CS307 Project1 Report

Basic information

安钧文 12012109　张海涵 12012222　lab session：7-8 Tue.

contributions: 50% 50%

Task 1: 安钧文，张海涵

Task2：create table 安钧文

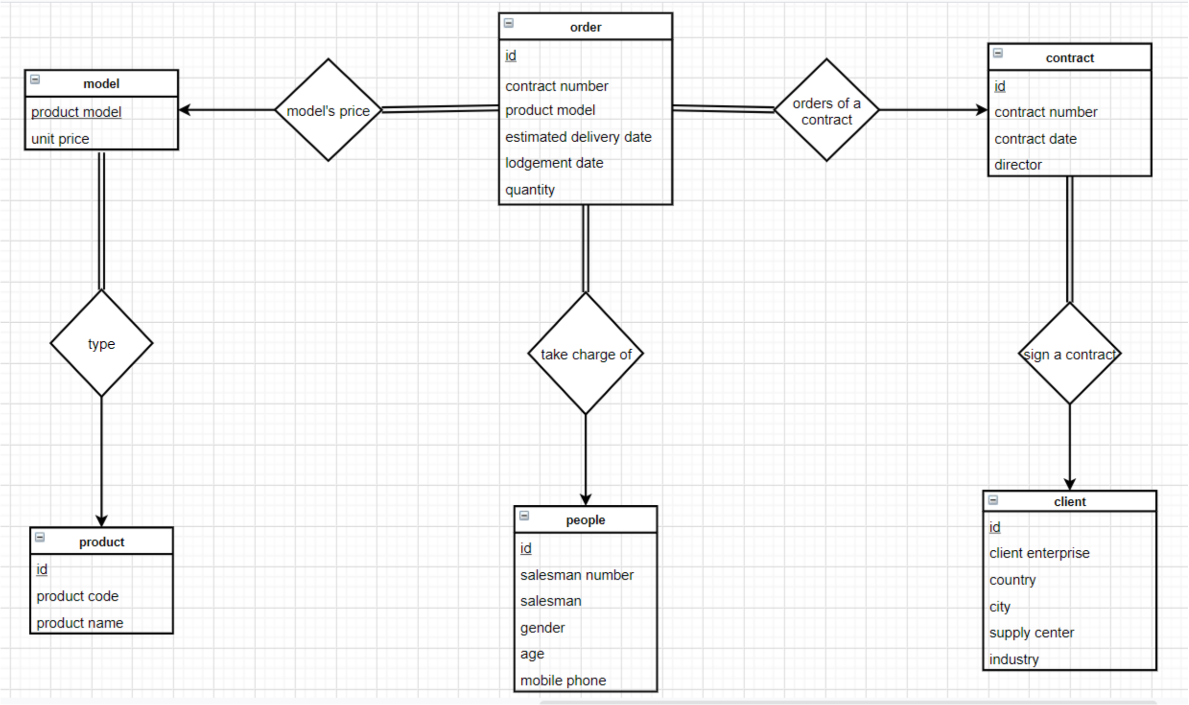
Task3：import data 安钧文， bonus 安钧文 张海涵

Taks4：DBMS 安钧文， File I/O 张海涵， indexing 安钧文，user privilege 张海涵，comparison with python 安钧文

report：安钧文，张海涵

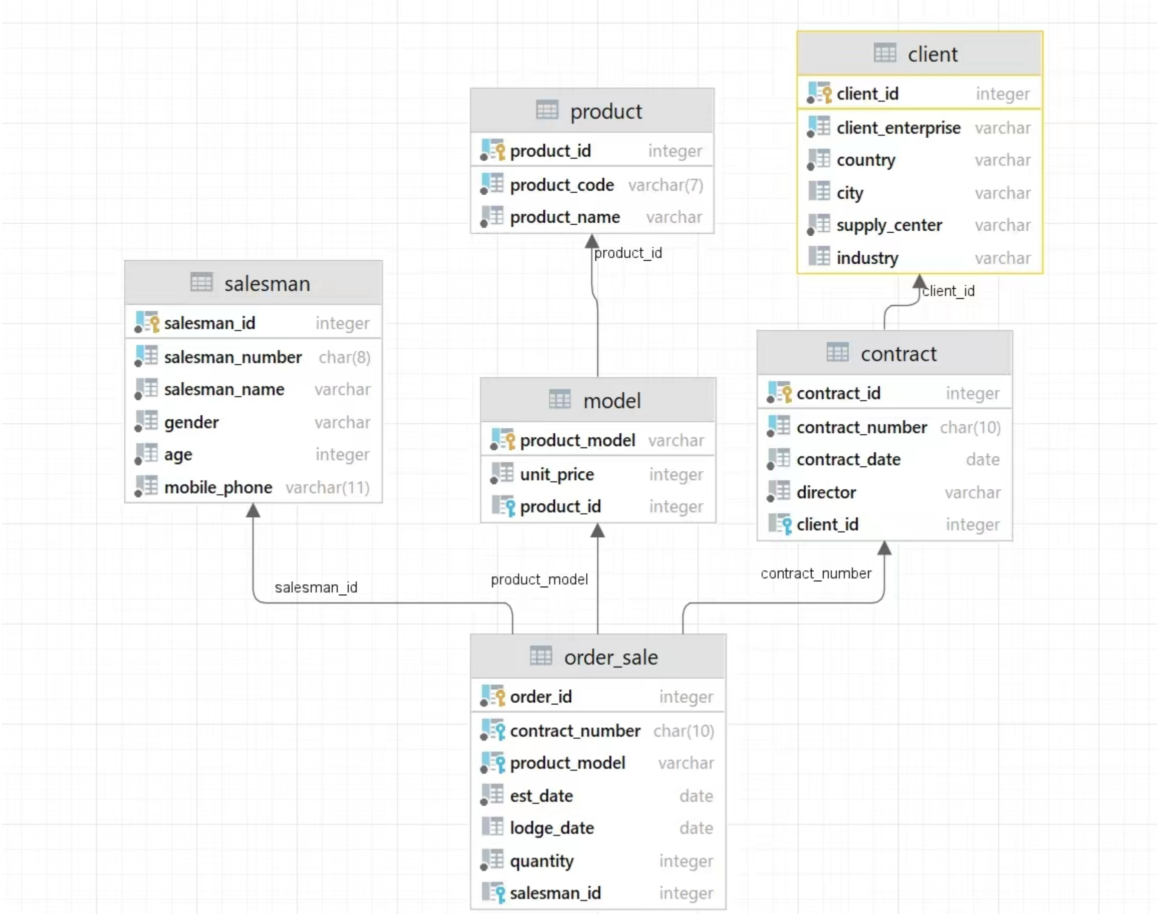
Task 1: E-R Diagram

Plotting tool: 迅捷画图



Task 2: Database Design

1) E-R diagram generated by DataGrip



2) Describe table design

According to the E-R diagram, we designed six entity set: `salesman`, `contract`, `model`, `order_sale`, `product` and `client` .

Firstly, we divided the data into four group: salesman, product, contract and client.

- For group salesman, we can easily group attributes: `salesman`(name), `salesman_number`, `gender`, `age` and `mobile_phone` . Considering about `salesman_number` is the identifier like student ID, so set `salesman_number` unique not null.
- For group product, noticed that one `product_code` has only one `product_name` and many `product_model`, so separated table `product` contains `product_code` and `product_name`, set `product_code` not null unique, and table `model` to store the specific message like model's `unit_price`. So the one-to-many relationship can be founded. We use the foreign key in model to constrain the relation.
- For group client, note that one `client_enterprise` has its own `supply_center`, `country`, `city` and `industry`, so we grouped them. In real life, client enterprise is unique, so set it not null unique.
- For group contract, noticed that one contract has one and only one `contract_date`, `director` and `client_enterprise`, but can contain many `product_model`s, with different `estimated_delivery_date` 、 `lodgement_date` and `quantity`. About this two kind of information, we create table `contract` to store the contract public information, set `contract_number` unique not null. And table `order_sale` to store the specific information. Considering that the info in `order_sale` depend on contract and its model, so we set `contract_number` and `product_model` as composite primary key. The relation between `contract` and `order_sale` is one-to-many, we use foreign key in order_sale to constrain the unique attributes.

Then describe the relations in our database:

- The relation between `product` and `model` is one-to-many.
- The relation between `salesman` and `order_sale` is one-to-many.
- The relation between `client` and `contract` is one-to-many.
- The relation between `model` and `order_sale`, `contract` and `order_sale` is one-to-many.

In many side, there are always foreign keys to constraint the relation.

Notes:

- Test for NF:

  It must satisfy 1NF because the info in `contract_info.csv` is already atomic.

  For 2NF, just need to test table `order_sale`. The attributes in this table is part of single line info in `contract_info.csv`. And the contract_number and product_model determined the single line. So it satisfies.

  The table design is also satisfy 3NF.
- In every table(except table `model`), set a serial primary key `id` to simplify the reference.

Task 3: Data Import

1) Script Code:

The script is modified from `AverageLoader.java` in the given demo. Here we only provide essential parts of the script. Some parts of code are omitted for simplification.

Method `openDB()`

This is the method which connects to our database and declares `insert...values` SQL statements for data importing.

```
private static void openDB(String host, String dbname,
                           String user, String pwd) {
    Class.forName("org.postgresql.Driver");
    String url = "jdbc:postgresql://" + host + "/" + dbname;
    Properties props = new Properties();
    props.setProperty("user", user);
    props.setProperty("password", pwd);
    con = DriverManager.getConnection(url, props);
    if (verbose) {
        con.setAutoCommit(false);
    }
    stmtClient = con.prepareStatement("insert into client (client_enterprise,
country, city, supply_center, industry) values(?,?,?,?,?)");
}//other insert statements omitted here
```

Method `closeDB()`

This is the method that closes the database after successful imports.

```
private static void closeDB() {
    stmtClient.close();
    con.close();
    con = null;
}
```

Method `loadData()`

This is an overloaded method designed for inserting different number of columns. Here we only show one of them.

```
private static void loadData(String a, String b, PreparedStatement stm)
    throws SQLException {
    stm.setString(1, a);
    stm.setString(2, b);
    stm.executeUpdate();
}
```

Method `main()`

This is the main method that we give parameters, do the imports, and measure the efficiency.

```
public static void main(String[] args) {
    String fileName = null;
    boolean verbose = false;
    //configure database parameters
    Properties defprop = new Properties();
    defprop.put("host", "localhost"); defprop.put("user", "checker");
    defprop.put("password", "123456");
    defprop.put("database", "contract_project");
    Properties prop = new Properties(defprop);
    try (BufferedReader infile = new BufferedReader(new FileReader(fileName))) {
        String line;
        String[] parts;
        String a, b, c, d, e, f; //column names
        int cnt = 0;

 openDB(prop.getProperty("host"),prop.getProperty("database"),prop.getProperty("
user"), prop.getProperty("password"));
        while ((line = infile.readLine()) != null) {
            parts = line.split(",");
            if (parts.length > 1) {
                a = parts[0];
                b = parts[1];
                loadData(a, b, stmtOrder);
                cnt++;
            }}
        con.commit();
        closeDB();
    }}
```

2) Script Descriptions

Prerequisites:

`contract_info.csv` should be filtered in advance. First, we only want the columns that matches the target table. Second, the rows in `unique` column must not contain duplicate values. A filtered `.csv` file is needed for the script to function properly.

For example, we need to import data to table `product`. First, we create a new `.csv` file that contains only `product code` and `product name`. Next, since we set `product code` as `unique`, we need to filter out all duplicate rows in the column. After that, we are ready to use the script.

Note that to reach the requirements, the user can either use a script written in java (code in zip file) or simply use Microsoft Excel's built in functions.

To distinguish from further tests, all `.csv` files used here does not include a serial id column, file names are `xxx_wout.csv` (`model` table itself doesn't have a id column, so this particular file won't be distinguished).

Steps:

1. Configure database parameters in `main`
2. Declare SQL statements in `openDB`, here we use `stmtProduct` (insert into table `product`) as an example.
3. Enter program arguments in "Edit Configurations". It should be `-v filename`. For example, `-v product_wout.csv`
4. Configure arguments to pass into `loadData` in `main`. The last argument should be your SQL statement, the rest are data parts to be imported. If successfully imported, you should see its speed and number of records imported,otherwise an exception.

   For example, `product_wout.csv` has 2 columns, thus `parts` is a String array of length 2. `parts[0]` is `product code` and `parts[1]` is `product name`(matched with the order in your .csv file). Call method `loadData(parts[0],parts[1],stmtProduct)`.

Cautions:

1. While passing arguments into `loadData`, please be aware of their order. Their order match the order of `?` in the SQL statement passed in.
2. Please put all the required files in the project directory.
3. If there exists foreign keys in a table to be imported, you should also include a `unique` column of the referenced table in your target .csv file.

   For example, include the referenced `client enterprise` in `contract_wout.csv` for table `contract`'s foreign key.
4. This script can only import data of one table at a time.

3) Import Data with Python

We implemented three methods to import data using Python 3.9.6. Methods 1 and 2 use `psycopg2`, method 3 uses `pandas` and `sqlalchemy`. All tests use table `order_sale` and data `order.csv`, full script is in the zip file.

1. Use `execute()`

This method is similar to the script described above. We use `execute()` function to insert data, which inserts data one row at a time. `commit()` after all inserts are executed.

2. Use `execute_batch()`

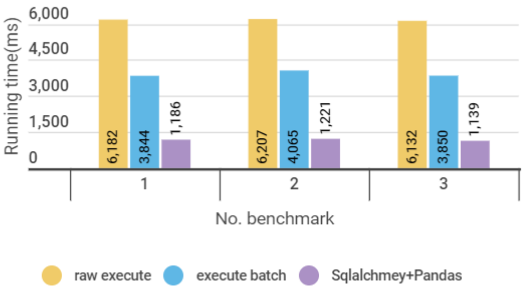
This method is similar to `GoodLoader.java` in the provided demo, which execute groups of statements at a time, improving the performance. `commit()` after calling `execute_batch()`. Note that before calling `execute_batch()`, all data rows needs to be converted to tuples, and appending these tuples into a `args_list` to pass in to `execute_batch()`. This preprocessing stage could cause reduced speed, here we only used file object and use iteration to process. There may exist other more optimized approach.

3. Use `create_engine` and `to_sql()`

This method uses two extra libraries. `sqlalchemy` is used to create a engine that connect to database, and `pandas` is used to import data to a table.

One critical flaw is that it will replace the original table and change the columns' data types according to the data types generated in `pandas`'s dataframe. In our example, `int` will be converted to `bigint`, and other types will be converted to `text`.

Test results:

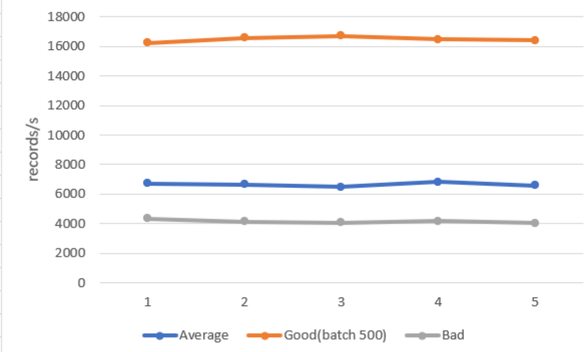


From the figure, `execute_batch()` is much faster than `execute()`, which shows that batch loading can increase performance by reducing the number of server roundtrips. Although method 3 is much better in performance, but it's destructive to the tables, so users should treat it with care. Another interesting fact is that method 1 and the original java script uses similar methods (execute one at a time, commit at last), but java is slightly faster than python.

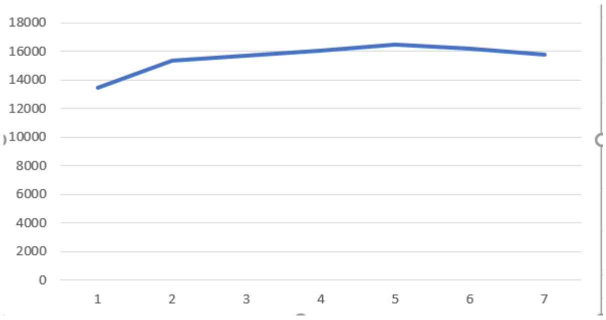
4) Other ways to import and optimize

Based on `GoodLoader.java` and `BadLoader.java` in demo, we modified it and import data. In `GoodLoader.java`, we use Batch to commit several statements in a time.

Then we test 5 times of the three methods, through the result, we can see the `GoodLoader` is about 2.5 times faster than `AverageLoader`, and about 4 times faster than `BadLoader`.



Try to find the optimized batch size, we change the size to 10， 50， 100， 200， 500， 1000 ， 2000, in each size we test for 5 times and calculate the average result. Then draw the diagram：



Through the diagram, we can see the fifth test, which batch size is 500, it reaches a local maximum. So we choose the 500 as the batch size.

Task4 - Compare DBMS with File I/O

1) Test environment

Hardware specification:

CPU: Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz

RAM: 16.0 GB

Disk: 1TB SATA/600 HDD

Software specification:

Operating System: Windows 10 Professional 21H2

DBMS: Postgresql 14.2

Programming languages: Java (version 1.8.0_321) , Python (version 3.9.6)

Development environment:

1. IDE: IntelliJ IDEA 2021.3.2 Ultimate Edition for Java development, PyCharm 2021.3.2 Ultimate Edition for Python development.
2. Libraries: `postgresql-42.2.5.jar` and `java.io.*` used for Java development, `psycopg2`, file object,`pandas 1.4.2` and `sqlalchemy` for Python development.

Data visualization tool: Infogram

2) Test data

DBMS:

In this task, we mainly used table `order_sale` to experiment. In some multi-table query tests, other tables were also used. Here we only provide DDL of table `order_sale`, other tables' DDL are included in zip file. We added foreign key constraint after the table is created using `alter table`.

```
create table order_sale
(
    order_id        serial primary key,
    contract_number char(10) not null,
    product_model   varchar  not null,
    est_date        date     not null,
    lodge_date      date,
    quantity        integer  not null,
    salesman_id     integer  not null,
    unique (contract_number, product_model)
);
alter table order_sale
    add constraint model_fk foreign key (product_model) references model
(product_model);
alter table order_sale
    add constraint contract_fk foreign key (contract_number) references contract
(contract_number);
alter table order_sale
```

```
    add constraint sm_fk foreign key (salesman_id) references salesman
(salesman_id);
```

Data file:

In this task, we directly use the export tables transfer to `.csv` as the file document. And mainly use `order.csv` for test.

3) Script Description

DataManipulation.java:

An interface which includes the methods that we will use. Here's a list of methods implemented:

```
public void openDatasource();
public void closeDatasource();
//single table queries
public String allOrders();
public String findOrdersByModel(String model);
public String findOrdersByQuantity(int min, int max);
//single table aggregate function queries
public int countAllContracts();
public String maxQuantityContract();
public String countOrdersByModel(String model);
public String mostOrdersContract();
//multiple table queries
public String countSalesmanOrders();
public String findOrdersByPrice(int min,int max);
public String findContractClientIndustry(String industry);
//insert
public void addOneOrder(String str);
//delete
public void deleteOrderByID(int id);
//update
public void updateQuantityByID(int quantity,int id);
```

DatabaseManipulation.java:

The script that conduct data manipulations using Database API. Method `openDatasource()` and `closeDatasource()` are used to connect and disconnect from the database. Other methods are used to manipulate data using SQL statements. After each `selection` query, result set is built into strings which are ready to print. While testing, the tester can also choose to print the query result to make sure the query fetch correct results.

For example, method `allOrders()` corresponds to `select * from order_sale`, and it returns every row and every column in `order_sale`.

Methods with parameters are also included, for example, `findOrdersByModel(String model)` corresponds to `select contract_number, product_model from order_sale where product_model like ?` , parameter model is passed to replace `?` .

FileManipulation.java:

The script that conduct data manipulations using file API.

In file IO, we don't need to consider the time cost in `openDatasource()` and `closeDatasource()`. Methods almost use many string functions like `split()`, `substring()`, use `hashmap` and `list` to record data and bind different table in multiple query.

Client.java:

The program that runs benchmark. It creates instances of `FileManipulation` and `DatabaseManipulation`, and benchmark them separately by calling `benchmarkFM()` and `benchmarkDM()`. `System.currentTimeMillis()` is used to get current time.

```
FileManipulation fm = new FileManipulation();
DataManipulation dm = new DatabaseManipulation();
benchmarkDM(dm);
benchmarkFM(fm);
```

4) Comparative analysis

Here we conducted comparative analysis between DBMS and file IO of four types of operation: `select`, `insert`, `update`, `delete`. We'll compare them on statement level. **Time spent on opening and closing database are not counted.**

Test results may differ depending on the testing environment and implementation.

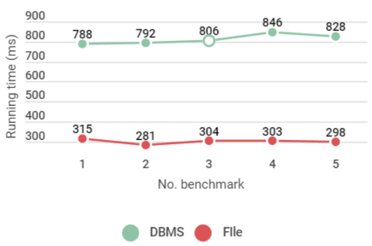
`Select`:

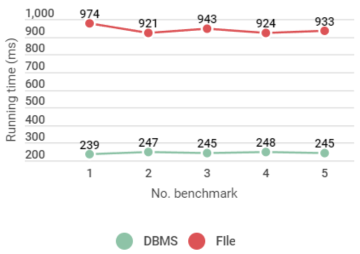
10 different `select` queries will be analyzed. Each type of query will be executed repeatedly for more than 10 times to get a clearer time comparison. Specific number of executions are stated below (`×10` means 10 repeated executions). Additionally, each benchmark will be run 5 times to make test result more robust and plausible. All values are in milliseconds (ms).

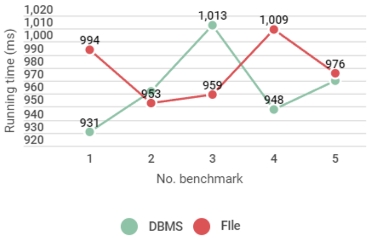
Single table selections:

```
-- single table selections
1. select * from order_sale -- ×10
2. select contract_number,product_model from order_sale where product_model like
?
-- "Mp437","YubaR4","TvBaseR1","Mp437%","YubaR4%" passed to ?, each parameter
×10, ×50 in total
3. select contract_number, product_model, quantity from order_sale where
quantity between ? and ?
-- (100,500) and (500,900) passed to (?,?), each parameter ×25, ×50 in total
```

Below are result figures of select statements 1 to 3 organized from left to right.







Observations and thoughts:

1. While selecting all rows in a table (statement 1), file IO is faster than DMBS. This could be due to implementation. While printing query result, it need to go through the result set and again and append each column to string, but this step is done using file IO during query, which means DBMS is spending twice the time of file IO to query and print.
2. While selecting specific rows using `like`, DBMS is faster than file IO. Different from selecting all rows, result set is much smaller, thus printing result takes much less time here. Additionally, there could be optimization in Postgres described here, where as file IO must iterate all rows to produce the results.

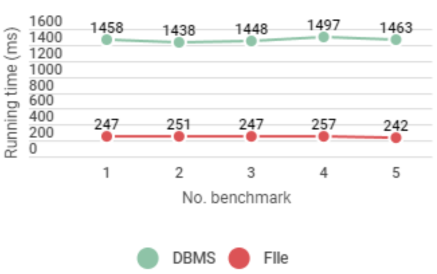
   > When all possible execution plans have been generated, the optimizer searches for the least-expensive plan. Each plan is assigned an estimated execution cost. Cost estimates are measured in units of disk I/O.
   >
   > After choosing the (apparently) least-expensive execution plan, the query executor starts at the beginning of the plan and asks the topmost operator to produce a result set. Each operator transforms its input set into a result set.

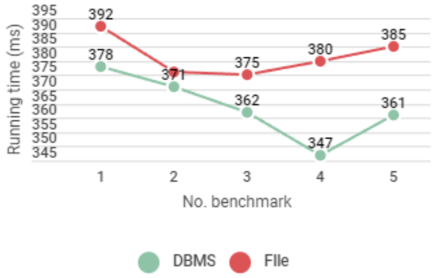
3. While performing range queries, DBMS and file IO have similar efficiency, again, this could be influenced by result set size of DMBS, and the extra time needed to print the results. In this experiment, the result set should be about half the size of all rows. Further tests on range query and result set are shown below.
4. File IO takes similar amount of time in test 2 and 3, because it just iterate all rows and print the queried rows, with no extra printing time needed.

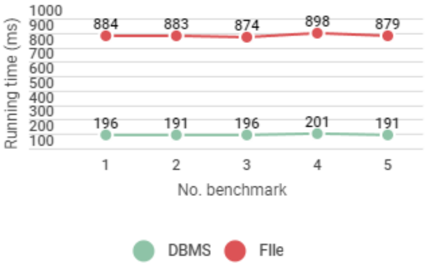
Single table aggregate function selections:

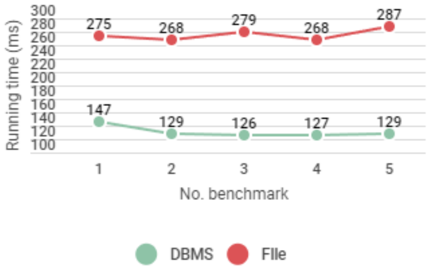
```
-- single table aggregate function selections
4. select count(distinct contract_number) cnt from order_sale -- ×10
5. select max(quantity), contract_number from order_sale group by
contract_number -- ×10
6. select count(*) from order_sale where product_model = ? -- "YubaR4","Mp437"
passed, each ×25, total ×50
7. select max(cnt) m, contract_number from (
                select count(*) cnt, contract_number from order_sale group by
contract_number  ) o_cnt group by contract_number order by m desc limit 1 -- ×10
```

Below are result figures of select statements 4 to 5 in the first row and 6 to 7 in the second row, both organized from left to right.









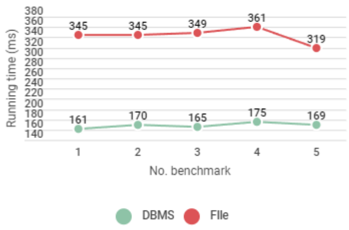
Observations and thoughts:

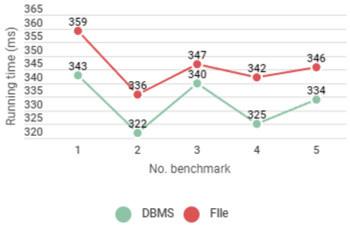
1. `distinct` keyword makes DBMS much slower in queries (test 4) than file IO. According to this page, `distinct` sorts the result set and filters out the duplicate records. In file IO, Hashmap is used to filter out duplicate data, which is much more efficient, as is shown in the figure.
2. `max()` has similar efficiency in both file IO and DBMS, but DBMS is slightly faster. File IO used Hashmap and Arraylist to filter and sort the data, which implies that Postgresql could also use similar technique to find max value.
3. DBMS is more than 4 times faster than file IO in test 6, and twice as fast as file IO in test 7 while performing subqueries, which again indicates Postgres' optimization query strategy.

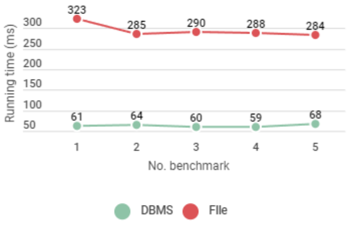
Multiple table selections:

```
-- multiple table selections
8. select count(*), o.salesman_id from order_sale o
    join salesman s on o.salesman_id = s.salesman_id
    group by o.salesman_id order by salesman_id asc -- x10
9. select contract_number,quantity,m.product_model,unit_price from order_sale o
    join model m on m.product_model = o.product_model
    where m.unit_price between ? and ? -- (400,500) x10
10. select o.contract_number, product_model, client_enterprise from order_sale o
    join (select * from contract c1
            join client c2 on c1.client_id = c2.client_id) client
    on o.contract_number = client.contract_number where industry = ? -- x10
```

Below are result figures of select statements 8 to 10 organized from left to right.







Observations and thoughts:

1. Generally, DBMS is faster than file IO while doing multi-table queries. As stated here, optimization could happen while performing these queries.

> If the query involves two or more tables, the planner can suggest a number of different methods for joining the tables.

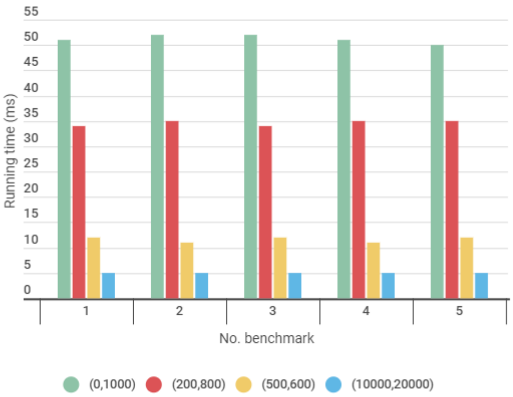
2. Test 9 involves `between and` keyword, still efficiency could be influenced by result set size.

Further experiments on result set size

The below query are performed with different parameters, 10 times for each set of parameter.

```
select contract_number, product_model, quantity from order_sale where quantity
between ? and ? -- (0,1000),(200,800),(500,600),(10000,20000)
```

Here's the result:



Now we can clearly see that result set size do have an influence on efficiency of DBMS. **This result only applies to SQL statements in java, and when in need of printing the result set**.

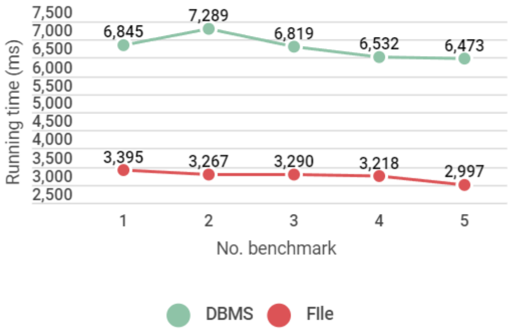
`Insert`:

We have prepared 25000 rows of new data to insert into the original order_sale table. We will use the following statement, and repeat it for 25000 times, each corresponding to a row of new data. We have ensured `contract_number` and `product_model` will remain unique so that they can be inserted into the table. The data and data generation script are in zip file.

Both methods' time count start after reading in the data and end when there's no more data in buffered reader.

```
insert into order_sale (order_id, contract_number,
product_model,est_date,lodge_date,quantity,salesman_id) values (?,?,?,?,?,?,?);
```

Results are shown below:



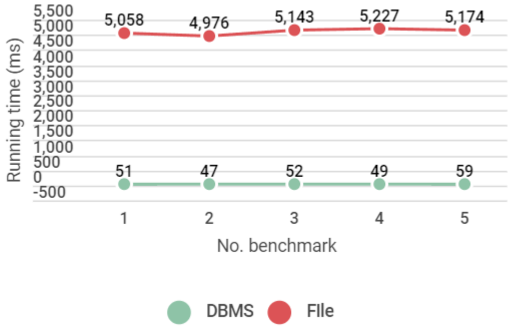
DBMS takes twice the time of file IO. This could be because DBMS requires extra steps to load data for every column (the parameters in `values()`), while file IO only has to write in the line directly.

`Update`:

We will update 100 rows' quantity attribute to 5000 using the below statement.

```
update order_sale set quantity = ? where order_id = ? -- order_id are from 1 to 100
```

Results are shown below:



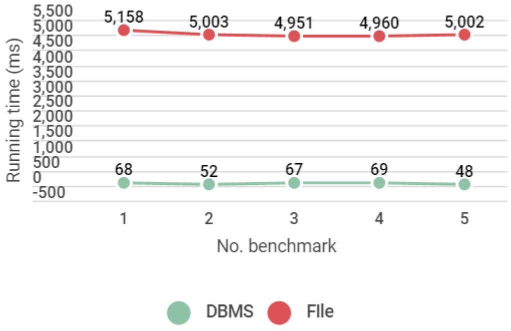
DBMS is more faster than File IO. This could be because the test statement updates a single line. File IO needs to read and rewrite after called every time. So when doing a large range of update, the more read and rewrite time will waste much times. But if we change the statement to find a range of `order_id` in a time, it will be more faster because it just need one time read and rewrite.

`Delete`:

Similar to `insert`, we will delete the data 100 rows at a time. The deleted rows are newly inserted in `insert` test.

```
delete from order_sale where order_id=? -- benchmark will run 5 times, so order_id are from 50001 to 50500
```

Results are shown below:



The result could be due to similar reason as `Update` because of the constraint of statement.

Note that all additional tests below use original `order_sale` table (without 25000 new columns or updated quantity).

5) Database indexing and file IO

Index types

According to document, there are multiple index types in Postgres, and the user can decide which index type to use with `using` keyword. Here, we will only focus on B+ tree index implementation.

B+ tree is a type of balanced binary search tree that contains keys of stored data. It's especially useful in database systems or file systems, as it can greatly reduce time spent on disk IO. Different from B-tree (which is used in Postgres), B+ tree is more efficient in doing range queries since the leaf nodes are doubly linked, so that it won't need to search from the root again and again.

In this project, we will compare the efficiency of different statements in DBMS with or without index, and in file IO with or without B+ tree optimization.

B+ tree implementation

We have included a basic B+ tree implementation in java, which can perform insert/search operations.

There are two classes: `Bplustree` and `Node`. Degree of the tree indicates how many keys can put in a single node, since we only have a table with 50000 rows in it, we will set max degree to 10000.

```
public class Bplustree {
    public Node root; //root of the tree
    public int maxD; //max degree
    public HashMap<Integer, String> map = new HashMap<>(); //key-value pair
}
public class Node {
    int maxD,minD,curD; //max degree, min degree, current number of keys in node
    Node left,right; //left/right leaf (if isLeaf=true)
    Node parent; //parent node
    ArrayList<Node> child; //child pointers
    ArrayList<Integer> keys; //stored keys (sorted)
    boolean isLeaf=false; //if true, it can access data using Bplustree's map
}
```

insert/search are declared in `Bplustree`. **Note that delete operation is not implemented, as we didn't use this operation.**

```
public void insert(int key, String val){} //insert key and it's value
public Object searchValue(int key) {} //search for key's value
public Object rangeSearch(int min, int max){} //search for values bounded by
keys min and max (min inclusive,max exclusive)
public Object updateValue(int key,String newValue){}//update value of key in
map, and return updated value
```

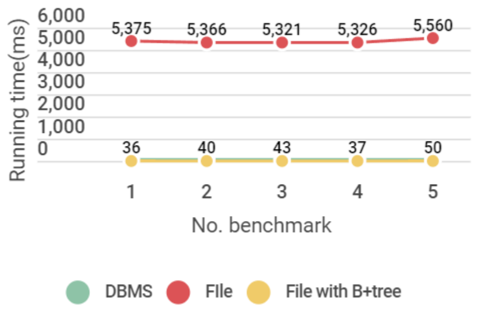
Primary key indexes

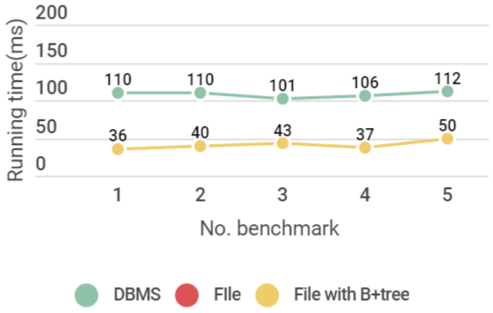
According to Postgres document, adding a unique or primary key constraint will automatically create a unique b-tree index on the column or group of columns used in the constraint.

While using DBMS to operate on these columns (e.g. `select * from order_sale where order_id = 35000`), indexing is used automatically to improve its efficiency. However, running data manipulation using file IO doesn't have that kind of improvement, so that it's significantly slower than DBMS.

We then used this statement to benchmark running time (repeatedly run 500 times) of DBMS, raw file IO and file IO with B+ tree. **Note that the process of building a B+ tree (create index) is also counted into running time in all related benchmarks.**

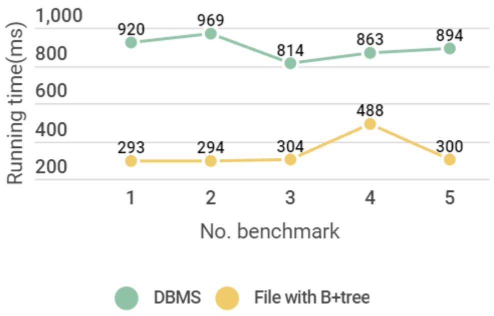
```
select * from order_sale where order_id=35000; -- single query
```





To test B+ tree's effectiveness on range queries, we benchmarked again using this statement. Range size is set smaller to reduce the extra print time.

```
select * from order_sale where order_id between 25000 and 30000; -- range query
```



B+ tree is significantly more efficient than raw file IO, and is even slightly faster than DBMS in single query. In range queries, B+ tree is significantly faster than DBMS, indicating B+ tree's effectiveness in searching elements in a range.

Non-primary key indexes

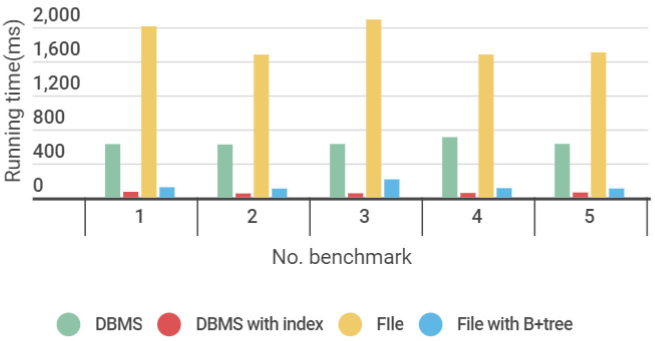
Besides indexing on primary key and unique columns, indexes can also be created on other columns according to document.

We created an index on product_model column and sort it in ascending order.

```
create index model_asc on order_sale (product_model asc);
```

We used this statement to benchmark running time (repeatedly run 100 times) of DBMS before and after creating the index, raw file IO and file IO with B+ tree. **Here create index statement for DBMS's time is not counted, as it is done in Datagrip.**

```
select contract_number,product_model from order_sale where product_model like
'Mp437';
```



As we can see, index and B+ tree both improved efficiency significantly. It's worth noting that `product_model` itself is not a unique column, which means there could be duplicate keys in B+ tree if not treated carefully. To solve it, I put all rows that have the same model into the same B+ tree key, and B+ tree's keys are created from table `model`, which has unique `product_model` column. Thus building the tree takes more time.

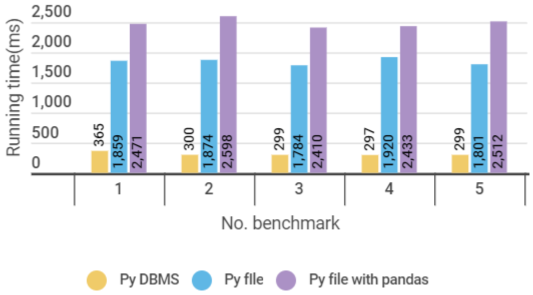
6) Comparisons with Python

We also tested performance in Python using `psycopg2` for DBMS, file object and `pandas` for file IO. Four statements are tested, and related python script is in the zip file.

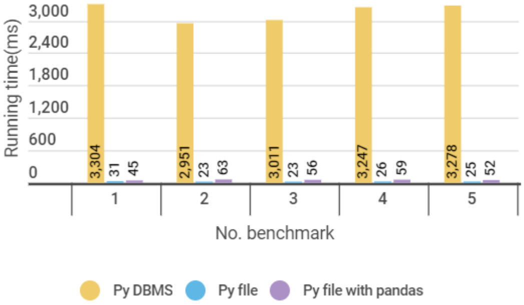
```
select * from order_sale where product_model like 'Mp437'; -- ×50
insert into order_sale (order_id, contract_number,
product_model,est_date,lodge_date,quantity,salesman_id) values (?,?,?,?,?,?,?,?);
-- ×25000
update order_sale set quantity = 5000 where order_id = 1 ; -- ×100
delete from order_sale where order_id=? ; -- ×100
```

Test methods and repeated times are the same as that in java, so that readers can also compare python data with java's.

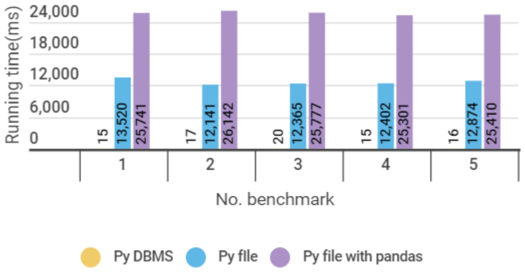
`select` result:



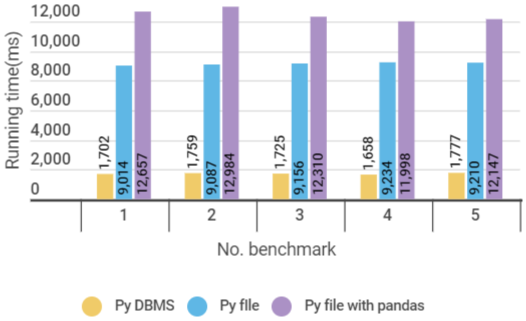
`insert` result:



`update` result:



`delete` result:



From the figures, DBMS performs better than file in `select`, `delete`, `update`, which bears the same result as that in Java. However, performance of DBMS in Java and Python differs, as `select` is faster in Python, but the rest are slower. File IO's advantage lies in `insert` (same as Java's file IO), because only writing to a file is much more efficient than DBMS.

7) User Privileges management

Design

Noticed that the database has only one user checker, which has the privileges of create role and is a superuser.

Considering the object and environment of contract, we design another four role: `company`, `director`, `person_manager` and `visitor`.

Denoted: Insert ->I, Delete ->D, Select -> S, Update -> U

The privileges of users and roles is:

	login	create role	password
checker	Y	Y	123456
company	Y	N	456
director	Y	N	cs307
person_manager	Y	N	654321
visitor	Y	N	123

All user can login with their own password and **select** all the tables in database.

Company has the right to operate their infos, so grant company **IDU** to table client. It can update their product and add model, so grant **IDU** to table product and model.

Company and director jointly take part in contract foundation, so set **IDU** to table **order_sale** and **contract**.

People_manager can control the salesman changes, so set **IDU** to table salesman.

Visitor has no right to change the data.

The privileges shows in this form:

(Y implies the role can **IDU** to this table, N implies the role can not **IDU** to this table.)

	order_sale	contract	client	product	model	salesman
checker	Y	Y	Y	Y	Y	Y
company	Y	Y	Y	Y	Y	N
director	Y	Y	N	N	N	N
person_manager	N	N	N	N	N	Y
visitor	N	N	N	N	N	N

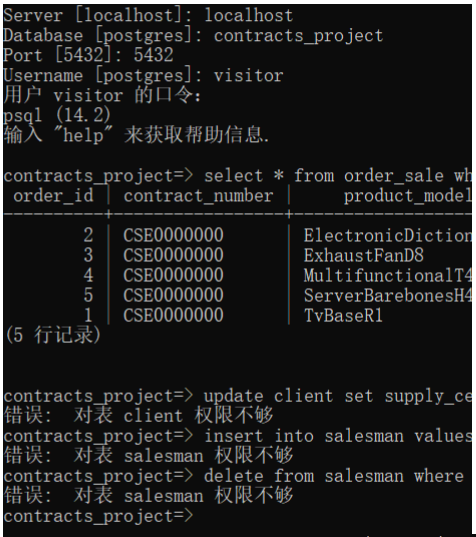
Test

Use some sql statements to check privileges.

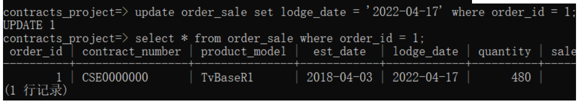
```
select * from order_sale where contract_number = 'CSE0000000';
update client set supply_center = 'ShenZhen' where client_id = 1;
insert into salesman values(991,'21112221','zhang','Male',26,13945620153);
update order_sale set lodge_date = '2022-04-17' where order_id = 1;
delete from salesman where salesman_id = 991;
```

We choose three examples:

Test for visitor:

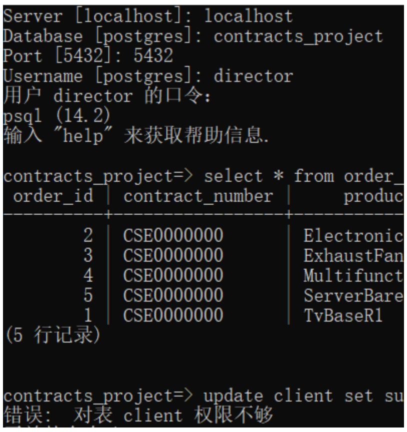


```
Server [localhost]: localhost
Database [postgres]: contracts_project
Port [5432]: 5432
Username [postgres]: visitor
用户 visitor 的口令:
psql (14.2)
输入 "help" 来获取帮助信息.

contracts_project=> select * from order_sale wh
 order_id | contract_number |     product_model
----------+-----------------+-------------------
        2 | CSE0000000      | ElectronicDiction
        3 | CSE0000000      | ExhaustFanD8
        4 | CSE0000000      | MultifunctionalT4
        5 | CSE0000000      | ServerBarebonesH4
        1 | CSE0000000      | TvBaseR1
(5 行记录)

contracts_project=> update client set supply_ce
错误:  对表 client 权限不够
contracts_project=> insert into salesman values
错误:  对表 salesman 权限不够
contracts_project=> delete from salesman where
错误:  对表 salesman 权限不够
contracts_project=>
```



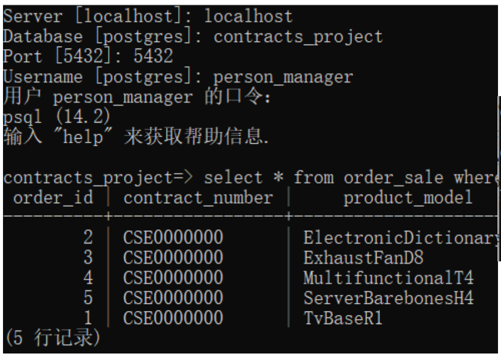
```
contracts_project=> update order_sale set lodge_date = '2022-04-17' where order_id = 1;
UPDATE 1
contracts_project=> select * from order_sale where order_id = 1;
 order_id | contract_number | product_model |  est_date  | lodge_date | quantity | sale
----------+-----------------+---------------+------------+------------+----------+-----
        1 | CSE0000000      | TvBaseR1      | 2018-04-03 | 2022-04-17 |      480 |
(1 行记录)
```

Test for director:

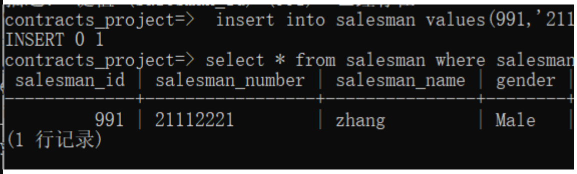


```
Server [localhost]: localhost
Database [postgres]: contracts_project
Port [5432]: 5432
Username [postgres]: director
用户 director 的口令:
psql (14.2)
输入 "help" 来获取帮助信息.

contracts_project=> select * from order_
 order_id | contract_number |      produc
----------+-----------------+------------
        2 | CSE0000000      | Electronic
        3 | CSE0000000      | ExhaustFan
        4 | CSE0000000      | Multifunct
        5 | CSE0000000      | ServerBare
        1 | CSE0000000      | TvBaseR1
(5 行记录)

contracts_project=> update client set su
错误:  对表 client 权限不够
```

Test for person_manager:



```
Server [localhost]: localhost
Database [postgres]: contracts_project
Port [5432]: 5432
Username [postgres]: person_manager
用户 person_manager 的口令:
psql (14.2)
输入 "help" 来获取帮助信息.

contracts_project=> select * from order_sale where
 order_id | contract_number |      product_model
----------+-----------------+---------------------
        2 | CSE0000000      | ElectronicDictionary
        3 | CSE0000000      | ExhaustFanD8
        4 | CSE0000000      | MultifunctionalT4
        5 | CSE0000000      | ServerBarebonesH4
        1 | CSE0000000      | TvBaseR1
(5 行记录)
```



```
contracts_project=>  insert into salesman values(991,'211
INSERT 0 1
contracts_project=> select * from salesman where salesman
 salesman_id | salesman_number | salesman_name | gender |
-------------+-----------------+---------------+--------+
         991 | 21112221        | zhang         | Male   |
(1 行记录)
```