ZEN AND THE ART OF DATABASE MAINTENANCE

EDGAR COSTA AND DAVID ROE

Abstract. The last decade has seen a proliferation of online mathematical resources. We discuss some of the technical challenges involved in creating and maintaining a mathematical database. In particular, we report on the transition of the *L-functions and Modular Forms Database* (LMFDB) between two database systems. We also highlight some of the improvements to the LMFDB that we have made as part of this transition.

1. Introduction

Mathematics has a long history of using computations to aid in forming conjectures and searching for counterexamples. In the past few decades, computers have taken on a central role, both in performing many calculations and in hosting the results. As computational and storage capacity has increased, the size of these results has grown to the point where the task of searching and maintaining the data requires specialized knowledge. We discuss some of the challenges involved, as well as the main tools available to address them. We focus on one particular case study: the effort to migrate the database system supporting *L-functions and Modular Forms Database* (LMFDB) [29] from MongoDB [31] to PostgreSQL [38]

We hope that this article will be useful to mathematicians aspiring to create or improve their own databases, to database engineers who are considering switching from MongoDB to PostgreSQL, and to mathematicians participating in large software projects for whom some of the LMFDB's lessons might prove useful.

We begin with a survey of existing mathematical databases to highlight the diversity of approaches used to disseminate mathematical data. While some focus on communicating theorems and propositions, we pay more attention to those which center on examples. Among these, we observe a range of search functionality. Since the LMFDB prioritizes the ability to search by properties of each object, we hope that our experience working with MongoDB and PostgreSQL will help other mathematical resources improve their capabilities in this area.

Next, we consider some of the important choices involved in setting up and maintaining such a database. We divide this discussion into three parts: how to encode and manipulate the data, how the user interacts with the database, and where it is physically stored. In each category, we present several options and a brief analysis of their pros and cons.

The technical core of the article is the comparison of MongoDB and PostgreSQL, along with our experience in switching between them. We were motivated to change systems primarily for performance reasons: many searches that we wanted to support stalled on common inputs. We did not see a way to resolve the issue within MongoDB. Several missing features in MongoDB also pushed us toward shifting:

support for arbitrary precision integers and transactions that allow multiple commands to be rolled back if there is a failure.[1] We summarize the major differences, including the distinction between a relational database and a document-oriented database, and describe some of the features of each that were important in different stages of the LMFDB's history. We also outline our process for both porting the data and updating the codebase to use a different system.

We finish by highlighting some of the benefits provided by performing such a major structural change. For example, the process of rewriting a large amount of code allowed us to refactor and encapsulate many of the LMFDB's core functions. Moreover, these code changes enabled us to dramatically improve performance and add new features, such as verification suites that check for consistency within the database. Going forward, the ability to make connections between different kinds of mathematics is one of the core aims of the LMFDB, and a relational database will help us achieve this goal.

2. Using databases in mathematics

In order to put our work on the LMFDB in context, we give an overview of some other existing mathematical databases.

We will be primarily interested in datasets consisting of information on mathematical objects, such as specific number fields, graphs or curves, rather than on mathematical statements. There are many valuable resources focusing on theorems and propositions, from general encyclopedias such as the Encyclopedia of Mathematics [16] and Mathworld [48] to the preprint server arXiv [5] to numerous domain specific references [1,14,33]. However, the design imperatives for such projects favor a different type of database than the ones we are interested in. Since the content of theorem-oriented sites is likely to be textual, the standard tools for internet searching such as Google tend to work effectively. Moreover, because humans are directly creating the content, the tools for inputting and managing data are different. Wiki [1,6,23,33] and blog [3] software is often used to make it easier for many users to be involved in content creation. Finally, when algorithms are generating data, it is possible to end up with orders of magnitude more content, leading to various challenges of scale not faced by human-created datasets.

One of the main roles that a mathematical database plays is to record the results of interesting computations. Such collections of examples are useful in many areas of mathematics, including group theory [7,15,22,49], Lie theory [2,8], graph theory [10], knot theory [28], integer sequences [44], Euclidean geometry [26], algebraic geometry [3] and number theory [11,29]. In each case, much of the utility of the database is shaped by how the results are displayed to a user, whether as a list of integers with annotations, a picture of a knot or a webpage showing various invariants of the object. Some databases [7,22,30] are embedded within computer algebra systems such as Sagemath [42], Magma [9], or GAP [18], which allows a user to perform further computations with the objects in the database easily. Others are available online in various formats; we will discuss different methods of data distribution in Section 3.

[1]MongoDB 4.0 added support for transactions but was released after we switched to PostgreSQL.

Collecting a large number of examples into one database makes possible a valuable paradigm for experimental mathematics: the ability to search through examples for a few that have particular specific desired properties or are counterexamples to a conjecture. In addition to the LMFDB, a number of other mathematical databases also have search functionality as one of their core features [8, 10, 27, 28, 37, 41, 44]. Searching requires a more substantial infrastructure than is needed for just displaying data; the discussion in Section 4 highlights some of the factors to consider when choosing a search engine.

A final role played by many online mathematical databases is to provide domain-specific exposition surrounding the examples that they host [1, 11, 14, 15, 23, 28, 41]. The LMFDB aims to provide enough definitions and background for mathematically literate users to be able to understand the contents. The main tool for doing so is a webpage element called a *knowl*, which is a bit of context-free knowledge. Within the LMFDB, knowls are embedded inside webpages or other knowls, and expand to provide more information when clicked. Such exposition is an important part of the presentation of a database, but is peripheral to our main topic of the functioning of the underlying system.

Another important aspect of a database project that falls beyond this article's scope is the social and financial efforts required to sustain a collaboration. There are numerous examples of projects that are no longer accessible because the authors have moved on or passed away. Projects have various ways of reaching out to potential developers, ranging from academic workshops and conferences, to the standard open source tools such as GitHub [19], to forms allowing users to easily submit data [10, 41, 44]. In this article, we limit our attention to the technical side of the job: from setting up hosting servers, to the different options for database software, to details on how the LMFDB functions.

2.1. **The LMFDB project.** The Langlands program, first formulated by Robert Langlands in the 1960s, is a set of widespread conjectures aimed at understanding and explaining the interconnections between a dizzying array of subfields of mathematics, including number theory, representation theory, algebraic geometry, and harmonic analysis—and in the 21st century its reach continues to expand. The Langlands program has been called "one of the biggest ideas to come out of mathematics in the last fifty years" and "the Grand Unified Theory of mathematics" [17, p. 3].

To provide compelling visual and computational displays of the Langlands program "in action," a database was created called the *L-functions and Modular Forms Database* (LMFDB), available at the website `http://www.lmfdb.org/`. The LMFDB was first conceived in 2007 [12] and remains the object of a significant amount of ongoing work by over one hundred mathematicians [29, Acknowledgments].

The LMFDB hosts a variety of databases, including some predating the project [13, 24, 25], and connects them through the Langlands program; we recommend online experimentation or [12] for more details.

3. Nuts and bolts

When it comes to hosting a mathematical database, there is a wide range of options available. Until the last several decades, the only choice was to use a print format, for example in a book or an article, and rely on a publisher to distribute its content and to make it available to the rest of scientific community. Since the publisher takes care of most of the technical issues, this method offers the benefit of simplicity to the author. Still, distribution in print provides several disadvantages compared to digital distribution via the internet. For instance, the act of publishing through print is a lengthy process, and the data is less immediately accessible than online distribution, putting barriers in the way of the database user. Furthermore, once the data is published, it is challenging to address mistakes, add new data or features. For last, it can only accommodate small databases, due to the physical requirements to print.

On the other hand, given that the digital distribution of mathematical data is not taken on by publishers, authors have many more choices to make. We break down these choices into the following related categories:

- back end — How to encode and manipulate the database in a machine readable format.
- front end — How the user interacts with the database.
- hosting — Where to physically store the database.

We will overview some of the possible options in each of these categories

3.1. **Back end.** We organize the options for how to encode the database in a machine readable format by how easy it is to search for content.

Human readable files. Storing the data set in a human readable format, like a plain text file or a comma-separated values file, is one of the most straightforward possible solutions. It does not require special software to access the data, very similar to a table in print. However, it can be cumbersome and impractical to search or manipulate the data without additional tools. Note that simple searches are possible using tools like grep or the find feature of a text editor. However, these methods do not offer the full search capability of a database.

Application dependent files. A more advanced solution is to store the dataset in an application dependent format. For example, the Small Groups Library [7] available in the GAP computer algebra system [18] uses a specific method to encode solvable groups. In many cases, such a choice enables direct manipulation of the desired objects. For example, using SageMath pickles makes it easy to load the data into an active SageMath session. However, the dataset will not likely be efficiently searchable, searching for objects with specific properties will usually require to loop over every item in the dataset (a sequential search). Moreover, specialized data formats are vulnerable to backward-incompatible changes in the software used to load them and are more difficult to maintain in the long run.

Database structure. Finally, one may store the dataset in a database format, i.e. in a format optimized for searching. As above, such a solution will require additional software for users to manipulate the data, but the data formats are more stable in practice since database software is widely used beyond mathematics. There are many types of database available, each optimized with different applications; in this article, we will focus on two:

- relational database — a collection of ledger-style tables with a fixed schema, where the structure of each row is constant. This kind of database has been prevalent for decades; major examples include Oracle [35], MySQL [34], PostgreSQL [38], and SQLite [21].

- document oriented database — a schema free database, where each object is stored in a single instance in the database, and every stored object can be different from every other. MongoDB [31] is one of the most famous instantiations of this paradigm.

In Section 4, we will contrast PostgreSQL and MongoDB.

3.2. **Front end.** There are many choices for how to enable a user to interact with data. Different options have varying degrees of accessibility and utility; we describe three possibilities below.

Minimal. The author may decide to provide direct access to the database. For example, the author may make the database files available through email or a webpage; or allow direct read access to a SQL server [8].

Computer algebra system. If the database is provided in a computer algebra system dependent files, these might evolve into being part of algebra system itself, either as a standard or an optional package. For example, the Small Groups Library [7] is available in GAP and the K3 Database in Magma [30]. This option is extremely convenient for users of those software systems but limits their availability to others.

Static website.
The user may also interact with the database through a collection of web pages displaying the data. These web pages can range widely in complexity. For example, one can manually curate each page individually [3, 23] or automatically generate them as static web pages [14, 15, 26, 28, 45]. Such webpages can provide a very usable interface, but their main weakness is that the html code for displaying the data must be duplicated for each object presented, introducing a large overhead that becomes prohibitive as the size of the database grows.

Dynamic website. Alternatively, one can instead use a web application to generate each page when it is accessed. This is essential for large datasets, as generating and storing a static web page for each object can be infeasible. The LMFDB follows such an approach, where the web page displaying the content is generated on the fly through a dedicated web application based on Flask [40] and HTML templates, and only a small amount of the content is human-curated. Other examples of mathematical databases using such an approach include [6, 10]. The main downside of this approach is a higher level of complexity when compared to the static option.

3.3. **Hosting.** Finally, we discuss some of the options for where to physically host the front end and the database. We focus on how each possibility varies in terms of cost, ease of maintenance and scalability.

Personal hard-drive. This is one of the simplest options, as it has no initial set up cost. However, it severely limits the database's accessibility to others due to the lack of a front end.

Personal web page. If one already maintains a personal web page, a convenient option is to also use it to host the database and the front end. However, the scope might be limited, as many universities will only allow their users to host static content.

Web framework. Another convenient option is to use a web framework service (e.g. GitHub, CoCalc, Wordpress/Wikidot/blogs). For example, some people might dump their database into GitHub, while others might also use it to host the front end. These options offer a lower initial set up cost with a minimal long term maintenance cost, however, the scope is also limited by the framework, since they control what software is used to host content.

On a university/personal server. This solution provides complete control. However, there is both a high initial cost in buying the servers and setting them up. Moreover, one needs to maintain the server for the duration of its life: installing security updates, dealing with service interruptions, and managing backups. In addition, the database's scope might be limited by the capabilities of the initial hardware choices. The LMFDB uses a couple of servers purchased through grants to provide a development platform for our contributors, and host the back end and the front end for `beta.lmfdb.org`

Using a hosting service. Servers in the cloud [4, 20] provide complete control and have no monetary startup costs, but require ongoing payments proportional to the resources used. The main benefit is that one does not need to worry about maintaining the servers themselves; cloud servers also have the ability to easily scale the resources available if necessary. This option is currently used by the LMFDB to host `www.lmfdb.org`.

4. MongoDB vs PostgreSQL

In 2009, when MongoDB was initially released as an open source project, it presented itself as an exciting new option to easily store mathematical objects. As a document-oriented database, where each document was stored independently in a JSON-like format, it did not require a schema, a boon when creating new collections of mathematical objects where the data requirements only gradually became clear. Moreover, Python bindings were available, a language that many computational number theorists were familiar with due to its use in SageMath. This setup worked well for many years; however, as the LMFDB grew, various shortcomings of using a non-relational database became more evident. In March 2018, we decided to transition from MongoDB to PostgreSQL, a popular open source implementation of the SQL language that grew out of earlier projects founded in the 1970s and 80s. As a relational database, PostgreSQL stores data in a structured way, where the data is organized in tables, and each table has a specified schema, i.e., each row must have the same layout, with the types of columns constant across all rows.

4.1. **Comparing two database systems.** MongoDB and PostgreSQL offer two different database paradigms. MongoDB, as a document-oriented database, is meant to facilitate the storage of unstructured data, where the fields and types present in each document can vary within a collection. This flexibility is extremely convenient since the data structure can organically evolve as the project develops. However, it can also easily lead to errors and unnecessary overhead for a structured database. For example, field names were kept short in MongoDB to save space, and multiple typos in the field names were found during the transition to PostgreSQL. Furthermore, differing layouts across documents required more complicated processing code. In contrast, PostgreSQL is a relational database with fixed schemas that enable queries across multiple tables. Since the LMFDB was already a uniform set of fields in MongoDB (so much so that an inventory application was written to extract a schema from the data), creating a formal schema was fairly straightforward. Database-level schema support improves robustness and performance, and PostgreSQL's JSONB type allows for schemaless columns when desired.

One benefit of switching has been that storing the data in PostgreSQL requires substantially less space than in MongoDB. For example, when converting our biggest MongoDB collection `Lfunctions.Lfunctions` to the PostgreSQL table `lfunc_lfunctions`, we observed a space savings of about 42%, going from 194GiB to 113GiB. These savings translate both to monetary savings (we were spending a total of about $2000 per year paying for storage space using MongoDB) and to query speed when a sequential search is performed.

Two main factors contribute to the smaller storage requirements. First, we no longer need to (repeatedly) store the name of each field in every document; this was a MongoDB requirement, as fields in a document can vary across a collection. For some of the LMFDB's collections, these names accounted for a substantial fraction of the storage requirement. In addition to the space benefits, the switch to PostgreSQL also relaxes the pressure to minimize the lengths of these names, improving readability.

The second factor contributing to space savings is PostgreSQL's support for arbitrary precision integers. As a mathematical database, the LMFDB frequently uses large integers, which had to be stored as strings in MongoDB. In addition to the storage consequences, various workarounds were required to sort and search the data correctly. The elimination of these hacks has simplified some of the supporting code.

While smaller tables help with query performance, indexes provide a more powerful tool for speeding up searches. Indexes facilitate the location of records with constraints on a specified set of columns by storing additional data. Having such indexes available is key to providing quick search results for typical queries. For example, both MongoDB and PostgreSQL support indexes based on binary trees, which work well for totally ordered data such as integers and strings. However, there is another query type that is not well supported by binary trees: given a column representing a set of integers, search for rows that contain a particular set, or are contained in a particular set. For example, we store the set of ramified primes R_K for number fields K, and we want to be able to search on number fields by specifying such constraints on R_K. Even though we supported such searches through our web interface, many such searches took several minutes to perform within MongoDB, which caused our web front end to timeout. In PostgreSQL, most of these queries finish in a couple of seconds, both when the number of results is not large enough to justify using an index (the first three rows of Table 1), and when generalized inverted indexes (GIN) [43] apply (the last row of Table 1).

The significant improvement in indexless queries is another considerable advantage of the transition to PostgreSQL. Such queries are relevant, since non-typical queries can be the most interesting mathematically, and since it is impossible to construct all indexes in advance because the LMFDB offers the user a lot of freedom

[2]We used WiredTiger for comparisons since it outperforms MMAP when using normal amounts of RAM

Query	result/total	MongoDB[2]	PostgreSQL
$\#\{K : 5 \notin R_K\}$	71%	6h 9min	2.93s
$\#\{K : \{2,5\} \subset R_K\}$	18%	10min 2s	6.9s
$\#\{K : R_K \subset \{2,3,5\}\}$	4%	14min 34s	3.42s
$\#\{K : R_K \subset \{2,3,73\}\}$	0.1%	8min 11s	170ms

Table 1. Times for counting the number fields K satisfying the given constraint on set of ramified primes.

to select search parameters. We present two such examples in Table 2 (the first for number fields and the second for elliptic curves), where the number of results is about 1000 in each case. We also note that successful queries are instantaneous on a second run, as both databases cache the results.

Query	MongoDB	PostgreSQL
$\{K : 5 \notin R_K, \lvert\operatorname{disc}_K\rvert \leq 2000\}$	1.95s	0.1s
$\{E : \#Ш = 4, \operatorname{rank} E = 1, \#E_{\text{torsion}} = 1\}$	1min 58s	1.95s

Table 2. Times for fetching the objects/rows satisfying the given constraint.

While search performance is vital for website usability, the process of uploading data and creating indexes is also essential to make the development sustainable and to aid in the implementation of new features. PostgreSQL's `COPY FROM` command allows for the rapid importing of data from plain-text files without a Python intermediary, which is useful when the data is generated by some other system such as Magma [9]. For example, we imported a 14 398 359 row table from a 680GB text file in about 8 hours. With MongoDB we estimate that would have taken us several days. Creating indexes in PostgreSQL is also dramatically faster than in MongoDB, approximately by an order of magnitude. Additionally, PostgreSQL offers a more diverse and mature set of open source front ends and administration tools, for example [36, 47], that lower the learning curve for the SQL language. The transition has substantially improved database management.

In addition to performance improvements, PostgreSQL makes new kinds of queries possible. Relational databases allow joining tables, returning values from both tables when they share some specified relationship. The existing schemas, having been designed for MongoDB, do not take advantage of this feature, but new tables on finite groups and on p-adic tori will.

4.2. Transition.

Once the decision had been made to switch from MongoDB to PostgreSQL, we faced two main tasks: to port the data and to update the LMFDB codebase to use PostgreSQL. The majority of the work was done by the second author between March 2018 and August 2018, with assistance from the first author starting in July.

Since PostgreSQL requires the specification of schemas, the initial step of the data migration involved defining the schema for each table. In practice, most of the tables had a schema already, which were made visible by a custom inventory module within the LMFDB.[3] Unfortunately, the accuracy of these schema varied:

[3]The inventory displayed the kinds of documents within each collection, the fields present on each type of document, the indexes present in each collection, and both machine and human

some collections had multiple independent kinds of records within the same table, while others had typos in some of the field names. Moreover, the inventory system itself proved to reflect the types of some columns inaccurately, since it inferred the type from finding a single record. For example, it concluded that a specific field was a real number, while some documents had complex values. Eventually, we overcame these issues and created a schema for each PostgreSQL table.

In some cases, there were multiple possible types to choose from for a given column. PostgreSQL 9.4 added support for a binary JSON type, which allows for unstructured data to be stored in a column. It provided an alternative to arrays that could also store an analog of Python's dictionaries, with strings as keys. For simplicity, we initially chose to use this JSONB type instead of arrays, since it simplified some of the supporting code. However, after much of the rest of the transition was complete, we realized that when arrays are possible (namely, when the type is uniform across all entries), they provide some significant advantages. Namely, arrays are more space efficient than the JSONB type, and indexes on arrays support more operations than indexes on a JSONB column. We are gradually transitioning JSONB to array columns where useful.

We also took advantage of the transition to change some data types that had been chosen due to the deficiencies in MongoDB. For example, many large integers had been stored as strings since MongoDB lacked an arbitrary precision data type. With the switch to PostgreSQL, we were able to use the numeric type instead, which simplified code used to sort and compute with the results.

Since one of the goals of the switch was to improve performance, we made some changes to the schemas to help speed up queries. In particular, we split some of the largest tables in half, putting columns that were used for searching into a search table and columns that were only used for display into an *extras table*. This significantly reduced the search table's size, improved index efficiency, and sped up indexless queries that had to traverse the whole table. Query performance is also affected by sort order, and the default sorting on many tables uses four or five columns. We added a primary key column to each table, and for large tables we ordered this primary key by the table's standard order. This feature decreased the size of indexes and improved their performance since search results could be ordered by a single numeric column rather than a combination of many columns. This optimization was feasible because the data in the LMFDB changes infrequently, and because each search page comes with a default order.

With the schemas created and export scripts written, the actual changeover of the data went reasonably smoothly. PostgreSQL provides a COPY FROM command to load data from a text file, which runs very quickly. After several false starts, exporting 400GB of data from MongoDB to text files took three or four days; copying it between servers on opposite sides of the Atlantic took less than a day, and loading it into PostgreSQL took less than a day.

Adapting the LMFDB codebase to use PostgreSQL took longer. The backend code for the LMFDB is written in Python, and the Python bindings for MongoDB and PostgreSQL take a very different approach. PyMongo [32] offers a high level interface to the database that integrates python data structures. Queries are performed by constructing dictionaries that specify the values of certain columns and

generated information on the types of each field. It was a very useful resource for constructing schema.

have special keys for adding more complicated constraints such as inequalities or boolean combinations of conditions. Psycopg [46], on the other hand, is a thin interface that offers the user the ability to execute SQL statements. Constructing queries using dictionaries worked well as a model for the LMFDB, since the search pages provided many inputs that could be processed independently into a dictionary. Moreover, we did not want to require LMFDB developers to learn SQL. Therefore, we implemented a high-level interface to PostgreSQL that translated a dictionary-based query system similar to PyMongo's into SQL SELECT statements while avoiding SQL injection attacks. This query interface has already proved useful in rapidly prototyping another project, researchseminars.org, and we plan to make it available as a stand-alone tool.

5. Benefits

5.1. **Abstraction.** As the LMFDB has grown, the most common paradigm for adding new features has been to take a functional section of the website, copy it to a new folder, and modify the templates and backend code. This approach has the benefit of easily producing valid code since it is iteratively making small changes to an already functional webpage. However, it leads to large quantities of repeated code, which make fixing bugs and adding features much harder, since the same change must be applied in dozens of locations. The standard solution is to encapsulate tasks into functions that can then be used repeatedly and changed as needed. Of course, the LMFDB has used functions like this since the beginning, but often these functions are not written in a way that allows them to be used in multiple sections of the website.

The task of switching from MongoDB to PostgreSQL was made more difficult by this low level of abstraction, but it also presented an opportunity. Since the database change necessitated changing a large number of files throughout the project, it offered a perfect time to make additional changes. We highlight three in particular to illustrate the kinds of encapsulation that accompanied the database change: a new data-management interface, more robust and uniform handling of search inputs, and common tools for creating statistics pages.

We centralized many of the scripts used to upload data into a suite of data management tools. There is now a single object available from the Sage command line that has methods for performing a variety of tasks: creating new tables and modifying existing schemas, copying data to and from text files, moving data between the beta and production versions of the database, accessing PostgreSQL's tools for analyzing slow queries, adding indexes and constraints, and tracking statistics as data is changed. The centralization of these tasks allows us to reduce code duplication and use best practices to handle errors.

Each section of the LMFDB has a search page, where users can input constraints they would like their curve, field or group to satisfy. Some input boxes require an integer or list of integers; others need a label string in a particular format; others a rational number. The processing code must process the user provided text, raising appropriate errors on invalid input, and transform the set of inputs into a database query. Before the database change, we had already begun the process of encapsulating this task. Each input box corresponds to a specific field or column in the database, and there is a processing function for each type of input (list of integers, floating point, etc.). As we switched each part of the LMFDB to PostgreSQL, we ensured that it was using these functions.

Once these queries have been constructed, we need to execute the search and pass the results to the Jinja [39] template that actually creates the webpage seen by the user. Various parts of this task are shared among all sections of the LMFDB. For example, we have to be able to handle errors in user input, errors or timeouts in the execution of the query, and jump boxes that allow a user to go straight to an object with a particular label. As part of the switch to PostgreSQL, we created a *search wrapper* that performs all of these jobs, and adopted it across the LMFDB. In addition to simplifying the code overall, this change made the user experience more uniform and allowed for the creation of new features, which will be described in the next section.

Many parts of the LMFDB had a statistics page prior to the database change. These pages described the number of objects with certain attributes; for example, a table of how many number fields were available by signature, or a table giving the fraction of elliptic curves with a specified torsion structure. Along with the database change, we created a class to support the database queries required for these statistics, and a common template to display them nicely. This change has the benefit that it automates the process for updating these counts when developers add data. It also makes it easier to create these statistics pages for more sections of the LMFDB, and makes it easier to implement new features such as user-requested statistics.

5.2. **New Features.** In addition to the intrinsic benefits of using PostgreSQL, the transition has eased the implementation of a number of new features. While many of these would have been possible using MongoDB, the improved abstraction made implementing them much more manageable. And some, such as dynamic statistics and verifications, rely on PostgreSQL's improved performance and the relational model.

PostgreSQL supports transactions, allowing one to roll back to a functional earlier state if some later action fails. We have built these transactions into our interface layer, making it possible to safely modify rows in place, rather than creating a copy of the whole table when changes are made.

The fact that all interactions with the database pass through our interface layer allows us to add custom logging. For example, all queries that take longer than a threshold are recorded, allowing us to focus on creating the right indexes. We also log high-level actions, such as adding new data to a table or changing a schema, along with a username to provide a record of what changes have been made to the data. These logs help us ensure the quality of our data.

The statistics infrastructure has enabled a new user interface which we refer to as *dynamic statistics*. Rather than presenting only a pre-selected set of views, the user can specify which variables they are interested in and the objects' constraints. The system then generates a display describing the possible values of those variables and the number of objects possessing each possible set of values. For example, a user can create a table to view how the weight and level of modular forms vary among forms with complex multiplication. As of June 2020 we are still in the process of enabling this feature throughout the LMFDB. We hope that this feature will provide researchers with a new method of interacting with the LMFDB, allowing them to look for large scale patterns in these objects in addition to searching for objects with particular properties.

Many browse pages in the LMFDB already had a link to view a random object. Because of the new search wrapper abstraction, it was easy to add the ability to return a random object satisfying a search query. This feature saves time when browsing since it allows you to more quickly reach the homepage of an object satisfying a particular constraint, such as CM elliptic curve or a weight one modular form.

During the transition to PostgreSQL, we modified the search results pages so that they do not always provide the total count. The count is provided only if it is sufficiently small or has already been cached (though it is available in every case if requested). This change dramatically improved response times in some cases, where a large number of results makes finding the first few much easier than counting them all.

Mathematical data differ from data in most other applications because it has a notion of correctness that can be checked, rather than just providing a record of a real-world measurement. Some of these verifications are internal, such as checking that the defining polynomial of a number field is irreducible or that the sum of the complex embeddings of classical modular matches the trace of the exact newform. Others rely on connections with other tables in the database, such as checking that invariants matchup between an elliptic curve and its corresponding modular form. These kinds of verifications across different tables are much faster using PostgreSQL, since the relational database model is explicitly designed to search across different tables. The switch to PostgreSQL made it possible to create a framework to write such consistency checks and run them whenever new data is added.

We are excited about the new features that we added to the LMFDB and we believe that the transition to PostgreSQL will aid with the long term maintenance of the project. We look forward to how the abstraction layer we have created around database interactions will help standardize the user experience across LMFDB. We hope others will learn from our experience in their efforts to leverage computation and data towards the study of mathematics.

References

[1] Scott Aaronson, Greg Kuperberg, and Christopher Granade, *Complexity Zoo*, 2005–2019. https://complexityzoo.uwaterloo.ca.

[2] Jeffrey Adams, Annegret Paul, Ran Tui, Susana Salamanca-Riba, Peter Trapa, Marc van Leeuwen, and David Vogan, *Atlas of Lie Groups and Representations*, 2002–2019. http://www.liegroups.org.

[3] Mohammad Akhtar, Tom Coates, Alessio Corti, Sergey Galkin, Vasily Golyshev, and Alexander M. Kasprzyk, *Fano Varieties and Extremal Laurent Polynomials*, 2010–2012. http://fanosearch.net.

[4] *Amazon Web Services*, Amazon.com, Inc, 2006–2019. https://aws.amazon.com/.

[5] *arXiv*, Cornell University, 1991–2019. https://arxiv.org/.

[6] Dror Bar-Natan and Scott Morrison, *The Knot Atlas*, 2006–2019. http://katlas.org.

[7] Hans Ulrich Besche, Bettina Eick, and E. A. O'Brien, *The groups of order at most 2000*, Electron. Res. Announc. of the AMS **7** (2001), 1–4.

[8] Birne Binegar, *Unipotent Muffin Research Kitchen*, 2009–2019. http://lie.math.okstate.edu/UMRK/UMRK.html.

[9] Wieb Bosma, John Cannon, and Catherine Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3-4, 235–265. Computational algebra and number theory (London, 1993). MR1484478

[10] G. Brinkmann, K. Coolsaet, J. Goedgebeur, and H. Mélot, *House of Graphs: a database of interesting graphs*, Discrete Applied Mathematics **161** (2013), 311–314. http://hog.grinvin.org.

[11] Chris Caldwell, *The Prime Pages*, 1994–2019. https://primes.utm.edu.

[12] John Cremona, *The L-functions and modular forms database project*, Found. Comput. Math. **16** (2016), no. 6, 1541–1553. MR3579716

[13] John E. Cremona, *Algorithms for modular elliptic curves*, 2nd ed., Cambridge Univ. Press, Cambridge, 1997.

[14] H.N. de Ridder et al., *Information System on Graph Classes and their Inclusions*, 2001–2014. http://graphclasses.org.

[15] Tim Dokchitser, *GroupNames*, 2017–2019. http://groupnames.org.

[16] *Encyclopedia of Mathematics*, Kluwer Academic Publishers, 2002–2012. https://www.encyclopediaofmath.org.

[17] Edward Frenkel, *Love and math: The heart of hidden reality*, Basic Books, 2013.

[18] *GAP – Groups, Algorithms, and Programming*, The GAP Group, 2019. https://www.gap-system.org.

[19] *GitHub*, GitHub, Inc., 2008–2019. https://github.com.

[20] *Google Cloud Platform*, Google Inc., 2008–2019. https://cloud.google.com/.

[21] Richard Hipp et. al., *SQLite*, 2000–2019. https://www.sqlite.org.

[22] Alexander Hulpke, *Constructing transitive permutation groups*, Journal of Symbolic Computation **39** (2005), no. 1, 1–30.

[23] Joel David Hamkins, Victoria Gitman, et al., *Cantor's Attic*, 2011–2019. http://cantorsattic.info.

[24] John Jones and David Roberts, *A database of local fields*, Journal of Symbolic Computation **41** (2006), no. 1, 80–97.

[25] ———, *A database of number fields*, LMS J. Comput. Math. **17** (2014), no. 1, 595–618.

[26] Clark Kimberling, *Encyclopedia of Triangle Centers*, 1999–2019. https://faculty.evansville.edu/ck6/encyclopedia/ETC.html.

[27] Maximilian Kreuzer and Harald Skarke, *Calabi-Yau data*, 2000–2019. http://hep.itp.tuwien.ac.at/~kreuzer/CY/.

[28] Charles Livingston and Jae Choon Cha, *KnotInfo: Table of Knot Invariants*, 2019. https://www.indiana.edu/~knotinfo/.

[29] *The L-functions and modular forms database*, The LMFDB Collaboration, 2009–2019. http://www.lmfdb.org.

[30] *Magma databases*, Magma, 2001–2018. http://magma.maths.usyd.edu.au/magma/download/db/.

[31] *MongoDB*, MongoDB, Inc, 2009–2019. https://www.mongodb.com/.

[32] *PyMongo*, MongoDB, Inc, 2009–2019. https://api.mongodb.com/python/current/.

[33] Vipul Naik, *Groupprops, The Group Properties Wiki*, 2008–2019. https://groupprops.subwiki.org.

[34] *MySQL*, Oracle Corporation, 1995–2019. https://www.mysql.com/.

[35] *Oracle Database*, Oracle Corporation, 1979–2019. https://www.oracle.com/database.

[36] *pgAdmin4*, The pgAdmin Development Team, 2019. https://www.pgadmin.org.

[37] Simon Plouffe, Adam Van Tuyl, Paul Irvine, Loki Jörgenson, David H. Bailey, Peter Borwein, Jonathan Borwein, and Yasumasa Kanada, *The Inverse Symbolic Calculator*, 1986–2019. http://wayback.cecm.sfu.ca/projects/ISC.

[38] *PostgreSQL*, The PostgreSQL Global Development Group, 1996–2019. https://www.postgresql.org/.

[39] Armin Ronacher, et. al., *Jinja 2*, 2008–2019. http://jinja.pocoo.org/.

[40] ———, *Flask: a Python microframework*, 2010–2019. http://flask.pocoo.org/.

[41] Martin Rubey, Christian Stump, et al., *FindStat – The combinatorial statistics database*, 2013–2019. http://www.FindStat.org.

[42] *Sagemath, the Sage Mathematics Software System*, The Sage Developers, 2019. http://www.sagemath.org.

[43] Teodor Sigaev and Oleg Bartunov, *Gin for postgresql*, 2008–2019. `http://www.sai.msu.su/~megera/wiki/Gin`.
[44] N. J. A. Sloane et al., *The On-Line Encyclopedia of Integer Sequences*, 1964–2019. `http://oeis.org`.
[45] *The syzygy database*, The SyzygyData Group, 2017. `http://www.syzygydata.com`.
[46] Daniele Varrazzo, *Psycopg*, 2000–2019. `http://initd.org/psycopg/`.
[47] Jakub Vrána, *Adminer*, 2019. `https://www.adminer.org`.
[48] Eric Weisstein, *Wolfram Mathworld*, 1995–2019. `http://mathworld.wolfram.com/`.
[49] Robert Wilson, Peter Walsh, Jonathan Tripp, Ibrahim Suleiman, Richard Parker, Simon Norton, Simon Nickerson, Steve Linton, John Bray, and Rachel Abbott, *ATLAS of Finite Group Representations*, 1985–2019. `http://brauer.maths.qmul.ac.uk/Atlas/v3/`.

Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
Email address: `edgarc@mit.edu`
URL: `https://edgarcosta.org`

Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
Email address: `roed@mit.edu`
URL: `http://math.mit.edu/~roed/`