DSCI-551

Project Presentation

Group 35

Yash Thakur

Demo: Data

Data Set 1: Sales

- **sales:** invoice_no str, customer_id str, category str, quantity int, price float, invoice_date str, shopping_mall str
- **customer:** customer_id str, gender str, age int, payment_method str

LINK https://www.kaggle.com/datasets/dataceo/sales-and-customer-data

Data Set 2: College

- **diversity_school:** name str, total_enrollment int, state str, category str, enrollment int
- **historical_tuition:** type str, year str, tuition_type str, tuition_cost int
- **salary_potential:** rank int, name str, state_name str, early_career_pay int, mid_career_pay int, make_world_better_percent float, stem_percent float
- **tuition_cost:** name str, state str, state_code str, type str, degree_length str, room_and_board int, in_state_tuition int, in_state_total int, **out_of_state_tuition int,** out_of_state_total int
- **tuition_income:** name str, state str, total_price float, year int, campus str, net_cost float, income_lvl str

LINK https://www.kaggle.com/datasets/jessemostipak/college-tuition-diversity-and-pay

Note: Engine works with almost any relational data, given input rows meet constraints mentioned in “Query Parsing and Constraints” slides.
Following dataset have been tested as well: https://www.kaggle.com/datasets/lokeshparab/amazon-products-dataset

Demo: Operations

Data Set 1: `Sales`

- project
- join + filter
- join + group by + aggregate + sort
- group by + aggregate
- update
- delete

Data Set 2: `College`

- Insert
- global aggregate
- join (3 tables) + filter + sort

Data Set 3: `Custom`

- Creation
- Referential Integrity
  - Insertion
  - Update Cascade
  - Delete Cascade

Query Structure

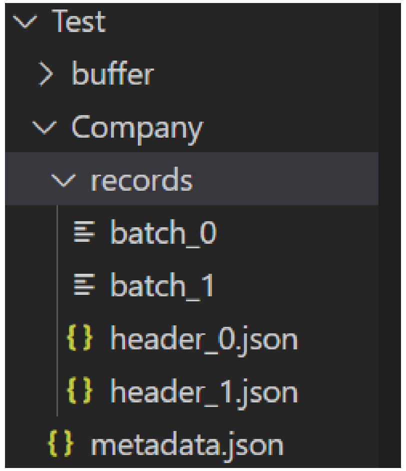
Keyword	Operation	Usage
@database	Create/Select database	*@database => <db_name>*
@batchsize	Set batch size for database	*@batchsize => <numeric_batch_size>*
@create	Create table	*@create => <table_name>*
@columns	Define table columns	*@columns => <column1_name> <data_type> <data_size>, <column2_name> <data_type> <data_size>*
@constraint	Foreign key constraints	*@constraint => [table1.col1, table1.col2 references table2.col1, table2.col2] [table1.col3 references table3.col3]*
@populate	Insert into table	*@populate => <table_name>*
@update	Update table rows	*@update => <table1_name>*
@delete	Delete table rows	*@delete => <table1_name>*
@values	Define values to insert/update	• Update: *@values => <col1> = "new value“* • Insert: *@values => [row1_value1 | row1_value2] [row2_value1 | row2_value2];*
@use	Select table for DML ops	*@use => <table_name>*
@project	Projection(supports aggregation)	• Project: *@project => <table_name>.<col1_name>, <table_name>.<col2_name>* • Project and aggregate: *@project => COUNT:<table1_name>.<col2>, AVG:<table2_name>.<col3>*
@filter	Filtering rows	*@filter => (((<table1_name>.<col1_name> == 600.00) AND (<table2_name>.<col2_name> == “xxx xxx”)) OR (<table1_name>.<col2_name> == 100))*
@combine	Join tables	*@combine => <table1_name>.<col1> = <table2_name>.<col2>*
@categorize	Group By	*@categorize => <table1_name>.<col1>, <table2_name>.<col2>*
@sort	Sorting	• *@sort => <table1_name>.<col1> DESC, <table2_name>.<col2> ASC;* • *@sort => AVG(<table2_name>.<col3>) DESC , COUNT(<table1_name>.<col2>)*

Query Parsing and Constraints

- @ is used as delimiter between various operations
- [] are used as delimiters between row values while inserting
- | is used as delimiter between column values while inserting
- Inside filter expressions, parenthesis and text must have space b/w them
- Inside filter expression strings must be surrounded by double quotes
- Spaces are handled by trimming preceding and trailing spaces in input values while inserting data
- Queries end with semicolon
- Characters @, [,], => are not supported to be passed as input data, as they are special characters used by engine
- Categorization must be followed by a projection with aggregate expression
- Null values are not supported
- Supported data types: str, float, int
- Supported aggregations : MIN, MAX, COUNT, AVG, SUM

Data Storage Model: Directory Structure

- For each database, the engine will create a directory *<db_name>/*
- Inside *<db_name>/,*
  - Engine creates directory *buffer/* to store intermediate results of query operations
  - for each table, engine will create a directory *<table_name>/*
- For each table directory, engine have the following files and directories:
  - *metadata.json*
  - *records/*
- Inside *records/* engine will create batch files to store actual data. Data for batch *i* is stored using 2 files*:*
  - *batch_i*
  - *header_i.json*

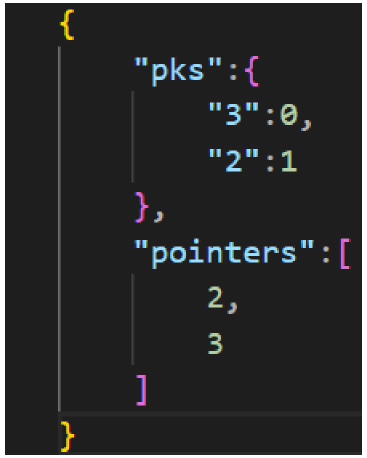


```
Test
  buffer
  Company
    records
      batch_0
      batch_1
      header_0.json
      header_1.json
    metadata.json
```

Directory structure of “Test” database with “Company” table

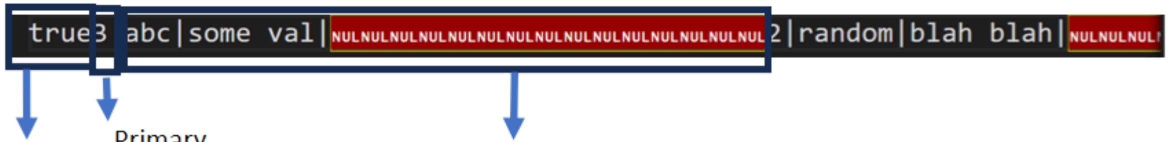
Data Storage Model: File Structure

header_i.json



```
{
    "pks":{
        "3":0,
        "2":1
    },
    "pointers":[
        2,
        3
    ]
}
```

batch_i (binary)



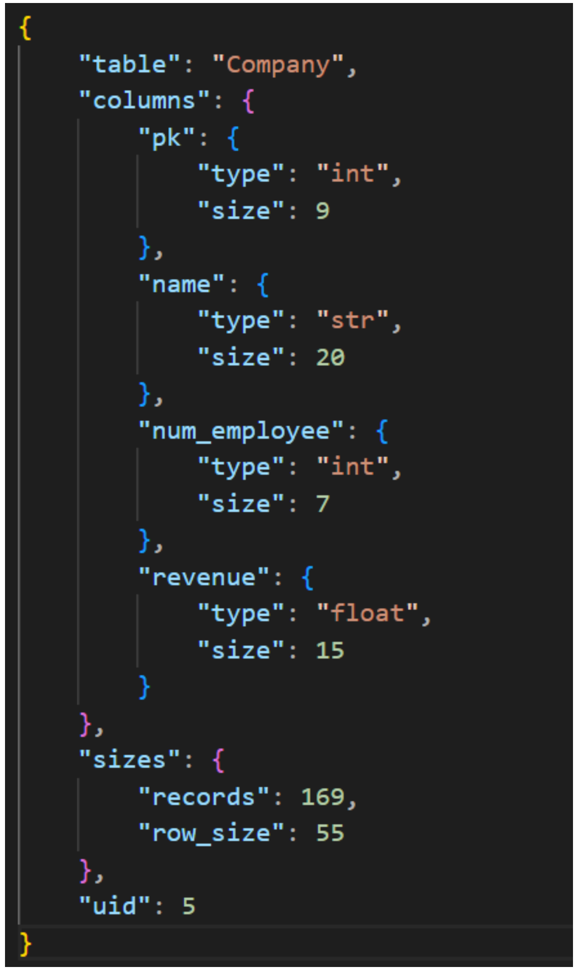
```
true3 abc|some val|NULNULNULNULNULNULNULNULNULNULNULNULNULNULNUL2|random|blah blah|NULNULNUL
```

IsFull flag

Primary Key of row at this offset

Row data with fixed size. Null padded

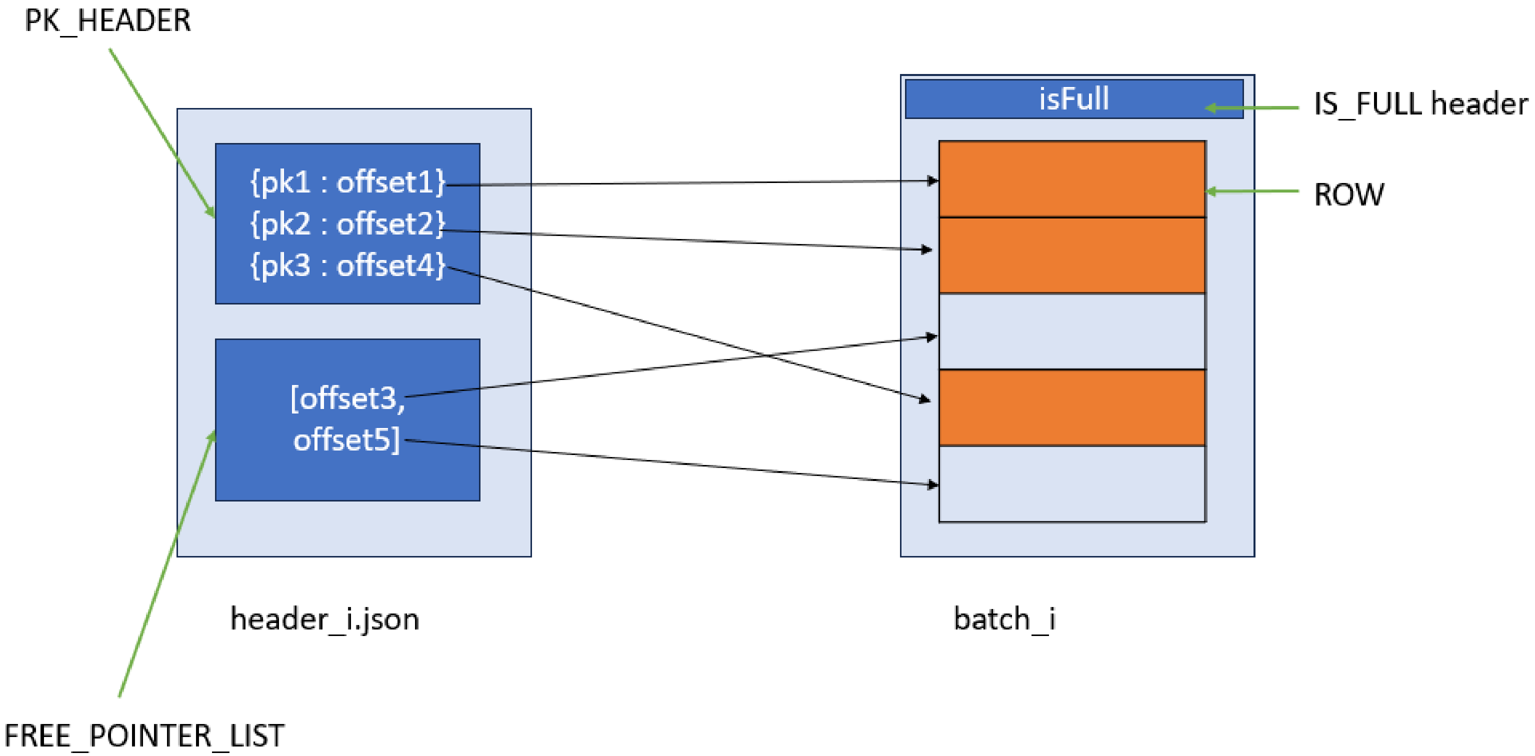
metadata.json



```
{
    "table": "Company",
    "columns": {
        "pk": {
            "type": "int",
            "size": 9
        },
        "name": {
            "type": "str",
            "size": 20
        },
        "num_employee": {
            "type": "int",
            "size": 7
        },
        "revenue": {
            "type": "float",
            "size": 15
        }
    },
    "sizes": {
        "records": 169,
        "row_size": 55
    },
    "uid": 5
}
```

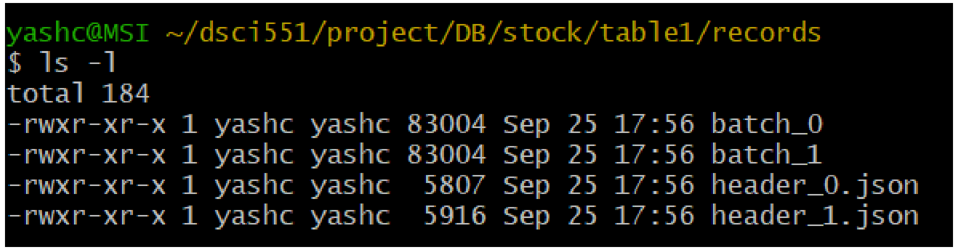
Data Storage Model: Storage Mechanism

- Each Batch file is limited to storing the “BATCH_SIZE” number of records that can be configured at time of creating the database
- The corresponding header_i.json file for a batch_i.json file has the list of pointers (FREE_POINTER_LIST) that point to the offset location of slots in the batch file which are available for storing row data (Figure 1).
- The primary key and offset location of each assigned slot of a batch file is stored in the primary key header (PK_HEADER) of the header file.



Data Storage Model: Design Decisions

- Storing offset locations and list of free pointers in header files allows engine to perform CRUD operations on data by only loading header file and not the complete batch file
- This improves performance as in case of real-world data with a considerably large batch size, batch file size would be considerably larger than header file
- The data structure used for the primary key header is a dictionary, and for pointers is a list. Dictionary would allow for O(1) lookup time after loading data
- BATCH_SIZE can be set while creating database and allows for tuning scalability and performance of engine



```
yashc@MSI ~/dsci551/project/DB/stock/table1/records
$ ls -l
total 184
-rwxr-xr-x 1 yashc yashc 83004 Sep 25 17:56 batch_0
-rwxr-xr-x 1 yashc yashc 83004 Sep 25 17:56 batch_1
-rwxr-xr-x 1 yashc yashc  5807 Sep 25 17:56 header_0.json
-rwxr-xr-x 1 yashc yashc  5916 Sep 25 17:56 header_1.json
```

header file size (~5KB) VS batch file size (>80KB)

Insert

To insert a row in a table, the engine perform following steps:

- Determines the unique primary key for the row using “uid” field of metadata.json
- Engine iterates over each batch file in **/<db>/<table_name>/records**, and checks if they are full using *isFull* flag (first 4 bytes only) for each file
  - If no file is found it creates a batch file and corresponding header file. Both files are initialized accordingly (free pointer list and isFull flag) and returned.
  - If a file exists, the engine returns the batch file and corresponding header file
- Next, engine pops the next free-slot offset from the pointer list in the header file, store the row at that offset and update the primary key header with the primary key of row and offset location
- If the file becomes full, engine updates the “isFull” flag
- Then increase the “uid” of metadata.json under **/<db>/<table_name>**/

Joins: Syntax Resolution

- Engine may not be able to join tables in the sequence user provides
- For example, if join expression is

  table_A.col1 = table_B.pk, table_C.pk = table_D.col2, table_A.col3 = table_D.col4

  engine can't join tables in the sequence: A → B → C → D, because C doesn't have a joining condition on A or B
- To address this problem system **topologically sorts** the tables, by first creating a graph with table names as nodes and 2 nodes have an edge if their corresponding tables have a joining condition between them
- In above case, one possible sorted sequence is: A → D → C → B

Joins: Nested Block Approach

Engine performs joins using nested block join approach
Example given the join - *table_A.col1 = table_B.pk, table_C.pk = table_B.col2* engine does:

```
For batchA_n in table_A/batches:
   batch1 = load_batch(batchA_n)
   For batchB_n in table_B/batches:
      batch2 = join(batch1, load_batch(batchB_n), table_A.col1=table_B.pk)
      For batchC_n in table_C/batches:
           batch3 = join(batch2, load_batch(batchC_n), table_C.pk=table_B.col2)
           /* perform filter, projection, sorting, aggregation, etc. on joined batch */
```

Once engine have the joined batch, engine can have one of the following sequence of operations:

→ filter → project → sort
→ filter → categorize → project(aggregate) → sort
→ filter → update
→ filter → delete

Where blue text indicates optional operations

Note

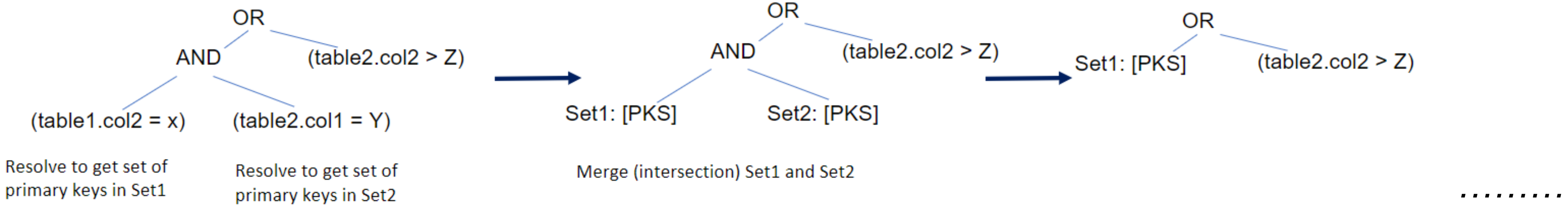
- For single table(zero joins) engine will only have one for loop
- Idea is described using an iterative approach, but implementation is done recursively

Filter

- Filter is performed once engine has created the joined batch
- To resolve a filter expression, engine use stack operations to process individual sub expressions in a post order tree traversal manner, producing a set of primary keys at the end of each processing step on a node
- A leaf expression is one which doesn't have any parenthesis inside it. To resolve a leaf expression, engine check which records/rows of the batch satisfy the expression and create a set of PKs/Indices that does satisfy
- If expression has an AND, engine takes intersection of the 2 sets of indices, if expression has OR, engine takes union
- Final set of indices will give us rows that satisfy the complete filter expression
- For example, the expression

  "(((table1.col2 = x) AND (table2.col1 = Y)) OR (table2.col2 > Z))"

  can have the following initial processing steps:



Note: At any point engine only have one or more set of numbers(indexes), where each set is limited to size BATCH_SIZE set by user

Update

Using ‘filter’, engine narrows down to Primary Keys (PK) of rows that we want to update
To perform update:

- Engine iterates over header files for the table we want to update
- If a filtered row’s PK is present in header file, overwrite old row in corresponding batch file by using offset location stored against the PK.
- To overwrite in batch file:
    - Erase – set the complete row as *‘\0’*
    - Write – write the updated row at offset location

Delete

Using ‘filter’, engine narrows down to Primary Keys (PK) of rows that we want to delete

To perform delete:

- Engine iterates over header files for the table we want to update
- If a filtered row’s PK is present in header file, delete old row in corresponding batch file by using offset location stored against the PK.
- To delete in batch file:
    - Erase – set the complete row as *‘\0’*
    - Remove PK entry in header file, and mark offset available
- Set the *isFull* flag of the header file to *False*

Categorize

Categorize is performed in 2 steps:

Partition –

- For each row, *group-by* column values are hashed to get file-name into which row information is written. Thus, all rows in same category/group are written into same file
- Files(intermediate results) are stored under */buffer/categorize* directory
- Each file has following information:
    - Columns – group-by columns
    - Count – total rows in the group
    - Sum – sum of values for each column on which aggregation is required

Aggregation –

- Using Count and Sum values for each group, system recreates batches of size BATCH_SIZE with AVG, COUNT and SUM values for each group

Sort

Sort is performed in 2 steps:

- Batches are sorted and written onto disk as runs in */buffer/sort/* directory
- Merge runs:
    - Iterate over all runs to find row with smallest/largest value. This is achieved by only checking the first row of each run
    - Output the row, and delete it from corresponding run
    - Repeat above steps until all runs are empty
    - Note: At any point in time engine only has 1 batch in memory

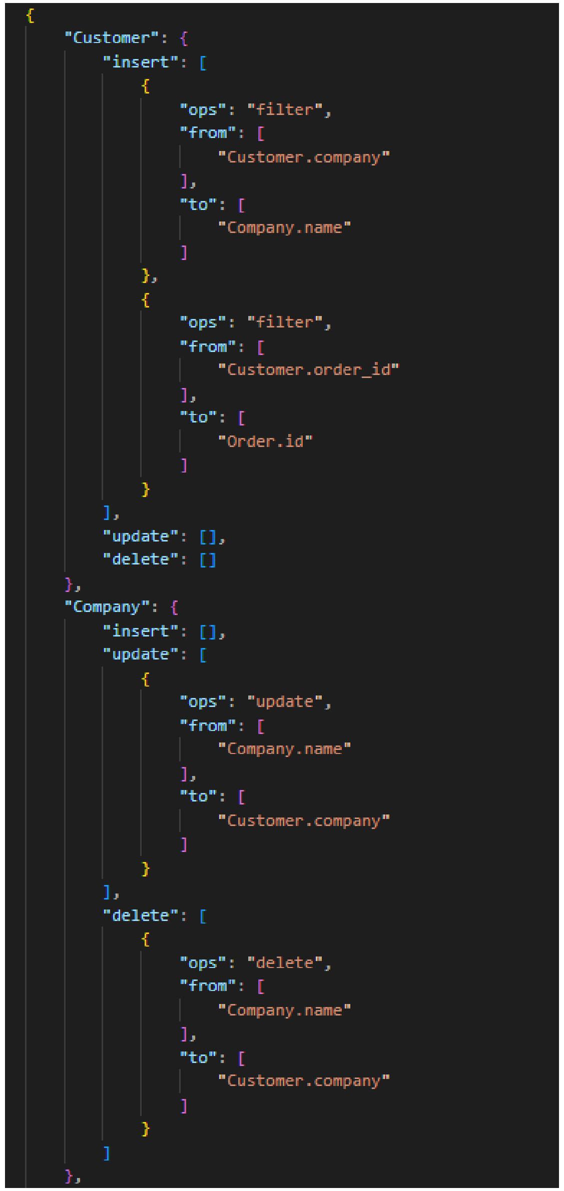
Referential Integrity Constraint: Implementation

- Engine allows a table to reference one or more tables with foreign key (comprising of one or more attributes)
- When a constraint is defined, details about constraints are stored under table/ directory inside *constraints.json* file
- It stores metadata, checks and cascade operations that engine needs to perform on tables to meet referential integrity constraints
- For instance, for following constraint on Customer table

```
 @constraint => [ Customer.company
     references Company.name] [
Customer.order_id references Order.id];
```

    - Any inserts into Customer needs to check for the corresponding entry in Company and Order table
    - Any updates or deletes to Order or Company table needs to reflect in corresponding rows of Customer table (Cascade)

constraints.json captures these details, and engine refers to this file to create intermediate queries for meeting referential integrity



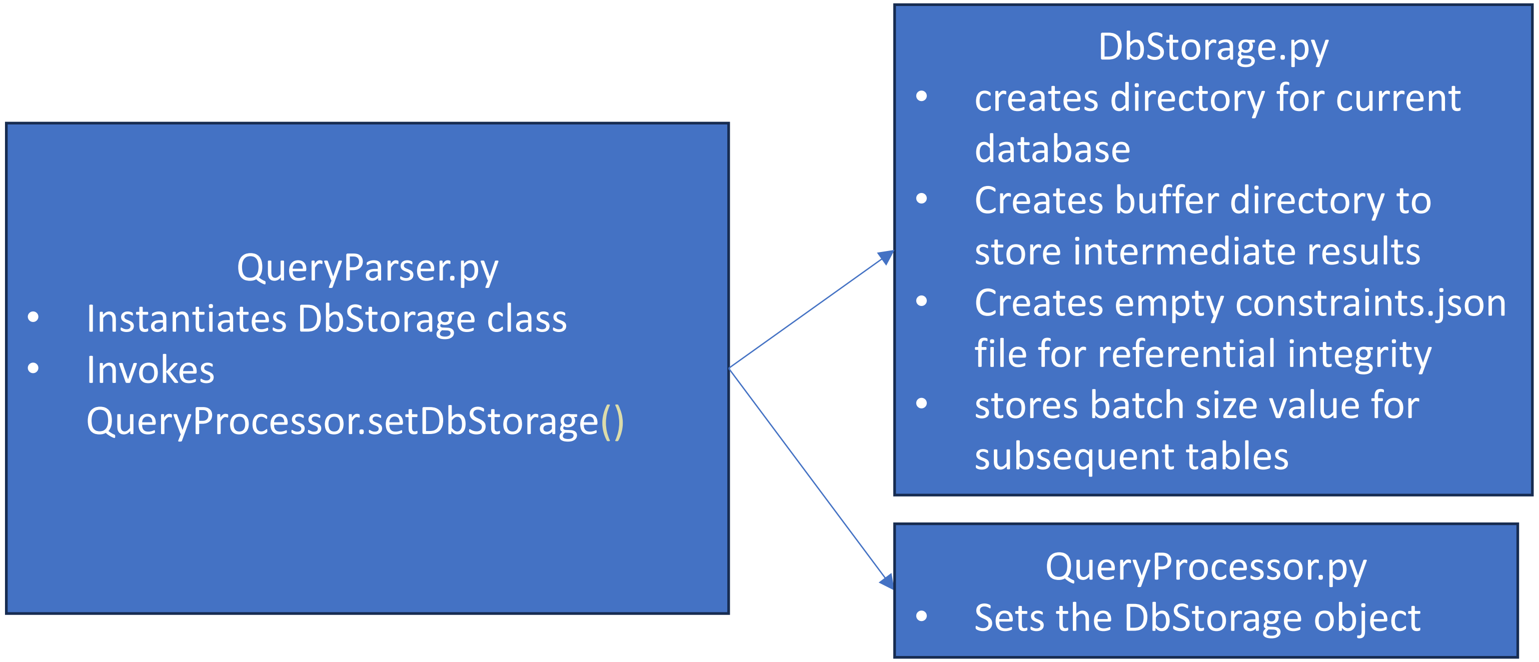
```
{
    "Customer": {
        "insert": [
            {
                "ops": "filter",
                "from": [
                    "Customer.company"
                ],
                "to": [
                    "Company.name"
                ]
            },
            {
                "ops": "filter",
                "from": [
                    "Customer.order_id"
                ],
                "to": [
                    "Order.id"
                ]
            }
        ],
        "update": [],
        "delete": []
    },
    "Company": {
        "insert": [],
        "update": [
            {
                "ops": "update",
                "from": [
                    "Company.name"
                ],
                "to": [
                    "Customer.company"
                ]
            }
        ],
        "delete": [
            {
                "ops": "delete",
                "from": [
                    "Company.name"
                ],
                "to": [
                    "Customer.company"
                ]
            }
        ]
    },
```

Picture on left shows:

- For an insert operation on Customer table:
    - Engine first filters/checks Company.name field, and Order.id field
- For update or delete on Company table:
    - Engine updates/deletes corresponding entry in Customer table found using value of *Company.name*

Engine uses this info while carrying out any delete/insert/update provided by user. Then it creates its own intermediate queries.

Thank You



QueryParser.py

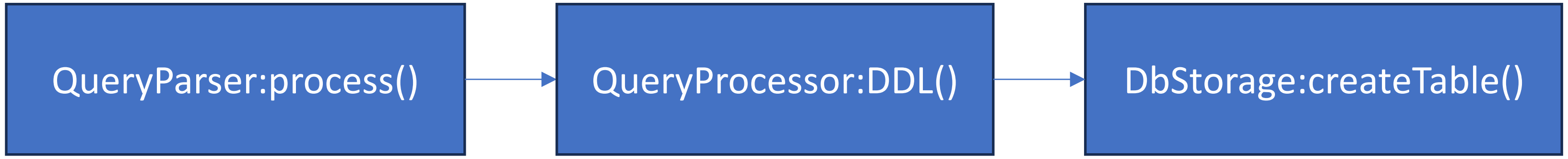
- Instantiates DbStorage class
- Invokes QueryProcessor.setDbStorage()

DbStorage.py

- creates directory for current database
- Creates buffer directory to store intermediate results
- Creates empty constraints.json file for referential integrity
- stores batch size value for subsequent tables

QueryProcessor.py

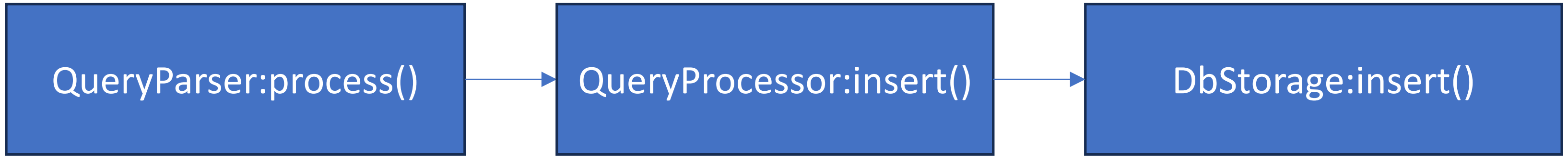
- Sets the DbStorage object



QueryParser:process()

QueryProcessor:DDL()

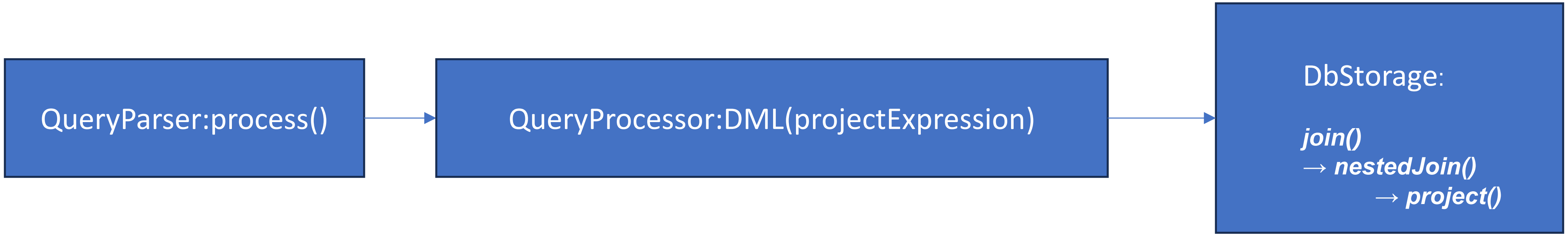
DbStorage:createTable()



QueryParser:process()

QueryProcessor:insert()

DbStorage:insert()

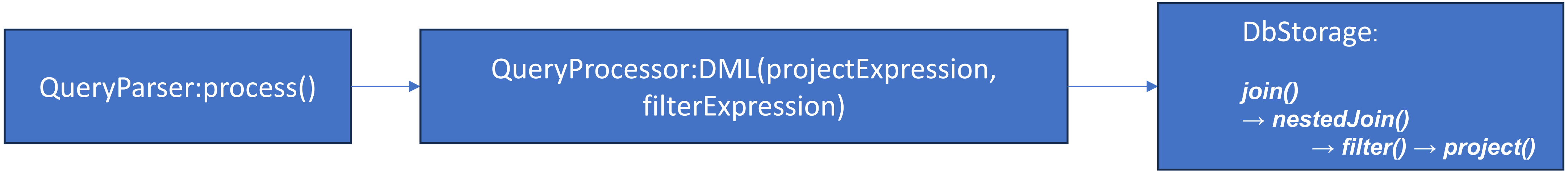


QueryParser:process()

QueryProcessor:DML(projectExpression)

DbStorage:

join()
→ nestedJoin()
→ project()

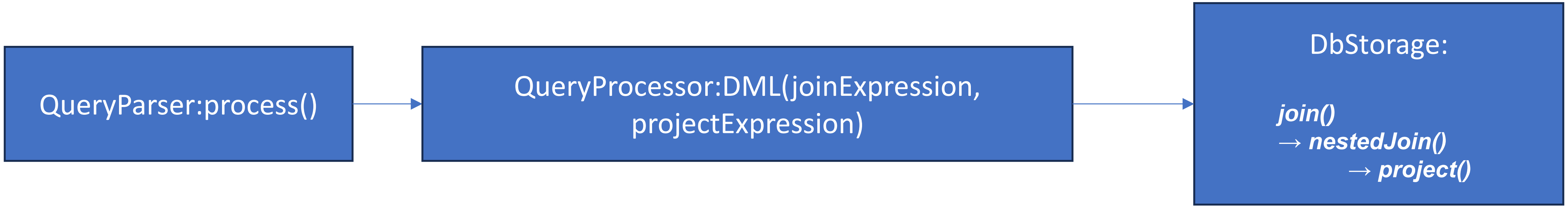


QueryParser:process()

QueryProcessor:DML(projectExpression, filterExpression)

DbStorage:

join()
→ ***nestedJoin()***
→ ***filter()*** → ***project()***

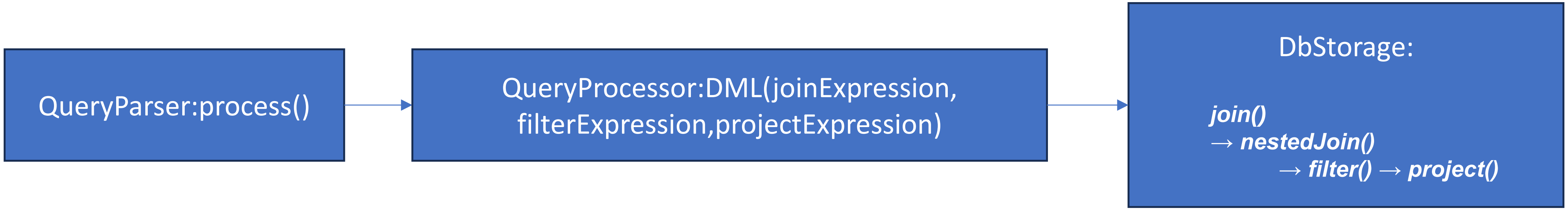


QueryParser:process()

QueryProcessor:DML(joinExpression, projectExpression)

DbStorage:

join()
→ ***nestedJoin()***
→ ***project()***

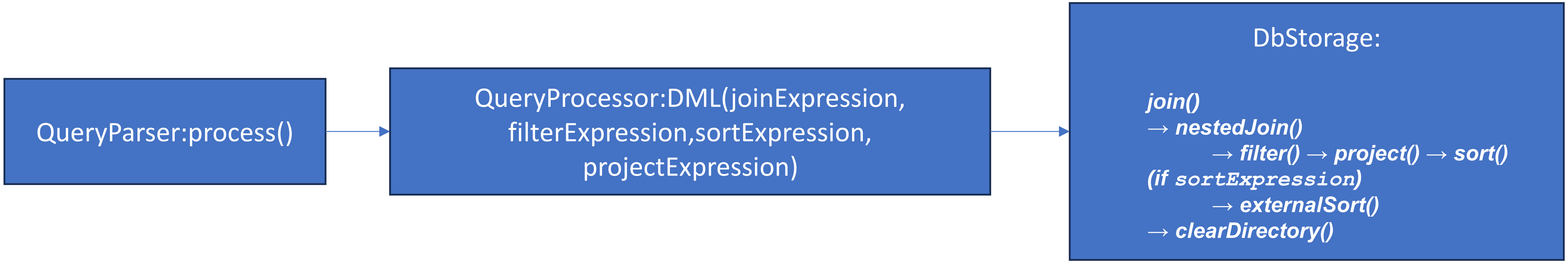


QueryParser:process()

QueryProcessor:DML(joinExpression, filterExpression,projectExpression)

DbStorage:

join()
→ nestedJoin()
→ filter() → project()

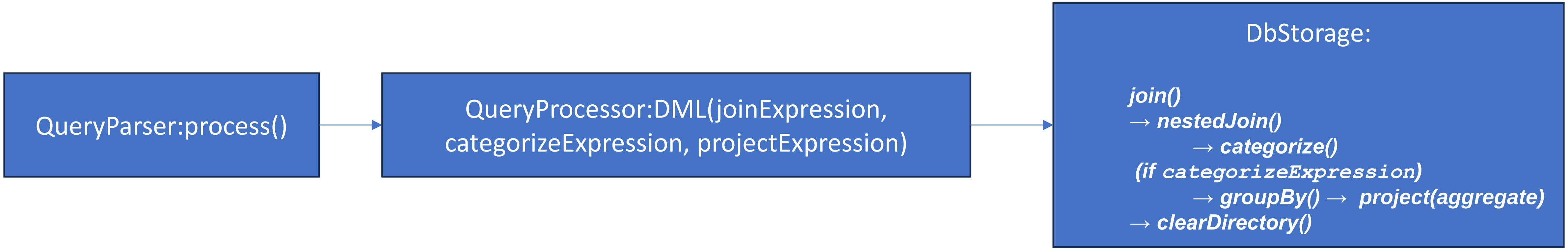


QueryParser:process()

QueryProcessor:DML(joinExpression, filterExpression,sortExpression, projectExpression)

DbStorage:

join()
→ nestedJoin()
→ filter() → project() → sort()
(if `sortExpression`)
→ externalSort()
→ clearDirectory()

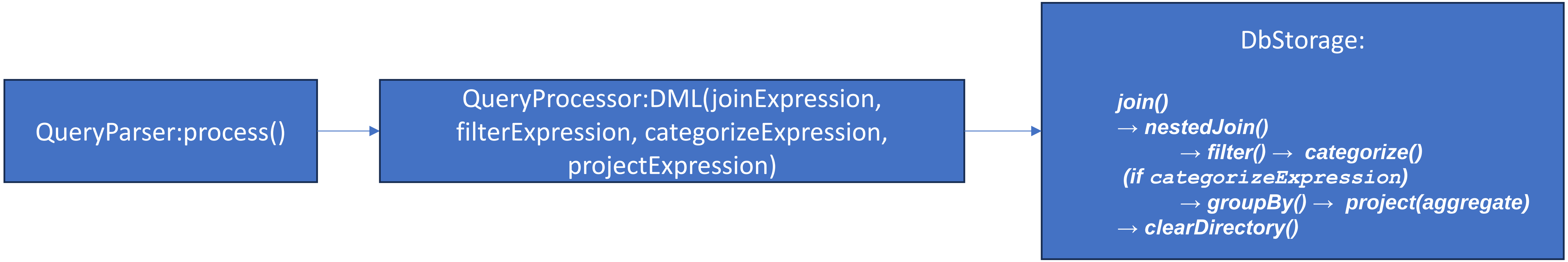


QueryParser:process()

QueryProcessor:DML(joinExpression, categorizeExpression, projectExpression)

DbStorage:

join()
→ nestedJoin()
→ categorize()
(if `categorizeExpression`)
→ groupBy() → project(aggregate)
→ clearDirectory()

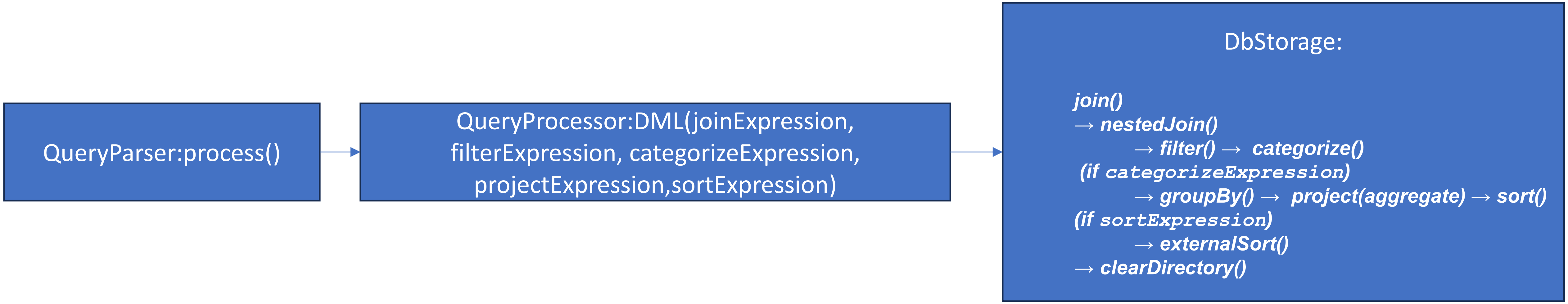


QueryParser:process()

QueryProcessor:DML(joinExpression, filterExpression, categorizeExpression, projectExpression)

DbStorage:

join()
→ nestedJoin()
→ filter() → categorize()
(if `categorizeExpression`)
→ groupBy() → project(aggregate)
→ clearDirectory()

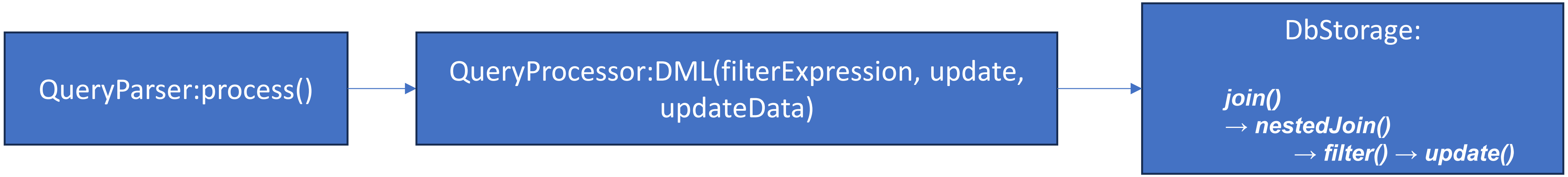


QueryParser:process()

QueryProcessor:DML(joinExpression, filterExpression, categorizeExpression, projectExpression,sortExpression)

DbStorage:

join()
→ nestedJoin()
→ filter() → categorize()
(if `categorizeExpression`)
→ groupBy() → project(aggregate) → sort()
(if `sortExpression`)
→ externalSort()
→ clearDirectory()

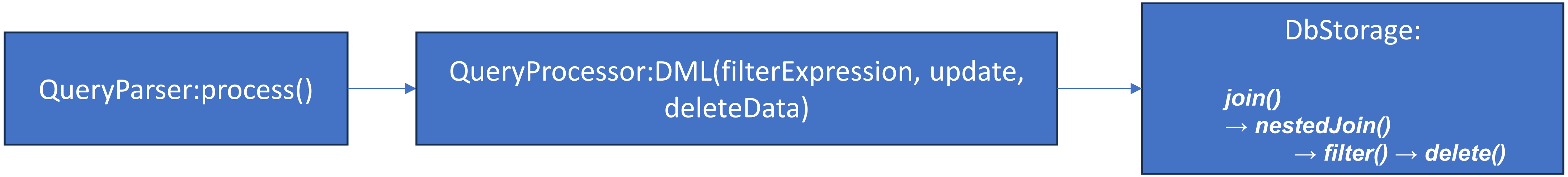


QueryParser:process()

QueryProcessor:DML(filterExpression, update, updateData)

DbStorage:

join()
→ nestedJoin()
→ filter() → update()

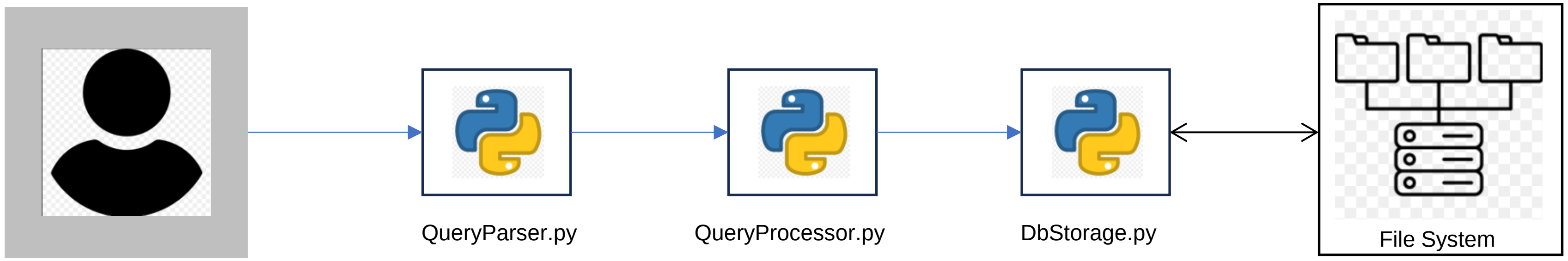


QueryParser:process()

QueryProcessor:DML(filterExpression, update, deleteData)

DbStorage:

join()
→ nestedJoin()
→ filter() → delete()



QueryParser.py

QueryProcessor.py

DbStorage.py

File System