PacTk: The Publishing Accelerator Toolkit

Digital Publishing Tools

for Indexing, Text Mining, and Dataset Curation

I Overview

The rise of a digital publishing ecosystem has significantly expanded the possibilities for academic/scientific publishing, both in terms of the range of content provided (for instance, augmenting text documents with multimedia presentations) and the breadth of reader experience in the context of text documents themselves (for instance, enhanced indexes, searching, and multi-document corpora). The **PacTk** toolkit provides software components and code libraries helping authors, researchers, and publishers to maximally leverage these new digital capabilities.

Use Case 1: Building Indexes to Enhance Search Capabilities

Ordinary text search simply tries to match one or more words within the exact text of a document. Curated indexes, by contrast, identify correlations between indexed terms and pages/paragraphs within the main text, even if the specific language in those paragraphs employs variant word-forms, or is conceptually relevant to the search term without explicitly mentioning it. When authors or editors take the time to compile an index, it is wasteful to ignore those efforts when users seek to perform text searches while reading the main document.

To effectuate a more precise and conceptual search functionality, **PacTk** supports enhanced capabilities that incorporate index files when available. In this context "indexes" are not only human-readable pages at the back of a book, but also structured metadata that can be embedded within **PDF** files and deserialized by conformant **PDF** viewers (we assume human-readable indexes are still published; these may be automatically generated from such metadata). When someone reading a text document searches for a term similar to one from the index — or selects an index heading from a list derived via index metadata — the search algorithm can locate matches via this metadata instead of or in addition to performing a brute-force search within **PDF** character streams. Special-purpose **GUI** windows can be shown that enable users to fine-tune the search keywords and options (see Figure 1 for an example of how such dialog windows might appear).

PacTk similarly provides tools for compiling indexes, serving as a guide to help authors/editors in the process. In addition to listing associations between index terms (also called "headings") and page numbers or paragraph codes (also called "locators"), index metadata should model entry/subentry relations, "*see also*" cross-references, and "redirection" ("*See*") cross-references. Dedicated **GUI** windows are necessary to allow authors/editors to examine index metadata systematically, by navigating through index entries in multiple ways (by sequential order of creation, alphabetical sequence, or following entry/subentry and cross-reference connections) and juxtaposing index-entry windows alongside the corresponding **PDF** page displays (see Figure 2 for one version of a single-entry dialog window).

Use Case 2: PDF Search for Creating and Updating Indexes

While curating an index, authors will often start by generating a basic list of terms relevant to the topic of the publication. Given this list, they will then search for instances of those headings in the document text. Although locators may often correspond to paragraphs without an exact match, a raw-text search can be a useful means to establish a provisional collection of locators that can be extended afterward. A -enabled **PDF** viewer lets users add words and phrases to a provisional heading-list — or to the index index, once initially constructed — by selecting a character-range in the **PDF** document. Alternatively, a privisional list might be built from data files, plain text files, or from a different index (e.g., the prior edition of a book that is being republished).

When performing such initial matches, it is helpful to vary the order or precision of search terms. For example, a person's name in an index will typically be listed and alphabetized via last name first, but will usually be matched in the text in the opposite order, or last name alone (e.g., the entry "Durkheim, Emile" should be searched as "Emile Durkheim" or just "Durkheim"). Likewise, instances of a single word within a phrase may be sufficient to confirm that the relevant stretch of text is indeed relevant for the entire phrase as a concept (for example, "House" in lieu of "House of Representatives"). Given these points, **PacTk** provides a front-end to **PDF** search functionality which takes a provided search term (such as an index heading) and enables users to quickly rearrange or exclude words from the keyphrase to construct the specific match presented for **PDF** search. Successful matches are highlighted both in the **PDF** page view and within secondary windows that show the raw text located within the relevant **PDF** page (see Figure 3). Presenting this contextual information allows the person compiling an index to verify that text matches are indeed conceptual hits for the relevant index entry, as opposed to false positives.

The screenshot in Figure 3 actually represents a variation on index-building, where **PacTk** was utilized to migrate an index for the second edition of a previously-published manuscript. In this case, the first-edition index provided a provisional set of headings whose locators had to be updated with paragraph id codes for the new book. We therefore employed a version of a **PacTk GUI** that had two pairs of page and raw-text views. In this variation, by matching terms against both documents, it is easy to confirm that a locator for the newer edition directly updates an index-entry from the prior book, by checking that the surrounding text is similar or identical. Once entries are updated between the two versions, then new headings (plus new page/paragraph-id locators for old terms) can be added.

Use Case 3: Customizing GUI Tools for Indexing

The **GUI**s in Figure 3 include two distinct **PDF** view windows, which might be used to review two different editions of a book whose index

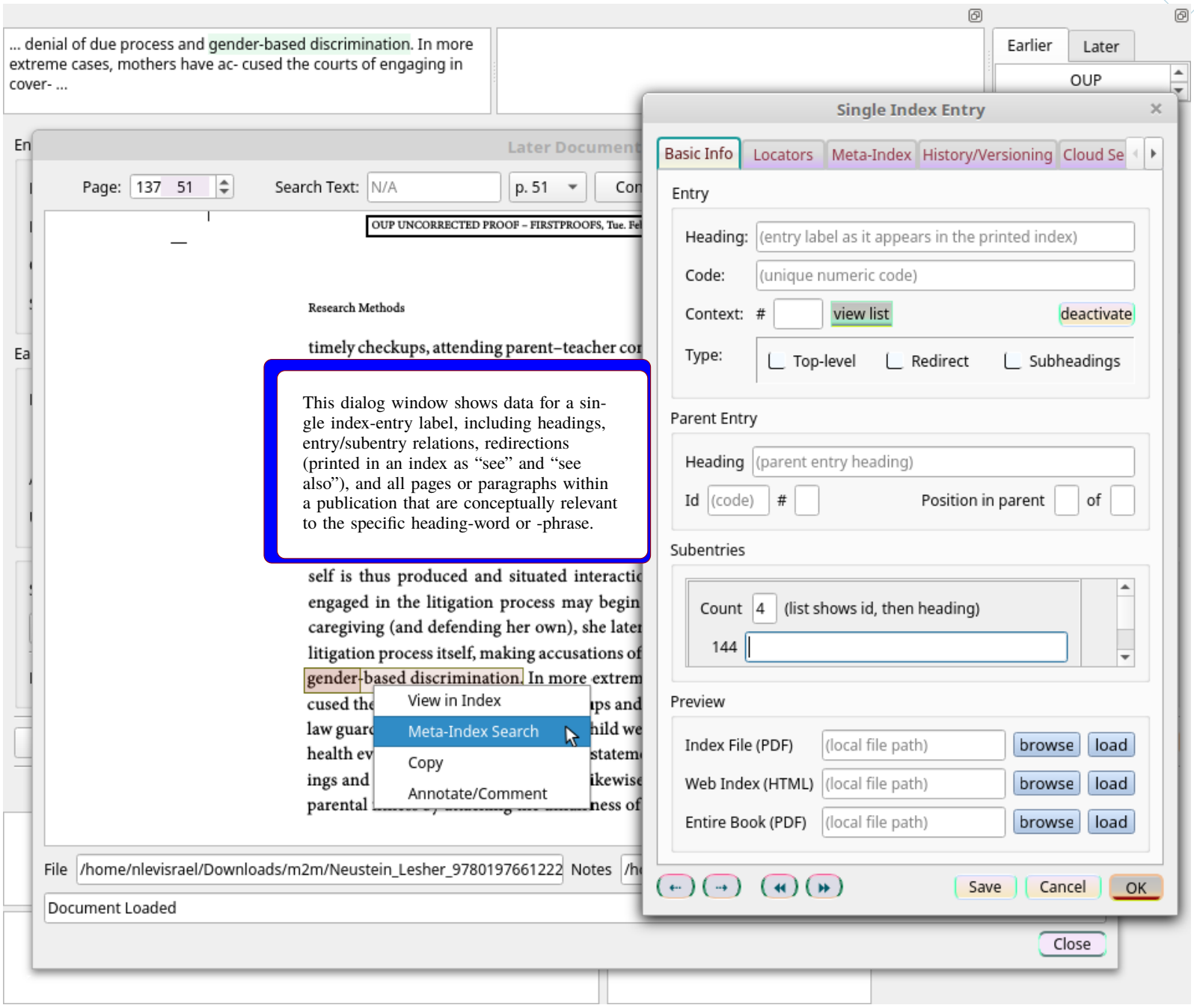


Figure 1: An index entry dialog window (and context menu for single and meta-index views)

is being updated. The software, that is, has two **PDF** view windows to show the two editions, plus a central window with a variety of entry lines, buttons, and check boxes, intended to automate the data transfer (from old index to new) as much as possible. After identifying all of the index terms from the prior edition and the corresponding page references, the software is designed to work through index entries one at a time — either in alphabetized sequence or starting from a specific entry. For each entry, users can click through the different page matches from the first edition, and the application then searches for similar terms in the second. Above-mentioned search features enable users to fine-tune the search key to be more precise if needed — for instance, inverting a proper name (Durkheim, Emile becoming Emile Durkheim), or restricting terms to the most important words (e.g., “‘Diagnostic Impression’ chart” might be reduced to “Diagnostic Impression”). Making these adjustments by clicking buttons is quicker and easier for the user than typing new search terms manually.

Continuing the example of migrating from a first to a second edition, once search terms are matched against the new document, the code shows matches that are highlighted in both editions; and also shows the plain text of the corresponding files in separate windows. It is worth mentioning that the highlights are constructed via native **GUI** overlays, rather than employing the internal **PDF** highlight mechanism — the former is more flexible, and allows features like semi-transparency and color variation (which prove quite useful during a session where multiple entries and/or term-variants have been processed). With these features, the user is able to quickly check whether a match replicates an entry in the first-edition index or else is a new paragraph-id locator. In either case the user will click to register the match as appropriate for the new document’s index and track the correlation between the old and new index — the software automatically identifies correct paragraph-id codes if those codes, rather than page numbers, are used to generate an index. When ready, the user can confirm that the data for the current heading is complete and proceed to the next one. The software shown in Figure 3 then generates **HTML** code for that entry, which ultimately gets folded into the compiled second-edition index (this might be in formats such as Word, **PDF**, **XML**, etc.). Users have the option of automatically storing the entry data on a web back-end, or via **FTP**, which comes in handy if multiple people are working on an index at the same time.

Visible in Figure 2 and Figure 3 are numerous **GUI** controls through which users can work through a list of entries via buttons, check boxes, context menus, and “dock” widgets (components that will either be affixed to the main window or float as separate windows, as desired). Some of these controls help to quickly process individual entries; others affect the search process by filtering the target search

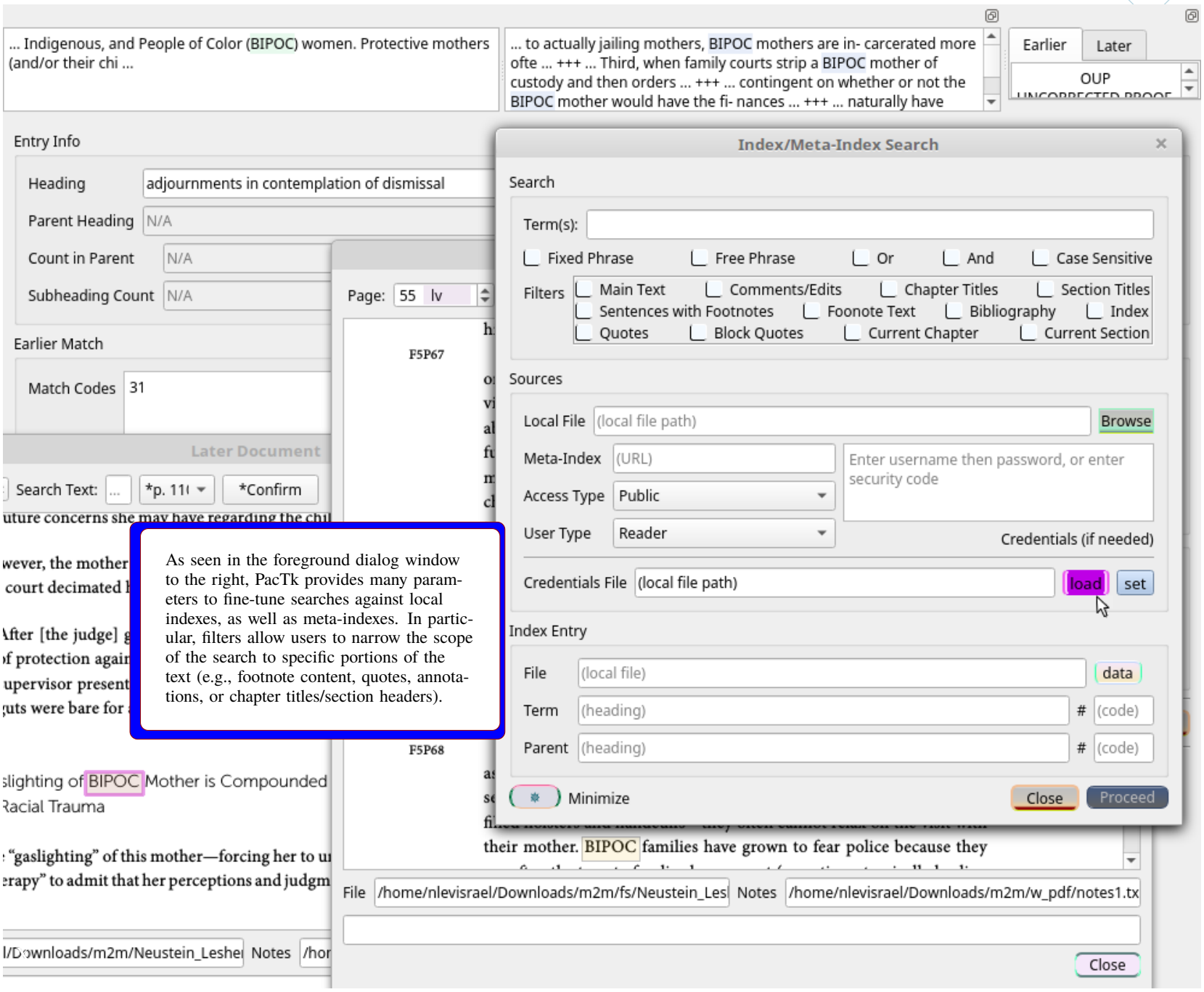


Figure 2: Dialog window for local and meta-index searches

area (e.g., restricting to certain pages — such as a specific chapter — or just to footnotes, etc.). These various GUI features enable the user to process a single entry much more quickly that manual text-editing. Given a project for a second edition, for instance, often it is possible to confirm a match between the earlier and later document, and update the later index accordingly (with new locators), in just a few seconds.

When in the process of curating an index, it may be important to examine PDF representations in several different variations. In particular, looking directly at document content as internally represented can clarify anomalies such as false negatives (caused by hyphenation, ligatures, nonstandard fonts, etc.). In other cases, raw text might provide more convenient visualization for the text around a match than is evident in the fully-rendered PDF view. The components show in Figure 3 present document content in three contexts: the PDF view itself; the unadulterated raw text for a given page; and an edited version of the raw text simplified so as to emphasize context around a match (in each case the match itself, if present, is highlighted with an overlay). The simplified raw-text view presents a window of approximately 30 characters on either side of the matched characters (users can reconfigure the exact number) allowing users to quickly check which words appear to the left and right of a given match. Such functionality helps one to double-check that a raw match is conceptually appropriate, or to determine the proper subentry within which to include the current locator ("gender bias", for example, could be divided into gender bias *in courts*, *in hiring*, etc. — glancing at whichever words surround the matched "gender bias" string might facilitate the user deciding which subheading is most appropriate).

Whether building an index from scratch or updating an earlier publication, these GUI capabilities streamline the indexing process. Rather than typing index entries directly into an index file — manually adjusting details like italics, subentry lists, roman vs. arabic numerals, and so forth — the user could rely on mouse clicks and context menus (with minimal actual typing) to compile a data structure holding all of the index entries. A human-readable index is then generated from that structured data, or authors could submit the index as a data file instead, which could be incorporated into Meta-Indexes discussed below.

The PacTk GUI tools are designed to flexibly interoperate, with few external code dependencies and an emphasis on autonomous, modular design. That is, individual window components and/or algorithm libraries may be combined together and fine-tuned for specific book

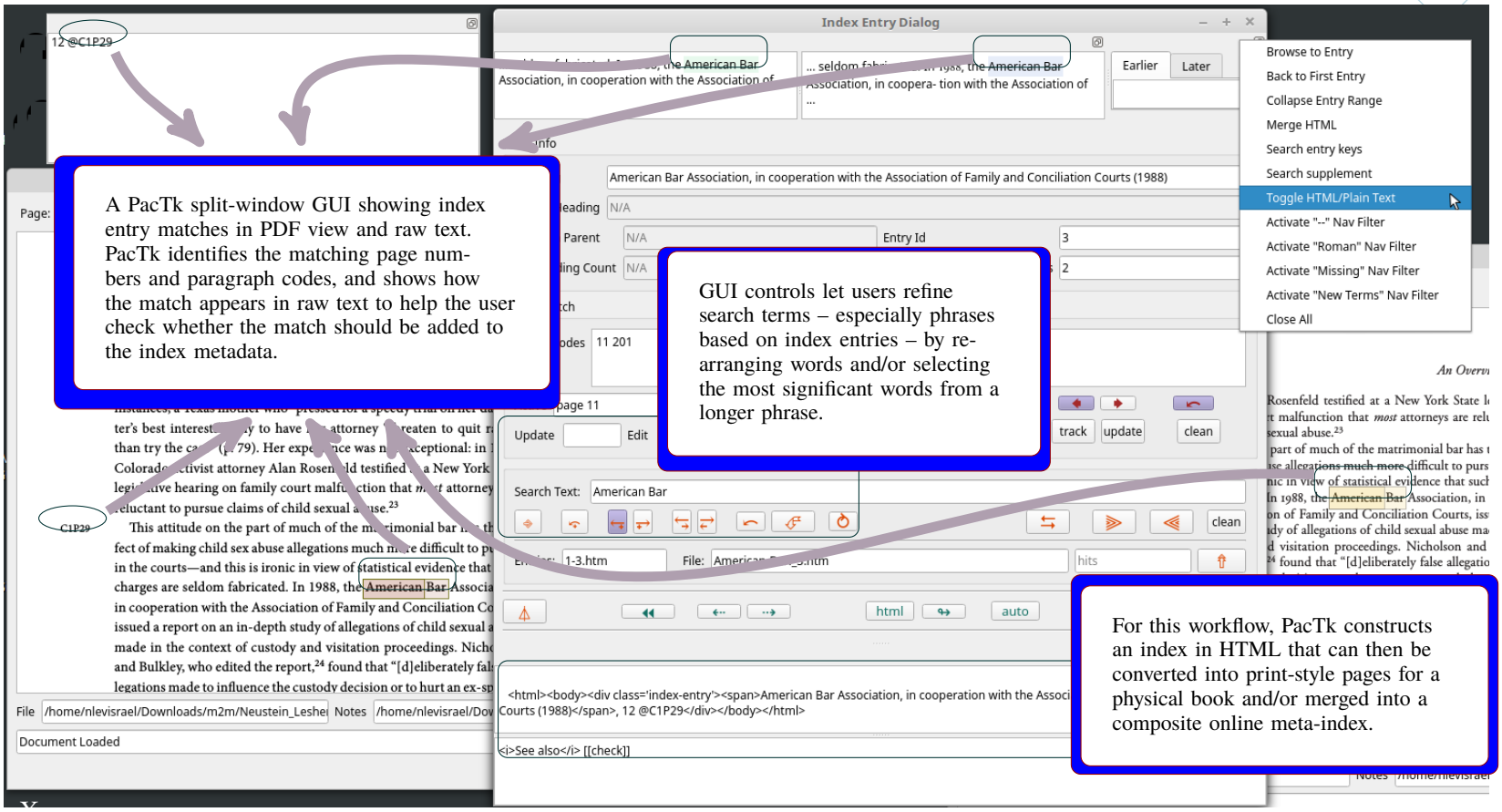


Figure 3: GUI functionality for compiling and curating indexes

projects. In the example discussed here, our requirements involved migrating data from an older edition of a book to a new index. Other publications might have different needs — for example, syncing an index to a published data set; or compiling an aggregate index from a volume with individual chapter authors. Modular, native-compiled **GUI** controls permit new secondary windows to be implemented so as to support convenient user interaction, or automation, to complete tasks requisite for specific publication's workflow (see the summary of **BdQS** below for more info on dynamic **GUI** customization).

A display with two **PDF** views may also be helpful for working on a single document. One window can show the text which an author or editor is currently addressing, while a second window could show content from a different chapter or section of the same book/article. This would operationalize side-by-side comparison, such as to check that earlier language is not repeated, or to ensure that a particular bibliographic reference is cited in at least one paragraph where the concomitant publication (or its topic) is discussed.

II Meta-Indexing, Cloud Services, and Text Mining

Use Case 4: Annotated PDFs as SVG Files

PDF annotation capabilities (which include editing features and comment boxes) are unfortunately very limited. Many **PDF** viewers cannot show annotations at all; those that do, often present annotations in small, hard-to-read box areas. Annotation support is not uniform: when a file is shared, there is no guarantees that recipients will be able to see annotations at all, or that they are presented in a usable manner. One solution to this problem is to convert **PDF** documents to a series of **SVG** files, that can be browsed as individual pages. **SVG** is a general-purpose format for creating **2D** graphics, and **SVG** versions of **PDF** pages can be annotated with overlays that employ the full range of **SVG** capabilities, including all mathematically-describable paths and shapes; semi-transparent colors; scalable text; and mouse/keyboard user interactions. For example, some actions performed in the process of editing page proofs may be more user-friendly (in terms of **GUI** interactions and User Experience) when the document is read through an **SVG** viewer rather than a **PDF** viewer.

Figure 4 shows a feature wherein annotated **PDF**s are converted to **SVG** files so as to better visualize the annotations (which in ordinary **PDF** viewers tend to be reduced to poorly-formatted comment boxes). Here the comment-text is enlarged and color overlays clearly identify which sections of text correspond to specific annotations. Figure 5 reflects special programming for a project where annotations were grouped into different categories, modeling linguistic/discursive properties of the **PDF**-rendered content; a master index was then compiled associating specific pages to the annotation categories, with hyperref links to access the individual pages as desired.

One benefit of **PDF**-to-**SVG** conversion is that **SVG** files thereby become independent resources that can be individually hosted online. Most web browsers have **PDF** plugins, so that **PDF** files can be viewed without being downloaded, but there are no guarantees that users will also be able to see **PDF** annotations in the browser. Moreover, there is no way to construct a hyperref link to a single page, section, or chapter; only to the **PDF** as a whole. With **SVG** files, on the other hand, each page will potentially have its own web **URL**, and links to single pages may be embedded in web resources (and also other **PDF** documents). Such web links can be useful for collaborative editing as well as referencing pages in files already published.

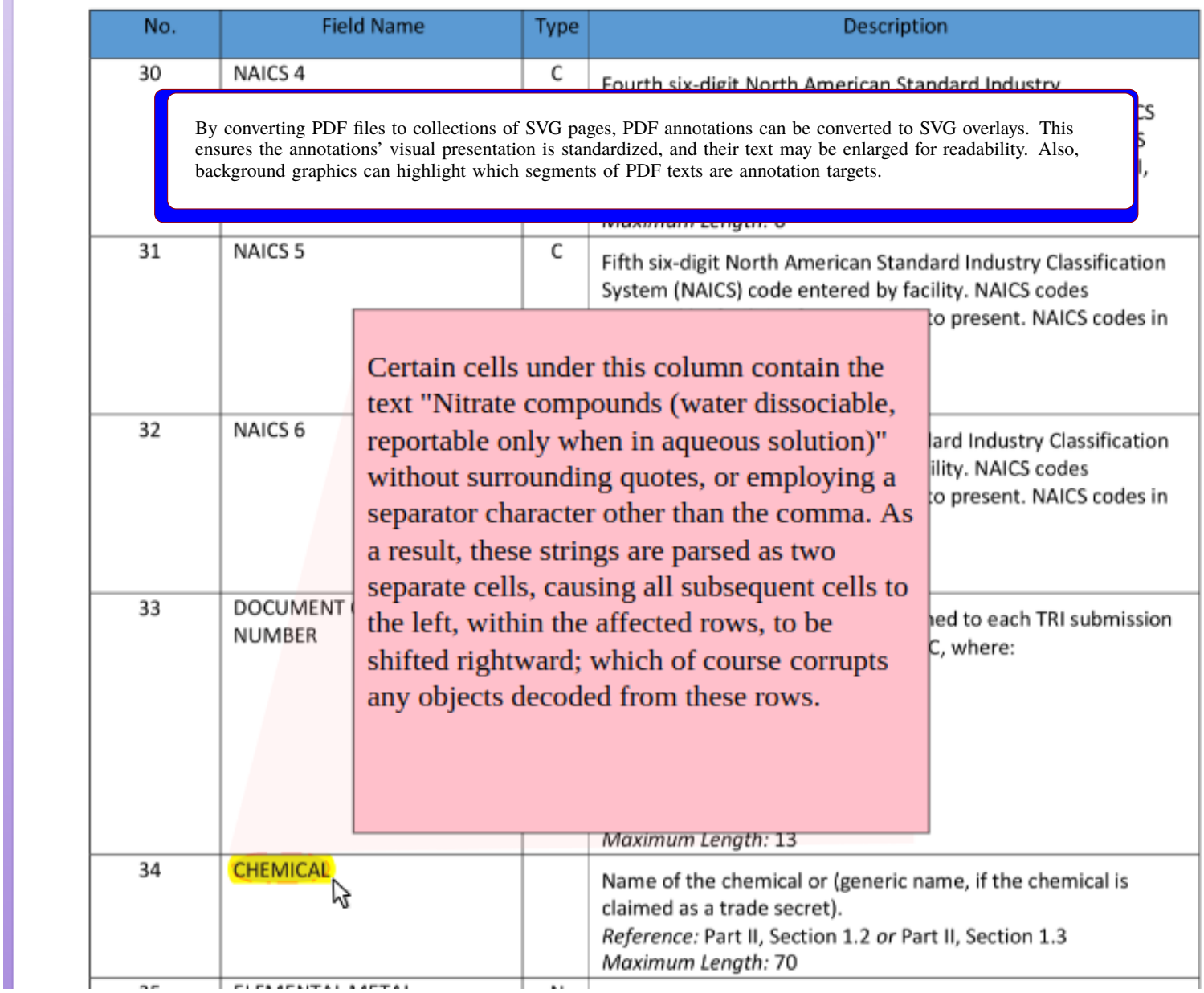


Figure 4: Mapping PDF pages to SVG views for improved annotation support

PacTk additionally supports technology for embedding web viewers in standalone native (i.e., desktop-style, rather than web-based) applications. In this manner, authors/editors can preview the appearance and functionality of **SVG** annotations before they are disseminated online. Special-purpose native/JavaScript interop allows the host application to respond to a wider range of user actions than browser-based code in isolation, so **PacTk** tools can be used to modify and style annotations, edits, and comment boxes while working directly with **SVG** files. By default, **PacTk** implements web/native interop via **QWebChannel**, one technology within the **Qt** application-development libraries. The combination of **QWebChannel**, **PDF**, and **SVG** is a powerful but rarely used programming paradigm. In addition to document publishing, browser-embedding can be a convenient tool for curating data sets, particularly in contexts such as **GIS** that often rely on a web interface for data acquisition; and for programming dataset-backed simulations (especially when user input/gestures are involved).

Use Case 5: Curating Meta-Indexes

The benefits of indexes for enhancing local text search may certainly be extended to more general web/database searches.

Because **PacTk** represents indexes via metadata, it is straightforward to merge indexes from multiple documents into an aggregate database, also known as a "meta-index." Meta-indexes may be compiled for individual book series; academic journals; or even large-scale document corpora. Such meta-indexes could be made available to users in one of two ways. On the one hand, users might enter search terms and query against an encompassing bibliographic database, without necessarily starting from one specific book or publication. Alternatively, a user might be reading a specific digital publication and request a local search that then expands to include a

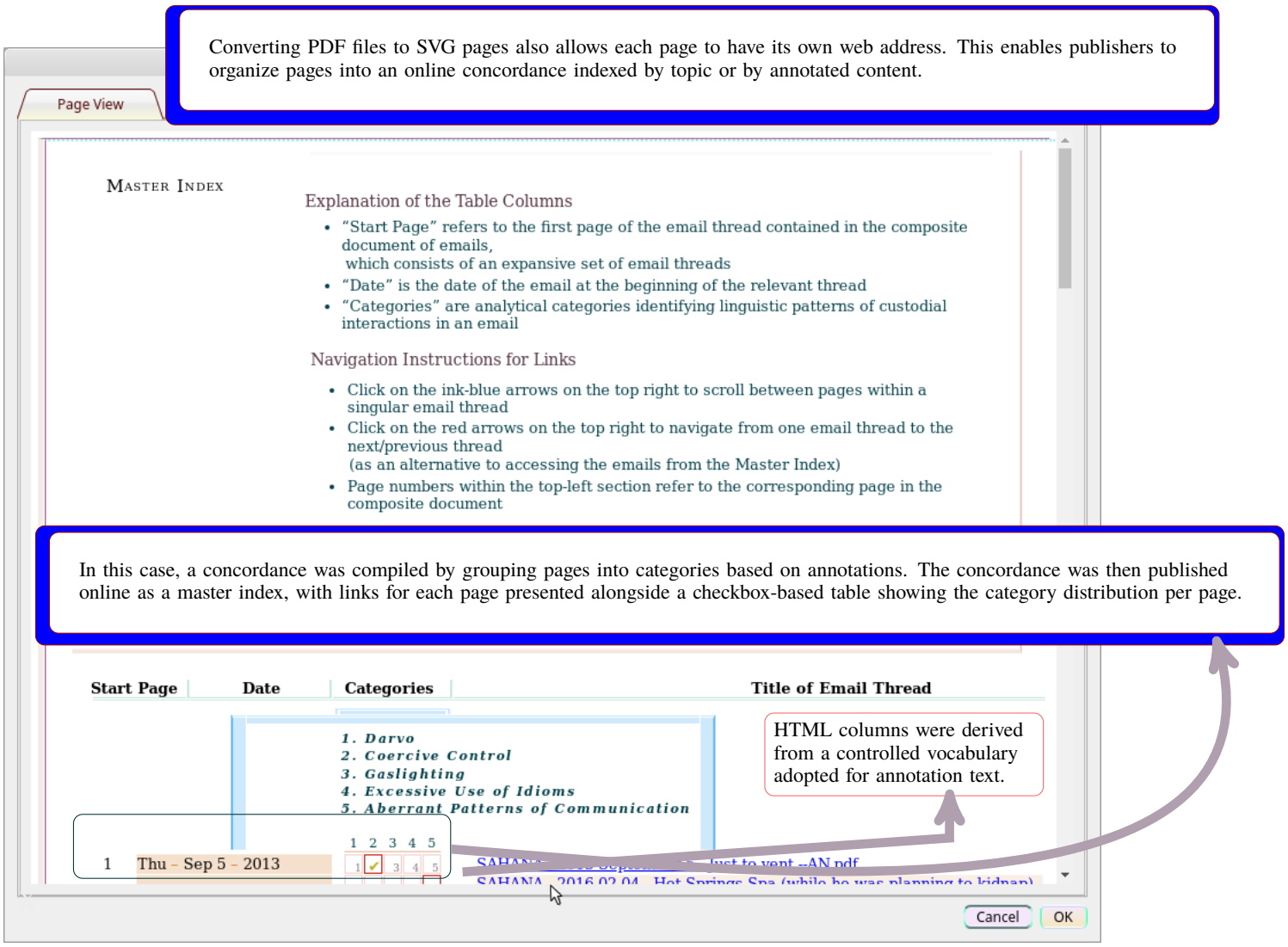


Figure 5: A category-based web-accessible index table for SVG pages

meta-index. That is, users could indicate an interest in match-results not only for the book they are currently reading but also for pages in other volumes stored alongside it in a book series or corpus archive.

Hosting and curating a meta-index is facilitated by **PacTk**'s structured metadata format. Because all books in a series/archive will share a metadata profile, new indexes can be merged into a preexisting database as soon as they are finalized. **PacTk** provides an **API** protocol for registering a new index as well as querying for results in a hosted meta-index. **PacTk** also offers native **API** tools such that **PDF** viewers can query against a database when expanding local to meta-index searches. Meta-index **API**s could likewise be used for search front-ends exposed to users via web pages for specific journals, book series, abstracts aggregators, and analogous publishers' web content.

Via conventional "forward indexing," new documents become merged into composite search "indexes" as soon as they are published. The term "index" in this content refers to word-by-word counts on the principle that the number of occurrences of some word or phrase, in a given document, provides a rough measure of the document's relevance to that word. More refined algorithms can modulate relevance scores by considering conceptual associations, and, in general, the phenomenon of multiple expressions designating the same (or closely related) referents (e.g., "the President" and "the White House"). Actual book indexes, of course, represent a more detailed model of such conceptual relationships because authors/editors explicitly select a set of terms that are especially significant for the material's theme/topic; and also consciously choose which conceptual links to explicate (via "see also" clauses, in particular). Meta-indexes can, of course, be layered on top of automatically-indexed search engines.

Whether or not meta-indexes work against a data set that includes a full search-engine database, the information content of a typical meta-index will be sufficiently large that query processing functionality would have to be hosted on a publically-accessible server. In many use-cases, a web- or cloud-based meta-index would be accessed from a single **PDF** file — we assume the reader initiates a meta-index search on the basis of material they are reading in the local file — in which case remote queries, through an **API**, could be sent and received by a conformant **PDF** viewer. In such contexts, users would not see the web/cloud content directly. However, publishers have good reason to also wrap meta-index access in a modern-style web site, for the alternative use-case wherein a researcher does not start from a particular book but rather visits a web portal to find materials on a topic that interests them. Figure 6 shows a hypothetical web front-end along with a screenshot with an example of two books from the same series, that could be linked by behind-the-scenes searches.

Existing bibliographic aggregators, such as Dimensions, pair **API** protocols with customized query languages for bibliographic databases. To support such technology, **PacTk** includes code for a Bibliographic-database Query Infrastructure (**BdQS**) for exchanging structured data with **API** hosts. Instead of **REST**-encoded **URL**s, more complex query prompts (and responses) can be encoded via **BdQS** for **POST** requests (native-compiled code can leverage easy-to-use request builder objects). The **BdQS** framework actually has several use cases within **PacTk** alongside **API** query factories; other use cases related to **GUI** programming were mentioned earlier.

The **PacTk** code base additionally includes prototype Cloud-Native service implementations for hosting meta-index databases. These

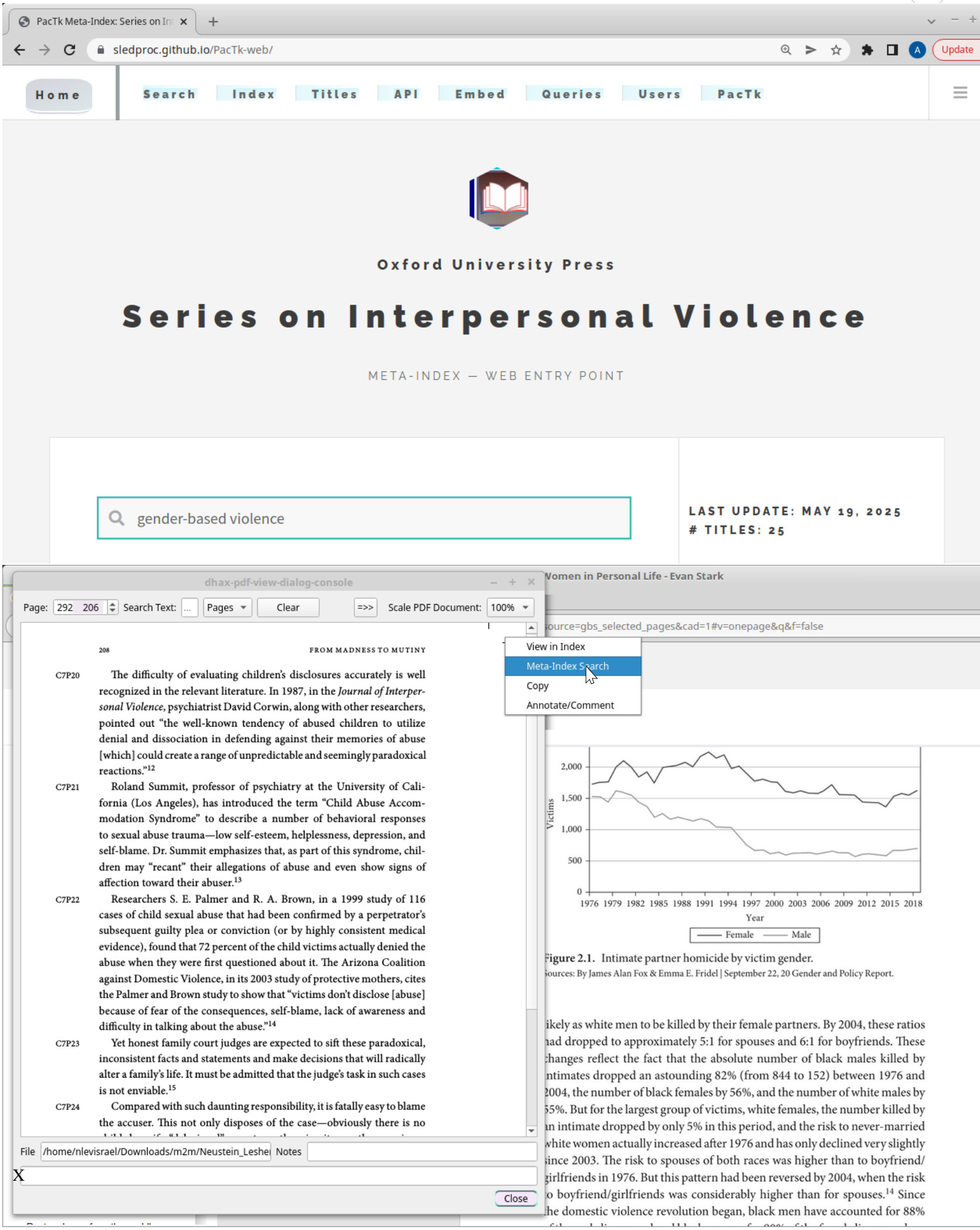


Figure 6: Meta-Index Search through a Web Portal

cloud applications — which are able to be developed and tested as self-contained desktop components before getting deployed online — encapsulate many details of meta-index management, database admin, and **API** support, with significant code reuse between native/desktop clients and **API**/cloud server endpoints.

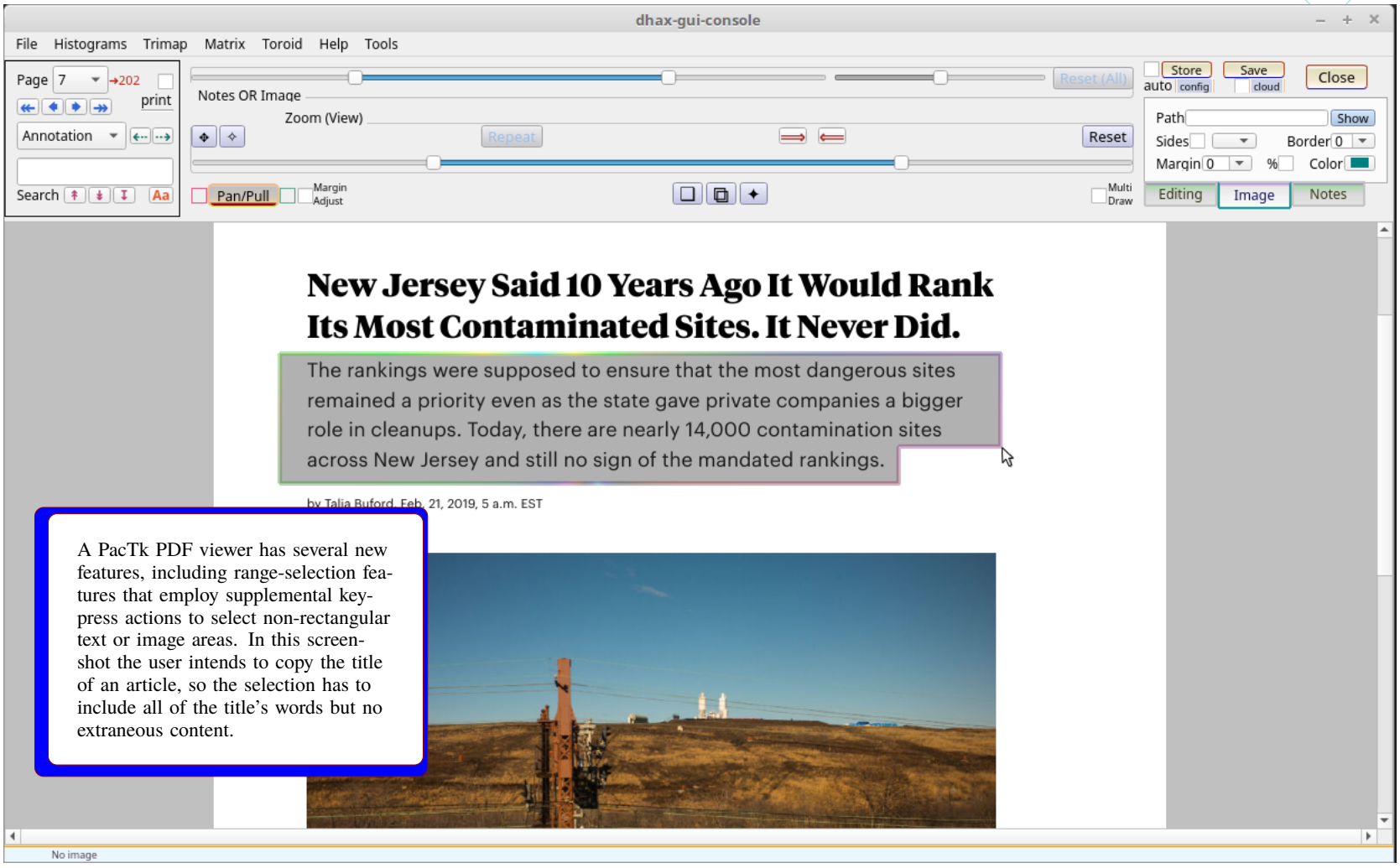


Figure 7: Flexible range-select features

Use Case 6: Machine-Readable Text Encoding

Because launching **PDF** text search for each **API** query would be very inefficient, cloud-based meta-indexes must compile machine-readable representations of text document contents, so as to search many books at once. This process also corrects for the inherent limitations of **PDF** text searches, which can be counter-productively stymied by such basic typographic conventions as ligatures, end-of-line hyphenation, apostrophes, footnote marks, and many other familiar presentational details.

With full metadata in place, **PacTk** indexes construct in-document locators that combine two different target origins. On the one hand, text spans are declared via machine-readable encoding for text mining, rather than (or in addition to) the internal **PDF** text representation, which is often an incomplete representation of the document text as understood by human authors and readers. Concordantly, on the other hand, the visible outlines of text-segments that match index terms are stored via **PDF** page coordinates, which may be traced by **SVG** or other graphics overlays. In both cases, the internal **PDF** searching and highlighting functionality is supplanted by a two-facet text and graphics representation. Ordinary **PDF** search may be employed initially to "bootstrap" an index, but once finalized the document metadata would be consulted as a basis for searches both locally and via **API**s.

Rigorous text encoding can also improve User Experience working locally with **PDF** files. For example, users could copy the text of a full sentence where a match occurs — not just the search term itself — with a single button-click. Figure 7 shows a range-select option that syncs mouse drags with key-presses to select a multi-line area that extends to the margins in the middle lines (a useful alternative to ordinary **PDF** ranges that can only be a simple rectangular box, thereby either cutting important words from left or right or including extraneous words around the range's start and end, which makes it difficult to select a full sentence or other meaningful unit of text). Advanced **GUI** programming in conjunction with machine-readable text encoding permits users to accomplish their goals in find, copy, and remote-search functionality with less hassle than traditional **PDF** viewers (which rely on flawed internal text representation).

Use Case 7: Rigorous Text Mining

The limitations of **PDF** text encoding serve as a hindrance to advanced text-mining algorithms. These lacunae became evident during the Covid pandemic, when scientists sought to gather intelligence from a broad pre-pandemic literature about Coronaviruses and the first (2002-2004) **SARS** outbreak, as well as new publications which quickly emerged summarizing research into **SARS-CoV-2**. The Allen Institute for **AI** (**AI2**), for example, compiled a large **CORD-19** corpus using a project-specific **JSON** encoding of all document texts. However, **AI2** pointed out that transcription errors were limiting the effectiveness of text-mining algorithms implemented against the corpus (an extended analysis of **CORD-19** and other Covid-related data integration projects can be found in our book *Innovative Data Integration and Conceptual Space Modeling for COVID, Cancer, and Cardiac Care*, Elsevier, 2022). We identified problems such as inconsistent

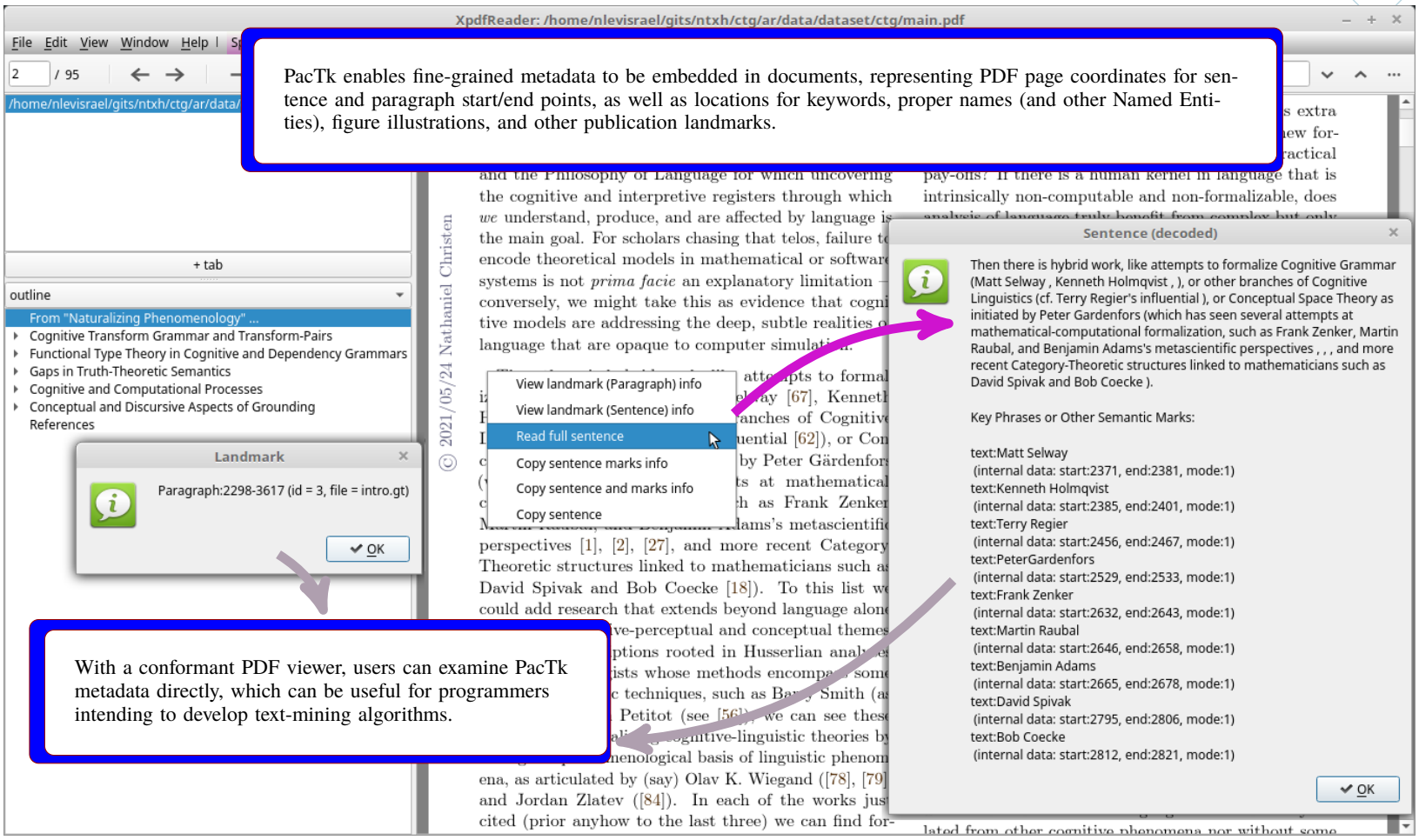


Figure 8: Enhanced GUI capabilities enabled by machine-readable text encoding

representation of medical and chemical terms/formulas — such that correlations between publications were not properly identified — and ambiguities in modeling the structural breakdown of natural-language text into paragraphs and sentences. **AI2** argued that new machine-readable text formats were necessary, rather than relying on **PDF** representations, and issued a "call for action" encouraging publishers and scientists to develop modernized text-encoding and metadata standards. We thereby developed a novel document-preparation system that generates structured text representations alongside conventional **PDF** files, which we used for composing the data-integration book mentioned above (and subsequent publications).

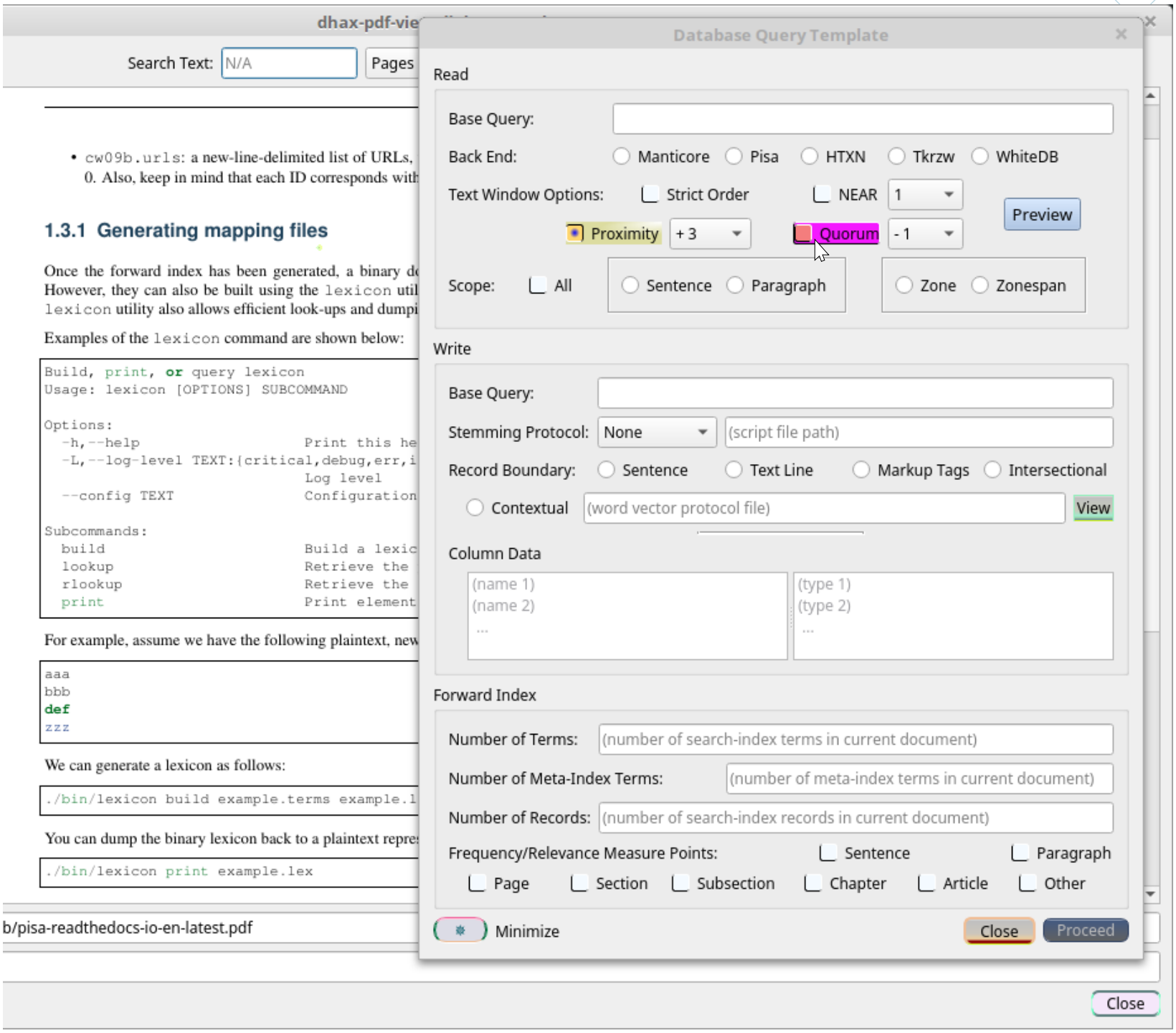
PacTk-enabled documents can utilize similar representations to expose full text content, alongside index metadata, to bibliographic databases. Rigorous encoding permits databases to systematically address ambiguities related to match contexts and query results. Suppose an **API** request models a search for a keyphrase and requests all sentences where the phrase appears, among a multi-document collection. In order to fulfill the request, a query engine would need to clarify the proper representation of a "sentence," as well as adjust for anomalies such as intra-word hyphenation. For example, requests for complete sentences may or may not specify that the content of footnotes or block quotes inside the sentence should be included. Or, the preliminary language preceding an enumerated or bulleted list might be considered a single sentence (even if it ends with a colon rather than perion), but in some contexts that preamble together with the overall list serve as an aggregate sentence. Apart from intra-document queries, such granular discursive constructs can be addressed by plugins to search-engine tools that could be employed as meta-index back-ends (see Figure 9).

In short, rhetorical features such as footnotes, quotes, figure illustrations, enumerations, and itemized lists all represent discursive structures that complicate the straightforward breakdown of text into sequences of paragraphs and sentences. Enhanced search, in particular, should allow queries to filter search targets into footnote text, section headings, chapter titles, quotes (inline and/or block), content within figures/diagrams, and annotations/comments, as well as document text in general. Query protocols should therefore recognize a plurality of discursive structures both to establish search parameters and to clarify the desired scope for contextual data (the screenshot in Figure 8 presents a visual example). These details can be exposed to **API**s via **PacTk**'s **BdQS** framework.

III Data Publishing

Use Case 8: Deserializing Data Files

In contemporary publishing, text documents are often paired with scientific data sets in accord with **FAIR**sharing research-transparency principles ("**FAIR**" stands for Findable, Accessible, Interoperable, Reusable). For someone intending to examine published data sets, raw data files are of limited value. Sometimes data sets can be loaded into proprietary software used by research labs, but many readers will not necessarily have access to such software (thereby failing the Accessibility and Reusability criteria). Consequently, individual scientific disciplines are often associated with domain-specific data publishing formats that are paired with dedicated deserialization tools



X

Figure 9: Configuring Default Back-End Queries

(i.e., computer code which reads raw data files into live memory, without relying on special-purpose software). Such code libraries are often maintained by individual institutions or organizations, such as academic departments, on behalf of a broad research community. Examples include **FASTQ** and **VCF** for genomics; **FITS** and **NETCDF** in astronomy and earth sciences; **PDB** and **MOL** in biochemistry; **IFC** in engineering; **SEG-Y** and **BAG** for geophysics and oceanography; **CONLL-U** in computational linguistics; **DICOM** and **OME-TIFF** for bioimaging; and **GIS**-indexed attribute layers and shapefiles for geospatial data sets. Among many others: this list is not at all exhaustive.

Some domain-specific file formats are derived from general-purpose formats such as **CSV**, **XML**, and **JSON**. Others use idiosyncratic text and/or binary serialization protocols, so that custom parsers are necessary so as to consume byte or character streams from raw data files. Even when the actual encoding is performed via a generic standard like **XML**, dedicated deserialization code will still be needed — generic parsers can convert a given **XML** document to a Document Object Model (**DOM**) infoset, but algorithms will have to navigate the document hierarchy and extract text strings from element nodes, which become the input for initializing object data fields.

In principle, supplying code libraries for parsing raw text files and/or traversing infosets embodies **FAIR** standards by ensuring that a typical reader who accesses published books or articles can likewise obtain usable research data. In practice, however, code libraries can lag behind the evolution of data standards, and **FAIR** access may be hindered by programming/technological minutiae: versioning conflicts, programming language barriers, operating-system conflicts, and similar obstacles. It is difficult to consistently maintain code libraries for a variety of different operating systems, coding languages, computational environments, and non-backwards-compatible versions of external dependencies, especially when the data format targeted by a library itself goes through multiple changes.

PacTk seeks to address these concerns by engineering the **BdQS** framework to support data deserialization as well as bibliographic database queries. That is, deserialization algorithms can be expressed in **BdQS** and compiled to a custom Intermediate Representation/bytecode stack. **BdQS** is implemented in largely self-contained native code than can be dropped as source files into any **C++** project (or compiled into dynamic libraries invoked by scripting languages). As a result, **BdQS** can be ported to many diverse environments and

embedded into many scientific applications, thereby ensuring that up-to-date deserialization capabilities are accessible for any file type for which a **BdQS**-based parser has been published. The entire **BdQS** stack could moreover be released as part of the source code contained in a published data set, so that deserialization functionality becomes part of the data set's overall package.

In general, **BdQS** deserializers can either parse character-streams directly — by registering callback functions responding to matches in Regular-Expression-like pattern language — or else post-process **XML**, **JSON**, **CSV**, and similar formats by declaring handlers for traversal events. **BdQS** provides a flexible suite of overload and redirection protocols to marshal post-parse navigation algorithms to **C++** methods that can granularly model preconditions for reading specific data fields (e.g., required nonempty, numeric range-validation, controlled vocabularies for string columns, etc.).

Use Case 9: Data-Set Visualization and Nanopublications

Embedding deserialization code is a good illustration of **FAIR**sharing priorities: ideally, all the resources necessary to (re)use a data set should be provided as part of the data package itself, rather than expecting users to download separate dependencies from external sources. Similar goals inform the design of data visualization features specific to individual data sets.

Some data sets might be loaded into readily-available software, but in many cases code for visualizing and studying raw data files should be provided as part of the data set itself. In these situations, the package can include source code files that, once compiled, produce an executable program which renders the concomitant data set in a visual, interactive manner. This may involve the use of charts or plots to show statistical trends; as well as structured **GUI** controls, or separate windows, to display the attributes of individual data objects. For the purpose of discussion, we refer to such self-contained resources specific to an individual data set as "Data Set Applications" (**DSA**s).

When a document (book or article) is explicitly paired with a published data set, information about the data set should be included in index/search metadata. This would allow keyword searches to be extended to raw data fields, as well as data-structure parameters (such as column labels, source-code annotations, procedure names, etc.). Moreover, sections of text can be isolated as cross-references for data-structure parameters as well as individual objects/records. For example, a specific column or field in tabular data structures can be cross-referenced with the paragraph in article text where that field is discussed (its observation/acquisition methodology, units of measurement, statistical distribution, etc.).

PacTk, in particular, envisions **PDF** documents links to supplemental resources that could take the form of graphics displays (showing things like statistical summaries) or dedicated **GUI** windows (showing, for example, structured data in tabular or hierarchical fashion). This model of supplemental presentations comports with the emerging idea of "nanopublications," which are independently citeable works that are typically smaller in scale than, but may be paired with, a research article. Depending on one's preferred definition, an interactive statistical diagram, or a **GUI** window that can browse through discrete objects/records in a data set, may be considered a nanopublication or an integrated collection of multiple nanopublications. In either case, **PacTk** allows the nanopublictions associated with a research document to be accessed interoperably from **PDF** (or **SVG**) files via the same conventions as indexing and text searches; and allows nanopublications to be integrated with meta-index databases for holistic corpora.

Use Case 10: Microcitations and Multimedia Cross-Referencing

Cross-referencing between research papers and data-set nanopublications is a special case of multi-media cross-referencing, where annotations defined in the context of a specific media type assert semantic or thematic relations to content with a different modality. Concrete examples of multimedia cross-referencing include geotagging video frames; linking **CAD** digital twins with product specs; annotating Regions of Interest within image series with coded labels associated with published raw data; embedding tags in 360° displays whose content includes **URL** strings that encode identifiers for data set objects; developing color- and/or shape-coded attribute layers for **GIS** overlays (so users can browse between digital maps and structured object displays); and constructing numeric tuples from **3D** graphics scenes such that particular camera orientation/location and zoom levels can be recorded as a unit, allowing the relevant **3D** content to be repeatedly visualized from a specific perspective (in other words, links can be embedded in **PDF** files, web pages, software, or any other context so that users can reconstruct a specific **3D** state, as constituted by camera angle/location and zoom settings).

The challenge when implementing multimedia cross-references is to ensure that viewers for two divergent media types can properly interoperate. With correctly interpreted cross-references, application users should be able to navigate from a window dedicated to one media/file to a window rendering a different format, and correctly isolate the portion of a file (not just the overall file) implicate in a cross-reference. For example, geotagging video frames should enable users to pause a video at a particular location and then, via the geotag, switch to a map display centered on the geospatial coordinates visible in the video (consider, e.g., a video documenting progress in an environmental remediation site). Conversely, **GIS** attribute layers would represent locations for which video content is available, and allow the users to watch the relevant videos starting from the appropriate time-point. Such interoperability is only possible if the components and/or applications responsible for rendering the two content-types share a common annotation and message-passing protocol.

The **PacTk** code base includes prototype viewers for some widely-used media types, including video (and audio) players, digital maps, 360°/panoramic photography, **3D** triangular-mesh displays, image series, and parse-representation formats associated with text mining and computational linguistics. These prototypes illustrate how cross-referencing protocols may be adopted, and can be extended to provide domain-specific plugins for **PDF** viewers or bundled into data-set application code. More generally, **PacTk** defines a cross-referencing protocol that ensures proper interoperability between multimedia displays and text/bibliographic databases. This protocol can then be

adapted for other media types, including special-purpose formats specific to individual scientific disciplines (such as **3D** displays for molecular visualization, in cheminformatics; Digital Model Repositories for oncology and other biomedical fields; radiomic biomarkers for diagnostic imaging; and audio files with waveform visualizers for spoken-language corpora).

Standardized cross-referencing protocols enable scientific applications to interoperate with assets from the full range of media types that might be published (as supplemental materials) alongside research works. Rather than relying on generic **PDF** viewers, scientists could then employ **PDF** applications specifically engineered to work with the data formats and visualization strategies specific to a given research discipline. Comments and annotations within data-set source code can serve to document the data set as a whole, but — aside from semi-formal code annotations — similar (and potentially more rigorous) details may be intrinsically expressed in code via declarations recognized by and incorporated into the compiler pipeline, and exposed through runtime reflection and queryable intermediate representations. **BdQS** in particular, more than conventional languages, encourages systematic introspection for **GUI** objects and their interrelationships, such that correlations between data-set entities and their user-interface representations may be rigorously identified.

IV Documenting Data Profiles via Code Annotations and Conventions

Publishing code alongside raw data sets facilitates accessibility and reusability. In addition, properly engineered computer code can augment data sets' explanatory and pedagogical dimension. Any information aggregate represents specific data "profiles" and acquisition modalities. Often these are informally discussed in research papers, but data sets themselves can document their observation protocols, pre-analytic transformations, statistical properties, and theoretical assumptions through annotations, type interfaces, and coding conventions.

In particular, "design by contract" programming techniques become particularly valuable in scientific-computing contexts. These techniques explicate what are otherwise merely implicit technical and theoretical assumptions. Preconditions on procedures, object attributes, and data-type constructions can specify units of measurement, valid ranges for numeric values, collections-type constraints (e.g., that two lists have the same number of elements), algorithm prerequisites, and similar expectations presupposed by the implementation of one or more function-bodies. The overall collection of contracts declared for a class interface or code library serves to illustrate theoretical paradigms germane to research from which data sets are compiled. Manifesting such contracts in computer code, not just as abstract or mathematical conventions, concretizes theoretical constraints in an environment that lends itself to exploration and visualization.

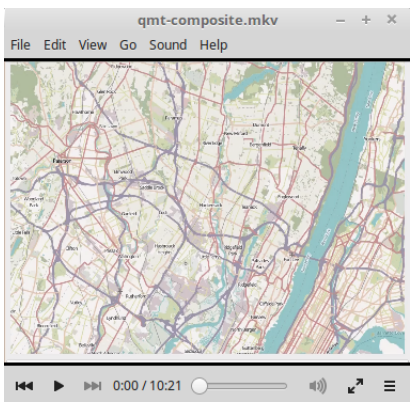
Analogously, algorithms expressed in computer code are amenable to being studied interactively, more so that quantitative formulae. By varying algorithms' sample domain, parameters, and scope — plus examining intermediate results — researchers can get an experiential grasp on quantitative constructions that might be harder to understand when presented just as cryptic equations.

Similar clarification can derive from a type system. There are many variations on a basic number-pair, for instance, differing in terms of how coordinates are labeled (x/y, row/column, width/height, etc.). Mixing incompatible pairs (vertical then horizontal and vice-versa, say) is a common source of coding errors, which may be avoided by construing each coordinate system as its own type (so a width/height pair could not be added to a row/column, for example). These type-level coordinate constraints document data-modeling paradigms for analogous reasons. A related example would be specific types for representing constrained character arrays, that are less flexible than generic strings but imperfectly encoded by straighforward numeric types (a good example is 5- or 9-digit US zip codes, which are not ordinary numbers because they need a specific digit count, and because leading zeros are significant). Within **BdQS**, granular number-pair (and number-tuple) types as well as "constrained character arrays") are built-ins with special syntactic and introspection support.

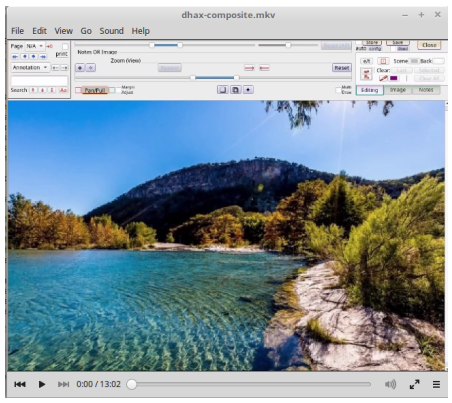
Given these considerations, **PacTk** engineers the **BdQS** environment to prioritize this pedagagical and replication-oriented dimension. **BdQS** spans a compiler and runtime infrastructure that highlights computational details which are intrinsic to data sets' scientific provenance, such as multiple-dispatch in terms of contract satisfiability and type-level guards for measurement scales and dimensional analysis. Accordingly, in addition to employing **BdQS** for bibliographic queries and deserialization, data-publishing curators may benefit from a **BdQS** scripting environment for analytic and **GUI**-integration tasks.

Video Links

For more information, please see our software demo videos, which show software components that could be integrated into dataset applications and **PDF** viewers:



GIS



360° photography

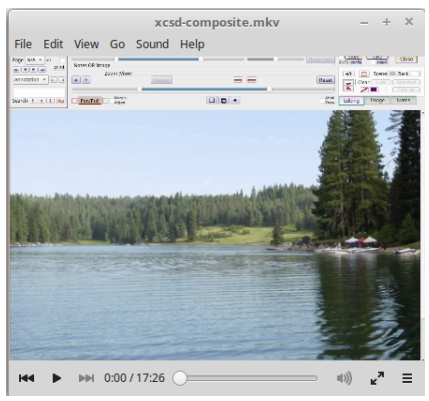


Image Processing