Cross-tier web programming for curated databases: a case study

IJDC | *Conference Pre-print*

Cross-tier Web Programming for Curated Databases: a Case Study

Simon Fowler
University of Edinburgh

Simon Harding
University of Edinburgh

Joanna Sharman
Novo Nordisk

James Cheney
University of Edinburgh

Abstract

Curated databases have become important sources of information across several scientific disciplines, and as the result of manual work of experts, often become important reference works. Features such as provenance tracking, archiving, and data citation are widely regarded as important features for the curated databases, but implementing such features is challenging, and small database projects often lack the resources to do so.

A scientific database application is not just the relational database itself, but also an ecosystem of web applications to display the data, and applications which allow data curation. Supporting advanced curation features requires changing all of these components, and there is currently no way to provide such capabilities in a reusable way.

Cross-tier programming languages have been proposed to simplify the creation of web applications, where developers can write an application in a single, uniform language. Consequently, database queries and updates can be written in the same language as the rest of the program, and at least in principle, it should be possible to provide curation features reusably via program transformations. As a first step towards this goal, it is important to establish that realistic curated databases can be implemented in a cross-tier programming language.

In this paper, we describe such a case study: reimplementing the web front end of a real world scientific database, the IUPHAR/BPS Guide to Pharmacology (GtoPdb), in the Links cross-tier programming language. We show how programming language features such as language-integrated query simplify the development process, and rule out common errors. Through a comparative performance evaluation, we show that the Links implementation performs fewer database queries, while the time needed to handle the queries is comparable to the Java version. Furthermore, while there is some overhead to using Links because of its comparative immaturity compared to Java, the Links version is usable as a proof-of-concept case study of cross-tier programming for curated databases.

This paper is a conference pre-print presented at IDCC 2020 after lightweight peer review. The most up-to-date version of the paper can be found on arXiv (Fowler et al., 2020).

Submitted 12 December 2020 ~ *Accepted* 19 February 2020

Correspondence should be addressed to Simon Fowler, Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB.

This paper was presented at International Digital Curation Conference IDCC20, Dublin, 17-19 February 2020

The *International Journal of Digital Curation* is an international journal committed to scholarly excellence and dedicated to the advancement of digital curation across a wide range of sectors. The IJDC is published by the University of Edinburgh on behalf of the Digital Curation Centre. ISSN: 1746-8256. URL: http://www.ijdc.net/

Introduction

Curated databases have become important data resources across several scientific disciplines. Such databases collect the current state of knowledge about a topic and many have become important reference works. They are constructed through the manual effort of experts, often over a long timespan, and it is widely appreciated that versioning and provenance-tracking are important for assessing the validity and freshness of the data, or tracing the origin of errors or discrepancies (Buneman et al., 2008). Unfortunately, implementing support for fine-grained provenance-tracking or versioning is a challenging task, usually performed on a system-by-system basis. Many such database projects, particularly smaller or shorter-term ones, lack the resources and expertise to do this.

A curated database is not just an isolated relational database, but also has surrounding infrastructure such as a web application for viewing or searching the data, and an editing interface used by the database curators to add or modify data. Each of these components are nontrivial to develop. A typical web application is really a distributed program involving code running on several "tiers": Java or Python running on the server, JavaScript and HTML on the web browser, and SQL on a database. Curation interfaces can be either web applications or traditional client-server database applications; in either case, modifying such a system is a nontrivial task, especially when the added functionality spans two or all three tiers.

Thus, augmenting an existing curated database web application (or designing a new system from scratch) to provide features such as versioning, provenance-tracking, or citation requires taking these requirements into account across two or more system layers, adding complexity beyond that of the basic functionality of the system. General-purpose techniques have been explored for supporting such features (Buneman et al., 2008, 2004), but there is currently no way to provide them in a reusable way.

Cross-tier web programming

Cross-tier programming languages (Chlipala, 2015; Cooper et al., 2006; Radanne et al., 2016; Serrano et al., 2006) have been proposed to simplify web and database application programming. The vision is that the programmer should only need to write a single program in a single language; the language implementation then takes care of the details of partitioning the program into client, server and database components, distributing the code, and coordinating their communication in the running program. A major benefit is the fact that database queries and updates can be written and checked for consistency in the same language as the rest of the program. In principle, advanced features such as provenance and versioning could be provided for such programs by program transformation: that is, by rewriting the program (possibly with some lightweight annotations) so that new functionality is implemented according to a high-level pattern. Indeed, Fehrenbach and Cheney (2018) have already shown how provenance tracking can be added as a programming language feature: a user must simply change a keyword in a query in order to obtain provenance metadata, rather than hand-crafting provenance tracking per application.

We argue that cross-tier programming is well-suited for curated databases: by using a single cross-tier language rather than a conventional multi-language approach, curated database developers should be able to focus on their application logic, and could (in the future) use pre-packaged techniques for provenance-tracking and archiving provided by the language implementation (or even a library). However, to date, cross-tier programming languages have not been widely used for curated databases. It is important to establish that such languages are capable of supporting the requirements of curated databases, and to identify any shortcomings that need to be overcome.

Contributions

In this paper we provide the first case study of cross-tier web programming for scientific databases by using Links, a functional, cross-tier web programming language (Cooper et al., 2006), to implement a workalike web front-end for the IUPHAR/BPS Guide to Pharmacology Database (GtoPdb), an important curated pharmacological database (Harding et al., 2018). Links is a research project that has been developed in Edinburgh over many years, and is not a widely-used mainstream programming language; however, by using it to develop case studies such as this one, we plan to demonstrate the value of cross-tier programming for scientific databases, and evidence the viability of language-based support for curation.

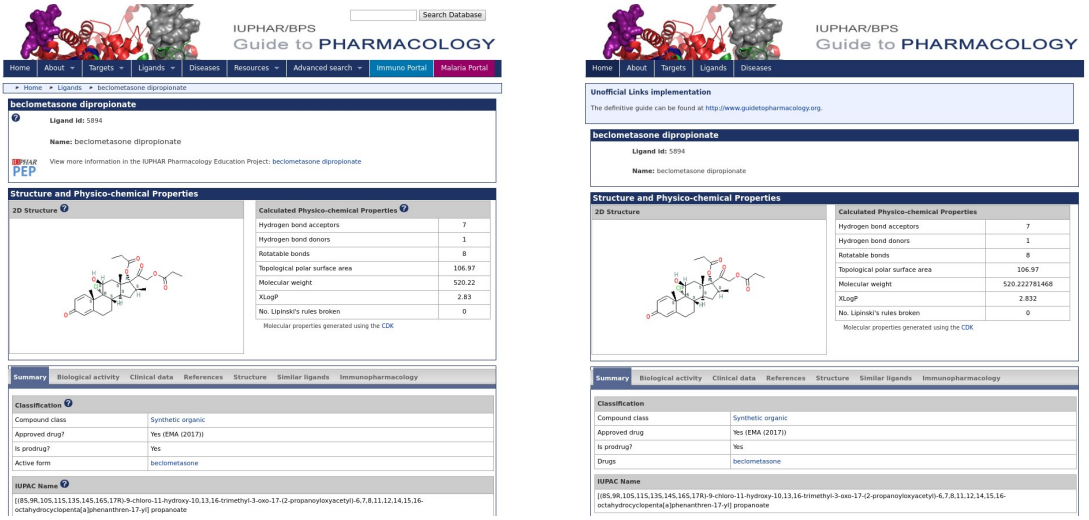
In the remainder of the paper, we describe the background of GtoPdb and why it is an interesting database to use as a case study, describe some aspects of the Links implementation, and report on the results of a performance evaluation which shows that the Links implementation performs comparably with the official Java implementation, as well as providing lower overall query counts and more predictable performance results for database queries. We conclude with a summary of lessons learned so far and directions for future work.

Background

The International Union of Basic and Clinical Pharmacology (IUPHAR) / British Pharmacological Society (BPS) Guide to Pharmacology (GtoPdb) is an expert-curated database that captures interactions between human proteins ("targets") and ligand molecules from the pharmacological and medicinal chemistry literature. The resource is open-access and intended as a "one-stop shop" portal to pharmacological information. It provides a searchable database with quantitative information on 3,000 drug targets and related proteins, organised into families, and 9,700 approved and investigational drugs, antibodies, and natural hormones, metabolites, and neurotransmitters that act on them. GtoPdb provides succinct overviews, key references and recommended experimental ligands for each target. It is a useful resource for researchers and students in pharmacology and drug discovery and provides the general public with accurate information on the basic science underlying drug action.

GtoPdb has its origin in IUPHAR-DB which was first compiled in 2003 (Harmar et al., 2009; Sharman et al., 2013, 2011). Its scope was expanded between 2012 and 2015 to define the data-supported druggable genome, and was renamed GtoPdb (Pawson et al., 2014; Southan et al., 2016). GtoPdb is distinguished by a unique model of data collection and curation, with the guidance and support of the Nomenclature Committee of IUPHAR (NC-IUPHAR) and its 96 target class subcommittees. These subcommittees comprise over 500 pharmacology experts who provide regular updates and contributions to GtoPdb. The GtoPdb web application communicates with two underlying databases: a main, PostgreSQL, database which contains the bulk of the data and a second, Oracle, database which contains ligand structure information. The main application layer is written in Java, with static pages written in Java Server Pages (JSP), JavaScript and HTML. The curation interface (i.e., the interface used by curators to create and edit the database) is a custom-built, standalone Java application with a GUI.

GtoPdb is a substantial curated database: the 2019 release comprises 89 megabytes of data contained in 181 tables. As a measure of scope, the Java codebase for the web interface (which includes some pages out of the scope of our reimplementation) stands at 17935 lines of code for data transformation code; 28819 lines of JSP rendering code; and a data access layer (which also contains query code used for the curation interface) consisting of 43129 lines of code, written over a period of 16 years.



(a) Official GtoPdb **(b)** Links reimplementation

Figure 1. Screenshots of official GtoPdb application and Links reimplementation

Reimplementing GtoPdb in Links

We now turn our attention to our reimplementation of the GtoPdb frontend in Links. Our reimplementation uses an unaltered copy of the PostgreSQL database release.
Figure 1 shows an example page, displaying a view of the ligand information for beclometasone dipropionate extracted from the database. Figure 1a shows the official version of the page, and Figure 1b shows our reimplementation in Links. The same underlying information is displayed on each page. The Links version has some minor differences such as different rounding used for floating-point numbers, as well as a banner to differentiate it from the official version.

GtoPdb Structure

The GtoPdb interface consists of nine main data pages:

Target List

GtoPdb groups pharmacological targets into different categories including G protein-coupled receptors, ion channels, nuclear hormone receptors, kinases, catalytic receptors, transporters, enzymes, and other protein targets. This page links to the family list for each type of target.

Family List

A *family* is a group of related pharmacological targets. The family list page displays a hierarchically-ordered tree of families.

Family Summary

The family summary page provides summaries of each target in the family, and links to the more in-depth object data pages.

Object Data

In GtoPdb, an *object* is a pharmacological target such as a receptor. The object data page displays all information associated with a target, and is the most complex page. The page can render 52 individual properties about each object (for example, associated interactions and 3D structures).

Disease List
A list of all diseases in the system.

Ligand Families
A list of ligands classified into families.

Ligand List
A list of all ligands in the system, with the ability to filter by category.

Ligand Data
Displays data associated with ligands, such as relevant interactions, structural information, and a summary of clinical use.

Disease Data
Displays information about a disease, including references to external databases, related pharmacological targets, and ligands known to affect the disease.

The official implementation contains additional smaller auxiliary data pages and searching functionality, but the nine pages above are the most prominent and involved, so we concentrate on these pages for our Links reimplementation.

Language-Integrated Query

In the Java implementation of GtoPdb, all database queries are carried out using SQL prepared statements. Our first major departure from the previous implementation is the use of Links's support for *language-integrated query*. While language-integrated query is best known in Microsoft .NET languages such as C# and F# (Meijer et al., 2006; Syme, 2006), it is also available as part of Links (Cooper, 2009; Lindley and Cheney, 2012), based on techniques developed originally in the Kleisli system (Wong, 2000).

Instead of constructing SQL statements directly, language-integrated query allows database queries to be written as a standard expression in a programming language. In particular, we use a flavour of language-integrated query pioneered by Trinder (1991), who adapts the notion of a list comprehension (similar to a mathematical set comprehension) to the setting of relational queries. As an example, consider an SQL expression which retrieves the names of all ligands which have been approved for use in humans. In SQL, we might write:

```
SELECT name FROM ligand WHERE approved
```

Given an appropriate Links declaration of the `ligand` table, we can write the above query as:

```
query { for (l <-- ligand) where (l.approved) [l.name] }
```

The query is written as a list comprehension, with elements generated by the `ligand` table, where the `where` clause filters each element to only consider those which are approved. Unlike embedded SQL, the query is also typechecked to ensure the table field names and query results are used consistently in the program.

Differently from most implementations of language-integrated query, Links supports efficient *nested queries*. A nested query is a query whose result type contains collections nested inside other collections. (In contrast, an SQL query always returns a *flat* table: a collection of records of values of primitive types such as integers and strings.) To illustrate, the following nested query returns records including the ligand name and its set of synonyms:

```
query {
  for (l <-- ligand)
          [(name = l.name, synonyms =
              for (l2s <-- ligand2synonym)
```

```
                where (l2s.ligand_id == l.ligand_id)
      [l2s.synonym])] }
```

This query will return a list of records of the form (`name`,`synonyms`), in which `name` is a string and `synonyms` is a list of strings. A natural, but inefficient, way to execute such a query is to first retrieve the set of all ligands (with their names and IDs), and then run one query per ligand to find its synonyms. In Links, the above nested query is instead transformed to two equivalent SQL queries. It is important to note that the ***shredding*** technique to implement nested queries proposed by Cheney et al. (2014) gives a guaranteed upper bound on the number of SQL queries needed to run a nested query: the upper bound is the number of occurrences of collections in the query result type, and this is independent of the number of records returned by a query.

A second useful feature of language-integrated query in Links is that certain user-defined functions can be used within queries for convenience, and such functions will be ***inlined*** to simplify the query expression to a form that can be translated to SQL. This relies on query normalisation (Cooper, 2009) to generate efficient SQL code directly from query expressions. As a simple example demonstrating both nested queries and user-defined functions at once, the nested query above can also be written as:

```
fun getSynonyms(id) {
  for (l2s <-- ligand2synonym)
    where (l2s.ligand_id == id)
    [l2s.synonym]
}
query {
  for (l <-- ligand)
    [(name = l.name, synonyms = getSynonyms(l.name))]
}
```

In this example, this capability makes the query expression longer (but arguably a bit more readable due to the extra documentation provided by the function name); however, such functions can also be *reused* across many query expressions, potentially saving a great deal of code repetition, and aiding maintenance.

Example: Listing Ligands

GtoPdb provides functionality for listing all ligands in the database, filtered by category: example categories include approved drugs, or ligands relevant to either immunopharmacology or malaria pharmacology. Let us consider this page as an extended example.

Each row in the displayed table shows the ligand's name, unique GtoPdb ID, synonyms or trademark names, and icons displaying whether the ligand is an approved drug; contains a radioactive or chemical (e.g. fluorescent) label; is relevant to immunopharmacology; is relevant to malaria pharmacology; or has an entry in the protein 3D structure database (PDB). We can gather this information through the use of a single query expression:

```
  query {
    for (l <-- ligand)
    where (ligandFilter(l, filterType))
      [ (id = l.ligand_id, name = l.name, approved = l.approved,
         radioactive = l.radioactive, labelled = l.labelled,
         immuno = l.in_gtip, malaria = l.in_gtmp,
         synonyms =
           for (l2s <-- ligand2synonym)
           where (l2s.ligand_id == l.ligand_id && l2s.display)
           [l2s.synonym],
         hasPDB = not(empty(
           for (p <-- pdb_structure)
           where (p.ligand_id == l.ligand_id)
           [ p ])))
      ]
  };
```

We begin by querying the ligand table. If the ligand matches a given predicate based on the filter type, then the query produces a record with the required information. Of particular interest are the synonyms and hasPDB fields of the output record, which are not fields in the ligand table but instead refer to other tables.

The synonyms field is a one-to-many relation from ligands to synonyms. As an example, the common painkiller paracetamol is also known by the trade names Panadol and Tylenol. The Java implementation gathers the relevant synonyms using a PostgreSQL view. To express this nested relation in Links, we use its support for nested queries (Cheney et al., 2014), as explained above, to populate the synonyms field with a collection of all relevant synonyms.

The hasPDB field should be true if the pdb_structure table in the database contains an entry with the same ligand ID as the current ligand. Note that we can use the native Links functions not and empty in Links code; these are translated to SQL EXISTS constraints.

Functional Predicates

Let us revisit how we filter the ligands to display. Some filters are based on boolean flags in the ligand table (for example, approved), or on the type field, and others perform more complicated tests. In the Java implementation, such filtering is implemented by building a query using string concatenation and Java conditional expressions. For example, to filter all approved drugs, the implementation uses code like the following:

```
if(type.equalsIgnoreCase("Approved")) {
  query += " WHERE approved IS TRUE ";
} else if (type.equalsIgnoreCase("Synthetic organic")) {
  query += " WHERE type = 'Synthetic organic' "
} ...
```

Each ligand type has a case which adds the correct type into the query, chained as else if clauses. Query strings generated in this way could be ill-formed, leading to failure at runtime (for example, if spaces between concatenated strings are omitted). Instead, we can take advantage of the fact that Links is a functional programming language, and define a function that tests whether a ligand matches a filter. We begin by defining a variant type (similar to an enum in Java) describing each filter:

```
typename Filter =
  [| Approved | SyntheticOrganic | EndogenousPeptide | Immuno
| ... |];
```

We then define a function called ligandFilter that given an entry in the ligand table, and a filter type, returns whether the ligand matches the filter:

```
fun ligandFilter(ligand, filterType) {
  switch (filterType) {
    case Approved           -> ligand.approved
    case SyntheticOrganic   -> ligand.type == "Synthetic organic"
    case EndogenousPeptide  ->
      ligand.type == "Peptide" && isEndogenous(ligand)
    case Immuno             -> ligand.in_gtip
    ...
  }
    }
```

Note that the EndogenousPeptide case calls another function isEndogenous, illustrating that we can use functions to break the query logic down into smaller parts. The ligandFilter function can be used directly in the query, and Links correctly inlines it (and isEndogenous) so that the eventual SQL query is similar to the one generated by the Java code. Using the number of lines of code as a rough measure of complexity, the Java version needs 145 lines of code to filter the list of ligands, whereas the Links version requires only 54, with the additional advantage that Links will always generate type-correct queries.

Displaying a Data Page

We have now seen an example of how Links can be used to implement a GtoPdb data page. More generally, in both the Java and Links implementations, the process for displaying each data page is as follows:

1. Parse any input arguments to the page request (for example, ligand or object ID)
2. Perform database queries to populate a data model
3. Parse all text fields in order to obtain a list of any referenced scientific literature and relevant ligands
4. Render the web page content and deliver the response

In the Java implementation, there is a single Java data model used for both the web interface and the curation tool, and sometimes this means that additional information must be retrieved but is not displayed. In the Links implementation, each page has its own data model based on what is to be displayed to the user, but the queries and processing code can be reused over different files. The Java implementation makes use of a data access layer which contains many methods to populate the model, whereas each Links page begins with a large nested query followed by a processing phase.

Text fields in GtoPdb may contain references to supporting scientific papers and crossreferences to ligands. As an example, consider the following excerpt, detailing comments about the agonist interactions targeting the D_1 dopamine receptor:

```
Some substituted benzazepines such as SKF-83959 are G-protein biased
agonists of the dopamine D<sub>1</sub> receptor and fail to activate
&beta;-arrestin recruitment <Reference id=28036/>; their ability to signal
through G<sub>q</sub>-mediated pathways has been controversial <Reference
id=33435/>.<br><br><Ligand id=6077/>, <Ligand id=9637/> and related
compounds exhibit slow dissociation rates from the D<sub>1</sub> receptor.
```

References to scientific papers are introduced using `Reference` tags, and references to ligands are introduced using the `Ligand` tag. The text also includes standard HTML tags. The above text would be rendered as shown below:

> Some substituted benzazepines such as SKF-83959 are G-protein biased agonists of the dopamine D_1 receptor and fail to activate β-arrestin recruitment [17]; their ability to signal through G_q-mediated pathways has been controversial [32].
>
> A68930, A77636 and related compounds exhibit slow dissociation rates from the D_1 receptor.

In order to display the text, we need to parse the `Reference` and `Ligand` tags, so that the required data about the corresponding references and ligands can be fetched in a subsequent query. This separate pass allows us to build a sorted, numbered, reference list, rather than storing references per data page in the database in a more fragile way.

Evaluation

Our main criterion for success was to ensure that Links is powerful enough to support implementing a real-world scientific database. As our case study manages to implement all key data display pages, we consider this objective met.

Nevertheless, it is important to ensure that our cross-tier methodology is not unacceptably detrimental to performance. We therefore evaluate our approach on four of the main data pages: the object data page; the disease data page; and the lists of all diseases and ligands.

We evaluate each page with respect to three dimensions:

Query Count

The number of queries executed when generating a given page. As the nested language-integrated query approach used by Links ensures that query count is bounded by the number of collection types in the result (Cheney et al., 2014), we would hope that the Links implementation would generate substantially fewer queries than constructing queries by hand.

Query Handling Time

The amount of time spent processing database queries. In the Links implementation, this also includes the time spent normalising the query into SQL and parsing the results into Links data structures, in addition to query execution itself.

Page Build Time

The amount of time spent building a page on the server, measured from the point at which the request is received until the point before the response is sent to the client. As Links is an interpreted language (as opposed to Java, which runs on a virtual machine incorporating a Just-In-Time (JIT) compiler), we would expect page build time to be slower on Links. We show results both including and excluding query handling time.

We collect the metrics by adding instrumentation code to the Links interpreter itself, and by implementing an instrumented version of the `PreparedStatement` class in the Java code. Measurements were performed on both the Links and Java versions of the code running locally on a laptop with an Intel Xeon E-2176M 2.7GHz CPU, and 8GB of RAM. We used the Python pandas library (McKinney, 2010) for data processing, and matplotlib (Hunter, 2007) to generate charts. The data generated by our experiments, as well as the code used to generate the charts, is publicly available on Figshare (Fowler, 2019).

Object Data

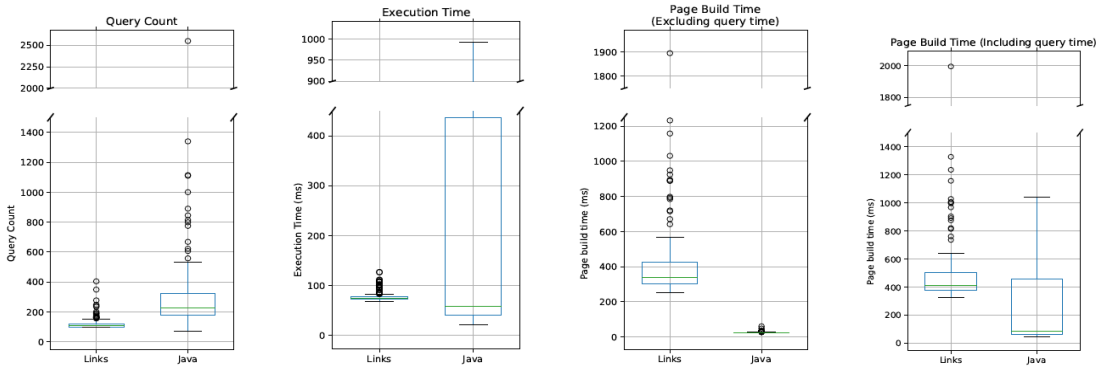
Figure 2 shows box plots detailing the number of queries, total query handling time, and page build time when displaying the object data page for 150 randomly selected object IDs. The data represents the arithmetic mean of 15 requests for each page.

Query count

Figure 2a shows the results for the number of queries generated to display the page. As expected due to the use of nested queries, the Links implementation generates a lower number of queries (median: 108) compared to the Java version (median: 229). Notably, the query count of the Links implementation is much more predictable, with a standard deviation of 43.38 in the Links implementation, in contrast to 275.15 in the Java implementation. The maximum query count in the Links implementation is 404, and the maximum query count in the Java implementation is 2549. The outlier in the Java implementation is due to object 262 (the Histamine H_1 receptor) being associated with an unusually large number of drug interactions: each interaction requires 9 database queries, along with additional queries to fetch information about the ligands associated with the interaction.

Query time

Figure 2b shows the query handling time for both implementations. The original Java implementation has a better median query time of 57.77ms compared to 101.36ms in the Links implementation: this disparity is likely due to a combination of query normalisation and marshalling the returned values into Links data structures.



(a) Query count **(b)** Query handling time **(c)** Page build time (excluding queries) **d)** Page build time (including queries)

Figure 2. Experimental evaluation of implementations (Object Data Page)

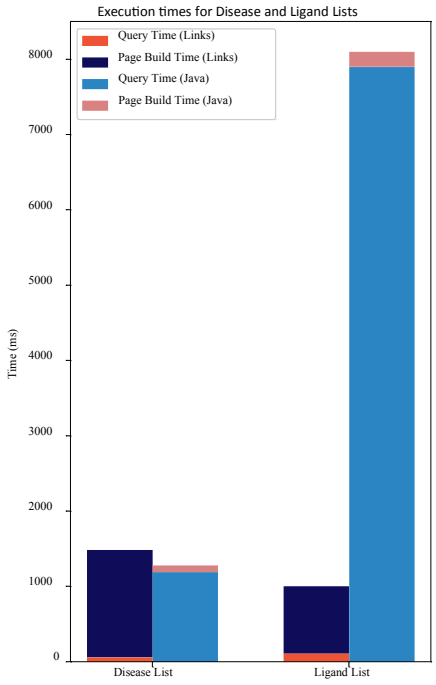


Figure 3. Experimental evaluation of implementations (Disease and Ligand Lists)

Nevertheless, again, performance of the Links implementation is more predictable with a standard deviation of 12.29 compared to 204.40 in the Java implementation.

Page build time

Figures 2c and 2d show the page build time for both implementations, excluding and including query handling time respectively.

As expected, the Java version performs substantially better than the Links version due to the maturity of the Java Virtual Machine and associated Java ecosystem. Concretely, the median page build time (excluding queries) for the Links version is 340.12ms and the median page build time for the Java version is 23.98ms. Additionally, the performance of the Java version is more predictable, with a standard deviation of 4.34 compared to 215.80 in the Links implementation.

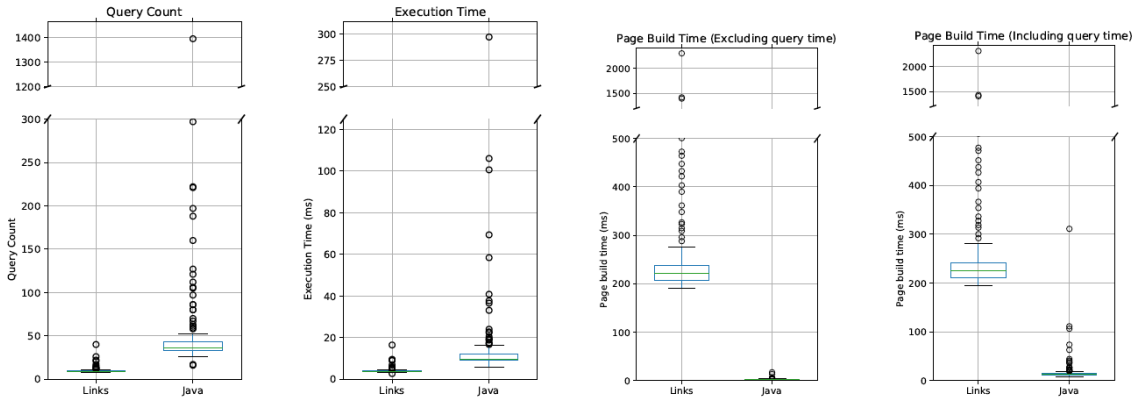
An additional bottleneck in the Links implementation is the implementation of a parser which is run on each text field in order to extract inline reference and ligand IDs contained within text fields. It is likely that improvements in this part of the code could lead to substantial performance improvements.

Disease and Ligand Lists

Figure 3 shows the experimental results for the pages listing all diseases and ligands: the data displayed is again the mean over 15 iterations.

Query count

The Links implementation substantially outperforms that of the Java implementations: the Links implementations use only two queries to gather the information required to display the lists, whereas the Java implementation of the disease list requires 8995 queries and the ligand list requires 30479. The additional number of queries is because the Java implementation populates a model which contains more information than is necessary for the page: as an example, the disease list page generates many queries retrieving links to external databases, but these are never displayed.



(a) Query count

(b) Query handling time

(c) Page build time (excluding queries)

d) Page build time (including queries

Figure 4. Experimental evaluation of implementations (Disease Data Page)

Query time

The number of additional queries required in the Java implementations is reflected in the time spent performing queries. Concretely, the mean query times in the Links implementation were 59.80ms for the disease list and 112.09ms for the ligand list, compared to 1192.15ms for the disease list and 7897.51ms for the ligand list in the Java implementation.

Page build time

As with the object data page, the Links implementation does not perform as well as the Java implementation when generating the page, due to the maturity of the underlying technologies. In fact, in spite of the large number of queries required, the Java implementation outperforms the Links implementation on the disease list (with an overall page generation time of 1483.95ms in the Links implementation and 1280.56ms in the Java implementation).

Disease Data

Figure 4 shows the results for the disease data page. Again, the results represent the arithmetic mean over 15 iterations for 150 randomly selected disease IDs.

The findings are consistent with the previously reported results, with Links performing substantially better on query count (Figure 4a) and comparably on query handling time (Figure 4b), but worse on page build time (Figure 4c). The outlier for query count and execution time in the Java implementation is Crohn's disease, which contains substantially more ligand interactions than the other disease pages considered. The same disease accounts for the outlier in the Links page build time, which is due to the necessity of parsing more description fields.

Related work

Our case study uses Links. There are other cross-tier languages, including Hop (Serrano et al., 2006), Ur/Web (Chlipala, 2015), and Eliom (Radanne et al., 2016). To the best of our knowledge, none of them has been used to implement curated databases.

Language-integrated query support is now being considered in several languages, for example including the Quill library for Scala.[1] We mentioned Links's support for nested queries as an important advantage of using Links for implementing GtoPdb. Similar techniques offering similar guarantees have been proposed by Grust et al. (2010), with the most recent step in this line of work being a language-integrated query library for Haskell called DSH (Giorgidze et al., 2010). It might be interesting to conduct a similar case study implementing GtoPdb in Haskell using DSH, and compare with the Links version; alternatively it may also be worthwhile to develop a Java or Scala implementation of language-integrated query that supports nested queries, that could be used natively with GtoPdb or other Java-based systems.

Conclusion and Future Work

In this work, we have produced the first real-world case study of a curated scientific database, the IUPHAR/BPS Guide to Pharmacology, in a cross-tier functional programming language. Our approach leverages language-integrated query, which makes it possible to write type-safe queries instead of manually constructing SQL.

GtoPdb is a substantial curated database, built over a period of 16 years. The current Links implementation runs on an unmodified version of the GtoPdb database release, with the 9

[1] Quill, https://getquill.io/

major data pages implemented. The codebase of our Links case study currently stands at 10981 lines of code after around 4 months of effort by the first author, who had previous experience with Links. While it may be tempting to attempt a direct comparison on lines of code for each page, such a comparison would be unreliable due to the difference in the structure of the two applications.

Finally, we have conducted a performance evaluation and shown that the use of Links does not impose unacceptable overheads; indeed, the use of language-integrated query and nested queries results in lower query counts and comparable time spent handling queries in general.

Future work

Our experience shows that Links is up to the task of implementing web application front-ends for curated databases, which is prerequisite to our goal of language support for data curation. We have already begun implementing the GtoPdb curation interface in Links; our next step is to turn our attention to the design and implementation of language features which will aid curation, such as archiving, inspired by the work of Buneman et al. (2004).

Our work has concentrated on relational databases; we also plan to investigate the theory and practice of language-integrated query for NoSQL databases, allowing us to implement case studies for a wider range of databases. This requires first adapting existing work on language-integrated query from relational to NoSQL data models and query languages.

Acknowledgements

This work was supported by ERC Consolidator Grant Skye (grant no. 682315) and an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS). The IUPHAR/BPS Guide to Pharmacology is supported by the International Union of Basic and Clinical Pharmacology (IUPHAR), and British Pharmacological Society (BPS).

References

Buneman, P., Cheney, J., Tan, W. C., and Vansummeren, S. (2008). Curated databases. In *PODS*, pages 1–12. ACM.

Buneman, P., Khanna, S., Tajima, K., and Tan, W. C. (2004). Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42.

Cheney, J., Lindley, S., and Wadler, P. (2014). Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD Conference*, pages 1027–1038. ACM.

Chlipala, A. (2015). Ur/Web: A simple model for programming the Web. In *POPL*, pages 153–165. ACM.

Cooper, E. (2009). The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, volume 5708 of *Lecture Notes in Computer Science*, pages 36–51. Springer.

Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2006). Links: Web programming without tiers. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer.

Fehrenbach, S. and Cheney, J. (2018). Language-integrated provenance. *Sci. Comput. Program.*, 155:103–145.

Fowler, S. (2019): Data for "Cross-tier web programming for curated databases: A case study". figshare. Dataset. https://doi.org/10.6084/m9.figshare.11369838.v1

Fowler, S., Harding, S. D., Sharman, J., and Cheney, J. (2020): Cross-tier web programming for curated databases: A case study. arXiv pre-print arXiv:2003.03845. https://arxiv.org/abs/2003.03845

Giorgidze, G., Grust, T., Schreiber, T., and Weijers, J. (2010). Haskell boards the Ferry - database-supported program execution for Haskell. Number 6647 in LNCS, pages 1–18. Springer-Verlag.

Grust, T., Rittinger, J., and Schreiber, T. (2010). Avalanche-safe LINQ compilation. *PVLDB*, 3(1).

Harding, S. D., Sharman, J. L., Faccenda, E., Southan, C., Pawson, A. J., Ireland, S., Gray, A. J. G., Bruce, L., Alexander, S. P. H., Anderton, S., Bryant, C., Davenport, A. P., Doerig, C., Fabbro, D., Levi-Schaffer, F., Spedding, M., Davies, J. A., and Nc-Iuphar (2018). The IUPHAR/BPS guide to PHARMACOLOGY in 2018: updates and expansion to encompass the new guide to IMMUNOPHARMACOLOGY. *Nucleic Acids Research*, 46(Database-Issue):D1091–D1106.

Harmar, A. J., Hills, R. A., Rosser, E. M., Jones, M., Buneman, O. P., Dunbar, D. R., Greenhill, S. D., Hale, V. A., Sharman, J. L., Bonner, T. I., Catterall, W. A., Davenport, A. P., Delagrange, P., Dollery, C. T., Foord, S. M., Gutman, G. A., Laudet, V., Neubig, R. R., Ohlstein, E. H., Olsen, R. W., Peters, J. A., Pin, J., Ruffolo, R. R., Searls, D. B., Wright, M. W., and Spedding, M. (2009). IUPHAR-DB: the IUPHAR database of G protein-coupled receptors and ion channels. *Nucleic Acids Research*, 37(Database-Issue):680–685.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Lindley, S. and Cheney, J. (2012). Row-based effect types for database integration. In *TLDI*, pages 91–102. ACM.

McKinney, W. (2010). Data structures for statistical computing in Python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

Meijer, E., Beckman, B., and Bierman, G. M. (2006). LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*.

Pawson, A. J., Sharman, J. L., Benson, H. E., Faccenda, E., Alexander, S. P. H.,

Buneman, O. P., Davenport, A. P., McGrath, J. C., Peters, J. A., Southan, C., Spedding, M., Yu, W., Harmar, A. J., and NC-IUPHAR (2014). The IUPHAR/BPS guide to PHARMACOLOGY: an expert-driven knowledgebase of drug targets and their ligands. *Nucleic Acids Research*, 42(Database-Issue):1098–1106.

Radanne, G., Papavasileiou, V., Vouillon, J., and Balat, V. (2016). Eliom: tierless Web programming from the ground up. In *IFL*, pages 8:1–8:12. ACM.

Serrano, M., Gallesio, E., and Loitsch, F. (2006). Hop: a language for programming the Web 2.0. In *OOPSLA Companion*, pages 975–985. ACM.

Sharman, J. L., Benson, H. E., Pawson, A. J., Lukito, V., Mpamhanga, C. P., Bombail, V., Davenport, A. P., Peters, J. A., Spedding, M., Harmar, A. J., and NC-IUPHAR (2013). IUPHAR-DB: updated database content and new features. *Nucleic Acids Research*, 41(Database-Issue):1083–1088.

Sharman, J. L., Mpamhanga, C. P., Spedding, M., Germain, P., Staels, B., Dacquet, C., Laudet, V., and Harmar, A. J. (2011). IUPHAR-DB: new receptors and tools for easy searching and visualization of pharmacological data. *Nucleic Acids Research*, 39(Database-Issue):534–538.

Southan, C., Sharman, J. L., Benson, H. E., Faccenda, E., Pawson, A. J., Alexander, S. P. H., Buneman, O. P., Davenport, A. P., McGrath, J. C., Peters, J. A., Spedding, M., Catterall, W. A., Fabbro, D., Davies, J. A., and NC-IUPHAR (2016). The IUPHAR/BPS guide to PHARMACOLOGY in 2016: towards curated quantitative interactions between 1300 protein targets and 6000 ligands. *Nucleic Acids Research*, 44(Database-Issue):1054–1068.

Syme, D. (2006). Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML*.

Trinder, P. W. (1991). Comprehensions, a query notation for DBPLs. In *DBPL*, pages 55–68. Morgan Kaufmann.

Wong, L. (2000). Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56.