BERUFLICHE HOCHSCHULE HAMBURG

Grouping memory-exceeding amounts of CSV records

Comparing PowerShell and MySQL

Mauricé Ricardo Bärisch

Term Paper
in course type Computer Science

Supervisor: Prof. Dr. Stefan Schiffner

Submission Date: May 19, 2024

Disclaimer

I confirm that this thesis type is my own work and I have documented all sources and material used.

Hamburg, May 19, 2024 Mauricé Ricardo Bärisch

Contents

Abstract

Performing aggregation and visualization on large datasets has become a major part of Information Technology. Such datasets are often exported by a third party software or service provider, and arrive in a transmittable format like JavaScript Object Notation (JSON) or Comma-Separated Values (CSV). This term paper compares the grouping operation on large CSV files between PowerShell's `Group-Object` command and MySQL's `GROUP BY` statement in regards of their return value, underlying algorithms and memory usage. Within two Docker containers limited in Random Access Memory (RAM), we simulate how PowerShell's and MySQL's grouping operations perform. As a result, we formulate best practices for aggregating large CSV files.

Keywords: Database (DB), Aggregation, Grouping, CSV

List of Tables

Listings

Glossary

AVD Azure Virtual Desktop. 1

CSV Comma-Separated Values. i, 1–3, 7, 8, 10, 11

DB Database. i, 7

DBMS Database Management System. iii, 1, 5, 9, 12

Docker A software for isolating applications bundled with their dependencies in containers. i, 7–11

I/O Short for Input/Output. Covers all kinds of input to and output from a system. Specifically, in this paper, we refer to it as read and write operations on the storage. iii, 7

JSON JavaScript Object Notation. i

MySQL A popular, open-source relational Database Management System (DBMS) that we are going to use in this paper. i, 1, 5, 7, 9–12

OS Operating System. 10, 11

PowerBI A data-visualization tool by Microsoft. 12

PowerShell A scripting language coming with Windows. Used for administrative and automation tasks. 1, 6–9, 11, 12

RAM Random Access Memory. i, 5, 11, 12

SQL The Structured Query Language is a popular query language based on the relational model. It is used by many known relational DBMS. 3, 7

swap Swapping is a technique where a certain amount of disk space is used to support the RAM. It comes with heavy performance penalty, as applications have to perform I/O operations to work with their application data. 10

1 Introduction

Nowadays, many processes that handle large amounts of data (not to confuse with Big Data[1]) already exist. Still, they must be maintained, verified and optimized to ensure both quality and efficiency.

1.1 Motivation

At q.beyond, we had to deal with just that: large monthly CSV exports (each file contains the data of a month) by a third-party service provider that our process was operating on. For verification, we collected the exports from a whole year, grouped their records by a column and exported each group as an individual Excel file. Each file contained two sheets: one with an aggregated monthly overview and the other with all the individual records of that particular group. In 2.3, we will explain why those two sheets require grouping, but only the latter requires every particular group member to be stored (in memory or storage).

We ended up writing a PowerShell Script, which, in our first draft, crashed the Azure Virtual Desktop (AVD) it was running on due to insufficient memory.

1.2 Main contributions

Based on this experience, we can define the purpose of this paper:

- **Problem:** We need to aggregate CSV data bigger than the available memory
- **Objectives:**
  1. Find a solution that can group large CSV data
  2. Proof that the solution works by running an experiment with data bigger than the available memory
- **Questions:**
  1. Why does the `Group-Object` command not work on limited memory?
  2. What are requirements for a grouping algorithm in order to run on low memory?

As an alternative solution, we will consider MySQL, a DBMS.

[1] Data with high variety, volume and velocity. Cannot be processed by conventional data processing software.

2 Background

2.1 Comma-Separated Values (CSV)

Specified in [1], CSV is a file format for storing a table in plain text. It has the following characteristics:

1. each row starts on a new line
2. the values for each column are separated by a delimiter (usually a comma)
3. there may be a header line those values specify the names of the columns

Our CSV export is structured in exactly this way. Let it be[2]:

```
tid,    pname,  amount,     comment
1,      Alice,  -1,         daily expenses
2,      Bob,    2,          pocket money
3,      Alice,  1,          friend
4,      Carl,   0,          dummy
5,      Dan,    2.99,       card game
...
```

Listing 1: CSV export

2.2 Relational Model

Introduced by E. F. Codd in [2], the relational model is a mathematical description of data management, structuring data in tuples and tuples in relations. A relational algebra is provided to act as a simple mathematical query language. Industrial query languages like SQL later implemented the features described in the relational model.

As further explained in [3], a relation is just a pair of a relation schema and a set of tuples (the rows of the relation).

2.2.1 Relational Schema

Arenas defines the relation schema based on three characteristics:

- **relation name**: the name of the table
- **relation attributes**: a list of uniquely identifiable attributes/columns
- **relation arity**: the number of attributes/columns

[4] defines the relation schema more formally as a database-wide mapping from relation names to either their respective arity (unnamed perspective) or their respective attributes (named perspective). Then, a row in a relation

[2] In this paper, we will use a simplification and abstraction of the original data mentioned in 1.1. The "..." indicate that it is indeed a large CSV with many more records.

is either defined as a tuple of values $a = (a_1, \ldots, a_n)$ (unnamed perspective) or a set of pairs $a = \{(c_1, a_1), \ldots, (c_n, a_n)\}$, where each pair consists of the column name c_n and the value a_n assigned to it (named perspective).

In this paper, we will use relations as described in the named perspective. For that, we must ensure that each column got a unique name. Luckily, the header from our CSV export statisfies this condition (see listing 1). Let its schema be:

```
Transaction [ tid, pname, amount, comment ]
```

Note that we do not assign any types to the attributes, as the CSV header row only consists of attribute names. This conforms with the understanding of the relational model from Schweikardt, who just assumes that the values for our columns origin from an infinite set containing any values for any columns.

The visual representation of a relation is a table. Below figure shows the corresponding relation to the first 5 records of the CSV export from listing 1.

tid	pname	amount	comment
1	Alice	-1	daily expense
2	Bob	2	pocket money
3	Alice	1	friend
4	Carl	0	dummy
5	Dan	2.99	card game

Table 1: Transaction

2.2.2 Selection and Projection

In chapter 12.1, [5] introduces relational algebra. It includes six comparison operators $\theta \in \{=, <>, <, >, \leq, \geq\}$ as well as the three logical operators $\{\wedge, \vee, \neg\}$. Furthermore, some language-specific operations are defined. We will focus on the selection and projection.

Given a relation R, the selection operator $\sigma_c(R)$ selects all rows from that relation that match the condition c (similar to the **where** clause in SQL). Such condition can consist of any valid expression using comparison operators, logical operators, attributes and constants.

For example, the selection $\sigma_{\text{pname=Alice}}(\text{Transaction})$ returns the following relation:

tid	pname	amount	comment
1	Alice	-1	daily expense
3	Alice	1	friend

Table 2: Selection of *pname=Alice* on Transaction

Similarly, the projection operator $\pi_{a1,\ldots,a_n}(R)$ projects all columns from a relation R stated in the projection list $a1,\ldots,a_n$, and ignores all others. Consider the query $\pi_{\text{pname}}(\text{Transaction})$:

pname
Alice
Bob
Carl
Dan

Table 3: Projection of *pname* on Transaction

Note that the projection removes fully duplicate rows, hence why Alice only exists once in the resulting relation.

As the visualizations suggests, both the selection and projection return a relation.

2.3 Grouping

Since a projection removes duplicated rows, projecting only the grouping column (the column whose distinct values should be the name of our groups), results in a relation containing all group names (see table 3).

2.3.1 Split into groups

We can use these group names to split the relation into distinct sub-relations consisting only of the elements with the same group name

$$G := \{\, \sigma_{\text{pname}=x}(\text{Transaction}) \mid (\text{pname}, x) \in \pi_{\text{pname}}(\text{Transaction}) \,\}$$

being a set of (sub)relations. We will refer to this kind of grouping as **split into groups**. An algorithm which splits a relation into groups hence must put every row into its respective group. In other words: a split into groups requires all rows to be loaded in memory (or storage).

Apart from this observation, our definition of G does not further specify the algorithm a database could use to perform the split. Usually, databases simply sort the relation on the grouping attribute, such that rows belonging to the same group are next to each other.

2.3.2 Aggregate on groups

Oftentimes, we do not require all the details of all the rows in a group. Instead, we only want a summary of our grouped data, also known as aggregation. While listing features missing in basic relational algebra, [3] provides a great explanation for aggregation and grouping:

> An extremely common feature of SQL queries is the use of aggregation and grouping. Aggregation allows numerical functions to be applied to entire columns, for example, to find the total salary of all employees in a company. Grouping allows such columns to be split according to a value of some attribute (...)

They also formally define and analyze aggregation as an extension to relational algebra, but we will instead use the (easier) aggregation operator presented in [6], section 8.4.2. Let's describe our aggregation goal with it:

$$_{\text{pname}}\mathfrak{I}_{\text{SUM amount}}(\text{Transaction})$$

$\mathfrak{I}$ is the aggregation operator, `pname` is the grouping column and `SUM` is the aggregation function we are using. For each group of Transaction, the expression sums up the column `amount` of all rows in that group.

Processing this statement could be, similar to 2.3.1, done by sorting the rows of Transaction by pname. One could, however, think of other algorithms. For example: store the name and current sum of each group while iterating over the rows in Transaction. We see that this expression has more opportunity for being optimized.

2.4 External sorting

Usually, a database either sorts the columns or creates a hashtable when eliminating duplicated rows or performing aggregation (see [7]). Let's focus on the sorting for now. So if we want to aggregate on groups with limited RAM, we need to sort with limited RAM.

Sorting data bigger than the available memory can be achieved by buffering temporary results to the storage. Such procedure is also known as external sorting, since it uses external files to store part of its state. Usually, a sort-merge strategy is used. [6] describes such an algorithm in section 18.2.

For our paper, there are two important conclusions:

1. Aggregating on large data will most likely require sorting.
2. We need external sorting strategies to sort data bigger than the available memory.

Most Relational DBMS support external sorting. In our simulation, we will see that MySQL will use such kind of technique to perform aggregation on large data.

2.5 PowerShell's sorting algorithm

So let's have a look at PowerShell's `Group-Object` command. The Microsoft documentation [8] for the Group-Object command doesn't elaborate on the underlying algorithm used. It does, however, reference the return type. `Group-Object` either returns an instance of type `GroupInfo` (see [9]) or a `hashtable` (see [10]).

Reading their documentations, what both these structures have in common is that they are storing all elements from all groups in memory. This means, `Group-Object` is splitting an input list into subgroups, as defined in 2.3.1. The underlying data structures do not make use of external sorting strategies either, as they are holding the elements in memory. This is why PowerShell crashes when trying to group data bigger than the available memory.

3 Experiment

3.1 Methods

We use the following methods to find a solution that groups large CSV data:

1. We started off with a **systematic literature research** to explain the bottleneck of our PowerShell implementation and find an alternative solution.
2. We build a **test**[3] to ensure that the new MySQL implementation does indeed aggregate the data in the same way our PowerShell implementation did.
3. We run an **experiment** where both solutions run in Docker containers limited in memory. We expect the MySQL implementation to run slowly but without crashes due to the external sorting strategy explained in 2.4.

3.2 Generating records

For this paper, we need to implement simple scripts which acts as the service provider introduced in 1.1. The scripts should generate data similar to the export we got from the service provider. Note that:

- The generated data must be large, large enough to exceed the memory.
- Nevertheless, the script itself should not crash from insufficient memory.
- Along with the generated data, the expected result from the aggregation should be provided for comparison.

For these reasons, we decided to let the scripts append the sample relation 1 to the CSV file / MySQL DB over and over again. We will use a large text in the description fields to bloat the records further.

The scripts work by repeating the sample relation 400 times in memory, then writing this chunk of data to a CSV/SQL file for another 200 times. In total, we get $5 * 400 * 200 = 400,000$ records, which are about 403.25MB of record data. Splitting the workload into multiple I/O operations ensures that we can generate all this data on very limited memory.

For more details, see the scripts in the appendix 4.3.

3.3 Using Docker

As explained on Docker's website [11], a container isolates an application from our operating system and files. It bundles the application source

[3] For simplification, we just compare the aggregation result from both implementations on the same test data

code and required dependencies together, which makes the application very portable as long as we got a software which can run our container.

We will use Docker[4] for that. Both our PowerShell and MySQL implementations will be created and run as Docker containers. Thus, the experiment will execute similar on different hosts, making it easy for others to repeat.

Docker uses so-called Dockerfiles to define the parts of a container.

3.4 PowerShell Implementation

Let us have a look at the Dockerfile for our PowerShell implementation.

```
1 FROM mcr.microsoft.com/PowerShell:lts-7.2-ubuntu-20.04
2 COPY . /
3 RUN pwsh /Generate-CSVExport.ps1
4 ENTRYPOINT [ "/docker-entrypoint.sh" ]
```

Listing 2: PS Dockerfile

The container for our PowerShell implementation needs a PowerShell installation. We use the PowerShell Ubuntu package [12] as a base image (line 1) and copy two PowerShell scripts into our container (line 2).

The first script is executed when the container image is created (line 3). It generates the large CSV file as described in 3.2. Note that we decided to ignore the `tid` attribute, as we do not need a Primary Key in CSV.

When the container is running, the second script (line 4) will be executed:

```
1  $csv = Import-CSV "./export.csv"
2  $groups = $csv | Group-Object -Property pname
3  $summary = @{}
4
5  foreach ($group in $groups) {
6      $summary[$group.Name] = 0
7      foreach ($record in $group.Group) {
8          $summary[$group.Name] += [int] $record.amount
9      }
10 }
11
12 $summary
```

Listing 3: aggregate.ps1

It imports all the CSV data (line 1) that we generated before and uses `Group-Object` to group the records by their `pname`. By iterating over the groups and their members, the script calculates the sum of amount for all members of a group. The aggregation is stored in a hashlist `{"group": <current_sum>}` and printed out at in line 12.

We can see that `Group-Object` requires all records to be loaded in memory. For large CSV files, this is not efficient.

[4]More precisely, the Docker Engine, an open-source containerization software.

3.5 MySQL Implementation

So let us look at the MySQL implementation:

```
1 FROM ubuntu/mysql
2 ENV MYSQL_ROOT_PASSWORD="root"
3 ENV MYSQL_DATABASE="grouping"
4 COPY . /
5 RUN /generate-db.sh
```

Listing 4: MySQL Dockerfile

Line 1 references the MySQL image [13], based off Ubuntu. We chose Ubuntu for both PowerShell and MySQL to make the containers more comparable.

We also provide environment variables for the DBMS root password and database name. These variables will be used by the base image.

After copying the generation script, we execute it (lines 4-5). The script creates:

1. An SQL file `a-init-schema.sql` for initializing the schema of our database. We use the schema defined in 2.2.1 and only add the mandatory data types `VARCHAR` and `INT`. We chose the column tid as the primary key.
2. A bunch of identical SQL files `b-init-rows-<i>.sql`, all inserting the same rows into our database over and over again.
3. A bash script `c-run-query.sh` which performs the aggregation query once the initialization is done.

The aggregation query is the equivalent to the operation implemented in PowerShell, summing up the `amounts` for each group grouped by `pname`:

```
1 SELECT pname, SUM(amount)
2     FROM Transaction
3     GROUP BY pname
```

Listing 5: MySQL aggregation

All files are written to `/docker-entrypoint-initdb.d` and prefixed with a-c to specify the order of execution. Like the documentation suggests, the MySQL base image does then run the scripts in alphabetical order once the DBMS has started. From here, it is easy to run the experiment.

3.6 Running the experiment

Preconditions: Docker is installed. A terminal is opened in a directory with all files needed for the PowerShell or MySQL implementation.

We will turn the Dockerfile into a Docker image and use it to run a container with these two commands:

```
docker build -t <tag> .
docker run --memory 1000M --memory-swap 1000M <tag>
```

Listing 6: Run experiment

Note that `<tag>` is just a tag for the image to easily reference it. We could as well look into Docker to find the id of the image and use that.

`--memory` limits the maximum amount of memory a container can use. `--memory`, on the other hand, also limits the size of swap files. By setting both parameters to the same value, we ensure that there is no swap file being used at all, thereby enforcing memory constraints (see [14]). 1000M (Megabytes) is a good start value for our experiments: It is most likely lower than what the Operating System (OS) along with the 403.52MB CSV export need to be loaded, but just enough for MySQL to run and import all the data.

Running the containers with enough memory results in the summary shown below. Compare the container output with the following table to proof that the implementation is working correctly:

pname	SUM(amount)
Alice	0
Bob	160,000
Carl	0
Dan	240,000

Table 4: Result of aggregation

4 Discussion

4.1 Evaluation

We have run the PowerShell container four, the MySQL container two times. While that is not a representative number, the isolation of our experiment within Docker containers indicates that container crashes indeed occure purely because of our application, or more precisely, because a given implementation was not able to perform the grouping operation on such low RAM.

Yet these experiments are just little indications and do neither measure the probability of a grouping operation to succeed, nor the minimum amount of memory needed to consistently finish the aggregation.

Moreover, as we have not looked at the underlying implementation or at least log outputs from MySQL and PowerShell, these experiments do not fully proof our assumptions.

Precisely, we do not know if MySQL actually used an external sorting algorithm as explored in 2.4. We argue that MySQL must have stored parts of its calculation to the disk, because the available memory was *probably* lower than the 403.25M the CSV data consumes. But we have not measured the memory usage and do not know how much the OS and other processes actually occupied.

4.2 Findings

Running the PowerShell implementation yielded the following results:

- **Import CSV:** The aggregation crashes on 1000MB and 1500MB when trying to import the whole CSV file.
- **Group-Object:** The aggregation crashes on 1700MB and works on 1800MB when trying to group the records imported before.

Not only did we proof that PowerShell needs a wasteful amount of memory to import the whole CSV file. If we compare the numbers of 1500MB (when the import still crashed) and 1800MB (when the grouping succeeded), we can also see that `Group-Object` has less than $1800 - 1500 = 300$MB of memory to perform. These 300MB are less than the size of the CSV file, but we found out in 2.5 that PowerShell'S `Group-Object` returns either a `GroupInfo` or a `hashlist` containing all records. How does this work?

While PowerShell does copy the imported records into their respective group, since they are objects, it does so **by reference** (see [15]). Copying by reference is a technique where only a reference to the object itself is copied. Now both variables point to the same object, thus any change to the copy would be reflected by the original.

Compare this to the results of the MySQL implementation:

- **Startup:** Initialization of the DB and inserting 400,000 records takes significantly more time.
- **Crash:** The implementation crashes at 700MB.
- **Success:** The implementation returns the correct sums at 1000MB.

Given that both implementations are based off Ubuntu, it is surprising that MySQL achieves aggregation with 1000MB. PowerShell needs more memory just to load the same dataset into memory. This is why we assume that the dataset is larger than the available memory (it was for PowerShell), and that MySQL manages this by using external sorting strategies to group the data.

This experiment is a very basic example of grouping large data. Of course, the data mentioned in 1.1 is far more complex, and so was the aggregation process. Nevertheless, the principles of this experiment still apply: simple grouping commands like PowerShell's `Group-Object` command are not able to handle large data. Only by using external sorting strategies, MySQL manages to aggregate data much larger than the available memory.

4.3 Future Work

As explained in 4.1, the results from this experiment are not certain. The experiment must be turned into a series of experiments in order to find a statistic significance and a minimum amount of RAM needed for our grouping operation. Furthermore, monitoring tools could be used to examine how much RAM of the container is actually assigned to our MySQL/PowerShell implementation.

Comparing a simple, versatile PowerShell command to a full-featured DBMS is not fair. A future study could focus on more specialized libraries or frameworks for grouping data in PowerShell.

Usually, the analysis of large data is done for a company goal. Communicating results and collaborating with other people might require visualization. In fact, this is what our company is going to do next: build a PowerBI visialization of the data mentioned in 1.1.

Appendix Scripts

Script for generating CSV records (PowerShell implementation):

```
$header = "pname,amount,comment"
$comment = "a" * 1000        # 1kb of data

$header | Out-File -FilePath "./export.csv"

# These rows contain a bit more than 2mb of data
$rows = @"
Alice,-1,$comment
Bob,2,$comment
Alice,1,$comment
Carl,0,$comment
Dan,3,$comment`n
"@ * 400

# Putting more than 400mb of data
for ($i = 0; $i -lt 200; $i++) {
    $rows | Out-File -FilePath "./export.csv" -Append
}
```

Listing 7: Generate-CSVExport.ps1

Script for initializing MySQL implementation:

```
#!/bin/bash

# a. create schema
schema=$"USE grouping;

CREATE TABLE Transaction (
    tid int AUTO_INCREMENT NOT NULL,
    pname VARCHAR(20) NOT NULL,
    amount int,
    comment VARCHAR(1024),
    PRIMARY KEY (tid)
);"
echo "$schema" > /docker-entrypoint-initdb.d/a-init-
    schema.sql

# b. insert data
comment=$(printf 'a%.0s' {1..1000})
command="INSERT INTO Transaction (pname, amount, comment
    ) VALUES"
sql=$"('Alice', -1, '$comment'),
('Bob', 2, '$comment'),
('Alice', 1, '$comment'),
('Carl', 0, '$comment'),
('Dan', 3, '$comment'),"
sql=$(printf "$sql\n%.0s" {1..400})
sql=${sql%?};

for i in {1..200}
```

```
do
    echo "$command$sql" > "/docker-entrypoint-initdb.d/b-init-rows-${i}.sql"
done

# c. run grouping query
query="mysql -uroot -proot -e 'USE grouping; SELECT pname, SUM(amount) FROM Transaction GROUP BY pname;'"
echo "$query" > "/docker-entrypoint-initdb.d/c-run-query.sh"
```

Listing 8: generate-db.sh

References

[1] Y. Shafranovich, “Common Format and MIME Type for Comma-Separated Values (CSV) Files,” RFC 4180, Oct. 2005. [Online]. Available: https://www.rfc-editor.org/info/rfc4180

[2] E. F. Codd, “A relational model of data for large shared data banks,” *Commun. ACM*, vol. 13, no. 6, p. 377–387, jun 1970. [Online]. Available: https://doi.org/10.1145/362384.362685

[3] M. Arenas, P. Barcel´o, L. Libkin, W. Martens, and A. Pieris, *Database Theory Querying Data*, preliminary ed. San Francisco, CA, USA: Santiago Paris Bayreuth Edinburgh, 2022.

[4] N. Schweikardt, T. Schwentick, and L. Segoufin, *Database theory: query languages*, 2nd ed. Chapman & Hall/CRC, 2010, p. 19.

[5] T. Halpin and T. Morgan, *Information Modeling and Relational Databases*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[6] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[7] J. Edgar. Algorithms for sql query operators. Simon Fraser University. [Online]. Available: https://www2.cs.sfu.ca/CourseCentral/454/johnwill/cmpt454_04queries.pdf

[8] Microsoft, “Group-object,” Microsoft. [Online]. Available: https://learn.microsoft.com/de-de/powershell/module/microsoft.powershell.utility/group-object?view=powershell-7.4

[9] ——, “Groupinfo class,” Microsoft. [Online]. Available: https://learn.microsoft.com/de-de/dotnet/api/microsoft.powershell.commands.groupinfo?view=powershellsdk-7.4.0

[10] ——, “Hashtable class,” Microsoft. [Online]. Available: https://learn.microsoft.com/de-de/dotnet/api/system.collections.hashtable?view=net-8.0

[11] Docker, “What is a container,” Docker, accessed: 2023-05-17. [Online]. Available: https://docs.docker.com/guides/walkthroughs/what-is-a-container/

[12] D. Microsoft, “microsoft-powershell - official image — docker hub,” Docker, accessed: 2023-05-17. [Online]. Available: https://hub.docker.com/_/microsoft-powershell

[13] D. Canonical, “ubuntu/mysql - docker image — docker hub,” Docker, accessed: 2023-05-17. [Online]. Available: https://hub.docker.com/r/ubuntu/mysql

[14] Docker, “Runtime options with memory, cpus, and gpus,” Docker, accessed: 2023-05-17. [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/#limit-a-containers-access-to-memory

[15] Microsoft, “about ref - powershell — microsoft learn,” Microsoft, accessed: 2023-05-17. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_ref?view=powershell-7.4