First Assignment: Enhance Wrangler with Byte Size and Time Duration Parsers

---

Step to Complete the Assignment

1. Fork & Setup

- Fork the repo: https://github.com/data-integrations/wrangler
- Clone it locally and set up using Maven:

```
git clone https://github.com/YOUR_USERNAME/wrangler.git
cd wrangler
mvn clean install
```

---

2. Grammar Modifications – ANTLR Lexer & Parser

Add Lexer Rules:

- Byte units

```
BYTE_UNIT: ('B' | 'KB' | 'MB' | 'GB' | 'TB' | 'PB');
BYTE_SIZE: DIGITS BYTE_UNIT;
```

- Time units

```
TIME_UNIT: ('ns' | 'us' | 'ms' | 's' | 'm' | 'h');
TIME_DURATION: DIGITS TIME_UNIT;
```

- Helper

```
fragment DIGITS: [0-9]+ ('.' [0-9]+)?;
```

Modify Parser Rules:

```
byteSizeArg: BYTE_SIZE;
timeDurationArg: TIME_DURATION;
```

Regenerate Parser:

```
mvn compile
```

---

3. Create Token Classes (wrangler-api)

ByteSize.java

```
public class ByteSize extends Token {
    private final long bytes;
    public ByteSize(String value) {
        this.bytes = parseBytes(value);
    }
    private long parseBytes(String value) {
        value = value.trim().toUpperCase();
        if (value.endsWith("KB")) return (long)(Double.parseDouble(value.replace("KB", "")) *
1024);
        if (value.endsWith("MB")) return (long)(Double.parseDouble(value.replace("MB", "")) *
1024 * 1024);
        GB, TB
        return Long.parseLong(value.replace("B", ""));
    }
    public long getBytes() {
        return bytes;
    }
}
```

TimeDuration.java

```
public class TimeDuration extends Token {
```

```
    private final long nanoseconds;

    public TimeDuration(String value) {

        this.nanoseconds = parseDuration(value);

    }

    private long parseDuration(String value) {

        value = value.trim().toLowerCase();

        if (value.endsWith("ms")) return (long)(Double.parseDouble(value.replace("ms", "")) *
1_000_000);

        if (value.endsWith("s")) return (long)(Double.parseDouble(value.replace("s", "")) *
1_000_000_000);

        return Long.parseLong(value);

    }

    public long getNanoseconds() {

        return nanoseconds;

    }

}
```

Register New Token Types

In TokenType.java, add:

- BYTE_SIZE,
- TIME_DURATION

---

4. Core Parser Updates (wrangler-core)

In visitor (e.g., RecipeVisitor.java):

```
@Override
```

```
public Token visitByteSizeArg(DirectivesParser.ByteSizeArgContext ctx) {
    return new ByteSize(ctx.getText());
}

@Override
public Token visitTimeDurationArg(DirectivesParser.TimeDurationArgContext ctx) {
    return new TimeDuration(ctx.getText());
}
```

---

5. Implement aggregate-stats Directive

AggregateStats.java *(implements Directive)*

- Read 4 arguments:
    - Input: **data_transfer_size, response_time**
    - Output: **total_size_mb, total_time_sec**
- Store totals in ExecutorContext.Store
- In execute(...):
    - Sum converted values per row
        - Finalize:
        - Convert total bytes → MB
        - Convert total time → seconds

---

6. Testing

Unit Tests:

- ByteSizeTest.java and TimeDurationTest.java
- Example cases: 10KB, 1.5MB, 2s, 100ms

Parser Tests:

- Update GrammarBasedParserTest.java with new syntax

Directive Test:

- Use TestingRig and assert the aggregate values:

```
Assert.assertEquals(expectedSizeMB, row.getValue("total_size_mb"), 0.001);
Assert.assertEquals(expectedTimeSec, row.getValue("total_time_sec"), 0.001);
```

---

7. Final Touches

- prompts.txt: Save prompts you used with AI tools
- README.md: Add usage for:

```
aggregate-stats :data_transfer_size :response_time total_size_mb total_time_sec
```

Second Assignment: Integration Assignment: Bidirectional ClickHouse & Flat File Data Ingestion Tool

1. Objective:

- Build a **web application** with a **frontend UI** and **backend logic**.
- **Bidirectional Data Flow**:
    - **ClickHouse ➔ Flat File** (export selected columns from ClickHouse to file).
    - **Flat File ➔ ClickHouse** (import selected columns into ClickHouse).
- **JWT Authentication** for ClickHouse connection.
- Allow **user column selection**.
- Report the **total number of processed records** after ingestion.

---

2. Core Requirements:

Application Type:

- Web application with **Frontend + Backend**.

Bidirectional Flow:

- **ClickHouse ➔ Flat File**: Read from ClickHouse, write to a flat file (e.g., CSV).
- **Flat File ➔ ClickHouse**: Read from flat file, insert into ClickHouse.

Source Selection:

- UI dropdown/button to choose:
    - "ClickHouse" (source or target).
    - "Flat File" (source or target).

ClickHouse Connection (as Source):

- UI Input fields:
    - Host
    - Port
    - Database
    - Username
    - JWT Token
- Use a **ClickHouse official client** (examples: clickhouse-driver for Python, clickhouse-go for Golang, etc.).

- Authenticate using **JWT**.
- **Client Library**

Language	Client Library	Link
Python	clickhouse-connect	Official modern client for Python. Supports HTTP(s) and native TCP connections.
Golang (Go)	clickhouse-go	Official Go client. High performance, supports native TCP protocol.
Java	clickhouse-jdbc	Official JDBC driver for Java applications.

Python:

Installation

pip install clickhouse-connect

Importing

import clickhouse_connect

Main script

client = clickhouse_connect.get_client(host='localhost', port=8123, username='default', password='your_jwt_token')

result = client.query('SELECT * FROM your_table')

print(result.result_rows)

- **Flat Files:**
  - **Read the header row (first line) of the file to fetch column names.**
  - **Display columns in the UI as checkboxes for user selection.**
- **For ClickHouse:**
  - **Connect using the client library (e.g., clickhouse-connect, clickhouse-go).**
  - **Execute a metadata query:**

SQL

SHOW TABLES;

DESCRIBE TABLE <table_name>;

  - Display table names, and after selecting a table, fetch and display column names as checkboxes.

Goal: Let the user select which columns they want to ingest.

Ingestion Process

- **Based on user-selected columns:**
    - Flat File: Read only the selected columns while processing rows.
    - ClickHouse Target:
        - Insert into a target table or create one if needed.
- **Efficiency Tips:**
    - Implement batching (e.g., 500–1000 rows per batch) instead of inserting row-by-row.
    - Alternatively, support streaming ingestion if the file is huge.

Example Batch Insert (Python clickhouse-connect):

python

```
client.insert('target_table', batch_rows)
```

Error Handling:

- Handle and display:
    - Connection failures.
    - Authentication errors.
    - Query/IO errors.
    - Ingestion failures.
    - Show friendly error messages in UI.

---

3. User Interface (UI) Requirements:

Source/Target Selection

- Allow the user to select:
    - **Source**:
        - ClickHouse database
        - OR Local Flat File
    - **Target**:
        - Always ClickHouse database (for ingestion)

Input Fields for Connection Parameters

- **For ClickHouse (Source or Target)**:
    - Hostname / IP
    - Port
    - Database Name
    - Username
    - Password
    - (Optional) SSL toggle
- **For Flat File (Source)**:
    - Local file picker (browse and select file)
    - Delimiter input field (default to ,)
    - (Optional) Encoding (UTF-8 default)

Mechanism to List Tables or Identify File

- **If Source = ClickHouse**:
    - After connecting, **list available tables** from the selected database.
    - Allow the user to **select a table**.
- **If Source = Flat File**:
    - **Parse header row** of the selected file to **identify column names**.
    - Optionally **preview a few sample rows**.

Column List Display with Selection Controls

- Display column names dynamically in the UI after connecting:
    - Show **checkboxes** next to each column.
    - Allow user to **multi-select** the columns they want to ingest.

Bonus Tip:
You can also allow **"Select All"** and **"Clear All"** buttons for convenience.

Action Buttons

Button	Purpose
Connect	Establish connection to ClickHouse or load the flat file
Load Columns	Fetch and show the columns from the source
Preview	Show a sample of data rows from source
Start Ingestion	Start the ingestion process based on selections

---

Status Display Area

- Show live status messages as the process flows:
    - Connecting
    - Fetching tables/columns
    - Previewing data
    - Starting ingestion
    - Completed successfully
    - Error encountered

Good practice: Use **progress bars or spinners** during long operations for better UX.

Result Display Area

- After operation completes:
    - **If successful**:
        - Show total record count ingested (e.g., " 50,000 records ingested.")
    - **If failed**:
        - Show a clear **error message**.
        - Optionally, show detailed error logs (expandable/collapsible).

---

4. Bonus Requirements:

Multi-Table Join (ClickHouse Source):

- Allow selection of **multiple tables**.
- Input fields for **JOIN Keys** or **Join Conditions**.
- Backend must build **JOIN queries** dynamically based on user input.
- Example:

SQL

```
SELECT a.id, a.name, b.price
FROM table1 a
JOIN table2 b ON a.id = b.id
```

- After JOIN ➔ allow column selection and ingest.

---

5. Optional Features (Enhancements):

Progress Bar:

- Show progress during ingestion (% or a loading bar).

Data Preview:

- Button to show **first 100 records** (selected columns) **before full ingestion**.

  SQL

  SELECT selected_columns FROM table LIMIT 100;

---

6. Technical Considerations:

Backend:

- Language: Prefer **Golang** or **Java**.
- But **Python**, **Node.js** are also acceptable.

Frontend:

- Options:
  - Simple: HTML/CSS/JS + Bootstrap.
  - Modern: React, Vue, Angular.

ClickHouse Instance:

- Run ClickHouse locally via **Docker**:

Bash

```
docker run -d --name clickhouse-server --ulimit nofile=262144:262144 -p 8123:8123 clickhouse/clickhouse-server
```

- Load **example datasets**:
    - uk_price_paid
    - ontime (use scripts from ClickHouse documentation)

JWT Handling:

- Use standard libraries to pass JWT Token when connecting to ClickHouse.

Data Type Mapping:

- Ensure column types of match.
- Example: map ClickHouse DateTime to Flat File string YYYY-MM-DD HH:MM:SS.
- Handle type mismatches gracefully.

---

7. Testing Requirements:

Datasets:

- Use **uk_price_paid** and **ontime** datasets.

Test Cases:

1. Single ClickHouse Table ➔ Flat File:
    - Select specific columns.
    - Verify exported record count.
2. Flat File ➔ New ClickHouse Table:
    - Upload a CSV.
    - Create table based on CSV header and types.
    - Verify data inserted properly.
3. (Bonus) Multi-Table Join:
    - Select 2+ tables.
    - Define JOIN.
    - Export joined results into Flat File.

- Verify counts.

4. Error Handling:
    - Test bad credentials ➔ authentication error.
    - Test network errors ➔ connection error message.
5. (Optional) Data Preview:
    - Show first 100 records in UI for user review before ingestion.
    - Connect to a source (ClickHouse or Flat File).
    - Select columns.
    - Click "Preview" before full ingestion.
    - Verify:
        1. The preview displays correct sample data.
        2. Data types and formatting are consistent.

---

8. AI Tools Usage:

- Use AI coding tools like **ChatGPT**, **GitHub Copilot**, **Tabnine**, **Replit AI**.
- You **must record prompts** used (e.g., "How to connect ClickHouse with JWT using Go?").
- Save all prompts in a prompts.txt file in your GitHub repository.

  Example :

1Prompt: "Generate React.js form with input fields for database connection parameters."

2Prompt: "Write Go code to connect to ClickHouse database using the official Go client."

3Prompt: "Suggest error handling approach for a file upload form in React."

4Prompt: "Example of SQL query to join two tables in ClickHouse."

---

9. Deliverables:

- Source Code: Check into **GitHub** (public or private repo with access).
- README.md:
    - Setup instructions (e.g., install dependencies, run server, etc.)
    - Config instructions (how to configure ClickHouse / file paths).
    - How to use the tool step-by-step.
- AI Prompts:
    - File prompts.txt in the repo listing all prompts used.