Game Reviews

This project implements a lightweight in-memory database system called **MiniDB**, designed to demonstrate efficient data handling using core data structures learned in the course. The system loads and processes the Steam dataset, then applies a manually implemented **AVL Tree** for fast searching and range queries. It also includes a **graph module** for exploring developer–publisher relationships and a set of **analytics functions** for generating statistical insights such as minimum, maximum, and averages. The entire system is integrated into a simple **Streamlit interface**, allowing users to search games, filter by price, explore graph connections, and view basic analytics interactively. Overall, the project showcases how data structures like AVL Trees and graphs can be combined to build a functional and efficient data management tool.

Utils (`data_loader.py` & `schemas.py`)

1. schemas.py

Purpose

`schemas.py` defines all **schemas** for every dataset file.

A schema tells the system:

- what columns exist in each CSV

- what data type each column must be converted into
- how records must be cleaned and validated

This ensures **consistent typing**, avoids errors, and allows all modules to treat data uniformly.

It is a static dictionary.

- **Time Complexity:** `O(1)` (constant lookup for schema)
- **Space Complexity:** `O(N)` where `N` = total columns across all schemas

  (because each schema entry is stored in memory)

2. data_loader.py

Purpose

`data_loader.py` loads, cleans, normalizes, and prepares all CSV datasets before they are passed to the MiniDB system.

The main responsibilities:

1. Open CSV files
2. Convert each value to the proper type using the schema
3. Replace missing values
4. Produce list of clean dictionaries
5. Return a final dictionary:

Function: load_csv()

Reads a CSV file line-by-line.

Time Complexity:

- Reads each row once → `O(n)` where `n` = number of rows

Space Complexity:

- Stores only one row at a time → `O(1)` additional memory

Function: clean_value()

Converts a raw string to:

- int
- float
- lowercase string

Handles empty/missing values.

Time Complexity: `O(1)`

Space Complexity: `O(1)`

Function: clean_record()

Cleans one CSV row using the schema.

If a CSV has `m` columns:

Time Complexity: `O(m)`

Space Complexity: `O(m)` for building cleaned row dictionary

Storage (avl_tree.py & datastore.py)

The `avl_tree.py` file implements a **self-balancing Binary Search Tree (BST, AVL Tree**.)

It ensures that search, insert, delete, and range operations all run in **O(log n)** time by maintaining a height-balanced property using rotations.

AVL Tree is used as the **indexing structure** in the DataStore.

Class: `AVLTree`

Contains all major AVL Tree operations.

`insert(root, key, value)`

Inserts a key-value pair into the tree.

- Recursively inserts a node.
- Recalculates heights.
- Detects imbalance and performs **LL / RR / LR / RL rotations**.

Time Complexity

- **Worst-case:** O(log n)
- Height update + rotation: O(1)

`delete(root, key)`

Deletes a node by key.

Steps:

1. BST deletion (3 cases: no child, one child, two children)
2. Replace with inorder successor when needed
3. Rebalance using rotations

Time Complexity

- **O(log n)**

`search(root, key)`

Standard BST search.

Time Complexity

- **O(log n)**

`in_order(root)`

Returns list of `(key, value)` pairs in sorted order.

Time Complexity

- **O(n)**

`left_rotate()` / `right_rotate()`

Used to rebalance tree when height difference ≥ 2.

Time Complexity

- **O(1)** per rotation

Purpose of the DataStore Module

`DataStore` is an **in-memory database-like structure** that:

- Stores raw records in a list
- Maintains **secondary indexes** using AVL Trees
- Supports:
    - Fast insert
    - Fast search by attribute
    - Range queries
    - Deletion and updates
    - Optional reuse of deleted record IDs

It acts like a **mini relational table + indexing engine**.

`insert_record(record)`

Stores the record and updates AVL indexes.

Steps:

1. Check all indexed attributes exist.
2. Choose ID:
    a. Reuse from `free_ids` or
    b. Append to end
3. Insert into AVL indexes:
    a. Key = record[attr]
    b. Value = record_id (or list)

Time Complexity is O(k log n) where k = number of indexed attributes

`search_by_attr(attr, key)`

Fetches all records with given attribute value.

Time Complexity

AVL search	O(log n)
Map IDs → records	O(m) where m = number of matches

`range_query(attr, low, high)`

low ≤ record[attr] ≤ high

Uses optimized AVL in-order traversal.

Time Complexity

- Traversal visits only nodes in range + ancestors

- **O(log n + r)** where r = number of returned records

  (Optimal for balanced trees)

`delete_record(record_id)`

Removes record from:

- Every AVL index (removes id from value lists)
- Marks slot as `None`
- Optionally stores id in `free_ids`

Time Complexity

- Remove from each AVL index: O(log n)
- Total: **O(k log n)**

`update_record(record_id, new_record)`

Steps:

1. Remove old values from AVL indexes
2. Overwrite record in place
3. Reinsert new values
4. Remove from free pool if exists

Time Complexity

- Delete + Insert per index = O(log n)
- **Total: O(k log n)**

`get_record(record_id)`

Returns record if exists.

Time Complexity

- O(1)

Space complexity for AVL O(n), for DataStore O(kn) where k is the number of indexed items.

Query_Engine (query_handler.py)

The `query_handler.py` file implements the **Query Engine** of the project.

It acts as a high-level interface between the user and the low-level `DataStore` (which stores records + AVL tree indexes).

Responsibilities

- Provide CRUD functions:
    - `insert_record()`
    - `search_record()`
    - `update_record()`

    - `delete_record()`
- Provide `range_query()`
- Ensure operations use AVL indexes for fast lookup
- Maintain a clean separation between:
    - **Storage layer** (raw array + AVL trees)
    - **Query layer** (this file)

2. Class: QueryEngine

Constructor: `__init__(self, data_store, key_attribute="id")`

Purpose

- Connect the QueryEngine to an existing `DataStore`
- Ensure the chosen primary key is indexed (necessary for O(log n) search)

Time Complexity

- Checking if a key is in a set → **O(1)**
- No expensive operations during initialization

Space Complexity

- Stores only references → **O(1)** additional space

3. CRUD OPERATIONS

3.1 INSERT — `insert_record(self, record)`

What it does

- Delegates directly to `DataStore.insert_record`
- Inserts a new record and automatically updates AVL indexes

Time Complexity

Depends on underlying DataStore:

- Insert into array → **O(1)**
- Insert into AVL tree index → **O(log n)**

Overall:

- **Average:** O(log n)
- **Worst-case:** O(log n)

Space Complexity

- Insert one record into storage → **O(1)** additional
- AVL tree node addition → **O(1)**

3.2 SEARCH — `search_record(self, key)`

What it does

- Uses AVL tree index on the primary key to quickly find matching records

Time Complexity

- AVL tree search → **O(log n)**
- Retrieving all records with that key → usually **O(k)** (k = matches, typically small)

Overall:

- **Average:** O(log n)
- **Worst-case:** O(log n + k)

Space Complexity

- Returning results list → **O(k)**

3.3 UPDATE — `update_record(self, key, field, value)`

What it does

1. Search via AVL index → fast
2. Iterates over **entire underlying storage list** to find records with that key
3. Replaces each matching record and updates index inside DataStore

Time Complexity

- Step 1: AVL search → **O(log n)**
- Step 2: Full scan over storage → **O(n)**
- Step 3: Updating a record in AVL → **O(log n)** per update

Overall:

- **Average case:** O(n)
- **Worst case:** O(n log n) (**if many records share same key**)

Reason for high cost

Because the engine:

```
for record_id, rec in enumerate(self.store.records):
```

It must scan the entire list.

Space Complexity

- Temporary updated record dictionary → **O(1)**

3.4 DELETE — `delete_record(self, key)`

What it does

Similar pattern to update:

1. Search via AVL index
2. Scan full storage
3. Delete matching records from DataStore (and AVL indexes)

Time Complexity

- AVL search → **O(log n)**
- Scan → **O(n)**
- AVL deletion → **O(log n)** each

Overall:

- **Average:** O(n)
- **Worst-case:** O(n log n)

Space Complexity

- No extra memory → **O(1)**

4. RANGE QUERY — `range_query(self, attr, low, high)`

What it does

- Delegates directly to DataStore range query
- DataStore uses AVL tree traversal (in-order)

Time Complexity

- AVL range traversal → **O(log n + m)**
    - m = number of returned items

Space Complexity

- Return list → **O(m)**

Analytics

search_by_name.py

Purpose:

Provides a search interface for finding applications by name within a 2000-record subset. Performs case-insensitive exact matching by converting both stored names and user input to lowercase.

Method Analysis

search_by_name_section(indexed_apps): Linear scan through 2000 records

- **Time Complexity:** O(N) where N = 2000 (string comparison for each record)
- **Space Complexity:** O(R) where R ≤ 20 (displayed results stored in session state)

indexed_engine.py

Purpose:

Constructs an indexed query engine using AVL trees for efficient lookups on application data. Builds a 2000-record subset and creates indexes on *appid* and *price*.

Method Analysis

build_applications_engine(cleaned_data, max_records): Builds AVL-indexed data store

- **Time:** O(N log N) for N ≤ 2000 insertions
- **Space:** O(N) for records + O(N) for index structures

build_indexed_engine_section(cleaned_data): Wrapper for UI

- **Time:** O(N log N) (dominated by engine build)
- **Space:** O(N) (engine + subset)

price_range.py

Purpose:

Enables range-based price queries using the AVL price index, allowing efficient filtering of applications by price.

Method Analysis

price_range_section(app_engine): Executes range query

- **Time:** O(log N + R) (logarithmic search + linear result collection)
- **Space:** O(R) (results cached in session state)

dataset_status.py

Purpose:

Displays a summary table showing loaded dataset names and row counts, giving an overview of available data.

Method Analysis

show_dataset_status(cleaned_data): Iterates through tables

- **Time:** O(T), T ≈ 10
- **Space:** O(T) for summary structure

graph_explorer.py

Purpose:

Implements a developer–publisher relationship graph explorer using custom graph structures. Provides text-based visualization, neighborhood views, BFS/DFS traversal, and graph statistics—all without external graph libraries.

Method Analysis

_build_small_dev_pub_graph(): Builds bipartite graph

- **Time:** O(V + E), V ≤ 300, E ≤ 5000
- **Space:** O(V + E)

render_graph_explorer(): Main UI

- **Time:** O(V + E) (graph build + algorithms)
- **Space:** O(V + E) + O(V) for traversal state

_render_neighbors_table(): Displays neighbors

- **Time:** O(D), D = degree
- **Space:** O(D)

_render_traversal(): Shows BFS/DFS order

- **Time:** O(V + E) (limited to 40 displayed)
- **Space:** O(V)

basic_analytics.py

Purpose:

Computes basic statistical metrics (min, max, average, median) for application attributes on the 2000-record subset.

Method Analysis

basic_analytics_section(indexed_apps): Computes statistics

- **Time:** O(N log N), dominated by sorting for median
- **Space:** O(N) for extracted values

search_by_appid.py

Purpose:

Provides efficient appid-based record lookup using the AVL index, demonstrating the benefits of indexed primary key queries.

Method Analysis

search_by_appid_section(app_engine): Indexed lookup

- **Time:** O(log N)
- **Space:** O(1)

analytics.py

Purpose:

Planning/stub file outlining the future analytics module for statistical calculations and dataset insights.

Graph (graph_model.py & graph_algorithms.py)

graph_model.py

Purpose:

Provides an adjacency-map graph structure (directed or undirected). Defines Vertex and Edge classes. Used mainly to build a developer–publisher bipartite graph for running BFS, DFS, shortest path, and connected components.

Method:

Stores adjacency dictionaries mapping vertices to their edges.

Vertices store: dataset element, kind ("developer" or "publisher"), and ID.

Edges store: endpoints and metadata such as collaboration weight and game IDs.

`build_dev_pub_graph()` creates vertices for developers and publishers, groups

apps by devs and pubs, and inserts edges for each collaboration, increasing weight for repeated collaborations.

Analysis:

Efficient representation using hash maps. Edge insertion avoids duplicates via lookup dictionary. Graph structure is bipartite (developer ↔ publisher only), which simplifies traversal and avoids cross-connections.

Time Complexity:

Vertex insertion: O(1).

Edge insertion: O(1) average.

Iterating neighbors: O(degree(v)).

Building the dev-pub graph: O(A + V + E), where A = applications, V = vertices, E = edges.

Space Complexity:

Vertices: O(V).

Edges and adjacency: O(E).

Temporary app-grouping structures: O(A).

Stored metadata (game sets): O(E).

graph_algorithms.py

Below are the algorithms implemented in `graph_algorithms.py`:

- BFS Traversal

- DFS Traversal
- Shortest Path (BFS)
- Connected Components

Each includes Purpose, Method, Analysis, Time Complexity, and Space Complexity.

1. BFS Traversal

Purpose:

Perform a breadth-first exploration of the graph starting from a given vertex and return the order of visited vertices.

Method:

Use a queue.

Start by visiting the initial vertex, mark it as visited, and enqueue it.

Repeatedly dequeue a vertex, process it, and enqueue all unvisited neighbors discovered via incident edges.

Analysis:

BFS explores layer-by-layer, giving the shortest distance in terms of edge count from the starting vertex. It is good for shortest path and reachability.

Time Complexity: O(V + E)

Space Complexity: O(V)

2. DFS Traversal

Purpose:

Perform a depth-first exploration of the graph and return the discovery order.

Method:

Use recursion.

Start from the initial vertex, mark it visited, then recursively visit each unvisited neighbor.

Analysis:

DFS explores deeply first before backtracking. Useful for exploring structure, detecting components, cycles, and ordering tasks. It may go deep on long paths and increase recursion depth.

Time Complexity: O(V + E)

Space Complexity: O(V)

3. Shortest Path

Purpose:

Find the shortest path between a start vertex and a target vertex in an unweighted graph.

Method:

Use BFS.

Maintain a parent map that records where each vertex was discovered from.

Once the target vertex is reached, reconstruct the path by backtracking through the parent pointers.

Analysis:

Guarantees a true shortest path because BFS expands vertices in increasing distance layers.

Time Complexity: O(V + E)

Space Complexity: O(V)

4. Connected Components

Purpose:

Find all connected components of an undirected graph.

Method:

Iterate through each vertex.

If the vertex is unvisited, treat it as the start of a new component.

Use a stack-based DFS to collect all reachable vertices into the same component.

Analysis:

Efficiently splits the graph into disjoint connected subgraphs.

Time Complexity: O(V + E)

Space Complexity: O(V)

UI (app.py)

The file **app.py** serves as the **main entry point** for the project's Streamlit user interface. Its primary purpose is to initialize the application, load the cleaned dataset once, and orchestrate all UI sections by calling functions that render different analytics and search components.

Key Responsibilities of This File

1. Set up the project environment

- Adjusts `sys.path` so Python can properly import project modules.
- Ensures the root directory of the project is discoverable by the rest of the code.

2. Configure the Streamlit application

- Sets the page title and layout with `st.set_page_config()`.
- Displays basic title and introductory text confirming that the UI is running.

3. Load and preprocess all CSV data

- Defines a cached function `load_clean_data()` using `@st.cache_data`.
- Loads all datasets through `DataLoader` and applies schemas.
- Ensures this expensive operation happens only once, improving performance.

4. Coordinate and render UI sections

The file does **not** contain analytics logic itself; instead, it **calls component functions** that belong to different UI sections:

- Dataset status visualization
- Indexed search engine builder
- Search by App ID
- Search by Application Name
- Price range search
- Basic analytics
- Graph explorer

The file's role is to **schedule these sections in order** and pass the cleaned dataset or built index structures to them.

5. Act as the controller of the entire UI

`app.py` works like a "controller":

- Initializes the data
- Creates shared objects (like `app_engine`, `indexed_apps`)
- Passes them to the appropriate UI components