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

1. Background

The CDAP Wrangler library currently parses various data types but lacks built-in support for easily handling units like Kilobytes (KB), Megabytes (MB), milliseconds (ms), or seconds (s). Users often need to perform calculations or conversions on columns representing data sizes or time intervals, which requires complex multi-step recipes.

This assignment aims to enhance the Wrangler core library by adding native support for parsing and utilizing byte size and time duration units within recipes. This involves modifying the grammar, updating the parsing logic, extending the API, and implementing a new directive to demonstrate the usage.

2. Objectives

- Integrate new lexer tokens (BYTE_SIZE, TIME_DURATION) into the Wrangler grammar.
- Update the relevant Java code (wrangler-api, wrangler-core) to handle these new token types.
- Implement a new aggregate directive (aggregate-stats) that utilizes these new types.
- Develop comprehensive test cases, including aggregation scenarios using the new units.

3. Detailed Tasks

- **Fork the repo - https://github.com/data-integrations/wrangler to your own github handle. All changes should be committed back to this repository with a section in Readme.md about the usage of these 2 new parsers.**
- **(a) Grammar Modification (wrangler-core/src/main/antlr4/.../Directives.g4):**
  - Add the lexer rules for BYTE_SIZE and TIME_DURATION (including helper fragments BYTE_UNIT, TIME_UNIT).
  - Modify relevant *parser* rules (e.g., value, or create new specific rules like byteSizeArg, timeDurationArg) to accept BYTE_SIZE and TIME_DURATION tokens where appropriate for directive arguments.
  - Regenerate the ANTLR Java parser/lexer code using the appropriate build process (e.g., mvn compile).
- **(b) API Updates (wrangler-api module):**
  - Create new Java classes ByteSize.java and TimeDuration.java extending Token [wrangler-api/src/main/java/io/cdap/wrangler/api/parser/Token.java]

        - These classes should parse the token string (e.g., "10KB", "150ms") in their constructor.
        - Provide methods to retrieve the value in a canonical unit (e.g., long getBytes() for ByteSize)
    - Add BYTE_SIZE and TIME_DURATION to the Token Types
    - Update usage definition and token definition to support specifying these new token types as valid directive arguments.
- **(c) Core Parser Updates (wrangler-core module):**
    - Modify by adding visit methods for the parser rules created/modified in step 3a (e.g., visitByteSizeArg, visitTimeDurationArg, or modifying visitValue if applicable).
    - Hint : chk ctx.getText()).
    - Add the created token instances to the TokenGroup.
- **(d) New Directive Implementation (wrangler-core module):**
    - Create a new directive class, which can do aggregation, implementing the Directive interface
    - **UsageDefinition (define()):** The directive should accept at least four arguments:
        1. ColumnName (source column with byte sizes).
        2. ColumnName (source column with time durations).
        3. ColumnName (target column name for total size).
        4. ColumnName (target column name for total or average time). Optionally, add arguments to specify output units (e.g., 'MB', 'GB', 'seconds', 'minutes') or aggregation type (total, average).
    - **Initialization :** Store the source and target column names provided in the arguments.
    - **Execution :**
        - This directive should operate as an *aggregate*. So you need a store to accumulate totals. Think about it. Hint : Chk ExecutorContext
        - For each row:
            - Read the byte size value from the source size column.
            - Read the time duration value from the source time column.
            - Add these values (converted to canonical units like bytes and nanoseconds) to running totals stored in the Store.
        - **Finalization :** This method (if using RecipePipeline's aggregation capabilities) or logic within the last execute call needs to:
            - Retrieve the final totals from the Store.
            - Perform unit conversions if required by arguments (e.g., convert total bytes to MB, total nanoseconds to seconds).

        - Return a *single* new Row containing the target columns with the calculated aggregate values (e.g., total_size_mb, total_time_sec).
- **(e) Testing (wrangler-core module):**
    - Add unit tests for the ByteSize and TimeDuration classes, verifying correct parsing and canonical value retrieval for various inputs (e.g., "10kb", "1.5MB", "5ms", "2.1s").
    - Add parser tests (e.g., in GrammarBasedParserTest.java or RecipeCompilerTest.java to ensure recipes using the new syntax are parsed correctly and invalid syntax is rejected.
    - Add comprehensive unit tests for the new AggregateStats directive. See the Test Case Specification below.

4. Test Case Specification: Aggregation

- **Input Data:** Create a list of Row objects or parse a simple file representing sample log or transaction data.
  E.g You can use TestingRig
- **Recipe:**
  String[] recipe = new String[] {
  // Example: Aggregate size (output MB), total time (output seconds)
  "aggregate-stats :data_transfer_size :response_time total_size_mb total_time_sec"
  // Add variations for average, median, p95, p99 and different output units if implemented
  };

- **Execution:** Use TestingRig.execute(recipe, rows) to run the recipe.
- **Expected Output:** Assert that the output contains a *single* row with the correctly calculated aggregate values.
    - **Size Calculation:** Sum all data_transfer_size values (converted to bytes) and then convert the final sum to Megabytes (MB) for the total_size_mb column (using 1 MB = 1024 * 1024 bytes or 1000 * 1000 bytes - be consistent!).
    - **Time Calculation:** Sum all response_time values (converted to nanoseconds or milliseconds) and then convert the final sum to seconds for the total_time_sec column. (If implementing average, divide by the number of rows before unit conversion).
    - Example assertion structure:
      // results = TestingRig.execute(recipe, rows);
      Assert.assertEquals(1, results.size());
      Assert.assertEquals(expectedTotalSizeInMB,

```
results.get(0).getValue("total_size_mb"), 0.001); // Use tolerance for
float/double
Assert.assertEquals(expectedTotalTimeInSeconds,
results.get(0).getValue("total_time_sec"), 0.001);
```

5. AI Tools Usage:

- We strongly encourage taking AI coding assistance using any tool of your choice
- Record the prompts you are using in these tools.

6. Deliverables

- Assignment will be only evaluated if committed to github.
- Modified Directives.g4 file.
- All new and modified Java source files (.java) within wrangler-api and wrangler-core modules.
- All new and modified unit test files (.java) within wrangler-core.
- Evidence of successful build and test execution.
- If AI tooling is used - the set of prompts which you have recorded to be sent or checked into github as a **prompts.txt** file

6. Evaluation Criteria (Example)

- Correctness: Do the new tokens parse correctly? Does the directive compute aggregates accurately? Do tests pass?
- Code Quality: Is the code well-formatted, commented, and easy to understand? Are existing patterns followed?
- Robustness: Does the code handle edge cases (e.g., zero values, large numbers, different unit cases)?
- Test Coverage: Are the new features adequately tested?

Integration Assignment: Bidirectional ClickHouse & Flat File Data Ingestion Tool

1. Objective:

Develop a web-based application with a simple user interface (UI) that facilitates data ingestion between a ClickHouse database and the Flat File platform. The application must support bidirectional data flow (ClickHouse to Flat File and Flat File to ClickHouse), handle JWT token-based authentication for ClickHouse as a source, allow users to select specific columns for ingestion, and report the total number of records processed upon completion.

2. Core Requirements:

- **Application Type:** Web application (backend logic + frontend UI).
- **Bidirectional Flow:** Implement both:
  - ClickHouse -> Flat File ingestion.
  - Flat File -> Clickhouse ingestion.
- **Source Selection:** UI must allow users to choose the data source ("ClickHouse" or "Flat File").
- **ClickHouse Connection (as Source):**
  - **UI Configuration:** Inputs for Host, Port (e.g., 9440/8443 for https, 9000/8123 for http), Database, User, and **JWT Token**.
  - **Authentication:** Use the provided JWT token via a compatible ClickHouse client library.
  - **Client Library:** Use a client from the official list: https://github.com/ClickHouse (Use any language of your choice - Golang, Python, Java).
- **Flat File Integration:**
  - **UI Configuration :** Local Flat File name, Delimiters
  - **Client Library** - Use any IO library.
- **Schema Discovery & Column Selection:**
  - Connect to the source and fetch the list of available tables (ClickHouse) or the schema of the Flat File data.
  - Display column names in the UI with selection controls (e.g., checkboxes).
- **Ingestion Process:**
  - Execute data transfer based on user selections.
  - Implement efficient data handling (batching/streaming recommended).
- **Completion Reporting:** Display the total count of ingested records upon success.

- **Error Handling:** Implement basic error handling (connection, auth, query, ingestion) and display user-friendly messages.

3. User Interface (UI) Requirements:

- Clear source/target selection.
- Input fields for all necessary connection parameters (ClickHouse source/target, Flat File).
- Mechanism to list tables (ClickHouse) or identify Flat File data source.
- Column list display with selection controls.
- Action buttons (e.g., "Connect", "Load Columns", "Preview", "Start Ingestion").
- Status display area (Connecting, Fetching, Ingesting, Completed, Error).
- Result display area (record count or error message).

4. Bonus Requirements:

- **Multi-Table Join (ClickHouse Source):**
  - Allow selection of multiple ClickHouse tables.
  - UI element to input JOIN key(s)/conditions.
  - Backend logic to construct and execute the JOIN query for ingestion.

5. Optional Features (Enhancements):

- **Progress Bar:** Visual indicator of ingestion progress (can be approximate).
- **Data Preview:** Button to display the first 100 records of the selected source data (with selected columns) in the UI before full ingestion.

6. Technical Considerations:

- **Backend:** Go or Java preferable. But any language accepted.
- **Frontend:** Simple HTML/CSS/JS, React, Vue, Angular, or server-side templates.
- **ClickHouse Instance:** Local (Docker) or cloud-based. Load example datasets for testing.
- **JWT Handling:** Use libraries to manage JWTs if needed, primarily pass the token to the ClickHouse client.
- **Data Type Mapping:** Consider potential type mismatches between ClickHouse and Flat File/CSV.

7. Testing Requirements:

- **Datasets:** Use ClickHouse example datasets like uk_price_paid and ontime (https://clickhouse.com/docs/getting-started/example-datasets).
- **Test Cases:**
  1. Single ClickHouse table -> Flat File (selected columns). Verify count.
  2. Flat File (CSV upload) -> New ClickHouse table (selected columns). Verify

count & data.
3. (Bonus) Joined ClickHouse tables -> Flat File. Verify count.
4. Test connection/authentication failures.
5. (Optional) Test data preview.

8. AI Tools Usage:

- We strongly encourage taking AI coding assistance using any tool of your choice
- Record the prompts you are using in these tools.

9. Deliverables:

- Source code - Git repository check in is a must-have. Else wont be evaluated.
- README.md with setup, configuration, and run instructions.
- (Optional) Short demo video.
- If AI tooling is used - the set of prompts which you have recorded to be sent or checked into github as a **prompts.txt** file