Managing Data…and Covid
An Experience Report

Alan G. Labouseur
Alan.Labouseur@Marist.edu
Marist College
Poughkeepsie, NY, USA

ABSTRACT

The Covid-19 pandemic presented a vast array of challenges for professors and students the world over. As we navigate Covid's long tail, many challenges remain. Those challenges – nasty as they are – can be recast in a constructive light and imbued with pedagogical significance as practical, common-ground motivational tools for teaching topics in data management. The author has done just that. In addition to teaching on full-time faculty in the Computer Science department at Marist College, he was a key member of their Covid-19 screening team. After designing and implementing data management systems for generating representative samples of the college population for surveillance testing, results tracking, and compliance monitoring, he used those experiences in new, hands-on ways to integrate data management theory with real-world practice in his classes. This experience report – wherein the author explains this journey and notes lessons learned – is an example of how, even once this pandemic has receded fully into the past, experiences like these can provide opportunities for educators to incorporate timely topics into their data management courses.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Information systems** → **Database design and models**; *Database management system engines*; *Database views.*

KEYWORDS

Motivation and Engagement, Practice and Theory, Undergraduate Data Management Courses

ACM Reference Format:
Alan G. Labouseur. 2022. Managing Data…and Covid *An Experience Report*. In *1st International Workshop on Data Systems Education (DataEd'22), June 17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3531072.3535328

1 INTRODUCTION

Students commonly complain that their courses lack "real world" applications. As teachers, we are always looking for authentic, hands-on ways to address this and better reach our students. While the Covid-19 pandemic presented many challenges, it also presented new and meaningful ways to inform data management theory with real-world practice in an immediate and authentic manner. (See the author's prior work [2] for details on applying Covid experiences to Algorithms and Operating Systems courses too.) This was not a one-time thing. New variants and ever-changing response models have been keeping us on our collective toes for a while. And since it looks like Covid may persist for some time, we might as well make the best of it.

As a member of the Covid testing team at Marist College[1], I designed and implemented database systems for generating representative samples of the college population for surveillance testing, recording test results, and monitoring testing compliance. My experience in real-world testing and tracking revealed new ways to inform data management theory with practice. This experience report presents my efforts to use it to motivate and illustrate core concepts in an undergraduate data management course, covering aspects of SQL, database design, and stored procedure programming, all in an immediate and authentic manner made possible by the fact that faculty and students were living the same first-hand experience in real time.

2 BACKGROUND

Many aspects of the Covid-19 pandemic could be used as examples for teaching data management. There are, for instance, several ways we might make use of graph databases (for disease spread models and exploring graph algorithms for computing traits like network density, maximum cliques, connected components, clustering coefficients, and the like). Those seem like productive topics but I chose to focus on a different area: Covid testing.

2.1 Covid Testing on Campus

A team of healthcare professionals, faculty, staff, and administrators conducted Covid screening via pooled surveillance testing [1]. As part of that team I incorporated several aspects of this endeavor into my data management classes. Generating daily representative samples of people from our community and tracking their compliance (because, believe it or not, every day there were people who skipped their test) provided a vivid landscape for discussing SQL queries, functions, subqueries, joins, and views as well as relational database design and stored procedure programming.

At the beginning of each semester I received a snapshot of student and employee data from *Banner*, our higher education ERP system, which I imported into the PostgreSQL object-relational database in a table called People, shown in Figure 1. A month or so

DataEd'22, June 17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9350-8/22/06…$15.00
https://doi.org/10.1145/3531072.3535328

[1] I was among only three people at Marist approved for access to this sensitive data.

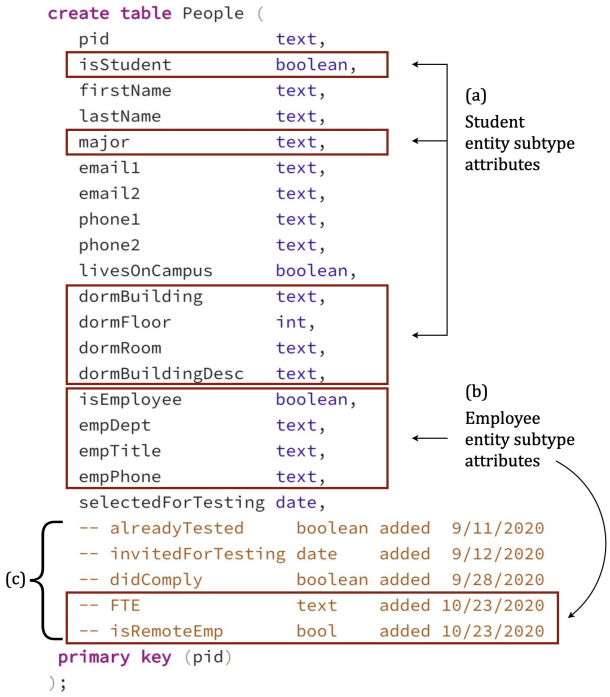

Figure 1: The People table in PostgreSQL (originally published in [2])

into surveillance testing it became clear that more attributes were needed, which I added, as noted in Figure 1c.

3 SQL

With our People data in place – and purposefully ignoring any design issues until later – we began discussions of elementary SQL.

3.1 Simple Queries

There are a plethora of obvious and simple queries to use here, e.g., on-campus computer science majors:

```
select lastName, firstName
from People
where isStudent
  and livesOnCampus
  and major = 'Computer Science';
```

We would revisit this query later to illustrate the power of views. Seeing this query in its original context helped ground our discussion of query rewriting and view execution later on.

3.2 Aggregate Functions

Though I did not find many uses for `MIN`, `MAX`, or `AVG`, there were opportunities for `COUNT` and `SUM`.

```
-- Students and non-students
select isStudent, count(pid)
from People
group by isStudent
order by isStudent ASC;

-- People who are both a student AND an employee
select count(pid)
from People
where isStudent and isEmployee;

-- People living on campus
select sum(subtotal)
from (select dormBuilding, dormFloor,
             count (pid) as subtotal
      from People
      where livesOnCampus
      group by dormBuilding, dormFloor
      order by dormBuilding, dormFloor) sub1;
```

In addition to exemplifying the use of `GROUP BY` and `ORDER BY`, these queries provided a good place to warn students about the effects of `NULL` on aggregate functions and why we wanted to count `pids` instead of rows using `COUNT(*)`. Students tend to have a difficult time with `NULL`, three-value logic, the need for the `is` operator, and what `coalesce` does. Exploring examples like this early on help clarify the theory with relatable practice. We would return to the annoyances resulting from `NULLs`[2] when discussing database design later on, and in the context of subqueries next.

3.3 Subqueries

The use of subqueries is a convenient way to begin showing students how to query data from multiple tables. It is also a natural place to talk about operators like `in` (see below) and `exists` (as well as their negations) and a useful segue into the set operations of union, intersection, and difference.

With simple queries and aggregate functions already in their SQL "toolbox", subqueries present an opportunity to introduce the idea of combining newly learned SQL elements with those that students already know. The query below illustrates this by combining aggregate functions and coalescing `NULLs` into a subquery, then grouping and ordering the results.

```
-- Exclusion counts by month of re-inclusion
select count(pid), extract(month from excludeStop)::text
       || ' - ' || coalesce(to_char(to_date(
     (extract(month from excludeStop)::text),'MM'), 'Month'),
       'No re-inclusion date') as "ReIncludeMonth"
from Excludes
where pid in (select pid from People)
group by "ReIncludeMonth"
order by "ReIncludeMonth" ASC;
```

If we are not careful combining tables with subqueries we can get wacky and nonsensical result sets. Addressing this makes for a good segue into primary and foreign keys, referential integrity, and strong vs. weak entities. Calling back to the annoyances of `NULLs` naturally motivates bringing up entity subtypes, which we discuss later in the semester (and later in this paper, in Section 4).

[2] See my Stack Overflow discussion for more on NULL behavior: https://stackoverflow.com/questions/5834471/

3.4 Joins

With the students' SQL "toolbox" getting fuller, it was finally time to introduce joins through the use case of monitoring testing compliance. We tested and tracked students by clusters of dormitories, which makes for a simple inner join query:

```
select d.building, p.*
from People p inner join Dorms d
     on p.dormBuilding = d.building
order by d.building ASC, p.lastName ASC;
```

Cross-referencing students to their dorms made for a join example everybody could relate to. Skipping a surveillance testing appointment was another example that students could (sadly) relate to.

```
-- Students who were selected for testing
-- but remain UNtested  according to the lab results.
select d.cluster, p.dormBuilding, count(pid) as "UNtested"
from People p inner join Dorms d
     on p.dormBuilding = d.building
where p.livesOnCampus and p.isStudent
  and selectedForTesting is not null
  and not p.previouslyTested
group by d.cluster, p.dormBuilding
order by dayNameToDayNumber(d.cluster) ASC,
         p.dormBuilding ASC;
```

That query contains a user-defined function, "dayNameToDayNumber", which I did not cover in class, but which could, I think, be incorporated into a discussion of stored procedures. (See Section 5.)

Members of our community would get excluded from testing for various reasons (e.g., players on a traveling sports teams, off-campus interns, our ever-changing quarantine population). To analyze test results for only active members of our population we would use another inner join, this time in combination with case-variant column output, an aggregate function, and that user-defined function.

```
-- Tested and UNtested ON-CAMPUS students by cluster/dorm
select d.cluster, p.dormBuilding,
       case
          when previouslyTested     then 'tested'
          when not previouslyTested then 'UNtested'
       end as "tested?", count(pid)
from People p inner join Dorms d
     on p.dormBuilding = d.building
where p.livesOnCampus and p.isStudent
group by d.cluster, p.dormBuilding, "tested?"
order by dayNameToDayNumber(d.cluster) ASC,
         p.dormBuilding ASC, "tested?" DESC;
```

For overall tracking we wanted everybody who was ever tested cross-referenced with the data from the People table. Since we want **all** of the already-tested population, but some may have been excluded, an outer join is called for.

```
-- Already tested at least once
select p.*
from alreadyTested t left outer join People p
     on t.id = p.pid;
```

3.5 Views

To illustrate the utility of views, we created them for OnCampusStudents, OffCampusStudents, and Employees.

```
create or replace view OnCampusStudents
as
select lastName, firstName
from People
where isStudent
  and livesOnCampus;
```

With these views in hand I had my students look at some of their earlier queries with instructions to use the views instead of base tables:

```
select lastName, firstName
from OnCampusStudents
where major = 'Computer Science';
```

This led to a discussion of view definitions being stored in the system catalog and an illustration of query rewriting. With this background, we wrote more views that would help simplify reasoning about members of our campus community who were currently excluded from testing.

```
create or replace view CurrentlyExcluded
as
select pid
from Excludes
where excludeStop is null
   or now()::date between excludeStart and excludeStop;
```

We would revisit the People table in the context of stored procedures later on. But before we could do that we would take up relational database design and address some of the People table's pressing deficiencies.

4 RELATIONAL DATABASE DESIGN

The deficiencies of the People table were made clear by the presence of NULLs in the data due to some attributes being relevant only to certain rows. This informed our initial database design discussions. After covering the normal forms and developing an appreciation for Codd and his rules, we looked to revise the People table. It was now apparent that there are multiple (sub)types of people and that we have a few different attributes for each. The fact that student-only and employee-only views eased query writing provided an excellent opportunity to introduce entity subtypes and their implementation as one-to-one relationships with optional participation on the subtype side. (This also proved to be a good time to note the use of a primary key simultaneously as a foreign key in the subtype tables.) We factored out the student and employee attributes from the People table as noted in Figures 1a and 1b into subtype entities. Having done this, we noted that we no longer needed the isStudent and isEmployee attributes, that functionality now being accomplished via pid membership in the new entity subtype tables. This led to a discussion of logical data independence, demonstrated by rewriting the OnCampusStudents, OffCampusStudents, and Employees views to make use of the new subtype tables and then executing unchanged our earlier queries against those revised views.

```
-- Asymptomatic surveillance COVID screening
-- The PostgreSQL random() function is based on the POSIX-standard erand48() family.
-- It's not good enough for cryptography, but it's plenty good for us.
create or replace function choosePeopleToTest()
   returns table (
        ...
    )
language plpgsql
as
$$
declare
   currentCluster       text;
   clusterRow           record;
   dormRow              record;
   goBackAtLeastThisFar date;
begin
   -- Establish the date prior to which we'll consider re-inviting people for testing.
   goBackAtLeastThisFar := (current_date - interval '13 days');
   for clusterRow in ( select distinct dayNameToDayNumber(d.cluster), d.cluster
                       from Dorms d
                       order by dayNameToDayNumber(d.cluster) ) loop
      currentCluster := clusterRow.cluster;
      for dormRow in (select building, numToTest
                      from Dorms d
                      where d.cluster = currentCluster
                      order by d.building) loop
         insert into workTable
         select p.pid,
                ...
         from People p inner join Dorms d on p.dormBuilding = d.building
         where p.livesOnCampus
           and p.isStudent
           and ( (p.selectedForTesting is null)
                 or
                (p.selectedForTesting < goBackAtLeastThisFar) )
           and p.dormBuilding = dormRow.building
         order by previouslyTested ASC, random()
         limit dormRow.numToTest;
      end loop; -- for dormRow
      -- This completes one cluster. Loop to the next.
   end loop; -- for clusterRow
   -- That's it. We're done. Return the results.
   return query (select * from workTable);
end;
$$
```

Figure 2: A portion of a PL/pgSQL stored procedure (originally published in [2])

5 STORED PROCEDURE PROGRAMMING

Near the end of the semester, armed with a full SQL "toolbox", we considered how to generate representative samples of our community for Covid screening. Figure 2 shows (most of) the PL/pgSQL code for pseudo-randomly selecting on-campus students who had not been recently tested or recently selected for testing, grouped by dormitory clusters. (The code for selecting off-campus students and employees is substantially similar, and excluded in the interest of space and anti-redundancy.) This code was used to demonstrate many features of stored procedures: local variable declaration and usage, date functions, (nested) for loops, advanced SQL inside of control structures, table manipulation, and the need for good formatting and comments to make sense of it. My students felt strongly connected to this because, by this time in the semester, all of them had been selected for Covid screening by this very code.

As mentioned in Section 3.4, it is natural to bring up user-defined functions in the same context as stored procedures (though their details are a bit different). Figure 3 shows the code for the "dayNameToDayNumber" function we used in the section on joins. It was quite useful in `ORDER BY` clauses to impose a natural weekday order on query results.

```
create or replace function dayNameToDayNumber(IN dayValue text) returns int
language plpgsql
as
$$
declare
   dayName   text;
   dayNumber int;
begin
   dayName := upper(dayValue);
   dayNumber = -1;
   case
      when dayName = 'MONDAY'    then dayNumber = 1;
      when dayName = 'TUESDAY'   then dayNumber = 2;
      when dayName = 'WEDNESDAY' then dayNumber = 3;
      when dayName = 'THURSDAY'  then dayNumber = 4;
      when dayName = 'FRIDAY'    then dayNumber = 5;
      when dayName = 'SATURDAY'  then dayNumber = 6;
      when dayName = 'SUNDAY'    then dayNumber = 7;
      else dayNumber = 0;
   end case;
   return dayNumber;
end;
$$
```

Figure 3: A PostgreSQL user-defined function

6 CONCLUSION

All told, this material addressed many areas of the "IM/Database Systems and IM/Data Modeling Core-Tier2" component of the ACM Computer Science Curricula 2013 Body of Knowledge in a practical and meaningful way. It is no surprise that circumstances affecting all of us should be well suited to motivate various topics in our data management courses. After a few semesters, my experience bears this out, and it aligns with the opinions of my students. This quote from a recent student puts it nicely:

> I wanted to thank you for an interesting and academically engaging semester. . . .But most of all I really appreciated how you connected the material in the course to the pandemic instead of just pretending it was a regular semester. The connection of CS courses to the real world has been something I have wanted to see more of in my classes so I really enjoyed that factor of the semester project. – Maria

6.1 Online Resources

Much of my SQL and PL/pgSQL source code (but none of the data) is available on GitHub at
https://github.com/Labouseur/CovidInTheClassroom
You are welcome and encouraged to use it. . . and to improve it.

ACKNOWLEDGMENTS

I sincerely thank Dean Roger Norton of the School of Computer Science and Mathematics for suggesting the use of my Covid testing work as the basis for this and other papers. Heartfelt thanks also go out to Dean Alicia Slater of the School of Science for trusting me with all that sensitive data, answering every question I had, and allowing me to be her sidekick in fighting this pandemic.

REFERENCES

[1] Ding-zhu Du and Frank K Hwang. 2000. *Combinatorial Group Testing and its Applications* (2nd ed.). Vol. 12. World Scientific, USA.
[2] Alan G Labouseur. 2021. COVID in the Classroom. *ACM Inroads* 12, 3 (2021), 32–38.