Relational Databases

Data Analytics Beyond Spreadsheets and Single-Table Databases

Bill Adams
Independent Contractor, Lava-Data

Link to Slides Available at End

About Bill Adams

My name is Bill Adams. I have over two decades of experience managing data in 24/7 high volume factories. I started off as a test and yield engineer. The value I brought was in having a degree in electrical engineering with a side of computer science. I have recently become an independent contractor and formed Lava Data.

SQL

To understand a relational database, one must understand SQL. It is written S-Q-L but generally pounced sequel, like a movie sequel. S-Q-L is a TLA for structured query language and it is the language used to extract data out of a relational database. It basically amounts to a limited, strict English statement for extracting data. SQL is fairly standard across relational databases, in the same way English is standard between America and Brittan.

SQL

Microsoft SQL Server

MySQL

To add to the confusion, these are pronounced Microsoft sequel server, and My S-Q-L. The pronunciation of MySQL is the outlier. And because I grew up on MySQL, you may hear me mispronounce SQL.

Now for your crash course on SQL, which is very important for working with databases.

Spreadsheet

	A	B	C	D	E
1	**Name**	**Hair**	**Zip**		
2	Shaggy	Dusty Bond	97214		
3	Velma	Brown	97214		
4	Fred	Blond	97701		
5	Daphne	Red	97701		
6					
7					

People

Here is a mock up of a simple spreadsheet. The first row is a header to indicate what is going to be found in that column. Then there are some rows of data where the data in a row, not a column, is related to one another. The sheet name is "People." Exciting stuff, right?

Spreadsheet → Row Based Database

	A	B	C	D	E
1	**Name**	**Hair**	**Zip**		
2	Shaggy	Dusty Bond	97214		
3	Velma	Brown	97214		
4	Fred	Blond	97701		
5	Daphne	Red	97701		
6					
7					

People

A spreadsheet in this form, with a header row and related data in a row is a great metaphor of a row-based database-table. There are other table types, such as column based or document databases, generally and collectively referred to as NoSQL – which counterintuitively means Not Only SQL. However, this talk is focused on relational, row-based tables. When I say table or database table, I mean row-based databases table.

Spreadsheet → Row Based Database

	A	B	C	D	E
1	**Name**	**Hair**	**Zip**		
2	Shaggy	Dusty Bond	97214		
3	Velma	Brown	97214		
4	Fred	Blond	97701		
5	Daphne	Red	97701		
6					
7					

People

People

ID	Name	Age	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

On the right is the same data as a database table. The header row becomes the column name, also known as the field name. For example, this first spreadsheet column is referenced by “A” but as a database table it is called “name”   The spreadsheet row number is analogous a unique ID, that is a row identifier that is unique and not found in any other row. The sheet name becomes the table name. Any empty columns and rows are no longer “just there”, although as more data is inserted into the table, new rows are added and more storage is consumed.

Spreadsheet

A:A or People!A:A

	A	B	C	D	E
1	**Name**	**Hair**	**Zip**		
2	Shaggy	Dusty Bond	97214		
3	Velma	Brown	97214		
4	Fred	Blond	97701		
5	Daphne	Red	97701		
6					
7					

People

In a spreadsheet, one can click the column letter -- "A" in this case – to select the entire column and its data. In a spreadsheet formula, it is written A:A or People!A:A where People is the name of the sheet.

SQL

	A	B	C	D	E
1	Name	Hair	Zip		
2	Shaggy	Dusty Bond	97214		
3	Velma	Brown	97214		
4	Fred	Blond	97701		
5	Daphne	Red	97701		
6					
7					

People

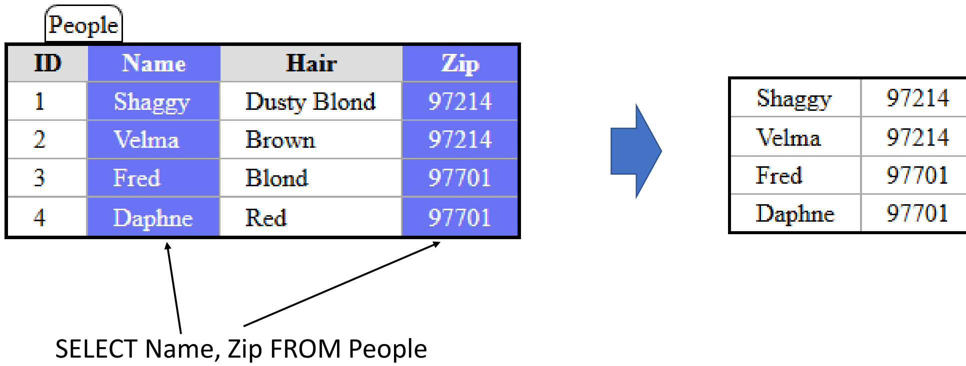
=People!A:A

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

SELECT Name FROM People

To achieve the same result in a relational database with SQL, the SQL statement is SELECT Name FROM People.



To select more than one column, list the columns separated by commas. Here I am selecting the Name and Zip code, and the results returned from the database contain only those columns.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

SELECT Name, Zip FROM People

Name	Zip
Shaggy	97214
Velma	97214
Fred	97701
Daphne	97701

The database does return the column name in the results, which I've added here. This is important for both clarity and because…

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

MeddlingKid	Zip
Shaggy	97214
Velma	97214
Fred	97701
Daphne	97701

SELECT Name AS MeddlingKid, Zip FROM People

It is also possible to rename the column in the results using the AS keyword.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

SELECT **Name, Zip** FROM People

Name	Zip
Shaggy	97214
Velma	97214
Fred	97701
Daphne	97701

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

SELECT **Zip, Name** FROM People

Zip	Name
97214	Shaggy
97214	Velma
97701	Fred
97701	Daphne

The order of the columns in the select effects the order of the columns in the results.

Normally one need not be concerned about this. None the less it is good to be aware.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

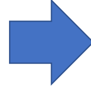
SELECT * FROM People

There is a special select -- the star -- which selects all columns in a table. Note that it is star and not asterisk in the parlance of database administrators. Say asterisk and it will be known a poser is in the midst.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701



ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

```
SELECT * FROM People
SELECT ID, Name, Hair, Zip FROM People
SELECT People.* FROM People
SELECT People.ID, People.Name, People.Hair, People.Zip FROM People
```

Here are alternative syntaxes which all produce equivalent results. The second statement selects each column individually. The third and fourth statements use the table dot field syntax.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

MeddlingKid	Zip
Shaggy	97214
Velma	97214
Fred	97701
Daphne	97701

```
SELECT Kid.Name AS MeddlingKid, Kid.Zip FROM People Kid
```

It is also possible to alias tables. Unlike aliasing columns, it is done without the AS keyword. You can see that in the table dot field syntax, the table alias has to be used. If I had written People.Zip it would produce an error when I try to run the query.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214

SELECT * FROM People **WHERE Zip = ‘97214’**

A where clause acts like the filter in a spreadsheet. Where clauses can be very complex and powerful, and is one place where a relational database really shines versus spreadsheet filters. In this example, there is a single criteria in the where clause to limit the data to only people in the 97214 zip code.

SQL

People

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

ID	Name	Hair	Zip
1	Shaggy	Dusty Blond	97214

SELECT * FROM People **WHERE Zip = '97214' AND Name = 'Shaggy'**

To filter on multiple columns, use the AND keyword. For a row to be included all of the criteria has to be met. Row 2 has a matching zip code value, but the name does not match, so it is not included in the results.

By the way, for the most part capitalization does not matter. For example the SELECT, FROM, WHERE, AND keywords could be lower-case. Personally I think having all-caps keywords increases readability.

SQL

SELECT * FROM People WHERE Age = 35

Field Name, Stored as an Integer — No Quotes

SELECT * FROM People WHERE Name = 'Fred'

Field Name, Stored as a String — Quotes

Fields in a database table have a type such as string, date or one of the various numeric types. When the field type in the table is a number it does not get quotes. Pretending I had a field of age, which is presumably an integer, the SQL would look like the top query. No quotes around the value we want to match. When the field is a string, such as a name, the value gets quotes around it. Without quotes, the database interprets the string as a field identifier. "Name" on the left side of the equals has no quotes: It is a table field. Fred on the right has quotes: it is a literal value to match.

SQL

SELECT * FROM People WHERE Zip = 97214

Stored as a String

Integer (no quotes)

Performance Hit from Conversion of String to Integer

Here is a pro tip that will put you ahead of many folks with database experience. If you have a string field in a table, such as a zip code – remember zip codes can have a dash in them – and your where criteria does not use a quoted value, it will work BUT it will suffer a performance hit because, to make the comparison and check for equality, the database will convert the zip code stored in the table to a number.

SQL

SELECT * FROM People WHERE Name = “Fred”

Also of note is that single quotes are standard. Double quotes may or may not work depending on the database. It is best to get in the habit of using single quotes. I found this out the hard way when I switched databases and had to do a massive code and query update.

SQL

```
SELECT    Table1.Field1,    Table1.Field2    FROM    Table1    WHERE    Field1 = 'Value'
```

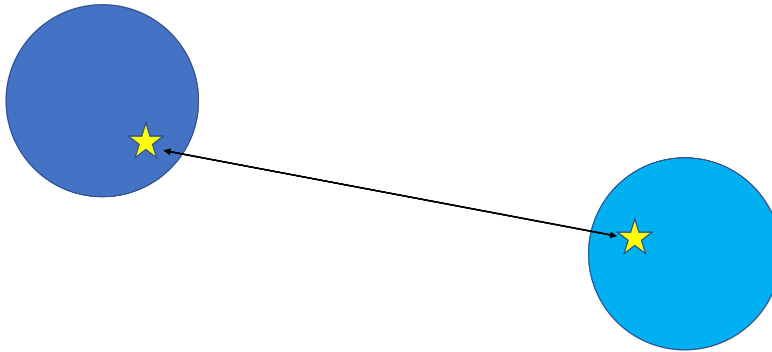
With that introduction to SQL, we are now ready to talk about relational databases.

Relational Database Management System (RDBMS)

Often you will hear “RDBMS” thrown around. Most likely you are not interested in the “management system” part of that acronym. That is up to your IT department. The management system is, for example, Microsoft SQL Server or MySQL. They both support relational data. Relational data is a first class citizen. They also support other types of data.

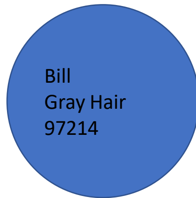
The reason you want to know all of this is for the common language with database folks – the administrators and vendors.

Relational Data



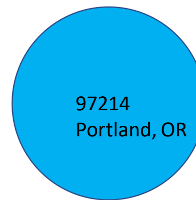
When there are two or more sets of data and some information links them, it is relational data.

Relational Data



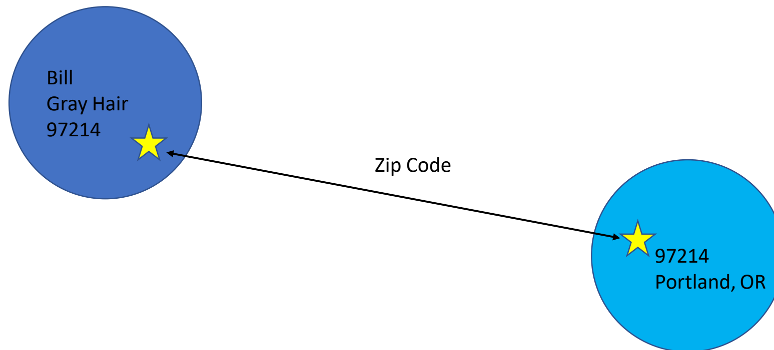
For example, here are some attributes about me. My name is Bill. I have gray hair, well at least what's left of it. And my office is in the 97214 zip code.

Relational Data



Here is another bit of data. The zip code 97214 is in Portland Oregon.

Relational Data



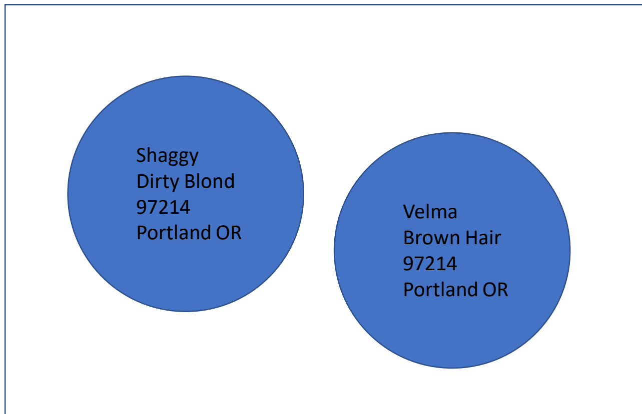
Through the relation of the zip code, we join the two bits of data and you know my office is in Portland Oregon. You probably already did this in your head. . Congratulations! You are a human relational database. It’s easy when there is only one zip code to remember. In order to further our discussion, I need to cover a few database concepts.

Database Concepts

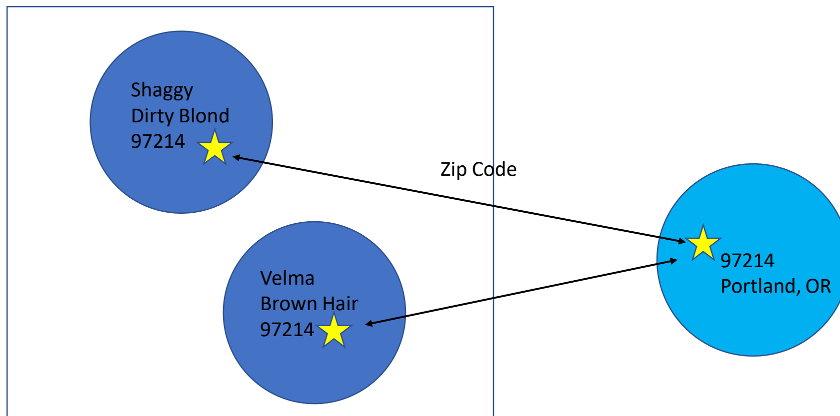
- Unnormalized
- Normalization
- Denormalized

There are important database concepts, especially for relational databases. They are unnormalized, normalization, and denormalization. I'll talk about all three. To start with.. Unnormalized.

Unnormalized



In this example, we have two people both with the zip code 97214 and city/state of Portland Oregon. Because these two share redundant data, and the same information is in each data bubble, it is unnormalized.



You can see here we moved the common data, the city and the state, into its own bubble and link – join if you will – the two on the zip code. When the common data is moved into its own data bubble, it is called normalization.

Normalization

	A	B	C	D	E
1	**Name**	**Hair Color**	**Zip**	**City**	**State**
2	Shaggy	Dusty Blond	97214	Portland	Oregon
3	Velma	Brown	97214	Portland	Oregon
4	Fred	Blond	97701	Bend	Oregon
5	Daphne	Red	97701	Bend	Oregon

	A	B	C
1	**Name**	**Hair Color**	**Zip**
2	Shaggy	Dusty Blond	97214
3	Velma	Brown	97214
4	Fred	Blond	97701
5	Daphne	Red	97701

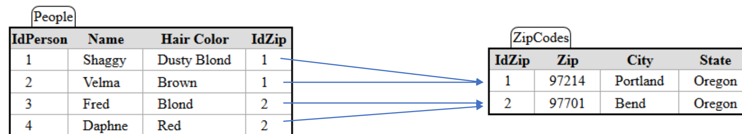
	A	B	C
1	**Zip**	**City**	**State**
2	97214	Portland	Oregon
3	97701	Bend	Oregon

In terms of a spreadsheet, you can think of it like moving the common data to its own sheet. Here we see the human members of the Scooby-Doo gang. In the top sheet, the four members share two cities. Doing a first-order normalization, we move the zip code and city information into its own sheet. An important thing to note here is that the number of cells with data (excluding the column headers) went from 20 in the unnormalized down to 18 in the normalized. This is not drastic in this small example, but if you think about this in terms of millions of records, the storage saving can be significant.

Normalization

People

ID	Name	Hair Color	Zip	City	State
1	Shaggy	Dusty Blond	97214	Portland	Oregon
2	Velma	Brown	97214	Portland	Oregon
3	Fred	Blond	97701	Bend	Oregon
4	Daphne	Red	97701	Bend	Oregon



People

IdPerson	Name	Hair Color	IdZip
1	Shaggy	Dusty Blond	1
2	Velma	Brown	1
3	Fred	Blond	2
4	Daphne	Red	2

ZipCodes

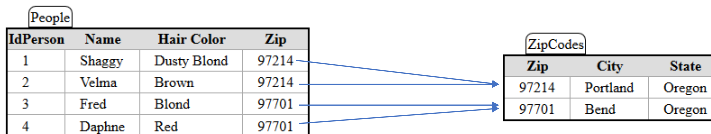
IdZip	Zip	City	State
1	97214	Portland	Oregon
2	97701	Bend	Oregon

The database denormalization looks very similar. There are a couple of different ways to do it in this example. We could move the zip code information to its own table with an ID linking the data. Notice that I have renamed the “ID” from the unnormalized table to IdPeople. It is a very good practice to name an ID field like that…by “ID” and the table-name. Imagine there are a hundred tables and each has a field named “ID”. Figuring out how the data is linked could get very confusing very quickly. Using a field name in this way helps humans figure out how to join the data by inspection.

Normalization

People

ID	Name	Hair Color	Zip	City	State
1	Shaggy	Dusty Blond	97214	Portland	Oregon
2	Velma	Brown	97214	Portland	Oregon
3	Fred	Blond	97701	Bend	Oregon
4	Daphne	Red	97701	Bend	Oregon



People

IdPerson	Name	Hair Color	Zip
1	Shaggy	Dusty Blond	97214
2	Velma	Brown	97214
3	Fred	Blond	97701
4	Daphne	Red	97701

ZipCodes

Zip	City	State
97214	Portland	Oregon
97701	Bend	Oregon

Alternatively, and probably better in this case, is to use the zip code as the link. Now the question is how do we join this data back together? Don't normalize in the first place is NOT the correct answer! It could be a database you do not control. Or the normalization could be for storage savings. Or, as you'll see later, maybe you are joining data from disparate sources.

Joining the Data in a Relational Database

All columns from all tables.
Repeated column names (Zip in this case)
will only be reported once.

Table.Field

SELECT * FROM People, ZipCodes WHERE People.Zip = ZipCodes.Zip

Two tables, separated by commas.

The values are the same.

This is how you join two tables in a relational database with SQL. We are getting all columns from all tables with the star. Both tables have a field called Zip. And we JOIN the tables using the Zip field in each table. Even though both table have a Zip field, the database will only return one Zip in the results. We list the tables FROM which we want to extract separated by commas..

Joining the Data in a Relational Database

SELECT * FROM People, ZipCodes WHERE People.Zip = ZipCodes.Zip

IdPeople	Name	Hair	Zip	City	State
1	Shaggy	Dusty Blond	97214	Portland	Oregon
2	Velma	Brown	97214	Portland	Oregon
3	Fred	Blond	97701	Bend	Oregon
4	Daphne	Red	97701	Bend	Oregon

Here are the results we get back. It looks like the unnormalized data.

A Relational Database CAN Join the Data on the Server

Multiple Tables -> One Dataset Result

A defining characteristic of a relational database is that it can join data on the server and return one result set.

Non-Relational Database Join

SELECT * FROM People

	A	B	C	D	E	F	G
1	IdPeople	Name	Hair Color	Zip			
2	1	Shaggy	Dusty Blond	97214			
3	2	Velma	Brown	97214			
4	3	Fred	Blond	97701			
5	4	Daphne	Red	97701			
6							

People

SELECT * FROM ZipCodes

	A	B	C	D	E	F	G
1	Zip	City	State				
2	97214	Portland	Oregon				
3	97701	Bend	Oregon				
4							

ZipCodes

What does a non-relational database join look like? The data is combined in code, or manually, externally from the database. Here we are going in reverse of the normalization process. We get each table of data back into its own sheet. The blank rows and columns are to remind you that we are back in spreadsheet land.

Joining Data in a Spreadsheet

=VLOOKUP(D2, ZipCodes!$A$1:$C$3,2)

E2 =VLOOKUP(D2, ZipCodes!$A$1:$C$3,2)

	A	B	C	D	E	F	G	H
1	IdPeople	Name	Hair	Zip	City	State		
2	1	Shaggy	Dusty Blor	97214	Portland	Oregon		
3	2	Velma	Brown	97214	Portland	Oregon		
4	3	Fred	Blond	97701	Bend	Oregon		
5	4	Daphne	Red	97701	Bend	Oregon		
6								
7								

People | ZipCodes | People-VLOOKUP

Ready

Now that we have the data across two sheets of a spreadsheet, we turn to the spreadsheet functions VLOOKUP or INDEX to get the data onto one sheet. A few comments about this. First, if you are using VLOOKUP or INDEX in a spreadsheet like this, it is a strong indicator that you should be storing your data in a relational database. Joining data like this in a spreadsheet is time consuming and error prone. Joining data like this will become untenable if you are, for example, trying to order parametric data with MES lot history for a large dataset, or have a dozen tables that need to be joined.

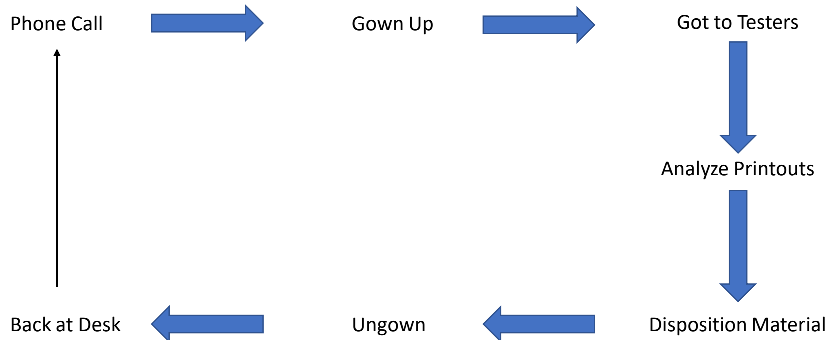
Oh, and If your take-away from this talk is to learn more about the VLOOKUP function, I'll gently weep and capture my tears in a glass filled with whiskey. Don't do it!

Relational databases are used for reporting data stores because of their ability to easily join data sets without new code being written for every join.

(Pause, silent read for crowd to read.) Using VLOOKUP in the spreadsheet example to join data is very specific to the data extracted. If there was a new set of data I wanted to join, I would have to create a new lookup function. I am not paid to be a vlookup expert. In the reporting world, relational databases allow easily generation of ad-hoc reports that combine data from many sources, including in ways that were not first considered when the tables were created.

Denormalization…

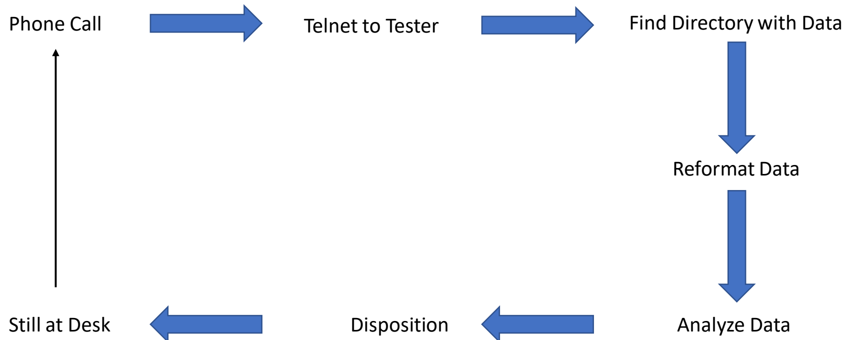
I skipped over denormalization. I'll come back to that in a bit.



When I introduced myself, I mentioned I started as a test and yield engineer. When there was a parametric problem with a lot, I'd get a call, gown up, go to the testers in the clean-room, make a decision, leave the factory, wait for the next call. Obviously there was more to my job than just that, but some days it sure felt like all I ever did.

Laziness is the Mother of Invention

Perhaps a better word would be efficiency. Or maybe I could call this my first ever "Lean" optimization. However one wants to phrase it, I eventually wrote a program that would format the results which were stored on the tester's file system to something human friendly. I would then telnet to the test system, run the program and view the results.

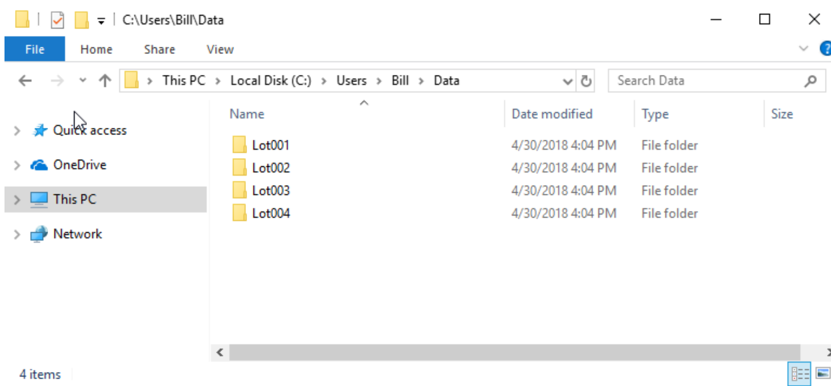


At least with my new process, my coffee would not get cold at my desk. But it still was data analysis in isolation since I was looking at data for one lot by its self, not in the context of similar products. And I was still manually searching for the files. So of course I took the next logical step and wrote a program to search the directories for me! I would enter a lot ID and the program would rsh into each tester and find the data, and present me with the results. Truly amazing times.

Put a Web Server On It

I had been playing around with this new thing called a “web server.” Keep in mind I was doing this back in late 1996 and the Apache web server had been released a year prior in 1995. The next step in my data analysis journey was to put a web front end on it. I could type in a lot number to a web page, and get back some data. This was convenient but had its own problems. The tester might be offline. The tester disks would fill up – which is especially bad at 3am when a hot lot is being tested.

Manual Data Management



Data management by file manager it was. I, or my co-worker, would have to archive the data to a file server. The data was sorted by lot so finding e.g. the last lot of a particular mask meant searching through every directory. And as the data archive grew, so did the length of time to find data. Since lots would be tested out of order, reporting on the number of wafers tested the previous day was nearly impossible.

By the way, having data stored and accessible by one key, such as the lot-id in my example, is a reasonable if imperfect metaphor for how column databases work.

Surprise! Use a Row-Based Table

Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018

Loading meta data file information, such as the lot number, the mask, the route, the date of the test into a database table allows for quick arbitrary searches. This process is best handled by a program, which can also manage archiving the original data files from the tester. But automated data file management is a whole other talk of its own. As I said, using a database to store metadata about the wafer allows for easy filtering.

Some Recently Tested Lots of a Mask

Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018

SELECT * FROM Wafers WHERE Mask = ‘Msk1’ ORDER BY TestDate DESC

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018

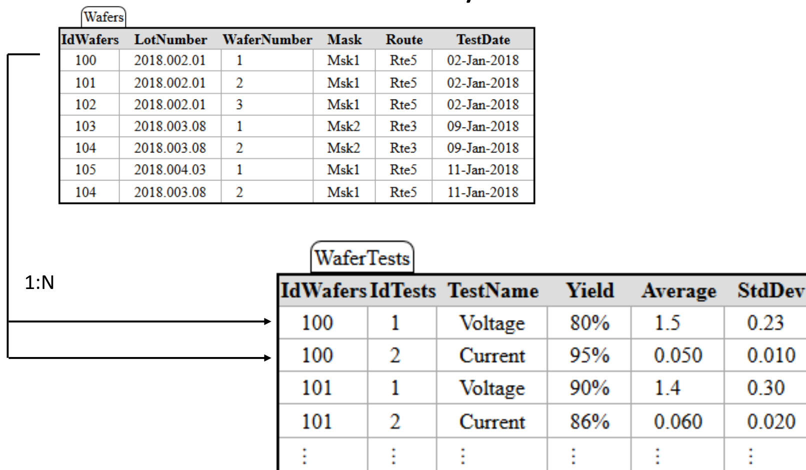
ORDER BY TestDate DESC

For example, here we query the main lot table for some recently tested lots for a single mask. I’ve added an ORDER BY clause and it behaves exactly as you would expect. It orders the results by the column or columns if needed. You can change the direction by using the ASC or DESC keywords, here I’ve used DESC for descending.

A row-based table allows for filtering on any field.

Another defining characteristic of a row-based table, i.e. a relational database, is that a filter can be applied to any column. One can have multiple filters that are applied to different fields from different tables.

Summary Test Data



Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018

WaferTests

IdWafers	IdTests	TestName	Yield	Average	StdDev
100	1	Voltage	80%	1.5	0.23
100	2	Current	95%	0.050	0.010
101	1	Voltage	90%	1.4	0.30
101	2	Current	86%	0.060	0.020
⋮	⋮	⋮	⋮	⋮	⋮

Using a relational database make for easy work to store summary statistics. This shows that there can be a one-to-many relationship. In this example, one wafer has two tests. This is referred to as 1:N in database parlance. This is a form of denormalization. Before we loop back around to that topic, there are two points:

1) This is NOT how I would design a test summary table. I'm keeping the design in this example very simple so the relation between the two tables is clear. 2) Always, always store measurements in MKS. You do not want another $125M Mars Climate Probe incident because one person decided to store values in milliamps but someone else expected the results in amps.

(The Mars Climate Probe was a $125M crash into Mars because spacecraft engineers failed to convert from English to metric measurements.)

Denormalization

People

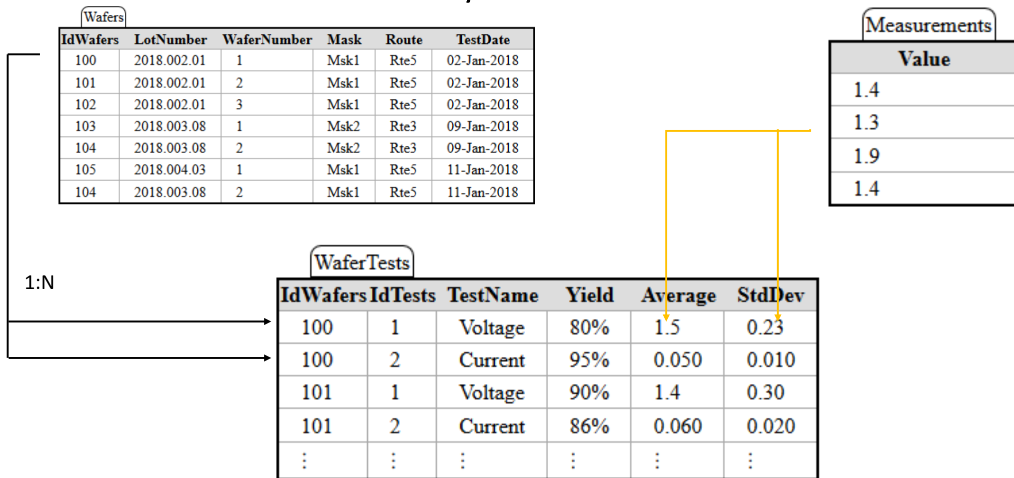
IdPeople	Name	Hair Color	Zip	City	Total Meddlers in City
1	Shaggy	Dusty Blond	97214	Portland	3
2	Velma	Brown	97214	Portland	3
3	Fred	Blond	97701	Bend	2
4	Daphne	Red	97701	Bend	2
5	Scooby	Brown	97218	Portland	3

ZipCodes

Zip	City	State
97214	Portland	Oregon
97218	Portland	Oregon
97701	Bend	Oregon

There are times when it makes sense to put normalized data back into the parent dataset, or not take it out in the first place. Putting data back is called denormalization and is used to simplify reporting or perhaps speed up a query. In this example, I've added Scooby who for some reason lives in a different Portland zip code than Shaggy. I've also added the city and number of folks in the meddling kids gang back in the main table. Now it is easy to report that Scooby is one-third of the Portland meddlers association. Denormalization has a much bigger impact for very large data sets. It is also a big part of so-called data warehouses.

Summary Test Data



Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018

Measurements

Value
1.4
1.3
1.9
1.4

1:N

WaferTests

IdWafers	IdTests	TestName	Yield	Average	StdDev
100	1	Voltage	80%	1.5	0.23
100	2	Current	95%	0.050	0.010
101	1	Voltage	90%	1.4	0.30
101	2	Current	86%	0.060	0.020
⋮	⋮	⋮	⋮	⋮	⋮

Technically the summary statistics are a form of denormalization because we could have the dataset calculate the average and standard deviation on site-level data. But if there are millions of parts, that can be slow, and it can greatly complicate the queries. The first denormalization is simply recording summary statistics for each test. There is another denormalization that is just as useful.

Wafers w/Denormalized Test Stats

Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate	MinYldTest	MinYldTestYld
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018	Voltage	80%
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018	Current	86%
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018	⋮	⋮
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018	⋮	⋮
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018	⋮	⋮
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018	⋮	⋮
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018	⋮	⋮

We can add the lowest yielding test and its yield into the main lot-information table. This makes it quick work to report and spot problem wafers and tests.

SQL: Aggregate Functions /Group By

Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate	MinYldTest	MinYldTestYld
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018	Voltage	80%
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018	Current	86%
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018	⋮	⋮
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018	⋮	⋮
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018	⋮	⋮
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018	⋮	⋮
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018	⋮	⋮

SELECT Mask, COUNT(*) AS Count FROM Wafers GROUP BY Mask

Mask	Count
Msk1	5
Msk2	2

I am not going to get deep into aggregate functions, regardless it is good if you are exposed to an example. Here I am using the COUNT function to sum up the number of records found per mask. You can see that in the table above, there are five records with Msk1 and two records with Msk2 – matching the results from the query. There are many other functions such as to get the average, the sum, minimum or maximum value. Aggregate functions and GROUP BY go hand-in-hand.

T-SQL (Transact SQL)

```
SELECT Wafers.*,
  CASE
    WHEN MinYld < 0.85 THEN ‘LOW YIELD’
    ELSE ‘-’
  END  YieldMessage
FROM Wafers
```

Wafers

IdWafers	LotNumber	WaferNumber	Mask	Route	TestDate	MinYldTest	MinYldTestYld	YieldMessage
100	2018.002.01	1	Msk1	Rte5	02-Jan-2018	Voltage	80%	LOW YIELD
101	2018.002.01	2	Msk1	Rte5	02-Jan-2018	Current	86%	-
102	2018.002.01	3	Msk1	Rte5	02-Jan-2018	⋮	⋮	⋮
103	2018.003.08	1	Msk2	Rte3	09-Jan-2018	⋮	⋮	⋮
104	2018.003.08	2	Msk2	Rte3	09-Jan-2018	⋮	⋮	⋮
105	2018.004.03	1	Msk1	Rte5	11-Jan-2018	⋮	⋮	⋮
104	2018.003.08	2	Msk1	Rte5	11-Jan-2018	⋮	⋮	⋮

T-SQL, also known as transact SQL allows for complex data manipulation. I have a basic example here that will either return LOW YIELD if the yield is less than 85 percent, or the dash if not. Like aggregate functions, I am not going to get deep into transact SQL either. Both of those could be presentations in their own right, and both require a strong SQL foundation. That is, more than thirty minutes of looking at power-point slides demonstrating basic SQL. None the less, being aware of both aggregate functions and transact SQL is a good thing.

The Big Table Join

FtWafers

FtLotNumber	Build Date	PhysicalWaferNumber
FT180002A	13-Nov-2017	6123.4-021
FT180002B	13-Nov-2017	7123.1-010

MesLot2Wafers

LotNumber	Slot	PhysicalWaferNumber
2018.002.01	1	6123.4-021
2018.002.01	2	6123.4-022
2018.003.05	6	7123.1-010

MesLotHist

LotNumber	Date	Operation	Tool
2018.002.01	13-Jan-2018	Photo	Stepper05
2018.003.05	14-Jan-2018	Etch	Etch02
2018.002.01	14-Jan-2018	Deposition	Evap06
2018.002.01	15-Jan-2018	Etch	Etch01

Wafers

IdWafers	LotNumber	WaferNumber	PWN	Mask	Route	TestDate
100	2018.002.01	1	9021.3-015	Msk1	Rte5	30-Jan-2018
101	2018.002.01	1	6123.4-021	Msk1	Rte5	02-Feb-2018
102	2018.003.05	6	7123.1-010	Msk1	Rte5	03-Mar-2018

WaferTests

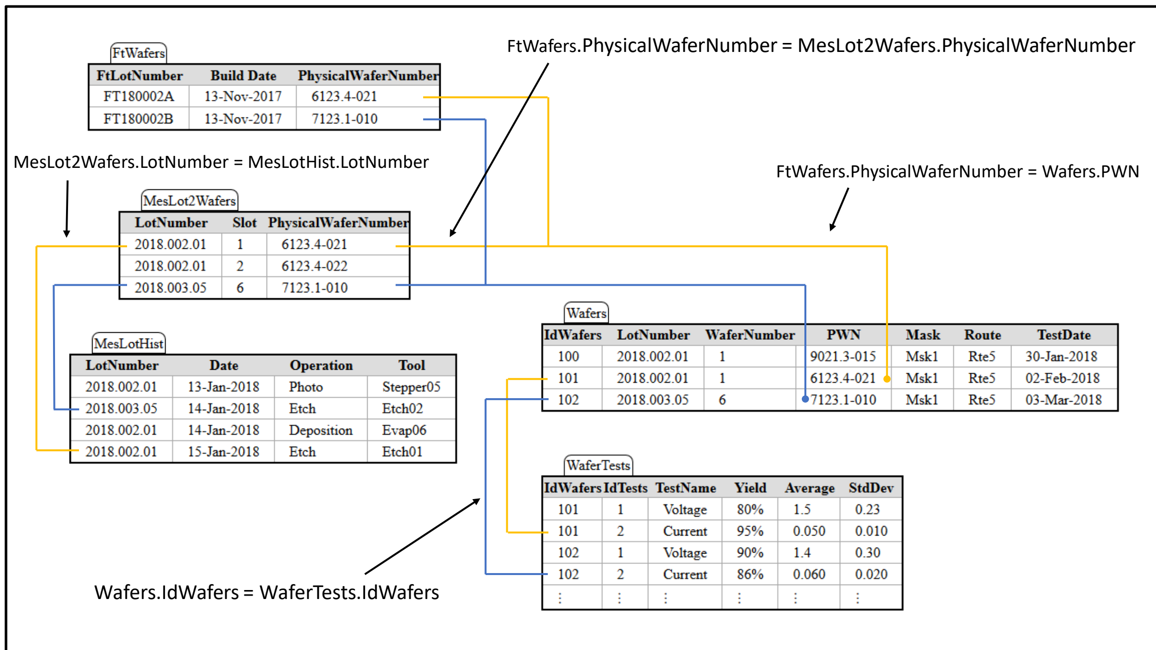
IdWafers	IdTests	TestName	Yield	Average	StdDev
101	1	Voltage	80%	1.5	0.23
101	2	Current	95%	0.050	0.010
102	1	Voltage	90%	1.4	0.30
102	2	Current	86%	0.060	0.020
⋮	⋮	⋮	⋮	⋮	⋮

Here is a call-back to Eric's presentation. Say you have a low-yielding package-part lot. With a table that contains the wafers used to build lots, a table that contains the lot for a wafer (obviously MES) and a table that has the tool used for a lot at an operation, you can map, and color-code graph, your package part data to the tool it saw in the factory.

The Big Table Join

```
SELECT FtLotNumber, Tool, Average
FROM FtWafers, MesLot2Wafers, MesLotHist, Wafers, WaferTests
WHERE
        FtLotNumber.PhysicalWaferNumber = MesLot2Wafers.PhysicalWaferNumber
  AND MesLot2Wafer.LotNumber = MesLotHist.LotNumber
  AND FtLotNumber.PhysicalWaferNumber = Wafers.PWN
  AND Wafer.IdWafers = WaferTests.IdWafers
  AND WaferTests.TestName = ‘Current’
  AND MesLotHist.Operation = ‘Etch’
```

Here is the equivalent in SQL.



Here is a busy slide with the table join criteria.

SQL Results

FtLotNumber	Tool	Current
FT180002A	Etch01	0.050
FT180002B	Etch02	0.060

RDBMS

- (Automate data load and archive).
- SQL is a Powerful Language.
- Allows for ad-hoc searches and reports.
- Allows for standardized reports.
  - Everyone runs the SQL against the same data.
- Focus on data analysis.
  - Quickly get data into JMP, Spotfire, etc. for deep dives.
- Enables Meeting the “3am Rule”.

The “3am Rule”

Systems and trouble-shooting documentation should be so simple that they are usable if one is awakened at 3am.

This is a personal rule. Well, more of a guideline born from many years of supporting factories 24/7. When a hot lot is stopped at 3am, one needs quick and simple systems to make a quick first-pass analysis. What little brain-power one has in the fog of having just woke up should be spent analyzing data – not spent finding and prepping data for analysis.

Put a Web Server on that RDBMS

Tools like JMP and Spotfire are great for deep dives and other engineering level analysis. However they are overly complex as a tool to enable operators to make a first pass look at data. Yes there are saved reports. They are not the best option for cycle-time critical manufacturing. Asking an operator to go through a complex process to then decide if there is a pass or fail situation is more complicated than it needs to be.

Live Demonstration

Let’s put all of this together. What I am about to show you is fake data. I wrote a program to generate some data, all with a nice gaussian distribution.

Relational Databases

Data Analytics Beyond Spreadsheets and Single-Table Databases

Bill Adams
Independent Contractor, Lava-Data

http://lava-data.com/cs-mantech.html