  - Dept table: (DID, DName)
- Query
  - ***SELECT*** SName, DName

    FROM Students, Departments

    WHERE MajorId = DID *AND* Age > 20;

SID	**SName**	**MajorId**	**Age**
1	Alice	10	18
2	Bob	20	20
3	Carl	30	22

DID	**DName**
10	ECE
20	Math
30	CS

Starting server

- Start-up program creates **SimpleDB instance**
  - See: `simpledb/server/Startup.java`
- Creates a (remote) **driver** to drive the SimpleDB inst.
  - See: `simpledb/remote`
- The SimpleDB instance includes various **managers**
  - Metadata manager
    - E.g., how many records each table contains
  - File/buffer manager
    - The files/buffers stored by this database
  - …
  - See: `*Mgr.java` in `simpledb/server, simpledb/metadata, simpledb/file, simpledb/buffer`

Client program

- Creates a **driver** that connects to the server
- Creates a **`Statement`** object from the driver
- The statement object will **execute** user query in string format

```
Driver d = new SimpleDriver();
conn = d.connect("jdbc:simpledb://localhost",
null);

Statement stmt = conn.createStatement();

String qry = "select dname, age"
         + "from student, dept "
         + "where DID = MajorId AND age>= 20 ";
ResultSet rs = stmt.executeQuery(qry);

while (rs.next()) {
      String dname = rs.getString("dname");
      int age = rs.getInt("age");
      System.out.println(dname + "\t" + age);
}

rs.close();
```

Parsing

- When the `stmt.executeQuery(q)` is called, a **`Planner`** will be created to *actually* execute the query, and the result is returned as a `ResultSet` object.
  - Meanwhile, a **`Transaction`** object, associated with the query, will also be created.
  - See: `simpledb/remote, simpledb/tx`
- The planner calls **`Parser`** to parse the query string into a `QueryData` instance, which contains the **structured data** of a query
  - The list of *tables* being queried
  - The *predicate* for selection
  - The list of *fields* for projection
  - See: `simpledb/parser`

Query data parsed from query:

SELECT SName, DName
FROM Students, Dept
WHERE MajorId = DID **AND** Age >= 20;

Fields
- SName
- DName

Tables
- Students
- Dept

Predicate
- (MajorId = DID) conj. (Age >= 20)

Terms
- MajorId = DID
- Age > 20

Expressions
- MajorId
- DID
- Age
- 20

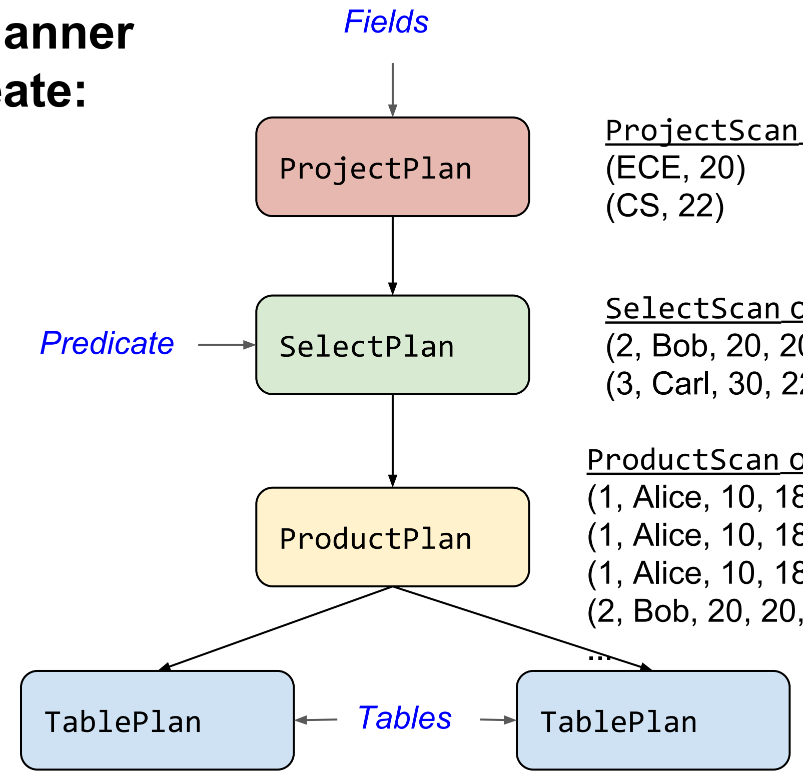
Query processing and optimization

- `Planner` creates a **`QueryPlan`** from `QueryData`, which is a ***tree*** of **`Plan`**'s.
  - Each relational operation can be processed in different ways, each way is represented as a **`Plan`**.
    - E.g, `IndexJoinPlan, HashJoinPlan, SelectPlan, IndexSelectPlan.`
  - Each **non-leaf `Plan`** contains **sub `Plan`'s**, while a **leaf** `Plan` must be a **`TableScan`**.
    - E.g., The `Plan` for a *join* operation contains two sub `Plan`'s, one for scanning the left-hand table and one for the right-hand table.
  - The `Planner` can use **optimization** techniques to create an cost-efficient query plan tree.
  - Folders: simpledb/query/*Plan.java, simpledb/planner, simpledb/opt

Query processing and optimization

- Each `Plan` object is associated with a **`Scan`** object, which actually computes/returns the output of the operation **record-by-record**.
  - **`getInt()`**, **`getString()`**, **`hasField(fld)`**, etc.
  - If parent **`Plan`** p wants to *read output* from its sub-plan **`p’`**, then **`p`** needs to call **`p’.open()`** to get a **`Scan s`**, and get values of a integer field by **`s.getInt()`**, and moves to *next* output by **`s.next()`**
  - See: `simpledb/query/*Scan.java`

A basic query planner will possibly create:



<u>ProjectScan output:</u>
(ECE, 20)
(CS, 22)

<u>SelectScan output:</u>
(2, Bob, 20, 20, 20, ECE)
(3, Carl, 30, 22, 30, CS)

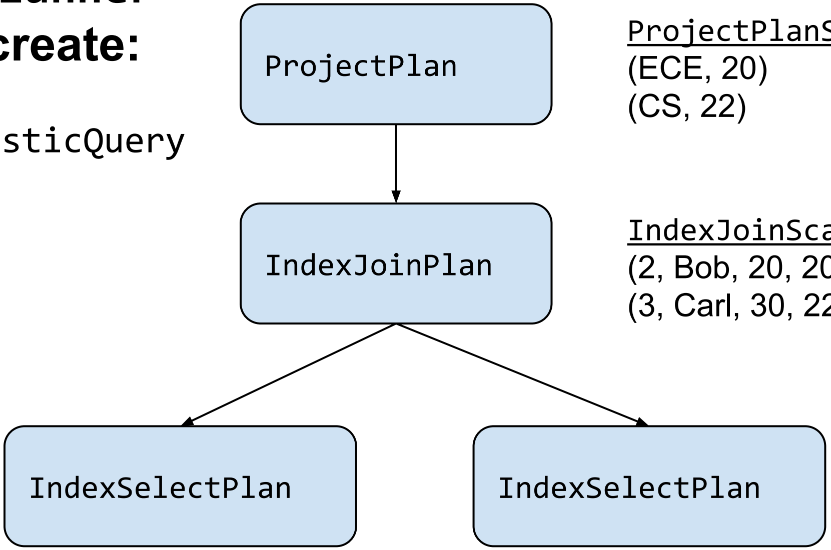
<u>ProductScan output:</u>
(1, Alice, 10, 18, 10, ECE)
(1, Alice, 10, 18, 20, Math)
(1, Alice, 10, 18, 30, CS)
(2, Bob, 20, 20, 10, ECE)
...

<u>TabbleScan output:</u>
(1, Alice, 10, 18)
(2, Bob, 20, 20)
(3, Carl, 30, 22)

<u>TableScan output:</u>
(10, ECE)
(20, Math)
(30, CS)

A better query `Planner` might possibly create:

(e.g., `simpledb/opt/HeuristicQueryPlanner.java`)



`ProjectPlanScan` output:
(ECE, 20)
(CS, 22)

`IndexJoinScan` output:
(2, Bob, 20, 20, ECE)
(3, Carl, 30, 22, CS)

Assuming table "students" has an index on "age"...

Assuming table "dept" has an index on "DID"...

`IndexSelectScan` output:
(2, Bob, 20, 20)
(3, Carl, 30, 22)

`IndexSelectScan` output:
(10, ECE)
(20, Math)
(30, CS)

How a basic query planner creates a query plan:

```
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 = SimpleDB.mdMgr().getViewDef(tblname, tx);
       if (viewdef != null)
          plans.add(SimpleDB.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 on the field names
    p = new ProjectPlan(p, data.fields());
    return p;
 }
```

Record scanning

- Each **leaf** in the query plan tree will be a **`TableScan`** plan, which actually reads **records** from the storage.
    - See `simpledb/query/TableScan.java, simpledb/record`
- The scanning can be either a simple **linear** scan or **index** scan.
    - An index scan will contain a `TableScan` object which returns records specified by the index
    - See `simpledb/index, simpledb/query`
- The records will be stored in disk **blocks**.
    - Each disk block is referred to by a **`Block`** instance, and the actually content of a block is stored by a **`Page`** instance. So `Block` and `Page` instances are always used together.
    - See `simpledb/file`

Using indexes

- Create an index *just like* creating a table
  - Specifying the table and the fields.
  - Managed by an `IndexManager` (similar to `TableManager`).
- When inserting records into the table, **also** *insert into the index*.
  - See `simpledb/index/planner/IndexUpdatePlanner.java`
- When using index for table scanning
  - Both a `TableScan` and the `Index` will be used
  - The `Index` will returns the **pointer** to a record, while the `TableScan` will move to that pointer to return the record **content**.
  - See `simpledb/index/query/*.java`

Example project

- To add a **new join algorithm**, which parts of the code will be touched?
    - **`Plan`**: create a plan class for the new join operation using your algorithm
        - It *estimates the **cost*** of join using the new algorithm
        - It *creates the* **`Scan`** *object* for your new algorithm.
        - It *determines the **schema*** of the output of this operation.
        - See: `simpledb/query/*Plan.java`
    - **`Scan`**: a new scan class corresponding to the new plan, which *actually* computes/returns the output of the join using your algorithm *one-by-one*.
        - The new plan will know how to create this new scan
        - See: `simpledb/query/*Scan.java`

Example project

- **`Planner`**: now the planner *knows* that your new join operation exists, and would consider *using* it when proper when it constructs the plan tree (e.g., based on cost estimation).
    - Make your own `Planner` or modify existing planners.
    - See: `simpledb/opt/*Planner.java, simpledb/planner/*Planner.java.`
    - Change simpledb/server/SimpleDB.java to switch to another planner.
- **`Parser`**: modify `Parser` if the SQL needs be extended.
    - See: `simpledb/parser`.