Project Final Report

Check our database on Github!

Introduction

The AvA Database project integrates SQL and NoSQL databases, offering versatile data management capabilities. It supports various field types and customizable settings, including default values and file extensions. The system can import data from CSV and JSON files, automatically adapting to the appropriate database format. A custom logging module enhances monitoring and debugging. Detailed usage guidelines and setup instructions ensure easy operation. The project includes constants and definitions to standardize operations and parameters across the system.

A Flask web server handles database operations like insert, update, and delete(and more advanced function). The project context manages configurations and database connections. Advanced field processing techniques support complex queries and database operations. Table management includes creation, deletion, and metadata handling, ensuring data integrity. Metadata handling is critical for maintaining data structure in both SQL and NoSQL systems. The table manipulation module implements advanced functionalities like sorting, grouping, and joining, providing robust data processing capabilities. This project represents a comprehensive solution for efficient and complex data management across diverse database systems.

Planned Implementation

SQL Database

Data Model

- Each row in a table represents an entry.
- Tables are organized in a fixed schema that outlines the type and constraints of each column.
- Each entry will be assigned a PK (Primary Key) per table.
- Foreign Keys will be used to represent relationships between tables.

Storage

Data will be stored in tables, Each table will be a separate file or set of files managed by the SQL database engine. We don't manage these files directly.

Programming Language

SQL-based database will be implemented by Java(after our discussion, now we have a unified language for both SQL and NOSQL), since the language has better performance.

Supported Operations

SQL naturally supports JOIN operations, which allows for combining rows from two or more tables based on a related column.

The other operations mentioned

- projection
- filtering
- grouping
- aggregation
- ordering
- insertion
- updation
- deletion are also fully supported in SQL databases.

"JOIN" will be supported in SQL database, since there are multiple tables can be operated.

Big Data

- SQL database is able to store large volumes of data efficiently, but after the amount of data reaches to million, optimizations such as indexing, caching, and partitioning can be used(). (we now use chunk to store large volumes of data)
- SQL database allows transactions, locking, and other concurrency controls, to manage multiple simultaneous operations.(no more concurrency is considered)

Other Details

SQL database will have a local interface to issue commands and show query results.

NoSQL Database

Data Model

- Each entry will be presented as an object. Each object will be stored as a JSON document.
- Objects in the same class will be stored together as a collection.
- Every entry will be assigned with a unique ID.
- The entry will hold the IDs of other entries as JSON fields.

Storage

We suppose every single entry can be loaded into memory. Each JSON document will be stored in a distinct file. Objects in the same collection will be stored under the same folder.

Programming Language

NoSQL database will be implemented with Python.

Supported Operations

"JOIN" is not supported by our NoSQL database (No tables to join for NoSQL!)

Other operations are supported, including:

- projection
- filtering
- grouping
- aggregation
- ordering
- Insertion, updation, and deletion are all supported.

Big Data

Only one JSON document will be loaded into memory for each thread. The JSON object will be cleared or replaced before loading the next JSON document. Only IDs of different objects will be kept in memory at the same time.

Other Details

NoSQL Database will be served as an HTTP server and handle HTTP requests only.

CLI

It's designed as a user interface to communicate with SQL or NoSQL servers. Queries will be sent by HTTP protocol, while the query results will be parsed and presented in standard output.

The CLI will be implemented with Python.

Architecture Design

Overview

The project architecture is centered around the integration of SQL and NoSQL databases, providing a unified interface for data manipulation and query processing. It leverages Python for backend development, Flask for API handling, and a custom query language designed for optimized data handling in limited memory environments.

Key Components

1. **Database Configuration (DBConfig):** Manages database settings, including type (SQL or NoSQL), port, directories for tables and metadata, supported field types, and file extensions.
2. **Data Import (import_data.py):** Handles data import from CSV and JSON files, auto-detecting column types for CSV files and loading JSON files for NoSQL databases.
3. **Logging (logger.py):** Provides a logging mechanism for tracking and debugging. Custom loggers for different components ensure detailed logging.
4. **Server Management (app/server.py):** Uses Flask to handle API requests, enabling operations like file retrieval, data insertion, updating, deletion, table creation, and dropping.
5. **Context Management (app/common/context/context.py):** Manages the application's runtime context, including configurations, database instances, table managers, and query engines.
6. **Table Management (app/common/table/table_manager.py):** Manages tables, including creation, deletion, and metadata handling. Ensures data consistency and manages temporary tables for complex operations.
7. **Command Line Interface (cli/cli.py):**

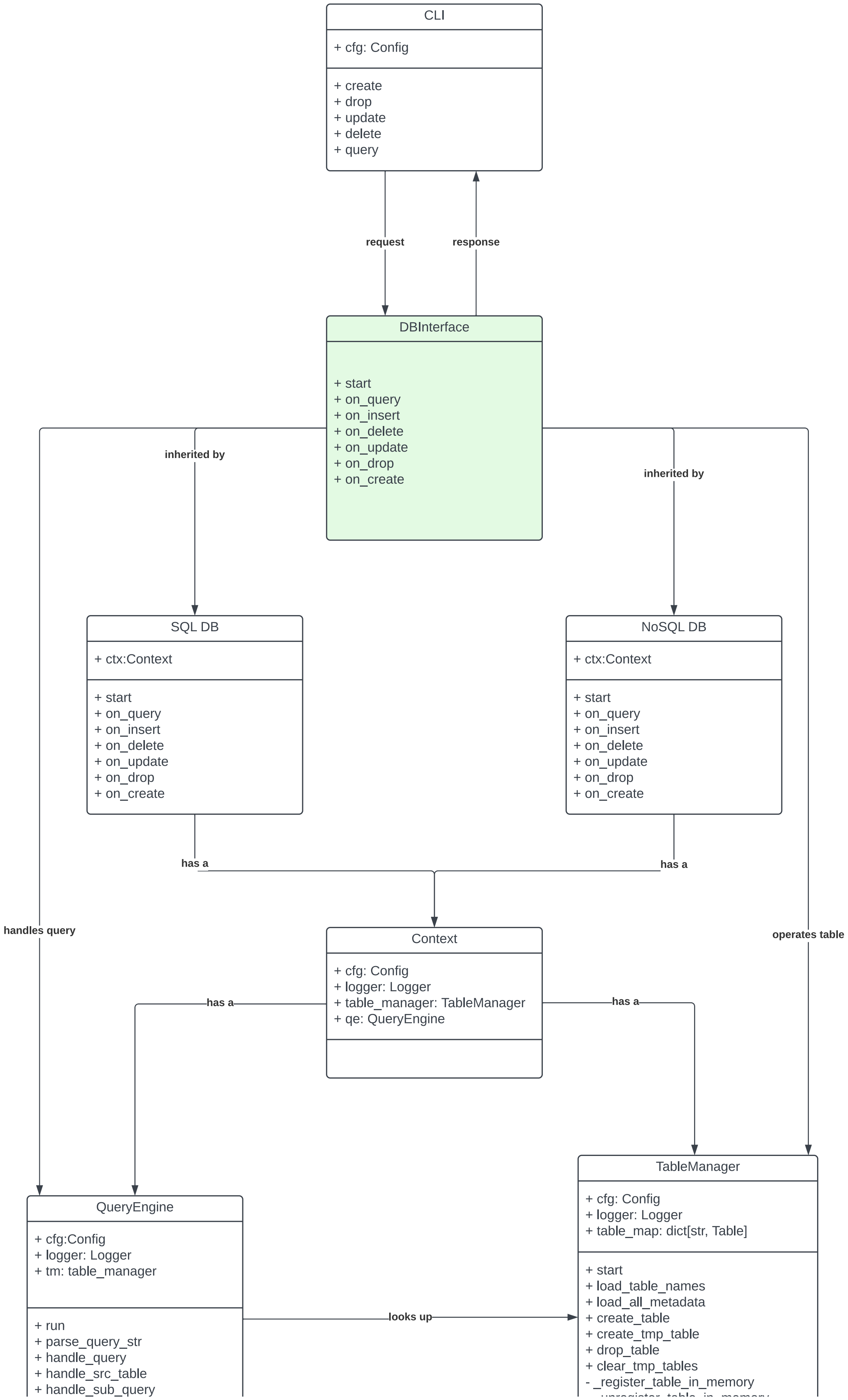
   Offers an interactive interface for users to directly communicate with the server. It translates command-line inputs into HTTP requests that are processed by `server.py`.
8. **Query Engine(app/common/query/query_engine.py):**

   Processes and executes queries by interfacing with the SQL or NoSQL database. It plays a crucial role in the efficient handling of database queries.
9. **Chunk Management(app/common/table/chunk_manager.py) :**

   Critical for handling large datasets in a limited memory space. This component ensures data is processed and stored in manageable chunks, allowing efficient data manipulation without overloading the system's memory. This approach is particularly important for operations like data import, query processing, and table management.

Flow Diagram

The flow diagram (presented below) illustrates the interaction between these components.



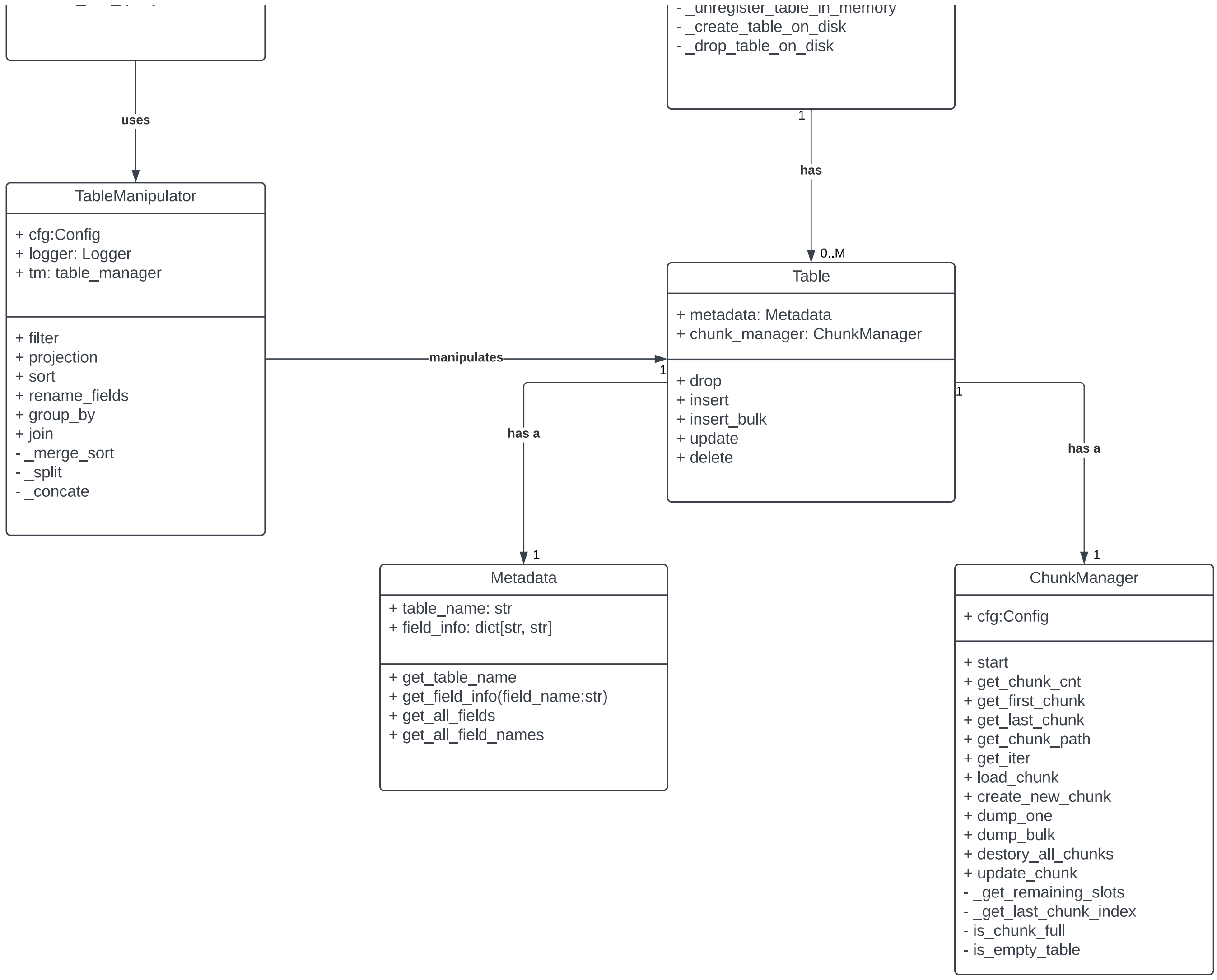


Diagram Description

1. **CLI**
    - interacts with users and send table operations or queries to flask server
    - formats query result for human to read
2. **DBInterface**
    - it's a flask server
    - parses requests (CRUD operations, table creation/dropping)
    - decides which database the CLI is talking to and send the request to corresponding database to handle
    - formats the query result before sending back to CLI
3. **Context**
    - holds instance of query engine, table manager, logger, and config
    - gets initialized when starting the flask server
    - be passed to all other modules to use
4. **TableManager**
    - loads all info of tables from the disk and registers them in memory

    - handles creation, deletion of tables
    - offers apis to create a temporary table, all temporary tables will be deleted when starting the flask server
5. **QueryEngine**
    - parses queries and decides the execution plan
    - uses `TableManipulator` to execute each step in the execution plan
6. **TableManipulator**
    - implements all operations including: projection/renaming/filtering/ for both SQL and NoSQL
    - uses `ChunkManager` to create/load/update/delete chunks of tables on the disk
7. **ChunkManager**
    - manages all chunks of a table on the disk
    - offers CRUD apis for chunks
8. **Logging:**
    - logs all operations across modules
    - useful for debugging

Implementation

Functionalities

Projection: This functionality involves selecting specific columns from a table. In the context of the AvA Database, the projection operation allows for tailoring the data output to include only the desired columns. This is particularly useful when dealing with large datasets, where you might only need a subset of the available data fields.

Filtering: Filtering refers to the capability to select data records based on specific criteria. The AvA Database can perform efficient filtering operations, allowing users to retrieve only those records that meet specified conditions. This is crucial for querying and analyzing data based on specific requirements.

Grouping: Grouping is the process of collating data records based on one or more columns. This feature in the AvA Database project is essential for organizing data into subsets for further analysis, such as aggregation or detailed examination of grouped data.

Aggregation: Aggregation involves computing summary statistics or other analytical operations on groups of data. The AvA Database supports various aggregation functions like count, sum, average, minimum, and maximum. These operations are particularly valuable in data analysis, allowing for the extraction of meaningful insights from grouped data.

Ordering: This function sorts data based on one or more columns, either in ascending or descending order. In the AvA Database, ordering is implemented to organize data in a specific sequence, which is crucial for reporting, data visualization, and further analytical processing.

Insertion: The database system allows for the insertion of new records into a database table. This functionality is fundamental to any database system, enabling the addition of new data into the existing dataset.

Updation: The AvA Database provides the capability to update existing records in the database. This feature is essential for maintaining the accuracy and relevance of the data stored in the database.

Deletion: This functionality pertains to the removal of existing records from the database. The ability to delete data is crucial for data management and ensuring that the database only contains current and relevant information.

Sorting (Ordering): The AvA Database's sorting functionality allows for the organization of data based on specified criteria. This can be done on one or more columns and can be configured to sort data in either ascending or descending order. Sorting is essential for data analysis, as it arranges data in a meaningful order, making it easier to understand, compare, and visualize. The database might employ algorithms like merge sort, which are particularly efficient for large datasets, ensuring that the sorting process is both fast and reliable.

Join: Join operations are crucial in relational database management systems, and the AvA Database supports them to combine data from two or more tables. This feature is based on a common field shared between the tables. The types of joins can include inner joins, left/right outer joins, and full outer joins. Joining is particularly important for relational databases as it allows for the relational model to be effectively utilized, enabling users to construct complex queries that gather data across multiple tables. This capability is vital for scenarios where data is normalized and spread across different tables but needs to be viewed or analyzed together.

Tech Stack

1. The project is mainly developed with `Python`. Some `Bash` scripts are also used for starting servers and cli.
2. `Flask` framework is used to serve both SQL and NoSQL database engines.
3. We use `Postman` to test apis offered by servers.
4. `Pytest` is used for unittests.
5. Version control system is `Git`. We use `Github` to collaborate.
6. We use standard libs of `Python` to implement the entire project. No other third-party libs are used. (e.g. pandas)

Implementation Screenshots

CLI

```python
cli > cli.py
import json
import requests
import config
import constant
import logger
import tempfile

logger = logger.get_logger(constant.CLI_NAME)


def download_file(url, query) -> str: ⋯


def print_file(file_name: str): ⋯


supported_apis = [constant.REQUEST_KEY_QUERY, constant.REQUEST_KEY_CREATE, constant.REQUEST_KEY_DROP, constant.REQUEST_KEY_DELETE, constant.REQUEST_KEY_


def is_request_valid(request_json) -> bool: ⋯


def parse_json_input(user_input: str): ⋯


def get_cfg(request_json) -> config.DBConfig | None: ⋯


def main():
    logger.info("User connects")
    print(constant.CLI_WELCOME)
    while True:
        user_input = input(constant.CLI_PROMPT)
        if user_input.lower() == 'exit':
            logger.info("User exits")
            break
        try:
            request_json = parse_json_input(user_input)
            if request_json is None:
                logger.error("Invalid user input: {}".format(user_input))
                continue
            cfg = get_cfg(request_json)
            if constant.REQUEST_KEY_QUERY in request_json:
                tmp_file = download_file('http://localhost:{}'.format(cfg.get_port()), json.dumps(request_json.get(constant.REQUEST_KEY_QUERY)))
                if tmp_file == "":
                    print("Get result from database failed")
                    continue
                print_file(tmp_file)
            else:
                method = [key for key in request_json.keys() if key in supported_apis][0]
                response = requests.get("http://localhost:{}/{}".format(cfg.get_port(), method), json=json.dumps(request_json[method]))
                print(response.text)
        except Exception as e:
            logger.error("An unexpected error occurred: {}".format(e))
            continue


if __name__ == "__main__":
    main()
```

Flask Server

app > server.py

```python
from app.common.context.context import Context
from typing import cast

app = Flask(__name__)


@app.route('/')
def get_file():
    query = request.json
    result, status = g.ctx.get_db().on_query(query)

    if not status.ok():
        return "failed to process query: {}, due to {} ".format(query, status.code()), 400

    if result is None:
        return "query result is empty for query {} ".format(query), 200

    result = cast(QueryResult, result)
    g.ctx.logger.info("result dat can be found under {}".format(result.get_result_file_path()))

    return send_file(result.get_result_file_path(), as_attachment=True, mimetype='text/plain')


@app.route('/insert')
def insert():...


@app.route('/update')
def update():...


@app.route('/delete')
def delete():...


@app.route('/drop')
def drop():...


@app.route('/create')
def create():...


def check_args():...


def get_db(logger: logger.Logger, cfg: config.DBConfig):
    if cfg.is_sql():
        return SQLDBFactory.instance()
    elif cfg.is_nosql():
        return NosqlDBFactory.instance()
    else:
        logger.error("unrecognized database type {}, unable to start".format(ctx.get_cfg().get_db_type()))
        return None


def start_db(ctx: Context) -> Status:
    status = ctx.get_db().start(ctx)
    if not status.ok():
        ctx.logger().error("failed to start {}, due to {}".format(ctx.get_cfg().get_db_type(), status))
        return START_FAILED

    return OK


@app.before_request
def before_request():
    # global context which could be used in entire life cycle
    g.ctx = ctx


if __name__ == '__main__':
```

QueryEngine: handle query

app > common > query > query_engine.py

```python
    def handle_query(self, q: dict) -> (Table | None, Status):
        # 1. handle src table, may contain subquery and joining
        src_table, status = self.handle_src_table(q)
        if not status.ok():
            self.logger.error("failed to handle query {} due to failed to parse src_tables".format(q))
            return None, INVALID_ARGUMENT

        res_table = src_table
        # 2. handle group by
        if constant.QUERY_GROUP_BY in q:
            columns = q[constant.QUERY_GROUP_BY]
            if len(columns) > 1:
                self.logger.error("failed to handle more than one group by in query {}!".format(q))
                return None, NOT_IMPLEMENTED
            reduce_options = []
            for column in q[constant.QUERY_DESIRED_COLUMNS_KEY]:
                if FieldNameProcessor.get_suffix(column) != "":
                    reduce_options.append(ReduceOption(column, ReduceOperation[FieldNameProcessor.get_suffix(column)]))
            self.logger.info("grouping by table {}".format(res_table.name))
            res_table = TableManipulator.group_by(res_table, GroupByOption(columns[0], reduce_options))
            self.logger.info("group by on table {} is finished".format(res_table.name))

        # 3. filter data
        if constant.QUERY_FILTER_KEY in q:
            expression = q[constant.QUERY_FILTER_KEY]
            self.logger.info("filtering table {}".format(res_table.name))
            res_table = TableManipulator.filter(res_table, Selector(expression))
            self.logger.info("table is filtered, new_table is {}".format(res_table.name))

        # 4. sorting
        if constant.QUERY_ORDER_BY_KEY in q:
            sort_options: list[dict] = q[constant.QUERY_ORDER_BY_KEY]
            if len(sort_options) > 1:
                self.logger.error("doesn't support order by 2 columns".format(q[constant.QUERY_ORDER_BY_KEY]))
                return None, NOT_IMPLEMENTED
            self.logger.info("sorting table {}".format(res_table.name))
            res_table = TableManipulator.sort(res_table, [
                SortOption(
                    option.get(constant.QUERY_ORDER_BY_COLUMN_KEY),
                    option.get(constant.QUERY_ORDER_BY_ASC_KEY)) if constant.QUERY_ORDER_BY_ASC_KEY in option else True  # default is asc
                for option in sort_options
            ])
            self.logger.info("table is sorted, new table: {}".format(res_table.name))

        # 5. projection
        # keep all columns by default
        if constant.QUERY_DESIRED_COLUMNS_KEY in q:
            columns = q[constant.QUERY_DESIRED_COLUMNS_KEY]
            modified_columns = []
            for column in columns:
                if FieldNameProcessor.get_inner_prefix(column) in self.prefix_map:
                    modified_columns.append(FieldNameProcessor.replace_inner_prefix(column, self.prefix_map[FieldNameProcessor.get_inner_prefix(column)]))
                else:
                    modified_columns.append(column)
            if len(modified_columns) == 0:
                self.logger.warn("empty table due to empty desired_columns field in query: {} ".format(q))
                return None, INVALID_ARGUMENT
            self.logger.info("projecting table {} to columns: {}".format(res_table.name, modified_columns))
            res_table = TableManipulator.projection(res_table, modified_columns)
            self.logger.info("table is projected to columns: {}, new table is {}".format(modified_columns, res_table))

        return res_table, OK

    def run(self, query: str) -> (Table | None, Status):
        q, status = self.parse_query_str(query)
        if not status.ok():
            return None, INVALID_ARGUMENT

        return self.handle_query(q)
```

TableManipulator: merge_sort

app > common > table > manipulator.py

```python
    @staticmethod
    def filter(src_table: Table, selector: Selector) -> Table: ···

    # TODO support nosql, which has nested fields i.e. a.b.c
    @staticmethod
    def projection(src_table: Table, desired_column: list[str]) -> Table: ···

    @staticmethod
    def _sort_each_chunk(src_table: Table, column: str, is_asc: bool) -> Table: ···

    @staticmethod
    def _merge_sort(src_table: Table, ways: int, column: str, is_asc: bool) -> Table:
        if ways <= 1:
            raise RuntimeError("invalid ways {} for merge sort".format(ways))

        total_chunks = src_table.chunk_manager.get_chunk_cnt()
        # if there are 0 - 100 chunks, then 1 run is needed
        # if there are 101 - 10000 chunks, then 2 run is needed
        # ...
        passes = math.ceil(math.log(total_chunks, ways))
        input_table = src_table
        step = 1
        generated_runs_cnt = total_chunks
        for _ in range(passes):
            cm = input_table.chunk_manager
            # total_chunks must be updated for each pass, as the size might shrink
            total_chunks = cm.get_chunk_cnt()
            generated_runs_cnt = math.ceil(generated_runs_cnt / ways)
            out_put_table, status = get_table_manager().create_tmp_table(src_table.metadata)
            if not status.ok():
                raise RuntimeError("failed to create new tmp table")
            idx = 0
            for i in range(generated_runs_cnt):
                runs = []
                # load at most ways chunks into memory
                for j in range(ways):
                    if idx >= total_chunks: ···
                    chunk, status = cm.load_chunk(idx)
                    if not status.ok():
                        raise RuntimeError("failed to load records")
                    runs.append([chunk, max(min(step, total_chunks - idx) - 1, 0), min(idx + step, total_chunks)])
                    idx += step
                idx_per_way, finish_cnt, new_records = [0 for _ in range(len(runs))], 0, []
                while finish_cnt != len(runs):
                    # if any way has run out of element, it should be ignored
                    elements = [(runs[j][0][idx_per_way[j]], j) for j in range(len(runs)) if idx_per_way[j] != len(runs[j][0])]

                    def sort_key(item):
                        return item[0][column]

                    next_idx = min(elements, key=sort_key)[1] if is_asc else max(elements, key=sort_key)[1]
                    # notice, here we don't have to flush chunk into memory whenever it's full
                    new_records.append(runs[next_idx][0][idx_per_way[next_idx]])
                    idx_per_way[next_idx] += 1
                    if idx_per_way[next_idx] == len(runs[next_idx][0]):
                        if runs[next_idx][1] != 0:
                            chunk, status = cm.load_chunk(runs[next_idx][2] - runs[next_idx][1])
                            if not status.ok():
                                raise RuntimeError("failed to load records")
                            runs[next_idx][0] = chunk
                            runs[next_idx][1] -= 1
                            idx_per_way[next_idx] = 0
                    if idx_per_way[next_idx] == len(runs[next_idx][0]) and runs[next_idx][1] == 0:
                        finish_cnt += 1
                status = out_put_table.insert_bulk(new_records)
                if not status.ok():
                    raise RuntimeError("failed to flush onto disk")
            step *= ways
            input_table = out_put_table

        return input_table

```

TableManager

app > common > table > table_manager.py

```python
# Manager of all tables, avoid data racing
class TableManager:

    def __init__(self, ctx: Context):⋯

    def load_table_names(self) -> (list[str], Status):⋯

    def load_all_metadata(self) -> (dict[str, Metadata], Status):⋯

    def _register_table_in_memory(self, table_name: str, metadata: Metadata) -> (Table | None, Status):⋯

    def _unregister_table_in_memory(self, table_name: str) -> Status:⋯

    # will override
    def _create_table_on_disk(self, table_name: str, metadata: Metadata) -> Status:⋯

    # fault tolerate
    def _drop_table_on_disk(self, table_name: str):⋯

    def drop_table(self, table_name: str) -> Status:⋯

    def create_table(self, table_name: str, metadata: Metadata) -> (Table | None, Status):⋯

    def check_consistency_between_metadata_and_table(self) -> bool:⋯

    def create_tmp_table(self, metadata) -> (Table | None, Status):⋯

    def is_tmp_table(self, table_name: str):⋯

    def clear_tmp_tables(self) -> Status:⋯

    def start(self) -> Status:
        if self.is_started():⋯

        status = self.clear_tmp_tables()
        if not status.ok():⋯

        # check all tables under tables subdir
        # any table under tables subdir must have a metadata file under metadata (no dangling table is allowed)
        if not self.check_consistency_between_metadata_and_table():⋯

        metadata_dict, status = self.load_all_metadata()
        if not status.ok():⋯

        for name, metadata in metadata_dict.items():
            if name in self.table_map:
                self.logger.error("duplicated table {} detected", name)
                return DUPLICATED_TABLE_CREATION_REQUEST
            table_path = os.path.join(self.tables_dir, name)
            if not os.path.exists(table_path):
                os.makedirs(table_path)
            self.table_map[name] = Table(table_name=name, metadata=metadata, ctx=self.ctx)

        self.state = TableManagerState.RUNNING
        self.logger.info("table manager started successfully")
        return OK

    def is_started(self) -> bool:⋯

    def get_table(self, table_name: str) -> Table | None:⋯


table_manager_singleton = None


def get_table_manager(ctx: Context = None) -> TableManager:⋯


if __name__ == "__main__":⋯
```

Table

```python
# iterate through a by chunks
class TableIterator:…


# The abstraction of actual tables storing on the disk
# It should support two different data formats: json or csv
# Supported operations: projection, filtering, sorting, deduplication, grouping by,joining two tables
# It also contains all information of the table including metadata, location of metadata and actual table on the disk
class Table:

    def __init__(self, table_name: str, metadata: Metadata, ctx: Context):
        self.name = table_name
        self.metadata = metadata
        self.chunk_cnt = 0
        self.logger = ctx.get_logger()
        self.cfg = ctx.get_cfg()
        self.chunk_manager: ChunkManager = ChunkManager(os.path.join(self.cfg.get_tables_dir(), table_name), metadata, ctx)
        self.chunk_manager.start()

    # drop this table, delete from disk
    def drop(self) -> Status:…

    # table should know how many chunks are on the disks, where to find each chunk, and how to update/delete/insert data into these trunks
    # record should be a json object
    def insert(self, record: dict) -> Status:…

    def insert_bulk(self, records: list[dict]) -> Status:
        # self.logger.info("inserting a record {} to table {}".format(record, self.name))
        status = self.chunk_manager.dump_bulk(records)
        if not status.ok():
            self.logger.warn("failed to insert record #{} to table {}".format(len(records), self.name))
            return status
        # self.logger.info("record {} is inserted to table {}".format(record, self.name))
        return OK

    # TODO add lock for all these operations on table
    def update(self, selector, new_record: dict) -> Status:…

    # TODO add lock for all these operations on table
    def delete(self, selector) -> Status:…
```

ChunkManager

app > common > table > chunk_manager.py

```python
class ChunkManager:

    # metadata is only used for SQL
    def __init__(self, table_path: str, metadata: Metadata, ctx: Context):…

    def start(self) -> Status:…

    def get_chunk_cnt(self):…

    def get_fist_chunk(self) -> (list[object], Status):…

    def get_last_chunk(self) -> (list[object], Status):…

    def get_chunk_path(self, chunk_idx) -> str:…

    def get_iter(self) -> ChunkIterator:…

    def get_remaining_slots(self, occupied_cnt: int) -> int:…

    def get_last_chunk_index(self) -> int:…

    # return a list of objects which can be iterated through
    # for sql, we have to transfer each row (which is simply a list to an object) to use it with ease
    def load_chunk(self, chunk_idx: int) -> (list[dict[str, object]], Status):…

    def create_new_chunk(self) -> Status:…

    def is_empty_table(self) -> bool:…

    def is_chunk_full(self, chunk: list[object]) -> bool:…
```

```python
    # always append new record to the last chunk
    # if the last chunk is full i.e. len(chunk) == config.chunk_size, then create a new chunk as the last chunk
    def dump_one(self, record: dict) -> Status:⋯

    def dump_bulk(self, records: list[dict]) -> Status:
        if self.is_empty_table():
            status = self.create_new_chunk()
            if not status.ok():
                return INTERNAL

        chunk, status = self.get_last_chunk()
        if not status.ok():
            self.logger.error("failed to append new record as unable to load last chunk")
            return status

        # fill the last chunk
        remaining = self.get_remaining_slots(len(chunk))
        insert_cnt = min(remaining, len(records))
        chunk += records[:insert_cnt]
        status = self.update_chunk(self.get_last_chunk_index(), chunk)
        if not status.ok():
            self.logger.error("failed to append records to the last chunk")
            return status
        records = records[insert_cnt:]

        # create new chunks and fill
        while len(records):
            status = self.create_new_chunk()
            if not status.ok():
                return INTERNAL
            size = min(self.max_chunk_size, len(records))
            status = self.update_chunk(self.get_last_chunk_index(), records[:size])
            if not status.ok():
                return INTERNAL
            records = records[size:]
        return OK

    def destroy_all_chunks(self) -> Status:⋯

    def update_chunk(self, chunk_idx: int, chunk: list[dict[str, object]]) -> Status:⋯
```

Learning Outcomes

1. **Designing a Unique Query Language**
    - **Challenge:** Creating a query language distinct from existing ones, which needed to be expressive, clear, and easy to parse, presented a unique set of difficulties.
    - **Solution:** We chose JSON as the format for our query language, striking a balance between expressiveness and simplicity, ensuring ease of parsing and clarity.
2. **Handling Multiple Source Tables and Subqueries**
    - **Challenge:** Managing references to specific fields across multiple tables and nested subqueries posed a significant challenge, particularly in maintaining clarity and accuracy for users and in the backend processing.
    - **Solution:** We devised a set of naming conventions that could be seamlessly integrated into our query language and the query engine. This innovation greatly simplified referencing fields in complex query scenarios.
3. **Integration of SQL and NoSQL**

- **Challenge:** A major challenge was integrating SQL and NoSQL databases to handle both CSV (structured) and JSON (possibly nested) data in a unified manner.
- **Solution:** We developed a system where components like the TableManipulator and ChunkManager are versatile enough to support both data formats. This approach minimized code duplication and streamlined our development process.

4. **Memory Management and Chunk-Based Data Storage**
   - **Challenge:** Given the constraint of limited memory, efficiently managing large datasets by storing and processing them in chunks was a significant technical challenge.
   - **Solution:** We carefully designed the ChunkManager, Table, and TableManipulator classes, simplifying the implementation of complex operations like merge sorting on chunks. To ensure the reliability of each API, extensive unit testing was conducted. This comprehensive testing approach reduced the likelihood of errors when integrating different system components.

Individual Contribution

Tasks	Assignee
Design Query Language	Weihao Zhao
Design Data Model & Import Data	Zhenyu Xiong
Design Testcases	Weihao Zhao
Implement SQL Database	Zhenyu Xiong
Implement NoSQL Database	Weihao Zhao
Implement CLI	Weihao Zhao
Draft Project Proposal	Weihao Zhao
Draft Midterm Report	Zhenyu Xiong
Draft Final Report	Weihao Zhao
Make Demo Video	Zhenyu Xiong

Conclusion

This project has been an exemplary journey in database management system design, especially in the context of an educational setting. By successfully integrating SQL and NoSQL databases and developing a custom JSON-based query language, our team not only met but exceeded the project's requirements. This accomplishment underscores our ability to address complex challenges, such as handling various data formats and implementing efficient memory management in a constrained environment.

The collaboration throughout this project was solid and effective, with each team member contributing significantly to the project's success. We navigated the complexities of database systems, developing solutions for managing multiple source tables, subqueries, and implementing chunk-based data storage. These experiences have greatly enhanced our understanding of databases, showcasing our adaptability and problem-solving skills in the face of technical challenges.

In essence, this project has not only fulfilled its academic objectives but has also provided us with invaluable insights into the practical aspects of database system design and development, laying a strong foundation for our future endeavors in this field.

Future Scope

1. **Concurrency Control**

   Introducing locks for operations on tables is an essential aspect of concurrency control in database systems, ensuring data integrity and consistency during concurrent access.

2. **Implementation of Mock Server in CLI:**

   A mock server setup in `cli/mock_server.py` is planned but not yet implemented. Completing this would enhance testing and simulation capabilities for the CLI.

3. **Type Conversion Handling:**

   Handling potential type conversions (e.g., int to float) during data manipulations is an area for development. This addition would increase the robustness and accuracy of data processing.

4. **Completion of Test Cases:**

   Developing comprehensive test cases is crucial for ensuring the system's reliability and performance under various scenarios.