DBMS Performance Evaluation Report

Shengli Zhou, 12212232

Introduction

Databases have a wide range of applications in our everyday life. The era of big data also places higher demands on the efficiency and stability of databases. This report firstly **compares the efficiency of SQL operations with file I/O operations** and analyzes the differences. Then the report also **compares between different constraints and data types** to **give suggestions in designing databases**.

Method

Project structure

In the following experiments, we use 4 sets of codes:

- One in PostgreSQL, which represents standard SQL operations.
- One in Java, which can **load `.sql` files and perform similar operations**. In this code, we implemented several basic keywords of SQL, including `ALTER`, `COMMENT`, `CREATE`, `DELETE`, `DROP`, `INSERT`, `SELECT`, `UPDATE`, `DEFAULT`, `NULL`, `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `LIKE`, `BETWEEN ... AND ...`, `UPPER`, `LOWER`, etc. Our code can also **handle logical and arithmetic calculations**.
  - Here we implement a Java code to read `.sql` file directly instead of using fixed commands because **the time consumption in analyzing commands should be taken into consideration**.
- In addition, we implement another two codes (in C++ and Java respectively), which only performs the manipulation process (i.e. ignoring the time consumption in analyzing SQL commands) to **evaluate the difference caused by different programming language**.

Implementation details

In the following section, the report will show the details in implementing the first Java code mentioned above.

File I/O strategy

The total runtime of a SQL command can be categorized into two parts, namely execution time and fetching time.

- The **execution time** stands for the time consumption in retrieving and manipulating data (i.e. the "true" runtime for PostgreSQL).
- The **fetching time** stands for the time cost by connecting the database and transmitting data (file I/O's time consumption for Java program).

To reduce the fetching time, DataGrip uses the strategy of packing multiple commands as a batch and send them to DBMS together. In our Java program, we use a similar way to accelerate the I/O process by using `java.io.BufferedReader`.

`java.io.BufferedReader` has a buffer pool which can store 8192 characters and can read multiple lines in the `.sql` file instead of reading line by line as the `Scanner.nextLine()` does. Therefore, it can reduce access to file systems and shorten the time consumption of file I/O.

Handling expression

The expression can be categorized into two classes including logical expression and arithmetic expression. Since operators of same precedence can be calculated from left to right, we define that logical operations have a higher precedence. Also, we assume that the expression is not enclosed in a pair of parenthesis.

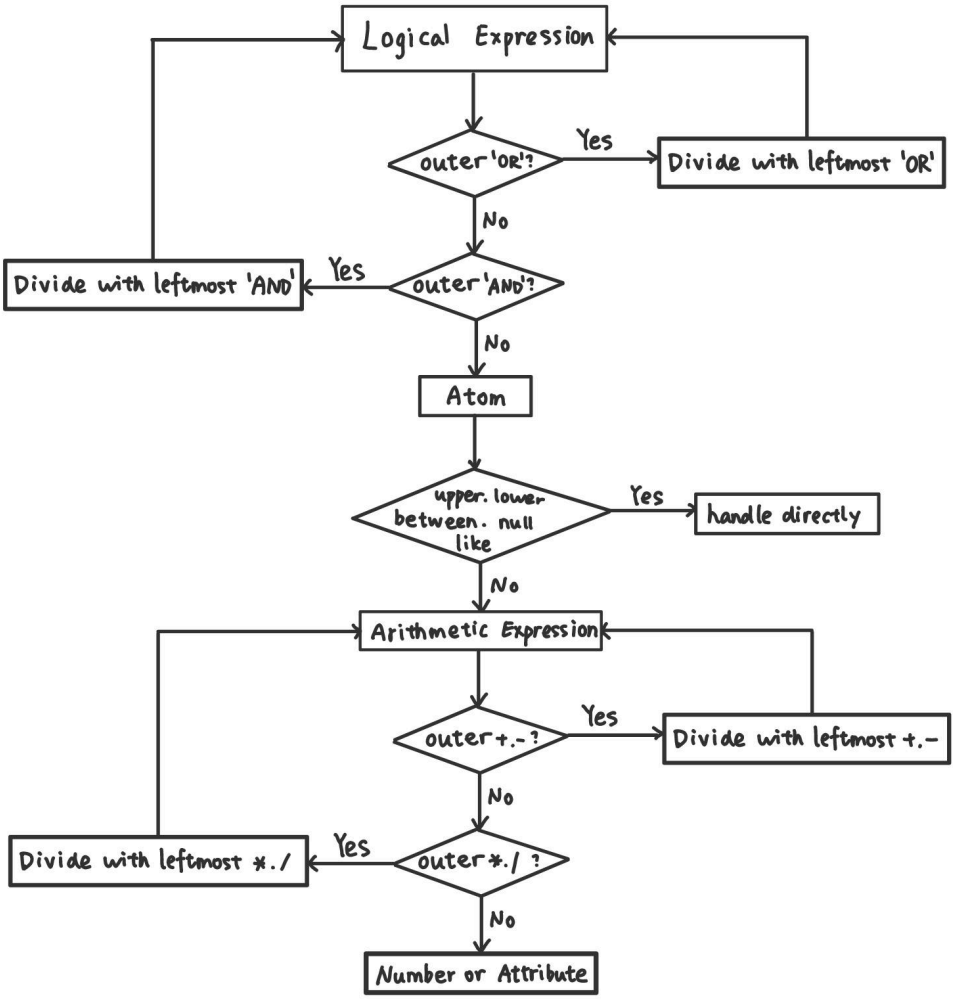

Figure 1: Process of handling expressions

For **logical operations**, according to the precedence of SQL, we firstly check if there are any `OR`s outside all parenthesis, if so, we divide the expression with the leftmost `OR`. Otherwise, we divide the expression with the leftmost `AND` outside all parenthesis if there is one. If there is neither `OR` nor `AND` outside all parenthesis, we treat the expression as an atom.

For an **atom**:

- If it has a pattern of `LIKE ...`, the code translates the subsequent string into regular expression (regex) and use `java.util.regex.Matcher` to search in the corresponding fields.
- If it has a pattern of `X BETWEEN Y AND Z`, we translate the requirement into `X >= Y AND X <= Z` and treat it as an logical expression.
- If it has token `UPPER`, `LOWER`, `IS NULL` or `IS NOT NULL`, the code checks the corresponding field of all records and make judgements.
- Otherwise, we regard the atom as an arithmetic expression.

The code use a similar method to handle **arithmetic expression** as handling logical expression. According to the precedence of SQL, we firstly search the leftmost `+` or `-` outside parenthesis to divide the expression. If there is no such operator, we seek for the leftmost `*` or `/` outside all parenthesis for division. If there's no `+`, `-`, `*` or `/` outside all parenthesis, we regard the string as either a number or the name of an attribute (depending on whether it can be parsed to `Double` in Java).

Since handling the string of arithmetic expression is quite time-consuming, we actually change the string into a **Prefix Notation** and the calculation on every record is based on the Prefix Notation instead of the original expression.

Data storage

We store a table in database using class `Table`. We use an `ArrayList` of class `Attribute` (which stores the information of an attribute) to store the attributes in the table. An `ArrayList` of `ArrayList<String>` is used for storing data in the table (each instance of `ArrayList<String>` is used for storing a record). We choose `ArrayList` instead of array in order to avoid exceeding bounds of the array.

`CREATE` command

The `CREATE` command is mainly used to create a table. The code divides the `CREATE TABLE` command into table naming part and attribute definition part. The latter one will be divided into several parts with exactly one attribute in each part. Class `Attribute` then process each piece of string and generate an instant of `Attribute` containing the name, type and other information of the attribute.

`ALTER` command

The `ALTER` command is mainly used to add a new column to the table. We first use the same method as creating table to add a new attribute, then we add a new element to each `ArrayList` representing a record in the table.

`DROP` command

When dropping a database, a schema or a table, just remove it from the `ArrayList` in the superior level. If it is then useless, Java's Garbage Collection will release the resources.

`INSERT` command

The code will add a new element to the `ArrayList` of the table and assign values of each attribute to the new element.

`SELECT`, `UPDATE` and `DELETE` command

The code will handle the expression describing the scope of the modification, then it will perform the corresponding operation.

Constraints

- For `NULL`, `NOT NULL` and `CHECK`, the code checks whether affected results are still valid every time after `INSERT` or `UPDATE` command.
- For `UNIQUE` and `PRIMARY KEY`, we use `java.util.HashSet` to check whether there are duplication in the table after each `INSERT` or `UPDATE` command.

`DEFAULT` keyword

If the default value for an attribute is not null, we set the value of the attribute when a record is created. We also check if the default value is valid when the default value is set or changed.

`COMMENT` command

The code has a field in class `Table` and class `Attribute` to store comments on the table or the attribute and can be set by the `COMMENT` command.

Other modules for the experiment

- The code can record runtime using classes in `java.time`, every time when it reads an empty line, it records the time and calculates the time consumption since the last empty line.
- Since our database is large, there can be quite a few result for the `SELECT` command. Therefore, the code prints the result in a `.csv` file to make it easier to read.
- After the whole execution, the code prints the timestamps and the final table into `.csv` files for analysis.

Timing method

During the experiment, we design several sets of SQL commands and use PostgreSQL or our Java code to perform the corresponding operations. Since **DBMS and JVM may not clear the cache after each operation**, the runtime may become shorter if we do the same operations repeatedly. Therefore, we apply the following two method to evaluate the average runtime more precisely.

- In the `.sql` file, we use the structure of

```
CREATE TABLE people(...);
INSERT INTO people VALUES ...
SELECT / ALTER / UPDATE ...
DROP TABLE people;
```

For PostgreSQL, as we drop the table in the last process, data in table is deleted and won't affect the subsequent duplicate experiments. For Java code, the stack memory is released after execution and will not affect the following experiments.

- Moreover, the **launching time** of PostgreSQL and JVM **decreases** if we keep running the program and the launching time can be ignored in real cases (DBMS is always working), so we should keep them activate in the experiments by keep running the program with relatively short separation.

In the following experiments, we use the following method to measure the time consumption in most cases:

- Keep running the program with about a minute's separation between two executions until the total runtime is relatively stable, specifically, when the range of the runtime of the last five experiments is lower than 30% of the average time.
- Take the average time of the last five duplicate experiments to be the measurement of time.

All measurements of time in the experiments are tested in this method if not specifically stated.

Experiments

Environment

The experiment is conducted under an environment with the following parameters:

Computer model	Dell Inspiron 15 3530
Processor	13th Gen Intel(R) Core(TM) i7-1355U 1.70 GHz
RAM	32.0 GB (31.7 GB available)
OS	Windows 11 (version 22621.2283), 64 bit
PostgreSQL	PostgreSQL 16rc1
DataGrip	DataGrip 2023.2.1
JDK	Oracle OpenJDK version 20.0.2

Table 1: Environment

All experiments were conducted with the computer connected to power and the "performance mode" turned on.

Comparison between Java and C++

Java interprets byte code during execution and C++ executes after compilation. Thus, C++ usually runs faster than Java.

Here we implemented a simple version of the above mentioned Java code which can do simple and fixed operation by both Java and C++ to **evaluate the runtime difference caused by programming language**. The comparison contains procedures including:

- `INSERT`: insert 1000000 records into a table (we use `vector<vector<string> >` in C++ to store data in a table).
- `SELECT`: select records where attribute `first_name` or `surname` contains string `ly`, then return a `vector / ArrayList` of the selected records' `id`.
- `ALTER TABLE`: add a new column to the table, i.e. add a element to the `vector / ArrayList` storing each record.
- `UPDATE`: change all `To` in `first_name` or `surname` to `TTOO`.

The average result of the 10 reduplicate experiments is shown in the table below (see raw data in Appendix 1):

Procedure	**Java's time (ms)**	**C++'s time (ms)**
`INSERT`	410.4	3006.5
`SELECT`	43.7	246.4
`ALTER TABLE`	118.8	142.8
`UPDATE`	134.7	1317.8

Table 2: Comparison between Java and C++

C++ is faster theoretically, but due to the details of implementation, Java is actually faster. Therefore, it is proper to compare Java with PostgreSQL in the following experiments.

Comparison on basic operations

In this experiment, we first create a table containing the following attributes:

```
CREATE TABLE people(
    id integer NOT NULL PRIMARY KEY,
    first_name varchar(30),
    surname varchar(30) NOT NULL,
    born integer NOT NULL,
    died integer,
    gender char(1) NOT NULL DEFAULT '?'
);
```

After that, we insert 200000 records (randomly generated) into the table:

```
INSERT INTO people VALUES(1,'Ijtokawwnonj','Tgzmbxmzf',1868,1934,'F');
INSERT INTO people VALUES(2,'Zgpicfvfnffvb','Fmcxoaeueh',1880,1923,'F');
INSERT INTO people VALUES(3,'Ukmdykgxhdggjn','Tuytngedxiuqi',1886,1911,'M');
-- 199995 records omitted
INSERT INTO people VALUES(199999,'Kittquecptz','Sbyrvzsxl',1867,1949,'M');
INSERT INTO people VALUES(200000,'Scmuf','Qawwhzudy',1888,1944,'F');
```

Finally, we execute multiple query / modification commands to get a benchmark for each operation:

```
SELECT * FROM people WHERE first_name like '%ly%' or surname like '%ly%';
-- Requirement 2, token 'ly' is chosen instead of 'XXX' because it occurs relatively
frequent in names.

UPDATE people SET first_name = REPLACE(first_name, 'To', 'TTOO'); -- BAD method

UPDATE people SET surname = REPLACE(surname, 'To', 'TTOO'); -- BAD method
-- Two BAD implementations of requirement 3
-- We will discuss this issue in the comparison of UPDATE command

ALTER TABLE people ADD COLUMN age integer;

UPDATE people
    SET age = died - born
    WHERE died IS NOT NULL;

UPDATE people
    SET age = 2023 - born
    WHERE died IS NULL;
```

`INSERT` command

For PostgreSQL, the runtime for all `INSERT` operations is 17 seconds, i.e. the average runtime for each `INSERT` statement is 84 μs.

To reduce the fetching time, DataGrip **packs every 1000 consecutive `INSERT` commands into 1 batch** and send it to DBMS. For every 1000 `INSERT` commands, the average execution time is 18.655 ms, i.e. the average execution time of each `INSERT` statement is 18.655 μs (See raw data in Appendix 2).

For Java, we can operate the table directly without "connecting" to it, therefore the runtime is equal to execution time. By running the program, we can get the runtime of inserting 200000 records is 2106 ms, i.e. 10.53 μs per `INSERT` command.

Consider only execution time, Java is faster. However, in most cases, databases are too large for storing all data in cache.

At the end of our Java code, we use `Java.io.BufferedWriter` to save the table into a `.csv` file and the runtime of the process is 1307397 ms (nearly 22 minutes). **If we consider the total runtime of manipulating data and saving data to the hard disk**, the average runtime of one `INSERT` command is 6.537 ms, **which is much slower than PostgreSQL**.

`SELECT` command

The runtime for `SELECT` command in PostgreSQL is 171 ms. While in Java, the runtime becomes 13201 ms, which is much slower than that in PostgreSQL.

In our Java code, every time when there is a `SELECT` command, the program goes through all values in the specified attribute and do **regex matching**. Let C be the time complexity for each regex matching, then the time complexity of our code is $C \cdot \Theta(n)$, which is time-consuming.

In PostgreSQL, the DBMS **creates an index for each value in the table and store indexes in balanced trees**. Therefore, the time complexity is reduced to $C \cdot \Theta(\log n)$, which is much faster.

`DELETE` command

The runtime in `DELETE` command is mainly consumed by selecting some specified records out (deleting them won't have higher time complexity). Therefore, the result of testing `DELETE` command is similar to that of `SELECT` command.

`UPDATE` command

We firstly use the following command:

```
UPDATE people SET first_name = REPLACE(first_name, 'To', 'TTOO');
```

Under this command, the runtime is 13 s 280 ms, which is far too slow for one command. Then we notice that during this command, all the lines is affected even if it doesn't have substring `To`. Therefore, we change the command into:

```
UPDATE people
    SET first_name = REPLACE(first_name, 'To', 'TTOO')
    WHERE first_name LIKE '%To%';
```

This time, the runtime becomes 1 s 702 ms and 2956 rows in the table is affected. The runtime is still slow but much better. **This experiment shows the importance of using functions carefully to avoid unnecessary calculations**.

For Java, the runtime of `UPDATE` command beats PostgreSQL: Java costs only 135 ms in average for every command. This is mainly because DBMS have to verify the modified data to ensure correctness and security of the table, but our Java code skips this step (it just simply does the replacement on strings).

Other commands

Besides the commands mentioned above, we also tested commands including `ALTER`, `COMMENT`, `CREATE` and `DROP`. These commands are of low time complexity (because they don't go through all the records) and usually takes only a few milliseconds for each command. Therefore, we don't compare them strictly in this report.

Comparison on constraints

In this section, we will evaluate the impact of some keywords describing constraints, namely `(NOT) NULL`, `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY` and `CONSTRAINT`. Definitions of tables are shown below.

- Original

```
CREATE TABLE people(
    id integer,
    first_name varchar(30),
    surname varchar(30),
    born integer,
    died integer,
    gender char(1)
);
```

- `(not) null`

```
CREATE TABLE people(
    id integer,
    first_name varchar(30) NOT NULL,
    surname varchar(30),
    born integer NOT NULL,
    died integer,
    gender char(1)
);
```

- `PRIMARY KEY`

```
CREATE TABLE people(
    id integer PRIMARY KEY,
    first_name varchar(30),
    surname varchar(30),
    born integer,
    died integer,
    gender char(1)
);
```

- `UNIQUE`

```
CREATE TABLE people(
    id integer UNIQUE,
    first_name varchar(30),
    surname varchar(30),
    born integer,
    died integer,
    gender char(1)
);
```

- `FOREIGN KEY`

```
CREATE TABLE gender_info(
    gender char(1) PRIMARY KEY,
    full_info char(8)
);

INSERT INTO gender_info VALUES('M', 'Male');
INSERT INTO gender_info VALUES('F', 'Female');
INSERT INTO gender_info VALUES('?', 'Unknown');
```

```
CREATE TABLE people(
    id integer UNIQUE,
    first_name varchar(30),
    surname varchar(30),
    born integer,
    died integer,
    gender char(1),
    CONSTRAINT gender_fk
        FOREIGN KEY (gender)
        REFERENCES gender_info(gender)
);
```

- `CONSTRAINT`

```
CREATE TABLE people(
    id integer UNIQUE,
    first_name varchar(30),
    surname varchar(30),
    born integer,
    died integer,
    gender char(1),
    CONSTRAINT name_uq
        UNIQUE(first_name, surname)
);
```

After using the codes above to create different tables, we run the test before to evaluate the runtime difference caused by constraints.

Version	`INSERT` (s)	`SELECT` (ms)	`UPDATE` (ms)
Original	18	45	704.50
`(NOT) NULL`	18	31	927.00
`PRIMARY KEY`	18	12	1480.75
`UNIQUE`	24	25	1408.00
`FOREIGN KEY`	18	27	2594.67
`CONSTRAINT`	21	17	3185.25

Table 3: Runtime difference caused by constraints

Note:

1. The `SELECT` command in experiment of `PRIMARY KEY` and `UNIQUE` has been modified as:

```
SELECT * FROM people WHERE id % 666 = 233;
```

if we use

```
SELECT * FROM people WHERE first_name LIKE '%ly%' OR surname LIKE '%ly%';
```

then there won't be much difference with the original experiment.

2. Similarly, we use

```
UPDATE people SET gender = '?' WHERE gender LIKE 'M';
UPDATE people SET gender = 'M' WHERE gender LIKE 'F';
UPDATE people SET gender = 'F' WHERE gender LIKE '?';
```

to evaluate the efficiency difference due to `FOREIGN KEY`.

Explanations to the result:

- The speed of `INSERT` command is similar except for `UNIQUE` constraints (see experiment `UNIQUE` and `CONSTRAINT`) because it will produce an additional uniqueness detection.
- The speed of `SELECT` command is faster with constraints since PostgreSQL can generate better indexes if there are some guarantees for the data. The speed of `SELECT` command is even faster if the specified field has constraints like `PRIMARY KEY` and `UNIQUE`.
- The speed of `UPDATE` varies a lot relatively. In the experiment of `PRIMARY KEY` and `UNIQUE`, the runtime increases as PostgreSQL have to check the uniqueness of the modified data and may also adjust the indexes. The runtime goes even longer under constraints of `FOREIGN KEY` or `UNIQUE` of multiple attributes since they are harder to check.

The effect of constraints in `JOIN`

In this experiment, we construct three similar tables of `gender_info` respectively to evaluate the effect of constraints in `JOIN`.

`TABLE gender_info` 1: having no constraint.

```
CREATE TABLE gender_info(
    gender char(1),
    full_info char(8)
);
```

`TABLE gender_info` 2: having a `UNIQUE` constraint.

```
CREATE TABLE gender_info(
    gender char(1) UNIQUE,
    full_info char(8)
);
```

`TABLE gender_info` 3: having a `PRIMARY KEY` constraint.

```
CREATE TABLE gender_info(
    gender char(1) PRIMARY KEY,
    full_info char(8)
);
```

And the remaining part is the same:

```
INSERT INTO gender_info VALUES('M', 'Male');
INSERT INTO gender_info VALUES('F', 'Female');
INSERT INTO gender_info VALUES('?', 'Unknown');

CREATE TABLE people(
    id integer UNIQUE,
    first_name varchar(30),
    surname varchar(30),
    born integer,
    died integer,
    gender char(1)
);

INSERT INTO people VALUES(1,'Ijtokawwnonj','Tgzmbxmzf',1868,1934,'F');
INSERT INTO people VALUES(2,'Zgpicfvfnffvb','Fmcxoaeueh',1880,1923,'F');
INSERT INTO people VALUES(3,'Ukmdykgxhdggjn','Tuytngedxiuqi',1886,1911,'M');
-- 199995 records omitted
INSERT INTO people VALUES(199999,'Kittquecptz','Sbyrvzsxl',1867,1949,'M');
INSERT INTO people VALUES(200000,'Scmuf','Qawwhzudy',1888,1944,'F');
```

```
SELECT count(*)
    FROM people JOIN gender_info ON people.gender=gender_info.gender;
```

We then execute the three `.sql` respectively for 5 times, measuring their runtime (in ms):

Experiment	Test 1	Test 2	Test 3	Test 4	Test 5	Average
No constraint	136	153	165	195	190	167.8
`UNIQUE`	107	149	151	145	110	132.4
`PRIMARY KEY`	55	79	112	91	80	83.4

Table 4: The effect of constraints in `JOIN`

The experiment tells that both `UNIQUE` and `PRIMARY KEY` constraint can promote the efficiency of `JOIN` command. The `PRIMARY KEY` constraint can have great acceleration on the command while the `UNIQUE` constraint has much smaller impact.

Here we implement `Join.java` to give an explanation to the result:

- A simple but slow method: enumerating all elements in the Cartesian product of the sets of rows in each table.

```
for (ArrayList<String> record1: table1.array) {
    for (ArrayList<String> record2: table2.array) {
        if (record1.get(ruleColumn1).equals(record2.get(ruleColumn2))) {
            ArrayList<String> record = new ArrayList<>();
            record.addAll(record1);
            record.addAll(record2);
            table.array.add(record);
        }
    }
}
```

- A better method: sort the records by the attribute used for joining.

If we know that the attribute used for joining in at least one table is unique, then the values of pointers `pnt1, pnt2` **monotonically increase** if we sort the records by the attributes. Therefore, we can use this method to **reduce the time complexity from $\Theta(n_1 n_2)$ to $\Theta(n_1 \log n_1 + n_2 \log n_2)$**.

```
// sort table1.array by ruleColumn1
int finalRuleColumn1 = ruleColumn1;
table1.array.sort((ArrayList<String> record1, ArrayList<String> record2) -> {
    if (record1.get(finalRuleColumn1) == null) {
        if (record2.get(finalRuleColumn1) == null) {
            return 0;
        }
        return -1;
    }
    if (record2.get(finalRuleColumn1) == null) {
        return 1;
    }
    return record1.get(finalRuleColumn1).compareTo(record2.get(finalRuleColumn1));
});
// sort table2.array by ruleColumn2
int finalRuleColumn2 = ruleColumn2;
```

```
table2.array.sort((ArrayList<String> record1, ArrayList<String> record2) -> {
    if (record1.get(finalRuleColumn2) == null) {
        if (record2.get(finalRuleColumn2) == null) {
            return 0;
        }
        return -1;
    }
    if (record2.get(finalRuleColumn2) == null) {
        return 1;
    }
    return record1.get(finalRuleColumn2).compareTo(record2.get(finalRuleColumn2));
});
// merge table1.array and table2.array
int pnt1 = 0, pnt2 = 0;
while (pnt1 < table1.array.size() && pnt2 < table2.array.size()) {
    ArrayList<String> record1 = table1.array.get(pnt1);
    ArrayList<String> record2 = table2.array.get(pnt2);
    if (record1.get(ruleColumn1).equals(record2.get(ruleColumn2))) {
        ArrayList<String> record = new ArrayList<>();
        record.addAll(record1);
        record.addAll(record2);
        table.array.add(record);
        if (uniqueTable == 1) {
            ++pnt2;
        }
        else {
            ++pnt1;
        }
    }
    else if (record1.get(ruleColumn1).compareTo(record2.get(ruleColumn2)) < 0) {
        ++pnt1;
    }
    else {
        ++pnt2;
    }
}
```

Moreover, if we don't have `UNIQUE` or `PRIMARY KEY` constraint on either attributes, we can still sort the records and do segmentation by the values of the attributes used for joining. Therefore, the runtime measured in the experiment with no constraint and `UNIQUE` constraint is close.

- If the constraint on one of the attributes (used for joining) is `PRIMARY KEY`, PostgreSQL can **make better indexes** for the records and the runtime can be greatly shortened due to **faster positioning on B+ trees** than sorting all records.

Comparison on data types

There are many similar groups of data types in PostgreSQL, `smallint`, `integer` and `bigint` for integers, `decimal` and `numeric` for high precision numbers, `real`, `double precision` for real numbers (may have precision loss), `char(n)`, `varchar(n)` and `text` for strings.

In this experiment, we consider a few situations and compare the efficiency of using similar data types for implementation.

Storing real numbers

In this experiment, we compare the efficiency of data types `integer`, `bigint`, `numeric(12, 2)` and `double precision`. We start from the following code:

```sql
CREATE TABLE people(
    id integer NOT NULL PRIMARY KEY,
    first_name varchar(30),
    surname varchar(30) NOT NULL,
    born integer NOT NULL,
    died integer,
    gender char(1) NOT NULL DEFAULT '?'
);

-- INSERT commands omitted.

SELECT * FROM people WHERE born BETWEEN 1987 AND 2004;

UPDATE people
    SET born = born - 10;

ALTER TABLE people ADD COLUMN age integer;

UPDATE people
    SET age = died - born
    WHERE died IS NOT NULL;

UPDATE people
    SET age = 2023 - born
    WHERE died IS NULL;
```

Then we change the type of `id`, `born`, `died` and `age` to `bigint`, `numeric(12, 2)` and `double precision` respectively and test the runtime of the `.sql` file.

Data type	`INSERT` (s)	`SELECT` (s)	`UPDATE` (s)	**Overall** (s)
`integer`	18.888	0.012	2.759	21.659
`bigint`	20.044	0.057	2.720	22.821
`numeric(12, 2)`	19.744	0.037	3.280	23.061
`double precision`	20.207	0.018	2.062	22.287

Table 5: Runtime difference in numeric data types

Explanations to the result:

- As `bigint` has a greater range than `integer`, it takes more time to do operations like `INSERT`, `SELECT` and `UPDATE` (there are more things to check and calculate).
- As `numeric` has higher precision than `double precision`, it is also slower in most operations.
- From the table above, we can conclude that: if a data type can store more digits or have higher precision, then it may have higher time complexity (actually also larger memory footprint). Therefore, if it is acceptable to use "inferior" data type for an attribute, then we shouldn't use "superior" ones in order to reduce time and memory cost.

Storing strings

In this experiment, we set the data type of `first_name` and `surname` to `char(30)`, `varchar(30)` and `text` respectively and compares the runtime.

Moreover, to show the differences on `SELECT` and `UPDATE` command, we change the commands to:

```sql
SELECT * FROM people WHERE first_name LIKE '%ly%' OR surname LIKE '%ly%';

UPDATE people
    SET first_name = replace(first_name, 'To', 'TTOO')
    WHERE first_name LIKE '%To%';
UPDATE people
    SET surname = replace(surname, 'To', 'TTOO')
    WHERE surname LIKE '%To%';
```

Data type	`INSERT` (s)	`SELECT` (s)	`UPDATE` (s)	**Overall** (s)
`char(30)`	15.897	0.044	3.045	18.986
`varchar(30)`	27.613	0.095	2.249	29.957
`text`	33.474	0.054	2.407	35.935

Table 6: Runtime difference in string data types

Explanations to the result:

- `char()` can store a string of fixed length, e.g. lengths of all `char(30)` strings are 30 no matter how long the original strings are. Though this data type may have **larger memory footprint**, it is **easier for DBMS to align** between different records, which can make `INSERT` and `SELECT` operations faster. For `UPDATE` command, however, `char()` is **slower due to the padding process** after each modification.
- `varchar()` can store a string of a fixed maximum length (it won't pad spaces like `char()`). Since it has a varying length, it is harder for DBMS to align, causing longer runtime in operations like `INSERT` and `SELECT`.
- `text` is the most generalized string data type which has a maximum length of 1GB (unreachable in most cases). Though it can **avoid the problem of exceeding maximum length**, it is also the **slowest**.

Optimization based on data

We also conduct an experiment where we replace our records with records in filmdb.sql. The runtime of both dataset is between 1 to 2 seconds, which doesn't show notable optimization based on data. The result may due to the dataset being too small.

Conclusion

1. In this project, we implemented a file-I/O-style DBMS based on Java, using techniques including **string / expression handling, regex matching, hash sets**, etc. In our program, all operations are performed in RAM, which is fast but unrealistic in real big data cases. **DBMS makes a good balance between fetching time and execution time, using hard disk and using cache, which has a good performance in handling big databases**.
2. When designing a table, we should **choose proper data types**, usually data types having **more restrictions** performs **better in efficiency**.
3. It's suggested to **add appropriate constraints** to tables as DBMS can make better indexes and have a higher speed in tasks like `SELECT`, `UPDATE` and `JOIN`.
4. We should **be careful when using functions** in operations like `SELECT` and `UPDATE` to avoid unnecessary calculations.

References

- Slides of Chapter 1-4 in Principles of Database Systems (H) Fall 2023, SUSTech (by Prof. Yu):
  - Chapter 1: bb.sustech.edu.cn/bbcswebdav/pid-407766-dt-content-rid-14826703_1/courses/CS213-30010154-2023FA/CS213-DB-01%281%29.pdf
  - Chapter 2: bb.sustech.edu.cn/bbcswebdav/pid-412680-dt-content-rid-14869756_1/courses/CS213-30010154-2023FA/CS213-DB-02.pdf
  - Chapter 3: bb.sustech.edu.cn/bbcswebdav/pid-414194-dt-content-rid-14953780_1/courses/CS213-30010154-2023FA/CS213-DB-03.pdf
  - Chapter 4: bb.sustech.edu.cn/bbcswebdav/pid-415032-dt-content-rid-14992719_1/courses/CS213-30010154-2023FA/CS213-DB-04.pdf
- filmdb.sql: https://bb.sustech.edu.cn/bbcswebdav/pid-410806-dt-content-rid-14836997_1/xid-14836997_1

Appendix

Appendix 1: Raw data of comparison between Java and C++

No.	INSERT(ms)	SELECT (ms)	ALTER (ms)	UPDATE (ms)
1	409	31	157	93
2	403	47	78	130
3	413	47	156	100
4	407	63	101	160
5	408	31	109	154
6	426	47	126	147
7	391	46	94	163
8	417	52	146	110
9	423	32	109	144
10	407	41	112	146
Average	410.4	43.7	118.8	134.7

Table 7: Raw data of Java's runtime

No.	INSERT(ms)	SELECT (ms)	ALTER (ms)	UPDATE (ms)
1	2937	250	187	1335
2	3018	250	140	1347
3	2844	235	141	1297
4	2953	234	141	1312
5	2921	250	125	1344
6	2890	235	140	1297
7	2859	250	141	1296
8	2843	250	141	1312
9	3715	250	131	1295
10	3085	257	141	1343
Average	3006.5	246.1	142.8	1317.8

Table 8: Raw data of C++'s runtime

Appendix 2: Raw data of runtime for SQL's `INSERT` statements

16	15	15	14	28	28	17	17	16	18
17	15	14	20	17	17	25	23	17	15
14	16	14	16	15	16	14	15	15	15
15	16	14	15	15	14	15	14	15	14
16	14	15	48	16	15	14	15	14	20
23	26	24	17	18	15	15	15	14	17
15	15	19	21	22	24	23	24	17	17
14	14	14	29	20	16	19	32	29	31
15	16	14	16	14	15	18	17	24	24
15	14	15	15	15	19	16	17	17	15
14	19	15	22	21	14	14	19	15	16
45	16	15	67	15	15	13	14	15	17
15	26	22	23	19	15	15	15	15	16
15	19	22	25	19	23	16	15	16	16
20	20	30	31	45	25	15	18	15	23
27	20	16	20	24	14	15	14	15	15
15	29	22	26	19	15	14	14	23	16
13	19	15	17	16	16	15	17	40	17
15	20	21	22	24	31	30	19	15	14
14	18	16	20	15	16	15	19	27	14

Table 9: Raw data of PostgreSQL's runtime for 1000 `INSERT` statements

Appendix 3: Source codes

All source codes are submitted with this report, including `Attribute.java`, `Join.java`, `Main.java`, `Table.java`, `UniqueValueSet.java`, `Utils.java`, `JavaSQLite.java`, `CppSQLite.cpp` (the last two codes are used for evaluating the difference caused by different programming language).