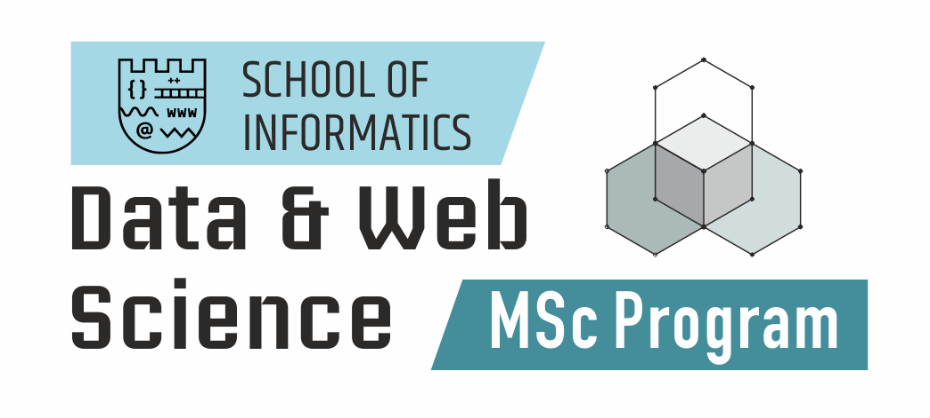
****

**Distributed Data Processing**

****

**Programming Assignment**

Vasiliki Zarkadoula

(135)

Contents

[Implementation Approach 2](#_Toc137629694)

[Database Files 2](#_Toc137629695)

[Table Creation 2](#_Toc137629696)

[Table Check 3](#_Toc137629697)

[Pipelined Hash Join 3](#_Toc137629698)

[Semi Join 3](#_Toc137629699)

[Lazy vs Eager Strategy 4](#_Toc137629700)

[Lazy vs Filter-Timestamps-Then-Join (FTTJ) Strategy 4](#_Toc137629701)

[Main 4](#_Toc137629702)

[Comparisons and Comments 5](#_Toc137629703)

[Application Containerization 7](#_Toc137629704)

# Implementation Approach

## Database Files

I first downloaded locally SQLite3 to be able to create two local empty database files. To create an SQLite database file on Windows using the SQLite3 command-line tool, I followed these steps:

1. Opened the command prompt.
2. Navigated to the directory where I wanted to create the database file using the cd command.
3. Launched the SQLite shell by typing sqlite3 and pressing Enter. This opens the SQLite shell prompt.
4. At the SQLite shell prompt, I used the command



to create a new database file. Replaced <database\_name> with the desired name for my databases files.

## Table Creation

After I created two SQLite3 databases locally, I used python’s library sqlite3 to connect to these databases, create tables in them and fill them with data. The implementation is done in the CreateTable class inside “create\_tables.py” file.

The CreateTable class is designed to simplify the creation of two tables, table1 in database1 and table2 in database2, and populate them with random data, including random timestamps. Both tables have the same four attributes: id (an auto-incrementing integer), name (a text field), email (a text field), and timestamp (a text field to store the generated timestamps). Tables can have different number of records, and they will be joined on id.

To generate random data, the class utilizes the generate\_timestamp() method, which takes a primary key (id) and an optional seed value as inputs. The method combines the primary key with the seed value to ensure randomization consistency. Using the primary key, it generates a random date within a fixed range (starting from June 1, 2023). It then adds a random time component (hour, minute, and second) to the date, resulting in a random timestamp.

The create\_table() method is responsible for creating a table in the specified database, using the provided connection and cursor objects. It first checks if the table exists and drops it if it does. Then, it creates the table with the appropriate column structure. Subsequently, it generates random data by iterating over a specified number of records. For each record, it generates a random name and email using the random module. Additionally, it calls the generate\_timestamp() method to obtain a random timestamp based on the record's primary key and seed. Finally, it inserts the generated data into the table.

By calling the create\_tables() method, you can trigger the creation of both table1 and table2 in their respective databases. Optionally, you can provide a seed value to achieve deterministic randomization.

## Table Check

Then, the CheckTables class in the “check\_tables” file verifies if the tables created in the previous implementation are populated with data, logs the records if they are not empty and returns the tables. It takes two database connections as inputs.

## Pipelined Hash Join

After tables are retrieved from the databases and saved in two lists, the HashJoin class in the “pipelined\_hash\_join.py” file implements the (double) pipelined hash join algorithm for joining two tables, table1 and table2. The class takes the table lists, the names of the tables, the specified timestamp difference limit, and a flag - if lazy strategy is followed- as inputs. The class initializes empty hash tables, ht1 and ht2, for probing and insertion and a counter to keep track of the joined records. It also initializes a logger to log the pipelined hash join operation.

The probe\_and\_insert() method performs probing and insertion in the hash tables. It takes a tuple representing a record from one of the databases and the respective hash tables as inputs. It compares the keys and the timestamp (if lazy strategy is enabled) of the given record with the records in the hash table and returns a result set if a match is found within the specified timestamp difference (if lazy strategy is enabled).

The perform\_pipelined\_hash\_join() method executes the double pipelined hash join algorithm. It iterates over the tuples from both tables, performs probing and insertion using the hash tables, and prints the join results. The method alternates between reading tuples from table1 and table2, probing one hash table, and inserting tuples into the other hash table. The join results are logged using the process\_join\_result() method.

The process\_join\_result() method prints the join result if it is not None. It increments the counter for counting the matching records.

## Semi Join

The SemiJoin class in “semi\_join.py” file implements the semi-join operation between two tables, table1 and table2. The class takes the table lists, the names of the tables, the specified timestamp difference limit, and a flag for lazy strategy and a flag for eager strategy as inputs. The class initializes a logger to log the semi-join operation.

The get\_largest\_table() method determines the largest table between table1 and table2. It compares the lengths of the table lists and returns the table with the greater number of records.

The perform\_semi\_join() method performs the semi-join operation using different strategies based on the provided flags. It obtains the largest table, S, and the other table, R.

If the lazy flag is set to True, lazy strategy is performed. This involves creating an index-like structure called S\_lookup in the form of a dictionary. The dictionary maps the 'id' values to their corresponding 'timestamp' values. The method iterates over each row in R and checks for matching rows in S using the indexed values in S\_lookup. Rows are added to the result set R1 if they satisfy the join condition based on 'id' and timestamp comparison.

If the eager flag is set to True, eager strategy is performed. In this approach, S\_lookup is created similarly to the lazy strategy. However, instead of fetching rows from R one at a time, all rows from R table are fetched in advance. This involves again creating an index-like structure called R\_lookup in the form of a dictionary. The dictionary maps the 'id' values to their corresponding row values. The join operation is then performed by comparing the indexed values in R\_lookup with the indexed values in S\_lookup. Matching rows are added to the result set R1 if they satisfy the join condition based on 'id' and timestamp comparison.

If both the lazy and eager flags are set to False, the same procedure as in lazy strategy is used, but without considering the timestamp comparison.

The method logs the matching rows, the memory size (in MB) used by the result and lookup dictionaries and increments the counter for counting the matches.

## Lazy vs Eager Strategy

In the lazy strategy, a single index-like structure is created for the larger table and then a row-by-row retrieval from the smaller table R is followed. In the eager strategy, index-like structures are created for both tables. The lazy strategy does not require storing the entire smaller table (R) in memory. Therefore, it can be more memory-efficient when dealing with large datasets.

## Lazy vs Filter-Timestamps-Then-Join (FTTJ) Strategy

Lazy and FTTJ are two approaches for performing semi-join and pipelined hash join operations with a timestamp difference limit.

Lazy strategy checks the timestamp during the join process and includes matching records if the timestamp difference is within the limit.

FTTJ strategy involves filtering the tables before performing the join operation. In this approach, the tables are first filtered to include only those records whose timestamps are different from the records in the other table by less than the timestamp difference limit. This filtering is performed using the filter\_tables() method in “main.py”. Then, those filtered tables are given as inputs to HashJoin and SemiJoin and the lazy flag is set to False.

## Main

The “main.py” script performs pipelined hash join and semi-join operations with both eager (for semi-join) lazy and FTTJ approaches. It connects to the two databases, creates tables, and checks their existence. Then it performs the join operations and logs the running time of every operation, the memory size (in MB) used by the result and lookup dictionaries (for semi-join) and the total number of resulted records.

# Comparisons and Comments

Lazy strategy proves to be significantly more efficient in terms of running time compared to FFTJ evaluation. It proves also to be more memory-efficient, as already mentioned. Additionally, I noticed that FFTJ strategy occasionally produces incorrect results by returning more records than lazy and eager strategies. To illustrate this issue, let's consider an example using two tables: Table1 from database1 and Table2 from database2.

Table1 of database1 contains the following records:

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **email** | **timestamp** |
| 26 | Name\_8 | email\_110@example.com | 27-06-23 23:58 |
| 27 | Name\_106 | email\_222@example.com | 28-06-23 20:30 |
| 28 | Name\_142 | email\_147@example.com | 29-06-23 3:47 |

Table2 of database2 contains the following records:

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **email** | **timestamp** |
| 26 | Name\_155 | email\_291@example.com | 27-06-23 18:03 |
| 27 | Name\_234 | email\_127@example.com | 28-06-23 1:58 |
| 28 | Name\_298 | email\_289@example.com | 29-06-23 7:25 |

In this scenario, the timestamp difference limit is set to 12 hours. In the FFTJ implementation, each timestamp record of Table1 is compared with every timestamp record of Table2. If the timestamp difference is less than the specified threshold, the tuple is stored in an auxiliary table. The same process is applied to Table2. Considering our example, the record with id 27 from Table1 is stored in the auxiliary table because its timestamp difference from the timestamp of the record with id 28 from Table2 is less than 12 hours. Similarly, the record with id 27 from Table2 is stored in the auxiliary table due to its timestamp difference from the timestamp of the record with id 26 from Table1 being less than 12 hours.

After the filtering process, these two records are joined because they share a common id attribute. However, their timestamp difference does not meet the 12-hour limit, resulting in an erroneous match that should not be included. This issue is effectively addressed through lazy and eager approaches, as they first check for equality in the join attribute and subsequently verify if the records satisfy the timestamp constraint.

The following tables illustrate the running time results (in seconds) , the memory size (in MB) used by the result and lookup dictionaries (for semi-join) and the number of matching records for different number of records inserted into the tables and different timestamp limits.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | timestamp\_diff | Lazy P.H.J. | FTTJ P.H.J | Lazy S.J | Eager S.J | FTTJ S.J |
| Running time | 6 | **0.0020** | 0.6220 | **0.0020** | 0.0029 | 0.3519 |
| Matched records | 6 | 54 | 54 | 54 | 54 | 54 |
| Size used | 6 | - | - | **0.0093** | 0.0138 | - |
| Running time | 12 | **0.0019** | 0.8529 | 0.0029 | 0.0030 | 0.3350 |
| Matched records | 12 | 80 | 82 | 80 | 80 | 82 |
| Size used | 12 | - | - | **0.0096** | 0.0141 | - |

Table : table1->100 records, table2-> 200 records

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | timestamp\_diff | Lazy P.H.J. | FTTJ P.H.J | Lazy S.J | Eager S.J | FTTJ S.J |
| Running time | 6 | 0.0269 | 37.0810 | **0.0260** | 0.0279 | 38.7190 |
| Matched records | 6 | 456 | 462 | 456 | 456 | 462 |
| Size used | 6 | - | - | **0.0739** | 0.1092 | - |
| Running time | 12 | 0.0280 | 29.0899 | **0.0239** | **0.0239** | 29.6715 |
| Matched records | 12 | 754 | 779 | 754 | 754 | 779 |
| Size used | 12 | - | - | **0.0763** | 0.1116 | - |

Table : table1->2000 records, table2-> 1000 records

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | timestamp\_diff | Lazy P.H.J. | FTTJ P.H.J | Lazy S.J | Eager S.J | FTTJ S.J |
| Running time | 6 | 0.2839 | - | 0.2479 | **0.2410** | - |
| Matched records | 6 | 4374 | - | 4374 | 4374 | - |
| Size used | 6 | - | - | **0.5990** | 0.8803 | - |
| Running time | 12 | 0.2809 | - | 0.2729 | **0.2530** | - |
| Matched records | 12 | 7453 | - | 7453 | 7453 | - |
| Size used | 12 | - | - | **0.6211** | 0.9025 | - |

Table : table1->10000 records, table2-> 20000 records

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | timestamp\_diff | Lazy P.H.J. | FTTJ P.H.J | Lazy S.J | Eager S.J | FTTJ S.J |
| Running time | 6 | 2.7700 | - | 2.7430 | **2.6659** | - |
| Matched records | 6 | 44066 | - | 44066 | 44066 | - |
| Size used | 6 | - | - | **10.3446** | 15.3447 | - |
| Running time | 12 | 2.8469 | - | 2.8730 | **2.5380** | - |
| Matched records | 12 | 75038 | - | 75038 | 75038 | - |
| Size used | 12 | - | - | **10.6212** | 15.6213 | - |

Table 4: table1->200000 records, table2-> 100000 records

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | timestamp\_diff | Lazy P.H.J. | FTTJ P.H.J | Lazy S.J | Eager S.J | FTTJ S.J |
| Running time | 6 | 28.1610 | - | 27.9059 | **26.1090** | - |
| Matched records | 6 | 437372 | - | 437372 | 437372 | - |
| Size used | 6 | - | - | **83.6367** | 123.6368 | - |
| Running time | 12 | 27.2889 | - | **24.9069** | 25.5150 | - |
| Matched records | 12 | 749713 | - | 749713 | 749713 | - |
| Size used | 12 | - | - | **85.8255** | 125.8256 | - |

Table 5: table1->1000000 records, table2-> 2000000 records

It is observed that lazy strategy usually demonstrates superiority over the other strategies in terms of efficiency (both time and memory usage). When comparing the running time results, both lazy pipelined hash join and lazy semi-join show similar performance, with semi-join slightly outperforming the pipelined hash join approach for larger datasets. FTTJ was not evaluated for larger datasets due to long due to long execution time.

# Application Containerization

To containerize my Python application and run it in Docker, I followed these steps:

1. Create a Dockerfile, making sure my project files (create\_tables.py, check\_tables.py, pipelined\_hash\_join.py, semi\_join.py, main.py and the two SQLite database files) are all located in the same directory. The Dockerfile sets the working directory in the container, installs the Python dependencies, copies the application code to the container, copies the databases folder into the container and defines the default command to run when the container starts
2. Build the Docker image:

* Open a terminal or command prompt
* Navigate to my project directory using the cd command
* Build the Docker image by running the following command



1. Run the Docker container using the following command:



To run my application, just navigate to my project folder and follow the above steps 2 and 3.