# Is a Dataframe Just a Table?

**Yifan Wu**
UC Berkeley, Berkeley, CA, USA
yifanwu@berkeley.edu

## Abstract

Querying data is core to databases and data science. However, the two communities have seemingly different concepts and use cases. As a result, both designers and users of the query languages disagree on whether the core abstractions – dataframes (data science) and tables (databases) – and the operations are the same. To investigate the difference from a PL-HCI perspective, we identify the basic affordances provided by tables and dataframes and how programming experiences over tables and dataframes differ. We show that the data structures nudge programmers to query and store their data in different ways. We hope the case study could clarify confusions, dispel misinformation, increase cross-pollination between the two communities, and identify open PL-HCI questions.

**2012 ACM Subject Classification** Information systems → Relational database query languages; Software and its engineering → Software usability; Software and its engineering → API languages

**Keywords and phrases** Usability of Programming Languages

**Digital Object Identifier** 10.4230/OASIcs.PLATEAU.2019.6

**Funding** *Yifan Wu*: NSF 1564351

**Acknowledgements** Thanks to my advisor Joe Hellerstein for the inspirations and to Devin Petersohn, Michael Whittaker, Remco Chang, Wenting Zheng, and Eric Liang for their valuable and kind feedback.

## 1 Introduction

Querying data is ubiquitous – application programmers query data to show relevant information, analysts query data to answer business questions, and scientists query data to find patterns for formulating and testing hypothesis. The uses cases are addressed by different communities.

Application programmers and business analysts are traditionally served by databases. During the 1970s, Codd's seminal paper defined a set of relational algebra over tables. A few years later, IBM developed SQL, a declarative language that can express the algebra. Since then, SQL has become the standard for database management systems. The data scientists are served by more general purpose programming environments like R and Python. Instead of using tables, they use *dataframe*, as seen in *pandas* (Python) [7], R [11], and Spark [13].

Database researchers find dataframes odd. A well-known database researcher, Joe Hellerstein, commented on Twitter in 2016, *Stop. A "data frame" is just a table. Thank you* [19]. In the tweet thread, Leo Meyerovich, a PL researcher, suggested that the systems must have been built for a reason. He found that dataframes helps with a *clean curation of basic DB, HPC, PL, etc. ideas.* Two years later, the difference in opinion continues – when prompted with whether the tweet has "aged well", Joe replied, *A "data frame" is a messy conflation of relations and matrices that wouldn't have evolved in a well-typed language, and which complicates the relevant algebraic operations involved.* [20].



While the database side considers dataframe APIs messy, the dataframe side seems to consider the database APIs inconvenient – from the creator of pandas, Wes McKinney, "[pandas] is what people use for data ingest, data prep, and feature engineering for machine learning models. The existence of other database systems that perform equivalent tasks isn't useful if they are not accessible to Python programmers with a convenient API." [3]. We see from the comment that the question of whether a dataframe is the same as a table is as much about the data structures themselves as it is about the API design, which, as well will show, is often influenced by the data structures both in terms of technical limitations as well as mental models – to understand tables and dataframes, we also need to understand accompanying languages like SQL and pandas.

Not only do system designers have different opinions, users of the query systems also have different views. On a Stack Exchange post, the first search result in Google for "pandas vs. sql" as of August 2019, the second most upvoted post claims that the comparison is "apples to oranges" [10].

Understanding the disagreement is important for language and library designers. Not understanding what functionalities are different or what features are desirable prevents the communities from learning from past lessons and leveraging existing techniques. It may also be confusing to the users to be presented with inconsistent messages and ideas. In this article, we evaluate the differences of the data structures of tables and dataframes, their mental affordances, the operations available, the mediums by which the operations are expressed and the effects on programming experience. We will take a bottom up approach, investigating existing languages – primarily SQL and Python pandas – to draw out relevant concepts and questions. Along the way, we will identify open questions for future work and speculations about the approaches.

## 2 Data Structures and Operations

The tabular format has existed for a long time as a way of organizing information, dating at least to the *Almagest* almost two thousand years ago [27]. It is no wonder that both the database and data science community have found the format useful. Despite sharing a similar tabular look, tables and dataframes are defined as different data structures and have different operations available. In databases, a table is a *set* of records (rows)[1]. A table is also called a *relation*. The *relational algebra*, proposed by Codd, defines the transformations available to these relations [17]. The design of relational algebra protects users from needing to know how the data is organized in the machine, and makes it possible for users to specify high-level queries, and leads to an inexhaustible number of optimization techniques. In data science, there is more than one definition of a dataframe, listed in Table 1. In the rest of this section, we explore what the definition means for the users of these languages[2].

[1] In practice, tables are often *multisets*, which means that there can be duplicate values. We do not go into a detailed discussion here since the key difference being investigated is that of order.

[2] We will use "language" uniformly to discuss both languages and libraries for simplicity.

**Table 1** Definitions of dataframe.

| lib/lang | definition |
|---|---|
| pandas | two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects [7]. |
| R | tightly coupled collections of variables which share many of the properties of matrices and of lists [11] |
| Spark | equivalent to a table in a relational database [13] |

## 2.1 Set v. Lists

Relational algebra is all about (multi)sets. As a result, the programmer cannot rely on the relative position of the row or column in the table. Dataframes are all about lists[3], where programmers can use the relative positions. As a direct consequence, many operations that are commutative with tables are not for dataframes. For example, concatenating dataframes in pandas takes in a list, whose order, if changed, returns a different result. Take `df12 = pd.concat([df1, df2])` and `df21 = pd.concat([df2, df1])`, `df12.iloc[0]` may not equal `df21.iloc[0]`. In SQL, union-ed tables are the same regardless or the order – `(SELECT * FROM df1) UNION (SELECT * FROM df2)` and `(SELECT * FROM df2) UNION (SELECT * FROM df1)` are the same.

The affordance of order has a big impact on how programmers think. First, being able to rely on *order* makes basic operations like finding a row with values of a certain rank much easier. For instance, if a list is already sorted by, say, `income`, finding the row with the maximum `income` (i.e. rank of 1) is as simple as selecting the first item `df.iloc[0]`. SQL is more verbose: `SELECT * FROM df WHERE income = (SELECT MAX(income) FROM df)`, or less idiomatically, `SELECT * FROM df ORDER BY income DESC LIMIT 1`. The difference may seem small, but consider instead accessing the row of rank, say, 100. The following query is in order: `SELECT * FROM df AS df1 WHERE (SELECT COUNT(*) FROM df AS df2 WHERE df1.income > df2.income) = 99`. There are more steps here than the list-based approach. Now the reader should try writing the query for the median. Our observation is further backed up by the fact that among the pandas APIs called in the top voted Kaggle kernels (as an approximation of pandas API use), `iloc` is more commonly used than `join` and `drop` [8].

Second, looping over a list is sometimes easier to use than mapping over a set. For instance, consider getting the smallest *missing* value, e.g., `{1,2,4,5,10}` misses 3. With loop-based thinking, a programmer can just pause after the first "gap", but in set-oriented thinking, one solution is to create a number of continuous integers, and then select the minimum value that's in the integers table, but not in the set at hand, e.g., `SELECT MIN(n) FROM numbers WHERE value NOT IN (SELECT value FROM t)`. Another example is computing the difference between conceptually consecutive items, like what month-to-month sales change. A SQL query would use a `JOIN` operator, `SELECT t1.month, t1.sales-t2.sales FROM t AS t1 JOIN t as t2 on t2.month = t1.month - 1`.

One may be tempted to draw the conclusion that lists are superior to sets from a programming experience perspective. However, it is unclear if that is due to a bias in education. Besides, the argument for programming without relying on order is increasingly

[3] And arrays, which is also ordered, the key feature discussed.

important for programming with large datasets on the cloud [12], where the cost of maintaining order is incredibly high, both in terms of performance, and engineering and compute resources. Sets are much superior to lists when it comes to performance since operations over sets can be easily parallelized. Since order is so expensive, it makes sense to have programmers ask explicitly for order only when it is needed, as opposed to being assumed to be always present. In addition, operators over sets tend to encourage more declarative thinking, since there is no order to base the procedural thinking on. The key question for the future from this section is understanding how difficult will it be to teach developers to program without a constant assumption about order.

## 2.2 Matrix vs. Table

In a table, rows and columns are fundamentally different abstractions – one could reason about rows but not columns, which would be second-order logic. Matrices do not differentiate between rows and columns and are traditionally used in linear algebra. Many, like dot-products, are awkward in SQL – the values would be stored in the schema `(rowID, colID, value)`. By contrast, matrices don't provide the logical operations natural to relations, like selection or join. However, dataframes provide a mix of matrix operations and relational operations, because they provide both logical operations and linear algebra like operations.

### 2.2.1 Mixing Matrix and Tables, the Good Part

This mixture can be handy. Consider Table 2. If a user wishes to compute the total sales for each year, it makes sense to apply an aggregation across the columns – `sales_df.apply(lambda row: sum(row) - sum.year, axis=1)`, rather than writing out all the column names manually – `SELECT fruits + nuts + dairy + meat ... FROM sales`. The limited operations allowed over tables could be casts as *Premature Commitment*, per Green's Cognitive Dimensions of Notations [18]. Perhaps the *messy conflation of relations and matrices* [20] is what makes scripting and "exploratory programming" easy [21].

**Table 2** An example table where programmatically iterating through the columns is desirable.

| year | fruits | nuts | dairy | meat | ... |
|---|---|---|---|---|---|
| 2017 | 100 | 40 | 300 | 400 | ... |
| 2016 | 200 | 150 | 200 | 400 | ... |
| ... | ... | ... | ... | ... | ... |

However, there is a better way to achieve the above-mentioned functionality in databases by changing the schema. Instead of having the year *values* as columns, the table can have the values in the rows, with a new schema: `year`, `category`, and `amount` (Table 3). Then the query is `SELECT year, category, SUM(amount) FROM sales GROUP BY year`.

**Table 3** An example table that contains the same data as the previous table, but different schema.

| year | category | amount |
|---|---|---|
| 2017 | fruits | 100 |
| 2017 | nuts | 40 |
| ... | ... | ... |

This style of schema design is not accidentally convenient. *tidyverse*, a popular ecosystem of libraries in R, encourages programmers to organize experiment variables as columns, and rows as observations (which makes the data relational). The data layout design increases the ease of manipulation and usage of libraries like `dylplr` and `ggplot2`, as well as better performance [29].

### 2.2.2 Mixing, the Bad Parts

A quirky outcome of the matrix thinking gone too far is that dataframes allow **duplicate column names**. For example,`pd.DataFrame([[1,2,3],[2,3,4]], columns=['a','a'])` evaluates without error and results in a dataframe with two column `a`'s.

Another issue is the additional notion of **indices** in dataframes. Since a pandas dataframe is a "dict-like container", it has an explicit notion of the index. Programmers can change and use indices – they can even use a "hierarchical" index called `MultiIndex` [6]. For instance, a dataframe of sales information (`store_id`, `date`, `item` etc.) can be indexed with the store type (e.g., Trader Joes) and state (e.g., California). Each state can have multiple store types. As a result, queries like finding all sales at Trader Joes in California: `df.loc[("California", "TraderJoes")]`.

Tables do not have an explicit notion of indices, but rather the concept of keys, which are made up of sets of columns. Following the example, the index can be replaced by two additional columns, `store` and `state`, then the same query can be written with `WHERE state ="California" AND store="TraderJoes"`. Tables do not need indices to achieve the same functionality.

In fact, indices make the downstream API more complex. Let's take a look at `concat` again. While `concat` was compared to `UNION`, the semantics of `concat` are much more complex. In SQL, `UNION` can only be applied to rows of the same width (and type), but in pandas, a programmer need to specify whether the concatenation is along rows or columns and how to match the indices or columns (full `concat` API: `pd.concat(objs,axis,join,ignore_index, keys, levels, names, copy, verify_integrity)`). An index is a redundant construct.

### 2.2.3 Joins

The join operator is the crown jewel of the relational operators. It accepts a join condition (selection) and a pair of tables as arguments and returns a table [25]. Joins enable programmers to derive all kinds of information from relationships between data.

Joins in dataframes (called `merge`), on the other hand, look very different as a result of the mixing of metaphors. Here's what it looks like: `pd.merge(left_df, right_df, how, on, left_on, right_on, left_index, right_index, sort, suffixes, copy, indicator, validate)`. The merge operator seem rather complex, with concepts like `left_index` and `suffixes` that are not needed for joins on relational tables. Furthermore, despite the complexity, the join condition is limited to column equalities (a concept called *equijoin* in databases). Non-equijoins are very common. For example, to find the weather of events by joining the events table and the weather table. In SQL this can be written as `SELECT * FROM events JOIN weather_log w WHERE events.date > w.start AND events.date < w.end`, and this cannot be expressed using the *merge* function alone in dataframe APIs in pandas or R. The lack of non-equijoin support is not an oversight. One key pandas maintainers, Jeff Reback, explained in a discussion on GitHub that adding non-equijoins is not Pythonic [1]. This is puzzling because, for joins between tables, both equijoins and non-equijoins are predicates over columns and are treated exactly the same algebraically [25].

In this sense, the `merge` is not actually a database join, as was claimed by both R and pandas' documentations [4, 5]. What further complicates the picture is that the `concat` operation (discussed briefly earlier) also have mixed in concepts of join. In fact, in dataframes, `concat` is much closer to `merge` than `UNION` is to `JOIN`; `concat` even takes a inner or outer join specification [4]! This odd overlap of concepts could cause additional confusion. For instance, the `concat` function requires that the names or indices of the columns match, but with `merge`, programmers could specify what columns to merge on, and the names do not always have to be the same. This makes rows and columns not actually symmetric, as the name "matrix" may initially suggest, further complicating the mental model a programmer must hold of dataframes.

Perhaps a better design is to have tables and matrices as two separate data structures each with their own operations.

## 3 Programming Workflows

Writing SQL and dataframe queries are very different experiences. The first is a declarative language, and the latter is invoking library functions in a host language. SQL was initially intended to be used by non-programmers like accountants and architects through a terminal interface [14, 15]. On the other hand, dataframe libraries are serving data scientists who can program. While SQL encourages programmers to think only about tables and reason about high-level functionalities, which are easier to optimize. Dataframes are used more procedurally, where programmers apply a sequence of operations on the dataset [13]. In this section, we analyze the different implications of the procedural versus declarative approach.

### 3.1 Accessing Functions

SQL was designed to be used just by itself, typed into a terminal. However, increasingly programmers need to access the query results in a general-purpose programming environment, which they can achieve using a wrapper, often constructing the query as raw strings with no IDE support – no syntax highlighting, no linting, no refactoring, and no type checking. It also makes accessing functions more difficult. Functions are useful for defining custom predicates and aggregations. In SQL, programmers can access custom user-defined functions (UDFs) by registering the function to the database via a wrapper, e.g., `sqlite3.create_function (<custom_agg>)`.

### 3.2 Code Reuse and Composition

Functions are also useful for sharing code that expresses the same logic, which is good for code reuse. Codesharing is easy with dataframes since they are manipulated with library calls in a general-purpose host language. In a similar vein, dataframe queries can also be composed using functions directly. For instance, the logic to pick different tables or columns can be embedded in normal functions. Composition reduces the complexity of analytic tasks with more succinct and readable code.

On the other hand, in SQL, the primary way to reuse code is via `VIEW`s, which are similar to tables, except that they are not computed and persisted to storage (*materialized*). Since SQL can only express first-order logic, the ability to reuse code using `VIEW`s is limited. For example, programmatically changing or iterating over different tables or columns is second-order logic and cannot be expressed in SQL. Programmers would have to generate code strings, which is brittle or manipulate abstract syntax trees, which is often an over-kill.

Consider a scenario where the programmer first creates an aggregation, then wishes to add a filter before the aggregation. Take a table of flight delay data (`delay`, `origin`, `time`, etc.). To aggregate based on `origin` in SQL, the programmer can write, `CREATE VIEW originAgg SELECT origin, COUNT(*) FROM flights GROUP BY origin`. After seeing the result, they plan to see the same aggregation but filter by flights thats have delays greater than an hour. However, there is no choice but to copy the query and add a where clause – `SELECT origin, COUNT(*) FROM flights WHERE delay > 60 GROUP BY origin`. There is no way to reuse the view `originAgg` because the `delay` column is no longer accessible. Although the copy-paste then edit seems fine, the problem becomes more pronounced with longer queries. With dataframes, the programmer can extract the operations `originAgg=lambda(df):df.groupby(['origin']).count()`, and pass the filtered dataframe to the `originAgg` function.

## 3.3 Debugging

The function chaining style of dataframes forces the programmer to flatten out the operations, and the results from a function can be inspected. However, this is not the case for SQL. For instance, a programmer may write a query using both `HAVING` (with `GROUP BY`) and `WHERE` clauses. When the result is unexpected, and the programmer wishes to inspect the intermediate result, they would have to write new queries that capture the logic individually. With dataframes, the programmer may already have the intermediate variables, and if not, they can just break the chain and inspect the intermediate values without writing new code.

## 3.4 Performance

There are many different sequences of operators – a *physical plan* in databases – that evaluate to the same values. Some of the sequences are slower and take more resources than others. With the exception of Spark, dataframe programmers currently have to figure out what is better themselves. To demonstrate, we borrow an example given by the pandas creator Wes McKinney: to sum the values of column `c2` from rows whose `c1` column is negative [23]. With pandas, there are at least two ways:

1. Find the rows, then sum – `df[df.c1 < 0].c2.sum()`. When these functions are invoked, pandas creates a temporary dataframe `df[df.c1 < 0]`, then sums `c2` column of that temporary object. The temporary dataframe can be wasteful if `df` contains a lot of columns.
2. Trim the dataframe down to just the `c2` column, using the index from applying the predicate on `c1`, then sum – `df.c2 [df.c1 < 0].sum()`. Since the df is first projected, it uses less memory.

In SQL, the solution is `SELECT SUM(c2) FROM df WHERE c1 < 0`, and the database will decide what sequence of operators to execute by using a *query optimizer*, which finds a better execution plan regardless of the specification.

This issue is not just limited to the order of operators; the choice of functions can also lead to performance issues. For example, the `apply` function in pandas prevents vectorized processing, and there are often alternatives, such as using a UDF to process the whole column instead of a row at a time, or sometimes to use the `iterrows` function to loop, leading to extensive discussions among users of dataframes [9].

Is the situation unredeemable? Not quite, when evaluated lazily, dataframe operators can still benefit from a subset of query optimization techniques available to SQL [13]. Furthermore, performance is not the only issue programmers care about. Sometimes it is confusing when a

query is slow, and we know that query optimizers don't always work very well [22]. It might be a better programming experience to be able to build a mental model and have a more predictable performance that the user can improve on. Perhaps a good programming system is not one that executes faster on average but one that respects the programmer's agency.

## 3.5 Code Comprehension and API Recall

Some programmers dislike the syntax of SQL, where the `ALL CAPS` syntax can seem ugly. This is partially due to historical reasons since syntax highlighting wasn't available when SQL was first created (although even today SQL is often run in strings and terminals, keeping the need for the caps).

Caps aside, we would argue, however, that SQL is actually more *role-expressive* than dataframe APIs. Role-expressiveness captures how easy it is for a programmer to parse code into mental structures [18]. Because SQL blocks are fairly constrained, reading the operations is straightforward, at least for simple queries, but this is not the case for dataframes. Consider the many ways of expressing `SELECT * FROM df WHERE a > 3` in pandas, a non-exhaustive list below, with different evaluation below. The syntax differences are partially caused by the use of syntax shortcuts, the use of defaults in function calls, and pandas exposing lower level executions.

- `df[df.a>3]`
- `df[df["a"]>3]`
- `df.loc[df.a>3]`
- `df.loc[df["a"]>3]`
- `df[lambda foo:foo.a > 3]`
- `df.loc[lambda df: df["a"] > 3]`
- `df.loc[df.apply(lambda r: r["a"] > 3, axis=1)]`
- `pd.DataFrame([r for r in df.itertuples() if r.a > 0])`

Having many different ways to express the same logic makes it hard for developers to understand programs of heterogeneous styles. Besides having varying ways to express the same simple logic, the sheer number of APIs ($> 200$) that are not only overloaded but also have default parameters that may change version to version, making it hard to remember the APIs. Developers may end up looking up documentation or search StackOverflow and break the flow. The question for the future is whether a library should optimize for short solutions to a large number of problems or optimize for a shorter list of APIs.

## 4 Conclusion and Future Work

So is a dataframe just a table? Our answer is that in the wild, dataframes are used differently from tables and carry many additional useful functionalities at the cost of clarity and performance. More PL-HCI work is needed to create language affordances that could help combine the best of both worlds for programmers and language creators. In particular, we had not time in this paper to investigate the design choices made by other languages that integrate relational operators into a host language, such as LINQ [24], FlumeJava [16], Eve [2], and Object Relation Managers. The creators of these languages have also made various discoveries and claims about the desired properties of the language that might yield insight into what the "Pareto-efficient" trade-off curve is between factors.

In the process of investigation, we have opened up new questions. Some are human factor questions, like whether it is inherent that programmers prefer lists over sets. Others are philosophical questions, like whether language designers should meet users where they are. We hope these questions can help inspire more discussions and analysis of a more HCI-oriented study of query languages, to help guide language and library creators with more material to discuss, less confusion, and better principles.

## 4.1 A Cliffhanger


Before we end, we must share another teaser controversy. A well-followed database researcher, Kelly Sommers, tweeted, *Why GraphQL when we could have used SQL?* [28]. Sommers continues to argue that *GraphQL exists because JavaScript developers finally realized HTTP API's were too limiting so they reinvented SQL over JSON because JavaScript developers are obsessed with reinventing everything into JSON API's.* The tweet has received over 2K likes. The creators of GraphQL, experienced engineers at Facebook, responded that SQL did not serve them well and that GraphQL is the result of listening and empathizing with the needs of the product developers [26]. Now GraphQL has almost 15K GitHub stars and a large and active developer community. *What should we make of it?*

## References

1 Conditional join (merge) in pandas. URL: `https://github.com/pandas-dev/pandas/issues/7480`.
2 Eve: Programming designed for humans. URL: `http://witheve.com/`.
3 HN thread about McKinny, Things I Hate About Pandas. URL: `https://news.ycombinator.com/item?id=15335462`.
4 Merge, join, and concatenate. URL: `https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html#set-logic-on-the-other-axes`.
5 Merge two data frames by common columns or row names, or do other versions of database join operations. URL: `https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/merge`.
6 Multiindex / advanced indexing. URL: `https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html`.
7 pandas.dataframe docuemntation. URL: `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`.
8 Usage of pandas api by kaggle usage. URL: `https://github.com/modin-project/study_kaggle_usage/blob/master/results.csv`.
9 When should i ever want to use pandas apply() in my code? URL: `https://stackoverflow.com/questions/54432583/when-should-i-ever-want-to-use-pandas-apply-in-my-code`.
10 Why do people prefer pandas to sql? URL: `https://datascience.stackexchange.com/questions/34357/why-do-people-prefer-pandas-to-sql`.
11 data.frame, r-core documentation, 2018. URL: `https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/data.frame`.
12 Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. Citeseer, 2011.
13 Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.
14 Raymond F Boyce, Donald D Chamberlin, W Frank King III, and Michael M Hammer. Specifying queries as relational expressions: The square data sublanguage. *Communications of the ACM*, 18(11):621–628, 1975.

15 Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

16 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, pages 363–375. ACM, 2010. doi:10.1145/1806596.1806638.

17 Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

18 Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

19 Joe Hellerstein. Stop. a "data frame" is just a table, August 2016. URL: https://twitter.com/joe_hellerstein/status/761364295510691840.

20 Joe Hellerstein. A "data frame" is a messy conflation of relations and matrices, March 2018. URL: https://twitter.com/joe_hellerstein/status/978335500250447878.

21 Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, pages 1265–1276, 2017.

22 Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

23 Wes McKinney. Apache arrow and the "10 things i hate about pandas", 2017. URL: https://wesmckinney.com/blog/apache-arrow-pandas-internals/.

24 Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

25 Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.

26 Nick Shrock. Graphql exists not just because, November 2018. URL: https://twitter.com/schrockn/status/1060314584525955072.

27 Nathan Sidoli. Mathematical tables in ptolemy's almagest. *Historia Mathematica*, 41(1):13–37, 2014.

28 Kelly Sommers. Why graphql when we could have used sql?, November 2018. URL: https://twitter.com/kellabyte/status/1059956838744158213.

29 Hadley Wickham and Garrett Grolemund. *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc., 2016.