

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

Mani Biglari 194233IASM

Design and development of a bibliographic database architecture

Master's thesis

Supervisor: Aleksei Tepljakov
Ph.D.

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Mani Biglari 194233IASM

Bibliograafilise andmebaasi arhitektuuri kavandamine ja arendamine

magistritöö

Juhendaja: Aleksei Tepljakov
Ph.D.

Tallinn 2025

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mani Biglari

11.05.2025

Abstract

Bibliographic databases are vital for research but often struggle with data heterogeneity and limited analytical capabilities. This thesis addresses these issues by designing and implementing a modern, scalable bibliographic database system incorporating contemporary data engineering practices.

Key objectives included developing a normalized database schema, integrating the Data Build Tool (DBT) for efficient ELT pipelines, and implementing a modular three-tier architecture (PostgreSQL, Python/Flask backend, web frontend). The methodology involved schema design, data ingestion from Crossref/OpenCitations, and DBT-managed SQL transformations for data cleaning and structuring.

The resulting system demonstrates a robust architecture for bibliographic data management. The integration of DBT significantly enhances the maintainability and reliability of data transformations. This work provides a practical blueprint for applying modern ELT principles with DBT in bibliographic information systems, improving efficiency and analytical potential over traditional approaches.

This thesis is written in English and is 100 pages long, including 5 chapters, 10 figures and 3 tables.

Annotatsioon

“Bibliograafilise andmebaasi arhitektuuri kavandamine ja arendamine”

Bibliograafilised andmebaasid on teadustöö jaoks üliolulised, kuid sageli on neil probleeme andmete heterogeensuse ja piiratud analüütiliste võimalustega. Käesolev lõputöö käsitleb neid probleeme, kavandades ja implementeerides kaasaegse, skaleeritava bibliograafilise andmebaasisüsteemi, mis rakendab tänapäevaseid andmetehnika praktikaid.

Peamised eesmärgid hõlmasid normaliseeritud andmebaasiskeemi väljatöötamist, Data Build Tool'i (DBT) integreerimist tõhusate ELT (Extract, Load, Transform) andmetorude haldamiseks ning modulaarse kolmekihilise arhitektuuri (PostgreSQL, Python/Flask taustaprogramm, veebipõhine kasutajaliides) implementeerimist. Metoodika sisaldas skeemi kavandamist, andmete sissevõttu Crossrefi/OpenCitationsi API-dest ning DBT-ga hallatud SQL-transformatsioone andmete puhastamiseks ja struktureerimiseks.

Valminud süsteem demonstreerib robustset arhitektuuri bibliograafiliste andmete haldamiseks. DBT integreerimine parandab märkimisväärselt andmete transformatsiooniprotsesside hooldatavust ja usaldusväärsust. See töö pakub praktilist näidet kaasaegsete ELT põhimõtete ja DBT rakendamisest teadusinfosüsteemides, parandades tõhusust ja analüütilist potentsiaali võrreldes traditsiooniliste lähenemistega.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 100 leheküljel, 5 peatükki, 10 joonist, 3 tabelit.

List of abbreviations and terms

API	Application Programming Interface
DBT	Data Build Tool
DDL	Data Definition Language
DOI	Digital Object Identifier
ELT	Extract, Load, Transform
ERD	Entity-Relationship Diagram
FK	Foreign Key
JSON	JavaScript Object Notation
NF	Normal Form
OAI-PMH	Open Archives Initiative Protocol for Metadata Harvesting
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
ORCID	Open Researcher and Contributor ID
PK	Primary Key
RDBMS	Relational Database Management System
SQL	Structured Query Language
UI	User Interface

Table of Contents

1	Introduction	10
1.1	Motivation and Problem Statement	10
1.2	Objectives and Research Questions.....	13
1.3	Thesis Structure	15
2	Background and Context	17
2.1	Bibliographic Databases in Academic Research.....	17
2.2	Metadata Standards and Formats.....	18
2.3	Relational Theory and Data Warehousing.....	20
2.4	Literature Review	21
2.5	Research Gaps	24
3	Methodology.....	26
3.1	The Three-Tier Architecture Pattern: A General Overview	26
3.2	Requirements and System Analysis	28
3.3	Database Architecture Design Strategy	34
3.4	Data Transformation Workflow with DBT	40
3.5	Application Tier Design Strategy	44
3.6	Research Evaluation Methods	47
4	Results	48
4.1	Implemented System Components	48
4.2	Database Implementation Results	49
4.3	Data Transformation Pipeline Results (DBT)	53
4.4	Application Tier Results.....	59
4.5	Evaluation Against Research Questions.....	61
5	Conclusion and Future Work.....	64
5.1	Summary of Findings and Contributions.....	64
5.2	Future Work.....	66
5.3	Concluding Remarks	68
	References	69
	Appendix 1 – Non-exclusive licence for reproduction and publication.....	72
	Appendix 2 – Complete DDL for the dwh Schema.....	73
	Appendix 3 – dwh_schema.yml configuration file	77
	Appendix 4 – scripts of the dbt custom singular tests	80
	Appendix 5 – scripts of the dbt models and the macro	82
	Appendix 6 – app.py python script.....	95
	Appendix 7 – Source Code Repository	100

List of figures

Figure 1. The Three-Tier Architecture in Software Engineering.	26
Figure 2. System Architecture Diagram.	30
Figure 3. Entity-Relationship Diagram for the Bibliographic Database.	37
Figure 4. ELT workflow showing Python ingestion, DBT transformation/testing, and Flask app presentation.	41
Figure 5. Structure of the dbt and dwh schemas in pgAdmin.	50
Figure 6. Terminal Output of <code>dbt run</code>	53
Figure 7. DBT Data Lineage Graph.	54
Figure 8. Terminal Output of <code>dbt test</code>	58
Figure 9. Interactive faceted-search interface of the Bibliographic Library application.	59
Figure 10. Entry Detailed Information Display.	60

List of tables

Table 1. Conceptual Schema Primary Entities.	35
Table 2. Conceptual Schema Associative Entities.	35
Table 3. Mapping Conceptual Entities to Relational Tables.	38

1 Introduction

This chapter introduces the motivation behind the development of a new bibliographic database, outlines the key challenges in the existing landscape, defines the scope of this thesis, and sets forth the objectives and research questions that will guide this work. Finally, it provides a brief overview of the thesis structure.

1.1 Motivation and Problem Statement

Importance of Bibliographic Databases

Bibliographic databases are a fundamental tool of the information society. Researchers, students, professionals, and other individuals utilize them to locate relevant literature for their work or areas of interest. For librarians, these databases serve as a means to store bibliographic data concerning publications, their locations, and other details essential for library operations. Furthermore, policymakers employ bibliographic databases to monitor the progress of science, particularly to compare anticipated and actual outcomes and to assess or compare disciplines, fields, or research groups [1]. The visibility of publications in widely used bibliographic databases accessible on the World Wide Web is crucial for authors and publishers, as it ensures the dissemination of their research [2]. Therefore, accurate and up-to-date bibliographic data is indispensable for researchers across various domains [3].

The utility and impact of these databases, however, are increasingly dependent on the sophisticated computer science techniques employed to navigate and analyse their vast contents. As the volume of scholarly output continues its exponential growth, the methods for information retrieval and knowledge discovery built upon these databases have necessarily evolved.

Historically, researchers relied on exact-match Boolean queries and early vector space models to retrieve scholarly documents from databases such as PubMed and MEDLINE. However, the limitations of Boolean-only interfaces and term-frequency-based ranking—particularly in handling user intent and semantic relationships—have spurred a

wave of innovations in retrieval algorithms, citation-based analysis, knowledge graph modelling, and artificial intelligence-driven search platforms.

Several lines of research have sought to measure and improve the speed and relevance of scholarly search by leveraging advances in computer science algorithms and data structures. Early foundational work addressed the challenge of layering relevance ranking on databases that support only Boolean queries, such as PubMed. For instance, algorithms developed by Hristidis et al. demonstrated how conjunctive query generation could emulate ranked retrieval atop Boolean interfaces, achieving high relevance while reducing query and data transfer costs [4], [5]. Complementary studies highlighted the trade-offs between Boolean and ranked retrieval, showing that hybrid approaches often outperform either method alone for systematic biomedical searches [6].

Concurrently, the incorporation of graph-based and citation-aware ranking methods marked a significant evolution in academic search. Systems combining BM25 or TF-IDF text scoring with citation network metrics—such as PageRank and in/out-degree analysis—were found to enhance retrieval relevance on scholarly benchmarks [7], [8]. These hybrid IR frameworks have been validated on datasets such as the ACL Anthology, with empirical improvements in nDCG, precision, and recall over text-only baselines.

The emergence of knowledge graphs and semantic embeddings further advanced academic search capabilities. Explicit Semantic Ranking (ESR), as deployed in Semantic Scholar, introduced knowledge graph embeddings to represent and rank queries and papers based on their semantic connections, yielding measurable improvements in retrieval quality on challenging queries [9]. Similar knowledge graph-driven methods, such as MedGraph and narrative query graph ranking, demonstrated superior performance to classical retrieval across extensive biomedical literature collections [10], [11].

Most recently, artificial intelligence and large language models (LLMs) have been integrated into academic search platforms. Systems like DocReLM leverage LLMs both to annotate training data and to traverse citation graphs as “search agents,” achieving substantial top-10 retrieval accuracy gains compared to Google Scholar and BM25 on targeted arXiv datasets [12]. Additional studies have explored transformer-based semantic matching, section-aware document alignment, and conversational interfaces powered by embeddings, reporting further improvements—though often without detailed latency or user-centric benchmarks [13], [14], [15].

While these advancements in search algorithms, ranking techniques, and AI-driven analysis demonstrate the power of modern computer science in extracting knowledge from scientific literature, their effectiveness fundamentally hinges on the quality, structure, and accessibility of the underlying bibliographic data repositories. Sophisticated retrieval systems require robust, scalable, and well-managed databases capable of handling diverse and complex information efficiently.

Key Challenges

Despite the critical role of bibliographic databases, several key challenges persist in existing systems. The complexity and variety of bibliographic data are continuously increasing. Data are often provided by different and heterogeneous data sources and require discovery and integration, posing a significant challenge. For instance, specialized systems and targeted data mining efforts are sometimes necessary to effectively extract and structure bibliographic data from specific national research portals, such as the Estonian Research Information System (ETIS). Such portals can present unique and persistent challenges, including a high incidence of publications lacking standard Digital Object Identifiers (DOIs), which significantly complicates metadata retrieval and linking, and inconsistencies in how data is made available, thereby necessitating tailored extraction techniques. These techniques might include parsing directly from PDF documents or even employing web scraping methodologies when API-provided information is incomplete or lacks direct links to source files, all in an effort to compile usable bibliographic entries [16]. Moreover, enforcing a multidimensional approach to the analysis and management of bibliographic data remains an area where a reference design pattern and a specific conceptual model are still lacking [3]. While the need for such a multidimensional perspective is not new, it needs to be extended beyond the analysis phase to the design of bibliographic databases and data access services. Traditional relational database management systems, while suitable for day-to-day data storage and transactional processing, may not be appropriate for performing complex analytical tasks on a regular basis.

Another significant challenge is author name disambiguation, which arises due to different authors sharing the same name. Existing strategies for addressing this problem include author grouping and author assignment methods, often relying on properties like co-authors, institutions, and keywords. However, these methods can be limited and

require sophisticated feature extraction and machine learning techniques for effective resolution [17].

Furthermore, the process of extracting and indexing textual data from publications, including handling synonymous relationships, compound terms, stemming, lemmatization, and stop-word removal, is crucial for topic-based analysis. Describing a collection of publications in terms of the topics it contains is essential for understanding research areas and identifying relevant trends. Integrating these diverse functionalities and providing flexible tools for data transformation and analysis remains a significant gap in many existing bibliographic database systems. The need for flexibly scaling data aggregation along analysis dimensions according to different aggregation criteria is often not well-supported [3].

Scope of the Thesis

This thesis addresses the aforementioned challenges by proposing the design and implementation of a comprehensive bibliographic database system. The scope of this work includes:

- Developing a scalable and normalized database architecture capable of storing and managing large volumes of heterogeneous bibliographic data.
- Integrating Data Build Tool (DBT) to streamline and manage data transformation processes within the database.
- Implementing a three-tier system architecture consisting of a database layer, a Python-based backend for data processing and API provision, and a user-friendly frontend for data interaction.

This thesis will focus on demonstrating the feasibility and benefits of this integrated approach for enhancing the efficiency and analytical capabilities of bibliographic data management.

1.2 Objectives and Research Questions

Based on the motivation above, the main objectives of this thesis are defined as follows.

Design a scalable and normalized database architecture

Develop a relational schema for the bibliographic database that is fully normalized and scalable. This includes modelling bibliographic records (e.g. articles, books, conference papers) and related entities (authors, organisations, etc.) in a way that minimizes redundancy and maximizes data integrity. The design should ensure that common operations (insertion of new records, updates to metadata, queries for references) are efficient and that the data remains consistent as it grows [18].

Integrate DBT for data transformation and loading

Incorporate Data Build Tool (DBT) into the data pipeline to perform Extract-Load-Transform (ELT) processes within the database environment. Unlike traditional ETL (Extract, Transform, Load) processes where data is transformed before loading, this project adopts the ELT paradigm, loading raw data first and then leveraging DBT to orchestrate transformations directly within the PostgreSQL database. By using DBT, the system should improve maintainability and accuracy of the data transformation workflows. This objective involves comparing the DBT-based approach to traditional ETL in terms of development agility and data quality. Notably, DBT's modular, SQL-driven transformations and testing framework can lead to faster development cycles and improved data reliability [19]. The thesis will implement transformation models in DBT to clean, normalize, and enrich incoming bibliographic records from various sources, demonstrating how this approach enhances the database's quality and consistency.

Implement a three-tier application

Develop the bibliographic system as a three-tier architecture, with a relational database as the data tier, a Python-based backend service as the middle tier, and a web-based frontend as the presentation tier. The backend will expose an API or interface to the data, enforcing business rules and handling interactions between the user interface and the database. The frontend will provide a user-friendly access for tasks such as searching references and displaying records. This objective will validate the benefits of a three-tier design in the context of bibliographic databases, such as independent scalability of each tier and improved security (since the client does not directly access the database).

Research Questions

1. How can a scalable and normalized relational database schema be effectively designed to accommodate the diverse metadata requirements of bibliographic records for bibliometric analysis?
2. How does integrating DBT compare to traditional ETL (Extract, Transform, Load) approaches in terms of efficiency, maintainability, and flexibility for transforming heterogeneous bibliographic data?
3. How can a three-tier architecture be implemented to provide a modular and accessible bibliographic database system that effectively separates data storage, business logic, and user interface concerns?
4. What are the key considerations and challenges in applying DBT for specific bibliographic data transformation tasks, such as data cleaning, entity linking, and the generation of derived bibliometric variables?

1.3 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2: Background and Context – This chapter establishes the foundational knowledge for the research. It delves into the significance of bibliographic databases in academic research, examines existing metadata standards and formats such as BibTeX and Dublin Core, and discusses relational database theory and data warehousing concepts relevant to the project. A literature review of current bibliographic systems and data transformation tools is presented, highlighting research gaps, particularly the limited application of modern data transformation methodologies like DBT in this domain.

Chapter 3: Methodology – This chapter details the systematic approach taken for the design and development of the bibliographic database system. It introduces the three-tier architecture (Presentation, Logic, and Data tiers) adopted for the project and elaborates on the system requirements and analysis that guided the design. The database architecture design strategy is presented, covering conceptual, logical, and physical schema design, including normalization, partitioning, and indexing strategies. The chapter explains the data transformation workflow using Data Build Tool (DBT), detailing the ELT process, model implementation strategy (staging, intermediate, target models), and the planned testing approach. Furthermore, the application tier design for the Flask-based backend

and web frontend is outlined, along with the research evaluation methods used to assess the system against the research questions.

Chapter 4: Results – This chapter presents the outcomes and findings from the implementation of the bibliographic database system. It describes the successfully implemented system components across the data, logic, and presentation tiers. The database implementation results are detailed, including the final logical and physical schema, partitioning, indexing, and full-text search capabilities. The chapter then showcases the data transformation pipeline results achieved with DBT, including workflow execution, model implementation specifics (JSON parsing, surrogate key generation, incremental loading), and the outcomes of the comprehensive testing strategy. Finally, the application tier results are presented, demonstrating the functionality of the Flask web application, its interactive user interface features (search, filtering, pagination, detailed record display), and the basic JSON API endpoint. The chapter concludes with an evaluation of these results against the initial research questions.

Chapter 5: Conclusion and Future Work – The final chapter summarizes the key findings and contributions of the thesis. It revisits the initial problem statement and objectives, demonstrating how the developed system addresses the challenges of bibliographic data management. The research questions are explicitly answered based on the evaluation presented in Chapter 4. The limitations of the current work, such as prototype scale and data source scope, are acknowledged. Finally, several avenues for future work are proposed, including expanding data integration, implementing advanced bibliometric and network analysis, leveraging machine learning and AI for semantic search and knowledge extraction, enhancing the frontend and visualization capabilities, and exploring cloud deployment and further performance optimization.

2 Background and Context

This chapter provides the essential background and context for the thesis. It begins by examining the crucial role of bibliographic databases in academic research, then discusses the prevalent metadata standards and formats used in these databases. Following this, it delves into the fundamental concepts of relational theory and data warehousing, highlighting their relevance to bibliographic data management. Finally, it reviews related work in the field of bibliographic systems and the use of data transformation tools.

2.1 Bibliographic Databases in Academic Research

Definition and Role

Bibliographic databases contain bibliographic records which provide descriptive information about relevant information sources. They facilitate literature discovery by allowing users to search and browse publications based on various criteria such as authors, affiliations, titles, and publication dates [20]. Furthermore, bibliographic databases are essential for citation tracking, enabling the identification of influential works and the understanding of the relationships between different publications through citations. This citation information is crucial for researchers to understand the lineage and impact of scholarly work [21]. The visibility of publications in these databases ensures the dissemination of research findings to a wider audience [1].

State of the Art

Several existing solutions for bibliographic databases are available, including both commercial and open-access options. Examples of widely used scholarly bibliographic databases for computer scientists include Google Scholar, Microsoft Academic Search, ACM Digital Library, IEEE Xplore, DBLP Computer Science Bibliography, and CiteSeerX [2]. Initiatives like Crossref and OpenCitations aim to provide openly accessible citation metadata [22].

However, these existing systems often have limitations. Some are proprietary in nature, which can restrict access and usage [23]. Even those that are freely accessible might have heterogeneous inclusion requirements and varying levels of data quality. Furthermore, many traditional systems lack integrated tools for flexible data transformation and analysis. Some databases are primarily designed for information retrieval rather than in-depth bibliometric analysis. The need for flexible scaling of data aggregation along analysis dimensions according to different criteria is often not well-supported. Moreover, the challenge of integrating data from different and heterogeneous data sources remains a significant hurdle [3]. While some systems offer APIs or OAI-PMH interfaces for data access, others, like Google Scholar and the Collection of Computer Science Bibliographies, do not [2].

2.2 Metadata Standards and Formats

Several metadata standards and formats are used for describing and exchanging bibliographic information.

BibTeX

BibTeX is a widely accepted bibliographic metadata format, particularly within the computer science community. It is often used with the LaTeX word-processing application. A BibTeX file is a plain text file containing one or more entries, each describing a publication. Each entry starts with an entry type (e.g., @article, @book, @inproceedings) followed by a citation key and a set of fields enclosed in curly braces. These fields represent different attributes of the publication, such as author, title, journal, year, volume, and pages. The citation key is a unique identifier used to reference the entry within a LaTeX document. When the LaTeX document is compiled, BibTeX generates a list of references based on the selected citation style and the information in the BibTeX entries. BibTeX is crucial for referencing in computer science due to its simplicity, widespread support, and integration with document preparation tools used by researchers. Many bibliographic databases offer BibTeX as an export format [2].

Dublin Core

The Dublin Core Metadata Initiative (DCMI) developed a common system for using metadata to describe web resources. This system aims to allow website authors to describe

their content in a way that can be discovered by keyword-based search engines. Dublin Core consists of a set of flexible metadata elements, including a core set of 15 elements and some additional ones. Examples of core elements include title, type, creator, issued, and bibliographicCitation. All elements are optional, can be repeated, and can appear in any order. Dublin Core Metadata can be presented in RDF/XML or embedded in the <head> section of HTML documents using <meta> tags. While Google Scholar supports Dublin Core tags, it is not recommended for journal papers as they are considered to “work poorly” [2].

BibTeX and Dublin Core are detailed due to their prominence in academic research and web metadata, respectively. Beyond these, other bibliographic data formats and protocols also serve various functions within the scholarly communication landscape. However, these are considered less central to the core design of this particular thesis—which prioritizes ingesting and transforming rich metadata directly from modern APIs like Crossref—and thus are mentioned more briefly. These additional formats and protocols include:

- MARC (Machine-Readable Cataloging) is a standard developed by the Library of Congress [2].
- MODS (Metadata Object Description Schema) is another metadata standard emanating from the Library of Congress and defines the structure of bibliographic records. MODS can be converted to and from BibTeX and EndNote [24].
- PRISM (Publishing Requirements for Industry Standard Metadata) is another metadata standard [2].
- Highwire Press tags are HTML meta tags recommended by Google Scholar for indexing [2].
- RIS (Research Information Systems) format was developed by Thomson Reuters for applications like EndNote and ReferenceManager to import and export bibliographic metadata. RIS is also a frequently offered export format by bibliographic databases [2].
- DBLP XML is a special XML format developed by the DBLP Computer Science Bibliography, essentially BibTeX written in XML [2].
- OAI-PMH (Open Archives Initiative Protocol for Metadata Harvesting) is a protocol used to exchange metadata [25].

2.3 Relational Theory and Data Warehousing

Operational vs. Analytical Databases

Relational database management systems (RDBMS) are traditionally used for day-to-day data storage and transactional processing, which is characteristic of Online Transaction Processing (OLTP) systems. OLTP systems are designed for high transaction volumes and focus on data integrity and consistency through normalization [26]. In contrast, Online Analytical Processing (OLAP) systems are designed for performing complex analytical tasks on large datasets, often involving multiple scans, joins, and summaries. OLAP systems often use a multidimensional data model, such as a star schema, to facilitate fast interactive browsing and analysis of hierarchical and summarized data. The response time of complex queries is a crucial factor in designing OLAP applications [1], [26].

Normalization and Data Modelling

Normalization is a process of applying a set of rules to database design, primarily to achieve minimum redundancy in the data. The goal is to organize data in tables in such a way that dependencies between attributes are clear and logical, reducing the likelihood of data anomalies during updates and insertions [27]. Basic principles of normalization include [18]:

- First Normal Form (1NF): Each cell in a table should contain a single value, and each column should have a unique name.
- Second Normal Form (2NF): The table must be in 1NF, and all non-key attributes must be fully functionally dependent on the entire primary key.
- Third Normal Form (3NF): The table must be in 2NF, and all non-key attributes must be non-transitively dependent on the primary key (i.e., no non-key attribute depends on another non-key attribute).

For bibliographic databases, data modelling involves identifying the key entities (e.g., publications, authors, journals) and their relationships. An entity-relationship diagram (ERD) is often used to visually represent the database schema and the connections between tables [27]. Primary keys uniquely identify records within a table, and foreign keys establish relationships between tables by referencing primary keys in other tables.

Data Warehouse vs. Database

Typical bibliographic needs for managing and retrieving publication metadata lean toward normalized operational structures (databases) to ensure data integrity, consistency, and efficient transactional operations like adding, updating, and searching records [26]. The relational model is well-suited for storing the structural information found in scientific articles [28].

However, bibliographic data can also benefit from partial warehousing and analytics features. For instance, OLAP technology can be applied to bibliographic databases for tasks such as periodic and ad hoc reporting, quality assurance, data integrity checking, and for research policy makers to monitor scientific development. This involves creating a data warehouse, which is a subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process. Data warehouses often use a multidimensional model (e.g., star schema) with fact tables containing measures (e.g., number of publications, citations) and dimension tables providing the context for analysis (e.g., year of publication, author affiliation). While building a full bibliographic data warehouse can be time-consuming, incorporating some analytical capabilities alongside a normalized operational database can enhance its utility for bibliometric analysis) [1], [26].

While building a full bibliographic data warehouse can be time-consuming, incorporating analytical capabilities alongside a normalized operational database can enhance its utility. A well-structured and documented schema, even if normalized like the `dwh` schema implemented in this project, facilitates direct connection using standard SQL clients (e.g., pgAdmin, DBeaver) for complex, ad-hoc analytical queries and enables integration with external Business Intelligence platforms (e.g., Tableau, Power BI) for sophisticated visualization and reporting beyond the scope of typical OLTP applications.

2.4 Literature Review

This section provides a critical analysis of existing literature concerning bibliographic databases and related methodologies. It aims to establish the context for this research by examining the current state of bibliographic systems and identifying gaps in their documented development and architecture. Furthermore, it explores the potential

application of data build tools (DBT) within academic contexts, specifically concerning bibliographic data.

Existing Bibliographic Systems

As it was mentioned previously, the literature review on this particular topic which includes the actual design and implementation is very limited, however, it was possible to find a published work by K. Karamcheti's titled as "Design and implementation of bibliographic database" (Master's Thesis, University of Nevada, Las Vegas, 2007) Karamcheti investigated the design and implementation of a bibliographic database. The thesis focused on the identification and implementation of recursive queries. It detailed the creation of various database tables, defining primary and foreign keys, and presented the database entity relationship using UML notations. The database included a master table with cite key entries and their corresponding entry types for different bibliographic categories such as articles, books, and proceedings [18]. While Karamcheti's work outlines the structural design of the database, it does not delve into the intricacies of its architecture or the detailed development process beyond the conceptual and logical modelling.

Several other academic works address aspects of bibliographic databases, each with a specific focus:

Modelling and Analysis

Ferrara and Salini discuss ten challenges in modelling bibliographic data for bibliometric analysis, proposing a multidimensional model. Their work emphasizes the need for a multidimensional approach not only in analysis but also in database design and data access services [3]. Mallig developed a relational database schema specifically for bibliometric analysis, providing SQL queries to demonstrate its utility in calculating bibliometric indicators [28]. Cobo et al. presented a relational database model tailored for science mapping analysis, focusing on its application in various stages of the science mapping workflow [27]. These works primarily concentrate on the logical modelling and the analytical capabilities of bibliographic data rather than providing comprehensive details on the physical architecture or the development process.

Bibliometric Applications and OLAP

Hudomalj and Vidmar explored the application of Online Analytical Processing (OLAP) technology to bibliographic databases, demonstrating its utility for reporting, quality assurance, and research policy analysis [1]. Georgieva-Trifonova proposed bgMath/OLAP, a system for warehousing and OLAP analysis of bibliographic data, aimed at monitoring and evaluating scientific development [26]. These studies highlight the analytical benefits of specific technologies applied to bibliographic data but do not provide an in-depth account of the underlying database development or architectural choices beyond the adoption of OLAP principles.

Indexing and Data Integration

Kusserow and Groppe surveyed the technical requirements for getting indexed by widely used bibliographic databases, offering insights into data formats and protocols [2]. Do et al. proposed a framework for integrating bibliographic data from heterogeneous digital libraries, including components for data collection, parsing, and duplicate checking [25]. Their work touches upon the challenges of data integration but does not detail the complete architecture of an integrated bibliographic database from the ground up.

Author Name Disambiguation and Data Quality

Silva focused on feature extraction for author name disambiguation in a bibliographic database, using the Authenticus database for Portuguese researchers [17]. Manghi et al. presented GDup, a system for entity deduplication in big data graphs for scholarly communication, addressing the problem of duplicate entities arising from multiple data sources [29]. These studies address specific data quality issues and propose solutions, but they do not provide a holistic view of bibliographic database development and architecture.

Network Analysis and Citation Analysis

Butt et al. developed a systematic metadata harvesting workflow for analysing scientific networks using data from Crossref and OpenCitations [22]. Zupic and Čater introduced various bibliometric methods, including citation analysis, co-citation analysis, and bibliographical coupling, providing a workflow for conducting bibliometric studies [21].

Smalheiser et al. presented a web-based tool for citation analysis, illustrating different types of citation relationships [30]. Mallig also discussed the use of a structured bibliometric database as a resource for various bibliometric networks [28]. While these works extensively utilize bibliographic data for network and citation analysis, they do not primarily focus on the development and architectural aspects of the underlying databases themselves.

2.5 Research Gaps

While the reviewed literature offers valuable insights into various aspects of bibliographic databases, many works lack comprehensive details regarding the end-to-end development process and the underlying system architecture. Some limitations and areas of incompleteness include:

- **Limited Architectural Detail:** Many studies focus on conceptual models or specific functionalities like analysis or disambiguation without providing a thorough exposition of the database architecture, including hardware considerations, specific software components, and data flow mechanisms.
- **Scarcity of Development Process Documentation:** The process of building and deploying a bibliographic database, including data acquisition strategies, schema evolution, maintenance procedures, and scalability considerations, is often not presented.
- **Focus on Logical vs. Physical Implementation:** The emphasis tends to be on the logical design of the database schema rather than the physical implementation details, such as indexing strategies, partitioning schemes, and optimization techniques for query performance.
- **Data Quality Challenges:** Several sources acknowledge the persistent challenges of data quality in bibliographic databases, including heterogeneity, incompleteness, and the presence of duplicates. However, detailed accounts of strategies and architectures specifically designed to address these issues throughout the development lifecycle are often missing.
- **Limitations of Existing Online Databases:** The literature points out that online bibliographic databases are often designed primarily for information retrieval rather than comprehensive bibliometric analysis, leading to limitations in data

accessibility and quality for analytical purposes. This motivates the development of in-house databases, but their detailed construction remains undocumented.

- **Challenges in Network Data Management:** While network analysis of bibliographic data is a prominent area, the complexities of storing and efficiently querying network data within a bibliographic database architecture are not always fully addressed.

Use of modern data transformation methodologies in Academic Context

A thorough review of the provided sources reveals no direct mention or precedent of using data build tools (DBT) or similar modern data transformation methodologies specifically within the context of bibliographic databases or academic research data management. The literature predominantly discusses traditional database management systems, relational models, and OLAP technologies for organizing and analysing bibliographic information. The focus is on schema design, query optimization within these established paradigms, and addressing data quality through techniques like deduplication and record linkage.

The absence of discussions around modern data transformation methodologies in the provided sources suggests a potential gap in the application of contemporary data transformation practices within the academic bibliographic domain. DBT's focus on modular data modelling, version control for data transformations, and automated testing could offer significant advantages in building and maintaining robust and analytical-ready bibliographic databases. This includes streamlining the process of cleaning, transforming, and integrating data from diverse sources, which is a recurring challenge highlighted in the literature.

The research questions introduced in this work in Section 1.2 are based on the gaps identified above.

3 Methodology

This chapter details the methodological approach undertaken in the development of the bibliographic database. The design and implementation of the system have been guided by principles of modularity, scalability, and maintainability, resulting in a three-tier architectural pattern. Furthermore, the selection of specific tools and technologies was driven by their suitability for managing bibliographic data, facilitating robust data transformations, and providing an accessible user interface.

3.1 The Three-Tier Architecture Pattern: A General Overview

Three-tier architecture is a foundational design pattern in modern software engineering that partitions applications into three logical and often physical domains: the presentation tier (user interface), the logic or application tier (business logic, workflows, validation), and the data or persistence tier (databases, storage systems) [31], [32]. As illustrated in Figure 1, this model logically separates an application into three distinct, interconnected layers or tiers, each responsible for a specific set of functionalities:

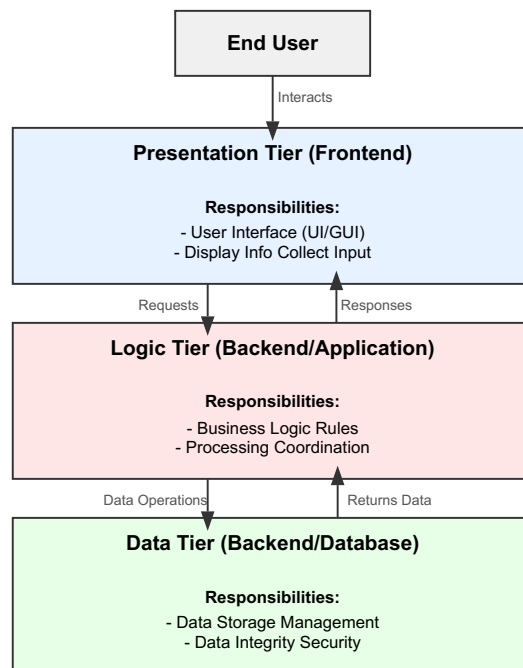


Figure 1. The Three-Tier Architecture in Software Engineering.

1. **Presentation Tier (Frontend):** This is the topmost layer, responsible for interacting with the user. It displays information to the user and collects user input. Its primary role is user interface (UI) management, handling how data is presented and how user actions are captured. It communicates user requests to the logic tier and presents responses received from it. Examples include web browsers rendering HTML pages, graphical user interfaces (GUIs) in desktop applications, or mobile app interfaces.
2. **Logic Tier (Backend/Application/Middle Tier):** This tier acts as the intermediary between the Presentation and Data tiers. It contains the core business logic, processing rules, calculations, and workflows of the application. It receives requests from the Presentation tier, processes them (often involving complex operations or coordination), interacts with the Data tier to fetch or store information, and sends the results back to the Presentation tier. This layer ensures that business rules are consistently applied and isolates the presentation layer from direct database access. Examples include web servers running application code (like Python/Flask, Java Servlets, Node.js) or dedicated application servers.
3. **Data Tier (Backend/Database Tier):** This is the foundational layer responsible for the persistent storage and management of application data. It typically consists of a database management system (DBMS) where data is stored, retrieved, updated, and deleted. The Logic Tier communicates with this tier to perform data operations. This layer ensures data integrity, security, and availability. Examples include relational databases (like PostgreSQL, MySQL), NoSQL databases, or other data storage systems.

Benefits of the Three-Tier Architecture:

- **Modularity:** Each tier can be developed, managed, and updated independently of the others, as long as the interfaces between them remain stable. This allows different teams to work on different tiers concurrently.
- **Scalability:** Each tier can be scaled independently based on specific load requirements. For example, if the application logic becomes a bottleneck, more resources can be allocated to the Logic Tier without affecting the other tiers.

- **Flexibility & Maintainability:** Changes or technology upgrades in one tier (e.g., switching the database or redesigning the UI) have minimal impact on the other tiers, simplifying maintenance and allowing for easier technology evolution.
- **Improved Security:** The separation enforces indirect access to the database (only via the Logic Tier), enhancing security by preventing direct client exposure to the data layer.
- **Reusability:** The business logic encapsulated in the Logic Tier can potentially be reused by multiple presentation interfaces (e.g., a web UI and a mobile app).

This architectural pattern provides a robust foundation for developing complex applications by promoting a clear separation of concerns, enhancing scalability, and simplifying development and maintenance.

3.2 Requirements and System Analysis

The development of a robust bibliographic database necessitates a well-defined architecture and a thorough understanding of system requirements. This section outlines the fundamental design principles, the rationale behind the selection of specific technologies, the chosen data sources, and the core functionalities implemented within this project.

System Architecture

To ensure a clear separation of concerns and facilitate maintainability and scalability, a three-tier architecture has been adopted for this bibliographic database project, as illustrated in Figure 2. This architectural pattern divides the system into 3 distinct layers:

1. Data Layer:

This foundational layer is responsible for the persistent storage of bibliographic data. It houses the PostgreSQL database, which stores both the initially ingested raw data (e.g., in the `raw_crossref` table) and the subsequently transformed and structured data produced by the Logic Layer. The database acts as the central repository, ensuring data integrity and availability for processing and analysis. The challenges associated with managing large volumes of bibliographic data [33] necessitate a robust and scalable

database solution like PostgreSQL. External data sources, namely the Crossref and OpenCitations APIs, provide the raw information that is ultimately stored in this layer.

2. Logic Layer:

This intermediary layer encapsulates the core processing logic, acting as the bridge between the external data sources and the final structured data used by the Presentation Layer. It encompasses two key functions:

- **Data Ingestion and Loading:**

The Python script (`crossref_opencitations_load.py`) resides in this layer. It handles the logic for interacting with the external Crossref and OpenCitations APIs, fetching the required metadata and citation information, performing initial parsing, and loading this raw data into the `raw_crossref` table within the Data Layer.

- **Data Transformation:**

We utilize DBT (Data Build Tool) to manage the SQL-based transformations within this layer. DBT's framework allows us to define data models as SQL queries that operate on the data stored in the Data Layer. These models are systematically organized into a pipeline comprising **staging models** (for initial cleaning and pre-processing steps), **intermediate models** (for structuring core entities and applying complex logic), and **target models** (optimized for the final schema and presentation). This layered approach ensures that data is progressively cleansed, transformed, structured, and integrated to meet analytical and presentation requirements. The use of DBT promotes version control for our data transformations, facilitates testing, and enhances the overall maintainability and reproducibility of our data pipelines. This aligns with the principles of fostering reproducible research, as mentioned in the context of shared platforms for bibliographic data [33]. The incremental building of these models, facilitated by DBT, allows us to process only new or changed data, improving the efficiency and reducing the processing time of our data pipelines [34].

3. Presentation Layer:

Intended to provide user interaction via a Flask-based web application (`app.py`). This layer was designed to consume processed data from the Logic Layer (specifically a

planned presentation view), focusing on user experience and data display without handling underlying storage or complex transformation logic.

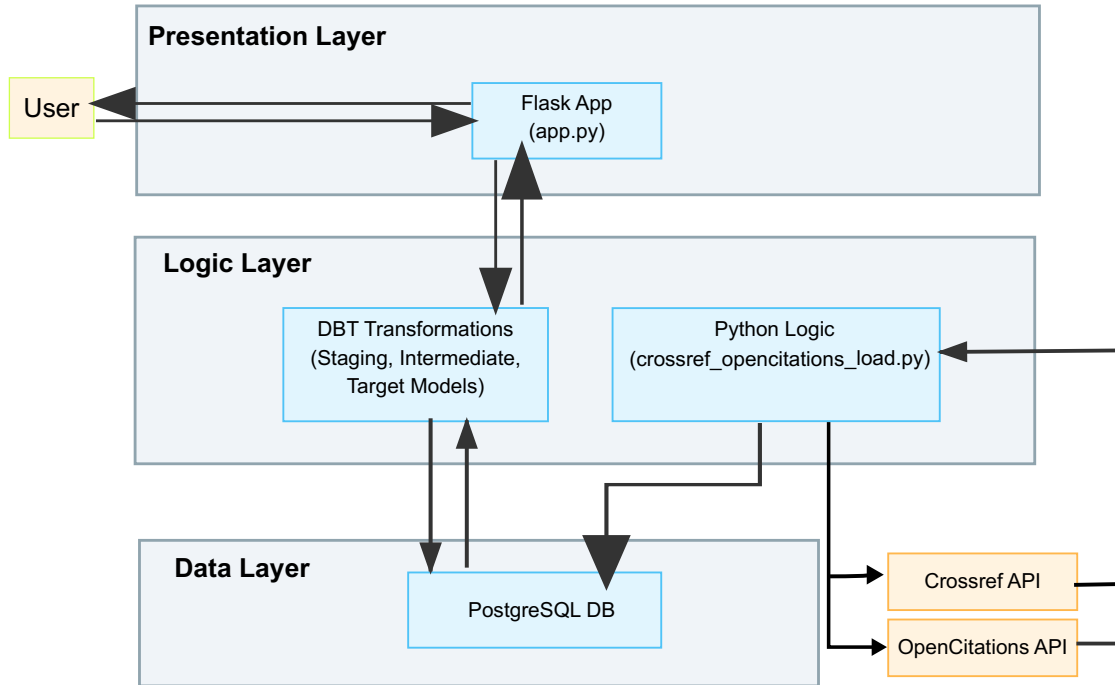


Figure 2. System Architecture Diagram.

Choice of Tools

The selection of tools was driven by their suitability for each architectural layer, capabilities in handling bibliographic data, alignment with modern data engineering practices, community support, and open-source nature.

- **PostgreSQL:** Chosen as the RDBMS for its reliability, advanced SQL features, data integrity support, scalability options (partitioning, indexing), and strong community support, making it suitable for managing both raw and transformed bibliographic data. Its scalability options, including replication and partitioning, provide a path for handling future growth in data volume and user traffic [35].
- **Python:** Selected for backend logic (data ingestion, API) due to its versatility, extensive libraries for API/database interaction, and widespread adoption.
- **DBT (Data Build Tool):** Adopted to manage SQL transformations within PostgreSQL. The key reasons were its ability to version control SQL transformations, facilitate modular development, automate testing, and improve

the maintainability and reliability of the data pipeline, aligning with modern data warehousing practices. DBT's focus on data quality and the structured management of transformations aligns with best practices in data warehousing and data integration [36].

- **Flask:** Chosen for the presentation tier due to its lightweight nature and ease of use for developing web applications and APIs with Python, suitable for a proof-of-concept.
- **Docker:** Planned for containerizing the PostgreSQL database to ensure a reproducible and isolated environment for development and testing, simplifying setup.

Key Libraries and Packages Used

Beyond the core frameworks, several key Python libraries were essential for the project's functionality, identified via import statements in the project's scripts:

- **Flask:** Provided the foundation for the web application, handling routing, requests, and responses (`app.py`).
- **psycopg2:** Enabled interaction with the PostgreSQL database, allowing Python scripts (`app.py`, `crossref_opencitations_load.py`) to execute SQL queries and fetch results.
- **requests:** Used by the data ingestion script (`crossref_opencitations_load.py`) to make HTTP calls to the Crossref and OpenCitations APIs.
- **pandas:** Utilized in the ingestion script (`crossref_opencitations_load.py`) for previewing fetched data in a structured DataFrame format before database insertion.
- **subprocess:** Employed by the ingestion script (`crossref_opencitations_load.py`) to execute the `dbt run` command programmatically via the `trigger_dbt()` function.
- Other standard libraries like `os`, `json`, `logging`, `sys`, and `urllib.parse` were used for file system interaction, JSON handling, logging, system functions, and URL manipulation, respectively.

Choice of Data Sources

The selection of data sources was guided by the need for comprehensive and openly accessible bibliographic and citation metadata.

- **Crossref:** Chosen as the primary source for publication metadata due to its role as an open registry for DOIs and provider of rich, near real-time scholarly metadata.
- **OpenCitations:** Selected to complement Crossref by providing open citation links, crucial for enabling citation analysis. Using both aimed to provide a comprehensive view without paywall restrictions.

The use of both Crossref and OpenCitations as data sources ensures a comprehensive view of scholarly communication without the limitations of paywalls or licensing restrictions.

Functional Requirements

The bibliographic database was designed to meet several key functional requirements:

- **Bibliographic Record Storage:** Store diverse entry types (articles, proceedings, books) with essential metadata (DOI, title, authors, dates) in a structured format.
- **Metadata Management:** Effectively manage relationships between entities (authors, affiliations, entries) using the relational model.
- **Scalable Data Ingestion:** Design an ingestion process capable of handling potentially large data volumes from APIs.
- **Analytical Queries:** Ensure the final schema supports common analytical queries (citation counts, co-authorship).
- **Integration with Transformations:** Integrate DBT seamlessly into the workflow for transforming raw data into a clean, reliable schema with enforced data integrity.
- **Type-Based Filtering:** Allow users to filter search results based on specific publication types (e.g., journal-article, proceedings-paper) selected via interactive checkboxes in the user interface. The interface should display counts of matching records per type alongside the total counts for context.

- **Detailed Record Display:** Present search results clearly in the user interface, including primary author information in the main table view. An expandable section per entry must reveal comprehensive details, including a full list of all authors associated with the entry (with ORCID links and primary author indication), an interactive abstract display (allowing users to toggle between a preview and the full text), linked DOIs for `citations` and `cited_by` data, and other relevant metadata fields (e.g., Container Title, Volume, Issue).
- **Configurable Pagination:** Provide users the ability to select the number of search results displayed per page (e.g., 10, 25, 50, 100) through a dropdown menu in the user interface.
- **Interactive Data Presentation:** Enhance usability by truncating long text fields (e.g., titles, DOIs, organization names) in the results table, providing an intuitive mechanism (e.g., an expandable “Show full” element) for users to view the complete content on demand.
- **Support for External Analytics and Visualization:** Ensure the final structured data schema (`dwh`) is designed and documented in a way that allows for direct querying via standard SQL database tools (like `pgAdmin` or `DBeaver`) and facilitates connection from external data visualization and Business Intelligence platforms (such as `Tableau` or `Power BI`) to enable advanced analytical exploration beyond the primary application interface.

Non-Functional Requirements

In addition to the core functional capabilities, the design also targeted key quality attributes.

- **Maintainability:** Enhanced through `DBT`’s modularity, version control, and testing framework.
- **Performance:** Addressed through database normalization, planned indexing strategies, and consideration of partitioning for potential large-scale data.
- **Reliability and Data Consistency:** Ensured through planned database integrity constraints (PKs, FKs) and `DBT` tests for data quality validation.
- **Basic Security and Access Control:** Included considerations for database authentication and secure credential management.

3.3 Database Architecture Design Strategy

The database design followed a top-down approach across three abstraction layers, focusing on capturing appropriate decisions while maintaining traceability. The goal was to create a schema optimized for both operational data management and analytical querying needs inherent in bibliographic datasets.

Conceptual Schema

This schema aimed to answer: “What real-world facts must the system remember?” Based on project requirements (citation analysis, author tracking) and source capabilities (Crossref/OpenCitations), the core entities identified were as follows.

Primary Entities: Entry (citable artefact), Author, Organization, Entry_Type (controlled vocabulary). The details and rationale are described in Table 1.

Associative Entities: Entry_Author (many-to-many link, preserves author order), Author_Organization (many-to-many link). The details and rationale are described in Table 2.

Table 1. Conceptual Schema Primary Entities.

Entity	Business meaning	Essential attributes	Invariants & business rules
Entry	Any citable artefact (journal article, pre-print, book chapter, dataset ...)	DOI, title, publisher, issued-date, language, abstract	<ul style="list-style-type: none"> • Every Entry must have exactly one Entry Type. • An Entry can exist with zero or many Authors (e.g., editorials).
Author	A person or collective credited on a work	full name, ORCID	<ul style="list-style-type: none"> • ORCID, when present, uniquely identifies an Author. • A single Author may have multiple concurrent or historical affiliations.
Organization	An institutional affiliation string normalized to a canonical form	name	<ul style="list-style-type: none"> • Name is treated case-insensitively; multiple spellings collapse to one Organization.
Entry_Type	Controlled vocabulary term supplied by Crossref (journal-article, proceedings-paper, ...)	type-name, optional description	<ul style="list-style-type: none"> • Vocabulary is finite and slowly changing; new types are appended, never altered.

Table 2. Conceptual Schema Associative Entities.

Link	Purpose	Cardinality notes
Entry_Author	Resolves the many-to-many between Entry and Author and stores author order and “primary author” flag	One row per (entry × author). Author sequence preserved via ordinality integer.
Author_Organization	Captures the many-to-many between Author and Organization together with affiliation order	Allows an Author to be counted under multiple institutions in bibliometric roll-ups.

Key Design Logic:

- Explicit associative tables were chosen over embedding lists (like JSON) to facilitate efficient relational querying (e.g., co-authorship analysis).
- Including ORCID in Author aimed to support future identity resolution.
- Modelling Entry_Type separately enables referential integrity and downstream filtering.
- Citation links were viewed as relationships between Entry records, to be realized via joins in the logical model rather than a separate conceptual entity.

This conceptual schema served as the blueprint for the subsequent logical design.

Logical Schema

The logical design translated conceptual entities into a relational structure optimized for data integrity and efficient transformation via DBT.

Normalization strategy:

- Third Normal Form (3NF): Targeted to minimize redundancy and update anomalies.
- Surrogate Keys: Planned use of deterministic hashes (via DBT macro) as primary keys for efficient joins and idempotent loads.
- Natural Keys Retained: Intended to retain keys like DOI or ORCID with UNIQUE constraints for external reconciliation.

The high-level mapping from conceptual entities to planned relational tables is shown in Table 3. The relationships were visualized in an Entity-Relationship Diagram as presented in Figure 3.

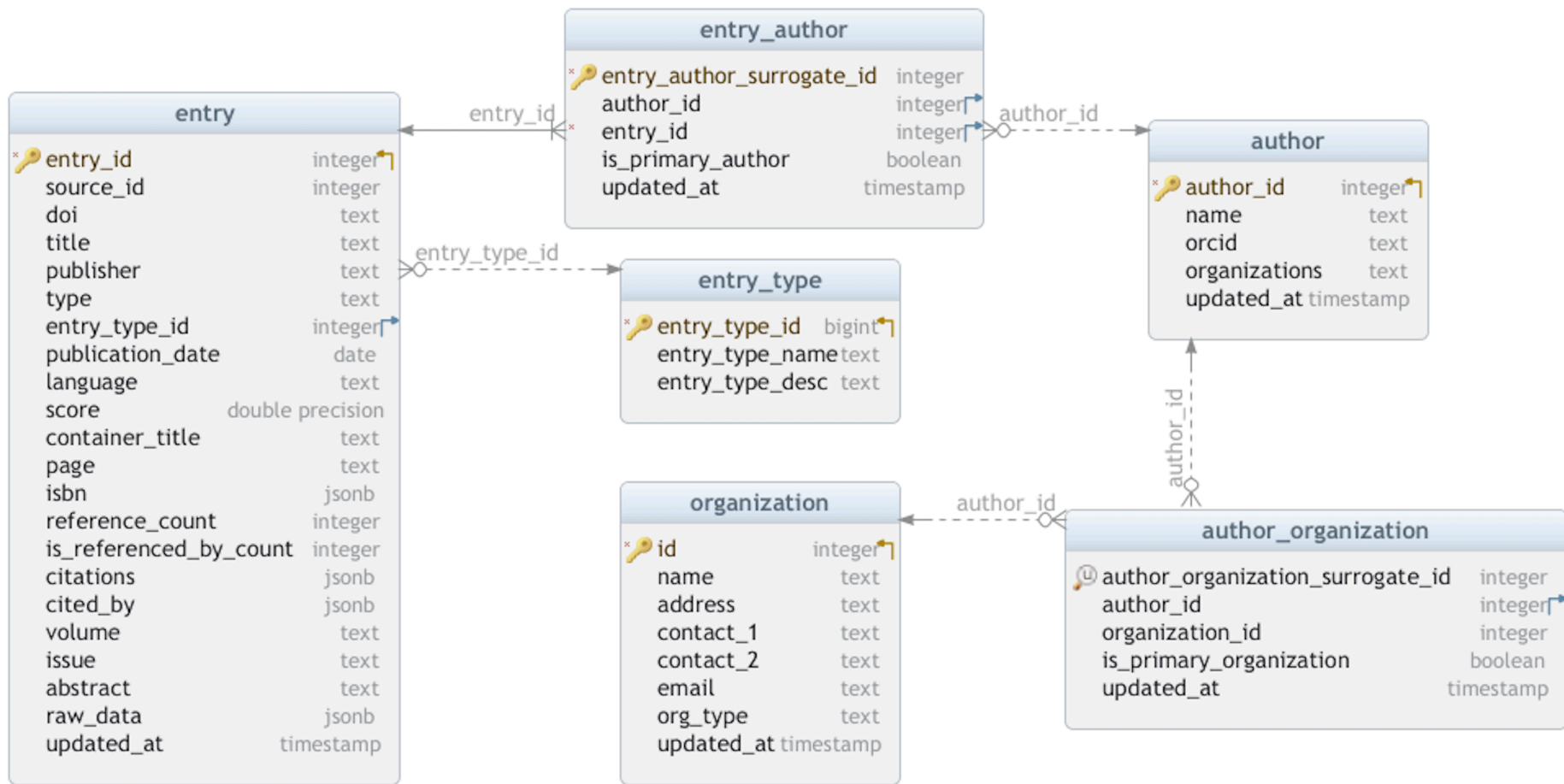


Figure 3. Entity-Relationship Diagram for the Bibliographic Database.

Table 3. Mapping Conceptual Entities to Relational Tables.

Conceptual entity	Relational table	Primary key	Important constraints
Entry	dwh.entry (partitioned)	(entry_id, publication_date)	UNIQUE (doi, publication_date) to obey partition-key rule
Author	dwh.author	author_id	UNIQUE (lower(name), orcid)
Organization	dwh.organization	id	UNIQUE (lower(name))
Entry Type	dwh.entry_type	entry_type_id	UNIQUE (entry_type_name)
Entry Author	dwh.entry_author (hash-partitioned)	(entry_id, author_id)	FK to both parent tables, ordinality NOT NULL
Author Organization	dwh.author_organization	author_organization_surrogate_id	FK to Author and Organization

Design Goals:

- **Minimal Redundancy:** Ensuring author names, organization names, etc., are stored once.
- **Time-Variant Support:** Planning to include `updated_at` timestamps for incremental processing and potential temporal analysis.
- **Flexible Analytics:** Designing a star-like topology to support OLAP-style rollups.
- **Separation of Concerns:** Defining clear boundaries between raw data ingestion, DBT transformations, and UI access.

Physical Schema Design Rationale

This layer focused on translating the logical design into concrete PostgreSQL objects, aiming to meet non-functional requirements like performance and scalability. The strategy included:

Partitioning Strategy:

- *Rationale:* To improve query performance by allowing the planner to skip irrelevant data partitions (partition pruning) and to manage large tables more effectively (e.g., for maintenance, bulk loading).
- *Planned Approach:* Intended to use PostgreSQL's declarative partitioning. The plan was to apply RANGE partitioning on the `dwh.entry` table using `publication_date` (yearly), aligning with common temporal filtering patterns. HASH partitioning was planned for `dwh.entry_author` based on `entry_id` to potentially improve join performance by co-locating related rows.

Indexing Strategy:

- *Rationale:* To accelerate data retrieval for common query patterns, especially search and joins.
- *Planned Approach:* Intended to create PRIMARY KEY indexes (implicitly created) and FOREIGN KEY indexes to enforce relationships and speed up joins. Crucially, planned to utilize GIN indexes with the `pg_trgm` extension on text fields like `title` and `author_name` to efficiently support case-insensitive

substring searches (`ILIKE '%term%'`). Considered indexes on frequently filtered or sorted columns like `publication_date`.

Constraint Policy:

- *Planned Approach:* Intended to enforce UNIQUE constraints on natural keys (DOI, ORCID, names where applicable) and use FOREIGN KEY constraints to maintain referential integrity. Considered using DEFERRABLE constraints to allow bulk validation at the end of DBT transactions.

Storage Parameters and Maintenance:

- *Planned Approach:* Acknowledged the potential need to tune PostgreSQL parameters (e.g., `fillfactor` for tables with frequent updates, `work_mem` for transformations, `autovacuum` settings) based on observed workload and data volume, although specific values would depend on testing with actual large volume of data. Planned to rely on `autovacuum` for routine maintenance and potentially schedule `ANALYZE` commands to keep statistics up-to-date.

3.4 Data Transformation Workflow with DBT

The core of the data processing logic was planned around an ELT (Extract, Load, Transform) workflow managed by DBT as illustrated in Figure 4, leveraging the capabilities of the PostgreSQL database.

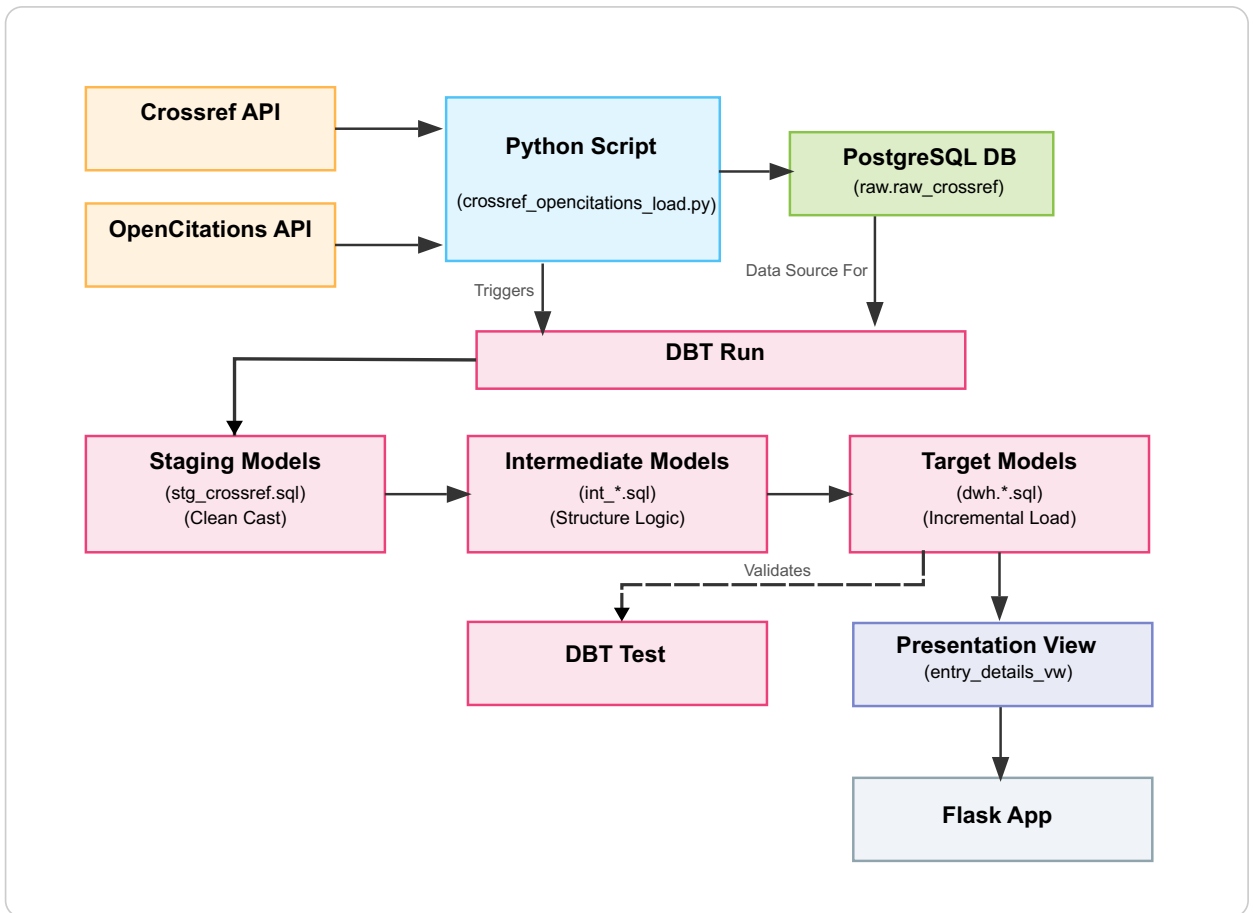


Figure 4. ELT workflow showing Python ingestion, DBT transformation/testing, and Flask app presentation.

DBT Project Setup Strategy

The DBT project was planned with the following configuration principles:

- **Source Definition:** Explicitly define raw data tables (e.g., `raw.raw_crossref`) as DBT sources (`sources.yml`) for reliable referencing in staging models (`{{ source(...) }}`).
- **Target Schema Definition:** Define final `dwh` tables also as sources where necessary to allow intermediate models to perform lookups (e.g., checking for existing IDs or values) without creating circular dependencies.
- **Custom Macro:** Plan to implement reusable SQL logic snippets as macros, such as the `surrogate_key()` macro for generating consistent primary keys.

- **Profiles Configuration:** Utilize `profiles.yml` to manage database connection details securely and separately from the model code.

Data Ingestion Process Overview

The planned workflow begins with data extraction and loading:

- The Python script (`crossref_opencitations_load.py`) was designed to fetch data from Crossref/OpenCitations APIs.
- Its role was to perform minimal initial parsing (e.g., extracting citation lists, basic metadata) and load the raw payloads into the `raw.raw_crossref` table in PostgreSQL (the 'E' and 'L' in ELT).
- The script was planned to trigger the DBT transformation pipeline (`dbt run`) upon successful data loading and commit.

DBT Model Implementation Strategy

The transformation logic ('T' in ELT) was designed to be organized into a sequence of dependent DBT models, progressing from raw data to the final structured tables in the `dwh` schema.

- **Layering Approach:**
 - *Staging Models* (`stg_*.sql`): Purpose: To select from raw source tables, perform initial cleaning (trimming, casing), basic type casting, and column renaming. Designed to provide a consistent base for downstream models.
 - *Intermediate Models* (`int_*.sql`): Purpose: To implement core business logic, structure data into entities (e.g., separate authors, organizations), handle complex transformations (like parsing JSON arrays), generate surrogate keys, and join intermediate entities where necessary.
 - *Target Models* (`dwh_*.sql`): Purpose: To select from relevant intermediate models, perform final joins to resolve foreign keys, and load data into the final `dwh` schema tables.
- **Incremental Processing:** Target models were planned to be materialized as incremental, using a `delete+insert` strategy based on an `updated_at`

timestamp. This aimed to efficiently process only new or changed data during subsequent runs.

- **Presentation View** (`entry_details_vw.sql`): Planned to create a denormalized view by joining relevant `dwh` tables. The purpose was to simplify data retrieval for the frontend application by providing a single interface, avoiding complex joins in the application code, and to structure data appropriately to support the planned complex display requirements of the frontend, such as providing necessary fields for detailed views including potentially aggregated author information.

DBT Testing Strategy

To ensure data quality and reliability (a key non-functional requirement), a robust testing strategy using DBT's capabilities was planned:

Generic Schema Tests: Intended to leverage built-in DBT tests defined in `schema.yml` file associated with the target models. This included planning tests for:

- `unique` and `not_null` constraints on primary keys and other mandatory columns.
- relationships tests to enforce referential integrity between tables (foreign key checks).
- Potentially `accepted_values` for columns with controlled vocabularies (like `entry_type_name`).

Singular Tests (Custom Logic): Planned to implement custom SQL queries (saved as `.sql` files in the tests directory) to validate specific business rules or data quality aspects pertinent to bibliographic data. Examples considered included checks for reasonable publication dates, valid DOI/ORCID formats (if present), and consistency rules (e.g., ensuring an entry with authors has a primary author marked). Each test query was designed to return failing rows, passing only if zero rows are returned.

DBT Workflow Execution Plan

The standard development and update workflow involves executing the DBT commands sequentially:

1. `dbt run`: Executes all defined models (`.sql` files) in the correct dependency order, applying materialization strategies (e.g., incremental updates for target tables).
2. `dbt test`: Executes all defined tests (`schema.yml` tests and custom `.sql` tests).

As implemented in this project's data loading script (`crossref_opencitations_load.py`), the `trigger_dbt()` function automatically executes `dbt run` after new data is successfully loaded into `raw.raw_crossref` table. Then `dbt test` execution would typically be run immediately after `dbt run` in the script, or in a manual workflow or an automated orchestration setup (e.g., using cron, Airflow) to ensure data quality before the data is consumed by downstream applications like the Flask frontend.

3.5 Application Tier Design Strategy

The Presentation Layer was planned as a web application utilizing the Flask framework for backend logic and standard web technologies (HTML, CSS, JavaScript) for the frontend interface. The design strategy focused on creating an accessible and interactive user experience while maintaining a clear separation between the primary user interface and a distinct API endpoint for programmatic access.

Backend Service Design (Flask)

The core strategy for the Flask backend (`app.py`) involved acting as an intermediary, handling user requests, querying the database, processing data, and serving it to the frontend. Two distinct service patterns were planned:

UI Rendering Logic: The primary route (`/`) was designed to manage the main user interface. The strategy involved:

1. Processing GET requests containing parameters for text search (title, author), publication type filtering (`selected_types` list), and pagination controls (`page`, `per_page`).
2. Interacting with the PostgreSQL database (`psycopg2`) by querying the pre-joined `dwh.entry_details_vw`.
3. Employing a query strategy that dynamically constructed `WHERE` clauses based on user input (using `ILIKE` and `ANY`). A key part of the strategy was to leverage database aggregation features (specifically `json_agg` within CTEs) to efficiently bundle related information, such as all authors for an entry, into a structured JSON format suitable for passing to the frontend template. This aimed to minimize complex data manipulation in the application layer.
4. Fetching summary counts for publication types (total vs. filtered) using separate `GROUP BY` queries to support the filter interface.
5. Rendering the main HTML template (`index.html`) via Jinja2, passing the processed query results (including the aggregated JSON data), type counts, and current request parameters.

JSON API Endpoint logic: (`/api/entries`): A separate GET endpoint was planned to provide basic programmatic data access. The strategy for this endpoint was simpler:

- Accept basic search parameters (`q`, title, author) and a page number.
- Execute a less complex query against `dwh.entry_details_vw`, using `ILIKE` and simple `DISTINCT ON` logic.
- Omit the advanced type filtering planned for the main UI.
- Return results directly as a JSON array.

Frontend Interface Design (`index.html`, CSS, JavaScript):

The frontend design strategy focused on usability and effective data presentation using standard web technologies.

Structure and Components: The interface (`index.html`) was planned using HTML, structured to include:

1. A primary search form with inputs for title and author.
2. A dropdown menu (`<select>`) enabling user selection of results per page.
3. A dedicated section for publication type filtering, designed with checkboxes (`<input type="checkbox">`), including an “All Types” control and display areas for contextual type counts.
4. A main results display area, envisioned as a table (`<table>`), showing key bibliographic fields. The design included handling long text via CSS-based truncation combined with interactive “Show full” elements.
5. An expandable details section associated with each result row, planned to contain the full aggregated author list, an interactive abstract, linked citations/cited-by DOIs, and other metadata fields.
6. Standard pagination controls (Previous/Next links).

Styling and Interaction:

- The strategy involved using CSS for visual styling, layout management (including responsive design considerations), and implementing features like text truncation and a background theme.
- Client-side JavaScript was planned to enhance interactivity, specifically for managing the logic of the type filter checkboxes (coordinating “All Types” with individual selections) and implementing the “Show more”/ “Show less” toggle functionality for the abstract display within the details section. Static assets like CSS and images were planned to be served from a dedicated static folder managed by Flask.

3.6 Research Evaluation Methods

This section outlines the planned approach for evaluating the implemented system (presented in Chapter 4) against the research questions (Section 1.2). Due to the prototype nature of the system developed using sample data on a personal computer, the evaluation will primarily rely on **qualitative assessment** of the system's design, features, and observed behaviour, rather than quantitative benchmarking.

The evaluation will address the research questions as follows:

- **RQ1 (Schema Design):** Assess the effectiveness of the implemented `dwh` schema in representing bibliographic metadata and its potential scalability based on the chosen normalization strategy and physical design features (partitioning, indexing), referencing the design rationale (Sec 3.2).
- **RQ2 & RQ4 (DBT Integration & Challenges):** Evaluate the DBT approach based on the implemented pipeline (Sec 4.3) by qualitatively assessing its maintainability (modularity, versioning), flexibility, and data quality benefits (via testing) compared conceptually to traditional ETL. Practical challenges encountered during implementation (e.g., handling JSON, incremental logic) will also be discussed.
- **RQ3 (Three-Tier Architecture):** Evaluate the implemented architecture based on its success in achieving modularity (separation of concerns) and providing accessibility (via UI and API). Potential scalability and performance will be discussed qualitatively based on design choices and observed prototype behaviour.

4 Results

This chapter presents the concrete outcomes and findings derived from the implementation phase of the bibliographic database system, following the design principles and methodology detailed in Chapter 3. The results described herein pertain to the functional prototype developed using sample data ingested from Crossref and OpenCitations, operating within the specific development environment outlined in Section 3.2. This chapter will detail the final implemented system components, the structure of the database as realized, the execution and validation of the data transformation pipeline, and the functionality of the application tier, concluding with an evaluation of these results against the research objectives set forth in Chapter 1.

4.1 Implemented System Components

The development effort successfully culminated in the implementation of the planned three-tier architecture, visually represented for this specific project in Figure 2. The final system comprises the following operational components, functioning cohesively as designed:

1. **Data Tier:** A PostgreSQL database instance was configured, containing three distinct schemas relevant to the ELT process:
 - **raw schema:** Housing the `raw_crossref` table, which serves as the initial landing zone for data fetched from the external APIs.
 - **dbt schema:** Containing tables generated by DBT's intermediate models (`int_*.sql`), holding structured data after initial transformations but before final loading.
 - **dwh schema:** Containing the set of normalized relational tables (`entry`, `author`, `organization`, `entry_type`, `entry_author`, `author_organization`) populated by the transformation process, along with the `entry_details_vw` presentation view designed for application access.
2. **Logic Tier:** This layer consists of two key functional parts:

- The Python ingestion script (`crossref_opencitations_load.py`), which successfully fetches data from Crossref and OpenCitations APIs based on user input, performs necessary initial processing, loads the raw data into the `raw.raw_crossref` table, and subsequently triggers the DBT pipeline execution.
 - The Data Build Tool (DBT) project, encompassing a collection of SQL models and associated tests, which manages the Extract-Load-Transform (ELT) process, transforming data from the `raw` schema into the structured `dwh` schema.
3. **Presentation Tier:** A Flask web application (`app.py`) was implemented, providing:
- A web user interface rendered via `index.html` for searching and displaying bibliographic records, incorporating CSS for styling and client-side JavaScript for enhanced interactivity (e.g., type filtering, abstract toggling).
 - A functional JSON API endpoint (`/api/entries`) enabling programmatic data access with search and pagination features.

These implemented components collectively form the working prototype evaluated in the subsequent sections.

4.2 Database Implementation Results

The PostgreSQL database was implemented successfully, realizing the logical and physical design strategies outlined in the methodology (Section 3.3).

Logical Schema Implementation:

The final database schema implemented within the `dwh` namespace accurately reflects the Entity-Relationship Diagram presented in the Figure 3. The core tables (`entry`, `author`, `organization`, `entry_type`) and associative tables (`entry_author`, `author_organization`) were created with their respective attributes and relationships, adhering to the principles of Third Normal Form (3NF). Surrogate keys (e.g., `entry_id`, `author_id`), generated deterministically by the DBT `surrogate_key` macro, serve as primary keys, facilitating efficient joins and data management. Key natural identifiers like `doi`, `orcid`,

organization.name, and entry_type.entry_type_name were retained as columns with appropriate UNIQUE constraints enforced to maintain data integrity and allow for external referencing. The overall structure in pgAdmin database administration tool, as visualized in Figure 5, demonstrates the normalized, relational model designed to minimize redundancy and represent bibliographic entities effectively.

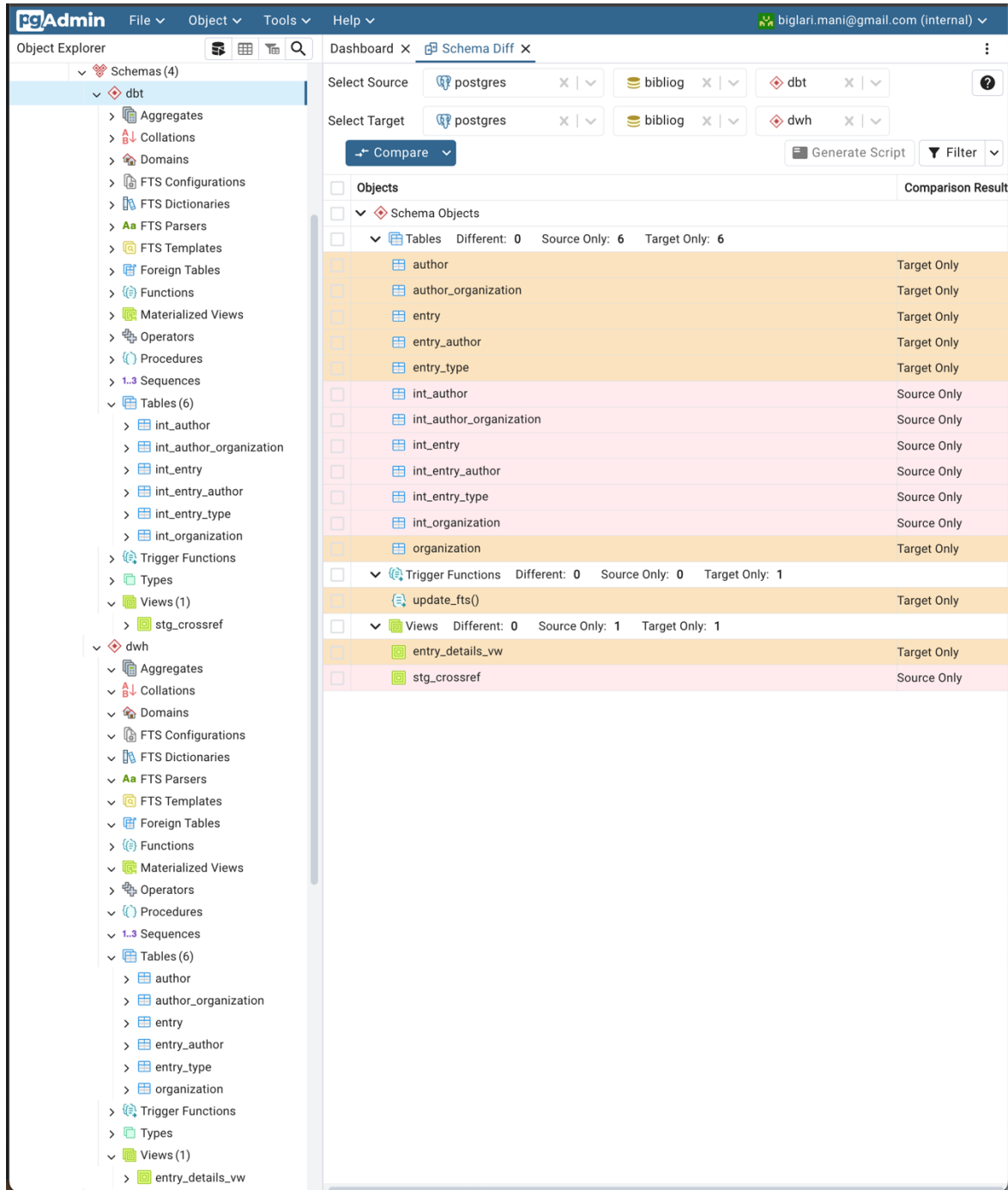


Figure 5. Structure of the dbt and dwh schemas in pgAdmin.

Physical Schema Implementation:

The complete Data Definition Language (DDL) script detailing all tables, columns, data types, constraints and optimizations is provided in Appendix 2.

Specific physical storage and access optimization features, planned in 3.3, were implemented to enhance performance and scalability:

1. **Partitioning:** PostgreSQL's declarative partitioning was successfully applied to the tables anticipated to grow largest:
 - `dwh.entry`: This table was partitioned by **RANGE** on the `publication_date` column. A PL/pgSQL script loop (included in Appendix 2) generated yearly partitions (e.g., `entry_y2020`, `entry_y2021`, ..., `entry_y2030`) along with a `entry_legacy` partition for pre-2000 entries. This physical structure, allows the query planner to potentially prune partitions based on date filters, improving query efficiency.
 - `dwh.entry_author`: This bridge table was partitioned by **HASH** on the `entry_id` column into 32 distinct partitions (`entry_author_p0` to `entry_author_p31`), distributing the author-entry links across multiple physical tables to potentially improve load balancing and join performance. The DDL for creating these partitions is also in Appendix 2.
2. **Indexing:** A suite of indexes was created to optimize common data retrieval operations:
 - Primary Key indexes were automatically created on the surrogate keys of all tables.
 - Foreign Key indexes were created on columns like `entry.entry_type_id`, `entry_author.author_id`, `author_organization.organization_id`, etc., to accelerate join operations.

- **GIN Trigram Indexes:** Crucially for search functionality, GIN indexes using the `pg_trgm` extension were implemented on `entry.title` (`idx_entry_title_trgm`) and `author.name` (`idx_author_name_trgm`). These indexes are specifically designed to provide efficient support for the case-insensitive substring (`ILIKE '%term%'`) searches performed by the application tier.
 - **Other Indexes:** An index on `entry(publication_date DESC)` (`idx_entry_recent`) was added to optimize default sorting by newest publication. A covering index `idx_entry_author_lookup` on `entry_author(entry_id, author_id)` was created to potentially enable index-only scans for queries retrieving both keys.
3. **Constraints:** Primary Key, UNIQUE (on `doi`, `orcid`, `organization.name`, `entry_type.name`), and Foreign Key constraints were implemented as defined in the logical schema. As planned, Foreign Key constraints were defined using `DEFERRABLE INITIALLY IMMEDIATE` to allow constraint validation at the end of DBT's transaction, potentially improving bulk insert performance.
 4. **Full-Text Search Feature:** A `tsvector` column named `fts` was added to the `dwh.entry` table. A trigger function (`dwh.update_fts`) and an associated trigger (`trg_update_fts`) were implemented to automatically populate this column with a concatenated vector of the entry's title and abstract upon insertion or update. A GIN index (`idx_entry_fts`) was created on this `fts` column to enable efficient PostgreSQL full-text search capabilities.
 5. **Storage Parameters:** For the prototype development using limited data, default PostgreSQL storage parameters were generally sufficient. The primary adjustment was setting `work_mem` to 128MB within the Docker container's PostgreSQL configuration to provide more memory for sorting and hashing operations during DBT model runs. Further tuning (e.g., `fillfactor`) would require analysis under realistic load conditions.

4.3 Data Transformation Pipeline Results (DBT)

The transformation pipeline, orchestrated using DBT, was successfully implemented and executed, transforming the raw data loaded by the Python script into the structured `dwh` schema.

Workflow Execution and Lineage:

The DBT project, containing models for staging, intermediate processing, and final target table loading, executed without errors. The `dbt run` command successfully materialized all models in the correct sequence based on their dependencies defined using the `ref()` function. The visual representation of these dependencies and the overall data flow is confirmed by the DBT-generated lineage graph presented in Figure 7, which accurately depicts the progression from the `raw.raw_crossref` source through the various `stg`, `int`, and `dwh` models to the final `entry_details_vw` model. Evidence of a successful pipeline execution, including model completion status and timings for the sample dataset, is shown in the terminal output captured in Figure 6.

```
manibiglari@Manis-MacBook-Pro bibliographic_dbt % dbt run
/Users/manibiglari/Desktop/bibliographic_db_project/dbt-env/lib/python3.9/site-packages/urllib3/_init_.py:35: Not
OpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.
3'. See: https://github.com/urllib3/urllib3/issues/3020
warnings.warn(
13:11:17 Running with dbt=1.9.1
13:11:17 Registered adapter: postgres=1.9.0
13:11:18 Found 14 models, 32 data tests, 3 sources, 670 macros
13:11:18
13:11:18 Concurrency: 4 threads (target='dev')
13:11:18
13:11:18 1 of 14 START sql view model dbt.stg_crossref ..... [RUN]
13:11:18 1 of 14 OK created sql view model dbt.stg_crossref ..... [CREATE VIEW in 0.07s]
13:11:18 2 of 14 START sql table model dbt.int_author ..... [RUN]
13:11:18 3 of 14 START sql table model dbt.int_entry ..... [RUN]
13:11:18 4 of 14 START sql table model dbt.int_entry_author ..... [RUN]
13:11:18 5 of 14 START sql table model dbt.int_entry_type ..... [RUN]
13:11:18 5 of 14 OK created sql table model dbt.int_entry_type ..... [SELECT 0 in 0.09s]
13:11:18 2 of 14 OK created sql table model dbt.int_author ..... [SELECT 267 in 0.10s]
13:11:18 6 of 14 START sql table model dbt.int_organization ..... [RUN]
13:11:18 3 of 14 OK created sql table model dbt.int_entry ..... [SELECT 400 in 0.10s]
13:11:18 4 of 14 OK created sql table model dbt.int_entry_author ..... [SELECT 351 in 0.10s]
13:11:18 7 of 14 START sql incremental model dwh.entry_type ..... [RUN]
13:11:18 8 of 14 START sql incremental model dwh.author ..... [RUN]
13:11:18 9 of 14 START sql table model dbt.int_author_organization ..... [RUN]
13:11:18 9 of 14 OK created sql table model dbt.int_author_organization ..... [SELECT 115 in 0.07s]
13:11:18 7 of 14 OK created sql incremental model dwh.entry_type ..... [INSERT 0 0 in 0.14s]
13:11:18 10 of 14 START sql incremental model dwh.entry ..... [RUN]
13:11:18 8 of 14 OK created sql incremental model dwh.author ..... [INSERT 0 267 in 0.14s]
13:11:18 11 of 14 START sql incremental model dwh.entry_author ..... [RUN]
13:11:18 11 of 14 OK created sql incremental model dwh.entry_author ..... [INSERT 0 386 in 0.10s]
13:11:18 10 of 14 OK created sql incremental model dwh.entry ..... [INSERT 0 400 in 0.12s]
13:11:18 6 of 14 OK created sql table model dbt.int_organization ..... [SELECT 79 in 0.27s]
13:11:18 12 of 14 START sql incremental model dwh.organization ..... [RUN]
13:11:18 12 of 14 OK created sql incremental model dwh.organization ..... [INSERT 0 79 in 0.03s]
13:11:18 13 of 14 START sql incremental model dwh.author_organization ..... [RUN]
13:11:18 13 of 14 OK created sql incremental model dwh.author_organization ..... [INSERT 0 115 in 0.03s]
13:11:18 14 of 14 START sql view model dwh.entry_details_vw ..... [RUN]
13:11:18 14 of 14 OK created sql view model dwh.entry_details_vw ..... [CREATE VIEW in 0.04s]
13:11:18
13:11:18 Finished running 6 incremental models, 6 table models, 2 view models in 0 hours 0 minutes and 0.72 second
s (0.72s).
13:11:18
13:11:18 Completed successfully
13:11:18
13:11:18 Done. PASS=14 WARN=0 ERROR=0 SKIP=0 TOTAL=14
manibiglari@Manis-MacBook-Pro bibliographic_dbt %
```

Figure 6. Terminal Output of `dbt run`.

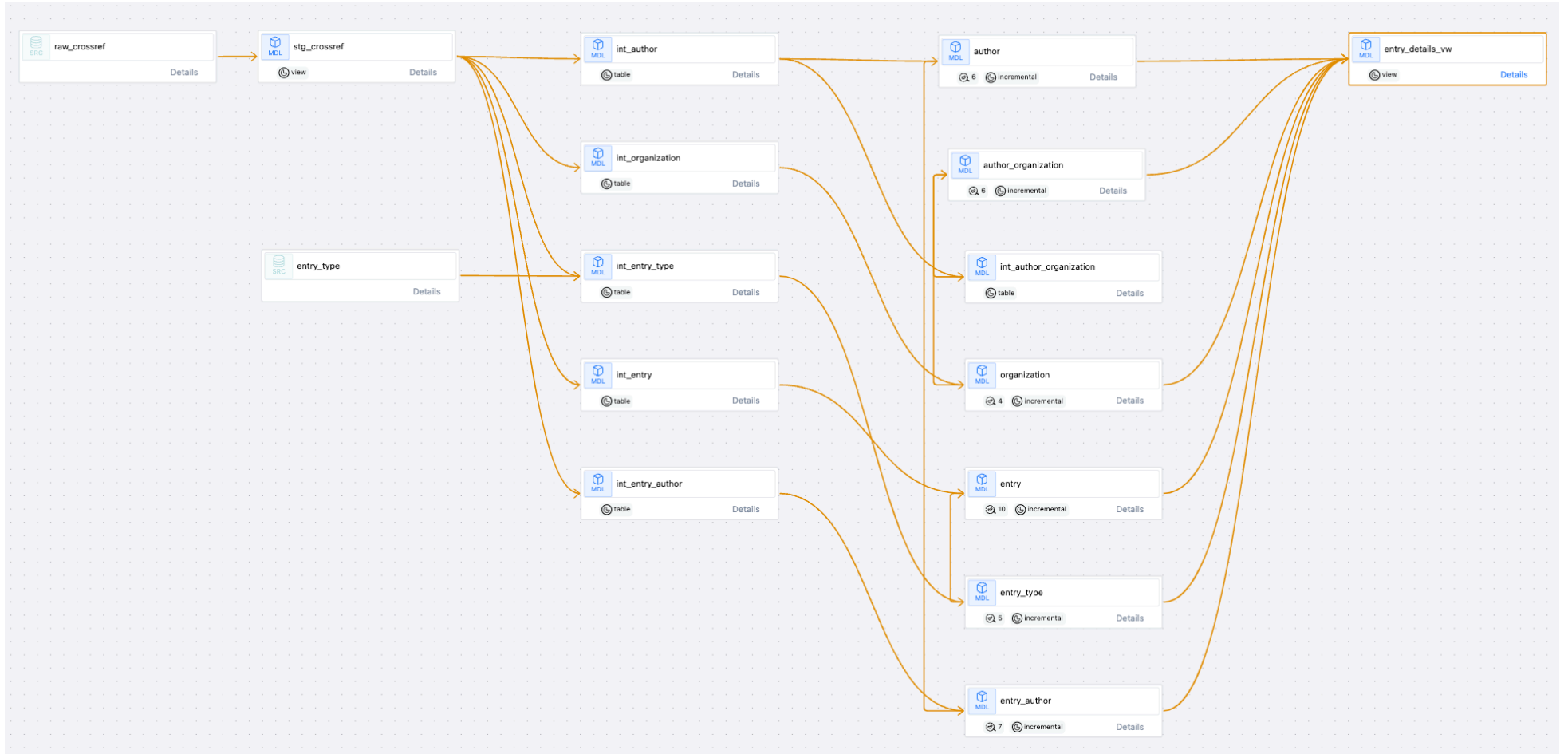


Figure 7. DBT Data Lineage Graph.

Model Implementation Details:

The implemented DBT models successfully executed the transformation logic planned in Section 3.4:

1. **Staging** (`stg_crossref.sql`): This model correctly selected data from the `raw.raw_crossref` source, applied basic cleaning functions like `TRIM()` for whitespace removal and `LOWER()` for standardizing casing on fields like `type` and `language`, performed necessary type casting (e.g., `issued::DATE AS publication_date`), and handled potential null values using `COALESCE` (e.g., `COALESCE(doi, 'unknown')`).
2. **Intermediate Models** (`int_*.sql`): These models handled the core structuring and business logic:

JSON Parsing: Models processing authors and organizations (e.g., `int_author.sql`, `int_entry_author.sql`, `int_organization.sql`, `int_author_organization.sql`) effectively utilized PostgreSQL's `jsonb_array_elements` function to flatten nested JSON arrays (representing authors and their affiliations) from the staging layer into distinct relational rows. Where necessary (e.g., in `int_entry.sql` for abstracts, `int_author_organization.sql` for names), `REGEXP_REPLACE` was used to strip potential HTML tags embedded in text fields.

Surrogate Key Generation: The custom `surrogate_key()` macro (defined in `macros/surrogate_key.sql`) was consistently invoked in models like `int_entry`, `int_author`, `int_organization`, etc., to generate unique, deterministic 64-bit integer surrogate keys based on combinations of relevant business attributes (e.g., `{{ surrogate_key(["name", "orcid", "organizations"]) }}` for `author_id`).

Entity Structuring & Logic: Model `int_entry_type.sql` successfully identified distinct publication type strings from the staging data and generated

new sequential `entry_type_ids` only for types not already present in the `dwh.entry_type` table (by querying it as a source). `int_entry_author.sql` correctly associated entries with authors and captured the author sequence using `jsonb_array_elements()` WITH ORDINALITY.

3. Target Models (`dwh.*.sql`):

Incremental Loading: All models populating the main `dwh` tables (`entry.sql`, `author.sql`, etc.) were configured with `materialized='incremental'` and `incremental_strategy='delete+insert'`. They incorporated the necessary `{% if is_incremental() %}` blocks to filter incoming data based on the `updated_at` timestamp, comparing it against the maximum `updated_at` value currently in the target table (`WHERE updated_at > (SELECT COALESCE(MAX(updated_at), '1900-01-01') FROM {{ this }})`). This ensures that only new or modified records are processed during subsequent runs.

Foreign Key Resolution: These models performed the final required joins between intermediate models to resolve foreign key relationships before inserting data into the target tables (e.g., `entry.sql` joins `int_entry` with `entry_type` on `entry_type_name` to get the correct `entry_type_id`).

Presentation View (`entry_details_vw.sql`): This model was successfully implemented and materialized as a PostgreSQL VIEW. It performs the necessary `LEFT JOIN` operations across all primary and associative tables in the `dwh` schema (`entry`, `entry_type`, `entry_author`, `author`, `author_organization`, `organization`) to provide a single, denormalized interface for the Flask application.

Testing Results:

The comprehensive testing strategy planned in Section 3.4 was implemented using DBT's testing framework to validate the quality and integrity of the data within the final `dwh` schema.

- *Generic Schema Tests:* As defined in `models/target/dwh_schema.yml` (see Appendix 3), tests for `unique` and `not_null` were applied to all primary keys and critical metadata fields. `relationships` tests were configured to enforce referential integrity for all foreign key relationships defined in the logical schema.
- *Singular Tests (Custom):* Specific SQL-based tests were created in the `tests/` directory (see Appendix 4) to validate domain-specific rules: `assert_entry_publication_date_is_reasonable.sql` (checking date ranges), `assert_doi_format_if_present.sql` (validating '10.' prefix), `assert_author_orcid_format.sql` (checking ORCID pattern via regex), and `assert_entry_has_primary_author.sql` (ensuring entries with authors have a primary author marked).
- **Execution Outcome:** The `dbt test` command was executed after `dbt run` during the pipeline execution. All the defined tests (combining generic and singular tests across all models and columns) passed successfully against the populated `dwh` schema using the sample dataset. The result of this successful validation is shown in Figure 8. This outcome confirms that the implemented transformation logic produced data that adheres to the defined structural constraints, referential integrity rules, and custom quality checks for the processed data.

```

manibiglari@Manis-MacBook-Pro bibliographic_dbt % dbt test
/Users/manibiglari/Desktop/bibliographic_db_project/dbt-env/lib/python3.9/site-packages/urllib3/__init__.py:35: NotOpen
nSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. Se
e: https://github.com/urllib3/urllib3/issues/3020
warnings.warn(
17:34:00 Running with dbt=1.9.1
17:34:00 Registered adapter: postgres=1.9.0

17:34:00 Found 14 models, 32 data tests, 3 sources, 670 macros
17:34:00
17:34:00 Concurrency: 4 threads (target='dev')
17:34:00
17:34:01 1 of 32 START test assert_author_orcid_format ..... [RUN]
17:34:01 2 of 32 START test assert_doi_format_if_present ..... [RUN]
17:34:01 3 of 32 START test assert_entry_has_primary_author ..... [RUN]
17:34:01 4 of 32 START test assert_entry_publication_date_is_reasonable ..... [RUN]
17:34:01 1 of 32 PASS assert_author_orcid_format ..... [PASS in 0.05s]
17:34:01 5 of 32 START test not_null_author_author_id ..... [RUN]
17:34:01 4 of 32 PASS assert_entry_publication_date_is_reasonable ..... [PASS in 0.05s]
17:34:01 2 of 32 PASS assert_doi_format_if_present ..... [PASS in 0.06s]
17:34:01 3 of 32 PASS assert_entry_has_primary_author ..... [PASS in 0.06s]
17:34:01 6 of 32 START test not_null_author_name ..... [RUN]
17:34:01 8 of 32 START test not_null_author_organization_author_id ..... [RUN]
17:34:01 8 of 32 START test not_null_author_organization_author_organization_surrogate_id [RUN]
17:34:01 5 of 32 PASS not_null_author_author_id ..... [PASS in 0.04s]
17:34:01 9 of 32 START test not_null_author_organization_organization_id ..... [RUN]
17:34:01 6 of 32 PASS not_null_author_name ..... [PASS in 0.04s]
17:34:01 8 of 32 PASS not_null_author_organization_author_organization_surrogate_id .... [PASS in 0.03s]
17:34:01 7 of 32 PASS not_null_author_organization_author_id ..... [PASS in 0.04s]
17:34:01 10 of 32 START test not_null_entry_author_author_id ..... [RUN]
17:34:01 11 of 32 START test not_null_entry_author_entry_author_surrogate_id ..... [RUN]
17:34:01 12 of 32 START test not_null_entry_author_entry_id ..... [RUN]
17:34:01 9 of 32 PASS not_null_author_organization_organization_id ..... [PASS in 0.03s]
17:34:01 13 of 32 START test not_null_entry_entry_id ..... [RUN]
17:34:01 11 of 32 PASS not_null_entry_author_entry_author_surrogate_id ..... [PASS in 0.03s]
17:34:01 14 of 32 START test not_null_entry_type_id ..... [RUN]
17:34:01 12 of 32 PASS not_null_entry_author_entry_id ..... [PASS in 0.04s]
17:34:01 10 of 32 PASS not_null_entry_author_author_id ..... [PASS in 0.04s]
17:34:01 15 of 32 START test not_null_entry_publication_date ..... [RUN]
17:34:01 13 of 32 PASS not_null_entry_entry_id ..... [PASS in 0.02s]
17:34:01 16 of 32 START test not_null_entry_title ..... [RUN]
17:34:01 17 of 32 START test not_null_entry_type_entry_type_id ..... [RUN]
17:34:01 14 of 32 PASS not_null_entry_entry_type_id ..... [PASS in 0.03s]
17:34:01 18 of 32 START test not_null_entry_type_entry_type_name ..... [RUN]
17:34:01 17 of 32 PASS not_null_entry_type_entry_type_id ..... [PASS in 0.03s]
17:34:01 16 of 32 PASS not_null_entry_title ..... [PASS in 0.04s]
17:34:01 15 of 32 PASS not_null_entry_publication_date ..... [PASS in 0.04s]
17:34:01 19 of 32 START test not_null_organization_id ..... [RUN]
17:34:01 20 of 32 START test not_null_organization_name ..... [RUN]
17:34:01 21 of 32 START test relationships_author_organization_author_id_author_id_ref_author_ [RUN]
17:34:01 18 of 32 PASS not_null_entry_type_entry_name ..... [PASS in 0.03s]
17:34:01 22 of 32 START test relationships_author_organization_organization_id_id_ref_organization_ [RUN]
17:34:01 19 of 32 PASS not_null_organization_id ..... [PASS in 0.04s]
17:34:01 21 of 32 PASS relationships_author_organization_author_id_author_id_ref_author_ [PASS in 0.03s]
17:34:01 20 of 32 PASS not_null_organization_name ..... [PASS in 0.03s]
17:34:01 23 of 32 START test relationships_entry_author_author_id_author_id_ref_author_ [RUN]
17:34:01 24 of 32 START test relationships_entry_author_entry_id_entry_id_ref_entry_.. [RUN]
17:34:01 25 of 32 START test relationships_entry_entry_type_id_entry_type_id_ref_entry_type_ [RUN]
17:34:01 22 of 32 PASS relationships_author_organization_organization_id_id_ref_organization_ [PASS in 0.03s]
17:34:01 26 of 32 START test unique_author_author_id ..... [RUN]
17:34:01 24 of 32 PASS relationships_entry_author_entry_id_entry_id_ref_entry_ ..... [PASS in 0.07s]
17:34:01 25 of 32 PASS relationships_entry_entry_type_id_entry_type_id_ref_entry_type_ ..... [PASS in 0.06s]
17:34:01 26 of 32 PASS unique_author_author_id ..... [PASS in 0.05s]
17:34:01 23 of 32 PASS relationships_entry_author_author_id_author_id_ref_author_ ..... [PASS in 0.07s]
17:34:01 27 of 32 START test unique_author_organization_author_organization_surrogate_id [RUN]
17:34:01 28 of 32 START test unique_entry_author_entry_author_surrogate_id ..... [RUN]
17:34:01 29 of 32 START test unique_entry_entry_id ..... [RUN]
17:34:01 30 of 32 START test unique_entry_type_entry_type_id ..... [RUN]
17:34:01 27 of 32 PASS unique_author_organization_author_organization_surrogate_id ..... [PASS in 0.04s]
17:34:01 31 of 32 START test unique_entry_type_entry_type_name ..... [RUN]
17:34:01 28 of 32 PASS unique_entry_author_entry_author_surrogate_id ..... [PASS in 0.03s]
17:34:01 30 of 32 PASS unique_entry_type_entry_type_id ..... [PASS in 0.03s]
17:34:01 29 of 32 PASS unique_entry_entry_id ..... [PASS in 0.04s]
17:34:01 32 of 32 START test unique_organization_id ..... [RUN]
17:34:01 31 of 32 PASS unique_entry_type_entry_type_name ..... [PASS in 0.02s]
17:34:01 32 of 32 PASS unique_organization_id ..... [PASS in 0.02s]
17:34:01
17:34:01 Finished running 32 data tests in 0 hours 0 minutes and 0.59 seconds (0.59s).
17:34:01
17:34:01 Completed successfully
17:34:01
17:34:01 Done. PASS=32 WARN=0 ERROR=0 SKIP=0 TOTAL=32
manibiglari@Manis-MacBook-Pro bibliographic_dbt % _

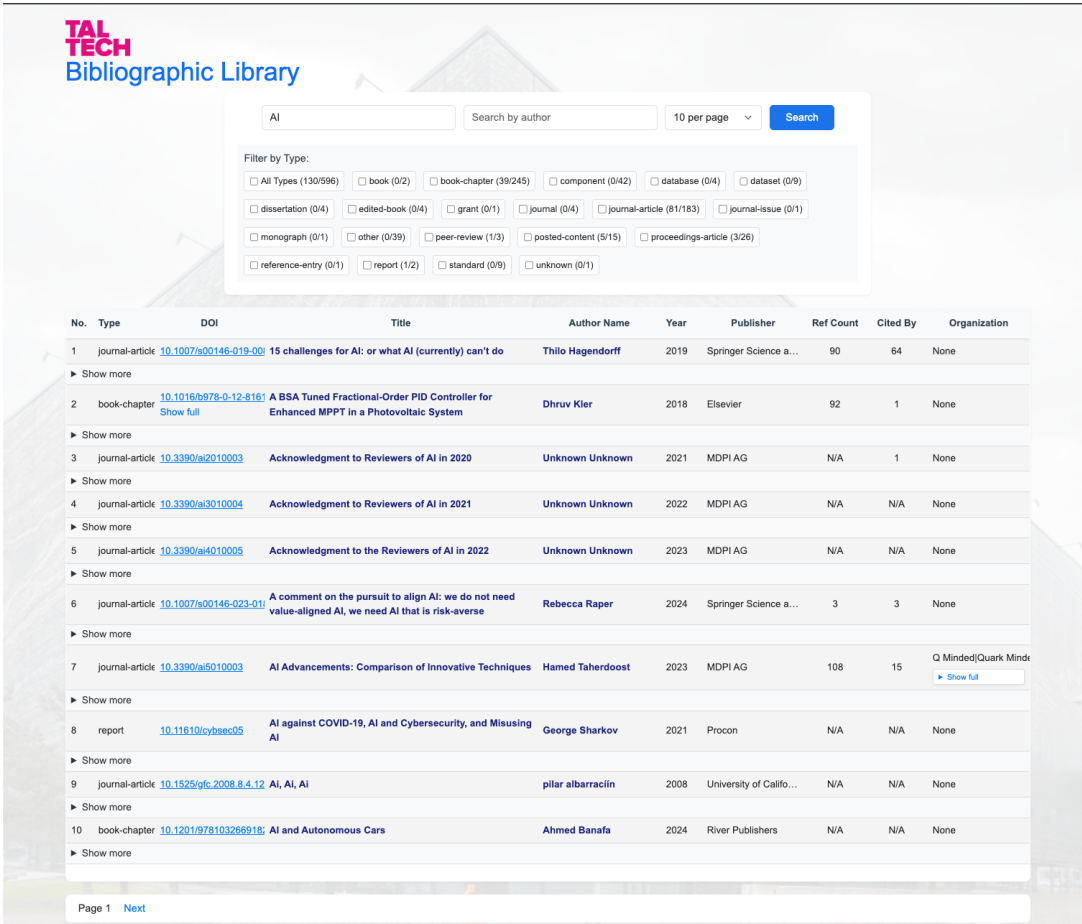
```

Figure 8. Terminal Output of dbt test.

4.4 Application Tier Results

The Presentation Tier, consisting of the Flask backend (`app.py`) and the interactive web frontend (`index.html`), was successfully implemented, providing functional access to the bibliographic data processed by the lower tiers and delivering the enhanced user experience features.

As shown in the Figure 9, the Flask application runs without errors and provides the core functionalities. The primary web user interface, rendered via `index.html`, allows users to effectively interact with the system. Search inputs for title and author are functional. The implemented **Type Filtering** section enables users to refine results using checkboxes, with client-side JavaScript managing the “All Types” selection logic. Associated filtered and **total counts for each type** are accurately displayed based on backend queries. Furthermore, the **results-per-page** dropdown was successfully implemented, allowing users to control pagination (10, 25, 50, or 100 results).



The screenshot displays the 'Bibliographic Library' application interface. At the top left is the logo 'TAL TECH Bibliographic Library'. The search area includes a search box with the text 'AI', a 'Search by author' field, a '10 per page' dropdown menu, and a 'Search' button. Below the search area is a 'Filter by Type:' section with a grid of checkboxes and their corresponding counts: All Types (130/596), book (0/2), book-chapter (39/245), component (0/42), database (0/4), dataset (0/9), dissertation (0/4), edited-book (0/4), grant (0/1), journal (0/4), journal-article (81/183), journal-issue (0/1), monograph (0/1), other (0/39), peer-review (1/3), posted-content (5/15), proceedings-article (3/26), reference-entry (0/1), report (1/2), standard (0/9), and unknown (0/1). Below the filters is a table of search results with columns: No., Type, DOI, Title, Author Name, Year, Publisher, Ref Count, Cited By, and Organization. The table contains 10 rows of results, each with a 'Show more' link. At the bottom left, it shows 'Page 1' and a 'Next' link.

No.	Type	DOI	Title	Author Name	Year	Publisher	Ref Count	Cited By	Organization
1	journal-article	10.1007/s00146-019-000	15 challenges for AI: or what AI (currently) can't do	Thilo Hagendorff	2019	Springer Science a...	90	64	None
▶ Show more									
2	book-chapter	10.1016/b978-0-12-816	A BSA Tuned Fractional-Order PID Controller for Enhanced MPPT in a Photovoltaic System	Dhruv Kler	2018	Elsevier	92	1	None
▶ Show more									
3	journal-article	10.3390/ai2010003	Acknowledgment to Reviewers of AI in 2020	Unknown Unknown	2021	MDPI AG	N/A	1	None
▶ Show more									
4	journal-article	10.3390/ai3010004	Acknowledgment to Reviewers of AI in 2021	Unknown Unknown	2022	MDPI AG	N/A	N/A	None
▶ Show more									
5	journal-article	10.3390/ai4010005	Acknowledgment to the Reviewers of AI in 2022	Unknown Unknown	2023	MDPI AG	N/A	N/A	None
▶ Show more									
6	journal-article	10.1007/s00146-023-011	A comment on the pursuit to align AI: we do not need value-aligned AI, we need AI that is risk-averse	Rebecca Raper	2024	Springer Science a...	3	3	None
▶ Show more									
7	journal-article	10.3390/ai5010003	AI Advancements: Comparison of Innovative Techniques	Hamed Taherdoost	2023	MDPI AG	108	15	Q Minded Quark Minded
▶ Show more									
8	report	10.11610/cybssec05	AI against COVID-19, AI and Cybersecurity, and Misusing AI	George Sharkov	2021	Procon	N/A	N/A	None
▶ Show more									
9	journal-article	10.1525/ycf.2008.8.4.12	AI, AI, AI	pilar albarracin	2008	University of Califo...	N/A	N/A	None
▶ Show more									
10	book-chapter	10.1201/978103266916	AI and Autonomous Cars	Ahmed Banafa	2024	River Publishers	N/A	N/A	None
▶ Show more									

Page 1 [Next](#)

Figure 9. Interactive faceted-search interface of the Bibliographic Library application.

Search results are presented clearly in the main HTML table. Techniques for handling long text fields (DOI, Title, Organization) were implemented using CSS truncation combined with expandable `<details>` elements (“Show full”), which function correctly. A significant feature as presented in Figure 10, is the implemented “Show more” expandable section for each entry. This section successfully reveals comprehensive details fetched from the database, including the full list of authors (aggregated using `json_agg` in the backend query), with the primary author marked and ORCID links provided where available. The interactive abstract display was realized, showing an initial preview and using a JavaScript toggle button (“Show more”/ “Show less”) to correctly control the visibility of the full text. Linked DOIs for citation and cited-by data are presented as functional hyperlinks, alongside other relevant metadata fields. Navigation through result pages using the “Previous” and “Next” links is operational.

No.	Type	DOI	Title	Author Name	Year	Publisher	Ref Count	Cited By	Organization
7	journal-article	10.3390/ai5010003	AI Advancements: Comparison of Innovative Techniques	Hamed Taherdoost	2023	MDPI AG	108	15	Q Minded Quark Mind
<div style="border: 1px solid #ccc; padding: 5px;"> <p>▼ Show more</p> <p>Abstract:</p> <p>In recent years, artificial intelligence (AI) has seen remarkable advancements, stretching the limits of what is possible and opening up new frontiers. This comparative review investigates the evolving landscape of AI advancements, providing a thorough exploration of innovative techniques that have shaped the field. Beginning with the fundamentals of AI, including traditional machine learning and the transition to data-driven approaches, the narrative progresses through core AI techniques such as reinforcement learning, generative adversarial networks, transfer learning, and neuroevolution. The significance of explainable AI (XAI) is...</p> <p>Show more</p> <p>Authors:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 5px;"> <p>Primary Author Hamed Taherdoost ORCID: https://orcid.org/0000-0002-8503-6739</p> </div> <p>Author Mitra Madanchian</p> <p>Container Title: AI Score: 10.758266 Volume: 5 Issue: 1 Language: en ISBN: N/A Page: 38-54 Entry Type: journal-article</p> <p>Citations:</p> <ul style="list-style-type: none"> • 10.1093/mind/LIX.236.433 • 10.1017/ATSP.2018.6 • 10.1016/j.csn.2017.12.006 • 10.23919/MIPRO.2018.8400040 • 10.5465/ami.2018.0084 • 10.1016/j.infoconol.2019.05.001 • 10.37256/icds.5120233284 • 10.1126/science.217.4566.1237 • 10.3390/computers12040072 • 10.1017/9781009072205 • 10.1007/978-3-030-70388-2 </div>									
Page 1 Next									

Figure 10. Entry Detailed Information Display.

The backend logic within `app.py` supports all these frontend features. The `home()` route correctly processes search, filter (`selected_types`), and pagination (`page`, `per_page`) parameters. The executed SQL query effectively uses CTEs and the `json_agg` function to aggregate author details into the necessary JSON structure passed to the template. Database queries for both the UI and API, executed via `psycopg2` against the `dwh.entry_details_vw`, function as implemented.

4.5 Evaluation Against Research Questions

This section evaluates the implemented system based on the results presented in Sections 4.1-4.4, providing answers to the research questions posed in Section 1.2 through the qualitative assessment methods outlined in Section 3.6.

RQ1 (Schema Design):

The implemented `dwh` schema (Section 4.2, Appendix 2), adhering to 3NF principles, proved effective in representing the diverse metadata elements ingested from Crossref and OpenCitations for the sample data. The normalization resulted in reduced data redundancy, for instance, storing author and organization names uniquely, and established clear, maintainable relationships via foreign keys. The structure, utilizing distinct tables for core entities (`entry`, `author`, etc.) linked via associative tables (`entry_author`, `author_organization`), successfully handled complex, many-to-many relationships like multiple authors per paper or multiple affiliations per author, which are essential for accurate bibliometric analysis. Scalability potential was directly addressed through the successful implementation of physical design features. Notably, RANGE partitioning on the `entry` table by publication date (Section 4.2) allows the query planner to potentially skip irrelevant yearly partitions (partition pruning) during time-bound analyses, significantly improving query efficiency on large datasets. Similarly, HASH partitioning on the large `entry_author` bridge table aids load distribution. Strategic indexing, particularly the implemented GIN trigram indexes on title and author names, provides crucial support for efficient, case-insensitive substring searches (`ILIKE`), a common requirement in bibliographic exploration. While quantitative benchmarking was

outside the scope of this prototype evaluation, these implemented features provide a solid foundation for handling larger data volumes.

RQ2 & RQ4 (DBT Integration & Challenges):

The integration of DBT for managing the ELT pipeline (Section 4.3) demonstrated clear practical advantages over anticipated traditional ETL scripting methods. The modular project structure, breaking down transformations into distinct staging, intermediate, and target models (Figure 4), resulted in highly organized, readable, and consequently more maintainable SQL logic. Debugging specific transformation steps or adapting to source data changes becomes significantly easier due to this modularity. Furthermore, version controlling the entire DBT project enhances collaboration and traceability of changes. DBT's automated dependency management ensured transformations executed in the correct sequence (Figure 7), preventing logical errors. The integrated testing framework proved invaluable for reliability; executing tests (generic schema tests like `not_null`, `unique`, `relationships`, and custom singular tests, (Figure 8, Appendix 3, Appendix 4)) after each `dbt run` provided automated validation of data quality and integrity rules, catching potential issues early and preventing the propagation of erroneous data into the final `dwh` schema.

Key challenges encountered during implementation centered on handling the semi-structured nature of the source data, specifically parsing nested JSON arrays (authors, affiliations) effectively within SQL, which required careful use of PostgreSQL's `jsonb_array_elements` function and robust error handling in intermediate models. Additionally, correctly implementing and debugging the incremental loading strategy (`delete+insert`) necessitated careful management of `updated_at` timestamps and unique keys to ensure idempotency and avoid data duplication or loss, demanding more attention than simple full-refresh approaches. Designing meaningful custom data quality tests beyond basic integrity checks also required careful consideration of bibliographic domain specifics.

RQ3 (Three-Tier Architecture):

The system successfully implemented the planned three-tier architecture (Section 4.1), and the results confirm its benefits in this context. A clear separation of concerns was achieved: PostgreSQL effectively managed data persistence and integrity (Section 4.2);

DBT and Python handled the distinct data transformation and ingestion logic (Section 4.3); Flask, along with HTML/CSS/JavaScript, managed the presentation logic, API provision, and user interface rendering (Section 4.4). This modularity proved beneficial during development, allowing, for example, changes to DBT transformation models without impacting the Flask application code, provided the presentation view interface remained stable. This separation inherently supports independent development, testing, and potential deployment scaling of each tier. The architecture provides effective accessibility through two distinct interfaces tailored to different needs: the interactive web interface (Figure 9) offers a rich, user-friendly experience for direct exploration, including the implemented type filtering, configurable pagination, and detailed data views; the basic JSON API endpoint (`/api/entries`) provides straightforward programmatic access suitable for automated systems or other integrations.

Crucially, the underlying structured `dwh` schema proved suitable for direct connection via standard SQL tools (like DBeaver), enabling advanced ad-hoc querying, and its design allows for integration with external BI platforms (e.g., Power BI, Tableau) for sophisticated visualization, demonstrating analytical potential beyond the implemented application.

The prototype application, including all its interactive frontend features, exhibited good responsiveness during manual testing with the sample data. The architectural design incorporates elements conducive to scaling, such as database partitioning and indexing, and the potentially stateless nature of the Flask application lends itself well to horizontal scaling (running multiple instances). While the `entry_details_vw` simplified application development logic, the complex query utilizing `json_agg` for the main UI, though performant in the prototype, represents a component whose performance characteristics under significantly larger data volumes or high concurrent user load would warrant further analysis and potential optimization (e.g., query tuning, caching strategies) in a production scenario.

5 Conclusion and Future Work

This thesis set out to address the challenges of data heterogeneity, maintainability, and limited analytical capabilities often found in traditional bibliographic database systems. By designing and implementing a modern bibliographic database architecture incorporating contemporary data engineering practices, this work successfully demonstrates a viable and improved approach. The project culminated in a functional prototype featuring a normalized PostgreSQL database optimized with physical design strategies, an ELT pipeline managed effectively by Data Build Tool (DBT), and a modular three-tier application structure providing user access via an interactive web interface and a basic API.

5.1 Summary of Findings and Contributions

The primary contribution of this thesis lies in the practical application and evaluation of modern data engineering tools and architectural patterns specifically within the bibliographic data domain. The key findings validate the effectiveness and benefits of the chosen methodologies.

The development process confirmed that a relational schema, normalized to Third Normal Form (3NF) and augmented with physical optimizations like table partitioning and appropriate indexing (including GIN trigram indexes for text search), can effectively structure diverse bibliographic metadata while providing mechanisms to support scalability and efficient querying.

The integration of DBT proved highly beneficial, successfully orchestrating an ELT pipeline within the PostgreSQL database. The results highlighted DBT's significant advantages in improving the development lifecycle through enhanced modularity (breaking down complex transformations), maintainability (facilitated by version control and clear SQL-based logic), and crucially, reliability (enforced through an integrated, automated testing framework for data quality and integrity). This demonstrated a marked improvement over the anticipated complexities of managing traditional ETL scripts.

The implementation of the three-tier architecture successfully enforced a clear separation of concerns. This modularity between the data tier (PostgreSQL), the logic tier (Python

ingestion script, DBT transformations), and the presentation tier (Flask application serving HTML/CSS/JS) was evident during development and offers inherent advantages for independent maintenance, testing, and potential scaling of each component. The resulting system provides functional and accessible interfaces for both direct user interaction via the web UI and basic programmatic access via the JSON API.

Answers to Research Questions

Based on the evaluation of the results presented in Chapter 4, the research questions posed at the outset can be answered conclusively. The findings demonstrate that (RQ1) a scalable and normalized relational schema can be effectively designed for diverse bibliographic metadata by applying 3NF principles alongside physical design features like partitioning and specialized indexing. Furthermore, (RQ2 & RQ4) the integration of DBT for an ELT workflow provides substantial improvements in maintainability, reliability, and flexibility compared to traditional ETL, although it requires careful handling of semi-structured data parsing and incremental loading logic within SQL, alongside domain-specific testing. Finally, (RQ3) a three-tier architecture was successfully implemented, achieving modularity and providing distinct, accessible interfaces (an interactive web UI and a basic API) by clearly separating the data, logic (Python/DBT), and presentation (Flask/Web Frontend) layers.

Limitations

While the project successfully demonstrated the proposed architecture, several limitations should be acknowledged:

- **Prototype Scale and Evaluation:** The system was developed and tested as a prototype using a limited sample dataset on a local development machine. Consequently, the performance evaluation remained qualitative. Rigorous quantitative benchmarking under realistic, large-scale data volumes and concurrent user loads was not performed, meaning the true scalability limits and potential bottlenecks (e.g., performance of complex queries with `json_agg`) were not empirically determined.
- **Data Source Scope:** Data ingestion was confined to the Crossref and OpenCitations APIs. Integrating a wider array of sources (e.g., PubMed, Scopus, institutional repositories via OAI-PMH) would introduce significant additional

challenges related to schema mapping, varying data quality levels, different API protocols or data formats, and potentially more complex entity resolution requirements.

- **Feature Scope:** The implemented system provides core functionalities for data storage, transformation, and interactive retrieval. Advanced features common in mature bibliometric systems, such as integrated topic modelling capabilities, complex citation network analysis beyond basic link storage, or built-in data visualization tools, were outside the scope of this implementation.
- **JSON API Functionality:** The implemented JSON API endpoint offers only basic search capabilities with fixed pagination and lacks the more advanced filtering options available in the main web UI. It serves primarily as a proof-of-concept for programmatic access rather than a fully developed, feature-rich API suitable for robust external integrations.

5.2 Future Work

The developed system provides a solid foundation that can be extended in several promising directions, incorporating more advanced techniques and expanding its capabilities:

Expand Data Integration and Entity Resolution

Incorporate a broader range of bibliographic data sources (e.g., Scopus API, PubMed E-utilities, arXiv API, OAI-PMH feeds). This would necessitate extending ingestion scripts, developing new DBT staging models, and critically, implementing more robust entity resolution techniques (potentially ML-based) to handle variations and merge records effectively across diverse sources.

Implement Advanced Bibliometric and Network Analysis

Integrate more sophisticated analytical functionalities. This includes enhancing the storage and querying capabilities for citation network analysis (perhaps exploring graph database extensions or libraries like NetworkX) to enable complex graph metrics and community detection. Calculating standard bibliometric indicators (e.g., h-index) could also be added.

Leverage Machine Learning and AI

Explore the application of ML and AI to enhance various aspects of the system:

- *Semantic Search and Recommendation:* Implement vector embeddings (e.g., using models like Sentence-BERT on titles/abstracts) to enable semantic search, finding conceptually related papers beyond keyword matches, and build recommendation engines suggesting relevant articles or potential collaborators.
- *Enhanced Author Disambiguation:* Apply advanced ML classification or clustering models, using features like co-authorship patterns, publication venues, topic distributions (derived from abstracts), and affiliation history, to significantly improve the accuracy of author name disambiguation.
- *Automated Knowledge Extraction:* Utilize NLP techniques, potentially including Large Language Models (LLMs), to automatically extract structured information from abstracts or full texts (if available), such as research methods, datasets used, key findings, or even construct a knowledge graph of scholarly entities and relationships.
- *Topic Modelling and Trend Analysis:* Employ advanced topic modelling techniques (e.g., dynamic topic models, hierarchical models) or transformer-based approaches to automatically identify fine-grained research topics, track their evolution over time, and potentially identify emerging research fronts.
- *Predictive Analytics:* Investigate the use of ML models to predict future research impact (e.g., citation counts) or identify potentially fruitful research collaborations based on network features and historical data.
- *Intelligent Data Quality:* Develop ML models for more sophisticated anomaly detection in metadata, identifying outliers or inconsistencies that might be missed by rule-based DBT tests.

Enhance Frontend and Visualization

Further develop the web user interface by adding features like user accounts, implementing more advanced search syntax (Boolean operators, field-specific queries), refining the UI/UX, and integrating data visualization libraries (e.g., D3.js, Plotly) to

present insights visually, such as publication trends, co-authorship networks derived from the data, or topic distributions.

Cloud Deployment and Performance Optimization

Migrate the system stack to a cloud platform for scalability and manageability. Conduct rigorous performance testing under load, optimizing database configurations, indexing, and computationally intensive queries or DBT models. Implement caching strategies where appropriate.

Refine Data Quality Monitoring

Enhance the data quality framework beyond DBT tests by integrating data profiling tools, establishing monitoring dashboards, and potentially developing semi-automated workflows for resolving identified data issues.

5.3 Concluding Remarks

This thesis successfully addressed the objective of designing and developing a modern architecture for a bibliographic database. By leveraging a normalized relational database, a modular three-tier structure, and the capabilities of DBT for managing an ELT pipeline, the project demonstrated a robust, maintainable, and scalable alternative to traditional approaches. The resulting functional prototype, complete with an interactive web interface, effectively showcases the benefits of integrating contemporary data engineering practices in the domain of scholarly information management. While limitations exist and avenues for future work, particularly those leveraging AI and machine learning, are plentiful, this research provides a valuable practical blueprint and a solid foundation for building a bibliographic information systems capable of handling the growing volume and complexity of scholarly data.

References

- [1] E. Hudomalj and G. Vidmar, 'OLAP and bibliographic databases', Kluwer Academic Publishers, 2003.
- [2] A. Kusserow and S. Groppe, 'Getting Indexed by Bibliographic Databases in the Area of Computer Science I MOTIVATION', 2014. [Online]. Available: <http://creativecommons.org/licenses/by/3.0/>
- [3] A. Ferrara and S. Salini, 'Ten challenges in modeling bibliographic data for bibliometric analysis', *Scientometrics*, vol. 93, no. 3, pp. 765–785, Dec. 2012, doi: 10.1007/s11192-012-0810-x.
- [4] V. Hristidis, Y. Hu, and P. G. Ipeirotis, 'Relevance-based retrieval on hidden-web text databases without ranking support', *IEEE Trans Knowl Data Eng*, vol. 23, no. 10, pp. 1555–1568, Oct. 2011, doi: 10.1109/TKDE.2010.183.
- [5] V. Hristidis, Y. Hu, and P. G. Ipeirotis, 'Ranked Queries over Sources with Boolean Query Interfaces without Ranking Support', in *IEEE 26th International Conference on Data Engineering and Workshops*, IEEE, 2010.
- [6] S. Karimi, J. Zobel, S. Pohl, and F. Scholer, 'The challenge of high recall in biomedical systematic search', in *Proceedings of the third international workshop on Data and text mining in bioinformatics*, ACM Digital Library, 2013.
- [7] S. Khalid and S. Wu, 'Supporting Scholarly Search by Query Expansion and Citation Analysis', in *Engineering, Technology & Applied Science Research*, 2020, pp. 6102–6108.
- [8] S. Khalid, S. Wu, A. Wahid, A. Alam, and I. Ullah, 'An Effective Scholarly Search by Combining Inverted Indices and Structured Search with Citation Networks Analysis', *IEEE Access*, vol. 9, pp. 120210–120226, 2021, doi: 10.1109/ACCESS.2021.3107939.
- [9] C. Xiong, R. Power, and J. Callan, 'Explicit semantic ranking for academic search via knowledge graph embedding', in *26th International World Wide Web Conference, WWW 2017*, International World Wide Web Conferences Steering Committee, 2017, pp. 1271–1279. doi: 10.1145/3038912.3052558.
- [10] H. Kroll, P. Sackhoff, T. Breuer, R. Schenkel, and W.-T. Balke, 'Ranking Narrative Query Graphs for Biomedical Document Retrieval (Technical Report)', Dec. 2024, [Online]. Available: <http://arxiv.org/abs/2412.15232>
- [11] I. A. Ebeid and E. Pierce, 'MedGraph: An experimental semantic information retrieval method using knowledge graph embedding for the biomedical citations indexed in PubMed', 2021. [Online]. Available: <http://www.nlm.nih.gov/pubs/factsheets/medline.html>
- [12] G. Wei *et al.*, 'DocReLM: Mastering Document Retrieval with Language Model', May 2024, [Online]. Available: <http://arxiv.org/abs/2405.11461>
- [13] G. Mitrov, B. Stanoev, S. Gievska, G. Mirceva, and E. Zdravevski, 'Combining Semantic Matching, Word Embeddings, Transformers, and LLMs for Enhanced Document Ranking: Application in Systematic Reviews', *Big Data and Cognitive Computing*, vol. 8, no. 9, p. 110, Sep. 2024, doi: 10.3390/bdcc8090110.
- [14] P. Schneider and F. Matthes, 'Conversational Exploratory Search of Scholarly Publications Using Knowledge Graphs', Oct. 2024, [Online]. Available: <http://arxiv.org/abs/2410.00427>

- [15] G. E. Lee and A. Sun, 'Mirror Matching: Document Matching Approach in Seed-driven Document Ranking for Medical Systematic Reviews', Dec. 2021, <https://arxiv.org/pdf/2112.14318>.
- [16] S. Arachchige Yasith Hasantha Ariyasena, 'Bibliographic data mining based on the Estonian Research Information System', TalTech, Tallinn, 2021.
- [17] J. M. B. Silva and F. Silva, 'Feature extraction for the author name disambiguation problem in a bibliographic database', in *Proceedings of the ACM Symposium on Applied Computing*, Association for Computing Machinery, Apr. 2017, pp. 783–789. doi: 10.1145/3019612.3019663.
- [18] K. Karamcheti, 'Design and implementation of bibliographic database', 2008. doi: 10.25669/p6v2-8sb2.
- [19] R. Lakshmanasamy and G. Ganachari, 'Integration of Dbt With Modern Data Stack Technologies', Oct. 2023. [Online]. Available: www.ijfmr.com
- [20] A. Rahaman, R. Dilip, and T. B. Vidyapeeth, 'A Study of Effective Structures Bibliographic Database: Essential Tool to Find Descriptive Records of Relevant Information Sources', 2023, doi: 10.53555/sfs.v10i1.298.
- [21] I. Zupic and T. Čater, 'Bibliometric Methods in Management and Organization', *Organ Res Methods*, vol. 18, no. 3, pp. 429–472, Jul. 2015, doi: 10.1177/1094428114562629.
- [22] B. H. Butt, M. Rafi, and M. Sabih, 'A systematic metadata harvesting workflow for analysing scientific networks', *PeerJ Comput Sci*, vol. 7, pp. 1–19, Mar. 2021, doi: 10.7717/peerj-cs.421.
- [23] P. L. Mabry, X. Yan, V. Pentchev, R. Van Rennes, S. H. McGavin, and J. V. Wittenberg, 'CADRE: A Collaborative, Cloud-Based Solution for Big Bibliographic Data Research in Academic Libraries', *Front Big Data*, vol. 3, Nov. 2020, doi: 10.3389/fdata.2020.556282.
- [24] B. Trushkowsky, K. Campbell, and J. Forbes, 'An Architecture for a Collaborative Bibliographic database', in *Proceedings of the 2007 conference on Diversity in computing*, 2007.
- [25] T. Do, D. Lam, and T. Huynh, 'A Framework for Integrating Bibliographical Data of Computer Science Publications', in *Computing, Management and Telecommunications (ComManTel), 2014 International Conference on*, 2014.
- [26] T. Georgieva-Trifonova, 'Warehousing and OLAP Analysis of Bibliographic Data', *Intell Inf Manag*, vol. 03, no. 05, pp. 190–197, 2011, doi: 10.4236/iim.2011.35023.
- [27] M. J. Cobo, A. G. López-Herrera, E. Herrera-Viedma, and D. Saucedo Aranda, 'A Relational Database Model for Science Mapping Analysis', 2015.
- [28] N. Mallig, 'A relational database for bibliometric analysis', *J Informetr*, vol. 4, no. 4, pp. 564–580, Oct. 2010, doi: 10.1016/j.joi.2010.06.007.
- [29] P. Manghi, C. Atzori, M. De Bonis, and A. Bardi, 'Entity deduplication in big data graphs for scholarly communication', *Data Technologies and Applications*, vol. 54, no. 4, pp. 409–435, Aug. 2020, doi: 10.1108/DTA-09-2019-0163.
- [30] N. R. Smalheiser, J. Schneider, V. I. Torvik, D. P. Fragnito, and E. E. Tirk, 'The Citation Cloud of a biomedical article: a free, public, web-based tool enabling citation analysis', *Journal of the Medical Library Association*, vol. 110, no. 1, pp. 103–108, Jan. 2022, doi: 10.5195/jmla.2022.1117.
- [31] V. Sadikov and W. Pidkameny, 'Complete Separation of the 3 Tiers - Divide and Conquer', May 2014, Accessed: Apr. 27, 2025. [Online]. Available: <https://arxiv.org/pdf/1405.1618>

- [32] A. Khalilipour, M. Challenger, M. Onat, H. Gezgen, and G. Kardas, 'Refactoring Legacy Software for Layer Separation', *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 2, pp. 217–247, Feb. 2021, doi: 10.1142/S0218194021500066.
- [33] P. L. Mabry, X. Yan, V. Pentchev, R. Van Rennes, S. H. McGavin, and J. V. Wittenberg, 'CADRE: A Collaborative, Cloud-Based Solution for Big Bibliographic Data Research in Academic Libraries', *Front Big Data*, vol. 3, Nov. 2020, doi: 10.3389/fdata.2020.556282.
- [34] M. S. Farhan, A. Youssef, and L. Abdelhamid, 'A Model for Enhancing Unstructured Big Data Warehouse Execution Time', *Big Data and Cognitive Computing*, vol. 8, no. 2, Feb. 2024, doi: 10.3390/bdcc8020017.
- [35] M. Patel and M. Bhise, 'Query Complexity Based Optimal Processing of Raw Data', *IEEE Region 10 Humanitarian Technology Conference, R10-HTC*, vol. 2022-September, pp. 38–43, 2022, doi: 10.1109/R10-HTC54060.2022.9929945.
- [36] A. A. A. Fernandes, M. Koehler, N. Konstantinou, P. Pankin, N. W. Paton, and R. Sakellariou, 'Data Preparation: A Technological Perspective and Review', *SN Comput Sci*, vol. 4, no. 4, Jul. 2023, doi: 10.1007/s42979-023-01828-8.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis¹

I Mani Biglari

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Design and development of a bibliographic database architecture”, supervised by Aleksei Tepljakov.
 - 1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
 - 1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons’ intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

11.05.2025

¹ The non-exclusive licence is not valid during the validity of access restriction indicated in the student’s application for restriction on access to the graduation thesis that has been signed by the school’s dean, except in case of the university’s right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Complete DDL for the dwh Schema

```
-- PostgreSQL DDL for the dwh Schema
-- Includes partitioning, indexes, and constraints.
-- Requires PostgreSQL 13+. Assumes schema "dwh" already exists.
-----
-- 0. Extension & Search Path -----
-----
CREATE EXTENSION IF NOT EXISTS pg_trgm WITH SCHEMA dwh;
SET search_path = dwh, pg_catalog;

-----
-- 1. Dimension Tables -----
-----

DROP TABLE IF EXISTS dwh.author CASCADE;
CREATE TABLE dwh.author (
    author_id    bigint PRIMARY KEY,
    name        text,
    orcid       text,
    organizations text, -- Raw text/JSON of affiliations
    updated_at  timestampz
);
CREATE INDEX IF NOT EXISTS idx_author_name_trgm ON dwh.author USING gin (name
dwh.gin_trgm_ops); -- For ILIKE search

DROP TABLE IF EXISTS dwh.organization CASCADE;
CREATE TABLE dwh.organization (
    id          bigint PRIMARY KEY,
    name       text UNIQUE, -- Canonicalized organization name
    address    text,
    contact_1  text,
    contact_2  text,
    email      text,
    org_type   text,
    updated_at timestampz
);

DROP TABLE IF EXISTS dwh.entry_type CASCADE;
CREATE TABLE dwh.entry_type (
    entry_type_id  bigint PRIMARY KEY,
    entry_type_name text UNIQUE, -- Controlled vocabulary type name
    entry_type_desc text
);

-----
-- 2. Fact & Bridge Tables -----
-----

-- Entry table, range-partitioned by publication year
DROP TABLE IF EXISTS dwh.entry CASCADE;
CREATE TABLE dwh.entry (
    entry_id          bigint          NOT NULL,
    source_id        bigint,
    doi              text,
    title            text,
```

```

publisher          text,
type               text, -- Original type string from source
entry_type_id     bigint, -- FK to dwh.entry_type
publication_date   date      NOT NULL, -- Partition key
language          text,
score             double precision,
container_title   text,
page              text,
isbn              jsonb,
reference_count   integer,
is_referenced_by_count integer,
citations         jsonb,
cited_by         jsonb,
volume           text,
issue            text,
abstract         text,
raw_data         jsonb,
updated_at       timestampz,
fts              tsvector, -- Precomputed full-text search vector
PRIMARY KEY (entry_id, publication_date) -- Includes partition key
) PARTITION BY RANGE (publication_date);

-- Generate yearly partitions for dwh.entry (2000-2030 + legacy)
DO $$
DECLARE
    yr int;
BEGIN
    FOR yr IN 2000..2030 LOOP
        EXECUTE format(
            'CREATE TABLE IF NOT EXISTS dwh.entry_y%s PARTITION OF dwh.entry
             FOR VALUES FROM (''%s-01-01'') TO (''%s-01-01'');',
            yr, yr, yr + 1
        );
    END LOOP;
    EXECUTE 'CREATE TABLE IF NOT EXISTS dwh.entry_legacy PARTITION OF dwh.entry
             FOR VALUES FROM (''1900-01-01'') TO (''2000-01-01'');'; -- Fallback
partition
END$$;

-- Entry-Author bridge table, hash-partitioned by entry_id
DROP TABLE IF EXISTS dwh.entry_author CASCADE;
CREATE TABLE dwh.entry_author (
    entry_author_surrogate_id bigint NOT NULL,
    author_id                bigint NOT NULL, -- FK to dwh.author
    entry_id                 bigint NOT NULL, -- Refers to entry_id in dwh.entry
    is_primary_author        boolean,
    author_sequence          integer, -- Order of author in the publication
    updated_at              timestampz,
    PRIMARY KEY (entry_id, author_id)
) PARTITION BY HASH (entry_id);

-- Generate hash partitions for dwh.entry_author (32 buckets)
DO $$
DECLARE i int;
BEGIN
    FOR i IN 0..31 LOOP
        EXECUTE format(

```

```

        'CREATE TABLE IF NOT EXISTS dwh.entry_author_p%s PARTITION OF
dwh.entry_author
        FOR VALUES WITH (MODULUS 32, REMAINDER %s);',
        i, i
    );
END LOOP;
END$$;

-- Author-Organization bridge table
DROP TABLE IF EXISTS dwh.author_organization CASCADE;
CREATE TABLE dwh.author_organization (
    author_organization_surrogate_id bigint PRIMARY KEY,
    author_id          bigint NOT NULL, -- FK to dwh.author
    organization_id    bigint NOT NULL, -- FK to dwh.organization
    affiliation_sequence integer, -- Order of affiliation for the author
    is_primary_organization boolean,
    updated_at         timestamptz
);

-----
-- 3. Indexes -----
-----
-- GIN trigram index for case-insensitive title search
CREATE INDEX IF NOT EXISTS idx_entry_title_trgm ON dwh.entry USING gin (title
dwh.gin_trgm_ops);

-- Full-text search index using precomputed 'fts' column
CREATE INDEX IF NOT EXISTS idx_entry_fts ON dwh.entry USING gin (fts);

-- Index for sorting entries by recent publication date
CREATE INDEX IF NOT EXISTS idx_entry_recent ON dwh.entry (publication_date
DESC);

-- Covering index for entry-author lookups
CREATE INDEX IF NOT EXISTS idx_entry_author_lookup ON dwh.entry_author
(entry_id, author_id);

-- Indexes for author-organization lookups
CREATE INDEX IF NOT EXISTS idx_author_org_author_lookup ON
dwh.author_organization (author_id);
CREATE INDEX IF NOT EXISTS idx_author_org_org_lookup ON dwh.author_organization
(organization_id);

-----
-- 4. Full-Text Search Trigger -----
-----
-- Trigger function to update the 'fts' tsvector column in dwh.entry
CREATE OR REPLACE FUNCTION dwh.update_fts() RETURNS trigger AS $$
BEGIN
    NEW.fts := to_tsvector('simple', coalesce(NEW.title, '') || ' ' ||
coalesce(NEW.abstract, ''));
    RETURN NEW;
END
$$ LANGUAGE plpgsql;

-- Trigger definition for dwh.entry updates/inserts
DROP TRIGGER IF EXISTS trg_update_fts ON dwh.entry;
CREATE TRIGGER trg_update_fts

```

```

BEFORE INSERT OR UPDATE ON dwh.entry
FOR EACH ROW EXECUTE FUNCTION dwh.update_fts();

-----
-- 5. Foreign Key Constraints -----
-----
-- Constraints added with DEFERRABLE INITIALLY IMMEDIATE as per thesis text

ALTER TABLE dwh.entry
ADD CONSTRAINT fk_entry_entry_type
FOREIGN KEY (entry_type_id) REFERENCES dwh.entry_type(entry_type_id)
DEFERRABLE INITIALLY IMMEDIATE;

ALTER TABLE dwh.entry_author
ADD CONSTRAINT fk_entry_author_author
FOREIGN KEY (author_id) REFERENCES dwh.author(author_id)
DEFERRABLE INITIALLY IMMEDIATE;

ALTER TABLE dwh.author_organization
ADD CONSTRAINT fk_author_organization_author
FOREIGN KEY (author_id) REFERENCES dwh.author(author_id)
DEFERRABLE INITIALLY IMMEDIATE;

ALTER TABLE dwh.author_organization
ADD CONSTRAINT fk_author_organization_organization
FOREIGN KEY (organization_id) REFERENCES dwh.organization(id)
DEFERRABLE INITIALLY IMMEDIATE;

-- Note: A direct FK from dwh.entry_author to the partitioned dwh.entry table
-- is omitted due to the complexity of referencing a composite partitioned key.
-- This relationship integrity is intended to be enforced via DBT tests.

-----
-- End of DDL Script -----
-----

```

Appendix 3 – dwh_schema.yml configuration file

```
version: 2

models:
  - name: entry # Corresponds to entry.sql -> dwh.entry table
    description: "Stores detailed information about each publication entry."
    columns:
      - name: entry_id
        description: "Surrogate primary key for the entry."
        tests:
          - unique # Ensures each entry_id is unique
          - not_null # Ensures entry_id is never NULL
      - name: publication_date
        description: "The date of publication."
        tests:
          - not_null # Publication date is mandatory
      - name: title
        description: "Title of the publication."
        tests:
          - not_null # Title is mandatory
      - name: entry_type_id
        description: "Foreign key referencing the entry_type dimension."
        tests:
          - not_null
          - relationships: # Checks referential integrity
              to: ref('entry_type') # References the dwh.entry_type table
              field: entry_type_id # The primary key column in dwh.entry_type

  - name: author # Corresponds to author.sql -> dwh.author table
    description: "Stores information about unique authors."
    columns:
      - name: author_id
        description: "Surrogate primary key for the author."
        tests:
          - unique
          - not_null
      - name: name
        description: "Author's name."
        tests:
          - not_null # Author name is mandatory

  - name: entry_author # Corresponds to entry_author.sql -> dwh.entry_author
    table
    description: "Junction table linking entries and authors."
    columns:
      # Note: Testing uniqueness on composite keys might require dbt_utils
      package
      # or custom tests. Here we test individual FKs.
      - name: entry_author_surrogate_id # If this is your intended unique key
        for the link
        tests:
          - unique
          - not_null
      - name: entry_id
        description: "Foreign key referencing the entry."
        tests:
```

```

- not_null
- relationships:
  to: ref('entry') # References dwh.entry table
  field: entry_id # Must match entry_id in dwh.entry
- name: author_id
description: "Foreign key referencing the author."
tests:
- not_null
- relationships:
  to: ref('author') # References dwh.author table
  field: author_id # Must match author_id in dwh.author

- name: organization # Corresponds to organization.sql -> dwh.organization
table
description: "Stores information about unique organizations."
columns:
- name: id # This is the organization_id
description: "Surrogate primary key for the organization."
tests:
- unique
- not_null
- name: name
description: "Organization name."
tests:
- not_null

- name: author_organization # Corresponds to author_organization.sql ->
dwh.author_organization
description: "Junction table linking authors and organizations."
columns:
- name: author_organization_surrogate_id
tests:
- unique
- not_null
- name: author_id
tests:
- not_null
- relationships:
  to: ref('author')
  field: author_id
- name: organization_id
tests:
- not_null
- relationships:
  to: ref('organization') # References dwh.organization table
  field: id # The primary key column in dwh.organization

- name: entry_type # Corresponds to entry_type.sql -> dwh.entry_type
description: "Dimension table for publication types."
columns:
- name: entry_type_id
tests:
- unique
- not_null
- name: entry_type_name
tests:
- unique # Type names should also be unique

```

- not_null

Appendix 4 – scripts of the dbt custom singular tests

```
-----  
-- 1. assert_author_orcid_format.sql:  
-- Checks if non-null ORCID values follow either the full URL format  
-- OR the standard numerical format (XXXX-XXXX-XXXX-XXX[X]).  
-- Passes if it returns 0 rows (i.e., all non-null ORCIDs match one of the  
patterns).
```

```
SELECT  
    author_id,  
    orcid  
FROM  
    {{ ref('author') }} -- References dwh.author table  
WHERE  
    orcid IS NOT NULL  
    -- Check if the ORCID does NOT match either the URL pattern OR the  
numerical pattern  
    AND NOT (  
        -- Pattern 1: Full URL (allowing http or https)  
        (orcid ~ '^https?://orcid\.org/\d{4}-\d{4}-\d{4}-\d{3}[\dX]$')  
        OR  
        -- Pattern 2: Just the numerical ID  
        (orcid ~ '^[\d{4}-\d{4}-\d{4}-\d{3}[\dX]$')  
    )
```

```
-----  
-- 2. assert_doi_format_if_present.sql:  
-- This test checks if non-null, non-'unknown' DOIs generally follow  
-- the common '10.' prefix format.  
-- It passes if it returns 0 rows.
```

```
SELECT  
    entry_id,  
    doi  
FROM  
    {{ ref('entry') }} -- References dwh.entry table  
WHERE  
    doi IS NOT NULL  
    AND doi <> 'unknown'  
    AND doi NOT LIKE '10.%' -- Check if it starts with '10.'
```

```
-----  
-- 3. assert_entry_has_primary_author.sql  
-- Finds entries that HAVE authors linked, but none are marked as primary.  
-- Assumes is_primary_author = TRUE indicates the primary author based on  
source order.  
-- Note: This assumes that entries with authors *should* have a primary  
author.  
-- It will NOT return entries that have no authors at all.  
-- Passes if it returns 0 rows.
```



```

SELECT
    e.entry_id
FROM
    {{ ref('entry') }} e
LEFT JOIN
    {{ ref('entry_author') }} ea
    ON e.entry_id = ea.entry_id AND ea.is_primary_author = TRUE
WHERE
    -- Condition 1: Ensure the entry actually HAS authors linked in the
junction table
    e.entry_id IN (SELECT DISTINCT entry_id FROM {{ ref('entry_author') }})
    -- Condition 2: Ensure that among those authors, none met the specific
LEFT JOIN condition (is_primary_author = TRUE)
    AND ea.entry_id IS NULL

-----
-- 4. assert_entry_publication_date_is_reasonable.sql
-- This test checks if any publication dates are suspiciously far in the
future
-- or too far in the past (e.g., before 1900).
-- It passes if it returns 0 rows.

SELECT
    entry_id,
    publication_date
FROM
    {{ ref('entry') }} -- References dwh.entry table
WHERE
    publication_date > (CURRENT_DATE + interval '1 year') -- More than 1 year
in the future?
    OR publication_date < '1900-01-01' -- Before the year 1900?

```

Appendix 5 – scripts of the dbt models and the macro

```
-- =====  
-- STAGING MODELS  
-- =====  
  
-----  
-- 1. stg_crossref.sql:  
  
WITH base AS (  
    SELECT  
        id,  
        doi,  
        title,  
        publisher,  
        type,  
        issued::DATE AS publication_date,  
        language,  
        authors,  
        score,  
        container_title,  
        page,  
        isbn,  
        reference_count,  
        is_referenced_by_count,  
        citations,  
        cited_by,  
        volume,  
        issue,  
        abstract,  
        raw_data  
    FROM {{ source('RAW', 'raw_crossref') }}  
)  
SELECT  
    id,  
    COALESCE(doi, 'unknown') AS doi,  
    TRIM(title) AS title,  
    TRIM(publisher) AS publisher,  
    LOWER(type) AS type,  
    publication_date,  
    LOWER(language) AS language,  
    authors,  
    score,  
    container_title,  
    page,  
    isbn,  
    reference_count,  
    is_referenced_by_count,  
    citations,
```

```

    cited_by,
    volume,
    issue,
    abstract,
    raw_data
FROM base
WHERE title IS NOT NULL

-----
-- staging sources.yml:

version: 2

sources:
  - name: RAW
    schema: raw
    tables:
      - name: raw_crossref
-----

-- =====
-- INTERMEDIATE MODELS
-- =====

-----

-- 2. int_entry.sql:
{{ config(materialized='table') }}

WITH distinct_entries AS (
  SELECT DISTINCT
    id AS source_id,
    doi,
    title,
    publisher,
    type,
    publication_date,
    language,
    authors,
    score,
    container_title,
    page,
    isbn,
    reference_count,
    is_referenced_by_count,
    citations,
    cited_by,
    volume,
    issue,
    REGEXP_REPLACE(CAST(abstract AS VARCHAR), '<[^>]+>', '', 'g') AS
abstract,

```

```

        raw_data,
        CURRENT_TIMESTAMP AS updated_at,
        {{ surrogate_key(["id", "doi", "title", "publisher"]) }} AS entry_id
FROM {{ ref('stg_crossref') }}
WHERE title IS NOT NULL
)
SELECT
    entry_id,
    source_id,
    doi,
    title,
    publisher,
    type,
    publication_date,
    language,
    authors,
    score,
    container_title,
    page,
    isbn,
    reference_count,
    is_referenced_by_count,
    citations,
    cited_by,
    volume,
    issue,
    abstract,
    raw_data,
    updated_at
FROM distinct_entries

```

-- 3. int_entry_type.sql:

```

WITH distinct_types AS (
    SELECT DISTINCT
        LOWER(type) AS type
    FROM {{ ref('stg_crossref') }}
),
new_types AS (
    SELECT
        type
    FROM distinct_types
    LEFT JOIN {{ source('DWH', 'entry_type') }} et
    ON distinct_types.type = et.entry_type_name
    WHERE et.entry_type_id IS NULL
),
max_id AS (
    SELECT COALESCE(MAX(entry_type_id), 0) AS max_id
    FROM {{ source('DWH', 'entry_type') }}
)

```

```

SELECT
    ROW_NUMBER() OVER (ORDER BY type) + (SELECT max_id FROM max_id) AS
entry_type_id,
    type AS entry_type_name,
    NULL AS entry_type_desc
FROM new_types

-----
-- 4. int_author.sql:

{{ config(materialized='table') }}

WITH distinct_authors AS (
    SELECT DISTINCT
        author->>'name' AS name,
        author->>'orcid' AS orcid,
        author->>'affiliations' AS organizations,
        CURRENT_TIMESTAMP AS updated_at
    FROM {{ ref('stg_crossref') }},
        jsonb_array_elements(authors) AS author
    WHERE author->>'name' IS NOT NULL
)
SELECT
    {{ surrogate_key(["name", "orcid", "organizations"]) }} AS author_id,
    name,
    orcid,
    organizations,
    updated_at
FROM distinct_authors
WHERE name IS NOT NULL

-----
-- 5. int_entry_author.sql:

{{ config(materialized='table') }}

WITH base_entries AS (
    SELECT DISTINCT
        id AS source_id,
        doi,
        title,
        publisher,
        authors,
        CURRENT_TIMESTAMP AS updated_at,
        {{ surrogate_key(["id", "doi", "title", "publisher"]) }} AS entry_id
    FROM {{ ref('stg_crossref') }}
    WHERE title IS NOT NULL
),
expanded_authors AS (
    SELECT

```

```

        be.entry_id,
        a.value,
        a.ordinality,
        be.updated_at
    FROM base_entries be
    CROSS JOIN LATERAL jsonb_array_elements(be.authors) WITH ORDINALITY AS
a(value, ordinality)
),
final AS (
    SELECT
        entry_id,
        REGEXP_REPLACE(CAST(value->>'name' AS VARCHAR), '<[^>]+>', '', 'g') AS
author_name,
        ordinality,
        updated_at
    FROM expanded_authors
    WHERE value->>'name' IS NOT NULL
)
SELECT
    entry_id,
    author_name,
    updated_at,
    ordinality,
    {{ surrogate_key(["entry_id", "author_name"]) }} AS
entry_author_surrogate_id
FROM final

```

--6. int_organization.sql:

```

{{ config(materialized='table') }}

WITH extracted_organizations AS (
    SELECT DISTINCT
        CASE
            WHEN jsonb_typeof(org) = 'object' THEN TRIM(org->>'name')
            ELSE TRIM(BOTH '' FROM org::text)
        END AS organization_name,
        CURRENT_TIMESTAMP AS updated_at
    FROM {{ ref('stg_crossref') }},
        jsonb_array_elements(authors) AS author,
        LATERAL jsonb_array_elements(
            CASE
                WHEN jsonb_typeof(author->'affiliations') = 'array' THEN
author->'affiliations'
                ELSE jsonb_build_array(author->'affiliations')
            END
        ) AS org
    WHERE author->>'name' IS NOT NULL
        AND author->'affiliations' IS NOT NULL
)

```

```

SELECT
    {{ surrogate_key(["organization_name"]) }} AS organization_id,
    organization_name,
    updated_at
FROM extracted_organizations
WHERE organization_name IS NOT NULL
    AND organization_name <> 'null'

-----
-- 7. int_author_organization.sql:

{{ config(materialized='table') }}

WITH base_authors AS (
    SELECT DISTINCT
        a.author_id,
        a.name AS author_name,
        a.organizations::jsonb AS organizations, -- Cast organizations to
jsonb
        CURRENT_TIMESTAMP AS updated_at
    FROM {{ ref('int_author') }} a
    WHERE organizations IS NOT NULL
),
expanded_organizations AS (
    SELECT
        ba.author_id,
        org.value AS organization,
        org.ordinality,
        ba.updated_at
    FROM base_authors ba
    CROSS JOIN LATERAL jsonb_array_elements(
        CASE
            WHEN jsonb_typeof(ba.organizations) = 'array' THEN
ba.organizations
            WHEN jsonb_typeof(ba.organizations) = 'string' THEN
jsonb_build_array(ba.organizations)
            ELSE '[]'::jsonb
        END
    ) WITH ORDINALITY AS org(value, ordinality)
),
final AS (
    SELECT
        author_id,
        REGEXP_REPLACE(CAST(organization AS VARCHAR), '<[^>]+>', '', 'g') AS
organization_name,
        ordinality,
        updated_at
    FROM expanded_organizations
    WHERE organization IS NOT NULL
)
SELECT

```

```

    author_id,
    REGEXP_REPLACE(CAST(organization_name AS VARCHAR), '['"', '', 'g') AS
organization_name, -- Remove double quotes
    updated_at,
    ordinality,
    {{ surrogate_key(["author_id", "organization_name"]) }} AS
author_organization_surrogate_id
FROM final

```

```

-----
-- intermediate sources.yml:

```

```

version: 2

```

```

sources:

```

```

- name: DWH
  schema: dwh
  tables:

```

```

    - name: entry_type
#    - name: organization
    - name: author

```

```

-----
-- =====
-- TARGET MODELS
-- =====

```

```

-----
-- 8. author_organization.sql:

```

```

{{
  config(
    materialized = 'incremental',
    incremental_strategy = 'delete+insert',
    unique_key = 'author_organization_surrogate_id'
  )
}}

```

```

WITH author_organizations AS (
  SELECT
    iao.author_id,
    iao.organization_name AS name,
    iao.ordinality,
    iao.updated_at,
    iao.author_organization_surrogate_id
  FROM {{ ref('int_author_organization') }} iao
)
SELECT

```



```

    ao.author_id,
    o.id AS organization_id,
    ao.author_organization_surrogate_id,
    CASE WHEN ao.ordinality = 1 THEN TRUE ELSE FALSE END AS
is_primary_organization,
    GREATEST(ao.updated_at, o.updated_at) AS updated_at
FROM author_organizations ao
JOIN {{ ref('organization') }} o
    ON LOWER(TRIM(ao.name)) = LOWER(TRIM(o.name))
{% if is_incremental() %}
WHERE GREATEST(ao.updated_at, o.updated_at) > (
    SELECT COALESCE(MAX(updated_at), '1900-01-01')
    FROM {{ this }}
)
{% endif %}
ORDER BY ao.author_id, ao.ordinality

```

```

-----
-- 9. author.sql:

```

```

{{
    config(
        materialized = 'incremental',
        incremental_strategy = 'delete+insert',
        unique_key = 'author_id'
    )
}}

SELECT
    author_id,
    name,
    orcid,
    organizations,
    updated_at
FROM {{ ref('int_author') }}
{% if is_incremental() %}
WHERE updated_at > (SELECT COALESCE(MAX(updated_at), '1900-01-01') FROM {{
this }})
{% endif %}

```

```

-----
--10. entry_author.sql:

```

```

{{
    config(
        materialized = 'incremental',
        incremental_strategy = 'delete+insert',
        unique_key = 'entry_author_surrogate_id'
    )
}}

```

```

WITH entry_authors AS (
    SELECT
        iea.entry_id,
        iea.author_name as name,
        iea.ordinality,
        iea.updated_at,
        iea.entry_author_surrogate_id
    FROM {{ ref('int_entry_author') }} iea
)
SELECT
    ea.entry_id,
    a.author_id,
    ea.entry_author_surrogate_id,
    CASE WHEN ea.ordinality = 1 THEN TRUE ELSE FALSE END AS is_primary_author,
    GREATEST(ea.updated_at, a.updated_at) AS updated_at
FROM entry_authors ea
JOIN {{ ref('author') }} a
    ON LOWER(TRIM(ea.name)) = LOWER(TRIM(a.name))
{% if is_incremental() %}
WHERE GREATEST(ea.updated_at, a.updated_at) > (
    SELECT COALESCE(MAX(updated_at), '1900-01-01')
    FROM {{ this }}
)
{% endif %}
ORDER BY ea.entry_id, ea.ordinality

```

-- 11. entry_type.sql:

```

{{ config(materialized='incremental')}}

SELECT
    entry_type_id,
    entry_type_name,
    entry_type_desc
FROM {{ ref('int_entry_type') }}
{% if is_incremental() %}
WHERE entry_type_id > (SELECT COALESCE(MAX(entry_type_id),0) FROM {{ this }})
{% endif %}

```

-- 12. entry.sql:

```

{{
    config(
        materialized = 'incremental',
        incremental_strategy = 'delete+insert',
        unique_key = 'entry_id'
    )
}}

```

```

SELECT
    ie.entry_id,
    ie.source_id,
    ie.doi,
    ie.title,
    ie.publisher,
    ie.type,
    et.entry_type_id,
    ie.publication_date,
    ie.language,
    ie.score,
    ie.container_title,
    ie.page,
    ie.isbn,
    ie.reference_count,
    ie.is_referenced_by_count,
    ie.citations,
    ie.cited_by,
    ie.volume,
    ie.issue,
    ie.abstract,
    ie.raw_data,
    ie.updated_at
FROM {{ ref('int_entry') }} ie
JOIN {{ ref('entry_type') }} et
    ON ie.type = et.entry_type_name
{% if is_incremental() %}
WHERE ie.updated_at > (
    SELECT COALESCE(MAX(x.updated_at), '1900-01-01')
    FROM (SELECT updated_at FROM {{ this }}) AS x
)
{% endif %}
ORDER BY ie.entry_id

```

```

-----
-- 13. organization.sql:

```

```

{{
    config(
        materialized = 'incremental',
        incremental_strategy = 'delete+insert',
        unique_key = 'id'
    )
}}

```

```

SELECT
    organization_id AS id,
    organization_name AS name,
    NULL AS address,
    NULL AS contact_1,
    NULL AS contact_2,

```

```

        NULL AS email,
        NULL AS org_type,
        updated_at
FROM {{ ref('int_organization') }}
{% if is_incremental() %}
WHERE updated_at > (SELECT COALESCE(MAX(updated_at), '1900-01-01') FROM {{
this }})
{% endif %}

```

-- 14. entry_details_vw.sql:

```

{{ config(materialized='view') }}

```

```

WITH entry_data AS (
    SELECT
        e.entry_id,
        e.source_id,
        e.doi,
        e.title,
        e.publisher,
        e.publication_date,
        e.language,
        e.score,
        e.container_title,
        e.page,
        e.isbn,
        e.reference_count,
        e.is_referenced_by_count,
        e.citations,
        e.cited_by,
        e.volume,
        e.issue,
        e.abstract,
        e.updated_at,
        et.entry_type_id,
        et.entry_type_name
    FROM {{ ref('entry') }} e
    JOIN {{ ref('entry_type') }} et
        ON e.entry_type_id = et.entry_type_id
),
author_data AS (
    SELECT
        ea.entry_id,
        a.author_id,
        a.name AS author_name,
        a.orcid,
        ea.is_primary_author,
        ea.updated_at AS entry_author_updated_at
    FROM {{ ref('entry_author') }} ea
    JOIN {{ ref('author') }} a

```

```

        ON ea.author_id = a.author_id
    ),
organization_data AS (
    SELECT
        ao.author_id,
        o.id,
        o.name AS organization_name,
        ao.is_primary_organization,
        ao.updated_at AS author_organization_updated_at
    FROM {{ ref('author_organization') }} ao
    JOIN {{ ref('organization') }} o
        ON ao.organization_id = o.id
)
SELECT
    ed.entry_id,
    ed.source_id,
    ed.doi,
    ed.title,
    ed.publisher,
    ed.publication_date,
    ed.language,
    ed.score,
    ed.container_title,
    ed.page,
    ed.isbn,
    ed.reference_count,
    ed.is_referenced_by_count,
    ed.citations,
    ed.cited_by,
    ed.volume,
    ed.issue,
    ed.abstract,
    ed.updated_at AS entry_updated_at,
    ed.entry_type_id,
    ed.entry_type_name,
    ad.author_id,
    ad.author_name,
    ad.orcid,
    ad.is_primary_author,
    ad.entry_author_updated_at,
    od.id as organization_id,
    od.organization_name,
    od.is_primary_organization,
    od.author_organization_updated_at
FROM entry_data ed
LEFT JOIN author_data ad
    ON ed.entry_id = ad.entry_id
LEFT JOIN organization_data od
    ON ad.author_id = od.author_id
where od.id is not null

```

```
-----  
-- =====  
-- macro  
-- =====  
  
-- surrogate_key.sql macro:  
  
{% macro surrogate_key(columns) %}  
  abs(  
    hashtext(  
      concat(  
        {%- for column in columns -%}  
          coalesce(cast({{ column }} as text), '')  
          {%- if not loop.last -%}, {% endif -%}  
        {%- endfor -%}  
      )  
    )  
  )  
{% endmacro %}
```

Appendix 6 – app.py python script

```
from flask import Flask, render_template, request, jsonify,
send_from_directory
import os
import psycopg2
from config import DB_CONFIG
import json

app = Flask(__name__, static_folder='static')

# Serve static files (fix missing CSS and images)
@app.route('/static/<path:filename>')
def static_files(filename):
    return send_from_directory(app.static_folder, filename)

# Database Connection
def get_db_connection():
    try:
        return psycopg2.connect(**DB_CONFIG)
    except psycopg2.Error as e:
        print(f"Database connection error: {e}")
        return None

# Add JSON filter
@app.template_filter('fromjson')
def fromjson(value):
    """
    Safely convert a JSON-encoded string coming from the database into a
    Python
    object for Jinja templates. If the value is already a Python list or dict,
    just return it unchanged.
    """
    if isinstance(value, (list, dict)):
        return value
    try:
        return json.loads(value)
    except (TypeError, json.JSONDecodeError):
        return []

# Define the selected columns
SELECTED_COLUMNS = """
    doi, title, author_name, is_primary_author, orcid, publisher,
    publication_date,
    language, score, container_title, page, isbn, reference_count,
    is_referenced_by_count,
    citations, cited_by, volume, issue, organization_name, entry_type_name,
    abstract
    """
```

```

@app.route("/", methods=["GET"])
def home():
    """Render the search page with optional query results."""
    title_query = request.args.get("title", "").strip()
    author_query = request.args.get("author", "").strip()
    selected_types = request.args.getlist("types")
    page = int(request.args.get("page", 0))
    per_page = int(request.args.get("per_page", 10))
    offset = page * per_page

    conn = get_db_connection()
    if not conn:
        return "Database connection error", 500

    try:
        cur = conn.cursor()

        # Get total counts first
        cur.execute("""
            SELECT entry_type_name, COUNT(DISTINCT title) as count
            FROM dwh.entry_details_vw
            GROUP BY entry_type_name
            ORDER BY entry_type_name
        """)
        total_type_counts = dict(cur.fetchall())
        entry_types = list(total_type_counts.keys())

        # Build search conditions
        search_conditions = []
        params = []

        if title_query:
            search_conditions.append("LOWER(title) LIKE LOWER(%s)")
            params.append(f"%{title_query}%")

        if author_query:
            search_conditions.append("LOWER(author_name) LIKE LOWER(%s)")
            params.append(f"%{author_query}%")

        # Build type filter condition
        if selected_types and "all" not in selected_types:
            search_conditions.append("entry_type_name = ANY(%s)")
            params.append(selected_types)

        # Main query with author grouping
        where_clause = f"WHERE {' AND '.join(search_conditions)}" if
search_conditions else ""
        sql_query = f"""
            WITH GroupedEntries AS (
                SELECT DISTINCT ON (title)

```



```

        title, entry_type_name, doi, publication_date, publisher,
        reference_count, is_referenced_by_count,
organization_name,
        container_title, score, volume, issue, language, isbn,
        page, abstract, citations, cited_by,
        author_name
    FROM dwh.entry_details_vw
    WHERE is_primary_author = true
        {' AND ' + ' AND '.join(search_conditions) if
search_conditions else ''}
    ),
    AuthorDetails AS (
        SELECT
            g.*,
            json_agg(
                DISTINCT jsonb_build_object(
                    'name', a.author_name,
                    'orcid', a.orcid,
                    'is_primary', a.is_primary_author
                )
            )::json as author_details
        FROM GroupedEntries g
        JOIN dwh.entry_details_vw a ON g.title = a.title
        GROUP BY g.title, g.entry_type_name, g.doi,
g.publication_date, g.publisher,
            g.reference_count, g.is_referenced_by_count,
g.organization_name,
            g.container_title, g.score, g.volume, g.issue,
g.language, g.isbn,
            g.page, g.abstract, g.citations, g.cited_by,
g.author_name
    )
    SELECT * FROM AuthorDetails
    ORDER BY title
    LIMIT %s OFFSET %s
"""
params.extend([per_page, offset])

# Build a second WHERE clause that matches the main query's logic,
# i.e. it also limits the rows to the primary author of each title.
filtered_where_clause = (
    "WHERE is_primary_author = true"
    + (f" AND {' AND '.join(search_conditions)}" if search_conditions
else "")
)

filtered_count_sql = f"""
    SELECT entry_type_name, COUNT(DISTINCT title) AS count
    FROM dwh.entry_details_vw
    {filtered_where_clause}
    GROUP BY entry_type_name

```

```

    """
    cur.execute(filtered_count_sql, tuple(params[:-2]) if params else ())
    filtered_type_counts = dict(cur.fetchall())

    # Execute main query
    cur.execute(sql_query, tuple(params))
    rows = cur.fetchall()
    columns = [desc[0] for desc in cur.description]
    entries = [dict(zip(columns, row)) for row in rows] if rows else []

except psycopg2.Error as e:
    print(f"Database query error: {e}")
    entries = []
    entry_types = []
finally:
    cur.close()
    conn.close()

return render_template(
    "index.html",
    entries=entries,
    entry_types=entry_types,
    type_counts=total_type_counts,
    filtered_type_counts=filtered_type_counts,
    selected_types=selected_types,
    title_query=title_query,
    author_query=author_query,
    page=page,
    per_page=per_page,
    entries_count=len(entries)
)

# API Endpoints
@app.route("/api/entries", methods=["GET"])
def api_get_entries():
    """API to fetch bibliographic records with pagination."""
    general_query = request.args.get("q", "").strip()
    title_query = request.args.get("title", "").strip()
    author_query = request.args.get("author", "").strip()
    page = int(request.args.get("page", 0)) # Get the page number, default to
0
    limit = 10 # Changed from 200 to 10 records per page
    offset = page * limit # Calculate offset for pagination

    conn = get_db_connection()
    if not conn:
        return jsonify({"error": "Database connection error"}), 500

    try:
        cur = conn.cursor()

```

```

sql_query = f"""
    SELECT DISTINCT ON (title, author_name) {SELECTED_COLUMNS}
    FROM dwh.entry_details_vw
    WHERE (%s = '' OR title ILIKE %s OR author_name ILIKE %s OR
publisher ILIKE %s)
    AND (%s = '' OR title ILIKE %s)
    AND (%s = '' OR author_name ILIKE %s)
    ORDER BY title, author_name, organization_name
    LIMIT {limit} OFFSET {offset}
    """

cur.execute(sql_query, (
    general_query, f"%{general_query}%", f"%{general_query}%",
f"%{general_query}%",
    title_query, f"%{title_query}%",
    author_query, f"%{author_query}%"
))

rows = cur.fetchall()
columns = [desc[0] for desc in cur.description]
results = [dict(zip(columns, row)) for row in rows] if rows else []

except psycopg2.Error as e:
    print(f"Database query error: {e}")
    results = []
finally:
    cur.close()
    conn.close()

return jsonify(results)

if __name__ == "__main__":
    app.run(debug=True)

```

Appendix 7 – Source Code Repository

The full source code for the project described in this thesis, ‘Design and development of a bibliographic database architecture,’ is publicly available on GitHub at the following URL:

[https://github.com/ManiBiglari/Bibliographic-Database-Project"](https://github.com/ManiBiglari/Bibliographic-Database-Project)