DBMS Performance Evaluation

Student ID: 12210505

Donghang Liu

Shenzhen, Guangdong
October 2025

Contents

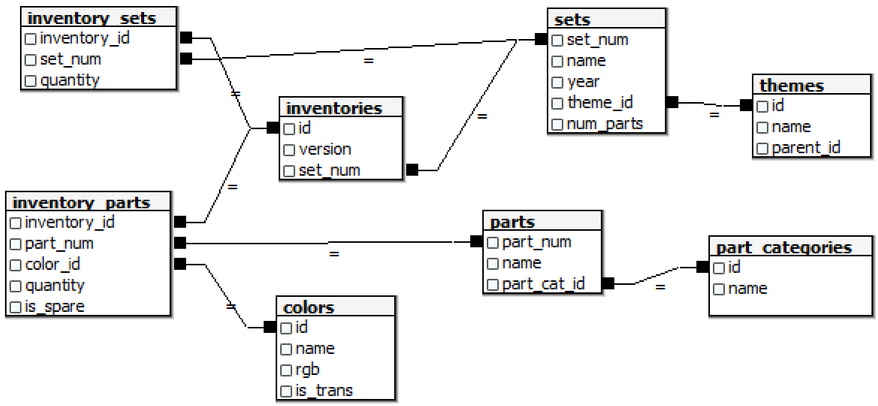

图 1: Relations

0.1 Data Source

To ensure the reliability and statistical validity of this experiment, a dataset with a relatively large sample size is required. After screening, we selected a suitable dataset from Kaggle—a well-known platform for open data sharing—with the file size constrained between 100MB and 1GB. The final dataset adopted in this experiment is the **LEGO Database**. (Data Source URL: `https://www.kaggle.com/datasets/rtatman/lego-database?select=downloads_schema.png`), which integrates comprehensive information on every official LEGO set in the Rebrickable database—including details of LEGO parts, sets, colors, and inventories of each set. Notably, this dataset has a reasonable size: it provides **sufficient data volume** to support in-depth analysis while **avoiding excessive computational burden** or operational inefficiencies caused by unnecessarily large files. Since the dataset focuses on organizing core information of official LEGO sets in a structured manner, it eliminates the need for redundant basic data cleaning and ensures rational associations between data dimensions, which **enables us to do Multiple table retrieval operations.** As data cleaning is not the key focus of this course, it allows us to directly focus our attention to database construction.

0.2 Data Files Overview

To make a simple discription for this database, we use the graph provided in the website.

As we can see, the database were constructed by 8 files:

themes,sets,inventories,inventory_sets,inventory_parts,parts,colors and part_categories. Then we will use these data to make experiments and draw the conclusions.

0.3 Data importing

0.3.1 Create the database

```
postgres=# create database lego_db with encoding 'UTF-8';
CREATE DATABASE
postgres=# \c lego_db
You are now connected to database "lego_db" as user "postgres".
lego_db=#
```

0.3.2 import themes.csv

```
lego_db=# create table themes(
lego_db(# id int primary key,
lego_db(# name varchar(80) not null,
lego_db(# parent_id int references themes(id));
CREATE TABLE
lego_db=# copy themes(id,name,parent_id)
lego_db-# from 'D:\SQLlab\lego\data\themes.csv'
lego_db-# csv header;
COPY 614
```

0.3.3 import sets.csv

```
lego_db=# create table sets(
lego_db(# set_num varchar(50) primary key,
lego_db(# name varchar(100) not null,
lego_db(# year int,
lego_db(# theme_id int,
lego_db(# num_parts int);
CREATE TABLE
lego_db=# COPY sets (set_num, name, year, theme_id, num_parts)
lego_db-# FROM 'D:\SQLlab\lego\data\sets.csv'
lego_db-# WITH (
lego_db(#   FORMAT CSV,
lego_db(#   HEADER,
```

```
lego_db(#   ENCODING 'UTF8'
lego_db(# );
COPY 11673
```

0.3.4 import parts.csv

```
lego_db=# create table parts(
lego_db(# part_num varchar(50) primary key,
lego_db(# name varchar(255) not null,
lego_db(# part_cat_id int);
CREATE TABLE
lego_db=# copy parts(part_num,name,part_cat_id)
lego_db-# from 'D:\SQLlab\lego\data\parts.csv'
lego_db-# with(
lego_db(# format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 25993
```

0.3.5 import part_categories.csv

```
lego_db=# create table part_categories(
lego_db(# id int primary key,
lego_db(# name varchar(255) not null);
CREATE TABLE
lego_db=# copy part_categories(id,name)
lego_db-# from 'D:\SQLlab\lego\data\part_categories.csv'
lego_db-# with(
lego_db(# format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 57
```

0.3.6 import inventories_sets.csv

```
lego_db=# create table inventory_sets(
lego_db(# inventory_id int ,
lego_db(# set_num varchar(50),
lego_db(# quantity int);
CREATE TABLE
lego_db=# copy inventory_sets(inventory_id,set_num,quantity)
lego_db-# from 'D:\SQLlab\lego\data\inventory_sets.csv'
lego_db-# with(
lego_db(# format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 2846
```

0.3.7 import inventories_parts.csv

```
lego_db=# create table inventory_parts(
lego_db(# inventory_id int,
lego_db(# part_num varchar(50),
lego_db(# color_id int,
lego_db(# quantity int,
lego_db(# is_spare varchar(10));
CREATE TABLE
lego_db=# copy inventory_parts(inventory_id,part_num,color_id,
   quantity,is_spare)
lego_db-# from 'D:\SQLlab\lego\data\inventory_parts.csv'
lego_db-# with(
lego_db(# format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 580251
```

0.3.8 import inventories.csv

```
lego_db=# create table inventories(
lego_db(# id int primary key,
lego_db(# version int not null,
lego_db(# set_num varchar(50));
CREATE TABLE
lego_db=# copy inventories(id,version,set_num)
lego_db-# from 'D:\SQLlab\lego\data\inventories.csv'
lego_db-# with(
lego_db(# format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 11681
```

0.3.9 import colors.csv

```
lego_db=# create table colors(
lego_db(# id int primary key,
lego_db(# name varchar(50) not null,
lego_db(# rgb varchar(20) not null,
lego_db(# is_trans varchar(10));
CREATE TABLE
lego_db=# copy colors(id,name,rgb,is_trans)
lego_db-# from 'D:\SQLlab\lego\data\colors.csv'
lego_db-# with(
lego_db(# format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 135
```

From this point forward, we have imported all the data into the PostgreSQL database and can proceed with further performance comparisons.

1. Unique Advantages of a DBMS Compared with Data Operations in Files

1.1 Retrieval comparison

1.1.1 Single table

Method

In the scenario of single-table retrieval, we will use the data from the `inventory_parts` table, with the goal of querying all spare parts under each inventory. We will implement this through two methods respectively: one is using C language, and the other is using the PostgreSQL command line. Both methods need to filter the rows in the inventory_parts table where **the value of the `is_spare` column is 't'**, and export the results to "spare_parts1i" (the file exported by the C language method, i is an integer ranging 0 to 9) and "spare_parts2i" (the file exported by the PostgreSQL command line method, i is an integer ranging 0 to 9) respectively. During the operation, it is necessary to record the running time of both methods simultaneously and compare the final actual results to ensure the accuracy of the output.

Among them, the code file corresponding to the C language implementation is **RetrievalSingle.c**, and the PostgreSQL commands are as follows:

```
lego_db=# SELECT NOW() AS export_start \gset
lego_db=# COPY (
lego_db(#   SELECT inventory_id, part_num, color_id, quantity
lego_db(#   FROM inventory_parts
lego_db(#   WHERE is_spare = 't'
lego_db(# ) TO 'D:\\SQLlab\\lego\\spare_parts2i.txt'
lego_db-# DELIMITER ','
lego_db-# HEADER;
COPY 29495
lego_db=# SELECT
lego_db-#   :'export_start' AS Start Time,
lego_db-#   NOW() AS End Time,
lego_db-#   EXTRACT(EPOCH FROM (NOW() - :'export_start'::timestamp)) * 1000 AS Total Time Cost;
lego_db-#  --Note: The timing feature of this method requires an additional carriage return at the end when copying and pasting.
```

Result

For PostgreSQL operations, we test 10 times and take the average time. The results are as follows:

Table 1.1: Time Statistics for PostgreSQL Data Export

Test	Start Time	End Time	Total Time Cost(ms)
1	2025-10-14 21:50:49.808658+08	2025-10-14 21:50:49.883783+08	75.125000
2	2025-10-14 21:55:36.375267+08	2025-10-14 21:55:36.576578+08	201.311000
3	2025-10-14 21:56:29.671241+08	2025-10-14 21:56:29.754062+08	82.821000
4	2025-10-14 21:57:00.147247+08	2025-10-14 21:57:00.237836+08	90.589000
5	2025-10-14 21:57:33.910414+08	2025-10-14 21:57:33.996542+08	86.128000
6	2025-10-14 21:57:59.004427+08	2025-10-14 21:57:59.179785+08	175.358000
7	2025-10-14 21:58:19.112477+08	2025-10-14 21:58:19.215618+08	103.141000
8	2025-10-14 21:58:41.726840+08	2025-10-14 21:58:41.807541+08	80.701000
9	2025-10-14 21:59:10.689873+08	2025-10-14 21:59:10.768968+08	79.095000
10	2025-10-14 21:59:32.404658+08	2025-10-14 21:59:32.522039+08	117.381000
	Average Time:109.17 ms	Average for 10 tests	

For C program, we also test 10 times and take the average time. In the .c file, we the program read the .csv file in every loop And the output of the program is:

Table 1.2: Performance Test Results of C Language Singly Linked List Retrieval and Export

Test No.	Total Time per Test (ms)	Remarks
1	284.00	Test 1
2	309.00	Test 2
3	301.00	Test 3
4	291.00	Test 4
5	309.00	Test 5
6	288.00	Test 6
7	348.00	Test 7
8	283.00	Test 8
9	282.00	Test 9
10	275.00	Test 10
Average Time Cost:297.00 ms		Average of 10 Tests

Analysis

As we can see from the test results, **the database' s single-table retrieval (with result export)** is approximately **3 times faster** than **the file-based data processing (reading CSV, filtering data, and exporting results)**.

Finally, the CompareRS.C program was utilized to compare the 20 result files, with the result confirming that all files shared **an identical set of data records!**(Direct line-by-line comparison of the files **failed, as the output order of the database results does not necessarily match that of the files.**)

1.1.2 Multiple table

A key advantage of databases over file operations lies in their ability to use multi-table joins, which can establish associations between multiple tables at once and directly perform data matching and extraction.

Method

Our task is to Query the details of "Castle" theme Lego sets released between 2000 and 2020 that contain black parts with the inventory quantity of ≥ 5(regardless of spare parts), and output the set number, set name, release year, theme name (Castle), (black) part number, and quantity of black parts. At last, we order it by the number of the inventory quantity.

This task **establish associations between 5 tables**, and we can use the following code to perform Database operations properly.

```
\timing on
SELECT
  s.set_num AS set_num,
  s.name AS set_name,
  s.year AS publish_year,
  t.name AS theme_name,
  ip.part_num AS part_ID,
  ip.quantity AS inventory_quantity
FROM
    sets s
    INNER JOIN themes t
        ON s.theme_id = t.id
        AND t.name = 'Castle'
    INNER JOIN inventories i
```

```
        ON s.set_num = i.set_num
    INNER JOIN inventory_parts ip
        ON i.id = ip.inventory_id
        AND ip.quantity >=5
    INNER JOIN colors c
        ON ip.color_id = c.id
        AND c.name = 'Black'
WHERE
  s.year BETWEEN 2000 AND 2020
ORDER BY
  ip.quantity DESC;
```

For file operation, we use **RetrivelMultiple.c** to do the task and direct print the output. We repeat it 5 times and check the time cost.

Result

The result of time cost is

Table 1.3: Respective Time Consumption of the Two Methods (Unit: ms)

trail	Database operation	file operation
1	156.87	265
2	114.68	247
3	153.89	243
4	108.16	247
5	115.54	237
avg	129.8	247.8

Analysis

We can observe that when performing multi-table join queries using database operations, the query speed is **significantly faster** than that of file operations even when the data scale is **not very large (around 10MB)**. The average time consumed by file operations can even be twice that of database operations. Additionally, it should be noted that in the file query, I used the clock() function. The precision of this function is approximately at the level of 10 to 15 milliseconds, so retaining an excessive number of decimal places in the results is meaningless. However, this precision is **sufficient for performance comparison.**

1.1.3 Retrival with Index

In the previous sections, we have compared the efficency of database operation and file operation. Now, let's turn our attention to another data structure that can speed up database queries: **indexes.**

Indexes help databases quickly locate and retrieve data stored in tables, thereby accelerating data query speed. When the amount of data is relatively large, using indexes can greatly improve the efficiency of data retrieval. Here, we will use parts.csv to conduct a performance test on databases with and without indexes.

Method

We use 2 different ways to import the data:

- Perform a standard import **without** setting a primary key.
- Set the first column as the primary key during the import process, and postgreSQL will **automatically create the primary key index!**

```
lego_db=# create table parts_no_key(
lego_db(# part_num varchar(50),
lego_db(# name varchar(255) not null,
lego_db(# part_cat_id int);
CREATE TABLE
lego_db=# create table part_with_key(
lego_db(# part_num varchar(50) primary key,
lego_db(# name varchar(255) not null,
lego_db(# part_cat_id int);
CREATE TABLE
```

As the data volume is not large enough(25994 in total), we wish to enlarge it 25 times using file operation(Enlarge_parts.csv)

Then we import the data into these two tables.

```
lego_db=# copy parts_no_key(part_num,name,part_cat_id)
lego_db-# from 'D:\SQLlab\lego\data\expanded_parts.csv'
lego_db-# with (format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
```

```
COPY 649825
lego_db=# copy part_with_key(part_num,name,part_cat_id)
lego_db-# from 'D:\SQLlab\lego\data\expanded_parts.csv'
lego_db-# with (format csv,
lego_db(# header,
lego_db(# encoding 'UTF8'
lego_db(# );
COPY 649825
```

And we make 3 sets of experiments: Simple query, Fuzzy query, Combined query.

We repeat each experiments 5*2 times.

```
--for simple query, we randomly choose a data with part_num equal
    to '13-3626cpr1927'(data in the middle), and do  the research
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT * FROM parts_no_key
lego_db-# WHERE part_num = '13-3626cpr1927';
--result are in the Results part
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT * FROM part_with_key
lego_db-# WHERE part_num = '13-3626cpr1927';
--result are in the Results part
```

```
--for fuzzy query, we select the data with part_num contains
   '3-3626cpr1927'(make it easy to justify the results, only
   '03-3626cpr1927','13-3626cpr1927','23-3626cpr1927')
lego_db=#
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT *  FROM parts_no_key
lego_db-# WHERE part_num LIKE '%3-3626cpr1927%';
--result are in the Results part
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT *  FROM part_with_key
lego_db-# WHERE part_num LIKE '%3-3626cpr1927%';
--result are in the results part
```

```
-- for combined query, we select the data with name = 'Technic
   Link Tread Wide with Two Pin Holes' and first two integers of
```

```
   part_num >15
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT *  FROM parts_no_key
lego_db-# WHERE name = 'Technic Link Tread Wide with Two Pin 
   Holes'
lego_db-#  AND CAST(SUBSTRING(part_num FROM 1 FOR 2) AS INT) >
   15;
--result are in the Results part
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT *  FROM part_with_key
lego_db-# WHERE name = 'Technic Link Tread Wide with Two Pin 
   Holes'
lego_db-#  AND CAST(SUBSTRING(part_num FROM 1 FOR 2) AS INT) >
   15;
--result are in the Results part
```

Results

We find that compared with the table without an index, **the query performance of the table with an index is extremely good.** Its index scan function can directly define the target, greatly increasing the execution efficiency. However, for fuzzy queries (full matching), its efficiency may be **even worse** than that of the table without an index (the reason will be explained later). For Combined query, we **do not find significant difference** between these two querys

Table 1.4: Comparison of Query Performance between Tables with/without Index

Compare Dimension	**Table without Index (parts_no_key)**	**Table with Index (part_with_key)**
Scanning Method	Parallel full - table scan (Parallel Seq Scan)	Index scan (Index Scan)
Number of Worker Processes	2 (Workers Launched: 2)	0 (Single - process Index Positioning)
Filtered Rows	216,608 rows/process	0 rows (Direct Target Positioning)
Execution Time	54 ms	1,405 ms
Relative Efficiency Improvement	-	Approximately 38 times

Table 1.5: Comparison of Core Performance Indicators for Fuzzy Query (Full Matching)

Comparison Dimension	**Table without Index (`parts_no_key`)**	**Table with Index (`part_with_key`)**
Scanning Method	Parallel full - table scan (Parallel Seq Scan)	Parallel full - table scan (Parallel Seq Scan)
Number of Worker Processes	2 (Workers Launched: 2)	2 (Workers Launched: 2)
Filtered Rows	216,607 rows/process	216,607 rows/process
Execution Time	75 ms	97 ms
Performance Difference	-	Execution time increased by approximately 29.3% (1.29 times slower)

Table 1.6: Comparison of Core Performance Indicators for Joint Query

Comparison Dimension	**Table without Index (`parts_no_key`)**	**Table with Index (`part_with_key`)**
Scanning Method	Parallel full - table scan (Parallel Seq Scan)	Parallel full - table scan (Parallel Seq Scan)
Number of Worker Processes	2 (Workers Launched: 2)	2 (Workers Launched: 2)
Filtered Rows	216,605 rows/process	216,605 rows/process
Execution Time	70-180 ms	80-180 ms
Performance Difference	-	Nearly the same speed, with large fluctuations

Analysis.1

We found that the indexed table seems to perform poorly in fuzzy queries (full matching) and joint queries. In fact, due to the **leftmost matching principle of indexes**, they are not suitable for complex fuzzy matching. Therefore, the index ultimately fails to achieve the optimization effect. For the same target, we manually choose to **use the pg_trgm extension to optimize fuzzy matching.**

```
lego_db=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
lego_db=# CREATE INDEX idx_col_trgm ON part_with_key USING gin(
    part_num gin_trgm_ops);
CREATE INDEX
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT * FROM part_with_key
lego_db-# WHERE part_num LIKE '%3-3626cpr1927%';
```

Table 1.7: Comparison of Core Performance Indicators for Fuzzy Query (Full Matching) (Normal Index vs trigram Index)

Comparison Dimension	Table with Normal Index (part_normal_key)	Table with Trigram Index (part_trigram_key)
Scanning Method	Full - table scan (Seq Scan)	Bitmap Heap Scan + Bitmap Index Scan
Index Type	B+ tree index (normal)	GIN index (based on pg_trgm)
Filtered Rows	216,607 rows	25 rows (post - index scan heap scan filtering)
Execution Time	97 ms	6.656 ms
Performance Difference	-	Execution time reduced by approximately 93.1% (13.6 times faster)

Analysis.2

We found that even with indexed tables, the performance in joint queries is **still poor**. In fact, for this table, due to the **highly specific nature of the `name` column, but without specifying it as `UNIQUE` when defining the table (and actually being uncertain whether it is `UNIQUE`), the advantage of the index is not utilized when performing `WHERE` filtering.** In addition, because a function operation like `CAST(SUBSTRING(part_num FROM 1 FOR 2) AS INT)` is performed on the `part_num` column, the database cannot directly use the index of `part_num` itself to quickly filter data, so the performance is no different. Therefore, we **create a normal index for the `name` column and an expression index based on the substring conversion of `part_num`:**

```
lego_db=# CREATE INDEX idx_part_with_key_name ON part_with_key (
   name);
CREATE INDEX
lego_db=# CREATE INDEX idx_part_with_key_part_num_prefix ON
   part_with_key (CAST(SUBSTRING(part_num FROM 1 FOR 2) AS INT));
CREATE INDEX
lego_db=# EXPLAIN ANALYZE
lego_db-# SELECT *  FROM part_with_key
lego_db-# WHERE name = 'Technic Link Tread Wide with Two Pin 
   Holes'
lego_db-#   AND CAST(SUBSTRING(part_num FROM 1 FOR 2) AS INT) >
   15;
```

Table 1.8: Comparison of Core Performance Indicators for Joint Query (Table with Normal Index vs Table with Optimized Index)

Comparison Dimension	Table with Normal Index (part_normal_key)	Table with Optimized Index (part_optimized_key)
Scanning Method	Parallel full - table scan (Parallel Seq Scan)	Bitmap Heap Scan + Bitmap Index Scan
Number of Worker Processes	2 (Workers Launched: 2)	1 (loops=1)
Filtered Rows	216,605 rows/process	25 rows (filtered after index scan hit)
Execution Time	80 - 180 ms	0.323 ms
Performance Difference	-	Execution time reduced by approximately 99.7% (about 371 times faster)

1.1.4 Conclusion

This section compares the retrieval performance of PostgreSQL and file operations(without index as we didn't define primary key in table `inventory_parts`), and finally explores the impact of indexes.

The result is that PostgreSQL performs better in all the tests; however, we finally noticed that if we have a **sufficient understanding** of the underlying logic of the database query function, we can optimize the index query method effectively, **improving the search efficiency by hundreds of times**, rather than making meaningless comparisons within the same order of magnitude. (Due to the randomness of database search algorithms, the search time fluctuates significantly, and in the worst case, the efficiency may even be lower than that of file search.) For the ”fairness” of the comparison, in the next chapter, we will **avoid using index queries** as much as possible when comparing update speed and accuracy.

1.2 Update comparison

Unlike file systems (eg .csv files) that require manual synchronization of cross-file data modifications, databases enable automated data management through **structured storage architectures and encapsulated relational logic**. This transformation shifts data maintenance from error-prone manual operations to system-driven automation, **ensuring efficiency, consistency, and security across all update scenarios.** Below is a detailed breakdown of database update capabilities, compared with traditional file-based approaches.

1.2.1 Efficiency: Simple comparison

Here, we need to copy all the data in the parts table where the `part_cat_id` value is 1. Then, add the prefix "new" to the `part_num` of the copied data and change the `part_cat_id` to 100. We will perform database operations and file operations (implemented with **Update.c**) respectively, and overwrite the original table and original file with the results.

Method

First, we copy 5 brand-new forms respectively for experiments. When performing file operations, output the updated files as parts_copy1.csv to parts_copy5.csv (because the time difference between file overwriting and file adding is only at the millisecond level from the underlying logic). Then, perform target operations on the database and files respectively, repeat the experiment 5 times to obtain the average time consumed for the update operation. Finally, export the obtained 5 sets of database experiment results as files, and compare the group relations of these 10 updated files through **CompareU.c** to determine the accuracy of the operations.

Database operations as follows:

```
--table parts_copy1 as an example:
lego_db=# CREATE TABLE IF NOT EXISTS parts_copy1 (like parts including all);
CREATE TABLE
lego_db=# INSERT INTO parts_copy1 (part_num, name, part_cat_id)
lego_db-# SELECT part_num, name, part_cat_id
lego_db-# FROM parts;
INSERT 0 25993
--then we do the operation and time how long the operation takes.
lego_db=# EXPLAIN ANALYZE
lego_db-# UPDATE parts_copy1
lego_db-# SET
lego_db-#   part_num = 'new_' || part_num,  -- part_num
lego_db-#   part_cat_id = 100
lego_db-# WHERE part_cat_id = 1;
--We will show the results in the Results part.
--And finally we get the results into .csv files. We won't take this time into account! as the update for database doesn't involve creating a new file!
lego_db=# COPY parts_copy1 TO 'D:/SQLlab/lego/data/parts_copy6.
```

```
    csv' WITH CSV HEADER ENCODING 'UTF8';
COPY 25993
```

Results

Table 1.9: Respective Time Consumption of the Two Operations (Unit: ms)

trail	Database operation	File operation
1	5.301	15.0
2	5.054	16.0
3	4.484	13.0
4	7.121	14.0
5	2.741	15.0
avg	4.940	14.6

The comparison result is that the output files are not all the same?!!!(CompareU.c)

analysis

Actually, when we check the mistake code, we found that we didn't consider rows like:
12046,"LIONS CUB, DEC.",4(multiple commas)
and letting the file operation seperate the datas simply by comma is a bad idea!
So we need to make some little changes:(NewUpdate.c)

Then finally we find that Database operations not only **enable fast implementation** and **code simplicity**, but also **adjust the reading logic according to the data format.** Here are the final result:(table 1.10, new code need to consider more, so it needs more time!)

Table 1.10: Respective Time Consumption of the Two Operations (Unit: ms)

trail	Database operation	(New) File operation
1	5.301	24.0
2	5.054	33.0
3	4.484	29.0
4	7.121	29.0
5	2.741	27.0
avg	4.940	28.4

1.2.2 Reliability: Atomicity Guarantee for Data Modifications

Atomicity, one of the four core ACID properties of databases ensures that a sequence of update operations is treated as an **indivisible unit**: either all operations succeed and are persisted, or all fail and the system rolls back to the pre-update state. This property eliminate the "partial update" risk that can't be avoided in file operations. This is particularly important in the financial industry. We need to ensure that the data operations performed are either fully executed or not executed at all.

Here are the codes and results:

codes:

```
postgres=# create database atomic_test;
CREATE DATABASE
postgres=# \c atomic_test
You are now connected to database "lego_db" as user "postgres".
atomic_test=# create table users(
atomic_test(# id varchar(10) primary key,
atomic_test(# balance int not null
atomic_test(# );
CREATE TABLE
atomic_test=# insert into users (id,balance)
atomic_test-# values ('A',1000),('B',500)
atomic_test-# on conflict (id) do update set balance = excluded.
    balance
atomic_test-# ;
INSERT 0 2
--
atomic_test=# select * from users;
 id | balance
----+---------
 A  |    1000
 B  |     500
(2 rows)
```

If we want to record that A gives 500 to B, we may let A minus by 500 and add 500 to B. **However, if we made a mistake will doing the operation, we can still make up for it.**

```
--accurate operation
atomic_test=# BEGIN;
BEGIN
atomic_test=*# update users set balance = balance - 500 where id
   = 'A';
UPDATE 1
```

```
atomic_test=*# update users set balance = balance + 500 where id
    = 'B';
UPDATE 1
atomic_test=*# COMMIT;
COMMIT
atomic_test=# select * from users;
 id | balance
----+---------
 A  |     500
 B  |    1000
(2 rows)
```

```
--made a mistake, using rollback function
atomic_test=# BEGIN;
BEGIN
atomic_test=*# update users set balance = balance - 500 where id
    = 'A';
UPDATE 1
atomic_test=*# update users set balance = balance + 500 where id
    = 'C';
UPDATE 0
atomic_test=*# --here we made a mistake
atomic_test=*# ROLLBACK;
ROLLBACK
atomic_test=# select * from users
atomic_test-# ;
 id | balance
----+---------
 A  |    1000
 B  |     500
(2 rows)
```

If we violet the rules of the database, it will report an error and roll back automatically.

```
atomic_test=# BEGIN;
BEGIN
atomic_test=*# update users set balance = balance - 500 where id
    = 'A';
```

```
UPDATE 1
atomic_test=*# insert into users (id,balance) values ('A',1000);
Error:duplicate key value violates unique constraint "users_pkey"
Detail: Key (id)=(A) already exists.
atomic_test=!# COMMIT;
ROLLBACK
atomic_test=# select * from users;
 id | balance
----+---------
 A  |    1000
 B  |     500
(2 rows)
```

1.2.3 Security: Permission Control

Database permission control is a **fine-grained access management system** that restricts ”who can perform which operations on which data”. This **prevents unauthorized modifications or accidental data damage** —an ability completely lacking in file systems, where permissions are limited to ”read/write access to entire files.”

File system

In file operations, each file has three types of permissions: read, write, and execute. However, that means a user with write permission can modify the file as he like, for instance, even deleting all the data (which might occur by mistake and is often difficult to recover from).

Database system

Database system provides permission control through a three-tier model: **Users** → **Roles** → **Privileges.** , here we use the database in **1.2.2** as an example. we define roles, grant roles with privileges and bind users with roles. Here are the code and the results:

```
atomic_test=# create role manager login;
CREATE ROLE
atomic_test=# create role editor login;
CREATE ROLE
atomic_test=# create role users login;
CREATE ROLE
```

```
atomic_test=# grant all privileges on table users to manager;
GRANT
atomic_test=# grant select,insert,update on table users to editor
   ;
GRANT
atomic_test=# GRANT SELECT (id) ON TABLE users TO users;
GRANT
atomic_test=# CREATE USER alice WITH PASSWORD 'alice_password';
CREATE ROLE
atomic_test=# CREATE USER bob WITH PASSWORD 'bob_password';
CREATE ROLE
atomic_test=# CREATE USER tom WITH PASSWORD 'a_password';
CREATE ROLE
atomic_test=# grant manager to alice;
GRANT ROLE
atomic_test=# grant editor to bob;
GRANT ROLE
atomic_test=# grant users to tom;
GRANT ROLE
```

Now we try to login using bob's account and delete all the data.

```
PS D:\pgsql> psql -U bob -d atomic_test
Password for user bob:

psql (17.6)
Type "help" for help.

atomic_test=> delete from users;
ERROR:  permission denied for table users
--Of course, bob can view and modify the table
atomic_test=> select * from users;
 id | balance
----+---------
 A  |    1000
 B  |     500
(2 rows)
```

```
atomic_test=> insert into users(id,balance) values ('c',1500);
INSERT 0 1
atomic_test=> select * from users;
 id | balance
----+---------
 A  |    1000
 B  |     500
 c  |    1500
(3 rows)
```

And we log in using tom's account

```
PS D:\pgsql> psql -U tom -d atomic_test
Password for user tom:

psql (17.6)
Type "help" for help.

atomic_test=> select * from users;
ERROR:  permission denied for table users
atomic_test=> select id from users;
 id
----
 A
 B
 c
(3 rows)
```

We can only observe the ID instead of all the datas.

1.3 Conclusion

Compared with traditional file operations, DBMS (Database Management System) has irreplaceable advantages in **retrieval efficiency** (faster speed, easier implementation of multi-table queries), **update reliability** (atomicity guarantee), and **data security** (fine-grained permissions). It is particularly more capable of meeting the needs for efficient, secure, and stable data processing in scenarios such as **structured data management, multi-table associated queries, and high-frequency updates**.

2. Comparison between PostgreSQL and openGauss

2.1 Introduction

PostgreSQL is one of the representatives of open source relational databases. It is positioned as a general-purpose and highly scalable database, suitable for small and medium sized business scenarios, data warehouses, scientific research scenarios, and enterprises with a high dependence on the open source ecosystem. It focuses more on **"flexibility" and "open source community support".**

GaussDB is a new generation enterprise level distributed database launched by Huawei based on **PostgreSQL9.2**. It supports **both centralized and distributed deployment** forms. Meanwhile, it has key capabilities such as **high availability, high reliability, high security, elastic scaling, one click deployment, fast backup and recovery, and monitoring and alarming on the cloud**, which can provide enterprises with **comprehensive, stable and reliable, highly scalable, and high performance enterprise level database services.**

Given that the virtual environment, built via **WSL and Docker containers**, provides **limited** system resources (**restricted** CPU cores, memory capacity, and storage I/O throughput), while openGauss is designed for large-scale data scenarios with high concurrency and massive data processing capabilities, conducting extensive experimental groups would not fully reflect its performance in real world enterprise-level scenarios, making it **unnecessary** to conduct too much experiments.

2.2 Preparation

Since GaussDB can only be used in **Linux systems**, we use **WSL** (Windows Subsystem for Linux) of the Windows system and **Docker containers** to generate a virtual Linux environment, and conduct experiments in this environment.

2.2.1 Connection to GaussDatabase

```
PS D:\> wsl
liudh@LAPTOP-DSCO5OOO:/mnt/c/Users/刘东航$ docker run --name opengauss --privileged=true -d -e GS_PASSWORD=Ldh@123456 -p 5432:5432 -v /mnt/d/SQLlab/lego/data:/var/lib/gaussdb/data enmotech/opengauss:latest
1162f2dc8433cf8911f364260263ddb61e365998b7b073056facb32ebd585d78
```

```
liudh@LAPTOP-DSC05000:/mnt/c/Users/刘东航$ docker exec -it opengauss bash
root@1162f2dc8433:/# cd /var/lib/gaussdb/data
root@1162f2dc8433:/var/lib/gaussdb/data# ls
colors.csv           inventory_parts.csv  parts_copy10.csv  parts_copy3.csv  parts_copy6.csv  parts_copy9.csv  themes.csv
expanded_parts.csv   inventory_sets.csv   parts_copy1.csv   parts_copy4.csv  parts_copy7.csv  parts.csv
inventories.csv      part_categories.csv  parts_copy2.csv   parts_copy5.csv  parts_copy8.csv  sets.csv
root@1162f2dc8433:/var/lib/gaussdb/data#
root@1162f2dc8433:/var/lib/gaussdb/data# su - omm
omm@1162f2dc8433:~$  gsql -d postgres -U gaussdb -W 'Ldh@123456'
gsql ((openGauss 6.0.0 build aee4abd5) compiled at 2024-09-29 19:14:27 commit 0 last mr  )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

openGauss=>
```

To ensure database security, the user password of GaussDB must be **at least 8 characters long** and must contain **at least three of the following categories**: uppercase letters, lowercase letters, numbers, and special characters (such as!, @, #, $, %,^, &, *).

2.2.2 Connection to PostgreSQL

```
PS D:\> wsl
liudh@LAPTOP-DSC05000:/mnt/c/Users/刘东航$ docker run -d --name postgredb -p 5433:5432 -e POSTGRES_PASSWORD=PGSQL@123456 -v /mnt/d/SQLlab
/lego/data:/var/lib/postgresql/data bitnami/postgresql:latest
9d750a2cffd2b9db62e537f766801f31d379a4d5b8d22467470e042da0d5929e
liudh@LAPTOP-DSC05000:/mnt/c/Users/刘东航$ docker exec -it postgredb bash
I have no name! [ / ]$ psql -U postgres
Password for user postgres:
```

```
psql (18.0)
Type "help" for help.

postgres=#
```

2.2.3 Create database and tables

```
openGauss=> CREATE DATABASE lego_db;
CREATE DATABASE
openGauss=> \c lego_db
lego_db=> CREATE TABLE inventory_parts (
lego_db(>  inventory_id INT,
lego_db(> part_num VARCHAR(50),
lego_db(> color_id INT,
lego_db(> quantity INT,
lego_db(> is_spare VARCHAR(10)
lego_db(> );
CREATE TABLE
lego_db=> \copy inventory_parts (inventory_id, part_num, color_id
    , quantity, is_spare) FROM '/var/lib/gaussdb/data/
    inventory_parts.csv' WITH (FORMAT csv, HEADER ON, ENCODING '
    UTF8');
```

```
postgres=# CREATE DATABASE lego_db;
CREATE DATABASE
postgres=# \c lego_db
You are now connected to database "lego_db" as user "postgres".
lego_db=# CREATE TABLE inventory_parts (
lego_db(#     inventory_id INT,
lego_db(#     part_num VARCHAR(50),
lego_db(#     color_id INT,
lego_db(#     quantity INT,
lego_db(#     is_spare VARCHAR(10)
lego_db(# );
CREATE TABLE
```

```
lego_db=# \copy inventory_parts (inventory_id, part_num, color_id
    , quantity, is_spare) FROM '/var/lib/postgresql/data/
    inventory_parts.csv' WITH ( FORMAT csv, HEADER ON, ENCODING '
    UTF8' );
COPY 580251
```

2.3 Performance Comparison

We still use the previous experimental design: Export all the relevant data of spare parts in the `parts_inventories` table of `lego_db`, and use explain analyze to check the time. Each of the two databases is repeated 5 times for comparative analysis.

Codes:

```
--Gaussdb
lego_db=> EXPLAIN ANALYZE select * from inventory_parts where
    is_spare = 't';
```

```
--PostgreSQL
lego_db=# EXPLAIN ANALYZE select * from inventory_parts where
    is_spare = 't';
```

Results:

Table 2.1: Performance Comparison between GaussDB and PostgreSQL

Comparison Dimension	GaussDB	PostgreSQL
Execution Plan Type	Single - thread Seq Scan (Sequential Full Table Scan)	Gather + Parallel Seq Scan (Parallel Full Table Scan)
Execution Time	37.08ms ~ 40.99ms (Average 39.2ms)	10.50ms ~ 16.59ms (Average 13.6ms)

We found that the efficiency of GaussDB is **stably lower** than that of PostgreSQL, and the main reason is that **it uses single scanning mode by default.**

Analysis:

As an enterprise level database, the default configuration of GaussDB focuses more on stability and is suitable for scenarios such as high concurrency transactions and complex multi-table associations. For the fairness of the experiment, we manually enabled the parallel full table scanning function of GaussDB and allowed two parallel workers (consistent with the number of parallels in PostgreSQL). According to the official documentation(openGauss 6.0.0), we modified the experiment.

Here are the modified code and results:

```
lego_db=> SET query_dop = 2;
SET
lego_db=> EXPLAIN ANALYZE SELECT/*+ tablescan(inventory_parts) */
    * FROM inventory_parts WHERE is_spare = 't';
```

Table 2.2: Performance Comparison between GaussDB and PostgreSQL (Updated)

Comparison Dimension	GaussDB	PostgreSQL
Execution Plan Type	Streaming (type: LOCAL GATHER dop: 1/2) + Seq Scan (Attempted parallelism but parallel degree not fully utilized)	Gather + Parallel Seq Scan (Parallel Full Table Scan)
Execution Time	31.615ms ~ 32.566ms (Average about 32.0ms)	10.50ms ~ 16.59ms (Average 13.6ms)

In fact, the performance of GaussDB is **still relatively poor** and it has not achieved full parallelism. The possible reason is that GaussDB adopts SMP (Symmetric Multi-Processing) parallel technology, but **the virtual system does not provide sufficient system resources**. Under such resource-constrained conditions, although the parallelism is set manually, the **database still voluntarily chooses a lower parallelism, resulting in low efficiency.**

However, another reasonable explanation is that **GaussDB performs more security-related operations, which results in slower speeds when it actually executes operations.**

2.4 Architecture Comparison

PostgreSQL: It adopts a **process model**, where each database connection corresponds to an independent process. It only supports centralized deployment and provides **only row storage in terms of data storage.**

GaussDB: It is based on a **thread pool model**, which effectively reduces the performance overhead caused by process switching. Its single kernel can support both centralized and distributed deployment forms at the same time, and can flexibly adapt to business scenarios of different scales. **At the storage level, in addition to supporting row storage, it also has the Ustore storage format, providing a richer range of choices for data storage.**

2.5 Availability

GaussDB, designed to meet enterprise-level requirements, offers **high availability**, primarily embodied in two key architectures:

- **Two places and three data centers**: Deploys three data centers across two geographic locations (primary center, intra-city backup, and remote backup), enabling rapid failover in response to regional disasters while minimizing data loss.
- **Three clusters in the same city**: Deploys three interconnected clusters within a single city, with real-time data synchronization and automatic traffic switching, ensuring uninterrupted services during single-cluster failures.

2.6 Security

PostgreSQL's access control provides security features such as row level security and role management, ensuring the security of its database. However, in comparison, GaussDB has stricter security specifications. It offers **stricter access control, security auditing, data encryption, and firewall support,** meeting enterprise level security requirements. **At the same time, it supports national cryptographic algorithms, satisfying the requirements for security compliance in the localized market of China.**

In fact, during database experiments, the complex security specifications of GaussDB are evident. For instance: When creating a new database, identity authorization is required.

```
openGauss=> CREATE DATABASE lego_db;
CREATE DATABASE
openGauss=> \c lego_db
Password for user gaussdb:
Non-SSL connection (SSL connection is recommended when requiring
   high-security)
You are now connected to database "lego_db" as user "gaussdb".
```

In addition, if there is no operation for a long time after GaussDB is connected, a timeout error will occur (discovered by chance).

```
WARNING:  Session unused timeout.
```

```
FATAL:  terminating connection due to administrator command
could not send data to server: Broken pipe
The connection to the server was lost. Attempting reset: Failed.
--We reset the setout time:
lego_db=> SHOW session_timeout;
 session_timeout
-----------------
 10min
(1 row)
lego_db=> SET session_timeout = 3600;--1 hour
SET
```

2.7 Conclusion

In this chapter, we conduct tests on the openGauss database. After a simple test of openGauss's performance, we focus on its design purposes (such as enterprise level high **availability and security**) and carry out a qualitative analysis of its functions.

In general, PostgreSQL, benefiting from the flexibility of the **open source ecosystem**, is more suitable for scenarios that require rapid function customization, it performs better in simple queries while GaussDB precisely meets the core business needs of enterprises, and has significant advantages especially in fields such as finance and government enterprise that have **extremely strict requirements for availability and security**. The difference in their features essentially reflects the design concepts of **open source general database and enterprise customized databases.**