An Approach to Data Integration and Explorative Query Processing in Scientific Data Management Platforms

Dissertation

vom Fachbereich Informatik der RPTU in Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation

von

Gajendra Doniparthi

Datum der wissenschaftlichen Aussprache	02.10.2025
Dekan	Prof. Dr. Christoph Garth
Berichterstatter	Prof. Dr.-Ing. Stefan Deßloch
	Prof. Dr.-Ing. habil. Bernhard Mitschang

DE-386

Abstract

Technological advancements in bioscience research are directly influencing the generation of vast amounts of complex and heterogeneous datasets from individual studies. Efficient research data management (RDM) solutions based on FAIR principles can help research groups standardize and package their study-specific results into uniquely identifiable digital objects that are easily traceable. To explore the inter-dependencies among datasets originating from different research disciplines, it is essential to deploy a generic, data-centric RDM solution that addresses inherent challenges and effectively manages complex datasets.

This thesis introduces *PLANTdataHUB*, an end-to-end scientific Research Data Management (RDM) ecosystem for plant science data that generates FAIR digital objects, known as Annotated Research Contexts (ARCs). We present an incremental approach to developing a set of search and exploration applications for the plant science community, utilizing the PLANTdataHUB solution. The goal is to facilitate interdisciplinary data analysis among participating research groups, enabling knowledge discovery, collaboration, and innovation.

Our research focuses on developing a framework for exploring large-scale, multi-model plant science datasets. A key contribution of our work is the introduction of a novel key-value index store within the polystore architecture. We propose a fast, scalable, space-efficient, and flexible indexing scheme that leverages purpose-built bitmaps for exploratory data analysis, supporting containment, point, and range query types. This index store complements the query processing mechanism, enabling the execution of cross-model queries across multiple data sources. Additionally, we extend the index management and conceptualize it as a two-player game to address the challenges of attribute selection and cost-based refinement, adapting to the query workloads.

Furthermore, we expand our research by implementing search applications that facilitate integrated metadata exploration through the *ARC Metadata Registry* and enable in-situ, on-demand querying of ARC datasets with *ARCXplore*.

I am eternally grateful to my mother, Hymavathi, and my elder sisters, Uma and Gowri, for all the sacrifices they have made in their lives to help me succeed.

Contents

Chapter 1

Introduction

The rapid advancements in high-throughput analysis of *-omics*[1] studies and the application of such technologies for monitoring vital players in the cell (DNA, RNA, proteins, metabolites, etc.) produce more data today than ever at lower cost, resulting in extended and complex data sets [153]. The integrated analysis of these data sets provides opportunities to get a deeper understanding of molecular phenomena [1,131,138]. Integrating multi-omics data allows us to gain insights at the molecular level by combining information from various biological layers, such as the genome, transcriptome, proteome, and metabolome, which helps in gaining a thorough understanding of complex biological systems [163]. The inherent complexity of multi-omics data brings several challenges related to data management, integration, and analysis.

It is widely accepted in the bio-science research community that an effective Research Data Management (RDM) platform based on FAIR principles is essential for packaging study-specific research results into uniquely identifiable and accessible FAIR-compliant digital objects. Both Research Data Management (RDM) and FAIRness (i.e., data that is findable, accessible, interoperable, and reusable) become the two key focus areas for individual bio-science communities from diverse fields of studies to improve the reproducibility, visibility, and accessibility of their heterogeneous research data [2, 6, 144, 145]. Integrating research results from various research groups provides better knowledge discovery, collaboration, and innovation opportunities. Understandably, it is common for larger research groups to deploy their proprietary RDM solutions to mitigate inherent challenges, while smaller groups leverage the existing open solutions.

A wide range of study-specific RDM solutions exist, such as public repositories, web-based data management platforms, stand-alone information systems, etc., that are apt for managing data from a single field of study. For example, public repositories such as MetaboLights [25, 26] and PRIDE [27] are popular open-source end-points for metabolomics and proteomics communities, respectively, to share study-specific data and curate knowledge. Users can do a direct free-text search or refine the search using search facets, which allows for filtering the search results by specific studies or organisms. Web-based platforms such as FLOW integrates bio-informatic analysis and database solutions, offering a user-friendly interface and web API, accommodating various genomics methods [143]. Similarly, stand-alone information systems such as openBIS [30] (Open Biological Information System) offer mechanisms for storing and managing research data. However, these data management solutions are not conducive to identifying the inter-dependencies across data from different study disciplines. To explore the inter-dependencies, it is prudent to have a generic data-centric RDM solution to manage omics datasets in local/remote hubs or repositories, however, with added integrated search functionalities [154]. Thus, published

[1] omics - an informal reference to a field of studies ending in -omics, such as Genomics, Proteomics, or Metabolomics.

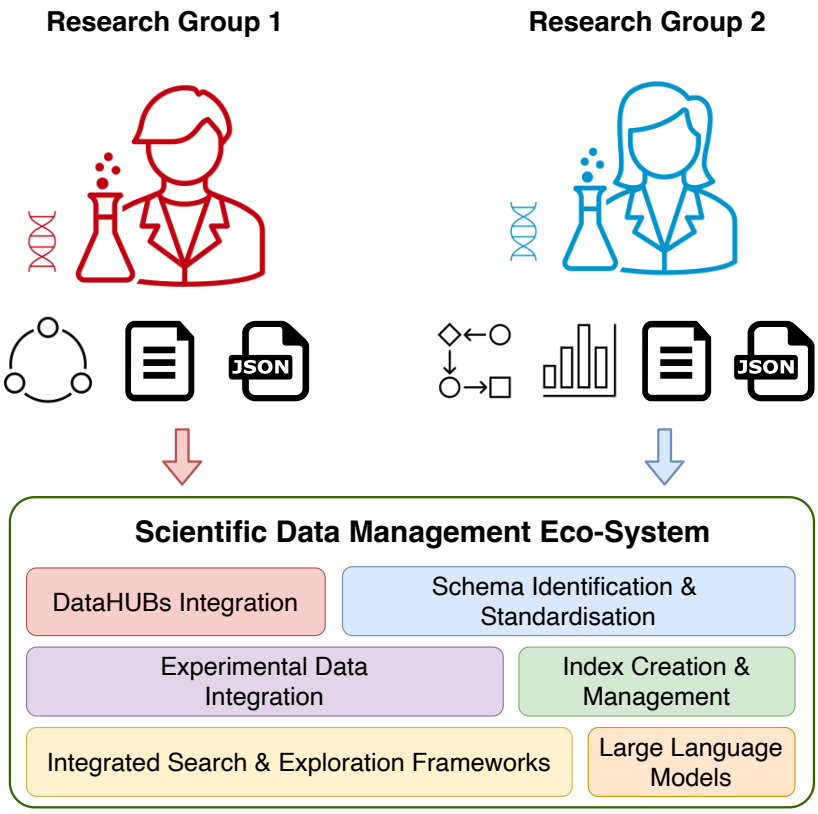

Figure 1.1: The various integration and management stages in a scientific data management platform involve handling heterogeneous datasets from individual research groups, which often include multi-model and unstructured data.

bio-science research results can be visible, accessible, and reproducible for subsequent reuse by the community [152].

The search and exploration applications play a pivotal role in fully integrating the standardized research data and facilitating the interdisciplinary analysis within and across various RDM eco-systems. They are expected to provide a near real-time consolidated view of the research data while hiding the heterogeneity of the data model and data exchange methodologies among the participating groups. Explorative query processing techniques enable scientists to interactively explore the datasets, identify patterns, and generate data-driven insights.

1.1 Scientific Data Management

Integrating bioscience data aims to combine information from various sources to derive meaningful biological conclusions while addressing challenges of high dimensionality and data complexity [162]. Figure 1.1 illustrates key stages in developing a data-centric ecosystem for scientific data management that facili-

tates the integrated search and exploration of datasets. The individual research groups produce various heterogeneous datasets that must be standardized and managed in the respective local or remote data hubs. It starts with creating guidelines and data exchange formats while adopting standard methods, frameworks, and security policies for accessing data from the public repositories (or data hubs). The process becomes more straightforward when participating research groups implement a generic Research Data Management (RDM) solution.

The next step is to develop additional tools and applications to standardize datasets as part of the RDM solution. Numerous metadata standards exist within individual research domains, making interoperability among standards a crucial aspect of research data management. Since sharing experimental metadata is essential for reproducibility, the RDM solution should enable directly importing or exporting metadata in multiple standardized formats. Software tools, such as annotation tools for metadata curation and conversion tools for generating standardized export formats, should be user-friendly and efficient.

The experimental data integration step can be broadly classified into metadata and measured data integration, as illustrated in Figure 1.2. Metadata integration involves creating search engines, registries, and discovery indexes that make standardized metadata searchable. This enhances the findability, visibility, and usability of datasets. Additionally, end-point repositories offer essential search functions to browse study-specific datasets from the relevant repositories. The main objective of metadata integration is to facilitate multi-omics search capabilities within experimental metadata and related text descriptions, allowing researchers to identify correlations among datasets and improve the accessibility of research results across various domains. For instance, popular open-source end-point public repositories like MetaboLights [25] and PRIDE [27] offer search functionality based on study-specific metadata. Moreover, lightweight metadata discovery index platforms such as OmicsDI [33] allow users to access, discover, and disseminate omics datasets from multiple study-specific repositories simultaneously.

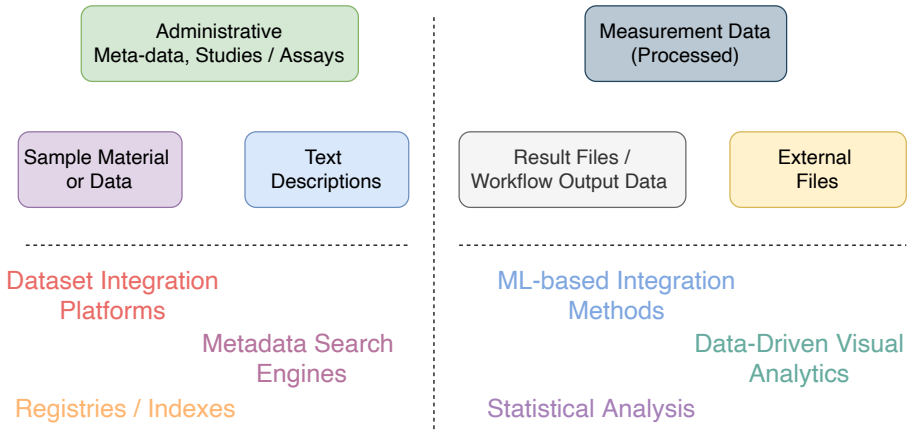

Figure 1.2: Data Integration Classification. Meta-data integration (Left) vs. Measurement Data Integration (Right)

The other aspect of bio-science data integration focuses on combining measurement and output datasets generated from computational workflows across various study-specific datasets. This integration helps to understand interactions among biological layers and derive valuable biological inferences. These methods employ powerful statistical and machine-learning techniques on subsets of the data of interest, which are analyzed simultaneously across multiple omics layers [162,163]. Additionally, data-driven visualization tools play a crucial role in the multi-dimensional analysis of different datasets [164]. Integrated multi-omics analysis methods require specific portions of relevant datasets extracted from one or more end-point repositories.

Indexing heterogeneous data from bioscience datasets is essential for faster query responses, particularly in bioscience data analysis. Flexible data models, combined with an efficient and rapid indexing scheme, can be particularly beneficial when handling diverse query workloads. In dynamic environments that require rapid access to data, adaptive indexing schemes are essential for significantly reducing query execution times.

The final step involves designing and developing search and exploration applications that meet general scientific data search requirements. These applications should provide a real-time view of the integrated data from multiple sources through user interfaces and APIs. The challenge lies in creating a generic platform that enables users to search for particular datasets using metadata and extract subsets of related measurement data for integrated analysis. It must transform standardized metadata from various repositories into a unified format while preserving the original data files in native formats to allow users to explore and query the files seamlessly, including experimental metadata, associated measurement data, and dataset structure as a cohesive unit.

As large language models (LLMs) gain popularity in bioscience research, they represent a promising alternative to traditional text search engines for exploring standardized research data. LLMs can potentially transform how we analyze integrated experimental metadata, providing a more powerful and insightful tool than conventional text search engines. Modern language models comprehend search keywords and the semantics behind user queries, allowing for more accurate results.

1.2 DataPLANT

DataPLANT[2], an NFDI-funded consortium of plant researchers, was formed to drive the democratization and digital transformation of research data in basic plant research. The group's mission is to research and develop an RDM eco-system that meets the community's requirements and enables the contextualization of research datasets in line with the FAIR principles. Within the DataPLANT project, we focus on large-scale plant science data modeling, novel data integration, and indexing methods that form the foundations for an integrated search and exploration framework. This framework enables interdisciplinary data analysis across participating plant science research groups. We primarily address the research gaps in modeling heterogeneous plant science datasets

[2]https://nfdi4plants.org/

(experimental metadata, contextual data, measurement data, etc.), seamlessly integrating data from multiple plant research groups, and developing novel data indexing mechanisms for multi-omics data analysis query workloads in scientific data management platforms.

The essential research outcome of the DataPLANT is *PLANTdataHUB* [8], an end-to-end scientific Research Data Management (RDM) ecosystem for plant science data that generates FAIR digital objects called Annotated Research Contexts (ARCs). These ARCs are maintained in centralized data repositories called dataHUBs. As part of PLANTdataHUB, we developed various supporting tools to facilitate easy curation, collaboration, and maintenance of ARCs on-premise and remote DataHUBs. ARCs and DataHUBs define the scope of this research work in integrating plant science data for multi-omics analysis.

This thesis presents an incremental approach to developing a set of search and exploration applications for plant science. These applications integrate information from various research groups utilizing PLANTdataHUB solutions. They offer researchers a consolidated, real-time view of integrated experimental metadata, measurement data, and contextual information from different omics projects, thereby facilitating multi-omics data analysis. The application's key components include innovative methods for integrating experimental metadata and measurement data, modeling and managing heterogeneous measurement data, and adaptive index management to efficiently handle cross-model query workloads. Additionally, we utilize plant science literature to pre-train large language models and then fine-tune them with the omics metadata corpus. This process enhances the models' ability to understand user queries in natural language, facilitating exploration of multi-omics data.

1.3 Research Tasks

We have organized our research and development into four distinct parts. Firstly, we introduce the *ARC Metadata Registry*, a cloud-native application that utilizes the ARC specification to integrate experimental metadata and measurement data from multiple DataHUBs [6]. The application is designed to connect with multiple DataHUBs simultaneously, providing a consolidated, real-time view of metadata from the ARCs across the DataHUBs. The underlying data model for experimental metadata features a novel approach that merges the schema of heterogeneous measurement data with the experimental metadata. The query execution is also adapted to accommodate the integrated data and schema. This data model complements our ARC specification within the PLANTdataHUB solution.

Secondly, we explore the appropriate system architecture for heterogeneous measurement data, focusing on the data model that supports efficient index management and effective data structures for cross-model query workloads. We start with a basic RDBMS-based framework to examine combined relational and attribute-valued data, utilizing schema-less attribute values to model the measurement data. We implement Bloom filter indices to enhance query response times, particularly for Boolean containment queries on large volumes of schema-less data. We then expand the architecture to support multi-model het-

erogeneous data sources and cross-model queries, replacing Bloom filter indices with space-efficient bitmaps. Additionally, we implement logical data partitioning and employ index structures that are fast, scalable, space-efficient, and flexible. We measure query latencies, precision, and storage space requirements to evaluate the multi-model architecture and the innovative indexing scheme.

We further enhance the index management module by advocating for system-level indexes that integrate data across heterogeneous sources. To achieve this, we extend the application architecture and introduce a new global index store. We move query processing to the middleware layer, reducing dependency on the underlying database engines during cross-model query execution.

Thirdly, we enhance the index management module by optimizing index creation and maintenance to align with query workloads. We propose a DB Cracking-based adaptive indexing model that addresses the challenges of indexing attribute selection and high cracking costs. Specifically, we focus on developing a novel adaptive indexing model suitable for dynamic environments such as scientific data management, which involves significant time, storage, and associated costs.

Lastly, we develop a standalone dataset exploration tool, *ARCXplore*, that executes an ETL pipeline to convert heterogeneous data files within the ARCs into an on-demand data warehouse. This allows users to run queries to explore metadata and measurement data directly. We also incorporate a query pre-optimization module into the latest architecture, where pre-optimization and index management collaborate to enhance cross-model query execution times.

1.3.1 Implementation

The PLANTdataHUB solution utilizes GitLab, a distributed version control system, to store and manage bioscience datasets that are packaged into Annotated Research Contexts (ARCs). This solution employs well-established services, including Life Sciences AAI and ORCID, for user identity management via KeyCloak. The applications developed within this framework share a common infrastructure and adhere to a unified design. In 2021, the DataPLANT consortium, along with its stakeholders, launched a production DataHUB for the plant science community. Since then, the repository has grown significantly, currently serving over 180 users who host more than 470 ARCs across various plant study disciplines.

Figure 1.3 presents the list of applications we developed as part of our research data management ecosystem. The *ARC Metadata Registry* is created using Node.js and the Express.js framework. This application is containerized and operates on Docker Swarm for high availability, integrating ARC metadata from various data hubs, and is currently deployed in production[3].

ARCXplore is an in-situ data exploration tool developed in Python that employs an off-the-shelf querying system, such as Apache Drill, for distributed query processing. It also utilizes Docker and Docker Compose for container orchestration and management.

[3]https://arcregistry.nfdi4plants.org/

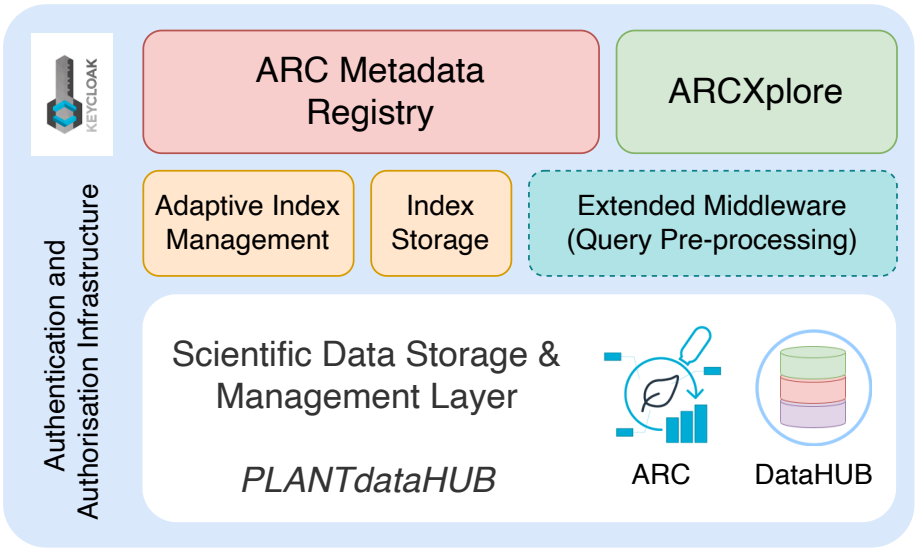

Figure 1.3: Search and exploration applications, as well as indexing methods, are implemented within the PLANTdataHUB ecosystem.

While the *ARC Metadata Registry* helps users explore ARC metadata, *ARCXplore* is tailored for querying heterogeneous data from a limited number of ARCs. Our research objective is to create a comprehensive exploration platform. This platform will enable users to query both structured metadata and semi-structured or unstructured measurement data (or raw data) together from ARCs across the data hubs. To achieve this, we implemented our adaptive index management and global indexing scheme on the existing Polystore architecture. We used Redis version 6.2.4 and developed a range of custom Redis commands to manage the index data structures and minimize data transfer between the Redis index store and the application. However, the extended middleware for query pre-processing and optimization using the global indexes is a work in progress.

1.4 Outline

This thesis is outlined as follows. In the next Chapter (2), we cover the terminology associated with scientific data management platforms. The chapter introduces fundamental bio-science concepts, including multi-omics, FAIR, and research data management. It discusses the essential data-centric features required for scientific data management systems to manage large-scale experimental metadata generated within the plant science research community. In Chapter 3, we discuss the research outcomes of the DataPLANT project, namely, end-to-end RDM solution PLANTdataHUB and Annotated Research Contexts (ARCs). We also outline the search requirements and the challenges associated with developing integrated search and analysis platforms within the scope of the RDM solution. Chapters 2 and 3 reference publications [1, 8] to present the

detailed research data management solution, to provide the background and context for introducing the main contributions of this thesis in the subsequent chapters, and to elicit the research requirements for this thesis.

In Chapter 4, we introduce our integrated metadata search application *ARC Metadata Registry*. We present the detailed architecture, design, and, importantly, how the application simultaneously connects with multiple DataHUBs and processes experimental metadata in near real-time. Here, we cover the integration of structured experimental metadata and directly reference the contents from our publication [6]. In Chapter 5, we propose using large language models for search and exploration of plant science metadata. We introduce *ArcBERT*, a BERT-based model pre-trained on plant science literature and fine-tuned with integrated omics metadata [3]. We also leverage existing indexing libraries for vector embeddings and fast nearest neighbor searches.

In Chapter 6, we provide details of the different architectures and indexing ideas we explored for querying semi-structured measurement data. In particular, we discuss the multi-model architecture and data partitioning, as well as utilizing index data structures specific to individual partitions for efficient query execution. We refer our publications [4, 5] for Chapter 6.

Chapter 7 discusses using global indexes for cross-model query evaluation in polystore architectures. We introduce the novel concept of using a key-value index store and various types of existing index data structures, namely, bitmaps and tree index structures. We outline the cross-model query execution using the key-value index store and discuss the experimental results and limitations. In Chapter 8, we extend the indexing mechanism to adapt to the query workloads. We propose a novel adaptive index management model based on DB Cracking methods, treating it as a two-player zero-sum game. We also introduce a new DB Cracking variant, leveraging the existing key-value index store and index data structures to evaluate the adaptive index management model. The contents of Chapter 7 and Chapter 8 are directly taken from our publications [2] and [7] respectively.

In Chapter 9, we introduce *ARCXplore*, a versatile standalone tool designed to extract and transform the heterogeneous data files of ARCs from dataHUBs. The tool is engineered to transform heterogeneous data files from individual ARC datasets into an on-demand warehouse, where users can directly run queries to explore both experimental metadata and measurement data. We then extend the middleware to create an integrated scientific data management framework with all the individual components. The framework utilizes a distributed query processing engine with a novel index management module for global adaptive indexes, as well as a query pre-optimizer to leverage these indexes and rewrite input queries for faster evaluation. We evaluate the feasibility of extending our solution with the existing distribution query processing engines before concluding the thesis in Chapter 10.

1.5 Publications

Below is the list of peer-reviewed publications that constitute the contents of this thesis.

[1] Jonathan Bauer, Marcel Tschöpe, Julian Weidhase, Timo Mühlhaus, Christoph Garth, Gajendra Doniparthi, Holger Gauza, Louisa Perelo, Cristina Martins Rodrigues, and Dirk von Suchodoletz. From DataPLANT's DataHUB to DataPUB (lication). In *International Workshop on Science Gateways*, 2023.

[2] Gajendra Doniparthi. Using a Key-Value Index-Store for Cross-Model Join Queries over Heterogeneous Data Sources. In Alberto Abelló, Panos Vassiliadis, Oscar Romero, and Robert Wrembel, editors, *Advances in Databases and Information Systems - 27th European Conference, ADBIS 2023, Barcelona, Spain, September 4-7, 2023, Proceedings*, volume 13985 of *Lecture Notes in Computer Science*, pages 45–58. Springer, 2023.

[3] Gajendra Doniparthi, Shashank Balu Pandhare, Stefan Deßloch, and Timo Mühlhaus. ArcBERT: An LLM-based Search Engine for Exploring Integrated Multi-Omics Metadata. In *17th International Workshop on Science Gateways, 17-19th June 2025, London, UK*, 2025.

[4] Gajendra Doniparthi, Timo Mühlhaus, and Stefan Deßloch. A Bloom Filter-Based Framework for Interactive Exploration of Large-Scale Research Data. In Jérôme Darmont, Boris Novikov, and Robert Wrembel, editors, *New Trends in Databases and Information Systems - ADBIS 2020 Short Papers, Lyon, France, August 25-27, 2020, Proceedings*, volume 1259 of *Communications in Computer and Information Science*, pages 166–176. Springer, 2020.

[5] Gajendra Doniparthi, Timo Mühlhaus, and Stefan Deßloch. A Hybrid Data Model and Flexible Indexing for Interactive Exploration of Large-Scale Bioscience Data. In *New Trends in Database and Information Systems - ADBIS 2021 Short Papers, Estonia, August 24-26, Proceedings*, 2021.

[6] Gajendra Doniparthi, Timo Mühlhaus, and Stefan Deßloch. Integrating FAIR experimental metadata for multi-omics data analysis. *Datenbank-Spektrum*, 24(2):107–115, 2024.

[7] Gajendra Doniparthi, Tim Otto, and Stefan Dessloch. On Modeling Adaptive Index Management as Adversarial Search. In *2024 IEEE International Conference on Big Data (BigData)*, pages 549–558, Los Alamitos, CA, USA, December 2024. IEEE Computer Society.

[8] Heinrich Lukas Weil, Kevin Schneider, Marcel Tschöpe, Jonathan Bauer, Oliver Maus, Kevin Frey, Dominik Brilhaus, Cristina Martins Rodrigues, Gajendra Doniparthi, Florian Wetzels, Jonas Lukasczyk, Angela Kranz, Björn Grüning, David Zimmer, Stefan Deßloch, Dirk von Suchodoletz, Björn Usadel, Christoph Garth, and Timo Mühlhaus. PLANTdataHUB: a collaborative platform for continuous FAIR data sharing in plant research. *The Plant Journal*, 116(4):974–988, 2023.

Chapter 2

Background

In this chapter, we cover the foundational topics of RDM and FAIR and discuss the essential features required in a research data management system for handling vast amounts of bio-science datasets. We focus on the data-centric aspects and limit our discussion to data standardization, packing, and preservation.

2.1 Multi-omics

Multi-omics refers to the integration and analysis of various types of omics data to comprehensively understand biological systems [1]. The different layers of omics include:

- **Genomics**: This area focuses on sequencing and analyzing the complete DNA sequence of an organism, providing insights into genetic variations, mutations, and the overall genetic blueprint.
- **Transcriptomics**: This field analyzes RNA transcripts to understand gene expression patterns and how they respond to different conditions or treatments.
- **Proteomics**: This involves the large-scale study of proteins, including their abundance, modifications, and interactions within the cell.
- **Metabolomics**: This area examines small molecules and metabolites, the downstream products of cellular metabolism, offering insights into metabolic pathways.
- **Epigenomics**: This field investigates modifications to DNA and histones that regulate gene expression without changing the underlying genetic sequence.

Each omics layer provides valuable information about specific aspects of biological systems. However, combined, they offer a more holistic understanding of cellular processes and disease mechanisms.

2.2 FAIR and RDM

Research Data Management (RDM) and FAIRness are essential for various bioscience communities to enhance the reproducibility, visibility, and accessibility of their diverse research data. Research Data Management outlines methods and processes for collecting, maintaining, using, preserving, and sharing complex research data. Within the biological science community, there can be several distinct RDM approaches, each with multiple stages. Process-oriented management at each stage in the data life cycle not only ensures the discoverability of

research data but also aids in its longevity [2]. RDM provides methods and storage solutions for packaging and preserving bio-science datasets as self-contained research output units.

The FAIR data principles act as guidelines to improve the Findability, Accessibility, Interoperability, and Reusability of these research output units [3]. The goal is to encourage researchers to build detailed, annotated studies that utilize qualified references to other studies, producing machine-readable datasets. These datasets should be identified through global unique identifiers and made accessible through standardized communication protocols. FAIR compliance is critical in multi-omics research, where datasets are complex and often require integration with other datasets.

- **Findable**: Humans and machines should easily locate data. This requires persistent identifiers (such as DOIs), standardized metadata, and proper indexing in public repositories.
- **Accessible**: Data should be accessible, meaning authorized users can retrieve it without unnecessary barriers. Even when data cannot be openly shared due to privacy concerns (e.g., clinical data), the metadata should still be available to guide potential collaborators.
- **Interoperable**: Data should be presented in formats that allow it to be integrated with other datasets. This often involves using standardized formats, vocabularies, and ontologies that enable cross-dataset comparisons and analyses.
- **Reusable**: To maximize the value of research data, it must be reusable by others. Data should be well-documented, with clear usage licenses and sufficient metadata to enable others to replicate and extend the work.

A successful RDM that follows the FAIR data principles should be able to generate research output units that bundle experimental metadata and measurement data together, along with the mechanisms for their creation, maintenance, and sufficient information to process the data. Each research output unit is identified by a unique identifier resolving to a machine-readable information record.

2.3 Experimental Metadata

From a data perspective, a typical biological experiment can be broadly characterized into four stages, as shown in Figure 2.1. The first stage includes curation of high-level *metadata*. It is the minimum required information to describe an experiment, allowing researchers to understand the biological and technical origins of the data. It is essential to provide a comprehensive project description, including the project title, people involved, publications, goals, and contextual information. The sequence of events or steps should be recorded before or during the experimental procedure. This includes information about the source and characteristics of the study sample, study protocols, sample preparation techniques used, and any other required information to prepare the instrument for the actual measurement.

The second stage involves conducting the experiment and collecting the raw

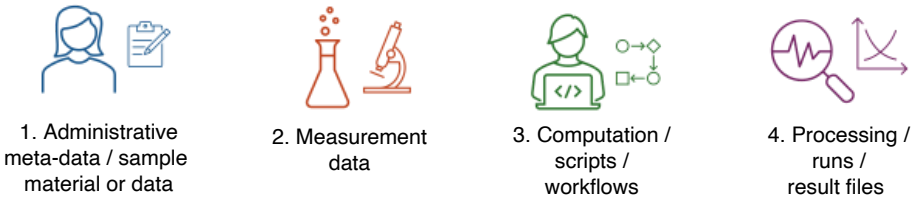

Figure 2.1: The stages of a typical bio-science experiment from a data perspective.

output data produced by the instrument. The format of these files depends on the instrument used and the type of experiment performed. These data files are generally expected to be large and have project-specific storage requirements. In most cases, the instruments generate data in a proprietary format, which must be converted into standard formats for further analysis using specialized software for identification and quantification. This information is usually maintained at the level of each sample tested.

The third and fourth stages of the experiment involve creating computational pipelines that include workflows, analysis scripts, or tools to ensure the pipelines are reproducible. These pipelines contain code generated from common workflow languages, including sets of processed data files created during pre-processing, data cleaning steps, and the tools used. Researchers perform computational analysis using the sets of processed data as input to the analysis steps from single or multiple studies. The system enables researchers to perform computational analyses collaboratively using workflow engines. The output datasets and other artifacts from a computational analysis can form soft metadata for further integrated analyses.

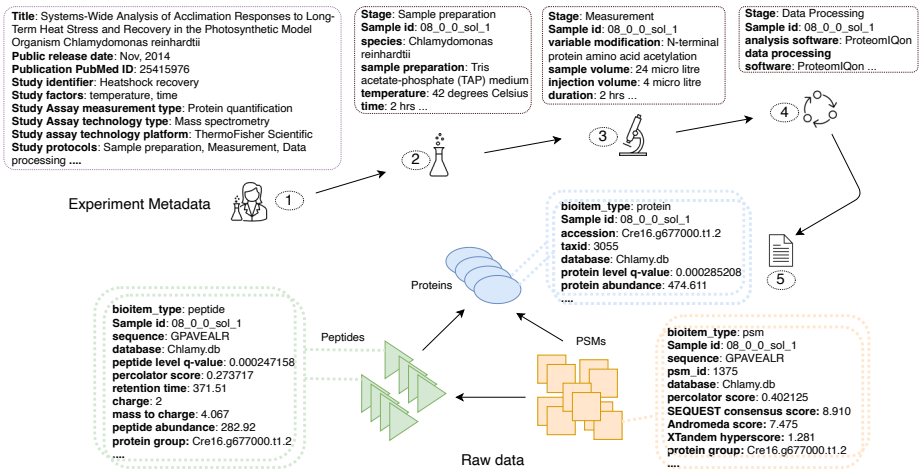

Figure 2.2: Example data from a sample Proteomics experiment

For the reproducibility of a bioscience experiment, it is equally important to capture the provenance of the metadata, the steps followed in the computational analysis, the tools used, the workflows, and the scripts, among other details.

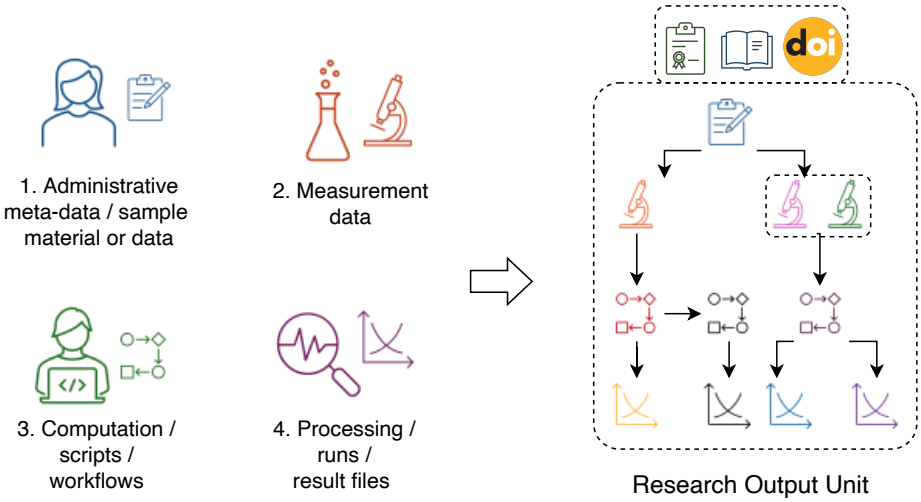

Figure 2.3: The standardized data generated at each stage of a typical bio-science experiment naturally form a hierarchical structure. For findability, the resulting research output units are granted a unique object identifier and associated metadata, including relevant publications.

Although capturing all information related to the experiment is preferred, annotating and managing large volumes of measurement data can be a challenging task. Achieving end-to-end data coverage requires significant effort in integrating, annotating, and automating data capture at every stage of the experiment. A sample dataset from a proteomics experiment is shown in Figure 2.2.

2.4 Research Output Units

The increasing demand for reproducible research results requires a standardized approach to package the research results, associated metadata, and software into one publishable entity. The data generated at each stage must be standardized and annotated, and the relationships among the data files from research processes are preserved to form a research output unit that can be uniquely identified, as shown in Figure 2.3. Process-oriented management at each stage in the data life cycle ensures the discoverability of the datasets, which, in turn, aids in the longevity of the research. There are a few key research data management topics to consider before implementing any data policies to manage complex bio-science research data [2, 4–6].

2.4.1 Data Standardization

In Research Data Management (RDM), data standardization is crucial for maintaining interoperable and reproducible results in public and private external repositories. Utilizing industry-standard data formats simplifies the integration of data with existing systems. The data collected from various experiments can be heterogeneous and diverse, depending on factors such as the discipline, type of experiment, equipment used, and procedures followed.

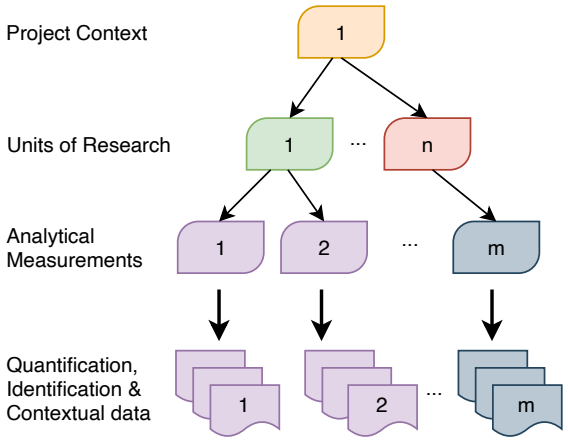

Figure 2.4: ISA Metadata Structure

Standardizing these heterogeneous datasets for multi-omics analysis primarily relies on adopting a widely accepted bio-science metadata standard. This process is essential for ensuring reproducible results in both public and private repositories. Additionally, using industry-standard data formats reduces the complexity of integrating with external systems. However, research groups often use different specifications, necessitating the use of conversion tools to achieve metadata interoperability. Therefore, metadata standardization ensures that research outputs can be stored, searched, and analyzed while maintaining interoperability and reusability. Numerous bio-science metadata specifications are available for structuring, curating, and enabling data sharing and integration [7–10].

For instance, the Investigation, Study, Assay (ISA) framework [7] is a widely used extensible framework that organizes metadata into categories: Investigation (the project context), Study (a unit of research), and Assay (analytical measurement) in the form of a hierarchy as shown in Figure 2.4. The ISA abstract specification [11] provides a standard format for representing experimental metadata tailored to specific project requirements, facilitating information exchange across project groups and public repositories.

2.4.2 Data Annotation

Controlled vocabularies and ontologies are crucial in tracking predefined and standardized parameter names within experimental metadata [12]. In particular, bio-science experiments benefit from these tools as they provide semantics, standard syntax, and formalism to the terms that scientists use. The contextual attributes, such as numerical and categorical data, are determined by the relevant vocabulary depending on the type of scientific experiment.

The extent to which datasets are standardized and annotated depends on the availability of tools for data curation within the specific data management systems used for the study, as illustrated in Figure 2.5. Typically, software tools that access ontologies with spreadsheet-style user interfaces enable scientists to curate and annotate experimental metadata effectively. However, standardizing

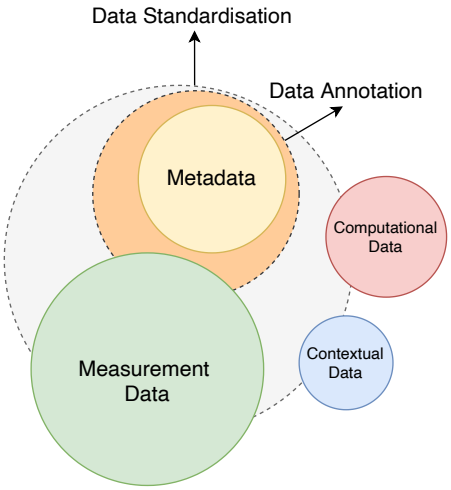

Figure 2.5: The extent of standardization of research data can vary. While experimental metadata can be fully standardized and annotated, annotating the measurement and contextual data within the dataset can be a challenging task.

and annotating comprehensive datasets—including measurement data, workflow scripts, contextual information, and output files—poses significant challenges. This suggests that additional steps are required to annotate the data for improved searchability, particularly regarding non-metadata information.

2.4.3 Data Relationships

The research datasets generated from individual experiments naturally form a hierarchical structure. At the top of this hierarchy is the metadata, followed by the sets of measurement data files generated for each tested sample, and at the bottom are external data files, which are artifacts of the computational pipelines, as shown in Figure 2.4. It is crucial to have a research data management solution that standardizes naming conventions and folder structures for organizing both metadata and measurement data files. Additionally, maintaining the relationships among workflow scripts, input data, configurations, output data, and other artifacts within this hierarchy is important. Ensuring that the metadata includes references that point to the exact folders and files within the dataset, or that provide direct links to external artifacts, is essential for guaranteeing the integrity and self-containment of the bio-science datasets.

2.4.4 Data Packaging

Data packaging involves bundling comprehensive information about a bio-science experiment, which includes experimental metadata, data production methods, software used for data analysis, computational workflows, output files, and descriptive text. There are three key aspects to consider when packaging research datasets in a machine-readable format. First, developing a specification for serializing the experimental metadata into a platform-independent format is es-

sential. This specification should capture the relationships within the dataset, explaining how a specific method generates output files and providing a physical link to the corresponding text file that describes the process.

Second, packaging the research output units with additional top-level information ensures reliable data identification, interpretation, and processing. This includes machine-readable unique identifiers, information about authors and research organizations, licensing details, direct links to relevant publications, specifics about the data packaging format, and references to standardized experimental metadata files. Third, standard archival and export formats must be employed to store the packaged research data in public or private repositories. RO-Crate and FAIR Digital Objects are well-known solutions for data packaging.

RO-Crate RO-Crate is an approach to package and aggregate research artifacts with their metadata and relationships [13]. RO-Crates are self-contained and rely on the Linked Data principles to maintain the top-level structured *RO-Crate Metadata File*, describing the bundled research artifacts. The RO-Crate utilizes Schema.org annotations in JSON-LD and can be archived using generic formats, such as BagIt, ZIP, and git.

FAIR Digital Objects An RDM platform using FAIR principles provides an ecosystem to progressively transform research data into Fair Data Objects (FDOs), which are self-contained, technology-independent digital objects that unify critical information about a research project [14]. An FDO bundles the data and the mechanisms for its creation, maintenance, and sufficient information to process it. Each FDO is identified by a unique Persistent Identifier (PID) that resolves to a machine-readable PID information record, making the contents relevant. A research project with journal publications referencing FDOs would be a stand-out with research results that are visible, accessible, and reproducible.

2.4.5 Data Preservation

Research data management (RDM) typically starts at individual research institutes, where most data is curated and stored locally. However, it is essential for publications to preserve research results and data in centralized repositories, making collaboration and data sharing vital. Endpoint repositories play a crucial role in safeguarding and disseminating research datasets. These method-specific repositories are designed to store datasets from various omics studies and provide interfaces for uploading experimental metadata, measurement files, and other artifacts separately or as bundled packages.

Repositories typically offer metadata search functionalities, allowing users to easily browse and locate specific datasets. By using bundled packages, there is potential for developing method-agnostic repositories, as these packages provide a universal structure for organizing datasets from individual omics studies. Furthermore, publishing the overarching metadata that describes the packaged datasets on search platforms like Google Dataset Search [15] enhances their discoverability and visibility. To ensure data security, repositories must have an Authorization and Authentication infrastructure.

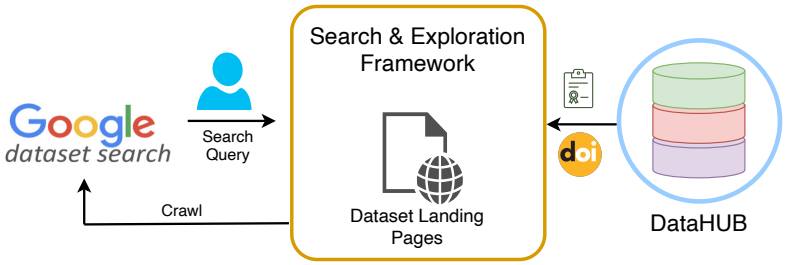

Figure 2.6: The process of citing a dataset involves several steps. DataHUBs are responsible for maintaining published datasets, while the search application creates landing pages that provide direct links to download these datasets. Public search engines, like Google Dataset Search, crawl the metadata related to the datasets embedded into the landing page, making it easier for users to discover them.

2.4.6 Data Citation

Dataset citation plays a crucial role in bioscience research, facilitating the discovery of relevant datasets and promoting their reuse [16]. It is essential to encapsulate datasets into individual research output units and provide them with the minimum required information outlined by data citation principles. This information includes a unique identifier, such as a Digital Object Identifier (DOI), along with the author, title, year of publication, and other relevant details, all of which are maintained on a specific dataset landing page.

These individual landing pages will serve as comprehensive resources, providing clear guidelines on how to cite the dataset, detailed information about its content, and specifics regarding data accessibility and licensing. Additionally, public search engines can index these landing pages, allowing them to appear in search platforms like Google Dataset Search [15] as shown in Figure 2.6. To enhance the visibility of datasets in public search engines, it is important to include additional markup on the landing pages. This extra metadata, coupled with journal publications that reference the datasets, will increase the visibility, accessibility, and reusability of research results.

Chapter 3

PLANTdataHUB

PLANTdataHUB [17] is a comprehensive FAIR solution designed for managing research data in plant science. In this RDM solution, the units of research output are referred to as *Annotated Research Contexts*, in short, ARCs. The PLANTdataHUB ecosystem, illustrated in Figure 3.1, emphasizes interoperability between data and services, community engagement, training, knowledge transfer, and long-term sustainability.

This platform supports participating research groups by providing digital resources and personal expertise to collaborate effectively and produce ARCs (FAIR digital objects) that are self-contained, interoperable, and reproducible. PLANTdataHUB provides a range of tools to facilitate ongoing FAIR collaboration, built on the ARC structure. Importantly, the solution utilizes GitLab to store and version individual ARCs. Some of the tools include

- on-premise git-based tools to help data stewards initialize and curate ARC folder structure and initial metadata files [18].
- annotation tools to assist in curating ontology-driven metadata annotation, accessing the required external ontologies [19].
- a range of converter tools to generate standardized export formats from the ARCs, pivoting on the existing metadata standards and abstract models.
- a data management plan generator to guide the writing process of a DMP (data management plan) in user-desired format for respective funding agency [20].

3.1 Annotated Research Context

The core idea behind Annotated Research Contexts (ARC) is to follow a fixed folder and file layout to organize experimental metadata, measurement data, workflows, software, and external files in a generic and interoperable format, ensuring the reproducibility of each experiment within a research project [21]. The format of ARC is standardized and flexible, allowing for adaptation and accommodation of simple to complex project scenarios across any omics studies. As shown in Figure 3.2, ARCs offer a single point of entry logic for both data management and computation. It provides end-to-end data coverage and easy-to-perform integrity checks to maintain the ARC quality.

The ARC specification [22] is designed to be flexible and can accommodate experimental metadata represented in any known metadata standard. The initial ARC specification for plant research data, in particular, allows for well-known standards such as ISA. The abstract model for the ISA standard is implemented in hierarchical ISA-JSON file format [23]. The ARC specification also utilizes

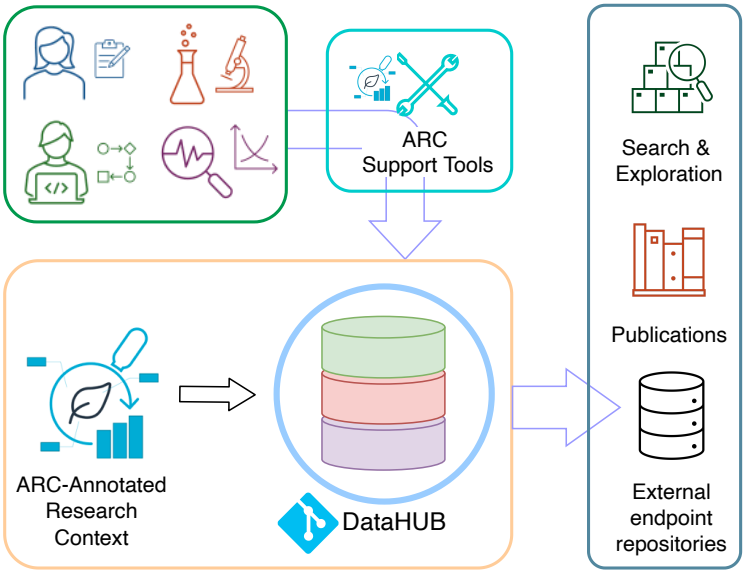

Figure 3.1: PLANTdataHUB model using ARCs as FAIR digital objects. Data stewards curate ARCs using support tools and maintain them in project-specific on-premise or remote DataHUBs. Additionally, ARCs can be exported into interoperable formats for use in external repositories or added as direct references in publications.

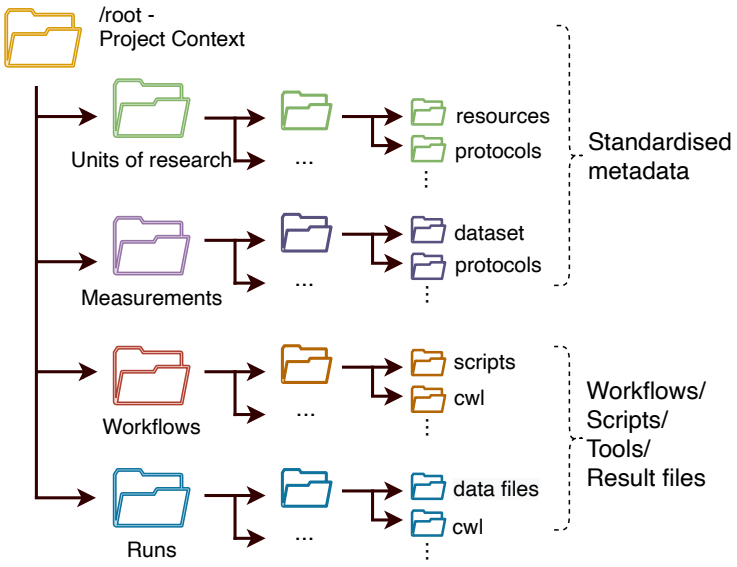

Figure 3.2: ARC folder specification packaging standard format metadata with workflows, scripts for computational pipelines, and result files/artifacts from workflow executions.

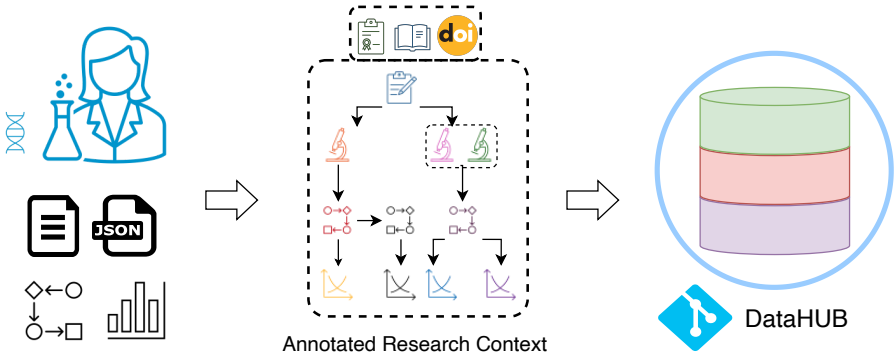

Figure 3.3: The heterogeneous datasets from individual experiments are standardized into ARCs using support tools and maintained in project-specific on-premise or remote DataHUBs.

a common workflow language for workflows and scripts to construct the computational pipelines. The artifacts from workflows and computational analysis go into their respective run folders. The ARC can also be compiled to other packing methods, such as RO-Crate [13].

The process of data management and digital object generation inherently yields additional ARC Object-level metadata, which, in turn, enriches data analyses. This may include the digital object identifier (DOI) assigned to a particular version of an ARC, details about the analyst or the data steward who recently modified the ARC in the repository (could be the same or different from the actual scientist who conducted the experiments), the current status of the ARC, its visibility (public or private), etc.

3.2 DataHUBs

Research data management typically begins at individual research institutes, where most data is curated and stored locally. However, it is crucial for publications to maintain research results and data in centralized repositories; hence, collaboration and data sharing are pivotal. In the PLANTdataHUB model, centralized repositories are known as *DataHUBs* as shown in Figure 3.3. They offer versioning of ARCs and make it easy to search and discover through standardized API access. The ARC model and associated tools enable easy initialization and curation of ARCs locally, while maintaining privacy during ongoing experiments. Research groups can also maintain private ARCs directly in associated DataHUBs, with strict access control restrictions. When the results are ready for publication, private ARCs can be made publicly accessible.

The PLANTdataHUB solution utilizes a distributed version control system (GitLab) to store and manage individual ARCs. It provides flexibility in maintaining each iteration of a design-test-repeat cycle of a laboratory experiment, thus avoiding the need for a new customized versioning tool. DataHUBs can be deployed on-premise or remotely, depending on the research groups' data sharing and security policies. Additionally, PLANTdataHUB provides various sup-

porting tools to enable easy curation, collaboration, and maintenance of ARCs on-premise and in remote dataHUBs.

A robust data security framework within the PLANTdataHUB eco-system is expected to complement ARCs, DataHUBs, and supporting tools & applications. A practical and user-friendly authentication and authorization infrastructure (AAI) is required to enforce data access policies seamlessly. PLANTdataHUB solution uses a Keycloak instance [24] as a gateway AAI to streamline user identification across the DataHUBs.

3.3 Related Work

A wide range of research data management platforms exist, catering to different needs and research disciplines, all working towards making research data FAIR. This section highlights recent state-of-the-art instead of presenting an exhaustive list. In particular, we group the related work that serves the core purpose of making research data discoverable, shareable, and reusable into three types: end-point repositories, standalone information systems, and scientific data management platforms.

Popular endpoint repositories, such as MetaboLights [25, 26] and PRIDE [27], primarily focus on storing and sharing data among the global scientific community. Their goal is not to manage scientific data throughout the entire experimental lifecycle; instead, they facilitate open access and public sharing of research data. These repositories typically provide user search features to facilitate searching and downloading datasets.

Zenodo [28], developed by CERN, is another open-access repository with a user-friendly platform that enables researchers across disciplines to publish datasets, software, and academic papers, providing DOIs for citation. Zenodo provides DOIs (Digital Object Identifiers) for all uploads, ensuring that datasets and other research outputs can be easily cited and referenced in future publications. The platform offers a straightforward interface that allows researchers to upload their data (including datasets, software, and multimedia files) and metadata. Furthermore, Zenodo offers seamless integration with ORCID, which helps link researchers to their published works. However, it lacks advanced metadata handling and collaboration features, making it more suitable for archival purposes rather than research data management. Figshare [29] is another open-access repository that emphasizes the support of open publication of research data and encourages collaboration. Like Zenodo, Figshare offers DOI assignment for datasets, ensuring that shared data can be appropriately cited in future research. However, Figshare is notable for its data visualization and open research sharing capabilities.

Stand-alone information systems such as openBIS [30] (Open Biological Information System) offer mechanisms for storing and managing research data. openBIS enables users to collect, integrate, share, and publish data, as well as connect to data processing pipelines. It is highly customizable and tailored for various data types and technologies. Key features of openBIS include a hierarchical data organization structure, flexible metadata management, robust data provenance tracking, and scalability to accommodate extensive datasets. It leverages

a hybrid data repository, featuring a relational database for index information and experimental metadata and a flat-file data store for raw and result data. Users must use a web interface, command-line tools, and APIs to interact with an openBIS instance. Although openBIS supports the data management needs of small research groups to large collaborative projects, it lacks the key features of PLANTdataHUB, namely continuous quality control and method-agnostic hubs for data sharing and integration.

The increasing need for collaborative research data management paved the way for systems that allow for structured management of complex bioscience datasets while adhering to FAIR principles. FAIRDOMHub is a repository and collaborative environment for systems biology research [31]. It enables researchers to organize, share, and publish various research assets, including data, models, protocols, and publications. The platform supports project-based organization, allowing for the structured management of complex datasets and facilitating research collaboration. By adhering to FAIR principles, FAIRDOMHub enhances data discoverability and reusability, promoting transparency and reproducibility in systems biology studies.

Flow is a web-based platform designed to address challenges in managing and analyzing large-scale bio-science datasets. It supports all stages of the research project, and different users can use it at different stages of the bio-science data life cycle [32]. Data on Flow is organized around several key object types, including projects, samples, data, and executions. To ensure findability, all data includes a unique and persistent ID, is accompanied by rich metadata, and is searchable via the web platform GUI or API. It integrates with version-controlled Nextflow pipelines, allowing users to execute complex bio-science workflows without requiring command-line expertise. This design supports reproducible and scalable analyses, enabling users to run parallel tasks efficiently. Flow can be deployed on local systems or cloud services, offering flexibility in computational environments. Flow links data with standardized metadata to ensure data integrity and reproducibility, including experimental conditions, quality control measures, and analysis parameters. Similar to PLANTdataHUB, Flow aims to provide a comprehensive solution for data analysis and management, thereby democratizing bio-science research. This enables experimental biologists to analyze, manage, and share their data effectively without requiring extensive computational expertise.

While FLOW focuses on workflow execution and metadata tracking, FAIRDOMHub is more suited for systems biology research, where structured data management and collaboration are key. FAIRDOMHub enables project-based organization, linking experimental data, models, and computational analyses while adhering to FAIR principles to ensure data findability and reusability. It is particularly beneficial for large-scale, collaborative research projects that require interlinking different types of biological data. On the other hand, PLANTdataHUB uses Annotated Research Contexts (ARCs), which serve as structured digital objects that facilitate data versioning, referencing, and sharing within plant research projects. Unlike FLOW, which emphasizes pipeline execution, PLANTdataHUB prioritizes FAIR data management and long-term sustainability of research data. In summary, FLOW excels in scalable sequencing data analysis, FAIRDOMHub provides a collaborative environment for systems biology,

and PLANTdataHUB supports structured data management in plant research. The choice among these platforms depends on whether a researcher needs a workflow execution tool (FLOW), a collaborative systems biology hub (FAIR-DOMHub), or a plant-specific research data repository (PLANTdataHUB).

Chapter 4

Integrated Metadata Search

One of the most challenging tasks in bio-science data management is standardizing and integrating study-specific datasets from different repositories to extract biologically relevant information. Typically, multi-omics data search is not a default function used in study-specific data management solutions that cater to specific omics data. Instead, the search functionalities are included as an additional layer, necessitating interchangeable data formats to obtain experimental metadata and associated contextual information from different study-specific repositories. The metadata must then be transformed into a common data representation, annotated, and indexed so users can query it through APIs or a web interface. Common user queries might include searching for a list of ARCs with a specific data steward in the study author list, identifying ARCs that have utilized a particular material sample from a designated plant species, or finding ARCs that have employed a specific type of high-throughput technology. When integrated with multiple on-premise or remote DataHubs that host ARCs across various omics domains, the results of these queries provide valuable insights for cross-domain research and analysis.

We can broadly categorize the ARC search requirements from the plant science community into two distinct research tasks.

- **Research Task 1**: Integrated metadata search from the top layers of the ARCs, i.e., standardized experimental metadata and ARC object-level metadata.
- **Research Task 2**: Exploration of both experimental metadata, associated measurement data (which is processed and annotated), and artifacts from iterations of computational analysis within an individual ARC or from a set of selected ARCs (same or from different omics domains).

Key design features of a search application capable of integrating multiple DataHUBs and implementing the aforementioned research tasks include:

- Flexibility in integrating multiple DataHUBs supporting different integration strategies and data exchange protocols.
- Robust data interchange mechanism, preferably with standardized data exchange formats.
- Effective and comprehensive data access through APIs and web-based user interfaces.
- Cloud-native application design with high availability.
- A user identification system complementing external authentication and authorization infrastructure.
- End-to-end logging for user support and troubleshooting.

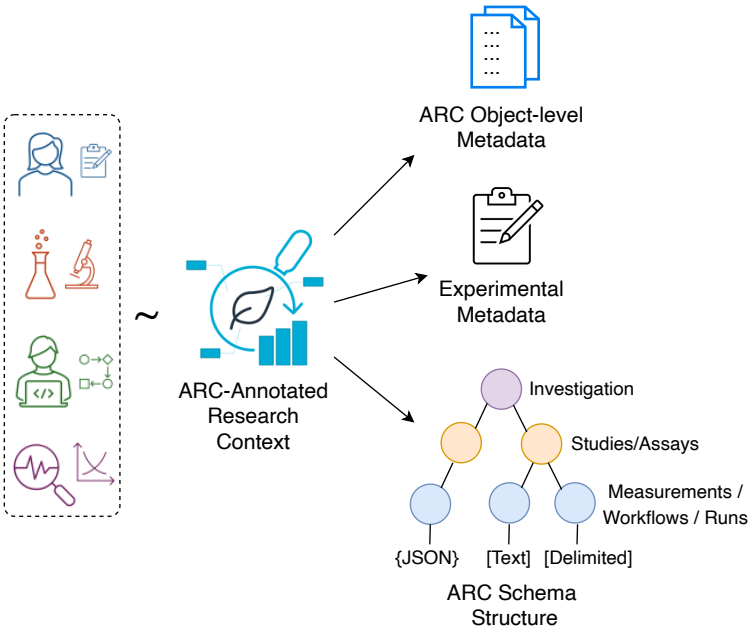

Figure 4.1: Representation of the ARC Data & Schema Models.

4.1 Data Model

An effective data model that balances functional requirements with complex query processing is essential for integrated data exploration. Searchable data from the ARCs can be analyzed using multiple data models. Furthermore, a schema structure that defines the relationships between experimental metadata and measurement data files within the ARCs is crucial for directly locating queryable data files.

S.No.	File type	Extensions	Data model	Database
1.	Measurement data, Output files from workflow executions	CSV, TSV, etc	Relational	Relational Databases
2.	Text descriptions, Scripts, Code	MD, TXT, SH, CWL, etc	Text	Text Search Engines
3.	Experimental metadata, Configuration files Object-level metadata	JSON, JSON-LD	Document	Document Databases

Table 4.1: List of data file types and associated data models

Figure 4.1 illustrates the distinction between experimental metadata and the ARC schema structure.

Metadata - The ARC specification requires that experimental metadata be standardized and uniform across different studies. In PLANTdataHUB, the experimental metadata is maintained in standard formats for easy manual curation. ARC tools convert this metadata into a standardized JSON format, adhering to the ISA-JSON abstract model or RO-Crate metadata JSON-LD.

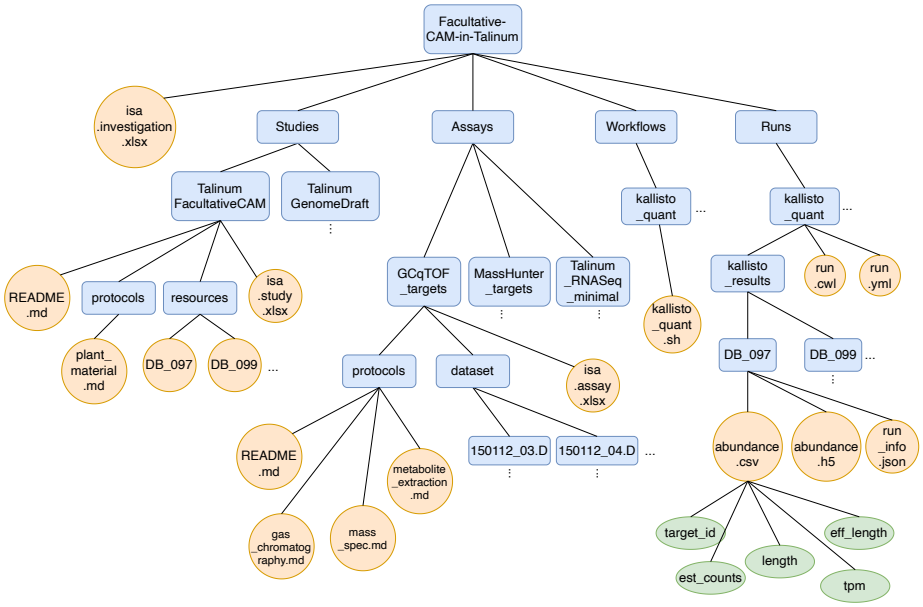

Figure 4.2: A snippet of the ARC Graph instance for the real-world ARC *Facultative-CAM-in-Talinum*. The box-shaped nodes are the ARC directories, the circled nodes are the files, and the oval-shaped nodes are the header fields.

Object-level metadata is the surface-level identification data captured for individual ARCs to enhance findability. It includes the ARC ID, DataHUB location, ARC owner, direct download link for dataset citation, last modified date, and more. All of this data is modeled in JSON format.

Data files encompass measurement data, output files from computational workflows, text descriptions, configurations, scripts, code, and additional content. These files are identified based on a predefined list of file extensions and are automatically categorized into their respective data models. A table of well-known file types and their associated data models is presented in Table 4.1.

ARC Schema - The standardized ARC folder hierarchy of each ARC instance helps identify the relationships among data files. The directories and files of each ARC can be naturally modeled in a graph format. In the ARC instance graph, nodes can represent directories, files, and header fields within the files. A directory node may connect to other directory or file nodes, reflecting the hierarchical structure. Additionally, file nodes can connect to their respective header field nodes. Importantly, these file and field nodes can be linked to ontology terms, facilitating schema structure queries using these controlled vocabulary terms.

Figure 4.2 illustrates a snippet of the ARC graph instance from the real-world ARC titled *Facultative-CAM-in-Talinum*. The nodes, specifically the directories, files, and fields, are color-coded for clarity. The ARC instance graph adheres to the hierarchical ARC folder structure and is created in two stages: initially, the instance graph is developed based on the native file system structure. In the second stage, related field nodes are added based on the file headers for

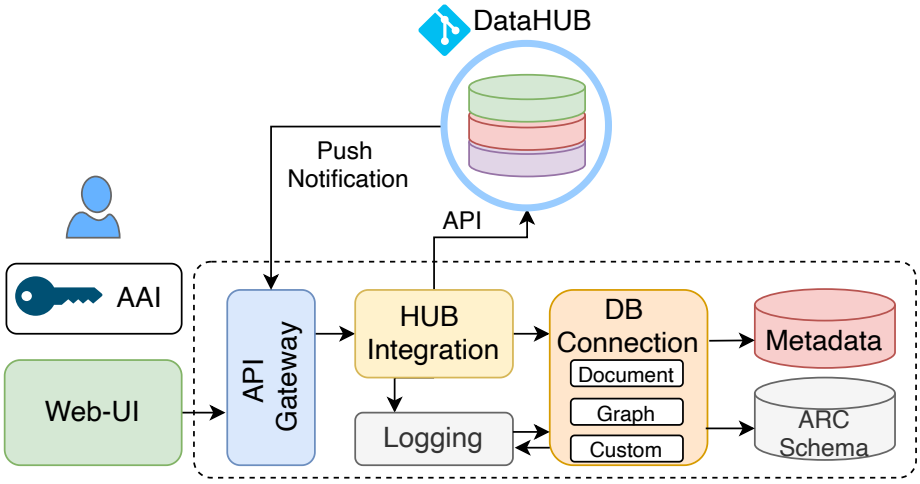

Figure 4.3: The architecture of ARC Metadata Registry application. The core functionalities are implemented as microservices.

file nodes with recognized file types (such as CSV and TSV). The field nodes can be associated with known ontology term nodes independent of the field name, which aids in identifying similar fields while overcoming linguistic and semantic variations in the field names.

4.2 ARC Metadata Registry

We introduce *ARC Metadata Registry*, a cloud-native application for integrated search and analysis of experimental metadata within the plant science community. The registry application within the PLANTdataHUB solution is designed to integrate multiple DataHUBs simultaneously and provide a consolidated, real-time view of data from the top layers of the ARCs across all DataHUBs. Most importantly, users can simultaneously search and explore data from different ARCs and across various DataHUBs, thereby providing a platform for multi-omics metadata exploration. It includes functionalities to search both ARC Object-level and standardized metadata, addressing *Research Task 1* in Section. 4.

4.2.1 Microservices

ARC Metadata Registry follows a multi-service architecture. The application modules are built as individual microservices that are loosely coupled and securely communicate with each other. Figure 4.3 shows the architecture and each service of the ARC Metadata Registry. API gateway service is the entry point that authorizes and authenticates incoming API requests. ARC Metadata Registry exposes a range of secure APIs to access both ARC Object-level and standardized metadata, i.e., the existing version allows for ISA & RO-Crate metadata serialized in JSON & JSON-LD, respectively.

HUB Integration is a crucial service that enables the integration of multiple

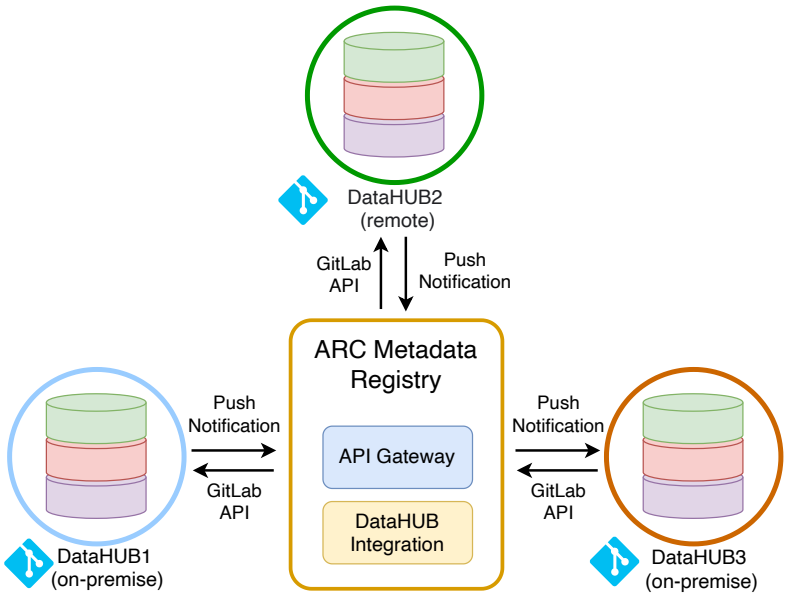

Figure 4.4: ARC Metadata Registry integrated with multiple on-premise and remote DataHUBs hosting ARCs.

DataHUBs and instantly downloads experimental metadata from ARCs. Each participating DataHUB requires a one-time setup to establish a two-way data exchange with the ARC Metadata Registry. Once set up, the ARC Metadata Registry receives real-time push notifications from DataHUBs whenever an ARC repository update occurs in GitLab, and the HUB Integration service downloads and processes the latest changes.

The DB Connection service abstracts the underlying databases, persisted metadata, and ARC schema by exposing a set of internal APIs for other registry services to access. In particular, the service is extensible to databases supporting various data models. It can also interface with multi-model architectures, such as Polystore systems or distributed query processing engines, thereby hiding data model heterogeneity. Since the experimental metadata is modeled as standardized JSON documents, any document database with decent text search functionality is the best fit. However, we use a separate graph data store for the ARC schema data.

The Web UI is a standalone service that provides web pages for users to search and explore the experimental metadata. By default, the web pages only show public ARCs. However, users can authenticate themselves through external AAI to access public (owned by any user) and private ARCs (owned by respective authenticated users). The UI also maintains the dataset landing pages for individual ARCs, enabling public search engines to index these pages and allow them to appear in search results.

The logging service logs all interactions among the other services in the ARC Metadata Registry application. Specifically, it maintains a stage-wise log of interactions between the HUB Integration service and the DataHUBs.

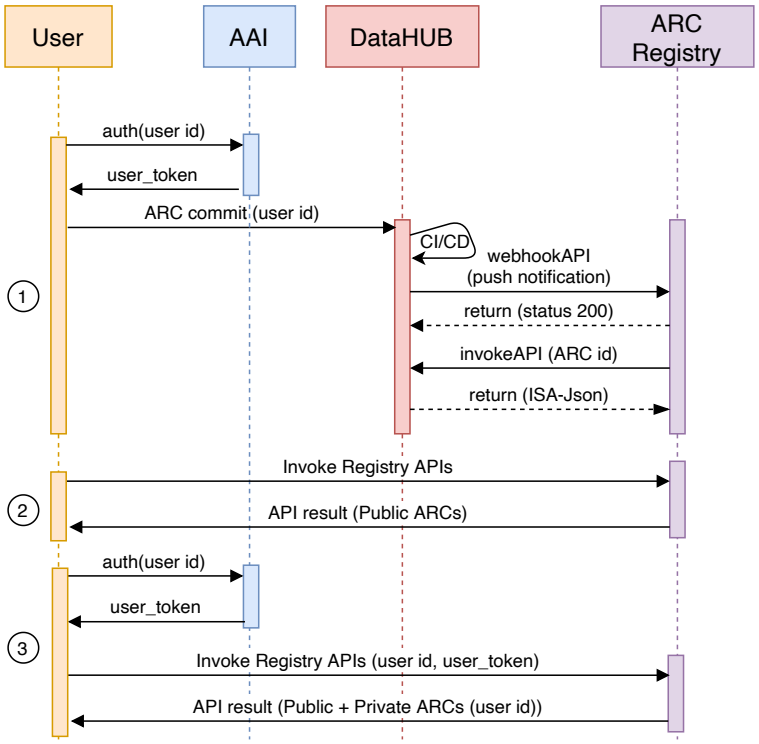

Figure 4.5: Sequence diagram depicting the interactions among the entities of RDM, i.e., Users, AAI, DataHUB, and ARC Metadata Registry. 1) Automated data exchange using the CI/CD pipeline between the DataHUB and ARC Metadata Registry. 2) A Query API that returns all Public ARCs when the user is not authenticated with AAI. 3) A Query API that returns Public and user-specific Private ARCs when authenticated.

4.2.2 DataHUB Integration

There are two essential aspects to integrating DataHUBs hosting ARCs. The first aspect is generating real-time notifications whenever a user updates an ARC. For instance, changes to an existing workflow in an ARC, adding sets of result files after executing a workflow, changes to the administrative metadata, etc. The second aspect is accessing ARCs from the DataHUBs through APIs while abstracting the mode of communication. DataHUBs and the ARC Metadata Registry are agnostic to the underlying data transfer protocols, such as HTTP/S and SFTP.

PLANTdataHUB utilizes customized GitLab for DataHUBs to manage ARCs, leveraging the built-in features of GitLab. ARCs are maintained as individual Git repositories within the GitLab servers. This makes it easy to set up system-level continuous integration and delivery (CI/CD) pipelines that can invoke ARC Metadata Registry APIs upon commits to individual ARC repositories. Figure 4.4 shows how the registry can be integrated with multiple GitLab servers simultaneously. After initial user authentication, the data exchange happens as follows

1. Any fresh user commits to individual ARC repositories triggers a CI/CD pipeline that performs two tasks, as shown in the first part of Figure 4.5. Firstly, as part of the pipeline, the internal script scans for updates in the metadata sections of ARC and generates a *JSON artifact* (in the exchange formats of either ISA-JSON or RO-JSONLD) within the repository.

2. The pipeline then triggers the webhook API of the ARC Metadata Registry with a notification in the form of a JSON payload containing ARC Object-level metadata, i.e., ARC repository ID, GitLab username, email ID, file commits, additions, etc.

3. The incoming webhook API request is routed to the HUB integration service, which parses ARC Object-level metadata. The service identifies the source DataHUB details and, in turn, connects to the DataHUB and downloads the JSON artifact from the respective ARC repository using generic GitLab APIs. It then parses the artifact and invokes the DB connection service to persist the experimental metadata and the ARC schema to the database.

4.2.3 Metadata Versioning

The ARC Metadata Registry stores data as collections of JSON documents in two formats: ARC Object-level metadata in a proprietary format and metadata (JSON artifacts) in a common, serializable format. It is important to note that the data is read-only, and no direct updates or deletes are allowed through user APIs. Furthermore, the ARC Metadata Registry versions the data in the JSON artifacts in the order they are received, as shown in Figure 9.2. This allows for easy tracking of changes to the ARC data over time. The visibility of the ARC repository is reflected in the JSON artifacts.

ARC Metadata Registry provides a variety of query APIs that enable users to search and explore versioned metadata. By default, the query APIs return all public ARCs matching the metadata. For instance, searching for the contextual parameter-value combination *night exposure temperature = 22 degrees* returns all the relevant experiments from public ARCs grouped by DataHUB name, ARC ID, ARC Version, Investigation, Study, and Assay Name. However, when a valid user token received from AAI is attached to the request header, the ARC Metadata Registry performs token introspection. Also, it returns matching metadata of the private ARCs owned by the respective user. The registry application also provides APIs to search the ARC folder hierarchies.

4.2.4 Dataset Search

The ARC Metadata Registry also hosts dataset landing pages for individual public ARCs, facilitating dataset discovery through search platforms such as Google Dataset Search and DataMed. The landing pages are generated offline for each dataset within the Joint Declaration of Data Citation Principles framework. Each landing page contains basic information about the respective ARC, a direct link to the dataset (Gitlab repository), a dataset description, the publication submission date, the publication ID, and the corresponding DOI. To ensure public search platforms crawl the landing pages, we embed the ARC

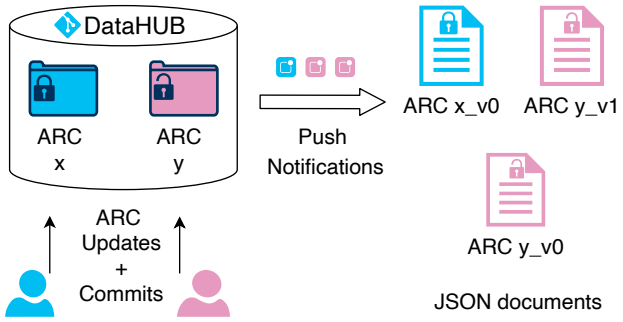

Figure 4.6: Metadata versioning, where each commit to ARC repositories is saved as a new ARC metadata version, reflecting visibility.

dataset-related information to each landing page using schema.org and other metadata standards.

4.2.5 Deployment

The individual services of ARC Metadata Registry, including the web interface, are developed in NodeJS with the ExpressJS framework. We utilize Express Gateway as our API gateway. Each service is containerized using Docker, and the application runs on Docker Swarm, which monitors services and ensures high availability. We use Keycloak for OAuth 2.0 services and AAI identity management.

4.3 Related Work

We compare ARC Registry with OmicsDI [33], an open-source platform developed to enhance the accessibility and discoverability of public omics datasets. OmicsDI serves as a centralized index and search engine for omics metadata, facilitating the location and integration of relevant datasets across disparate repositories.

Individual repositories such as PRIDE and MetaboLights cater to specific data types but often lack integration and interoperability. OmicsDI addresses this issue by aggregating metadata from these public repositories, allowing users to perform seamless searches across different domains and data types. OmicsDI functions by collecting and harmonizing metadata from participating data repositories. Similar to the ARC Metadata Registry, it does not store raw data; instead, it creates an index based on standardized metadata. Each dataset is described using a standard metadata model and is enhanced with natural language processing and ontology mapping to improve search relevance and ensure consistent terminology. In comparison, as the DataHUBs in the PLANTdataHUB model are not specific to any one method, meaning that the ARC Metadata Registry does not require additional efforts for metadata transformation and harmonization.

OmicsDI utilizes a distributed and scalable architecture. Data providers regularly submit metadata in standardized formats, typically JSON, through APIs

or various export tools. OmicsDI processes and indexes these submissions into a centralized system, allowing users to access the data via a web interface or programmatic access, such as RESTful APIs. The platform offers full-text search, faceted filtering, and semantic enrichment to help users discover datasets of interest, even if they are not familiar with specific repository terminologies.

Similar to the ARC Metadata Registry, OmicsDI monitors dataset reuse and citation, providing metrics that highlight the impact and visibility of specific datasets. This practice promotes proper data citation and helps researchers identify high-impact resources.

4.4 Summary

This chapter introduces ARC Metadata Registry, a microservice-based application that provides an integrated view of experimental metadata from individual DataHUBs within the PLANTdataHUB solution. The metadata is structured and standardized, making it easy for scientists to explore data relationships. This structured format ensures straightforward implementation and maintenance with conventional databases.

Chapter 5

LLMs for Metadata Search

As large language models (LLMs) gain popularity in bioscience research, they represent a promising alternative to traditional text search engines for exploring standardized research data [34]. Modern language models comprehend search keywords and the underlying semantics of user queries, enabling more accurate results. LLMs can potentially transform how we analyze integrated experimental metadata, providing a more powerful and insightful tool than conventional text search engines.

A comprehensive list of large language models (LLMs) trained on extensive biological corpora [35] exists for the bioscience domain. Domain-specific LLMs such as BioBERT [36] and SciBERT [37] have demonstrated effectiveness in tasks like question answering and text mining. These models are developed by initially pre-training a BERT [38] model on general-domain corpora (for instance, Wikipedia) and then further training it on specific-domain corpora. BioBERT is fine-tuned with biomedical texts from sources like PubMed and PMC, whereas SciBERT is fine-tuned with scientific literature from Semantic Scholar. Fine-tuning models on specific biological natural language processing tasks yields a significantly enhanced understanding of biological terminology.

There are three key aspects to consider when utilizing LLMs to develop a system for exploring multi-omics metadata in plant sciences. First, leveraging the RDM solution and its features is essential to building a comprehensive corpus of integrated omics metadata. Second, the model needs to be fine-tuned; this involves using a pre-trained model based on plant science literature and adapting it with the omics metadata corpus to enhance its ability to understand user queries in natural language. Lastly, since the metadata specifications follow a hierarchical structure, the model must grasp these layers to match user queries meaningfully. In this chapter, we present the following.

- We introduce *ArcBERT*, a Sentence-BERT-based language model [39] for integrated metadata exploration. The model is pre-trained using plant science literature from PubMed [40] and fine-tuned with integrated omics metadata.
- We propose using an indexing mechanism to index the vector embeddings from ArcBERT and leverage the FAISS library for fast nearest neighbor searches.
- We evaluate the effectiveness of using ArcBERT for metadata exploration compared to traditional text search engines.

5.1 Model Approach

Large Language Models (LLMs) are built on Transformer architectures and typically contain hundreds of millions to billions of trainable parameters. They

are trained on extensive textual corpora, which enables them to excel in understanding natural language. These models can also be fine-tuned for specific domains and disciplines. For instance, Bidirectional Encoder Representations from Transformers (BERT) [38] models are commonly used for tasks such as classification, named entity recognition (NER), and summarization. In contrast, Generative Pretrained Transformers (GPT) [41] models are primarily utilized for text generation and translation.

A crucial step in adapting any sentence-level semantic model involves transforming text input into dense vector representations that capture semantic meaning. Sentence-BERT [39], for example, converts each sentence into a fixed-length vector, allowing for semantic similarity measurement using cosine distance. Sentences that express similar ideas are mapped to points that are close together in this vector space, even if the surface wording differs. When fine-tuned with integrated multi-omics metadata, the model can understand user queries in natural language and provide semantically similar query results, unlike traditional text search engines that rely on keyword matching.

5.1.1 ArcBERT

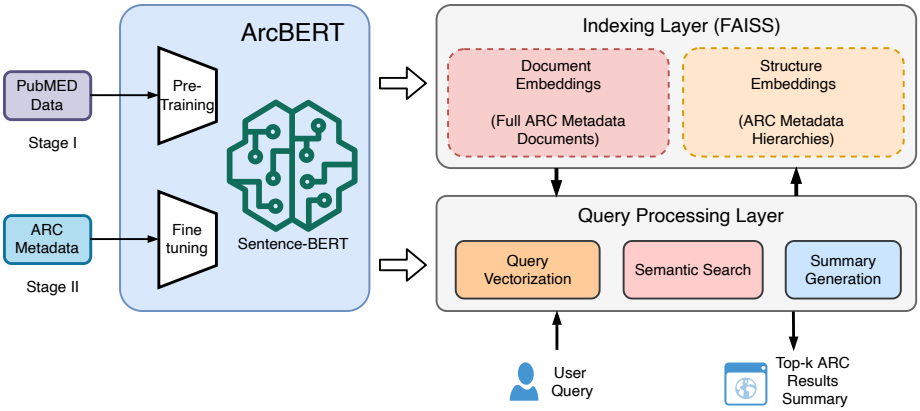

Figure 5.1: ArcBERT architecture showcasing the Sentence-BERT model, the indexing layer and the query processing layer.

We present ArcBERT, a domain-specific semantic search system based on the Sentence-BERT architecture as shown in Figure 5.1. This system is designed for Annotated Research Contexts (ARCs) that contain highly structured metadata from various omics studies. To efficiently retrieve relevant experimental metadata based on semantic similarity, we create a dense semantic vector index using FAISS, which enables rapid retrieval of the top-k most semantically similar ARC metadata for any given input query [42]. The model and indexing system are designed to maintain semantic similarity and preserve the hierarchical relationships within the metadata, ensuring meaningful query matching across the different metadata layers.

Pre-training: We developed a domain-specific training corpus using PubMed, a widely utilized public digital archive, to train our model with a meaningful

representation of the language found in plant science literature. We selected English articles published between January 2000 and December 2023 that contained the keywords “plant science” or “plant biology” in either the title or abstract. The extracted dataset was then cleaned and organized into a list of JSON objects, with each object representing one article. Before training the Sentence-BERT model, we concatenated relevant fields from each article—such as the title, abstract, methodology, and results—into a single string.

Fine-tuning: We refine the model using layers from the metadata, field-specific annotations, and nested relationships that accurately represent multi-omics data in practice. The ARC specification provides detailed information about the broader project context, key contributors, study designs, assay types, and workflows. By fine-tuning the model with multi-omics metadata structured according to the ARC specification, we can adapt its embeddings to reflect the underlying semantics and context. We leverage the snapshot of integrated multi-omics data directly from the *ARC Metadata Registry*, discussed in Chapter 4. The dataset is cleaned and flattened into plain text, as shown in Listing 5.1, by combining elements from its title, description, studies, assays, contributors, and any available ontological references into the field "search_text".

Document Embedding: After fine-tuning the ArcBERT model, every ARC document in the dataset is flattened into a respective "search_text" field, which is then transformed into a fixed-length embedding vector. However, performing similarity comparisons between a query vector and the model's embeddings becomes computationally expensive as the number of embeddings increases, rendering brute-force methods inefficient. We utilize FAISS, a library designed for fast nearest-neighbor searches over dense vectors, to accelerate query processing. FAISS offers various indexing structures; for this purpose, we use the $IndexFlatIP$ type, which calculates inner products between vectors. Since the embeddings are normalized to unit length during preprocessing and fine-tuning, these inner products correspond to cosine similarity scores—a widely used metric for semantic comparison at the sentence level. When a user query is submitted, it is processed through the ArcBERT model to generate an embedding in the same vector space. This embedding is then compared against the FAISS indices to identify the most relevant matches based on cosine similarity.

```
"title": "Systems-wide investigation of responses to moderate
    ↪ and acute high temperatures in the green alga
    ↪ Chlamydomonas reinhardtii.",
"description": "Algae cultures were grown mixotrophically
    ↪ (TAP). After 24h of 35/40 degree celsius, the cells
    ↪ were shifted back to room temperature for 48h. 'omics
    ↪ samples were taken.",
"people": ["Ningning Zhang (Donald Danforth Plant Science
    ↪ Center)"],
"publications": "Systems-wide analysis revealed shared and
    ↪ unique responses to moderate and acute high
    ↪ temperatures in the green alga Chlamydomonas
    ↪ reinhardtii.",
"studies": ["HeatstressExperiment"],
"assays": ["Proteomics", "Transcriptomics", "Metabolomics",
    ↪ "Growth"],
```

```
"search_text": "Systems-wide investigation of responses to
    ↪ moderate and acute high temperatures in the green alga
    ↪ Chlamydomonas reinhardtii. Algae cultures were grown
    ↪ mixotrophically (TAP). After 24h of 35/40 degree
    ↪ celsius, the cells were shifted back to room
    ↪ temperature for 48h. 'omics samples were taken.
    ↪ Ningning Zhang (Donald Danforth Plant Science Center)
    ↪ Systems-wide analysis revealed shared and unique
    ↪ responses to moderate and acute high temperatures in
    ↪ the green alga Chlamydomonas reinhardtii.
    ↪ HeatstressExperiment Proteomics Transcriptomics
    ↪ Metabolomics Growth"
```

Listing 5.1: A snippet of the training dataset for fine-tuning the model with the project information flattened into the string "search_text"

Structure Embedding: The flattened string for each ARC document provides a comprehensive summary of the document. However, relying on it as a single block during retrieval restricts the system's ability to highlight specific metadata elements corresponding to a user's query. In scientific data analysis, users often focus on specific layers of metadata, such as study or assay names, data stewards or institutions, and experimental parameter values relevant to a particular buffer condition.

To enhance search capabilities, the ARC metadata is broken down into smaller, more semantically meaningful segments, known as "chunks." Each chunk corresponds to a distinct metadata category, such as studies, assays, parameters, individuals, or publications, and is processed independently. This approach enables our system to match entire documents and pinpoint the exact section within the ARC hierarchy that corresponds to the user's query. For example, the natural language query *drought stress experiments on Chlamydomonas reinhardtii* may relate more closely to a specific study description rather than the title or abstract.

After the fine-tuning phase, each ARC document from the dataset is divided into chunks and embedded separately using the fine-tuned ArcBERT model. These embeddings are then stored in an FAISS index, enabling retrieval at the whole document and individual chunk levels. This dual indexing strategy enhances precision and interpretability, allowing users to see which layers of ARC metadata are most relevant to their query. Previous semantic search systems have employed a similar approach to improve retrieval granularity and enhance user trust in model outputs [43]. To support this method, each embedding is stored along with the information presented in Table 5.1, which is utilized to provide context-aware highlights in the retrieval interface and to aid in filtering and ranking based on the type of matched content. Indexing the documents and structures can be extended to include any new ARC metadata generated.

Hybrid Scoring: Matching keywords from user queries can be highly valuable in scientific data exploration, particularly when using structured or formal metadata. This is particularly important for input queries that are short, partially structured, or composed solely of technical terms rather than complete

Field	Type	Description
chunk_text	String	The extracted chunk of data from an ARC document (e.g., a study description or assay name).
field_type	String	Field name in the ARC structure. (e.g., study, assay, person, publication)
arc_id	String	A unique identifier that links the chunk to its parent ARC document.
chunk_index	Integer	Position of the chunk in the data field (used for ordering in case of multiple entries).
embedding	Vector	Dense representation computed by ArcBERT.

Table 5.1: Chunk data structure format for semantic indexing.

sentences in natural language. To address this, we leverage a hybrid scoring model that balances semantic and lexical similarity [44]. The final relevance score is derived by combining normalized cosine similarity based on FAISS with BM25 scores. Both scores are scaled between 0 and 1 before being merged using a weighting strategy. This scoring model is used for both complete document retrieval and partial chunk retrieval, as demonstrated in Equation 5.1.

$$S_{\text{final}} = \alpha \cdot S_{\text{D}} + (1 - \alpha) \cdot S_{\text{cmax}} \tag{5.1}$$

where S_{D} represents the weighted similarity score between the query vector and the ArcBERT document embeddings, while S_{cmax} denotes the maximum weighted score among all chunks. The weight parameter, α, can be adjusted to prioritize either the document or chunk scores. Additionally, the final score S_{final} can be boosted when the user specifies certain metadata structure filters, such as investigation titles, assays, or studies.

Post-Retrieval Summary: The system offers a flexible interface, allowing users to define natural language queries interactively. The query processing layer generates query embeddings, performs semantic matching, and ranks the top-5 results using the hybrid scoring model. The retrieved documents are summarized in natural language, providing users with a coherent overview that enhances their understanding of the content. In the final stage of the retrieval process, we integrate Google's Gemini API to create concise, structured summaries for each top-ranked result (top-k). These summaries are designed solely to aid in interpreting the results and are not intended to influence their ranking or interpretation.

Metadata Updates: As we refine the ArcBERT model using the latest snapshot of the ARC Metadata corpora from the Registry application, newly created ARC documents can be processed directly into document and chunk embeddings for indexing. The system will continue to incorporate the new ARCs for semantic matching. However, the update process must be streamlined to ensure the model is fine-tuned at regular intervals with the most up-to-date metadata corpora.

5.2 Experiments

We assess the effectiveness of ArcBERT for metadata exploration compared to the traditional text search engine, ElasticSearch. Our primary objective is to evaluate the relevance of the top 5 retrieved results and their corresponding rankings.

5.2.1 Setup

All experiments were conducted on a dedicated machine running Ubuntu 22.04.5 LTS. The server is equipped with an AMD EPYC 7662 64-core processor and 1 TB of RAM. Model training tasks were accelerated using an NVIDIA A100 PCIe GPU with 80 GB of memory. The development environment is based on Python 3.10, utilizing PyTorch 2.0 as the primary deep learning framework.

For sentence-level semantic retrieval, we implemented the sentence-transformers library (v2.2.2), which is built on Hugging Face Transformers. We utilized FAISS (v1.7.3) for approximate nearest neighbor search in high-dimensional spaces, while Elasticsearch (v8.9.1) served as the baseline keyword-based retrieval engine.

ArcBERT, based on the all-mpnet-base-v2 architecture, is trained on structured ARC metadata. It was optimized using the Multiple Negatives Ranking Loss (MNRL) function, which promotes semantic coherence in the embedding space. Additionally, the Gemini summarization module was accessed via the official Google Generative AI client, utilizing the Gemini-1.5-flash model.

5.2.2 Benchmark Query-set

We have curated a set of 122 test queries to assess the effectiveness of ArcBERT compared to ElasticSearch. These queries are logically organized into nine distinct categories, each representing different user query requirements. The queries range from simple keyword searches (abbreviated as KW) to more complex inquiries that incorporate hierarchical layers of metadata, such as parameters (KWPAR), studies (KWSUD), assays (KWA), or natural language queries (SWK and SEM). This query set is designed to simulate real-world scenarios where users might approach data exploration from varied perspectives, whether by seeking specific experimental techniques or referencing known contributors or publications. An example query includes

- Semantic: *Which ARC investigates how disease resistance genes are constantly active in crops like tomatoes and potatoes?*
- With keywords: *Which ARC study about* ***global expression patterns*** *of* ***R-genes*** *in tomato and potato?*

5.2.3 Query Execution

Each query is executed separately on both retrieval systems, with the top five results selected based on matching scores and averaged for each query category. Notably, including structured fields such as people, parameters, assays, and studies enables hierarchical filtering, allowing users to refine their search results

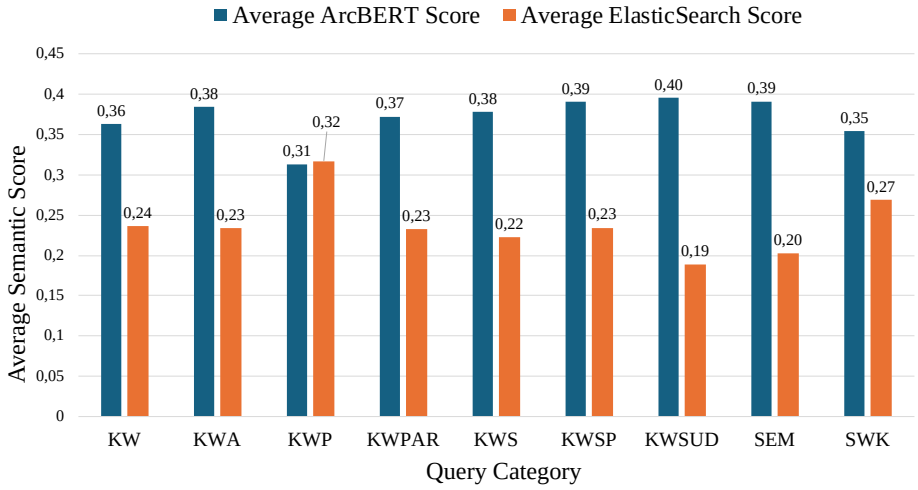

Figure 5.2: Average top-5 similarity scores across query categories for ArcBERT and ElasticSearch.

further. These filters mimic how users may narrow their queries when exploring complex metadata.

We compare each result set averaged for respective query categories with multiple retrieval metrics, including mean rank, mean reciprocal rank (MRR), and similarity scores between the query and the results. These metrics assess the precision of ranking and the semantic alignment of the results. MRR highlights the importance of retrieving relevant documents early, while mean rank provides a broader view of how results are distributed among the top ranks.

A consistent normalization strategy for the scoring values is essential to ensure meaningful comparisons between two retrieval systems. ElasticSearch utilizes the BM25 ranking algorithm, which produces unbounded output scores that can vary significantly based on the frequency and distribution of query terms within the indexed corpus. Without normalization, these variations could lead to misleading quantitative assessments, particularly when comparing performance across shared metrics, such as Mean Reciprocal Rank (MRR), or evaluating the quality of ranked outputs. This approach is consistent with evaluation methods used in large-scale information retrieval (IR) tasks such as TREC and CLEF, where different systems are benchmarked using shared ranking-based metrics instead of relying on their native scoring outputs. To achieve this, we apply a global min-max normalization strategy for the ElasticSearch scores, as described below:

$$\text{Normalized Score}_{\text{ES}} = \frac{S - S_{\min}}{S_{\max} - S_{\min}} \tag{5.2}$$

5.2.4 Results

Average Retrieval scores: Figure 5.2 demonstrates that ArcBERT consistently achieves higher average similarity scores for the top-5 results across a

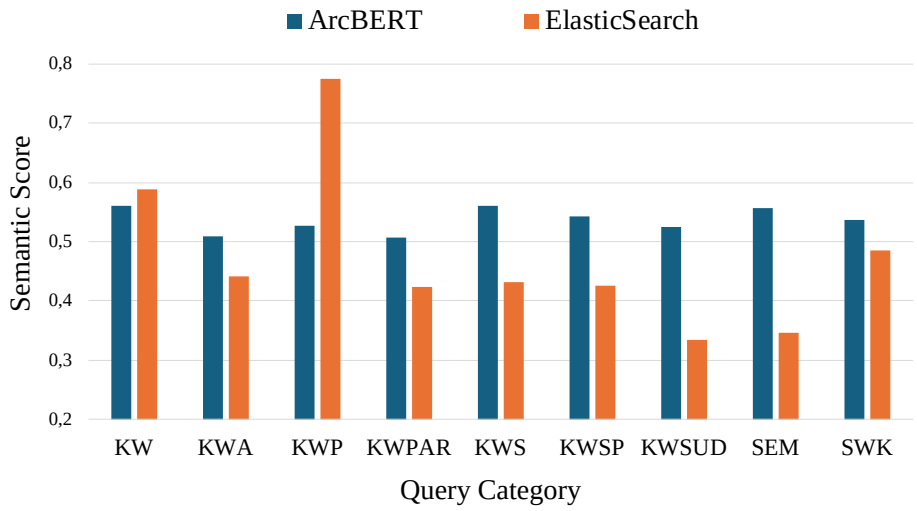

Figure 5.3: Mean similarity score of the top-1 result per query category.

wide range of query categories. These results indicate that language models like ArcBERT offer significant advantages in understanding semantically rich and natural language queries, allowing them to deliver result sets more closely aligned with user intentions. While Elasticsearch efficiently retrieves the best matching top result for queries containing specific keywords, it becomes less effective when considering the average scores for the top 5 results. This finding is expected, as token-based retrieval systems typically perform better when user queries are closely aligned with indexed metadata fields. Interestingly, ArcBERT's hybrid approach—which utilizes cosine similarity and lexical BM25 scoring—enhances consistency across various query categories for the top-5 results.

Mean top rank scores: Figure 5.3 illustrates the average score of the top-ranked retrieved results for ArcBERT and ElasticSearch across different query categories. ArcBERT consistently achieves significantly higher first-rank scores in categories where the queries do not show substantial lexical overlap with ARC metadata, thanks to its semantic encoding capabilities. In contrast, ElasticSearch struggles in these categories with first-rank scores. However, ElasticSearch performs comparably or even better in keyword-heavy categories such as KW, KWP, and KWA, where it benefits from exact token matches.

Mean reciprocal ranks: The Mean Reciprocal Rank (MRR) is a metric that measures how well a system ranks the highest expected result within a list of retrieved documents. Figure 5.4 compares the MRR across different query categories. Unsurprisingly, ElasticSearch outperforms ArcBERT in all categories, demonstrating its effectiveness in ranking the best-matching documents at the top. In contrast, the ArcBERT scoring model places a greater emphasis on consistent semantic matching.

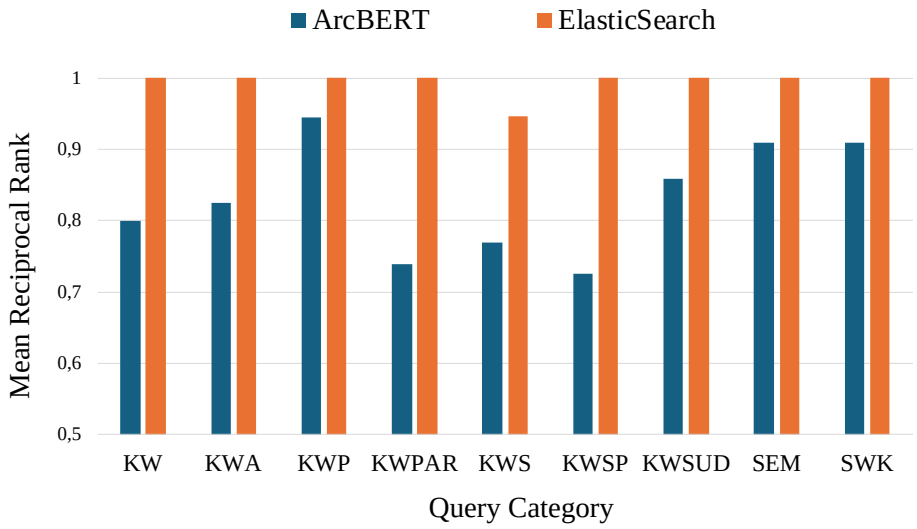

Figure 5.4: MRR comparison across the query categories.

5.3 Summary

In this chapter, we introduce ArcBERT, a semantic search engine based on Sentence-BERT, designed to help users explore omics metadata through natural language queries. ArcBERT is pre-trained and fine-tuned to comprehend plant science terminology and the various layers of integrated omics metadata. We evaluated the effectiveness of ArcBERT by comparing it to a traditional text search engine using a query dataset that simulates real-world user query patterns. As anticipated, the results of our experiments show that ArcBERT can understand natural language-style queries and the structure of the metadata, outperforming text search engines in retrieving consistent results that are semantically closer to user queries. However, we also observed that text search engines are still effective at retrieving the best matching results through keyword matching.

In the subsequent chapters, we focus on managing heterogeneous measurement or raw data from bioscience datasets and evaluating multi-model architectures and indexing ideas for the interactive exploration of complex datasets.

Chapter 6

Data Exploration & Indexes

The semi-structured or unstructured measurement data and the contextual information generated from the instruments used during experiments can include details such as instrument parameters, configuration files, workflows, annotated images, and other relevant information. This raw data is often heterogeneous and diverse, varying based on the domain, the type of experiment conducted, the measurements taken, the equipment used, and the procedures followed. Creating data management systems that allow analysts to interactively and efficiently explore research data at acceptable latencies presents significant challenges due to the sparsity and volume of the measurement data formats [4].

In PLANTdataHUB, the hierarchical file system layers within the ARC specification present challenges in developing integrated data analysis platforms that allow users to explore file contents across these directory hierarchies. Given the expected size of the data, innovative storage solutions, efficient information management, and rapid indexing mechanisms are essential. Furthermore, since most experiments are conducted collaboratively, the distributed nature of the stored data presents further challenges, not only for searching but also for integrating systems with existing knowledge bases and public repositories.

In this chapter, we individually explore design ideas, data modeling, data partitioning, and indexing options to propose the architecture of the data management system that can address *Research Task 2* in Section 4 of Chapter 4. We first evaluate the datatypes within traditional relational databases for semi-structured raw data, and extend the architecture to a multi-model approach, utilizing a relational database for structured metadata and NoSQL stores for semi-structured raw data. We begin with a basic interactive evaluation framework that uses the in-memory data structures and indexing options outside the data stores. In particular, the evaluation framework is an abstraction from diverse data models and query interfaces while catering to the unique plant science querying requirements of applying modification patterns on a specific dataset. Thus, it enables the progressive exploration of research data while interactively evaluating filter conditions on large volumes of contextual information, all in memory. We design our system with typical research data management applications in mind, where *writes* and *reads* are frequent, but *updates* and *deletes* are negligible.

In typical data analysis, a data analyst conducts exploratory analysis on the experimental metadata and contextual information to generate resultant datasets that satisfy specific modification (or refinement) patterns. These patterns are Boolean combinations of attribute-value pairs that either represent a snapshot of the environment in which a specific experiment was conducted or identify and quantify the measurements recorded by the instruments. Consider the data snippet in Figure 6.1, which illustrates the acclimation reactions of plant samples to varying temperatures and light conditions. In the study, the abundance

of proteins from each sample is found by measuring the peptide instances under various experimental conditions. Example parameters include *charge*, *retention time*, and *mass-to-charge ratio*, among others. For simplicity, the contextual parameters and values of each instance are represented as attributes $P_1, P_2, ..., P_n$. Different Boolean combinations of these contextual parameters form the *Modification Patterns*, say, $((P_m \cup P_n) \cap P_k)$, $(P_m > 1.8 \ \cup P_k = high)$ etc. To find the list of *Samples* and their corresponding *Instances* that contain a given modification pattern requires a relational join and the containment predicate condition evaluated on the JSON column. However, the modification patterns are not known in advance and are constructed during the data analysis. Moreover, the modification patterns are varied and applied to the same dataset to extract multiple result sets, which are then fed to subsequent computation workflows. This approach makes one-shot querying time-consuming and impractical for large datasets.

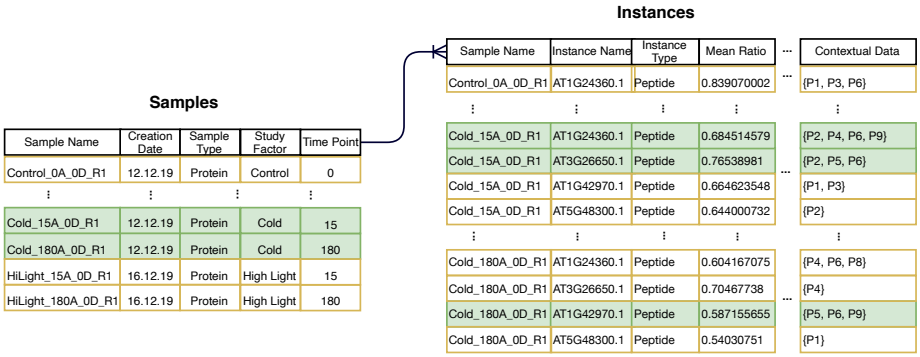

Figure 6.1: Relational+JSON to capture structured research data and schema-less contextual information. Highlighted tuples for the refinement pattern $((P_2 \cup P_5) \cap P_6)$.

6.1 Related work

In the relational space, while SQL can easily handle the structured portion of the data and work seamlessly in standalone relational database systems, the schema-less semi-structured data does not comply with the n-ary storage model. A Decomposed Storage Model (DSM) [45] is one of the early solutions for handling sparse semi-structured attribute-value data. The introduction of JSON data types in RDBMS platforms enabled multi-modeling by storing JSON data natively without flattening it into a relational format [46]. However, the schema-less JSON data can only be considered as text documents for indexing. Extensions of DSM, such as Dynamic Tables [47], were used for biomedical data to avoid many relational joins when querying, unlike PIVOT/UNPIVOT operators. As the Dynamic Tables are part of the relational database, one-shot querying and scalability will still be problematic for interactive exploration with large volumes of measurement data.

In the NoSQL space, multi-model databases that integrate different data mod-

els into a single database can help maintain structured metadata and semi-structured measurement data within the same database. However, these single-model NoSQL stores are often extended to support multi-model data without relational capabilities. This means the data needs to be denormalized and duplicated to compensate for the lack of relational joins.

6.2 Evaluation Framework

In our initial approach to interactive exploration, we rely on loading the superset of data into main memory and allowing the user to drill down into the data by evaluating predicate expressions. For the example data snippet shown in Figure 6.1, the data analysts can perform interactive exploration and progressive filtering of the relational data while evaluating containment, point, and range predicate expressions on the augmented attribute values. The evaluation framework enables progressive data filtering based on the relations of interest while evaluating predicate expressions against the respective JSON data. It allows the analyst to download a snapshot of the filtered data at any stage of exploration.

Within an active user session, each analysis task in our evaluation framework begins with defining the search space. It offers a set of operations with certain semantic constraints to aid data analysts with each step of the interactive analysis. A typical interactive analysis task includes:

1. Defining the search space.
2. Initializing the search space by loading the *signatures* of the data tuples into the main memory.
3. Filtering the data tuple signatures progressively. Allow for interactive predicate construction and evaluation of predicates.
4. Materialization of the search space snapshot at any interactive exploration stage.

Search Space Definition:

The first step in the analysis task is defining the search space, an abstract two-dimensional data structure within the main memory. Each relation in the hierarchy can be represented as one of the dimensions or axes of this space. An initial selection predicate can be associated with the definition of each dimension, and this predicate is evaluated when loading the data tuple signatures into memory. This process helps initialize the search space data structure with the specific data tuples we are interested in.

A *dimension* is formally defined as $D_{dim} = \{R, S, P_i\}$, where: - R is the mandatory relation from the hierarchy, - S is the signature definition for each data tuple of the relation, and - P_i is the initial predicate expression.

Both the signature definition and the initial predicate expressions are optional. By default, the *id* of each data tuple in the relation is regarded as its signature.

Initialization:

The Search Space data structure is formally defined as $\Omega = \{D_x, D_y, P_x, P_y\}$, where D_x and D_y represent the definitions of the dimensions. The framework automatically aligns these dimensions to adhere to a strict parent-child relationship derived from the hierarchy (which does not necessarily have to be an immediate parent-child relationship). For instance, it is not permissible for both search space dimensions to have the same *Samples* relation for exploration.

The symbols P_x and P_y are used to track sets of attributes in the form of Boolean expressions, which are evaluated based on their corresponding attribute-value JSON data. Initially, these are set to *null* and are subsequently updated after each interactive operation. During the initialization phase, the framework combines the two relations from the hierarchy and projects only the *id*s of the data tuples that meet the specified initial predicate expressions in the dimension definitions.

Once the search space is initialized, the analyst can perform various exploration operations and view the updated search space after each action. The set of operations that can be performed includes: *update*, *mockupdate*, *merge*, *save*, *load*, and *delete*.

Exploration & Progressive Filtering:

The *update* operation takes a dimension name and a set of contextual attributes as its parameters. It modifies the search space by retaining the signatures of data tuples from the specified dimension that contain any of the provided attributes. The goal is to utilize membership queries on indices to filter out the signatures that do not meet the Boolean predicate condition.

Consider the initialized search space $S_{sp} = \{D_x, D_y, P_x = \emptyset, P_y = \emptyset\}$. When the operation $update(S_{sp}, D_y, [a_1, a_2, ..., a_n])$ is executed, it evaluates the Boolean predicate expression $a_1 \cup a_2 \cup ... \cup a_n$ on the dimension D_y and prunes the signatures from the search space accordingly. As a result, the updated search space becomes $S_{sp} = \{D_x, D_y, P_x = \emptyset, P_y = (a_1 \cup a_2 \cup ... \cup a_n)\}$.

Subsequent update operations on the search space narrow the list of signatures to those that satisfy the respective Boolean conditions. In summary, the Boolean predicate expressions of subsequent *update* operations on the same search space are equivalent to the logical *AND* of individual sets of *OR*-ed attributes.

The *merge* operation performs a logical *OR* on all the signatures from two search spaces and updates the predicate expressions for each dimension accordingly. Conversely, the *mockupdate* operation functions like the *update* operation but does not modify the search space. Instead, it returns a summary of the operation's result, allowing the analyst to anticipate the outcome of the *update* operation without making any changes to the search space.

Materialization:

A key feature of the framework is its flexibility in materializing data tuples and contextual information offline. When prompted by the user, the complete set of data tuples is retrieved from the database using the signatures from the search

space and downloaded to disk in the selected data format, such as CSV, Tab-delimited, or JSON. The data analyst can also materialize and download the search space after each exploration operation, such as *update* or *merge*.

6.3 Containment Queries

We begin with simple containment queries to evaluate modification patterns and use Bloom filters to index the data that complements any data model, and the phases of interactive exploration. For the data already in a relational format, we leverage the existing indexing techniques, such as B+Trees, and the relational joins offered by the underlying databases for operations, including initializing the search space. However, to evaluate containment queries on JSON data, we start with Bloom filters, which are highly space-efficient in representing the sets of contextual attributes and support basic Boolean expressions with OR and AND.

A Bloom filter [48] is an array of m bits for representing a set $S = \{x_1, x_2, x_3, ..., x_n\}$. To begin with, the bit array is initialized to zero. The idea is to use a set of k uniform hash functions, $h_i(x), 1 \leq i \leq k$ to map elements $x \in S$ to random numbers uniform in the range $1..m$. The basic operations on Bloom filters include adding an element to the set and querying whether an element is a member of the set. To add an element x to the set, it is fed to the k hash functions, and the bits of the returned positions are set to 1. To test if an element is a set member, the element is fed to the k hash functions to receive k array positions. The element is not a set member if any bits at those k positions are 0. This way, the Bloom filter guarantees that there exist *no false negatives*, but it can return *false positives* due to possible hash collisions. This is because the respective bits were previously set to 1 by other elements. Assuming that the hash functions are perfectly random, it is possible to estimate the probability of false positives. For a Bloom filter size of m bits, with k uniform hash functions, after n insertions (each using k hash functions), the probability of a specific bit that is not set is, $\left[1 - \frac{1}{m}\right]^{kn}$. Besides individual element operations, such as element insertion and membership queries, Bloom filters can also perform set union and intersection [49, 50].

6.3.1 Data Partitioning

We use partitioning to divide the data tuples in the relation into non-overlapping subsets using a unique and contiguously increasing primary key or tuple identifier. The Bloom filters are created and maintained per segment/partition and attribute as shown in Figure 6.2. They are serialized and persisted in secondary storage. Here, we leverage Bloom filters to pre-compute the query results for each attribute and enable analysts to compose and evaluate modification patterns interactively during analysis. Consider a database relation R with primary key p and a set of attributes $a_1, a_2, ..., a_k$, where the primary key p is a contiguously increasing integer number.

Let $t_1, t_2, ..., t_n$ be the tuples in the relation R with respective keys $p_1, p_2, ..., p_n$. A segment S_i is a subset of tuples with keys ranging from $p_i, p_{i+1}, ..., p_{i+x}$ in contiguously increasing order, where x is the segment size. The first segment

S_1 contains the tuples with keys ranging from $p_1, p_2, ..., p_x$. The segments are ordered disjoint sets of keys i.e. $S_1 \cap S_2 \cap ... \cap S_n = \emptyset$ and $S_1 \cup S_2 \cup ... \cup S_n = R$.

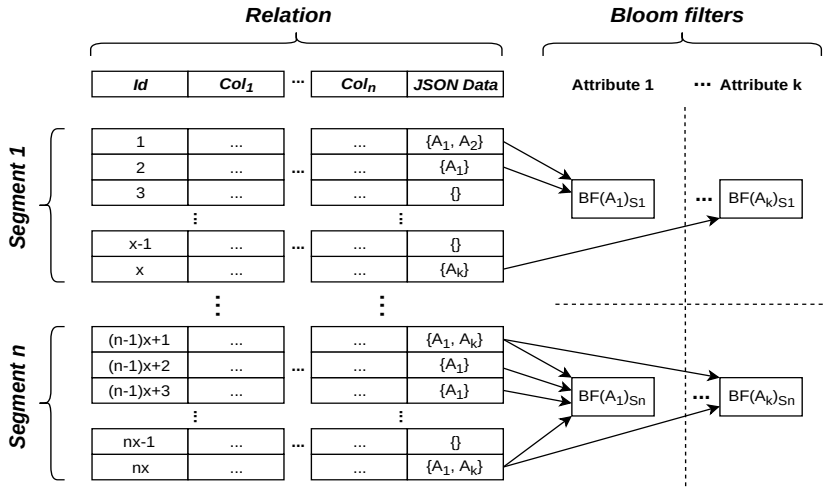

Figure 6.2: The initial partitioning approach.

The segment size x can be optimized. However, it is fixed to a constant size before building the Bloom filter indices since any changes to the segment size require rebuilding the index. We set the segment size multiple times larger than the total number of attributes to be indexed. This reduces the number of Bloom filters, resulting in faster retrievals. However, setting the segment size to a large number would result in maintaining large-sized Bloom filters. For example, for a segment size of 100 000, 4 hash functions, and a fixed false positive rate of 0.01, the approximate size of the bit vector required is 128 KB. To achieve a fixed bit vector size, two other parameters, which can be tuned but have to be set before building the indices, are the number of hash functions and the false positive rate. The partitioning approach is vertically scalable and, as such, does not limit the number of attributes to be indexed or the number of data tuples in the relation.

6.3.2 Experiments

We compare our Bloom filter indexing with other methods, such as Inverted Indices and Vertical Partitioning, to evaluate our basic framework and investigate the following aspects:

1. How fast are interactive response times when Bloom filter indices are used to evaluate Boolean predicate expressions on attribute-value format data?
2. What is the average time to perform an end-to-end analysis task?
3. What are the storage space requirements across the three different methods?
4. What is the overall impact of false positives?

We address the above points using three databases of varying sizes: $Small \approx 50GB$, $Moderate \approx 104GB$, and $Large \approx 520GB$. We use PostgreSQL version 11.7 as the database for all our experiments performed on a Ubuntu 18.04.3 LTS 64-bit server. The server hardware configuration consists of a 2x6-core Intel Xeon Bronze 3104 CPU at 1.70 GHz, 128 GB RAM, and a 3 TB HDD. The application layer and the front end are developed in Spring Boot Framework 2.1.6 and deployed on the same server.

Test Data

To simulate a real-world environment, we use uniformly distributed synthetic research data for our evaluation. We believe that dense synthetic data is the best-case scenario for evaluating our framework, as we focus on measuring latencies and disk space requirements. The test databases for our experiments are created with a simplified hierarchical data model where we only consider two levels of the standardized bio-science research data hierarchy (bio-samples and measured bio-instances). We simulate the research data by considering eight different biological instances, i.e., *protein, lipid, peptide, gene, etc.*. We generate 500 Bio-Instance data tuples for each Bio-Sample of any type. Similarly, each Bio-Instance is enriched with 300 randomly chosen attributes from the test vocabulary, containing 2550 unique attributes. For example, the *Small* database contains 8 750 bio-samples with 4.3 million related Bio-Instances, each containing 300 contextual attribute-value pairs in JSON format, totaling 1.3 billion attributes. Similarly, the *Large* database contains 95 000 Bio-Samples, 47.5 million related Bio-Instances, and 14.2 billion attribute-value pairs as contextual information.

Apart from the usual B+Tree indices on the relational format data tuples of both Bio-Samples and Bio-Instances, we create Bloom filters and Inverted indices per each attribute in the JSON column of the Bio-Instances relation. We employ a segmentation approach, indexing contextual information per segment and attribute. We use the segment size of $2^{15} = 32\,768$ for the experiments. The Bloom filters per segment and attribute are created with a fixed number of hash functions and a false positive rate of 1%. Similarly, the inverted indices are created per segment and attribute, serialized, and persisted to disk. For Vertical Partitioning, we create separate relations for each attribute with the *id*s of Bio-Instance tuples containing that attribute as part of its JSON data, i.e., a total of 2550 single-column tables, each for one attribute.

Results

We simulated *Modification Patterns* by evaluating our experimental analysis tasks with eight interactive *update* operations per task. Each operation is evaluated on a set of 5 contextual attributes. The attributes for the experiments are selected randomly from the test vocabulary, and all analysis tasks are repeated for each bio-instance type (*peptide*, *metabolite*, etc.). The results are then averaged for each database. The average number of signatures per each bio-instance type loaded into the memory, i.e., the *Search Space* are 0.5 million, 1.1 million, and 6 million for databases *Small*, *Moderate* and *Large* respectively. As the data is uniformly distributed, each *update* operation proportionately reduces the grid size.

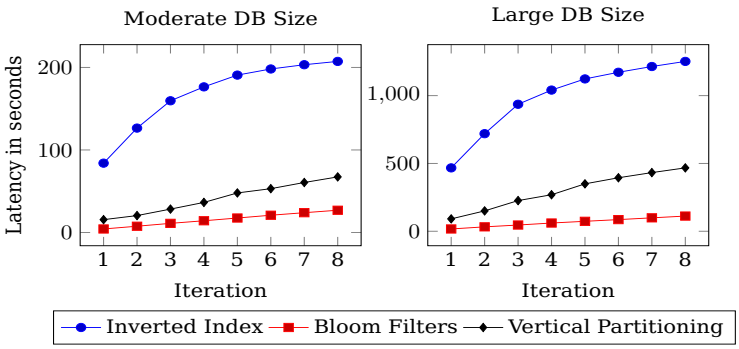

Figure 6.3: Latencies measured per iteration

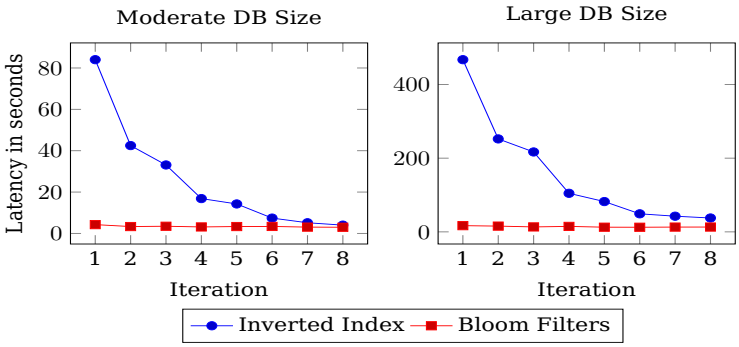

Figure 6.4: Cumulative latencies

Figure 6.3 shows the latencies measured at the application layer at the end of each *update* operation with *Moderate* and *Large* databases. We also notice the same pattern with *Small* database. It is clear that when using inverted indices, the early set of *update* operations take a long time due to list operations such as sorting, merging, etc. However, the latencies improve as the number of signatures is reduced after each subsequent iteration. In contrast, regardless of the search space size, Bloom filter indices are significantly faster, with nearly the same latency per iteration. With *Large* database and initial *Search Space* size of 6 million signatures, each interactive operation with inverted indices takes several minutes compared to the consistent latency of just ≈14 seconds per operation with Bloom filter indices.

Figure 6.4 shows the average cumulative latencies of the interactive operations. We use the vertically partitioned *id*s as a baseline to compare the total time taken at each iteration of the analysis task. When using vertical partitioning, we let the database engine perform the unions and intersections of the *id*s. Here, too, Bloom filter indices prove much faster than the baseline vertical partitioning and the inverted indexes. It only takes less than 2 minutes when using Bloom filters to perform all eight interactive operations on the significant search space of 6 million signatures, compared to the vertical partitioning and the inverted indices, which take nearly 8 minutes and more than 20 minutes, respectively.

We also measured the approximate size of indices in the uncompressed form

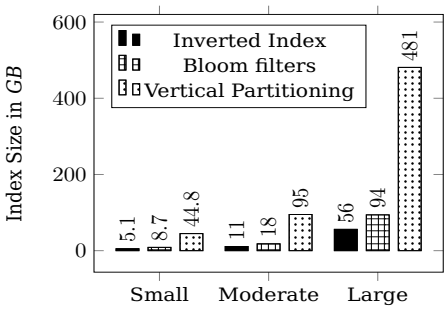

Figure 6.5: Index sizes measured

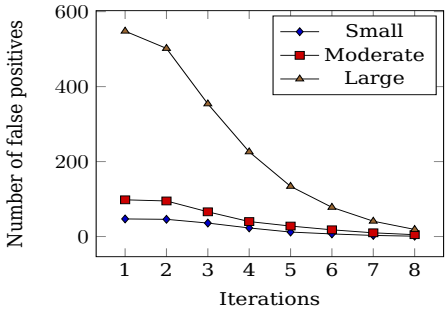

Figure 6.6: Effect of False positives

to estimate the storage space requirements. Figure 6.5 shows the index sizes in GB across different databases measured with the segment size 32 768 to create Bloom filters and inverted indices. Since we used 2550 unique attributes, Vertical Partitioning of *id*s generated the same number of relations. We observe that the inverted indices used less storage space than Bloom filters, i.e., $\approx$ 10% of the data size vs. $\approx$ 17%. However, as expected, Vertical Partitioning required $\approx$ 91%, which is almost the size of the entire data in the respective database. When the search space is left with relatively few filtered signatures, the false positives are expected to be near zero. We noticed a similar trend in our experiments, and the number of false positives decreased relatively with each *update* operation, almost reducing the search space by 50%. Figure 6.6 shows the decreasing trend of false positives per iteration with different database sizes.

Our experiments demonstrated that Bloom filter indices are often faster than traditional inverted indexes. In contrast, the Bloom filter indices' storage space requirements are significantly lower than those of the decomposed storage for attribute-valued data modeled in JSON format. While Bloom filters are efficient for answering containment queries, their use for range queries can be challenging. To address this, we extend the existing framework by utilizing lightweight bitmaps and tree structures that are enhanced with bitmaps. This modification allows us to evaluate both containment and point-and-range queries. As a result, we can more effectively evaluate complex predicates, including arbitrary range queries on attribute-value data.

6.4 Point & Range Queries

We continue the partitioning approach, as it also ensures that an index on a particular attribute is created in parts (per segment), thus making it faster to access, even when maintained in a distributed environment. Figure 6.7 shows how the indexes per segment are created, each per each measurement/contextual parameter, and is built for index-only scans. We now explore using tree structures for point and range queries, replacing the Bloom filters with bitmap indexes. The idea is to encode each segment's set of satisfying tuple IDs on the relational side into a compressed data structure suitable for the given query type. For example, a simple containment query requires just a bitmap index of the segment size with encoded 1 and 0 bits, where a 1 bit at a particular

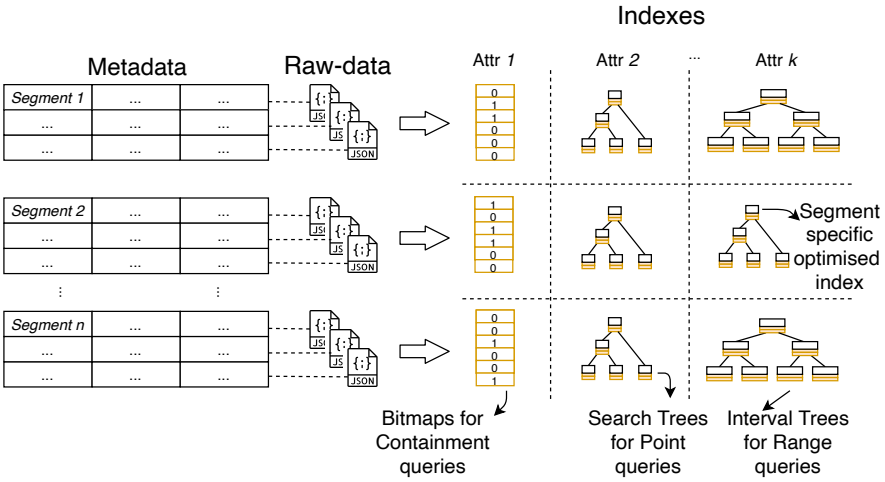

Figure 6.7: Partitioning of data and flexible indexes.

position in the bitmap indicates that the respective tuple ID in that segment contains the requested attribute. Such a set of bitmap indexes for all attributes and segments is good enough to evaluate any boolean combination of containment queries, i.e., refinement patterns on any selected attributes during analysis, without referencing the semi-structured measurement/contextual data from the NoSQL data store. Similarly, a tree structure augmented with the tuple IDs created on the attribute's values in each segment is ideal for point queries. We then utilize interval trees [51] for range queries, which are tree structures similar to those used for point queries, except that the nodes store value ranges instead of a single value. Here, the nodes in the tree structure are augmented with encoded bitmaps that contain 1s at certain positions where the respective tuple IDs of the segment fall within the specified range. As expected, the interval ranges introduce false positives in the query result. The number of false positives depends on the interval between the query and the interval range. It can also optimize the interval sizes of range index structures at the segment level.

Usually, the domain experts decide beforehand the index type for an individual known measurement/contextual attribute, depending on the expected querying type. Although the point indexes also handle range queries, approximate results are allowed during analysis, primarily when querying for value ranges of specific attributes. Between the index size, search time, and accuracy, a trade-off exists, where range index structures can be optimized, and interval trees can help reduce interactive latencies compared to using point indexes for answering range queries.

6.4.1 Interval Trees

We use the simple case of interval trees where the intervals do not overlap. An interval $[x_1, x_2]$ is an ordered pair of real numbers that represents the set $\{x \in \mathbb{R} : x_1 \leq x \leq x_2\}$. For simplicity, we consider only closed intervals where both endpoints are included in the set. Two intervals i and i' are considered as overlapping if $i \cap i' \neq \emptyset$ i.e. $i_1 \leq i'_2$ and $i'_1 \leq i_2$. Basic tree data structures, such

as binary search trees and red-black trees, can maintain a dynamic set of elements, with each element x containing an interval. These interval tree structures can support insert, delete, and search operations and are useful for answering range queries, particularly closed-ended range queries such as $-5 \leq k \leq 5$. When a binary search tree is extended for non-overlapping interval ranges, each node n in the tree holds the element x, which is assumed to be part of the interval $[x_1, x_2]$ with x_1 being the low endpoint and x_2 being the high endpoint. Given a query interval of $q.int$, the search operation to find the intersecting interval node is expected to take $O(\log n)$. For a given attribute k, consider a segment S_i with tuple id-value pairs $(p_i, v_i), (p_{i+1}, v_{i+1}), \ldots, (p_{i+x}, v_{i+x})$ where x is the segment size. After sorting by value, we use binning, which splits the data into disjoint intervals of predefined value ranges. Additionally, we ensure that repeated values are enclosed within the same interval, thereby avoiding any interval overlaps.

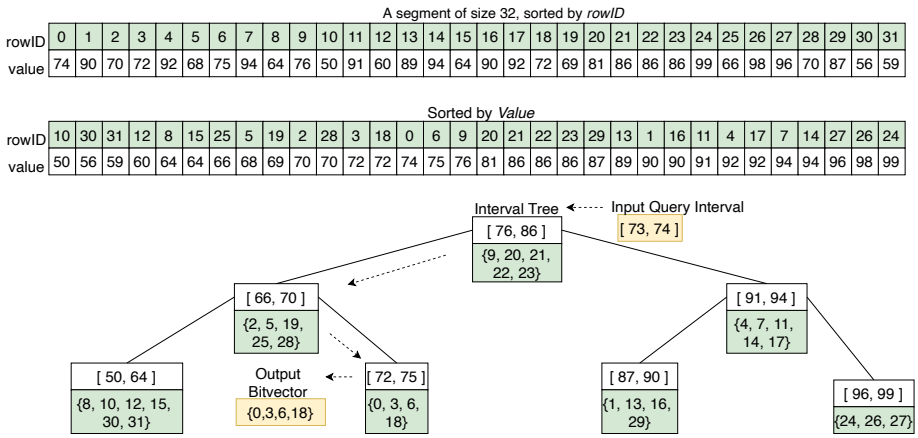

Figure 6.8: Interval tree index for range queries

For each interval, individual bitmaps of size x with the set of corresponding tuple IDs are generated before being augmented with the tree nodes, as shown in Figure 6.8. The tree data structure is serialized to disk for the given attribute and segment S_i. Two conditions must be checked to find the resulting interval $r.int$ intersecting with the query interval $q.int$. One, the start and/or endpoint of $r.int$ is in $q.int$, or $r.int$ completely encloses $q.int$. The search algorithm finds all the intervals intersecting the given query interval $q.int$. It returns one bitmap per entire segment, i.e., it performs a logical OR of the bitmaps from each intersecting interval. In scenarios where the analyst issues a complex query with multiple attributes, say, $(0.0045 \leq protein\,abundance \leq 0.0046)$ AND $(98 \leq mass/charge\,ratio \leq 100)$, the query gets parsed into individual queries per attribute. The respective indices are then used to evaluate either the point or the range query to get the bitmaps per segment for each attribute. Lastly, the bit-wise OR or AND operation is performed on the individual bitmaps for each attribute to get the final bitmap per segment resulting from the complex query.

The main disadvantage of interval trees is the fixed interval sizes. The size of the interval and the incoming query type affect the overall accuracy of the index

data structure, as the number of false positives in the query result depends on the difference in the interval spans between the query interval and the interval range.

The segment size x varies for each relation in the hierarchy and can be individually optimized. However, it is fixed to a constant size before building the indexes for individual relations in the hierarchy, since any changes to the segment size require rebuilding all existing indexes. We recommend having the segment size multiple times larger than the total number of attributes to be indexed. This reduces the number of individual indexes that must be maintained, and the retrievals would be faster. However, setting the segment size to a large number would result in working with large index structures.

6.4.2 Experiments

We use uniformly distributed synthetic research data for this evaluation, and the test database is of size ≈40GB. The test database is created with a simplified hierarchical data model. We only consider two levels of the standardized bio-science research data hierarchy (bio-samples and measured bio-instances). We simulate the research data by taking eight different types of bio-instances, i.e., *lipid, peptide, gene, etc.*. We generate 500 bio-instance data tuples for each bio-sample of any type. Similarly, each bio-instance is enriched with 300 randomly chosen attributes from the test vocabulary, which contains 2,550 unique attributes. The database includes 7,000 bio-samples with 3.5 million associated bio-instances, each containing an array of 300 contextual attribute-value pairs in JSON format, bringing the total count of attributes in the database to approximately 1 billion.

We use PostgreSQL version 11.7 for *meta-data* and MongoDB version 4.4.0 for *raw data* in all our experiments on a server running Ubuntu 18.04.5 LTS 64-bit. The server hardware configuration is as Intel Core i7-8700 CPU @ 3.20GHz ×12, 16GB RAM, and 500GB SSD. The application layer and the front end are developed in Spring Boot Framework 2.1.6 and deployed on the same server. Figure 6.9 shows the basic system architecture.

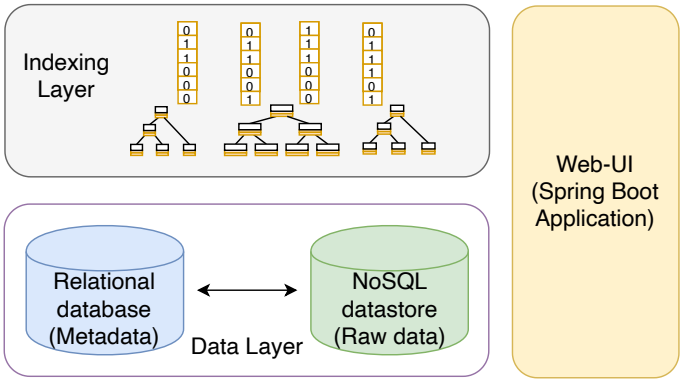

Figure 6.9: System architecture

We simulate refinement patterns by evaluating our analysis tasks with five in-

teractive *update* operations per task. Each operation is evaluated on a set of 5 contextual attributes. The attributes for the experiments are selected randomly from the test vocabulary, and all analysis tasks are repeated for each bio-instance type (*peptide*, *gene*, etc.). The results are then averaged. The average number of signatures per bio-instance type loaded into the memory is 0.4 million, and it took ≈3.1 seconds to initialize the search space. As the data is uniformly distributed, each *update* operation proportionately reduces the search space size. Our experiments use the segmentation approach with a fixed segment size of $2^{15} = 32\,768$.

Query Evaluation

We primarily measure the performance of our bitmap index-based interval trees for the point and range queries. We investigate the following

1. How fast are interactive response times when bitmap index-based interval trees are used to evaluate point and range queries?
2. What are the storage space requirements across different index types?
3. What is the precision when using interval trees for point and range queries?

To address each of the above points, we have built five types of indexes from the given test dataset, i.e., one *Point Index* and four range indexes *NR2*, *NR4*, *WR8*, *WR16* with varied interval sizes 2, 4, 8, and 16, respectively. For example, the narrow range index *NR4* for a particular attribute is constructed using interval value ranges, where the resulting tuples per segment in that range are approximately 4, excluding duplicates. We use randomized binary search trees for all indexes, and the bitmaps are compressed, except that the nodes in the point index are augmented with the tuple IDs. Figure 6.10 shows the index size comparisons. As expected, the wide range indexes only used ≈40% of the storage space compared to the point index for the same-sized data set. Although it facilitates faster query responses, this comes at the expense of accuracy, as the wide range index used for a point query is expected to return a high number of false positives. For containment queries, bitmap indexes are the most space-efficient compared to Bloom filters and inverted indexes.

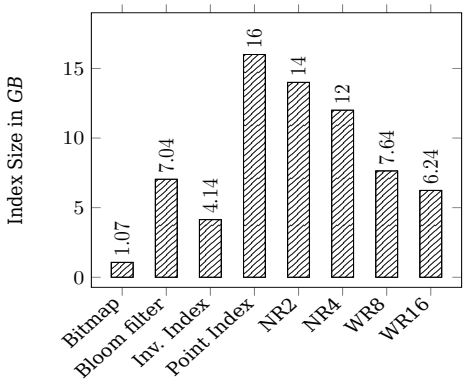

Figure 6.10: Index size comparison

To measure both the interactive latencies and the accuracies, we run three different types of queries on these indices. The point queries with an equality predicate should return all tuples in a segment for an attribute with the exact matching value. We consider narrow-range queries as the query intervals where the resulting tuples per segment are equal to or less than 8, and the query intervals for wide-range queries, where the resulting tuples per segment are more than 8. We vary this input query interval range from 1 to 128, where 1 is the point query. Figure 6.11 shows the precision scores and latencies measured at the application layer at the end of each interactive exploration operation for each input query interval range. As expected, the point index achieves a perfect precision score of 100%, setting the benchmark for other indices. Although the range indexes fare poorly for point queries in comparison, it is evident that the higher the input query interval range, say 128, even the wide range index achieves higher precision.

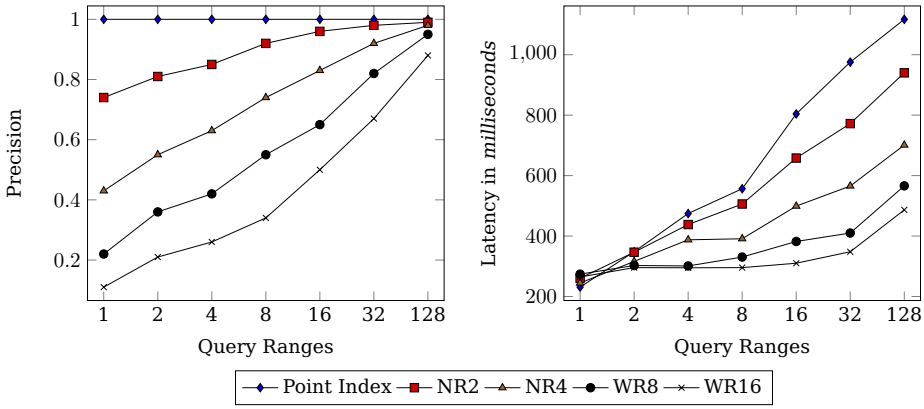

Figure 6.11: Precision and latencies measured for point and range queries

Concerning interactive latencies, the wide range index is the fastest overall. The reason is the smaller index structure, faster traversals, and faster query response. The average latency measured with bitmap indexes for containment queries is still 3× faster than the range queries. By design, the materialization phase of the analysis removes any false positives before downloading the explored search space to the disk. This gives us the leverage to tune and operate the multi-step analysis at slightly lower precision levels, achieving faster interactive query response times and reduced storage space. Nevertheless, in environments where higher accuracy is expected, point indexes remain the best fit for any query, albeit at the expense of increased interactive latency and storage space.

6.5 Summary

This chapter introduces a foundational evaluation framework and various index data structures, providing a multi-step exploratory data analysis process for heterogeneous bioscience data. We propose a data partitioning method that complements a fast, scalable, space-efficient, and flexible indexing scheme. The challenges posed by the underlying data model heterogeneity can be addressed using either a multi-model or a polystore architecture. Our experiments demon-

strated that our indexing scheme is effective within these architectures and can further enhance query execution times.

In the next chapter, we outline a data management system that employs a polystore architecture with an extended index store. This system will utilize the indexes for query optimization and execution. We will measure the effectiveness of these indices for cross-model queries using the evaluation framework and compare the results with those of the state-of-the-art.

Chapter 7

Polystore Architecture

In this chapter, we explore the polystore architecture for managing datasets, leveraging the data partitioning and indexing scheme introduced in the previous chapter to facilitate faster query executions. The idea is to develop the architecture with multiple data stores per data model and the middleware for cross-model query parsing, optimization, and execution. We also establish the foundation for enhancing the polystore architecture with a specialized key-value index store and provide a detailed description of the implementation specifics for each index type.

Polystores utilize virtual integration methods to integrate heterogeneous data stores, providing a single or multiple query interface APIs. The objective of polystores is to use specialized data stores, leveraging the internal processing capabilities of individual database engines per data model and optimizing for different workloads [52]. The design principle of modeling systems based on data, rather than trying to fit complex multi-model data into a single data model, makes polystores a viable data management solution for maintaining and analyzing diverse datasets spanning multiple data models [53, 54]. Polystore architectures support data transformation and integration during query processing, striving to reduce operational complexity depending on the type of query loads. In contrast, traditional data management solutions that follow materialized integration methods and ETL(Extraction-Transformation-Loading) pipelines first require extracting data from the data sources. Later, they must be transformed into a suitable data model and loaded into a data store or warehouse to run the analytic queries [55].

Typical polystore systems are generally designed around the individual data stores per data model and their respective query processing capabilities [56]. To achieve semantic completeness, they strive to maintain all capabilities of the underlying storage engines. However, polystore systems designed for specific business requirements, such as multi-omics, must overcome various challenges when using multiple data sources. For example, underlying data sources may have limited join capabilities, and complex cross-model queries executed on multiple data sources may require rewriting into individual subqueries and joining the intermediate results at the middleware layer [57, 58]. Despite the challenges, one of the expectations for data analysis and exploration is to enable faster execution of cross-model queries or bind-joins [1]. The primary focus of any polystore system is generating optimized query execution plans while leveraging the indexing and join capabilities of the underlying database engines. Although existing polystore systems can handle query rewriting, overall query performance depends directly on the query performance of individual data stores. In particular, when using NoSQL data stores that generally offer limited join capabilities or data stores with primitive indexing support, we need novel indexing and joining techniques at the polystore level to boost the overall query execution time.

7.1 Cross-model Joins

Bind-join or cross-model join is the type of join that gets evaluated between several data sources. In virtual integration systems, a simple bind-join includes at least two relations from different data sources with a join predicate. These bind-joins could be of any join type, where the result set gets queried from both data stores [59, 60]. The original query is typically rewritten as multiple sub-queries to push the join conditions. The query optimizer analyses and rewrites the incoming user query into subqueries native to the respective data sources. The optimizer estimates the result sizes of individual sub-queries, where the sub-query with the expected smaller query result set takes precedence. The intermediate results from the first sub-query are used to rewrite the second sub-query and are pushed down to the other data source to evaluate the join predicate [61]. Typical mediator-based systems desire that one of the sub-queries

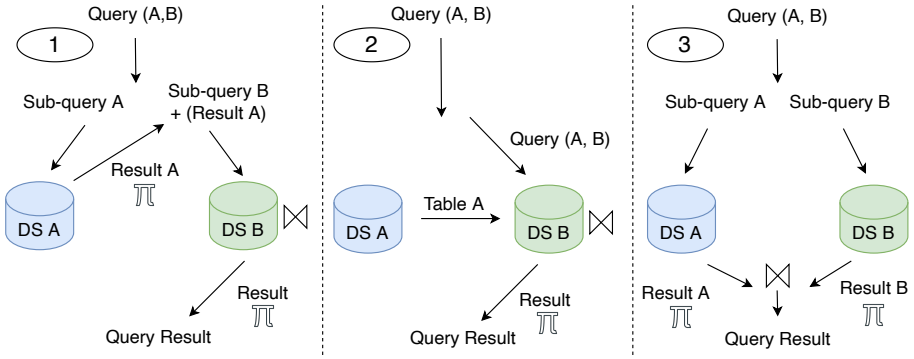

Figure 7.1: 1) Sub-query push down, 2) Table migration, 3) Middleware joining the intermediate results

yields a smaller result set, such that the intermediate results can be pushed down to the other participating data store. For example, consider two distinct heterogeneous data sources of different data models, DS_A and DS_B, as part of a polystore system. Let us assume that both data sources support the IN operator such that the values from the intermediate result from one sub-query can be pushed to the other data store. Depending on the query type, the system can perform one of the steps after query optimization.

Sub-query pushdown

Here, the middleware pushes the result of the first sub-query with an expected smaller result set to the other data store, say, DS_B as shown in Figure 7.1(1). The middleware uses the data store DS_B to perform the actual join. It works well when the distinct items of the result set produced by sub-query A are low, such that the full result can be included in the IN operator of sub-query B and pushed down to data store DS_B. Also, the intermediate results can only be iterated per value in data stores where the IN operator is not supported. In that case, a separate sub-query can be executed per each value, and the results are incrementally combined.

Table migration

In this case, the entire table (perhaps, the smaller table) from one of the data sources is migrated to the other to speed up the join and overall query response. As shown in Figure 7.1(2), the original query is routed to the data source where all the tables for the join will be available to perform the join operation. The actual join query performance depends on the native implementation of the individual data store. Additionally, the data may need to be cast or transformed into the data model of the target data store, which could be a bottleneck.

Middleware joining the intermediate results

Here, the expected extensive query result set of sub-query prompts the middleware to join the individual result sets within the middleware layer as shown in Figure 7.1(3). Depending on the optimized query execution plan, the middleware decides whether to push all projected columns or just the IDs for join predicates. In the latter case, the middleware must combine the second sub-query results and include all the projected columns from the first sub-query before returning the final query results to the user.

7.2 Motivating Example

Consider two data sources, DS1 and DS2, where DS1 contains a list of bio-science experiments, and DS2 comprises collections of bio-entities (data output from individual experiments). Several options exist to answer cross-model join queries like *list of all bio-entities with the property search_engine_score greater than 0.5 generated by experiments with buffer_condition X.*

1. using sub-query pushdown, where one sub-query (say, *list of all experiments with a buffer_condition X*) is pushed to DS1. The intermediate query result from the first sub-query is used to rewrite the other sub-query (*list of bio-entities with search_engine_score* > 0.5) such that the actual join happens in DS2, which is similar to performing a semi-join in DS2 by sending the join keys from DS1.
2. or pushing both the sub-queries to individual data stores and joining the intermediate results at the middleware layer.
3. or using table migration, where one of the smaller collections (say, experiments from DS1) is entirely migrated to DS2 before executing the join query directly in DS2.

Assume that the middleware has access to global index structures on experiment collections in DS1 for the attribute *buffer_condition* $= X$ and bio-entity collections in DS2 for the attribute *search_engine_score*. In the options mentioned above, the middleware can directly get the list of experiments (IDs) from the join operation's global index without executing the sub-query in DS1. Similarly, the second sub-query can be refined using the other global index on the attribute *search_engine_score*. Additionally, we continue to utilize the native optimizers, indexes, and query processing engines of the underlying data stores to process sub-queries, which can further enhance the overall cross-model query execution.

7.3 Related Work

We present some relevant approaches that perform data integration involving multiple data sources and cross-model querying. Polystores and distributed query processing frameworks are well-known classes of systems that enable query processing across heterogeneous data stores and support multiple query interfaces. Polystore systems enable query processing across heterogeneous data stores and support multiple query interfaces [52]. Wrapper or mediator-based technologies were one of the early models used to overcome heterogeneity, where individual wrappers encapsulate each data source and provide uniform access to data. The wrapper architectures provide data access APIs, query language, and capabilities to help support global query evaluation and optimization.

Garlic, one of the early wrapper-based systems, standardizes the description and access of information from data sources [62]. Garlic models the data as object collections where individual wrappers supply the description of the data source in Garlic definition language to maintain the global schema at the system level. The wrappers and the middleware optimize the query depending on the type. In particular, Garlic uses a bottom-up approach for bind-join queries involving multiple data sources. Using the wrapper join capabilities, the middleware first decides on the inner data source/node and the outer node. The middleware passes the values produced by the outer node of the join to the inner node for evaluation of a subset of join predicates. Garlic relies on the wrappers of the inner node of a bind-join to provide some mechanism for posing parameterized queries to pass the values from the outer node.

BigDAWG is a recently developed full-scale polystore system that uses virtual integration principles to enable an integrated view and query capabilities on multiple heterogeneous data sources [53]. The BigDAWG polystore architecture supports various database systems with different data models and uses a middleware layer that provides a uniform multi-model interface. The groups of individual data sources by the data model form *islands* say, relational, array, or text. The operators defined on the individual data model are translated into queries for the underlying data sources by *shims*, which connect the *islands* to the underlying data sources. BigDAWG evaluates cross-engine queries in two ways: it performs shuffle joins by migrating data to different data stores or splits the input query into sub-queries per data store and integrates the intermediate results [63].

In the Cloud Multi-Datastore polystore system, the querying language CloudMDsSQL provides a high level of control where each sub-query targets a specific data store [64, 65]. CloudMDsSQL also breaks down the incoming cross-engine user query into sub-queries for individual data stores. It uses a wrapper or mediator-based approach; thus, the sub-queries are pushed to the individual data stores when required. In the case of bind-joins, the mediator must either combine the intermediate results or the resulting values from one sub-query, which are packaged into a subsequent sub-query and pushed to the data store, where it performs the actual join.

MuSQLE [66] is another polystore system for SQL-based analytics over heterogeneous data sources. It supports a generic SQL engine API that undertakes

the integration of underlying database engines. It uses Spark data frames to provide functions to push an SQL sub-query into the respective execution engines. QUEPA [67] introduces augmented search to let users query the polystore without knowing the structure of data in the underlying data stores. The QUEPA system represents the data in key-value pairs and captures the relationships between the data objects from individual data stores in undirected graphs. Augmented search includes automatic expansion of the query result over one data store with the data relevant to the query, but stored elsewhere in the polystore.

In general, the primary focus of each polystore system is generating the optimized query execution plans and using the join capabilities of the underlying database engines. There is no separate indexing layer to improve the query performance, particularly cross-engine joins. PolyphenyDB proposes using polystore indexes that are to be accessed by the query engine to simplify the query before its execution [68]. Similar to our approach, PolyphenyDB suggests utilizing these indexes to streamline the rewriting of subqueries before they are pushed to the underlying data stores.

Open-source distributed query processing frameworks such as Apache Drill support querying data from multiple data sources [69]. It enables SQL-based querying over both the relational and NoSQL data stores. However, Drill can only leverage indexes created in the data sources for index-based query plans that use indexes versus full table scans to access data. PrestoDB [70] is another well-known analytical query execution engine over heterogeneous data sources. It provides engine-specific connectors similar to Apache Drill and provides a distributed execution model for querying data from external data sources.

In summary, the above approaches either do not use an additional indexing layer at all or, in the case of Drill, have severe limitations in index usage. Our research, presented in this chapter, tries to fill this gap.

7.4 Proposed Architecture

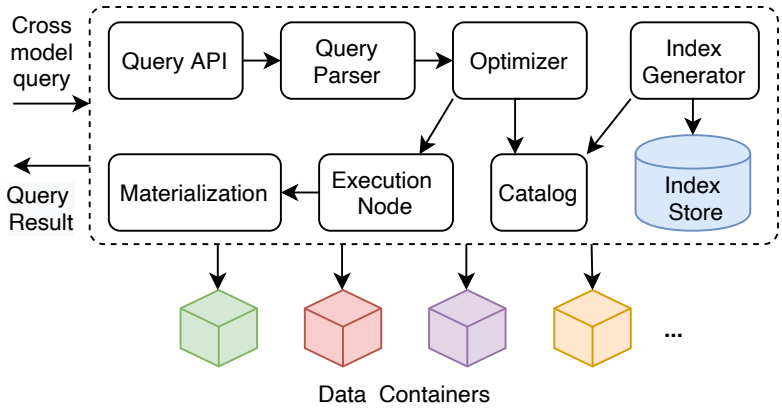

Figure 7.2: Proposed polystore architecture for multi-omics data management.

We provide a brief overview of our proposed polystore architecture, consisting of several layers and associated components, as shown in Figure 7.2. As discussed earlier, individual -omics study datasets can be packaged into research contexts containing measurement data, experimental metadata, annotations, tools, and scripts. One of the non-trivial tasks in using a polystore is selecting a suitable data model for the various parts of the complex, heterogeneous data. Traditional polystores systems are designed around having at least one database engine per data model. Their optimizers try to maximize the individual query processing abilities of the underlying database engines. In contrast, our polystore system replaces the data stores with data containers of any data model. Query optimization and execution are based on Polystore indexes, thereby avoiding the need for data stores with superior indexing and query processing capabilities.

Data containers

The data containers are generated from the individual -omics research studies. They can be any primitive data stores of any data model and provide uniform access to data through APIs. We model the raw data as collections of objects with attribute values. Each object within the collection gets identified with a unique monotonically increasing sequence number. Given the ID, each data container supports the minimum functionality of returning the object from the collection. A standard program at the source generates data containers from individual research contexts. Once attached to the multi-omics application, the data from individual containers are indexed and ready for multi-omics analysis.

Schema Catalog

The global schema is maintained in the catalog, which is required for query parsing and optimization. The data references within and across the object collections are explicitly identified from the schema. The catalog also maintains the local schema information from the individual data containers when attached to the polystore.

Query API, Parser & Optimizer

The query parser receives the incoming user queries through the query API and then performs the syntax checks and validations against domain query constraints. The query optimizer is responsible for the planning and optimization, where the parsed query gets translated into a system-specific execution plan. To prepare the optimized query plans, it identifies the query type, the data containers involved, and the list of indexes required from the index store accordingly.

Index store

A separate key-value store to maintain the index data structures augmented with bit vectors to quickly retrieve object IDs of respective data tuples/documents/nodes from individual data containers. The keys for individual index data structures follow a specific naming convention for easy identification and retrieval of indexes during query optimization and execution.

Execution node

It is the middleware component responsible for executing the optimized query execution plans. It utilizes a querying concept that complements the indexes stored in the index store. We continue to use the basic evaluation framework with the polystore indexes; query execution starts by initializing the n-dimensional search space, which involves selecting the superset of the object IDs. It then progressively filters the IDs by evaluating the predicate conditions on the attribute values, where the search space gets updated after evaluating each predicate condition. Ultimately, the search space, comprising a list of object IDs, materializes into the query result in the user's chosen data model.

Index Generator

As the name suggests, the index generator is responsible for creating and maintaining polystore index data structures. It is an offline process where data analysts select attributes from data object collections to be indexed ad hoc. Also, the metadata of individual polystore indexes is maintained in the catalog and used during query optimization.

Materialization

Since our query execution mechanism retrieves the list of individual object IDs, the materialization module is responsible for retrieving corresponding data from individual data containers and formatting the final query result.

7.5 Key-value index store

The key-value model is the simplest form of data modeling, where keys are mapped to values, and is ideal for creating and maintaining polystore indexes. While the data type of the *key* is usually a string, data structure stores such as Redis [71] provide a collection of native data types and data structures for storing the *values*, such as sets, lists, or bitmaps. When augmented with relevant IDs of data tuples or documents, the middleware can utilize these data structures during query processing to improve query performance, particularly in cases involving cross-model queries. It needs to know the right *key* to retrieve the corresponding *value* in the form of a data structure augmented with IDs that satisfy the given query. For example, multi-omics queries on -omics data are the bind-joins executed with several data containers. The resulting data tuples/documents satisfy specific refinement patterns, which are essentially a set of Boolean combinations of attribute-value pairs. We can quickly evaluate these refinement patterns by encoding the attribute value pairs as bit vectors, each corresponding to a specific object ID.

An independent key-value store maintains the index data structures to retrieve objectIDs of respective data objects from individual data containers. The index generator creates and maintains the global index data structures. It ensures that indexes are created only for partitions on a particular attribute value from the object collection. Also, the metadata of each index is maintained in the catalog. The execution node follows steps in the plan to execute any select-

(filter)-join-project query. The materialization module helps retrieve data from individual data containers and formats the final query result.

We classify the index data structures into two categories. The *base indexes* are generated on the attributes from individual data partitions of object collections from each data container. The value type of the base indexes can be lists, hash maps, or bitmaps that can directly return the corresponding objectIDs from a specific data partition. The *indexes on indexes* are the tree data structures built on top of the base indexes within a data partition and return a pointer, i.e., the *key* of the respective base index. The base indexes can be used to evaluate the containment or equality predicates, and the tree indexes can be effectively used for range predicates.

7.5.1 Challenges

Using the key-value store to maintain indexes within a polystore landscape presents several challenges. Firstly, we need a strict naming convention for the keys to identify the exact attribute name, value, collection, and data container. During query optimization, the optimizer identifies the list of index structures required in the execution plan, based on the query type.

Secondly, the size of the data structures should be manageable to minimize the overhead of transferring data from the index store to the middleware. To keep the index sizes manageable, we maintain the given data across the individual collections in partitions of non-overlapping subsets.

Thirdly, compression techniques are needed to minimize the storage space requirements for index data structures. Since we use bit vectors, there are effective state-of-the-art compression techniques that we could implement to reduce the overall storage space [72,73]. Finally, we need to maintain the indexes according to data updates.

In the subsections below, we use the terms "data store" and "data containers" interchangeably to describe how the polystore indexes are generated. We generate polystore indexes only for atomic data types, such as numbers, strings, or booleans, and not for compound data types, such as arrays and nested objects. Also, we interchangeably use the terms objectID and rowID, denoting either a tuple, document, or node in a relational, document, or graph model.

7.5.2 Base indexes

Bitmap indexing is a proven indexing technique for large datasets. A primary bitmap index consists of a bit vector for each distinct value of the domain of the indexed attribute. Bitmap indexes demonstrate significant benefits, especially for equality or low-selectivity queries, in quickly retrieving all satisfying row IDs [74, 75].

We use the bitmap index to demonstrate the base index creation process, which is similar to other indexes, such as lists and hash maps. For the n_{th} position of the attribute $attr$ with value x, the n_{th} position of the bit vector corresponding to the value is set to 1. To retrieve the rowIDs for value x, we take the corresponding positions of 1s in the bitmap index for $attr = x$. Consider the ex-

ample bitmap indexes in Figure 7.3 where the attribute *attr* has a total of three indexes, one for each distinct value of the domain. By partitioning the data into non-overlapping subsets of size 5, we maintain one bitmap index for each distinct value of the domain for each partition. Each bit vector's size would be the same as the partition size, and the individual bit vector becomes the *value* of a certain *index-key* in the index store. The monotonically increasing rowIDs in the base data make it trivial to compute the actual rowIDs from fragmented bit vectors. For a fixed segment size s, the bit position k in a bitmap index of a particular data partition p, the corresponding rowID in the base data can be computed as $((p-1)*s)+k$.

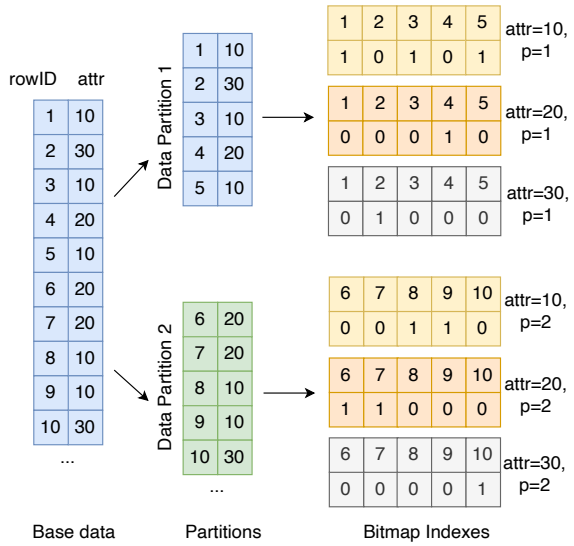

Figure 7.3: Fragmented bitmap indexes

Fragmenting the indexes offers flexibility in creating and maintaining the bitmaps per data partition. We can parallelize the index creation and confine the updates to specific partitions. The most important advantage is the flexibility to distribute the indexes across a key-value store cluster.

The naming of the individual bit vector in the index store plays a vital role in query optimization and during execution. While generating the fragmented indexes for each data partition, we follow a bottom-up convention in naming the bit vectors. We name the keys in the index store in the format *<index type>_<data store>_<collection>_<partition>_<attribute>_<value or range>*. It is easy to reconstruct the index key for a specific attribute value, given the data store and collection names. The index type (bitmap) and the actual value of the attribute must be part of the key, as shown in Figure 7.4. The optimizer can access the index store's catalog of all existing index keys while preparing the query execution plans. During query processing, the execution node uses the index keys to retrieve the corresponding bit-vectors and perform subsequent bit-wise operations.

Bit vectors are particularly handy when evaluating Boolean expressions, espe-

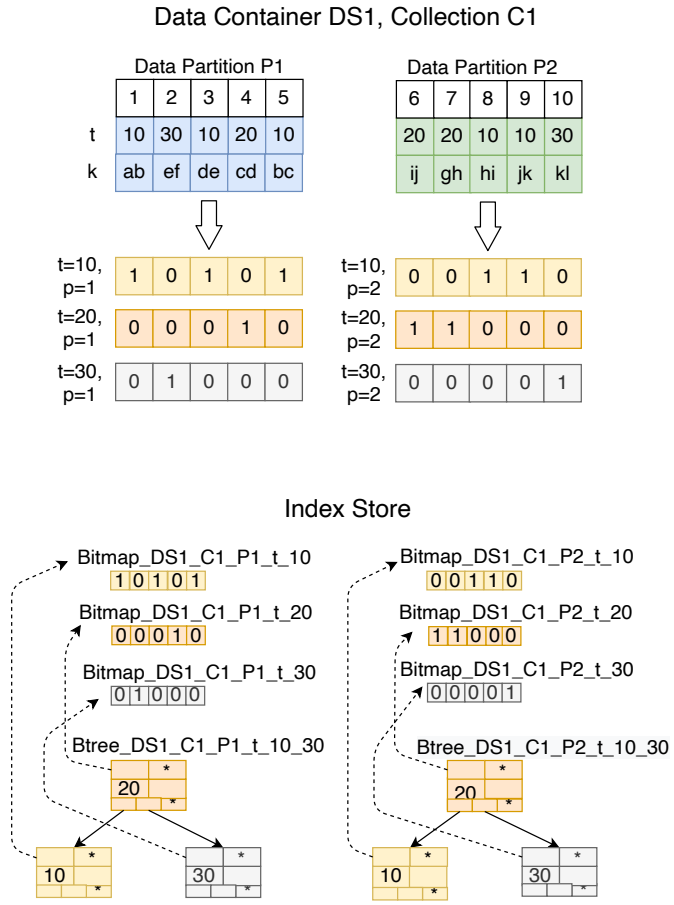

Figure 7.4: Bitmap and tree indexes per partition.

cially equality predicates. For the example in Figure 7.4, to find the matching rowIDs satisfying the predicate expression $t = 10\ OR\ t = 30$ from the collection C1 and data store DS1, the execution node needs to retrieve a total of four bitmap index-keys for all partitions of attribute t corresponding to the values 10 and 30. As the partition sizes are the same, we can perform a bit-wise Boolean OR operation between the respective bitmap indexes by partition to compute the resulting rowIDs 1,2,3,5,8,9 and 10.

7.5.3 Tree indexes

We build *indexes on indexes* as tree structures augmented with pointers to the respective base indexes of a data partition. The payload of each node in the tree index structure would be the *key* of a particular base index in the index store. The tree indexes can effectively evaluate the range predicates for data types such as integers and strings. The tree index would return the respective *keys* of the individual base index structures that can directly give the corresponding objectIDs. As shown in Figure 7.4, the tree nodes consist of the values of the attribute t, and each node-key pointing to the base index (in this case, a bitmap

index) that can directly return the positions of IDs.

We introduce interval tree indexes to facilitate range queries on data types such as float. The primary difference between the standard tree index structure and the interval tree index is that node keys contain bounded intervals rather than a single value. We consider only closed intervals and include both endpoints. When we extend a tree structure for non-overlapping interval ranges, each node n in the tree holds the element x, which is assumed to be part of the interval $[x_i, x_j]$ with x_i being the low-end point and x_j becoming the high-end point.

Data: Index Name N, Attribute $A_k[]$, Interval Size S, Base Index Type T
Result: I-Tree index N for the Attribute A_k

```
1 initITreeIndex(N) ;
2 Sort the values of the array A_k[] ;
3 Split array A_k[] into bins B[] of interval sizes S ;
4 foreach bin b in B[] do
5 |   baseIndexKey ← generateBaseIndex(b, T) ;
6 |   addToITreeIndex(baseIndexKey, N) ;
7 end
```

Algorithm 1: Interval tree index generation

Algorithm 1 outlines the interval tree creation process. We first sort the data by the attribute's value when generating the interval tree index structure for an attribute within a data partition. We then use binning to split the data into intervals of pre-defined sizes. Also, we ensure that the repeated values are enclosed in the same interval to avoid overlaps. Binning divides the attribute's values into arbitrary bins, each containing a disjoint range of values. Each bin's distinct value effectively produces a primary base index. For each interval, we first generate the individual base indexes of the data partition size with the set of corresponding objectIDs, before augmenting with the tree nodes. The node keys consist of the intervals of the attribute's values, and the payload is the pointer to the base indexes for each interval range. The search algorithm finds all the intervals intersecting the given query interval q and returns a base index per partition. To find the resulting interval r intersecting with the query interval q, we must check two conditions. One, the start and/or endpoint of the interval r is in q, or the interval r completely encloses the query interval q.

7.5.4 Index updates

In the multi-omics scenario, when there are changes to the data within the container, we need to capture the changes to reflect the same in the indexes. As we generate and maintain the indexes per data partitions, updates and deletes are confined only to the specific partitions. Any changes to the values of the attributes of the data require updating the respective indexes, depending on the type of indexes existing for the attribute. Since the tree indexes are mere pointers to the underlying base indexes of values or value ranges of an attribute, they would require no changes unless the data changes require a new base index.

In the case of an existing tree index for the attribute, the first step would be identifying the affected base index per the changes to the attribute value in the

partition. For the existing values of the domain, we update the base indexes that correspond to the old value of the attribute, and when the attribute value gets changed to a new distinct value of the domain, a new base index that corresponds to the latest value of the domain needs to be generated. The existing tree index gets updated with the new base index.

Data: iTreeIndexKey I, prevValue pV, currValue cV, rowID pos
Result: Updated I-tree index

```
1  pVBitmapIndexKey ← getBitmapIndex(I, pV) ;
2  updateBitmapIndex(pVBitmapIndexKey, pos, 0) ;
3  cVBitmapIndexKey ← getBitmapIndex(I, cV) ;
4  if cVBitmapIndexKey <> null then
5  |  updateBitmapIndex(cVBitmapIndexKey, pos, 1) ;
6  end
7  else
8  |  cVBitmapIndexKey ← generateBitmapIndex(cV) ;
9  |  addToITreeIndex(cVBitmapIndexKey, I) ;
10 end
```

Algorithm 2: Interval tree index updates

7.5.5 Redis index store

Redis key-value store offers to add new user-defined data structures through Redis Modules. We implemented two types of tree structures, namely, B-tree and I-tree data structures in C++, and integrated them as new data types with Redis version 6.2.4. Table 7.1 lists the new Redis commands we use to generate the polystore index data structures and for quick data lookup during query processing. These new commands help minimize data transfer between the index store and the middleware since Redis only returns the value (a reference key to a bitmap) instead of the entire data structure itself. Redis offers the flexibility to distribute the indexes across the cluster since there can be many indexes per attribute.

7.5.6 Binding attributes

We can leverage the relationships in the base data while generating the polystore indexes and speed up the cross-model queries. The attributes of a particular collection referenced by the other collections from different data stores can become potential binding attributes. The global schema must be known beforehand to identify the attributes being referenced. The binding attributes can also be the natural join partners among the data collections. Binding attributes help maintain the data relationships and hierarchies among the underlying data stores, independent of the data model. They can primarily act as a link between data residing in multiple data sources and can be picked before generating the polystore indexes. For data stores with limited join capabilities, the binding attributes between the individual collections of the same data store can be used to perform a regular join operation within the collections of the data source.

Like polystore indexes, we can maintain binding attribute cross-references in

Redis Command	Description
BTREE.CREATE [key] [nodetype] [order]	creates the B-tree and is assigned to the given *key*, where the *nodetype* is the data type and the degree of the B-tree is *order*.
BTREE.INSERT [key] [nodeKey] [objectId]	inserts the *nodeKey* value and *objectID* into the tree with the given *key*.
BTREE.FIND [key] [nodeKey]	finds the *nodeKey* inside the tree and returns its payload of the list of IDs.
ITREE.CREATE [key] [dataType]	creates an interval tree with the specified *key* and the interval *dataType*.
ITREE.INSERT [key] [min] [max] [value]	inserts a new interval, bounded by min and max, into the i-tree and attaches the *value* to its payload. The *value* argument is another Redis key string that points to the appropriate interval bitmap.
ITREE.FINDINTERVALS [key] [min] [max]	returns all intervals of the i-tree that intersect with the query interval bounded by min and max.
ITREE.FINDVALUES [key] [min] [max]	returns all the bitmaps corresponding to the result intervals.

Table 7.1: New Redis commands for the index data structures.

the form of key values in the index store, where the *key* represents a distinct domain value of the binding attribute. The *value* is the list of data partitions referenced by the value of the binding attribute.

7.6 Cross-model query execution

Our multi-omics application accepts user queries in the application-specific standard JSON query format to hide the heterogeneity of data sources from the end-users. However, the user queries can also be submitted in standard SQL, where an additional SQL parser decomposes them into the JSON query format before identifying the data stores, collections, binding attributes for joining, and indexes needed to execute the query. Typical analytical queries include cross-model joins between multiple data sources.

The optimizer then transforms the JSON query plan into an internal JSON execution plan with pre-defined logical sections in the sequence in which the execution node executes the plan, as shown in the Algorithm 3. For user queries involving a single data source, the optimizer simply rewrites the query in its native format so that the execution node can push it directly to the source. However, when using the indexes, the optimizer follows a pipeline of tasks to generate the optimized operator tree before writing it in a JSON execution plan. Apart from the traditional optimization techniques such as join ordering and selection/projection push-down, the optimization pipeline also includes the steps of range condition creation to effectively use the interval trees and index

selection to pick the most suitable indexes for each predicate. In some instances, pushing the sub-queries and joining the intermediate results can be faster than using the indexes, needing to evaluate the combined cost of executing the sub-queries within the data stores against the cost of using the indexes.

Data: Input Query Q_i
Result: Query result Q_R

```
1  if not cross-model query then
2  |  Q_s ← toNativeFormat (Q_i) ;
3  |  execSubQueryPushdown (Q_s) ;
4  end
5  else
6  |  if exist(indexes) then
7  |  |  Q_t ← generateOptimizedOpTree (Q_i) ;
8  |  |  Q_e ← generateIndexExecPlan (Q_t) ;
9  |  |  execIndexPlan (Q_e) ;
10 |  end
11 |  else
12 |  |  Q_s ← generateSubQueryPlan (Q_i) ;
13 |  |  execSubQueryPushdown (Q_s) ;
14 |  end
15 end
16 def execIndexPlan(Q_e):
17 |  S ← initQueryResultSpace () ;
18 |  p ← readPrimarySource(Q_e) ;
19 |  evalSubQuery(p) ;
20 |  loadDataPartitions(S, p) ;
21 |  DS[] ← readOtherDataSources(Q_e) ;
22 |  foreach DataSource d in DS[] do
23 |  |  P[] ← fetchPredicates(d) ;
24 |  |  foreach Predicate p in P[] do
25 |  |  |  applyPredicate (S, p) ;
26 |  |  end
27 |  end
28 |  Q_R ← materialize(S) ;
```

Algorithm 3: Cross-model query optimization and execution

Algorithm 3 shows how a cross-model query gets executed from a query execution plan, which consists of two main logical sections: initialization and predicate evaluation. The optimizer prepares the initialization steps using the original cross-model query between the participating data stores. It can either include the optimized sub-queries for initializing the respective dimension of the query result space with the data partitions or the list of objectIDs retrieved using the indexes. Each dimension corresponds to an individual data store in the cross-model query. The predicate section consists of decomposed query predicates in the form of an array of attribute names and the respective index types and names, as well as the simplified Boolean operations between the predicates to be applied progressively to the respective dimension of the query result space.

Ultimately, the execution node finds the list of objectIDs satisfying the query predicate for each dimension. It passes them on to the materialization module to extract the data from individual data stores and format the final query result.

The query optimizer ensures that the number of sub-queries needed to be pushed down to the individual data stores is minimized to take advantage of the indexes. A sample index execution plan is given in Listing 7.1. Consider the example of two data sources, DS1 and DS2, containing the list of bio-science experiments and bio-entities collections, respectively. To execute the cross-model join query and fetch list of all bio-entities having the property *q_value* between 0.0001&0.0003 and *abundance* between 400.5&480.8, that are generated by experiments with buffer_condition X, the system pushes the first sub-query and fetch the list of all objectIDs of the experiments with a buffer_condition X from DS1 (PostgreSQL) to initialize the query result space, which is an in-memory data structure with the list of objectIDs and binding attribute values. It then uses the binding attribute cross-references to identify the referencing data partitions of bio-entities collections from DS2 (MongoDB). It progressively filters the objectIDs of DS2 by evaluating the predicate conditions on the attribute values, where the query result space gets reduced after evaluating each predicate condition. Lastly, the query result space with the remaining list of objectIDs is materialized into the final query result.

Using interval trees and data partitions in predicate evaluation might introduce false positives, thus affecting the precision of the intermediate query result [76]. The materialization module performs the condition re-check for false positives before materializing the final query result.

```
    "queryResultSpace": {[{
            "subQuery": "SELECT id AS sample_id FROM samples WHERE
                ↪ buffer = 'X'",
            "dataSourceID": "postgresql",
            "bindingAttributes": ["sample_id"],
            ....
        },{ "dataSourceID": "mongo",
            "collectionName": "protein",
            "bindingAttributes": ["sample_id"],
        "predicates": [
            { "attributeName": "q-value",
               "indexType": "itree",
               "interval": [0.0001, 0.0003]},
            {  "attributeName": "abundance",
               "indexType": "itree",
               "interval": [400.5, 480.8]}],
               ....
```

Listing 7.1: A snippet of index execution plan with two data sources

7.6.1 False positives

The usage of binding attribute cross-references and the interval trees in predicate evaluation introduce false positives, thus affecting the precision of the intermediate query result. In the case of interval trees, the number of false positives in the query result depends on the difference in the interval spans between the query interval and the interval range. For example, when an interval tree gets

used to answer a point query, the number of false positives in the resulting list of rowIDs correlates with the interval sizes in the index and the input query interval size. Also, the wider interval ranges reduce the interval tree's size, thus contributing to faster traversals and better query responses.

In the case of the binding attribute cross-reference, since a single data partition might contain the data referencing the other instances of the same binding attribute, there might be false positives in the resulting rowIDs that satisfy the query predicate. The materialization module that formats the final result eliminates the false positives when materializing the query result.

7.7 Experiments

Our experiments primarily measure the effectiveness of using system-level indexes for cross-model queries. We use our multi-omics application and evaluate the query performance against the state-of-the-art BigDAWG and Apache Drill. We consider a set of benchmark queries of varying degrees of query selectivity and data types and predicate expressions on a synthetic data set for our evaluation. We compare the average query execution times against the state-of-the-art that performs joins for the cross-model queries. Our tests also include measuring the storage space requirements for indexes, and we present the correlation between the partition size, index sizes, and the query execution time.

We use two bio-science data sets for evaluation in our experiments: PRIDE and Synthetic data. As mentioned in the earlier chapters, bio-science experiments follow specific description standards to capture the *metadata*, which is the minimum required information to describe an experiment to understand the biological and technical origins of the data. Depending on the bio-science domain and the technology, the instruments generate chunks of unstructured instrument-specific data, which is then processed through quantification and identification software to produce *raw data* containing lists of bio-entities such as proteins, metabolites, etc.

The PRIDE data contains the raw data (≈1 million protein instances) extracted from selected Proteomics experiments from the PRoteomics IDEntifications Database [77]. It includes a set of measurement/contextual attributes of various data types that follow either a uniform or skewed distribution for each bio-entity in the raw data collections. Since the raw data from PRIDE requires considerable manual processing in transforming from domain-specific formats to standardized document format, we use synthetic data, a more significant data set with 20 million protein instances, which is the same as the PRIDE data in a standardized format. We use one bio-sample relation to represent the proteomics experiment metadata and collections of bio-entities (for example, proteins) in JSON format for raw data. There is no loss of generality since the central aspect of our evaluation is to execute cross-model joins between the participating data stores in a heterogeneous data store setting. We list the individual benchmark queries and their characteristics in Table 7.2. Listing 7.2 shows the benchmark query Q10. The query selects all samples from the *samples* table in PostgreSQL and evaluates the predicate expression on the MongoDB data collection *protein*. The combinations of Boolean expressions of

Query#	Data Type	Predicate Type	Operations	Query Selectivity
Q1	Integer	Point	AND	0.0335
Q2	Integer	Point	OR	0.3996
Q3	Integer, String	Point	AND, OR	0.1864
Q4	Float	Range	AND	0.00001
Q5	Float	Range	OR	0.0076
Q6	Integer, String, Float	Point, Range	AND, OR	0.0965
Q7	Integer, String, Float	Point, Range	AND, OR	0.0311
Q8	Integer, String, Float	Point, Range	AND, OR	0.2947
Q9	Integer, String, Float	Point, Range	AND, OR	0.6488
Q10	Integer, String, Float	Point, Range	OR	0.9475

Table 7.2: Benchmark queries

the attributes represent the modification patterns. In this case, we have a set of attributes evaluated for the Boolean OR operation, and the query result set is very large. They execute cross-model joins between the bio-samples relation and the proteins collection in JSON format to retrieve the matching protein instances. We run the queries thrice and average the measured execution times. We vary the contextual/measurement attributes of different data types, predicate types, and the complexity of the predicate expression from the raw data to have different query selectivities.

```
SELECT _id, segment_id, segment_position FROM mongo.cdb.`protein`
    WHERE cast(sample_id AS INT) IN (SELECT id FROM postgresql.`
        samples`) AND
    (((sampling_frequency >= 500 AND sampling_frequency <= 1500)
        OR (best_search_engine_score >= 0.3 AND
            best_search_engine_score <= 0.4)
        OR reliability = 3)
        OR (num_peptides_unique = 6
        OR (sampling_frequency >= 100 AND sampling_frequency <=300)
        OR (sample_mass >= 0.1 AND sample_mass <= 0.2))
        OR (modifications = 'True'
        OR (search_engine_score >= 0.3 AND search_engine_score <=
            0.4)
        OR reliability = 2));
```

Listing 7.2: Benchmark query Q10 executed in Apache Drill

We use the *de.NBI* cloud (German Network for Bio-Informatics Infrastructure) that runs on the OpenStack platform for our experiments. The virtual machine configuration is eight vCPUs, 32GB RAM, and a 200GB attached storage disk volume. It runs on Rocky Linux v8.5. We deploy the data stores of our multi-omics application in Docker (v20.10.18) containers and use PostgreSQL for the structured metadata, MongoDB for the semi-structured bio-entities (proteins) in the raw data, and Redis for the index store. As shown in Figure 7.5, our multi-

omics application and Apache Drill share the same data stores. We use the publicly available Dockerized BigDAWG application with its own PostgreSQL and Vertica containers. We copy the raw data in a relational format to the Vertica container before executing the benchmark queries in BigDAWG.

Our multi-omics application maintains the indexes created on measurement/-contextual attributes in the Redis index store deployed with the custom modules. We use bitmaps for base indexes and the tree indexes over the base indexes. The index node runs in a basic cluster setup of three primaries and three secondary nodes. We do not use native indexes on the measurement/contextual attributes in the data stores with the raw data. However, to compare the impact of using the native indexes, we separately run the benchmark queries in Apache Drill with the native indexes created on the individual attributes of the raw data in the MongoDB instance.

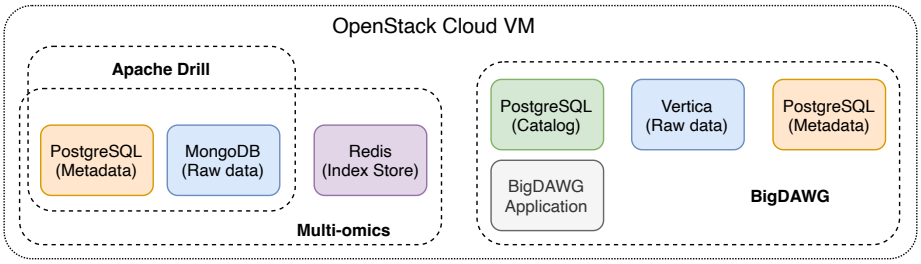

Figure 7.5: Setup in *de.NBI* cloud VM

7.7.1 Results

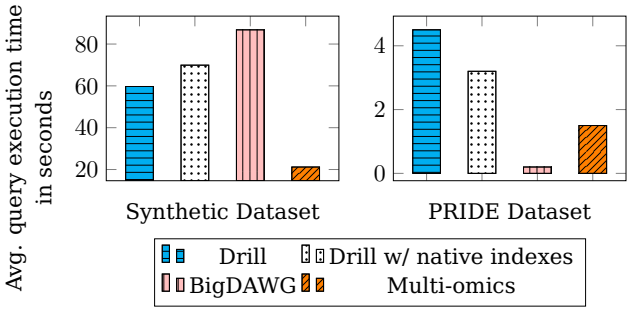

Figure 7.6: Average query execution times.

Query Execution Times: We observe that Apache Drill executed the cross-model queries by joining the intermediate results from the sub-queries at the middleware layer, and BigDAWG used table migration since the bio-samples relation in the PostgreSQL is smaller than the protein relation in the Vertica instance. Figure 7.6 shows the average query execution times across the systems between the two data sets.

We run a set of low selectivity ($< 2\%$) queries on the smaller PRIDE data set and measure the average query execution times when there is limited data movement

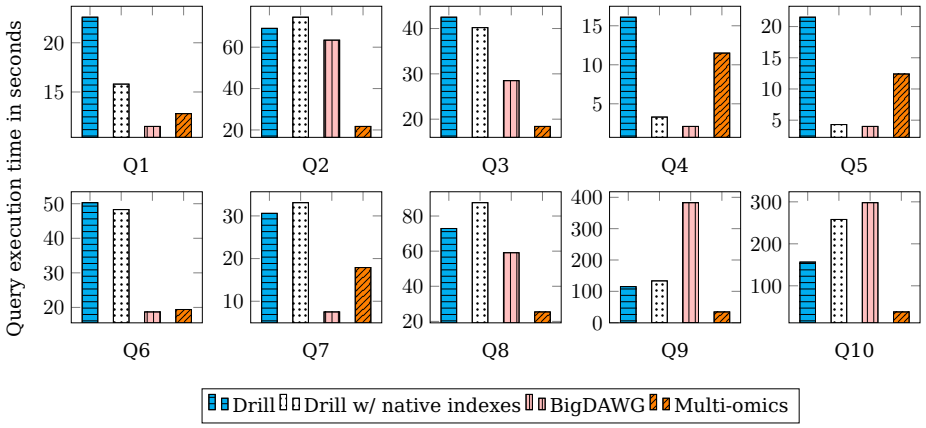

Figure 7.7: Query execution times across polystores per each benchmark query.

between the data stores and the respective middleware layer. We observe that the table migration method of cross-model execution is effective. BigDAWG is faster overall when the Vertica instance performs the joins. In other words, Apache Drill is slower due to the overhead in sub-query push-down and joining the intermediate results. However, Apache Drill with native indexes performed better as the sub-queries pushed to MongoDB executed slightly faster using the native indexes. Our multi-omics application is slower than BigDAWG because of the higher average query initialization time. However, it is faster than both variants of Apache Drill.

In the case of the more extensive synthetic data set, our multi-omics application outperformed the state-of-the-art in average query execution time. In particular, we observe that the benchmark queries Q9 and Q10 influenced the average query execution times of both Apache Drill and BigDAWG. We give the break-up of query execution times for individual benchmark queries in Figure 7.7. As expected, BigDAWG performed better in the lower query selectivity scenarios for the benchmark queries Q1, Q4, Q5, Q6, and Q7. The reason is that the table migration method of cross-model join is powerful when combined with the superior join capabilities of a Vertica instance with the raw data in relational format. However, with higher query selectivity, i.e., Q2, Q3, Q8, Q9, and Q10, our query processing using the global indexes is very effective compared to the other cross-model querying methods.

Even with the more extensive synthetic data set, Apache Drill performed cross-model joins by sub-query push-down and combined the intermediate results by performing a hash-join at the middleware layer. When available, the MongoDB instance planned the sub-queries using the native indexes. Our multi-omics application outperformed Apache Drill without the native indexes for all benchmark queries. With the native indexes, Apache Drill is faster only when executing the low selectivity queries Q4 and Q5.

Partition Size: To find the optimal data partition sizes of indexes that can yield the best average query execution times, we evaluated the benchmark queries on the synthetic data set by varying the partition sizes of 2^{12}, 2^{13},

2^{14}, 2^{15}, and 2^{16}. We generate the base bitmap and tree indexes for each partition size on a set of measurement attributes of varying data types from the protein collection. We run the benchmark cross-model queries to measure the average query execution time. We use the roaring bitmap compression [72] for the bitmap indexes in the index store.

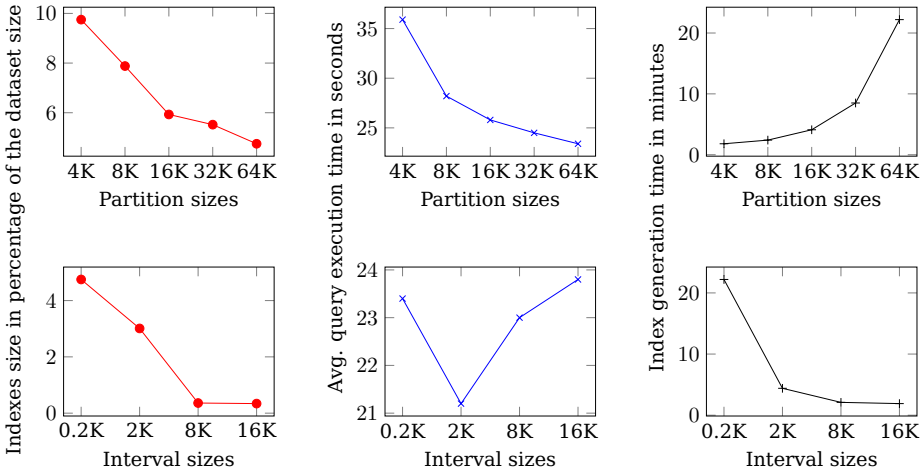

Figure 7.8: Index sizes vs. average query execution time vs. index generation time. The plots in the top row are the measurements taken by varying the data partition sizes, while the interval size for the i-trees is 256. The plots below have varying interval sizes for the fixed data partition size of 64K.

Figure 7.8 shows that the average query execution time improves as we increase the partition size. However, the average time it took to build the compressed indexes increased with the partition size. Similarly, the overall storage space for the indexes in the index store decreased since the number of compressed indexes required to be maintained is fewer with a larger partition size. The optimal partition size is close to 2^{16}.

Query Selectivity: Figure 7.9 gives the correlation between the query selectivity and the execution times. The query execution times of Apache Drill are proportional to the selectivity, with the higher selectivity queries taking longer execution times. In comparison, BigDAWG query execution times followed a different pattern, where higher selectivity queries took longer execution times, and the low selectivity queries were relatively faster. Similar to Apache Drill, the pattern of query execution times of multi-omics polystore is also proportional to the selectivity, although the query execution times are relatively faster overall. Our approach of using the bitmap indexes at the polystore level is quite powerful, particularly when executing the cross-model queries on large-scale data sets.

In Figure 7.10, we give the break-up of the query execution times for the benchmark queries run in our multi-omics polystore. For lower selectivity queries, the time taken to initialize the search space is significant compared to the total execution time. However, for benchmark queries with complex predicate

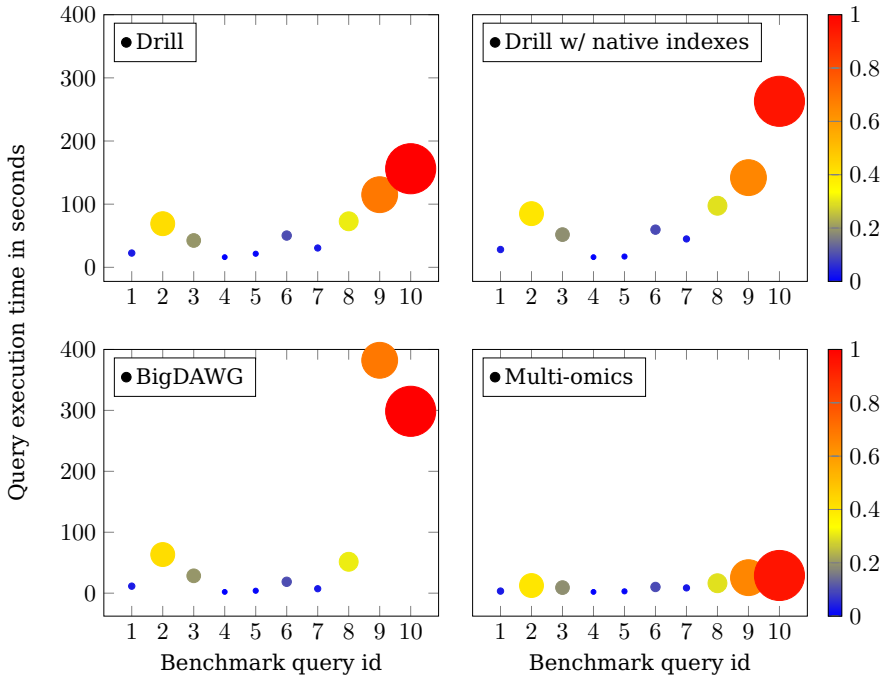

Figure 7.9: Query performance across polystores by selectivity. The size of the dots correlates with the query selectivity. The dots are color-coded for easy visual reference.

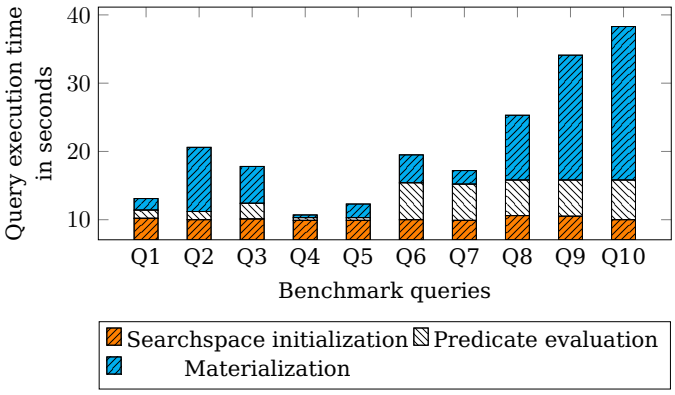

Figure 7.10: Query execution time break-up of benchmark queries evaluated on multi-omics polystore.

expressions and larger query result set sizes, as expected, predicate evaluation and query result materialization took more time in comparison.

7.7.2 Limitations

Our query execution is similar to the sub-query push-down and is orthogonal to the existing polystores; however, adapting in its current form requires changes to the optimizers and execution engines. Additionally, our system currently supports only basic query types, including containment, point, and range queries. It requires additional processing at the middleware layer to accommodate complex analytical queries. When using global indexes, we must consider the additional storage space requirements for the index store, index compression, and the time it takes to generate them. The storage space required for the indexes highly depends on the cardinality of the data and the number of attributes indexed. Lastly, the indexes can be generated and tuned per the underlying data and the workloads. In particular, the partition size can be adapted rather than using a fixed size.

7.8 Summary

This chapter makes a compelling argument for utilizing system-level indexes to integrate data from heterogeneous sources. We proposed a polystore architecture centered around a global index store. By moving query processing to the middleware layer, we reduced reliance on the underlying database engines during cross-model query execution. Our experiments demonstrate that multi-omics applications built on a Polystore architecture and an additional index store can perform comparably to state-of-the-art systems. We showed that both the index store and system-level indexes significantly accelerate cross-model queries when accessing extensive bio-science datasets and evaluating complex Boolean predicate expressions.

While the index store and the global indexes in a polystore system are a promising start, it is impossible to create all the indexes a priori in highly dynamic environments such as multi-omics data management. Additionally, query workload shifts are expected; thus, there is a need for an adaptive indexing strategy. In the next chapter, we will expand on adaptive indexing to effectively handle diverse query workloads and manage global indexing cost-efficiently. Adaptive indexing addresses the challenge of optimally selecting attributes for indexing based on query workloads, thereby reducing the costs associated with index management.

Chapter 8

Adaptive Indexing

Efficient index management is critical for scientific data management systems operating in dynamic environments, especially when dealing with diverse query workloads. To facilitate seamless exploratory querying of large-scale bio-science datasets, it is desirable to ensure a quick gateway to the data and significantly reduce query execution times. As a result, adaptive index management plays a pivotal role in creating, optimizing, and cost-effectively maintaining indices to accommodate constantly shifting query workloads.

Adaptive index management streamlines index creation and refinement with minimal manual effort, balancing cost and benefit. Unlike traditional offline indexing, where indices are built *a priori*, adaptive indexing dynamically responds to query workloads to manage the indices and continually improve query performance [78]. Database Cracking [79] exemplifies adaptive indexing by partitioning queried attributes' values based on incoming query predicates, eventually aiming to create a sorted complete index over time. Advanced cracking methods have been developed to enhance robustness, convergence speed, and overall query response times, catering to various workload types and data distributions [80–84]. Nevertheless, one of the main challenges is the potentially high cost associated with the cracking process for each incoming query. Attributes rarely queried may not justify the creation of an index for unnecessary costs and additional storage space. A complete index on unused attribute values is less beneficial than a partial index on exact values required for specific query workloads. Therefore, a cracked index may become obsolete as workload patterns change. Another challenge is the optimal selection of attributes for indexing, as the cracking algorithms cannot autonomously select attributes based on query workloads, resulting in the indexing of every query attribute by default.

We propose an index management model that builds upon the DB Cracking methods to address these challenges. We explore the dynamics of adaptive indexing using adversarial search. The model utilizes heuristics to assess the potential value of creating or refining an index to make future queries faster, leveraging a complementing DB Cracking variant. Together, it prioritizes achieving the worst-case benefit quickly for the utility of sets of indices rather than aiming for the best-case benefit. In other words, if managing a particular index for each subsequent user query results in high costs and limited future benefits, the model decides against creating or refining such an index. This underscores the importance of considering the long-term utility and cost-benefit of indices. Table 8.1 shows an example query workload sequence. The model and the DB Cracking method adjust to the workload by initiating index actions on attribute a but avoiding index actions on attributes b and c. Based on the observed workload, the model predicts that attribute a will likely be seen again and continue to refine the index. On the other hand, without the model, DB Cracking performs index actions for all attributes by default. In this chapter,

Query Workload	DB Cracking	Model + DB Cracking
Q1 → a	Create(a)	No Action
Q2 → a	Refine(a)	Create(a)
Q3 → a, b	Refine(a), Create(b)	Refine(a)
Q4 → b	Refine(b)	No Action
Q5 → a, c	Refine(a), Create(c)	Refine(a)
...	...	...

Table 8.1: An example workload sequence in which the model selects attribute a over b and c adapting to the workload.

1. We examine index management as a two-player adversarial search problem built on robust DB Cracking methods. We develop a model that aids in selecting the optimal attributes for indexing, adapting to the workload while reducing the cost of targeted index refinement.
2. We argue the need for a new DB cracking variant that complements the model by mitigating bias toward frequently occurring elements in the data and promoting fair refinement. Therefore, we introduce "Statistical DB Cracking".
3. We evaluate our adversarial search model built on Statistical DB Cracking with various workload types and demonstrate that it improves overall query response times while quickly adapting to the workload shifts and data distributions. Furthermore, we showcase that Statistical DB Cracking is comparable to existing DB Cracking variants in robustness and convergence.

8.1 Related Work

Optimal selection of indexing attributes has been a research topic for decades [78, 85, 86]. Modern index management frameworks [87–90] utilize machine learning and deep learning models to predict optimal indexing strategies for future query workloads, thereby reducing the overall cost of index maintenance. These frameworks typically leverage historical query workloads and the current state of database indices to implement cost-based reward functions for decision-making. However, they lack support for targeted index refinement, i.e., enabling refinement at the specific data level to save time and reduce index size.

Regarding reducing the costs associated with targeted index refinement, [79] emphasized the benefit of not further partitioning a cracker column from a certain point in time and the need for cost models to make decisions while processing queries. Holistic indexing [91] extends adaptive indexing techniques by continuously refining partial indices during periods of low CPU utilization. A parallel vectorized database cracking algorithm operates in the background. The system maintains the *index space*, comprising a list of candidate indices earmarked for incremental optimization. The index space includes the list of seen attributes from the query workload and the unseen attributes that are

either automatically decided by the system or manually inserted by the user. The candidate indices are strategically prioritized for refinement based on the combinations of partition size and attribute frequency within the workload or by random selection. In contrast to our model, which focuses on restricting the index refinements to yield conservative benefits quickly, holistic indexing persistently refines indices to strive for optimal benefits. Both holistic indexing and our model rely on the query workload to select attributes.

8.2 Adversarial Search

Our approach to index management involves modeling it as an adversarial search problem, also known as a two-player game. This is a deterministic and non-cooperative game where players take turns making moves and trying to outperform each other. It's a zero-sum game, meaning that one player's gain is the other player's loss, with the total utility remaining constant. In our model, we consider the application or user triggering selection queries with a set of attributes as one player, while the other is the index manager responsible for managing the indices of the attributes using DB Cracking methods. The interactions happen sequentially, with each move influencing subsequent decisions, ultimately leading to the creation or refinement of indices. One noteworthy aspect is that only the index manager can decide to change the state of the indices after each query execution, taking actions to maximize its chances of long-term success. The query workload cannot directly alter the system's state; it can only influence the index manager's actions. The index manager has perfect informa-

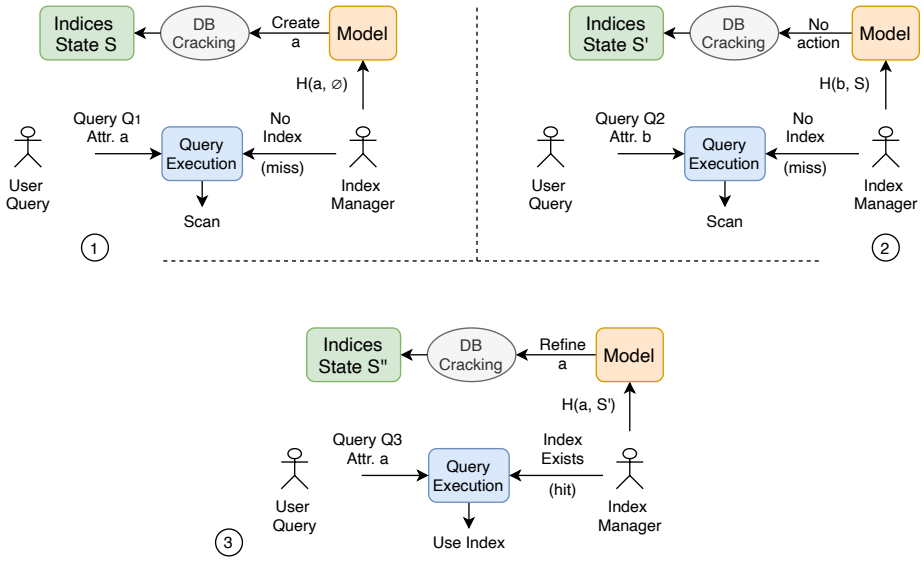

Figure 8.1: Index Management modeled as a two-player adversarial search problem in which the input query workload and the index manager act as agents/-players. The model uses the heuristic function H to determine the optimal index action for the incoming query attribute. The underlying DB Cracking method creates the index or performs targeted refinement.

tion and complete knowledge of the current state of the indices and past queries. The model is robust and resilient to the workload shifts and complements the underlying DB Cracking method.

As illustrated in Figure 8.1, we consider the simple case of one attribute per query, where each query execution is counted as one round of game-play. The query side "wins" a round when no partial or complete index is available for the attribute during query execution. After each round, the index manager uses heuristic functions based on the history of hits for the indexed attributes, including the query attribute, and the current status of the indices to decide whether to create or refine an index for the query attribute. This is done to maximize the likelihood of index hits in subsequent rounds. The model relies on the underlying DB Cracking method for target refinement of an index. The heuristic functions are tuned to the used DB Cracking variant to maximize the results. For instance, a novice heuristic function may involve a coin toss for decision-making. However, this simple approach does not complement the DB Cracking methods and cannot guarantee that the indices will adapt to the query workload as needed.

8.3 Database Cracking

DB Cracking method [79] uses a cracker column to represent the attribute being indexed and a cracker index to track the indexed portions of the attribute using an AVL tree. As shown in Figure 8.2, the algorithm splits the cracker column into two or three sorted sets based on the incoming select queries, ensuring that the queried part of the cracker column is indexed. This reduces the effort required to index the remaining portions. Index refinement occurs iteratively on the partially indexed intervals of the cracker index. This technique has an advantage over creating indices *a priori* for less selective queries up to the break-even point for simple sorting. It is observed that the initial DB Cracking algorithm is not robust against all types of workloads, in particular, sequential workloads [80].

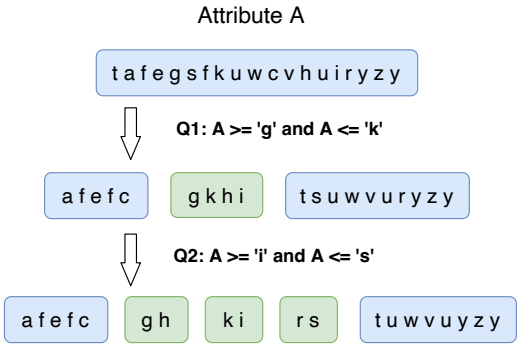

Figure 8.2: Database Cracking. The unsorted data is initially trisected by the first query at 'g' and 'k'. The second query then fine-tuned the interval from 'i' to 's' by overlapping with the initial query.

8.3.1 Stochastic DB Cracking

Stochastic DB Cracking [84], one of the improved variants of DB Cracking, indexes the cracker column in two phases. In the initial data-driven phase, the algorithm leverages the cracker column's central data point (or a random point close to the center) as a pivot element until the resulting subsets have reached a suitable reduction scale. Cracking twice in the data-driven phase can further split the data into quartiles (called DD2C, and when a random point pivot is used, the variation is called DD2R) as shown in Figure 8.3. The original DB Cracking algorithm is applied to these cracked subsets in the second query-driven phase. In contrast to initial DB Cracking, Stochastic DB Cracking demonstrates robustness in managing diverse workloads and resilience in handling various queries for the underlying data distributions.

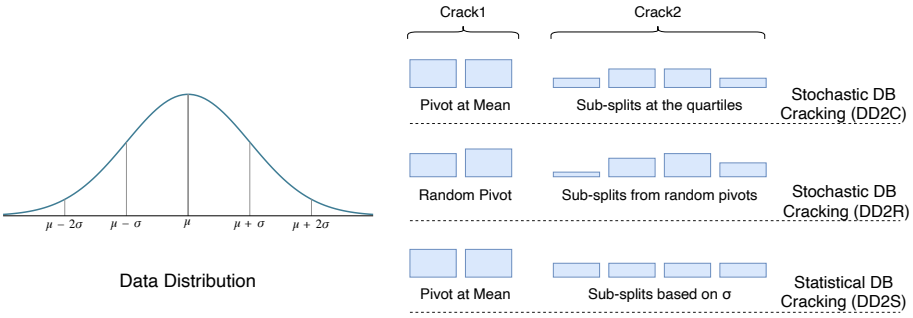

Figure 8.3: Comparison of data-driven phases across DB Cracking variants.

8.3.2 Index Refinement

The stages of workload-driven index creation and targeted refinement for an attribute when using DB Cracking methods can be represented as a state machine, as shown in Figure 8.4. The state machine sets the action rules for the

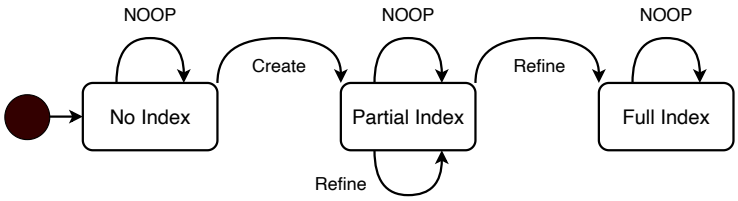

Figure 8.4: Illustration of the state machine for index management. The states represent the status of an index for an attribute. The outgoing edges depict possible actions from the respective states.

index manager in the two-player game. Each state denotes the current status of the index for an attribute, and the outgoing edges represent the possible actions allowed at respective stages. For example, from "No Index", the index manager can either create a "Partial Index" or perform no action and return to "No Index". The decision to create a "Partial Index" or perform no action can be based on the outcome of the heuristic function, thus preventing the DB

Cracking algorithm from possibly refining the index for each executed query. The action *NOOP* is possible in all states. This action is feasible regardless of the attribute and status of an index, and it does not change the status. *Refine* action in the "Partial Index" state invokes the cracking algorithm to split the index of an attribute based on the workload intervals. It can be repeated for each incoming query interval until the index reaches a "Full Index" state, i.e., the interval limits are identical for each index interval; otherwise, the status continues to be "Partial Index"—the other action *Create* is self-explanatory.

While the index manager decides on the action to perform, the underlying cracking algorithm still determines the split interval ranges as per the query predicate intervals. If a specific interval range for an attribute is already split, the cracking algorithm does not split further, resulting in a "win" for the index manager. However, we focus on maximizing the wins for the scenarios that might otherwise result in a loss. Hence, selecting a cracking algorithm that is fair in refinement is imperative. Although Stochastic DB Cracking is robust, the limitations are evident when considering data distribution, such as normal distribution. In the data-driven phase, the pivot point is the mean or a frequently occurring element near or equal to the mean. Consequently, interval boundaries shrink towards the mean, increasing the likelihood of further expensive data partitions at the distribution's tails while decreasing near the mean due to narrower partitions.

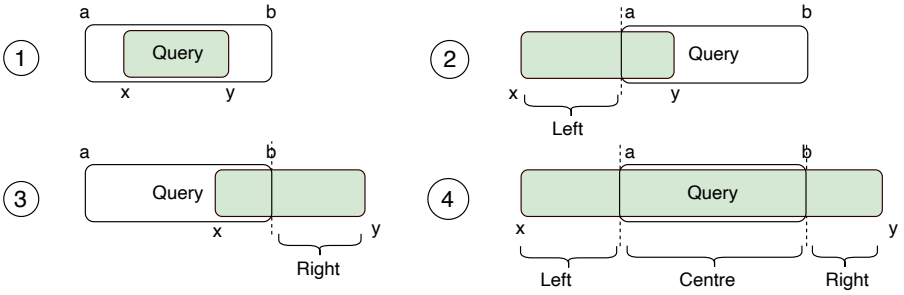

Figure 8.5: Illustration of the $CrackInThree()$ algorithm, dividing a given interval $[x, y]$ (coloured) into new intervals — none (1), two (2)(3), or three (4) — to the left, middle, and right, determined by the query interval limits $[a, b]$. The case where a given interval does not intersect the query's interval limits is ignored.

8.3.3 Statistical DB Cracking

To eliminate the bias of frequently occurring elements and enable fair refinement, we introduce Statistical DB Cracking. This method also employs a two-stage index refinement process: a data-driven phase and a query-driven phase. The statistical DB Cracking algorithm also considers the mean as the pivot. However, double cracking in the data-driven phase (DD2S) leads to splitting the data using the standard deviation, considering all data points and measuring the spread. Instead of partitioning the data exclusively at the respective mean value, this approach uses multiples of the standard deviation around the expected value as a pivot element for partitioning, as given in the Algorithm 4.

The advantage here lies in the interval limits, though not the absolute number of values within the intervals of the same size. Consequently, both rarely occurring and frequently occurring values are deterministically treated equally in the indexing, unlike in DD2R, which uses workarounds involving random pivots. The underlying $CrackInThree()$ algorithm is depicted in Figure 8.5. It should be noted that the second quartile is equal to the mean μ, so any difference becomes apparent only after the secondary refinement in the data-driven phase as standard deviations and quartiles progressively diverge. This divergence naturally concludes when the interval's number of values becomes sufficiently small, minimizing the distinction between random, center, or standard deviation pivot selections.

Data: *attr*, a, b
Result: Cracked index

```
1  function createOrRefine(attr, a, b):
2      if no index exists for attr then
3          index ← crack.create() ;
4          low ← lowest value of attr ;
5          high ← highest value of attr ;
6          μ ← mean of attr ;
7          σ ← standard deviation of attr ;
8          index.insert(low, high, attr) ;
9          crackInThree(index, attr, low , μ) ;
10         crackInThree(index, attr, μ − σ, μ + σ) ;
11         crackInThree(index, attr, a, b) ;
12     end
13     else
14         crackInThree(index, attr, a, b) ;
15     end
```

Algorithm 4: Statistical DB Cracking

8.4 Background

The two-player game presents a well-known problem, and various algorithms have been developed to find optimal actions for zero-sum games [92, 93]. Here, we outline key algorithms to optimize outcomes under unfavorable conditions.

The *Minimax* algorithm employs a tree data structure in which each level takes turns representing the players' moves and the game's status. The leaf nodes contain the results of an evaluation/utility function, which determines the outcome of the sequence of actions from the leaf to the root node. The algorithm's depth-first search selects the minimum or maximum of the node values alternately for each level, making it a powerful game decision-making tool. Traditionally, one player aims to maximize the value, while the other seeks to minimize it. However, the complexity increases when switching between Min and Max players, requiring separate processing. A simplified approach called *Negamax* avoids this distinction. It focuses on determining the maximum value at each step and negates the result when transitioning to a higher level, offering a more

streamlined and efficient alternative.

Expectiminimax introduces chance nodes to assess the expected value of random events such as a dice roll. By integrating these chance nodes with the min and max nodes, Expectiminimax offers a more comprehensive calculation of the weighted average based on the probability of reaching the child node rather than relying solely on utility values.

The Minimax and Negamax algorithms traverse the entire tree, ensuring that even the calculation of certain branches not affecting the result at the root node is considered. During the depth-first search, using α and β values optimize the process, indicating the minimum value one player will reach (α) and the maximum value the other will reach (β), respectively. An α-cut occurs if the α value of a minimizing node is greater than the current value, and a β-cut occurs if the β value of a maximizing node is less than the current value. This *Alpha-Beta Pruning* mechanism prunes unnecessary branches efficiently [93].

8.5 Approach

Our index management model can be viewed as a two-player game involving a tree data structure that allows the index manager to anticipate future query attributes and make decisions based on possible actions. When a query attribute is received, the index manager creates individual tree data structures for each potential future action, depending on the current index status. The depth of these trees determines the maximum runtime of the algorithm and its ability to predict future queries. For example, a two-level tree can approximate the next query. In contrast, a ten-level tree can anticipate up to the fifth query, though with decreased precision and increased computational complexity. The model considers all attributes and their respective index states in each round of play to determine the best possible action. The number of these tree structures corresponds to the number of attributes, their current index states, and potential future actions.

For example, in the scenario shown in Figure 8.6, when a new query attribute a is encountered, and the attribute b already has an index, the model constructs tree structures for possible actions "NOOP," "Create" for a and "Refine" for b. The trees are expanded for both seen and unseen attributes, with each level representing possible future actions for the following query. The tree edges are weighted using a heuristic function, $P[X = a_i]$, which indicates the probability of encountering an attribute a_i in the immediate future. The index manager then utilizes the Expectiminimax algorithm, guided by a utility function, to identify the optimal action by selecting the tree with the highest valuation at the root node and executing it. In the event of a tie, the index manager disregards "NOOP" and performs one of the other available actions.

8.5.1 Heuristics

The selection of an arbitrary attribute a_i is based on probabilities commonly referred to as move by nature. The probability distribution $P[X = a_i]$ for the actions of one of the players can be established by using simple probabilistic

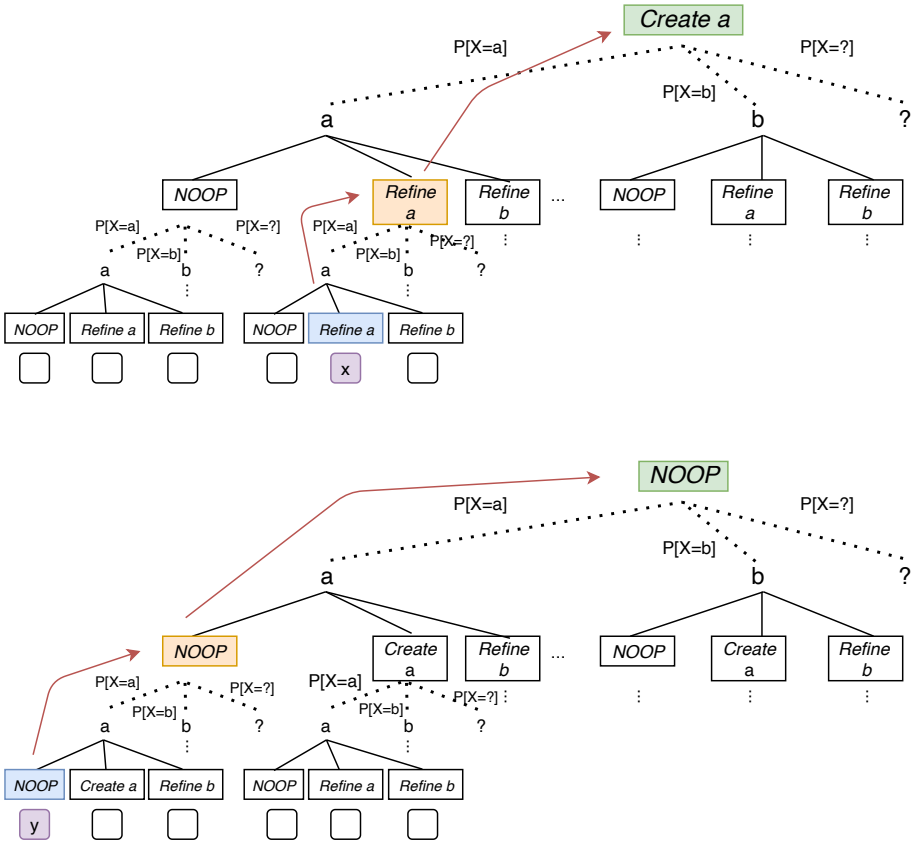

Figure 8.6: Partial Expectiminimax tree data structures for the actions "NOOP" and "Create a". The dotted lines represent the weighted edges indicating the probability of encountering an attribute in the immediate next round of the game. Solid lines indicate possible future actions. The selected leaf nodes and the weighted paths to the root are indicated. Note that attribute "b" has already been seen with an existing index, but the "Refine b" tree is not included.

functions to complex machine learning models that match the application requirements. Here, we use a simple histogram H to estimate the probability of occurrence of already seen and unseen attributes *a posteriori*. For an attribute a_i, it can be approximated as

$$1 = \underbrace{\sum_{a_i \in A} P[X = a_i]}_{seen} + \underbrace{\left(1 - \sum_{a_i \in A} P[X = a_i]\right)}_{unseen} \tag{8.1}$$

$$= \sum_{a_i \in A} \frac{n_i}{n} + \left(1 - \sum_{a_i \in A} \frac{n_i}{n}\right) \tag{8.2}$$

where $n_i = h_i - 1$, $n = \sum_{h_i \in H}(h_i)$ and h_i is the frequency of the attribute a_i in the histogram H.
Example: Before the first query, i.e. $A = H = \emptyset$, the probability of an unknown attribute is $1 - \sum_{a_i \in A} \frac{n_i}{n} = 1 - 0 = 1$. After two occurrences of attribute a_1 and one occurrence of attribute a_2, i.e. $A = \{a_1, a_2\}$ and $H = \{(a_1, 2), (a_2, 1)\}$, the probabilities that these will occur in the future are $P[X = a_1] = \frac{1}{3}$, $P[X = a_2] = 0$ and $1 - \sum_{a_i \in A} \frac{n_i}{n} = \frac{2}{3}$. Frequent attributes that occur more than once are thus weighted higher, and the probability of new attributes are directly related to the occurrences of other attributes in the past.

8.5.2 Utility Function

For a given tree depth d, the utility function is applied to each leaf node of the action paths, which is then utilized in the Expectiminimax algorithm for decision-making. The utility function calculates the potential future outcomes (hits or misses) for each action path in the tree data structures. Since we are using Expectiminimax, based on the Negamax algorithm, the distinction between minimum and maximum results is no longer relevant. Therefore, the final result is considered positive for both players. To this end, we introduce a simple utility function (8.3) and its more refined version (8.4). Player A, the index manager, aims to maximize hits. Player B, an arbitrary query workload, seeks to maximize misses. Let's assume that $hasWon(p)$ indicates the victory of player p. A possible utility function can be represented in the form of

$$f(p) = \begin{cases} h - m & p = A \wedge hasWon(A) \\ m - h & p = B \wedge hasWon(B) \\ -1 \cdot (m - h) & p = A \wedge hasWon(B) \\ -1 \cdot (h - m) & p = B \wedge hasWon(A) \\ 0 & Tie \end{cases} \tag{8.3}$$

h for the number of hits and m for the number of misses, where the gains for one player come at the expense of the other. As depicted in Figure 8.6, the values x and y of the utility function at the leaf nodes, multiplied by the weighted probabilities of encountering an attribute from the edges, determine the highest valued path to the root. However, this naive function fails to consider the costs involved in the creation and target refinement of an index using a DB Cracking method.

As Statistical DB Cracking achieves even splits of interval boundaries, leading to lower costs for subsequent splits and convergence in the benefit of an index, a fair utility function in the form of

$$f(p) = \begin{cases} h' - (m + o') & p = A \wedge hasWon(A) \\ (m + o') - h' & p = B \wedge hasWon(B) \\ -1 \cdot ((m + o') - h') & p = A \wedge hasWon(B) \\ -1 \cdot (h' - (m + o')) & p = B \wedge hasWon(A) \\ 0 & Tie \end{cases} \tag{8.4}$$

can be defined whereby

$$h' = \begin{cases} min(1 + log_b(b), 1 + log_b(h)) & h \geq 1 \\ 0 & h = 0 \end{cases} \tag{8.5}$$

and

$$o' = \begin{cases} max(0, 1 + log_b(\frac{1}{o})) & o \geq 1 \\ 0 & o = 0 \end{cases} \tag{8.6}$$

and o is the number of operations. A higher base b provides a more nuanced gradation through weighting operations and the hits considered in the utility function. For creating a full index with Statistical DB Cracking on a dataset of an attribute with n different values, an optimal scenario would require $log_2(n)$ refinements. Since refinement is no longer possible from the "Full Index" status, and no costs are incurred after that, the base b can be assumed as $log_2(n)$. Considering a relation with 10^6 distinct values for an attribute. In the optimal scenario, Statistical DB Cracking would require approximately $log_2(10^6) \approx 19.93$ refinements to achieve a full index. This implies that no further costs are anticipated beyond this point, a factor considered in the cost function with $b = 19.93$ and $o'(19.93) = 0$.

8.5.3 Pruning

Pruning in Expectiminimax, akin to Alpha-Beta Pruning, is challenging due to the weighted sum in the chance nodes. A workaround is to place bounds on the possible values of the utility function, which also bounds the value of leaf nodes [94]. As a result, an upper bound on the value of a chance node can be established without examining all its children. Additionally, for the seen attributes, the index manager updates the tree and recomputes the weighted probabilities of the edges, eliminating the need to re-create the trees for each incoming query.

8.6 Index Structure

A flexible data structure is essential to creating and refining indices. In DB Cracking, two primary data structures are commonly used: one holds the data (e.g., cracker column), and the other holds metadata about the ranks of the targeted data sections (e.g., cracker index as an AVL-Tree). This combined data structure requires additional work because the refinement needs to be done

on the data itself and the metadata about the ranges of the refinement. To simplify this process, we utilize Interval Trees (I-Trees) stored in memory using a key-value database cluster, such as Redis, which can naturally maintain the metadata as part of the data structure [95].

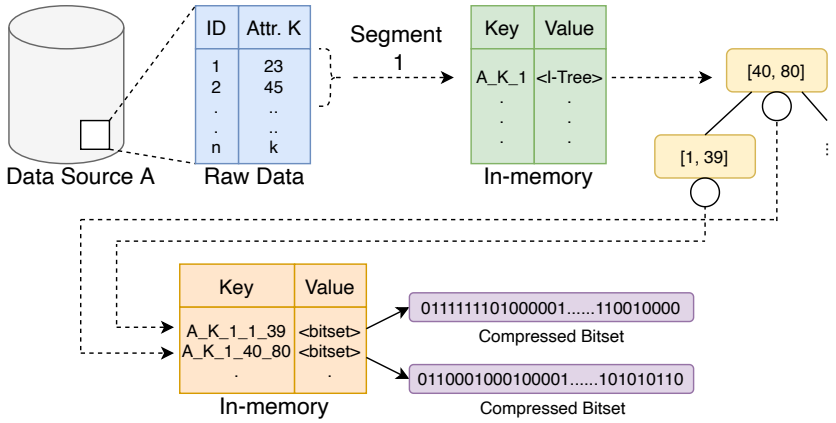

Figure 8.7: Example illustrating the origin of raw data from a DataSource A, and the structure of the I-Tree index located in-memory key-value store.

Figure 8.7 shows the architecture where the data is segmented into sizes of n to fit into the main memory. One I-Tree is created for each data segment of an attribute and acts as the value of the key-value entry. The I-Tree data structure is implemented as a Redis module supporting Roaring Bitmap Compression [72, 96]. It consists of nodes with non-overlapping data intervals after each targeted refinement step, with inclusive interval boundaries. The tuple identifier (ID) of the data per segment is not stored directly as a value in the node. Instead, a pointer to a compressed raw bitmap is stored, indicating the tuple's position per segment.

8.7 Experiments

We conduct two sets of experiments. First, we compare Statistical DB Cracking with other cracking variants to assess data insight time, robustness, and convergence. Second, we evaluate the Minimax model for index management and compare it against using the direct DB Cracking algorithms without a model. We focus on average query response time, index refinement time, and index hit ratio for different data distributions and query workloads. All the experiment scenarios tested assume an environment with zero workload knowledge.

Setup: All experiments were performed on a server running Rocky Linux 9.2 (Blue Onyx). The server has 256GB of RAM and an AMD EPYC 7742 64-core processor. Computational tasks were limited to 8 cores. Following the approach established in prior studies [80, 81, 83, 84], we used a synthetically generated dataset with 10^6 entries and four data columns: *tid*, $\mathcal{U}$, $\mathcal{N}$, and $\mathcal{Z}$. In this dataset, *tid* represents the unique row number, $\mathcal{U}$ follows a uniform distribution in the range $[0, 10^6)$, $\mathcal{N}$ adheres to a normal distribution in the range $[0, 10^6)$

with $\mu = \frac{10^6}{2}$ and $\sigma = \frac{10^6}{\sqrt{12}}$, and $\mathcal{Z}$ is subject to a Zipf distribution in the range $[0, 10^6)$ with a shape parameter $\alpha = 0.6$. The generated I-tree indices are stored in a Redis Cluster version 7.0.4, configured with a 3-Master-3-Replica setup.

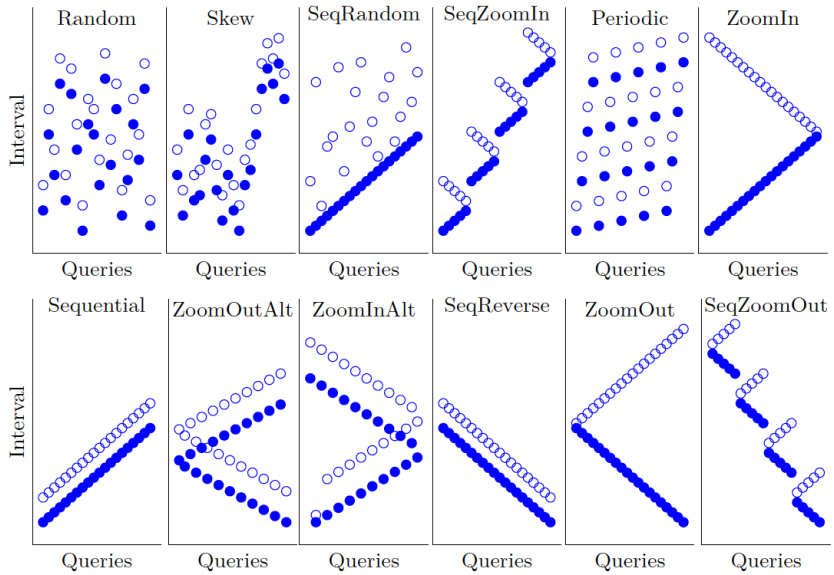

Figure 8.8: Different query patterns of input queries.

8.7.1 Statistical vs. Stochastic DB Cracking

We compare Statistical DB Cracking (DD2S) to other variants where double cracking is performed in the data-driven phase, namely, Stochastic DB Cracking with double central pivot (DD2C), Stochastic DB Cracking with double random pivot (DD2R), and Scan as the baseline. Our measurements focus on both single query and accumulated query response times. Each test run involves 10^3 queries that are structured as:

```
SELECT tid FROM syntheticData
    WHERE x BETWEEN y AND z;
```

where $x \in \{\mathcal{U}, \mathcal{N}, \mathcal{Z}\}$ and $y, z \in [0, 10^6)$, ensuring $|y - z| = 0.01 \cdot 10^6$ to achieve an optimal selectivity of 0.01. This selectivity value has been commonly used in prior studies [80,81] for comparing various cracking alternatives. Different query patterns are generated as shown in Figure 8.8 to test each cracking alternative, allowing for a systematic analysis using a random sequence and, alternatively, a realistic analysis based on typical search patterns of users, such as jump or zoom patterns. For brevity, we discuss specific insights, including:

Data-Insight Time

As illustrated in Figure 8.9, Statistical DB Cracking reduces the data insight time and associated query response time. It exhibits similar, if not superior,

performance to Stochastic DB Cracking variants across the workloads and data distributions.

Overhead The single query response time gradually increased with each index refinement. The degradation becomes noticeable around the 100^{th} query, leading to an intersection of the response times of DB Cracking, DD2C, DD2R, and DD2S with the time required for a scan. As depicted in Figure 8.9, this can be attributed to the overhead generated by the growing number of intervals in the index.

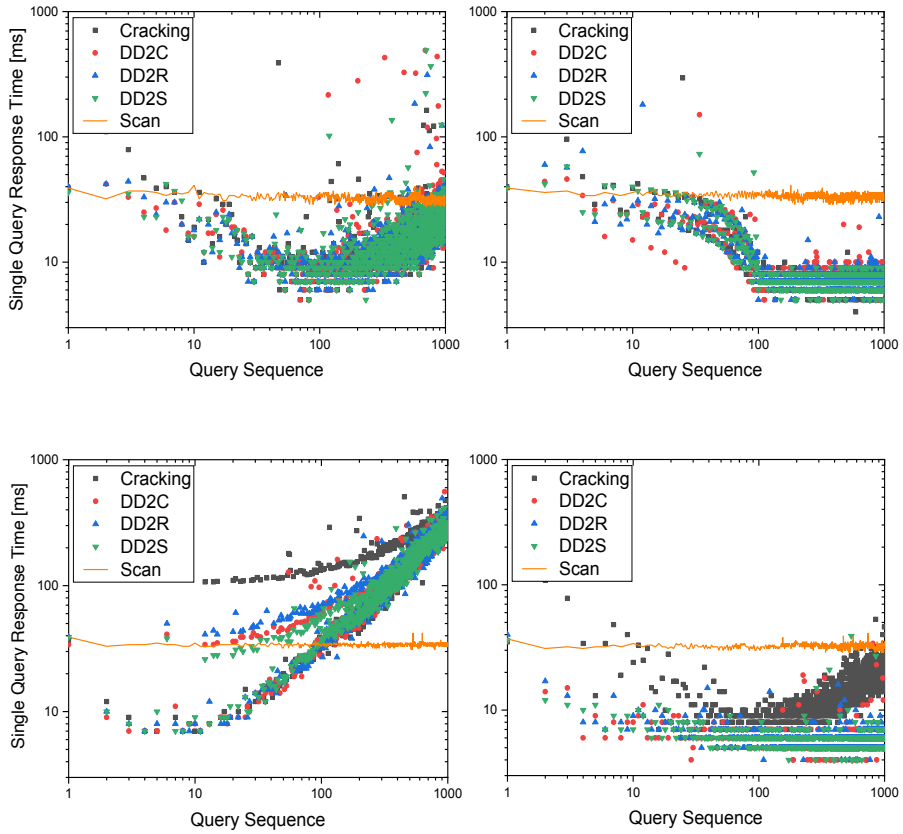

Figure 8.9: The Single Query Response Time of random, periodic, and sequential query workloads on $\mathcal{N}$ respectively (in the respective order from left). The rightmost graph is the Single Query Response Time of random workload on $\mathcal{U}$.

Sequential Workload As shown in Figure 8.9 and Figure 8.10, query workloads that exhibit a sequential order present a worst-case scenario for DB Cracking and its derivatives DD2C, DD2R, and DD2S. A similar pattern is observed in earlier studies as well [84]. This unfavorable situation arises from the limited number of values sorted per each refinement of an interval. In the optimal case, an interval is halved. However, only a handful of elements are sorted in

sequential workload. The fact that all derivatives DD2C, DD2R, and DD2S also encounter this worst-case scenario stems from their reliance on a two-fold data-driven refinement followed by exclusive query-driven actions, which can only be alleviated by increasing the data-driven proportion (i.e., DDxC, DDxR, and DDxS with $x \gg 2$).

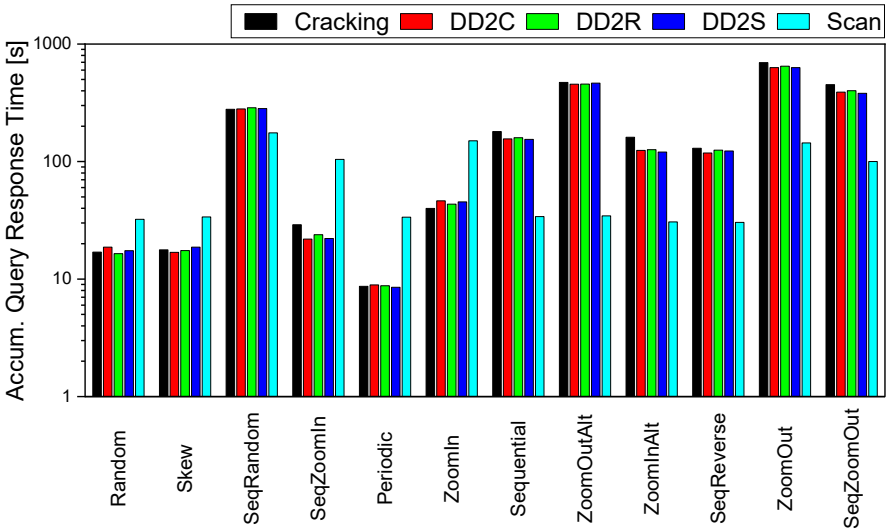

Figure 8.10: Accumulated Query Response Time of all workloads on $\mathcal{N}$.

Periodic Workload In contrast to other workloads, periodic workload gradually improved the single query response time, as shown in Figure 8.9, resulting in comparable times to scan, DB Cracking, and other variants. The optimal performance is achieved after approximately the 100th query, beyond which there is no further improvement. This validates the hypothesis that, from this point onward, the overhead surpasses the number of intervals, thereby offsetting any improvement in the response time for the periodic workload.

Robustness

In query processing, robustness pertains to the consistent and prompt response across diverse workloads. As shown in earlier experiments, the DB Cracking algorithms lack robustness against sequential workloads. After the data-driven phase (double cracking), these algorithms start executing query-driven cracks, reverting to initial DB Cracking. Improving robustness typically involves fine-tuning the balance between the data-driven and the query-driven phases as highlighted in [84]. It is important to acknowledge that improved robustness comes at the cost of data insight time. In Figure 8.11, DDC, DDR, and DDS are compared against each other and benchmarked against DB Cracking and Scan without the overhead to facilitate a direct comparison of the pure algorithms for the sequential workload. DB Cracking is closer to the scan as the number of executed queries increases. As expected, DDC, DDR, and DDS work robustly, even in this worst-case scenario. The cumulative query response times in Figure 8.11 also show that DDS initially has a lower response time than the

other derivatives.

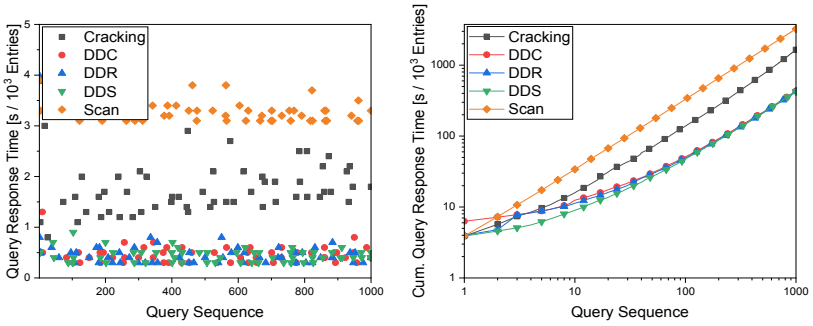

Figure 8.11: The Query Response Time (left) and Cumulative Query Response Time (right) of sequential workload on $\mathcal{U}$ without the overhead.

Convergence

On datasets with a uniform ($\mathcal{U}$) and normal distribution ($\mathcal{N}$), no significant differences in Convergence are observed among all DB Cracking variants as shown in Table 8.2. It is generally low, with a maximum value of 0.21% for $\mathcal{U}$ and 3.21% for $\mathcal{N}$. This is attributed to the high number of unique values in the ground truth of datasets $\mathcal{U}$ and $\mathcal{N}$. In the case of the uniform dataset $\mathcal{U}$, which has 10^6 different values, a complete index would require 10^6 intervals. Thus, a unique dataset presents the worst-case scenario. All algorithms reach the maximum value of 99.83% on $\mathcal{Z}$ with sequential workloads because the early query intervals extensively cover the dataset range.

8.7.2 Minimax Model

In this section, we evaluate the effectiveness of using a two-player game model for index management. Throughout the experiments, we chose Statistical DB Cracking and associated utility function with the Minimax model. We divide the evaluation into two parts based on the type of workload.

Random Query Intervals In the first part, we vary the distribution of the query attributes and use random interval ranges to generate a unique set of queries in each workload. We generate workloads of 1000 queries each in the form of

```
SELECT tid FROM syntheticData
    WHERE f(x) BETWEEN y AND z;
```

where $x \in [0, 10^3)$, $y, z \in [0, 10^6)$, ensuring $|y - z| = 0.01 \cdot 10^6$ and the variable attribute x is assigned at runtime to $\mathcal{U}, \mathcal{N}, \mathcal{Z}$ by the function $f : [0, 10^3) \rightarrow \{\mathcal{U}, \mathcal{N}, \mathcal{Z}\}$. The variations in the query attribute x include: *1)* using a single attribute, called "Unique" *2)* distinct attributes from the range $[0, 10^3)$, called "Uniform" *3)* attributes with a skewed distribution in a 20/80 ratio, where the

Table 8.2: Convergence measured for different DB Cracking variants across different workloads and data distributions.

Algorithm	**Workload**	**Convergence (%)**		
		$\mathcal{U}$	$\mathcal{N}$	$\mathcal{Z}$
Cracking	Random	0,20	3,21	1,33
DD2C	Random	0,21	3,21	3,18
DD2R	Random	0,21	3,21	4,05
DD2S	Random	0,21	3,21	3,18
Cracking	Skew	0,20	3,19	1,50
DD2C	Skew	0,21	3,19	3,36
DD2R	Skew	0,21	3,19	4,17
DD2S	Skew	0,21	3,19	3,36
Cracking	SeqRandom	0,20	1,66	99,83
DD2C	SeqRandom	0,21	1,67	99,83
DD2R	SeqRandom	0,21	1,67	99,83
DD2S	SeqRandom	0,21	1,67	99,83
Cracking	SeqZoomIn	0,20	0,14	99,83
DD2C	SeqZoomIn	0,21	0,14	99,83
DD2R	SeqZoomIn	0,21	0,14	99,83
DD2S	SeqZoomIn	0,21	0,14	99,83
Cracking	Periodic	0,02	0,18	0,93
DD2C	Periodic	0,03	0,18	2,78
DD2R	Periodic	0,03	0,18	3,70
DD2S	Periodic	0,03	0,18	2,78
Cracking	ZoomIn	0,20	0,27	0,93
DD2C	ZoomIn	0,21	0,28	2,78
DD2R	ZoomIn	0,21	0,28	3,59
DD2S	ZoomIn	0,21	0,28	2,78
Cracking	Sequential	0,20	0,07	99,83
DD2C	Sequential	0,21	0,08	99,83
DD2R	Sequential	0,21	0,08	99,83
DD2S	Sequential	0,21	0,08	99,83
Cracking	ZoomOutAlt	0,20	0,58	0,93
DD2C	ZoomOutAlt	0,21	0,58	2,78
DD2R	ZoomOutAlt	0,21	0,58	3,59
DD2S	ZoomOutAlt	0,21	0,58	2,78
Cracking	ZoomInAlt	0,20	0,14	65,39
DD2C	ZoomInAlt	0,21	0,15	66,32
DD2R	ZoomInAlt	0,21	0,15	66,20
DD2S	ZoomInAlt	0,21	0,15	65,86
Cracking	SeqReverse	0,20	0,07	99,83
DD2C	SeqReverse	0,21	0,08	99,83
DD2R	SeqReverse	0,21	0,08	99,83
DD2S	SeqReverse	0,21	0,08	99,83
Cracking	ZoomOut	0,20	0,27	0,93
DD2C	ZoomOut	0,21	0,28	2,78
DD2R	ZoomOut	0,21	0,28	3,53
DD2S	ZoomOut	0,21	0,28	2,78
Cracking	SeqZoomOut	0,20	0,14	99,83
DD2C	SeqZoomOut	0,21	0,14	99,83
DD2R	SeqZoomOut	0,21	0,14	99,83
DD2S	SeqZoomOut	0,21	0,14	99,83

first 20 percent correspond to a single attribute, and the remaining 80 percent are random attributes selected from the range $[0, 99]$, called "Skew" *4)* random attributes from the interval $[0, 99]$, called "Random" and *5)* periodic attributes from the interval $[0, 99]$, called "Period".

We compare the Minimax model against *1)* a "random model" where the decision to apply the Statistical DB cracking method (DD2S) is decided by a coin toss *2)* a "no model" scenario where just Statistical DB Cracking (DD2S) is used after each query *3)* scan as the "baseline" and *4)* using complete, a priori generated index, the "optimal" scenario.

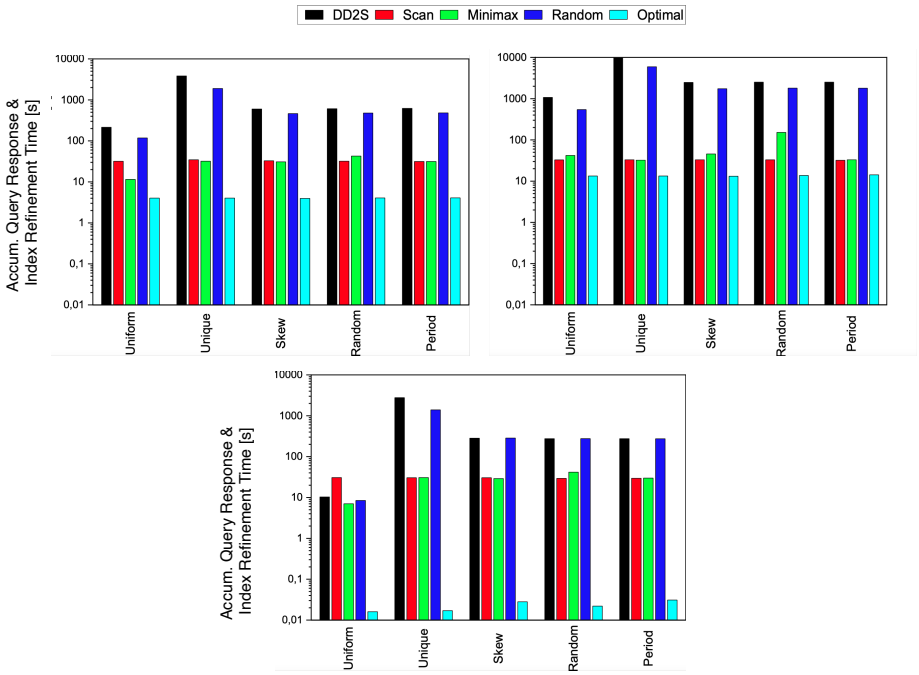

Figure 8.12: Accumulated query response and index refinement time for different workloads on $\mathcal{U}$, $\mathcal{N}$, $\mathcal{Z}$ respectively (in the respective order from the left).

Accumulated query response and index refinement times are shown in Figure 8.12. The total time costs using the Minimax approach are, on average, comparable to those of a scan. The convergence rate is expected to be low because of the distinct interval ranges in the workload. However, the pure query response time of the Minimax approach is generally more favorable than that of a scan as the model continues to adapt to the workload. On the other hand, other approaches resulted in higher costs due to expensive refinement. It is important to note that fewer refinements lead to reduced overhead, and hence, the Minimax approach is more favorable than the pure DD2S and random approaches. Despite DD2S requiring twice as many refinements, the total time for the random approach are relatively similar, with only a slight improvement in query response time for DD2S. This is due to factors mentioned earlier: the number of elements to be sorted decreases with each refinement, at the expense of increased overhead for targeted refinement for each new interval. The total

time comparison on *ps_availqty* from *partsupp* relation with $8 \cdot 10^5$ tuples of the TPC-H dataset also showed similar trends as shown in Figure 8.13.

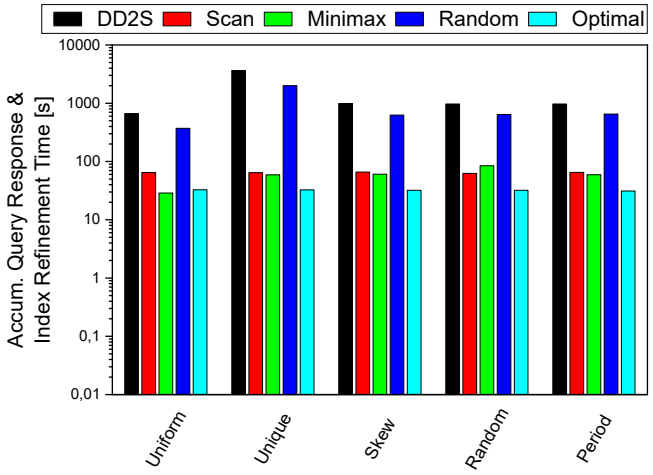

Figure 8.13: Total time comparison on *ps_availqty* from *partsupp* relation with $8 \cdot 10^5$ tuples of the TPC-H dataset.

Fixed Query Intervals To assess how quickly the Minimax model can adapt to varying workloads, we now keep the query interval size and the number of distinct intervals fixed while changing the distribution of the query attributes in this second part of our evaluation. This allows us to observe the model's convergence rate towards the baseline. We ensure the repetition of queries in the workload by keeping the number of intervals fixed and increasing the number of queries to 10^4. Additionally, we measure the query-hit ratio to demonstrate the effectiveness of the Expectiminimax algorithm, which aims to maximize the model's success rate. A "hit" for the query occurs when no partial or complete index is available or the index manager notifies the Statistical DB Cracking algorithm to perform a targeted index refinement, i.e., a new cracking of the index for the given query interval.

Based on the hit ratios depicted in Figure 8.14, it is evident that the Minimax model quickly adjusts to workloads where the query attributes follow a normal distribution. The total time cost per query execution converges more rapidly toward the baseline and remains lower than using only the Statistical DB Cracking method without the model. However, when the query attributes are uniformly distributed in the workload, the adaptation is relatively slower, while the convergence of total time costs per query execution remains similar. This observation arises from the worst-case scenario inherent in a two-player game, where unrestricted index refinement is not permitted. Despite the refinement costs theoretically decreasing with each iteration as the number of elements to be sorted decreases, these constraints persist. With a larger query interval size of 10^4, the hit ratio for DD2S improves. However, the Minimax model converges rapidly compared to DD2S, as shown in Figure 8.15. Subsequent experiments where the base b of the Minimax model's utility function varied for the hits and misses did not reveal any significant differences.

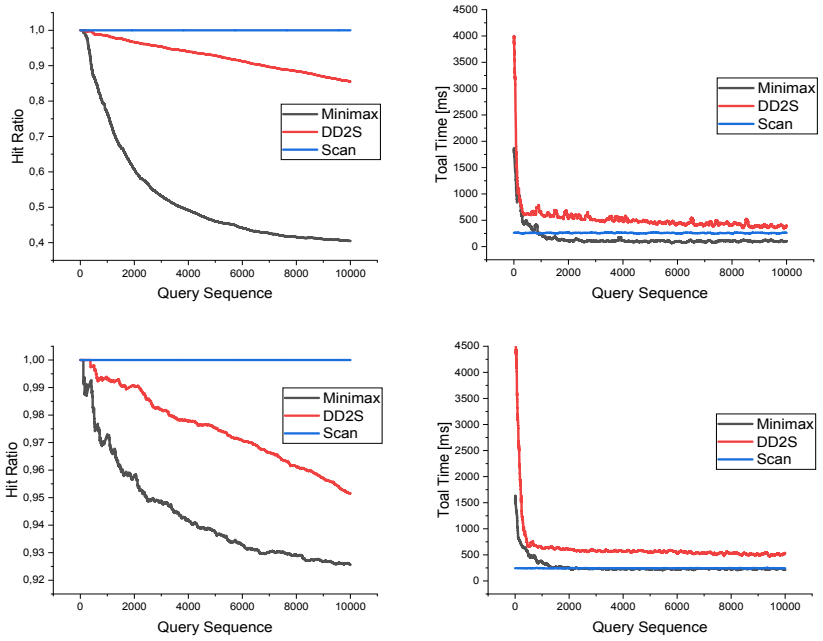

Figure 8.14: Hit ratios and the moving average of total time with fixed interval size 10^3 for query workloads on $\mathcal{N}$. The top row charts are for the workload with normally distributed attributes, and the bottom row is for the workload with a uniform distribution of attributes in the query workload.

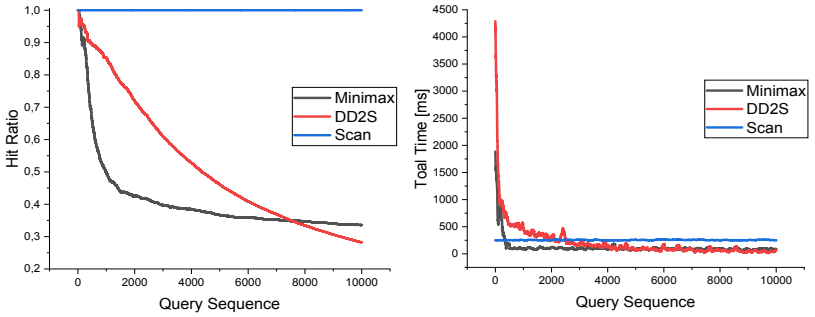

Figure 8.15: Hit ratios and the moving average of total time with fixed interval size 10^4 for query workloads on $\mathcal{N}$.

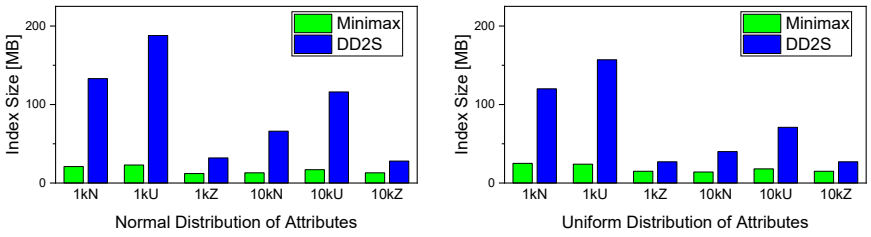

Figure 8.16: Comparison of storage space usage between Minimax and DD2S for different interval sizes and data distributions.

Lastly, we compare the storage space usage in Figure 8.16. The cost benefits of using the Minimax model concerning storage space utilization outperformed the Statistical DB Cracking algorithm without the model across workloads and data distributions. In particular, the space savings are several-fold higher for the highly selective queries.

8.8 Summary

This chapter introduces an innovative adaptive index management model based on DB Cracking methods, treating it as a two-player zero-sum game. The model addresses two critical limitations of DB Cracking variants: the autonomous selection of indexing attributes and high cracking costs. Additionally, we propose a new DB Cracking variant, Statistical DB Cracking, to complement the two-player model, ensuring fair index refinement by mitigating bias toward frequently occurring elements in the data. The experiments conclusively demonstrated that statistical DB cracking exhibits similar, if not superior, performance to other DB cracking variants regarding data-insight time, robustness, and convergence. Moreover, the model utilizing the Expectiminimax algorithm proved significantly more effective for index management than direct DB Cracking algorithms, displaying robustness by adapting to varying query workloads and data distributions. Notably, this model offered significant cost-effectiveness in terms of storage space utilization. In summary, the two-player model based on adaptive indexing is a viable alternative in environments where time, storage space, and associated costs are significant.

So far, we have been able to independently test the polystore architecture, indexing scheme, and adaptive index management required to build the comprehensive exploration platform. Our next research step is to fine-tune the middleware to adapt query processing and generalize the solution to an existing distributed query processing system.

Chapter 9

Integrated Data Exploration

In PLANTdataHUB, the data storage layer is composed of GitLab instances known as DataHUBs, where the ARCs are maintained. This design choice creates a decentralized storage system, enabling individual research groups to manage their data locally. While integrating the structured metadata from the ARCs across the various DataHUBs is relatively straightforward, maintaining the large volumes of measurement data from all ARCs within a centralized storage platform is more challenging. Therefore, there is a need for tools that allow data analysts to pre-select a list of ARCs, facilitating on-demand access to heterogeneous measurement data with minimal data movement, all while adhering to PLANTdataHUB's security policies. In this chapter, we will:

- Introduce a data exploration tool that enables direct querying of heterogeneous data from ARC datasets.
- Generalize and extend the previously presented indexing scheme and adaptive indexing to off-the-shelf distributed query processing engines, such as Apache Drill. We will outline the necessary middleware extensions for query rewriting and optimization, utilizing global indexes.
- Present the implementation details and describe how the individual components can be integrated with the ARC Metadata Registry. This integration aims to upgrade the existing application into a comprehensive ARC data exploration platform.

To enable integrated data exploration of experimental metadata and substantial amounts of heterogeneous raw data packed into research output units, we often require implementing complex transformation pipelines and complex data-to-query workflows, which can result in potential data loss and significant delays when using traditional databases. In dynamic environments, having quick access to data is crucial, and the ability to scale infrastructure is essential to meet changing workload requirements. Scientists can leverage in-situ query processing [97] for quick and on-demand data analytics. From an infrastructure standpoint, container orchestration helps standardize processes and provides rapid analytics support within scientific environments. It offers a fast data access gateway and ensures easy setup across various infrastructure environments.

A tool designed for on-demand data exploration of ARCs could prove invaluable in rapidly preparing data for analysis using off-the-shelf query processing systems. Such an enhancement would significantly improve data management, especially in the biosciences [98] and other specialized fields, where prompt data analysis and transformation are essential for reducing complexity and extracting meaningful insights. Additionally, such tools integrate well with RDM solutions that feature decentralized data storage layers. In this chapter, we introduce *ARCXplore*, a versatile tool designed to address the challenges of exploring heterogeneous datasets and engineered to enable in-situ data exploration, allowing

scientists to work with ARCs without requiring extensive data transformation pipelines.

9.1 ARCXplore

ARCXplore systematically traverses the hierarchies within the ARCs and identifies the data files (experimental metadata, measurement data, contextual data), software, configuration files, and other relevant files based on predetermined file extensions. An internal ETL process ensures that the data is standardized, loaded into containerized data sources, and ready for querying. ARCXplore has been developed to transform heterogeneous data files within the individual ARCs into an on-demand multi-model warehouse where users can directly run queries using a distributed query processing engine for integrated data exploration.

ARCXplore employs a set of modules that process either a single or multiple ARC datasets in a sequence. Figure 9.1 illustrates the individual components of ARCXplore, which were developed in Python, utilizing some of its built-in functionalities. It uses Docker and Docker Compose for container orchestration and management.

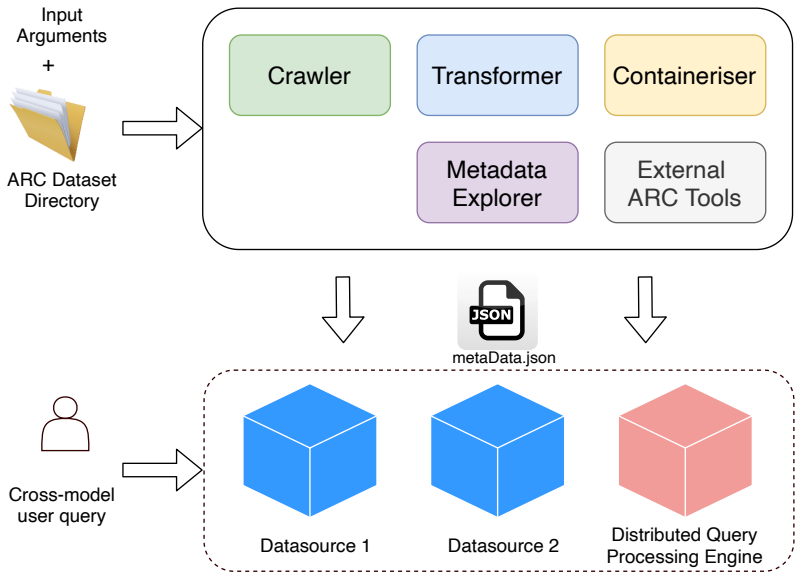

Figure 9.1: Architecture of ARCXplore

9.1.1 Crawler

The primary objective of the Crawler is to systematically traverse the directory structure, examining the directories and subdirectories, and identifying files based on their file extensions. It has several functional sub-components that allow it to efficiently traverse ARC datasets and identify potential target files needed for collection. It plays a crucial role in accurately identifying files,

making it essential for determining the data models and their respective data sources.

The Crawler module parses the input arguments to identify the root folder for each ARC used in ARCXplore's operations. Once initialized, it traverses the directory hierarchies and subdirectories of each ARC dataset, searching for files with predefined extensions.

- *Traversal*: The directory traversal leverages the in-built `os.walk` method in Python to efficiently traverse directories and navigate through the root folder and subsequent subdirectories. Each iteration of `os.walk` allows the Crawler to identify files within its traversal process's current working directory. This, in turn, allows it to selectively collect files that match the provided file extension list.
- *Data models*: The directory traversal takes input parameters to either focus on file types from a particular data model or simultaneously traverse files from multiple data models, aiding in the early discovery of diverse data files within the ARC dataset. The identification of the files is performed through file extension matching, leveraging the `any` functionality in Python [99]. It efficiently iterates through the list of file extensions and returns the exact matches.
- *Rules*: The Crawler module uses configurable exclusion rules to eliminate the unwanted system directories or artifacts in the ARC datasets during the directory traversal. The `os.walk` functionality supports exclusion rules, commonly referred to as *Blacklists*, which specify files and directories that the Crawler should not traverse or consider. This includes the working directories the tool creates or the system-generated temporary files.

9.1.2 Transformer

The Transformer module takes the individual directory paths collected by the Crawler as input and initiates the transformation pipeline. It recognizes predefined file types and their associated data models, such as tabular, document, and key-value formats. The module converts these files into standardized formats suitable for ingestion into databases that support the respective data models.

The primary function of the Transformer module is to identify the file extension and its corresponding data model. It then invokes the appropriate conversion function tailored for each data model. Each file extension is linked to a respective conversion function that addresses the specifics of the underlying data model. For example, when the module encounters files representing tabular data, it invokes the class designed for tabular structure, which converts the data into a standardized format, such as CSV or TSV, that can be directly ingested into a relational database.

Currently, ARCXplore supports three data models: tabular, key-value, and document. Table 9.1 lists the supported data models and the respective default databases we use. This approach aligns with ARCXplore's goal of accommodating multiple data sources for in-situ data exploration.

Data Model	Description	Database
Tabular Data	Organized into rows and columns, resembling spreadsheets (e.g., CSV, Excel). Converted to preserve structure in CSV format.	PostgreSQL
Key-Value Data	Utilizes key-value pairs (e.g., JSON, YAML). Structure preservation for data integrity. Converted to JSON format.	Redis
Document Data	Contains unstructured and semi-structured textual information (e.g., JSON, XML, plain text). Retains text while standardizing structure for a flexible database.	MongoDB

Table 9.1: Data Models and respective databases supported by ARCXplore.

Metadata Explorer

The Metadata Explorer module works in conjunction with the Transformer and Containerizer modules. While transforming individual data files, the Metadata Explorer systematically collects surface-level metadata, including attributes such as file size, creation date, and file type. This metadata is compiled into a standardized JSON artifact called `metaData.json`, providing a comprehensive overview of the data for each ARC dataset. The amount of metadata extracted varies depending on the data model. For specific data models, such as tabular data, the Metadata Explorer can extract additional information, including column names, column data types, and statistical values, such as the minimum and maximum values for each column.

9.1.3 Containerizer

The Containerizer module creates an on-demand data warehouse that houses transformed data files from the ARCs, organized by their respective data models into individual data containers. This module encapsulates and isolates the converted files using Docker containers, which ensures that each data container is portable, reusable, and reproducible. Each container includes a database Docker image suitable for its specific data model. Containerization improves data management by grouping data, dependencies, and configurations into self-contained units [100]. This encapsulation simplifies deployment across different environments and reduces the time required for data preparation.

A key feature of the Containerizer is its ability to load transformed data into containers according to their data models while strictly isolating file types. This isolation is essential for maintaining data integrity and preventing cross-contamination between data models. For instance, tabular data is kept separate from key-value or document data, ensuring that each data model remains distinct. This approach prevents data model-specific databases from interfering with one another, thereby guaranteeing data purity and reliability.

Orchestration

The Containerizer module utilizes scripts to dynamically generate container orchestration artifacts, including Docker Compose files that describe the containerization environment. These scripts are designed to detect new data models and automatically add corresponding containers. They also execute specific import scripts for each database, tailored to every unique data model identified. This process of dynamically generating orchestration artifacts automates the setup of containerized environments, making it highly adaptable to the changing needs of data exploration.

Data Ingestion

Importing data into containers is managed through a series of import scripts, each designed for a specific data model and its corresponding database. Since different databases offer unique methods for ingesting bulk data, these import scripts are crucial for each database type.

For instance, the script for importing tabular data into a relational database, such as PostgreSQL, oversees data ingestion, schema creation, and indexing, ensuring efficient querying of tabular data. In contrast, when dealing with document data, scripts are crafted to handle the ingestion and indexing of semi-structured data within document databases, enabling the storage and retrieval of complex, semi-structured information. Additionally, the scripts are optimized for attribute-value formatted data, enabling efficient storage and retrieval of key-value pairs and hierarchical data structures in key-value databases.

9.1.4 ARC Data Processing

The data flow of ARCXplore, as illustrated in Figure 9.2, begins with the initiation of external ARC-specific tools that validate the ARC. This validation process is crucial for ensuring the integrity and conformity of the research output unit [101]. It guarantees that the individual modules of ARCXplore can rely on a valid base directory structure. After the validation, the crawler component performs its primary function: data discovery and directory traversal. Once the data files are cataloged according to their extensions, a list of file paths is generated, which serves as input for the next module.

The transformer component processes a list of paths and produces transformed data files in their respective working directories for each data model, such as *doc_file , csvs, kv_files*. It also generates a ***metaData.json*** file containing surface-level metadata for the transformed files.

During the containerization stage, the transformed, standardized, and metadata-enriched datasets, along with the associated scripts, are prepared for loading into individual Docker containers. The containerizer then creates an orchestration artifact, which allows for easy bootstrapping of individual containers. Additionally, it includes a distributed query processing engine that enables querying of data from various heterogeneous data sources. Finally, the module imports the transformed data files using specific import scripts to populate the respective databases with the transformed data.

```
# Code for grouping files by data model
```

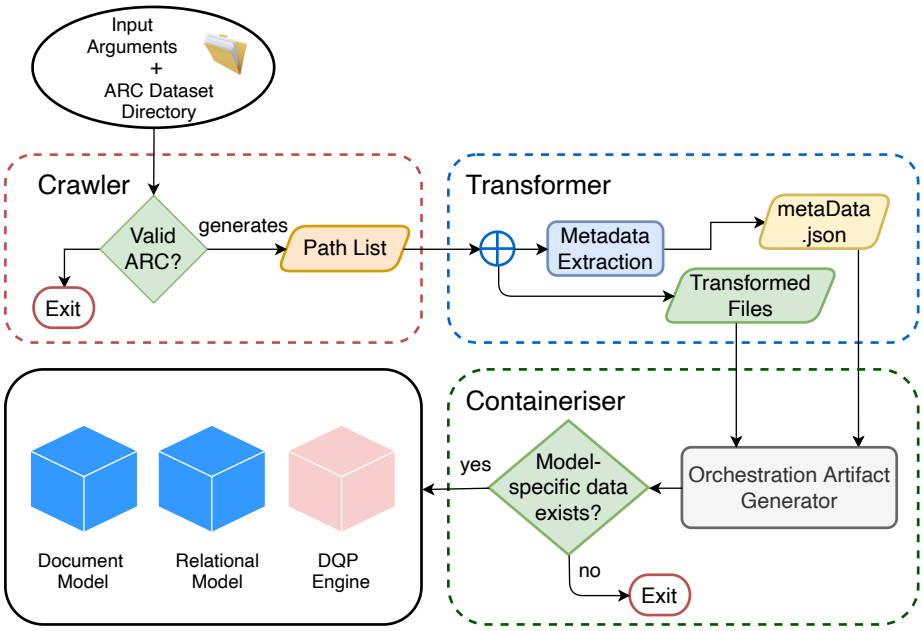

Figure 9.2: The data flow of ARCXplore.

```
def group_files_by_data_model(transformed_files):
    data_model_groups = {}
    for file in transformed_files:
        data_model = extract_data_model(file)
        if data_model not in data_model_groups:
            data_model_groups[data_model] = []
        data_model_groups[data_model].append(file)
    return data_model_groups
```

Listing 9.1: Grouping Files by Data Model

```
# Code for generating Docker Compose configurations
def generate_docker_compose(data_model_groups):
    docker_compose = "version: '3'\n\nservices:\n"

    for data_model, files in data_model_groups.items():
        if data_model == "tabular":
            docker_compose += generate_tabular_docker_compose(files
                )
        elif data_model == "graph":
            docker_compose += generate_graph_docker_compose(files)
        elif data_model == "keyvalue":
            docker_compose += generate_keyvalue_docker_compose(
                files)
        elif data_model == "document":
            docker_compose += generate_document_docker_compose(
                files)
    docker_compose += generate_drill_docker_compose()
    # Additional Docker Compose configurations can be added here
    return docker_compose
```

Listing 9.2: Generating Docker Compose Configurations

9.1.5 Input Parameters

ARCXplore can be configured to process files from the ARCs that follow a single data model or diverse data models simultaneously. By parameterizing the data model type, ARCXplore automatically determines the appropriate database, import scripts, and containerization process that best align with the characteristics of the selected data model. This approach optimizes resource utilization and enhances overall efficiency when handling heterogeneous data sources.

```
python -m run_converter [Model] <<path/to/dataset>>
```

Listing 9.3: Input parameters to run ARCXplore.

As illustrated in Listing 9.3, the [*Model*] parameter can be set to one of the following keywords: ["tabular," "document," "keyvalue," "multimodel," "domain-specific"]. The *multimodel* option enables ARCXplore to adapt automatically, accommodating heterogeneous data models simultaneously. The *domain-specific* option is similar to *multimodel*, but it performs additional, optional dataset integrity checks using external tools. For instance, when working with ARCs, the *domain-specific* parameter conducts ARC integrity checks and extracts ARC-specific metadata before invoking the *multimodel* sub-case.

9.1.6 Experiments

We conduct experiments to evaluate the efficiency and performance of ARCXplore using various datasets, with a primary focus on traversal and transformation speed. Our experiments are designed to assess ARCXplore's capability to handle both real-world and synthetic datasets.

- **Crawler Efficiency and Traversal**: We evaluate the data exploration capabilities of ARCXplore by testing its crawling efficiency. Specifically, we assess how effectively ARCXplore can traverse complex file systems, locate the necessary files, and exclude irrelevant ones. ARCXplore is deployed to crawl through various datasets with differing levels of directory complexity, and we measure the traversal speed, with a particular focus on the time it takes to locate and parse different file types.

- **Data Transformation**: We measure the time ARCXplore requires to transform files from different data models into standardized formats. This involves evaluating the duration needed for ARCXplore to convert these diverse datasets into formats suitable for their respective data models.

- **ARC Containerization**: We test ARCXplore with various real-world ARC datasets, measuring the turnaround time needed to crawl individual ARC datasets, transform the relevant files, group the files by their data model types, and orchestrate the containers in a complete cycle. For this process, we expose ARCXplore to numerous real-world ARC repositories and track the time it takes to initiate dataset processing and ultimately generate the container orchestration artifacts.

Setup

We conduct our experiments in a controlled environment on the de.NBI cloud. The experimental setup for ARCXplore utilizes a virtual machine configuration with eight virtual CPUs, 256 GB of RAM, and a 2TB attached hard disk volume. The chosen operating system is Rocky Linux v8.5. Within this environment, ARCXplore operates using Python version 3.9. Various utility libraries are utilized, including NetworkX version 3.1, Pandas version 2.0.1, psutil version 5.9.5, and PyYAML version 6.0.1. During the experiments, these libraries assist in data processing, system resource management, and configuration handling.

The settings for these experiments have been standardized. All experimental environments run with datasets located in dedicated folders. ARCXplore is initiated using the Python command, with [Model] replaced by the options "tabular," "multimodel," or "domain-specific."

```
python -m run_converter [Model] <<path/to/dataset>>
```

Datasets

In our experiments, we utilize a variety of datasets to validate the data model types of "tabular" and "multimodel". The tabular datasets only contain files in a particular data format, for example, as shown in the Table 9.2, the IMDB dataset is comprised solely of TSV files. A "multimodel" dataset, as the name suggests, is comprised of more than one data model. For example, the PRIDE dataset [77] consists of files in various formats, including CSV and JSON. The real-world ARC datasets are also used as "multimodel" datasets.

Folder	File Count	Total Size	Data Model
MovieLens Dataset	6	2.1GB	Tabular
TPCH1G	9	1.1GB	Tabular
IMDB dataset	8	6.2GB	Tabular
TPCH5G	10	5.2GB	Tabular
TPCH10G	11	11GB	Tabular
Antifibrotic Effects of Oxa	61	492KB	Multimodel
Goxr Characterization	79	980KB	Multimodel
Molecular plant responses	192	82.8 GB	Multimodel
PRIDE Dataset	347	1.5GB	Multimodel
LPA2 Complexome Profiling	2551	63.3	Multimodel
Synth4k	4078	2.8MB	Multimodel
Synth12k	12181	336KB	Multimodel
Brassicaceae Transposons Dataset	12710	3.1GB	Multimodel
Synth36k	36572	2.8MB	Multimodel
Synth80k	80878	4.1MB	Multimodel
Synth87k	87171	2.9MB	Multimodel
Synth160k	162606	4.1MB	Multimodel
MIMIC II Dataset	200048	4.9GB	Multimodel

Table 9.2: A table illustrating the datasets used.

- **TPCH Dataset**: The TPCH dataset is synthetically generated using the TPC-H benchmark tool [102]. We use the command `./dbgen -s [Scale]` to produce the datasets, where the `-s` parameter determines the scale of generation. For example, `-s 1` produces 1 GB of data, while `-s 10` generates 10 GB. The dataset consists of multiple CSV files.
- **IMDB Dataset**: The IMDB dataset [103] contains a collection of TSV files that provide comprehensive information about various aspects of movies, including titles, ratings, episodes, and movie names. This dataset is categorized as a tabular dataset due to its structured representation in TSV format.
- **MIMIC II Dataset**: The MIMIC-II dataset collects healthcare data derived from the Medical Information Mart for Intensive Care (MIMIC) II database. It includes patient records, medical measurements, and clinical information [104]. We classify the MIMIC-II dataset as a multimodal dataset because it contains files from different data model types, including structured data (such as patient demographics) and unstructured data (like clinical notes).
- **MovieLens Dataset**: The MovieLens dataset captures user ratings and tagging activity from MovieLens [105], a movie recommendation service. It includes 25 million ratings and 1.1 million tags for 62 423 movies. All files are in CSV format.
- **Synthetic Datasets**: These datasets, referred to as 'SynthNN,' were synthetically generated using a custom bash script. The numerical suffix after 'Synth' indicates the number of files, measured in thousands, within each dataset. The script allows for variations in folder structures, resulting in configurations that range from deep to shallow and wide to narrow.
- **PRIDE Dataset**: The PRIDE (PRoteomics IDEntifications) dataset [77] contains a collection of proteomics data obtained from mass spectrometry experiments. It comprises 347 files gathered from a selected set of proteomics experiments available in the PRIDE repository [27]. These files encompass various data formats, making them suitable for multimodal data analysis.

Results

Crawler: We exclusively use the "multimodel" datasets to measure the performance of the Crawler module. The objective is to measure and analyze the traversal times; thus, having a deep and wide file structure is an important aspect of measuring the crawler's efficacy. We split the datasets into two distinct groups:

- The first group includes smaller or medium-sized "multimodel" datasets, as listed below.
  - Antifibrotic Effects of Oxa
  - PRIDE Dataset
  - LPA2 Complexome Profiling

  - Synth12k
  - Brassicaceae Transposons
- The second group comprises datasets for load testing, particularly the synthetic datasets with a substantial number of files.
  - Synth12k, Synth36k, Synth80k, Synth87k, Synth160k
  - MIMIC II dataset

The experiment results are presented in Figures 9.3 and 9.4, which depict the crawling times for the two groups of datasets.

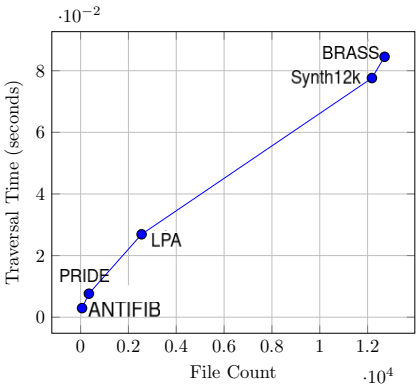

Figure 9.3: Graph showing the traversal times for small and medium datasets

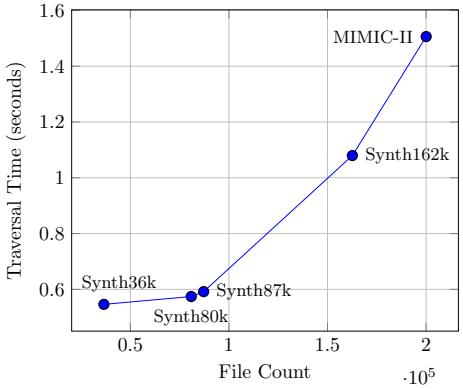

Figure 9.4: Graph showing the traversal times for the synthetic datasets

Figure 9.3 illustrates a linear relationship between the number of files within a dataset and the time required for traversal. This behavior can be attributed to the moderate complexity of the datasets, which remains consistent. As the dataset size increases, the traversal time follows a predictable pattern, suggesting that the crawler module maintains efficiency even with larger datasets. This characteristic is particularly valuable in real-world scenarios where datasets can vary significantly in size and complexity.

In contrast, Figure 9.4 presents a linear trend that gradually transitions into an exponential growth pattern as the folder structures of the datasets become more complex. Unlike the datasets shown in Figure 9.3, those depicted here have a considerably larger number of files and a much greater depth in their folder hierarchies. The departure from linearity can be attributed to the complexity introduced by these deeper directory structures. As the crawler traverses deeper into the folders, the number of subdirectories it must navigate increases exponentially, resulting in a corresponding rise in traversal time.

This highlights the importance of optimizing the crawler for datasets characterized by both large file counts and intricate folder hierarchies to ensure efficient processing of large datasets. While the crawler exhibits linear behavior regarding file count, its efficiency can be compromised by datasets with complex and deep directory structures. This indicates a need for tailored optimization strategies to handle such scenarios more effectively.

Dataset Name	File Count	Overall Size (GB)
TPCH1G	8	1.0
MovieLens	6	2.1
TPCH5G	8	5.1
IMDB	7	6.1
TPCH10G	8	10.8

Table 9.3: Dataset details.

Data Transformation: We evaluate the performance of ARCXplore's transformer module in converting files from various datasets into standardized data model formats. As shown in Table 9.3, we utilize TPC-H datasets of differing sizes, along with the IMDB and the MovieLens datasets. These particular datasets were selected due to their lower file counts but larger file sizes, as well as their common data model, which in this case is *TSV* or *CSV*.

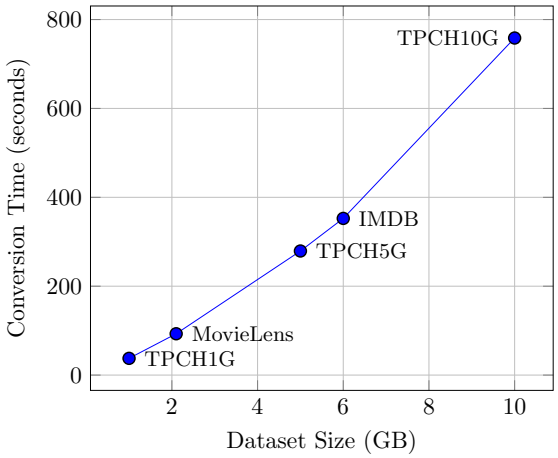

Figure 9.5: Conversion times of different tabular datasets.

The results shown in Figure 9.5 with the TPC-H and IMDB datasets show a linear relationship between the total dataset size and conversion time. This consistency can be attributed to the datasets' similar data model, primarily, which was mostly structured and tabular data in *TSV* format. This near-linear behavior conveys the module's stability and scalability, which proves advantageous for handling datasets of varying sizes.

Containerization: Here, we test all the functional components of ARCXplore at once, including domain-specific integrity checks, using the external ARC validation tool *ARCCommander* [101]. We select a range of real-world ARC datasets for this experiment, as shown in Table 9.4.

Figures 9.6 and 9.7 illustrate the execution times for the selected ARC datasets based on overall dataset size and file counts. We observe that datasets with a small number of files but larger overall sizes require significant execution time.

ARC Dataset Name	File Count	Total Size
Antifibrotic Effects of Oxa	61	1.8GB
Goxr Characterization	79	980KB
LPA2 Complexome Profiling	2551	63.1GB
Brassicaceae Transposons Dataset	12710	3.1GB
Molecular plant responses	192	82.8 GB

Table 9.4: ARC datasets with respective sizes and file counts.

Conversely, datasets containing a greater number of files with much smaller sizes, such as small CSV records, also demonstrate longer execution times. These ARC datasets consist of numerous small files, often in CSV or JSON format, which contributes to a higher file count.

This observation highlights that the balance between file size and file count directly impacts execution duration. Larger files inherently take more time to process due to their increased data volume. In contrast, datasets with a higher file count, even when consisting of smaller files, also require extended execution times. In summary, execution duration is influenced by the interplay between dataset size and file count, with both factors playing a critical role in overall performance.

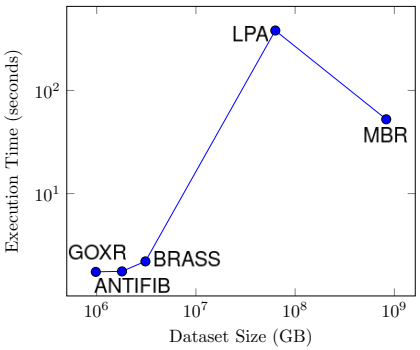

Figure 9.6: Graph showing the overall execution times for the ARC datasets based on dataset size

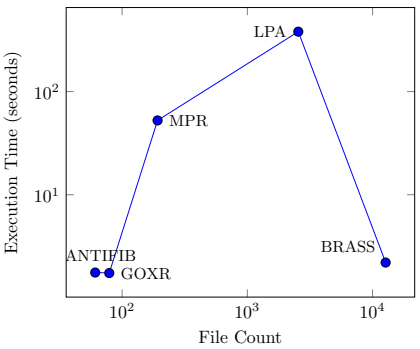

Figure 9.7: Graph showing the overall execution times for the ARC datasets based on file count

9.1.7 Related Work

As related work, we present Skluma and Apache NiFi, two tools that have inspired the foundational capabilities of ARCXplore. Skluma is a modular, scalable system designed to extract metadata from scientific files [106]. Its primary purpose is metadata extraction through various *extractor* components that can be launched either locally or within separate Docker containers, making scalability straightforward.

The crawler in Skluma is responsible for listing the contents of folders from different data sources and is invoked by the Orchestrator. The Orchestrator

is the module that implements the API and manages metadata extraction, as well as the resources utilized by both the crawler and extractors. Skluma can accommodate a wide range of file extensions, including images.

Apache NiFi is an open-source data integration tool that specializes in managing data flows [107]. It consists of processors and components that facilitate the extraction, transformation, and loading of data. The processor interface provides access to FlowFiles [108], including their attributes and content. The Processor is the fundamental building block used to create a NiFi data flow. Among its many features, NiFi offers effective and efficient data extraction.

Apache NiFi can handle a variety of data formats, including CSV (Comma-Separated Values), Parquet, XLSX, and more. We are particularly interested in these formats because they align with the transformation tasks performed by ARCXplore, ensuring compatibility between both tools in this process. Additionally, Apache NiFi offers a wide range of components that enable users to transfer files and categorize them based on their file extensions.

9.2 Extended Middleware

Since ARCXplore can transform ARC datasets into on-demand data warehouses featuring containerized data stores and a distributed query processing engine for cross-model query handling, we now introduce our indexing scheme into this architecture. We aim to evaluate the feasibility of integrating global indexes with an adaptive indexing scheme into an existing distributed query processing engine. For this evaluation, we utilize an established distributed query processing engine, such as Apache Drill, and incorporate extended middleware with custom components for query rewriting and pre-optimization. The basic idea is to rewrite the query predicates in cross-model user queries using global indexes, thereby evaluating the extent to which these indexes help accelerate overall query execution.

We focus solely on simple $SELECT$ queries, excluding complex aggregate functions for this assessment. In this context, rewriting a query involves updating predicates of the generic form $< attribute > BETWEEN < min > AND < max >$ to $< primaryKey > IN(< key1 >, < key2 >, ...)$ by leveraging the indexes. Figure 9.8 illustrates the proposed architecture, which incorporates the extended middleware and the indexing scheme.

Parser : The parser module is responsible for reading SQL strings and syntax trees, converting them into a relational format while validating the semantics of the queries. This validation includes checking that each identifier used in the query exists in the schema and ensuring compliance with type system rules. Additionally, the extended middleware features a local in-memory caching system, which prevents the need to re-parse the same user query each time it is submitted. It can recognize previously processed queries and retrieve their optimized versions. Furthermore, the parser module can be configured to parse structured queries in JSON format, providing enhanced capabilities.

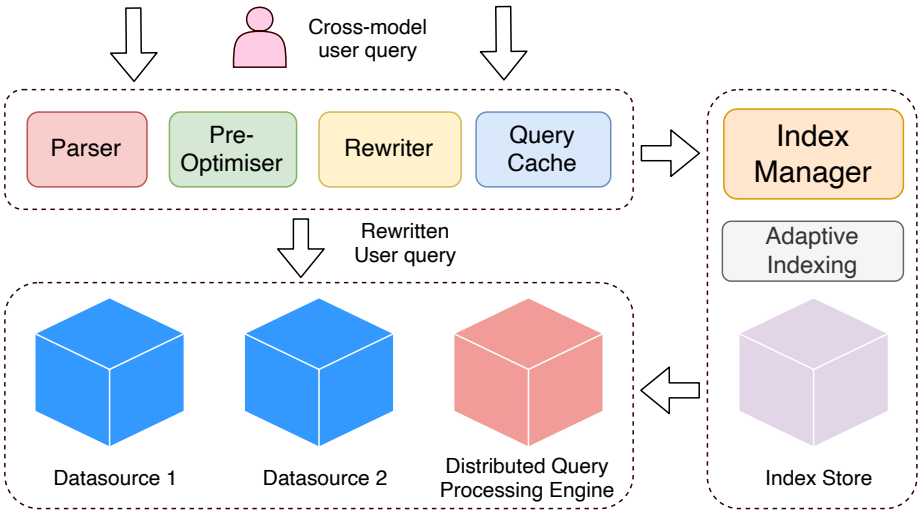

Figure 9.8: Distributed query processing engine supported with extended middleware and index management.

Pre-optimizer : The pre-optimizer analyzes the submitted query, invoking the rewrite engine for each predicate it finds. It is also responsible for loading plugins, if necessary, and injecting the appropriate dependencies. A parsed query in relational logic can be rewritten into any format using the rewrite engine.

Rewriter : The rewriter goes through each predicate in the query to identify suitable indexes. It employs rule-based decision-making to determine whether an index should be used for rewriting. When an appropriate index is found, the rewriter modifies the predicate. If no index is available, it returns the original predicate.

Index Manager : The index manager is responsible for handling each predicate in a query, which includes retrieving, creating, and refining the index, as illustrated in Figure 9.9. The index manager can support a wide range of adaptive indexing algorithms. When the index manager receives a predicate, it follows these steps:

- If there is no existing index, the manager returns an empty collection and creates a new index in the background.
- If an index is present, the manager returns the existing index and invokes the adaptive indexing algorithm to refine the index in the background.

9.3 Experiments

We evaluate the feasibility of integrating a priori-generated indexes to rewrite incoming user queries and speed up overall query execution in a distributed query processing engine. Since index creation and refinement occur in the back-

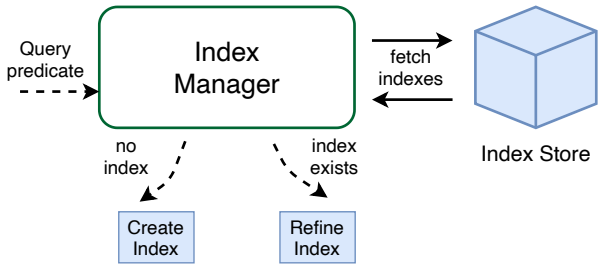

Figure 9.9: Index manager

ground asynchronously, this experiment focuses on measuring query rewriting and end-to-end execution times.

Setup: All tests are run on a VM operating on Rocky Linux release 9.2 (Blue Onyx) and powered by an AMD EPYC 7742 8-Core Processor, 265 GB of RAM, and 2.5 TB of storage running in deNBI cloud [109]. The test data sets consist of the IMDB movie database and the TPC-H databases. The Redis index store functions as a cluster with six nodes.

Benchmark Queries

query	indexed data type	tables	predicate type	predicate selectivity	matching tuples
Q1	-	movies	-	1.0	4803
Q2	-	movies	-	0.0	0
Q3	string	movies	point	0.0002	1
Q4	integer	movies	point	0.005	26
Q5	-	movies	-	0.86	4138
Q6	integer	movies	range	0.1	488
Q7	string	movies	range	0.122	586
Q8	integer, string	movies	range, point	0.10	489
Q9	string	movies, genres	point	$3.4 * 10^{-8}$	2
Q10	integer	movies	point	0.002	10

Table 9.5: Benchmark queries against the IMDB data set to measure the query rewriting and execution times.

query	indexed data type	tables	predicate type	predicate selectivity	matching tuples
Q1	-	region	-	1.0	5
Q2	string	region	point	0.2	1
Q3	string	region, nation	point	0.2	5
Q4	string	nation	range	0.08	2
Q5	-	region, nation	-	0.32	3231
Q6	integer,	region, nation	point	0.16	3231
Q7	string	supplier	range	0.4	49894
Q8	string	supplier	range, range	0.009	13770
Q9	integer	part	range	0.2	40474
Q10	integer	part	range	0.04	8183

Table 9.6: Benchmark queries against the TPC-H data set.

```
1SELECT * FROM movies
2WHERE title BETWEEN 'Lord of the Rings' AND 'Seabiscuit'
```

Listing 9.4: Example SQL query

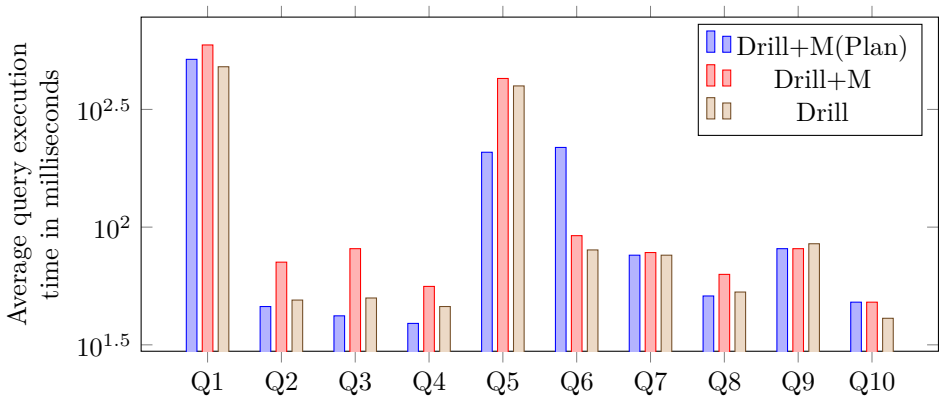

Figure 9.10: Average end-to-end execution times measured for the benchmark queries against IMDB dataset.

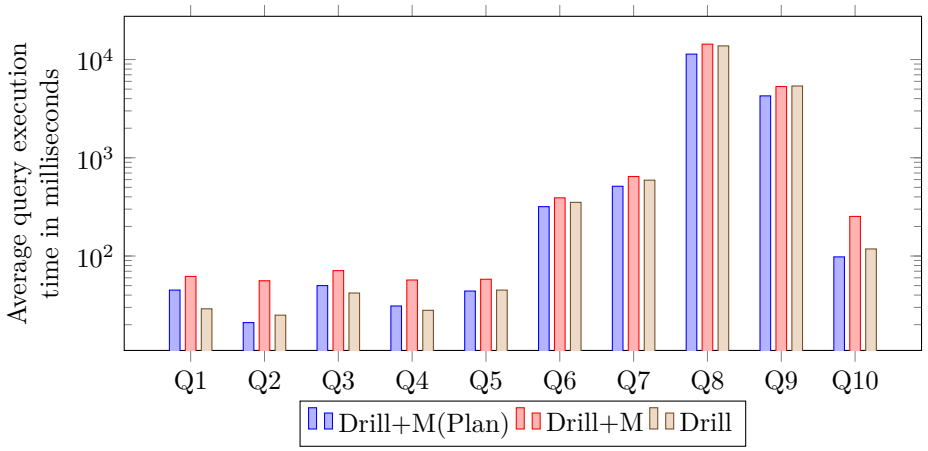

Figure 9.11: Average end-to-end execution times measured for the benchmark queries against the TPC-H dataset.

```
1SELECT * FROM movies
2WHERE id IN (19, 20, 22, 58, 70, 75, 77, ...)
```

Listing 9.5: Rewritten query from Listing. 9.4

Query Execution: We compare the overall execution times of Apache Drill with and without the extended middleware, which includes the index store and query rewriting engine. For this analysis, we utilize the benchmark queries listed in Tables 9.5 and 9.6 from the IMDB and TPC-H datasets, respectively. Each benchmark query is executed a hundred times with query caching enabled.

In the initial iteration, both the index store and query cache are empty. For all subsequent iterations, we ensure that all indexes are available and fully refined for every possible predicate. This approach minimizes the execution time by

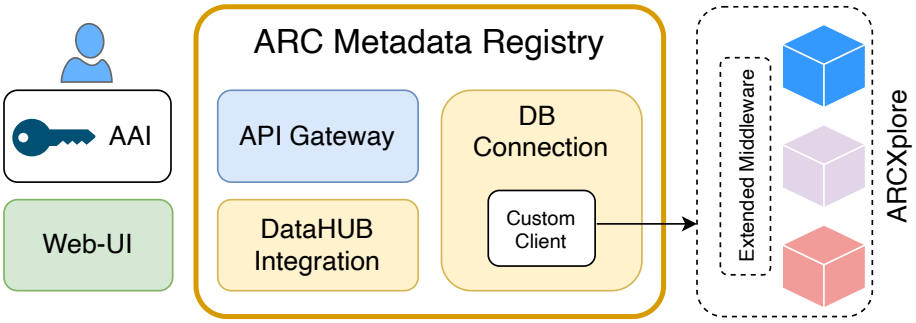

Figure 9.12: The ARC Metadata Registry expanded into a data exploration platform for both structured metadata and semi-structured or unstructured raw data from ARCs.

avoiding the overhead of refining the indexes and focusing solely on measuring query execution performance.

We execute the benchmark queries in two different ways. First, we run direct SQL queries through an HTTP API, which requires the middleware to rewrite the SQL query before executing it in Apache Drill. Listings 9.4 and 9.5 provide a sample query along with its equivalent rewritten version. As illustrated in Figures 9.10 and 9.11, it is evident that the additional overhead of SQL query rewriting (Drill+M) results in slower end-to-end query execution compared to direct query execution (Drill).

We also attempt to push the rewritten physical query plans (Drill+M(Plan)) directly to Apache Drill through the middleware, thereby reducing overhead. Our observations indicate that utilizing rewritten physical query plans improves execution times compared to executing input queries directly in Drill. As shown in Figure 9.10, the average execution times for all queries, except Q1, Q6, and Q10, are improved when using physical plans. Similar results are observed with the TPC-H dataset in Figure 9.11; the average execution times are improved, except for queries Q1, Q3, and Q4 with relatively complex query predicates.

9.3.1 Data Exploration Platform

Our research objective is to develop a comprehensive exploration platform that enables users to query both structured metadata and semi-structured or unstructured measurement data (or raw data) from ARCs across various data hubs. As part of our future work, we plan to enhance the existing ARC Metadata Registry into a comprehensive data exploration platform capable of handling both structured metadata and heterogeneous raw data from selected ARCs across multiple data hubs.

Figure 9.12 illustrates the proposed architecture, which requires the development of a custom client to connect with the on-demand multi-model warehouse. The custom client will accept user queries in either SQL or JSON format, passing them on to the extended middleware for further optimization and execution before returning the query results. The modular and extensible design of the ARC Metadata Registry enables seamless integration with the ARCXplore tool,

allowing users to select measurement data from the ARCs and execute exploration queries.

9.4 Summary

In this chapter, we introduced *ARCXplore*, an in-situ data exploration tool that enables on-demand querying of ARC datasets. ARCXplore can transform heterogeneous data files from single or multiple ARCs into an on-demand, multi-model warehouse with a distributed query processing engine, allowing users to run cross-model queries directly.

We also discussed the potential for enhancing the middleware to incorporate global adaptive indexes, along with a custom query rewriter and cache. Our experiments showed improvements in query execution times, particularly when queries are rewritten and submitted as physical plans. This suggests that using an index store in the context of distributed query processing is beneficial.

As a direction for future work, we plan to extend the architecture to support cross-model queries. In this context, leveraging global indexes and query rewriting can help minimize the number of subqueries. In the best-case scenario, the rewritten cross-model query could execute entirely on a single data source.

Chapter 10

Conclusion

It is widely acknowledged within the bioscience research community that an effective Research Data Management (RDM) platform, based on FAIR principles, is essential for packaging study-specific research results into uniquely identifiable and accessible FAIR digital objects. The integrated analysis of these research results enhances opportunities for knowledge discovery, collaboration, and innovation. This thesis introduces *PLANTdataHUB*, an end-to-end, data-centric FAIR RDM solution specifically designed for the plant science community. Our research goal is to design and develop search and exploration platforms that facilitate the integrated analysis of datasets generated by research groups using the PLANTdataHUB solution. These datasets consist of both structured experimental metadata and semi-structured or unstructured measurement and contextual data. Our RDM solution defines the scope and application requirements for this thesis.

In PLANTdataHUB, the units of research output are referred to as Annotated Research Contexts (ARCs), which are self-contained, interoperable, and reproducible datasets. The ARC specification establishes a standardized folder and file layout to accommodate experimental metadata along with measurement data, contextual information, workflows, software, and other relevant data. This specification creates uniquely identifiable digital objects, facilitating data versioning, referencing, and sharing across plant research projects. DataHUBs serve as repositories that maintain ARC datasets for individual research groups within the PLANTdataHUB solution. It employs a distributed version control system to store and manage ARCs locally, thereby enabling a collaborative environment for systems biology that supports structured data management. This thesis presents an incremental approach to developing a suite of search and exploration applications for plant science within the framework of the PLANTdataHUB solution. We have implemented a wide range of tools and applications that are currently in production, serving the plant science community across various study disciplines.

We started by integrating structured metadata from the ARCs to meet the search requirements of the plant research community. To achieve this, we developed the ARC Metadata Registry, a cloud-native application that seamlessly integrates experimental metadata from ARCs across various research groups in real-time. The ARC Metadata Registry offers comprehensive data access through APIs, supports external authentication and authorization frameworks, and serves as an ideal platform for interdisciplinary data exploration and analysis. This application serves as a centralized index and search engine, providing an integrated search of plant science datasets stored across various repositories without the need for metadata transformation or harmonization, unlike other metadata indexes for plant sciences. It also features dataset landing pages that facilitate dataset discovery through public search platforms, such as Google Dataset Search. The iterative development of research data man-

agement (RDM) solutions for the plant research community necessitates the creation of complementary tools and search applications that keep pace with the constantly changing functional requirements. To achieve this, we employ modularization, abstraction, and a microservice architecture in our application design.

The availability of multi-omics metadata from all the ARCs within the PLANTdataHUB solution opened up new opportunities to leverage this clean and comprehensive corpus for training language models. We have developed ArcBERT, a sentence-BERT-based language model designed for metadata exploration and analysis. This model is pre-trained on plant science literature and fine-tuned using the ARC metadata corpus to enhance its ability to understand user queries in natural language. Additionally, we utilized existing indexing mechanisms to index vector embeddings, enabling fast nearest-neighbor searches. Our experiments demonstrated that ArcBERT can effectively comprehend natural language-style queries and the hierarchical structures within the metadata, outperforming traditional text search engines in retrieving consistent results that are semantically closer to user queries. The ability for users to write queries in plain English is invaluable in the research domain, where keyword-centric data retrieval still plays an important role.

We later shift our research focus from experimental metadata to explore the large-scale measurement and contextual data generated by the plant science communities. The PLANTdataHUB solution promotes a decentralized storage system, allowing individual research groups to manage their datasets locally. Since maintaining large volumes of measurement data from all ARCs within a centralized storage platform can be challenging, we prioritized solutions that enable data analysts and users to select specific ARCs for integrated data analysis. This approach facilitates on-demand access to heterogeneous measurement data while minimizing data movement. We primarily focus our research on polystore architecture, flexible indexing schemes, adaptive indexing to the query workload shifts, and the essential middleware to integrate these components.

To effectively utilize polystore architectures for diverse datasets, we propose utilizing global indexes. For the specialized query needs of the bio-sciences domain, adding an index store can improve the cross-model query processing capabilities of polystores. Our findings demonstrate that multi-omics applications developed on polystore architectures, incorporating an additional index store, can perform competitively with state-of-the-art systems. In particular, global indexes enhance the performance of cross-model queries when accessing extensive heterogeneous datasets and evaluating complex Boolean predicate expressions.

We extended our research by refining the indexing scheme to adjust to changes in query workload, addressing the challenges of index attribute selection and the need for cost-based index refinement. We proposed an adaptive index management model based on the existing DB Cracking methods, while introducing a new cracking variant, called Statistical DB Cracking, to ensure fair index refinement by mitigating bias towards frequently occurring elements in the data. Our model proved significantly more effective for index management and offered cost-effectiveness in terms of storage space utilization.

Lastly, we present *ARCXplore*, an in-situ data exploration tool designed to instantly transform the packaged research datasets (ARCs) into an on-demand warehouse equipped with a distributed query processing engine for cross-model queries. This application establishes a rapid ETL (Extract, Transform, Load) pipeline, enabling the transformation of heterogeneous measurement data from the ARCs and loading it into data-model-specific containers. Additionally, it can include a distributed query processing engine to handle cross-model queries. We have also implemented our flexible indexing scheme and adaptive index management as an extension of the on-demand warehouse. This enhancement enabled us to assess the effectiveness of using a key-value index store within existing off-the-shelf distributed query processing engines. Our experiments demonstrated that extending the middleware resulted in improvements in query execution times, particularly when queries were submitted as physical plans.

The full implementation of the extended middleware for query pre-processing and optimization, utilizing global indexes, along with integration with the existing ARC Metadata Registry application, is currently in progress. Once this integration is complete, it will provide a comprehensive exploration platform for plant science datasets within the PLANTdataHUB solution. However, the applications must be extended to accommodate data exploration across multiple RDM ecosystems, thus necessitating innovative data migration and harmonization techniques.

The use of index stores for faster query processing in other virtual integration platforms or federated systems also opens up new avenues for further research, particularly in cross-model query optimization by augmenting semantics using integrated metadata and ontologies. In this context, it is essential to develop a new domain-specific query language that is more expressive and better aligned with the general query requirements of multi-omics analysis.

As large language models (LLMs) become increasingly popular in information retrieval, a noticeable shift is occurring towards training these models, particularly in the field of plant science. FAIR-RDM solutions, such as PLANTdataHUB, aim to structure and standardize large-scale bioscience datasets, making a comprehensive collection of data (structured metadata and semi-structured or unstructured measurement or raw data) readily available for training domain-specific models with minimal effort required for data transformation and harmonization. As an avenue of future work, fine-tuning and leveraging LLMs for specific biological natural language processing tasks can enhance the exploration of multi-omics data, significantly improving opportunities for biological knowledge discovery, collaboration, and innovation.

List of Figures

List of Algorithms

List of Tables

Listings

Bibliography

[1] I. Subramanian, S. Verma, S. Kumar, A. Jere, and K. Anamika, "Multi-omics data integration, interpretation, and its application," *Bioinformatics and Biology Insights*, vol. 14, p. 1177932219899051, 2020, pMID: 32076369. [Online]. Available: https://doi.org/10.1177/1177932219899051

[2] C. Pommier, F. Coppens, H. Ćwiek-Kupczyńska, D. Faria, S. Beier, C. Miguel, C. Michotey, F. D'Anna, S. Owen, and K. Gruden, *Plant Science Data Integration, from Building Community Standards to Defining a Consistent Data Lifecycle.* Cham: Springer International Publishing, 2023, pp. 149–160. [Online]. Available: https://doi.org/10.1007/978-3-031-13276-6_8

[3] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, p. 160018, Mar 2016. [Online]. Available: https://doi.org/10.1038/sdata.2016.18

[4] X. Wang, C. Williams, Z. H. Liu, and J. Croghan, "Big data management challenges in health research - a literature review," *Briefings in Bioinformatics*, vol. 20, no. 1, pp. 156–167, 2019. [Online]. Available: https://doi.org/10.1093/bib/bbx086

[5] D. Gomez-Cabrero, I. Abugessaisa, D. Maier, A. Teschendorff, M. Merkenschlager, A. Gisel, E. Ballestar, E. Bongcam-Rudloff, A. Conesa, and J. Tegnér, "Data integration in the era of omics: current and future challenges," *BMC Systems Biology*, vol. 8, no. 2, p. I1, Mar 2014. [Online]. Available: https://doi.org/10.1186/1752-0509-8-S2-I1

[6] W. Wruck, M. Peuker, and C. R. Regenbrecht, "Data management strategies for multinational large-scale systems biology projects," *Briefings in Bioinformatics*, vol. 15, no. 1, pp. 65–78, 10 2012. [Online]. Available: https://doi.org/10.1093/bib/bbs064

[7] S.-A. Sansone, P. Rocca-Serra, D. Field, E. Maguire, C. Taylor, and et al., "Toward interoperable bioscience data." *Nature genetics*, vol. 44, no. 2, pp. 121–6, 2012. [Online]. Available: https://www.nature.com/articles/ng.1054

[8] J. D. Tenenbaum, S.-A. Sansone, and M. Haendel, "A sea of standards for omics data: sink or swim?" *Journal of the American Medical*

Informatics Association, vol. 21, no. 2, pp. 200–203, 09 2013. [Online]. Available: https://doi.org/10.1136/amiajnl-2013-002066

[9] S. A. Chervitz, E. W. Deutsch, D. Field, H. Parkinson, J. Quackenbush, P. Rocca-Serra, S.-A. Sansone, C. J. Stoeckert, C. F. Taylor, R. Taylor, and C. A. Ball, *Data Standards for Omics Data: The Basis of Data Sharing and Reuse*. Totowa, NJ: Humana Press, 2011, pp. 31–69. [Online]. Available: https://doi.org/10.1007/978-1-61779-027-0_2

[10] H. Ćwiek-Kupczyńska, T. Altmann, D. Arend, E. Arnaud, D. Chen, G. Cornut, F. Fiorani, W. Frohmberg, A. Junker, C. Klukas, M. Lange, C. Mazurek, A. Nafissi, P. Neveu, J. van Oeveren, C. Pommier, H. Poorter, P. Rocca-Serra, S.-A. Sansone, U. Scholz, M. van Schriek, Ü. Seren, B. Usadel, S. Weise, P. Kersey, and P. Krajewski, "Measures for interoperability of phenotypic data: minimum information requirements and formatting," *Plant Methods*, vol. 12, p. 44, Nov. 2016.

[11] S.-A. Sansone, P. Rocca-Serra, A. Gonzalez-Beltran, D. Johnson, and I. Community, *ISA Model and Serialization Specifications 1.0*, Oct. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.163640

[12] K. Dumschott, H. Dörpholz, M.-A. Laporte, D. Brilhaus, A. Schrader, B. Usadel, S. Neumann, E. Arnaud, and A. Kranz, "Ontologies for increasing the FAIRness of plant research data," *Front Plant Sci*, vol. 14, p. 1279694, Nov. 2023.

[13] Soiland-Reyes, Stian and Sefton, Peter and Crosas, Mercè and Castro, Leyla Jael and Coppens, Frederik and Fernández, José M and Garijo, Daniel and Grüning, Björn and La Rosa, Marco and Leo, Simone and others, "Packaging research artefacts with RO-Crate," *Data Science*, vol. 5, no. 2, pp. 97–138, 2022.

[14] K. De Smedt, D. Koureas, and P. Wittenburg, "FAIR Digital Objects for Science: From Data Pieces to Actionable Knowledge Units," *Publications*, vol. 8, no. 2, 2020. [Online]. Available: https://www.mdpi.com/2304-6775/8/2/21

[15] Natasha Noy and Matthew Burgess and Dan Brickley, "Google Dataset Search: Building a search engine for datasets in an open Web ecosystem," 2019.

[16] H. Cousijn, A. Kenall, E. Ganley, M. Harrison, D. Kernohan, T. Lemberger, F. Murphy, P. Polischuk, S. Taylor, M. Martone, and T. Clark, "A data citation roadmap for scientific publishers," *Sci Data*, vol. 5, p. 180259, Nov. 2018.

[17] H. L. Weil, K. Schneider, M. Tschöpe, J. Bauer, O. Maus, K. Frey, D. Brilhaus, C. Martins Rodrigues, G. Doniparthi, F. Wetzels, J. Lukasczyk, A. Kranz, B. Grüning, D. Zimmer, S. Deßloch, D. von Suchodoletz, B. Usadel, C. Garth, and T. Mühlhaus, "PLANTdataHUB: a collaborative platform for continuous FAIR data sharing in plant research," *The Plant Journal*, vol. 116, no. 4, pp. 974–988, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/tpj.16474

[18] T. Mühlhaus, D. Brilhaus, M. Tschöpe, O. Maus, B. Grüning, C. Garth, C. Martins Rodrigues, and D. von Suchodoletz, *DataPLANT-Tools and Services to structure the Data Jungle for fundamental plant researchers.* heiBOOKS, Apr. 2022, p. 132–145. [Online]. Available: https://books.ub.uni-heidelberg.de/heibooks/catalog/book/979/chapter/13724

[19] DataPLANTcommunity, "nfdi4plants/ARCitect: Arcitect," Sep. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8307729

[20] X.-R. Zhou, S. Beier, D. Brilhaus, C. M. Rodrigues, T. Mühlhaus, D. von Suchodoletz, R. M. Twyman, B. Usadel, and A. Kranz, "DataPLAN: a web-based data management plan generator for the plant sciences," *bioRxiv*, 2023. [Online]. Available: https://www.biorxiv.org/content/early/2023/07/10/2023.07.07.548147

[21] C. Garth, J. Lukasczyk, T. Mühlhaus, B. Venn, J. Krüger, K. Glogowski, C. Martins Rodrigues, and D. von Suchodoletz, *Immutable yet evolving: ARCs for permanent sharing in the research data-time continuum.* heiBOOKS, Apr. 2022, p. 366–373. [Online]. Available: https://books.ub.uni-heidelberg.de/heibooks/catalog/book/979/chapter/13751

[22] DataPLANTcommunity, "Annotated Research Context Specification, v1.1-rfc," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8302662

[23] S.-A. Sansone, P. Rocca-Serra, A. Gonzalez-Beltran, D. Johnson, and I. Community, *ISA Model and Serialization Specifications 1.0 - ISA Json format*, https://isa-specs.readthedocs.io/en/latest/isajson.html, 2016, [Webpage; Accessed on: 2024-02-07].

[24] M. Christie, A. Bhandar, S. Nakandala, S. Marru, E. Abeysinghe, S. Pamidighantam, and M. Pierce, "Using keycloak for gateway authentication and authorization," 10 2017.

[25] K. Haug, R. M. Salek, P. Conesa, J. Hastings, P. de Matos, M. Rijnbeek, T. Mahendraker, M. Williams, S. Neumann, P. Rocca-Serra, E. Maguire, A. González-Beltrán, S.-A. Sansone, J. L. Griffin, and C. Steinbeck, "MetaboLights–an open-access general-purpose repository for metabolomics studies and associated meta-data," *Nucleic Acids Res*, vol. 41, no. Database issue, pp. D781–6, Oct. 2012.

[26] K. Haug, K. Cochrane, V. C. Nainala, M. Williams, J. Chang, K. V. Jayaseelan, and C. O'Donovan, "MetaboLights: a resource evolving in response to the needs of its scientific community," *Nucleic Acids Research*, vol. 48, no. D1, pp. D440–D444, 11 2019. [Online]. Available: https://doi.org/10.1093/nar/gkz1019

[27] H. Hermjakob and R. Apweiler, "The Proteomics Identifications Database (PRIDE) and the ProteomExchange Consortium: making proteomics data accessible," *Expert Review of Proteomics*, vol. 3, no. 1, pp. 1–3, 2006, pMID: 16445344. [Online]. Available: https://doi.org/10.1586/14789450.3.1.1

[28] E. O. F. N. R. . OpenAIRE., "Zenodo: Research. shared." https://zenodo.org/, 2013, webpage; Accessed: 2025-02-14.

[29] J. Singh, "Figshare," *Journal of Pharmacology and Pharmacotherapeutics*, vol. 2, no. 2, pp. 138–139, 2011. [Online]. Available: https://doi.org/10.4103/0976-500X.81919

[30] A. Bauch, I. Adamczyk, P. Buczek, F.-J. Elmer, K. Enimanev, P. Glyzewski, M. Kohler, T. Pylak, A. Quandt, C. Ramakrishnan, C. Beisel, L. Malmström, R. Aebersold, and B. Rinn, "openBIS: a flexible framework for managing and analyzing complex data in biology research," *BMC Bioinformatics*, vol. 12, no. 1, p. 468, Dec 2011. [Online]. Available: https://doi.org/10.1186/1471-2105-12-468

[31] K. Wolstencroft, O. Krebs, J. L. Snoep, N. J. Stanford, F. Bacall, M. Golebiewski, R. Kuzyakiv, Q. Nguyen, S. Owen, S. Soiland-Reyes, J. Straszewski, D. D. van Niekerk, A. R. Williams, L. Malmström, B. Rinn, W. Müller, and C. Goble, "FAIRDOMHub: a repository and collaboration environment for sharing systems biology research," *Nucleic Acids Research*, vol. 45, no. D1, pp. D404–D407, 11 2016. [Online]. Available: https://doi.org/10.1093/nar/gkw1032

[32] C. Capitanchik, S. Ireland, A. Harston, C. Cheshire, D. M. Jones, F. C. Lee, I. Ruiz de los Mozos, I. A. Iosub, K. Kuret, R. Faraway, O. G. Wilkins, R. Arora, M. Hallegger, M. Modic, A. M. Chakrabarti, N. M. Luscombe, and J. Ule, "Flow: a web platform and open database to analyse, store, curate and share bioinformatics data at scale," *bioRxiv*, 2023. [Online]. Available: https://www.biorxiv.org/content/early/2023/08/29/2023.08.22.544179

[33] Perez-Riverol, Yasset and Bai, Mingze and da Veiga Leprevost, Felipe and Squizzato, Silvano and Park, Young Mi and Haug, Kenneth and Carroll, Adam J and Spalding, Dylan and Paschall, Justin and Wang, Mingxun and others, "Discovering and linking public omics data sets using the Omics Discovery Index," *Nature biotechnology*, vol. 35, no. 5, pp. 406–409, 2017.

[34] H. Y. I. Lam, X. E. Ong, and M. Mutwil, "Large language models in plant biology," *CoRR*, vol. abs/2401.02789, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.02789

[35] Q. Zhang, K. Ding, T. Lyv, X. Wang, Q. Yin, Y. Zhang, J. Yu, Y. Wang, X. Li, Z. Xiang, X. Zhuang, Z. Wang, M. Qin, M. Zhang, J. Zhang, J. Cui, R. Xu, H. Chen, X. Fan, H. Xing, and H. Chen, "Scientific large language models: A survey on biological & chemical domains," *CoRR*, vol. abs/2401.14656, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.14656

[36] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang, "Biobert: a pre-trained biomedical language representation model for biomedical text mining," *CoRR*, vol. abs/1901.08746, 2019. [Online]. Available: http://arxiv.org/abs/1901.08746

[37] I. Beltagy, K. Lo, and A. Cohan, "Scibert: A pretrained language model for scientific text," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019,*

Hong Kong, China, November 3-7, 2019, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 3613–3618. [Online]. Available: https://doi.org/10.18653/v1/D19-1371

[38] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[39] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 3980–3990. [Online]. Available: https://doi.org/10.18653/v1/D19-1410

[40] Online, "Pubmed," https://pubmed.ncbi.nlm.nih.gov/, webpage; Accessed: 2025-04-14.

[41] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020.

[42] P. P. Ghadekar, S. Mohite, O. More, P. Patil, Sayantika, and S. Mangrule, "Sentence meaning similarity detector using faiss," in *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, 2023, pp. 1–6.

[43] V. Karpukhin, B. Oguz, S. Min, P. S. H. Lewis, L. Wu, S. Edunov, D. Chen, and W. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 6769–6781. [Online]. Available: https://doi.org/10.18653/v1/2020.emnlp-main.550

[44] A. Askari, A. Abolghasemi, G. Pasi, W. Kraaij, and S. Verberne, "Injecting the BM25 score as text improves bert-based re-rankers," in *Advances in Information Retrieval - 45th European Conference on Information Retrieval, ECIR 2023, Dublin, Ireland, April 2-6, 2023,*

Proceedings, Part I, ser. Lecture Notes in Computer Science, J. Kamps, L. Goeuriot, F. Crestani, M. Maistro, H. Joho, B. Davis, C. Gurrin, U. Kruschwitz, and A. Caputo, Eds., vol. 13980. Springer, 2023, pp. 66–83. [Online]. Available: https://doi.org/10.1007/978-3-031-28244-7_5

[45] G. P. Copeland and S. Khoshafian, "A decomposition storage model," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, S. B. Navathe, Ed. ACM Press, 1985, pp. 268–279. [Online]. Available: https://doi.org/10.1145/318898.318923

[46] Z. H. Liu and D. Gawlick, "Management of flexible schema data in rdbmss - opportunities and limitations for nosql -," in *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. [Online]. Available: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper5.pdf

[47] J. Corwin, A. Silberschatz, P. L. Miller, and L. N. Marenco, "Application of information technology: Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems," *JAMIA*, vol. 14, no. 1, pp. 86–93, 2007. [Online]. Available: https://doi.org/10.1197/jamia.M2189

[48] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[49] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, 2010. [Online]. Available: https://doi.org/10.1109/TKDE.2009.57

[50] M. C. Jeffrey and J. G. Steffan, "Understanding bloom filter intersection for lazy address-set disambiguation," in *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, R. Rajaraman and F. M. auf der Heide, Eds. ACM, 2011, pp. 345–354. [Online]. Available: https://doi.org/10.1145/1989493.1989551

[51] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[52] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson, "Enabling query processing across heterogeneous data models: A survey," in *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, J. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang, and M. Toyoda, Eds. IEEE Computer Society, 2017, pp. 3211–3220. [Online]. Available: https://doi.org/10.1109/BigData.2017.8258302

[53] V. Gadepally, P. Chen, J. Duggan, A. J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The bigdawg polystore

system and architecture," *CoRR*, vol. abs/1609.07548, 2016. [Online]. Available: http://arxiv.org/abs/1609.07548

[54] K. Kaur and R. Rani, "Managing data in healthcare information systems: Many models, one solution," *Computer*, vol. 48, no. 3, pp. 52–59, 2015. [Online]. Available: https://doi.org/10.1109/MC.2015.77

[55] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," vol. 26, no. 1, p. 65–74, mar 1997. [Online]. Available: https://doi.org/10.1145/248603.248616

[56] A. Davoudian and M. Liu, "Big data systems: A software engineering perspective," *ACM Comput. Surv.*, vol. 53, no. 5, pp. 110:1–110:39, 2021. [Online]. Available: https://doi.org/10.1145/3408314

[57] Z. Kaoudi and J. Quiané-Ruiz, "Cross-platform data processing: Use cases and challenges," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 1723–1726. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00223

[58] P. P. Khine and Z. Wang, "A review of polyglot persistence in the big data world," *Inf.*, vol. 10, no. 4, p. 141, 2019. [Online]. Available: https://doi.org/10.3390/info10040141

[59] Y. Li, E. Lo, M. L. Yiu, and W. Xu, "Query optimization over cloud data market," in *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, G. Alonso, F. Geerts, L. Popa, P. Barceló, J. Teubner, M. Ugarte, J. V. den Bussche, and J. Paredaens, Eds. OpenProceedings.org, 2015, pp. 229–240. [Online]. Available: https://doi.org/10.5441/002/edbt.2015.21

[60] C. Bondiombouy and P. Valduriez, "Query processing in multistore systems: an overview," *Int. J. Cloud Comput.*, vol. 5, no. 4, pp. 309–346, 2016. [Online]. Available: https://doi.org/10.1504/IJCC.2016.10001884

[61] V. Josifovski, T. Katchaounov, and T. Risch, "Evaluation of join strategies for distributed mediation," in *Advances in Databases and Information Systems, 5th East European Conference, ADBIS 2001, Vilnius, Lithuania, September 25-28, 2001, Proceedings*, ser. Lecture Notes in Computer Science, A. Caplinskas and J. Eder, Eds., vol. 2151. Springer, 2001, pp. 308–322. [Online]. Available: https://doi.org/10.1007/3-540-44803-9_24

[62] M. T. Roth and P. M. Schwarz, "Don't scrap it, wrap it! A wrapper architecture for legacy data sources," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 1997, pp. 266–275. [Online]. Available: http://www.vldb.org/conf/1997/P266.PDF

[63] A. M. Gupta, V. Gadepally, and M. Stonebraker, "Cross-engine query execution in federated database systems," in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 2016, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2016.7761648

[64] B. Kolev, O. Levchenko, E. Pacitti, P. Valduriez, R. Vilaça, R. C. Gonçalves, R. Jiménez-Peris, and P. Kranas, "Parallel polyglot query processing on heterogeneous cloud data stores with leanxcale," in *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, N. Abe, H. Liu, C. Pu, X. Hu, N. K. Ahmed, M. Qiao, Y. Song, D. Kossmann, B. Liu, K. Lee, J. Tang, J. He, and J. S. Saltz, Eds. IEEE, 2018, pp. 1757–1766. [Online]. Available: https://doi.org/10.1109/BigData.2018.8622187

[65] B. Kolev, R. Pau, O. Levchenko, P. Valduriez, R. Jiménez-Peris, and J. Pereira, "Benchmarking polystores: The cloudmdsql experience," in *2016 IEEE International Conference on Big Data (IEEE BigData 2016), Washington DC, USA, December 5-8, 2016*, J. Joshi, G. Karypis, L. Liu, X. Hu, R. Ak, Y. Xia, W. Xu, A. Sato, S. Rachuri, L. H. Ungar, P. S. Yu, R. Govindaraju, and T. Suzumura, Eds. IEEE Computer Society, 2016, pp. 2574–2579. [Online]. Available: https://doi.org/10.1109/BigData.2016.7840899

[66] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris, "Musqle: Distributed SQL query execution over multiple engine environments," in *2016 IEEE International Conference on Big Data Washington DC, USA, December 5-8, 2016*.

[67] A. Maccioni and R. Torlone, "Augmented access for querying and exploring a polystore," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*.

[68] M. Vogt, A. Stiemer, and H. Schuldt, "Polypheny-db: Towards a distributed and self-adaptive polystore," in *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, N. Abe, H. Liu, C. Pu, X. Hu, N. K. Ahmed, M. Qiao, Y. Song, D. Kossmann, B. Liu, K. Lee, J. Tang, J. He, and J. S. Saltz, Eds. IEEE, 2018, pp. 3364–3373. [Online]. Available: https://doi.org/10.1109/BigData.2018.8622353

[69] Online, "Apache drill," http://drill.apache.org/, webpage; Accessed: 2025-02-14.

[70] ——, "Prestodb," https://prestodb.io/, webpage; Accessed: 2025-04-26.

[71] ——, "Redis," https://redis.io/, webpage; Accessed: 2025-04-26.

[72] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Softw. Pract. Exp.*, 2016.

[73] K. Wu, K. Stockinger, and A. Shoshani, "Breaking the curse of cardinality on bitmap indexes," in *Scientific and Statistical Database Management, 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008, Proceedings*, ser. Lecture Notes in Computer Science, B. Ludäscher and N. Mamoulis, Eds., vol. 5069. Springer, 2008, pp. 348–365. [Online]. Available: https://doi.org/10.1007/978-3-540-69497-7_23

[74] P. E. O'Neil and D. Quass, "Improved query performance with variant indexes," in *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona,*

USA, J. Peckham, Ed. ACM Press, 1997, pp. 38–49. [Online]. Available: https://doi.org/10.1145/253260.253268

[75] E. J. O'Neil, P. E. O'Neil, and K. Wu, "Bitmap index design choices and their performance implications," in *Eleventh International Database Engineering and Applications Symposium (IDEAS 2007), September 6-8, 2007, Banff, Alberta, Canada*. IEEE Computer Society, 2007, pp. 72–84. [Online]. Available: https://doi.org/10.1109/IDEAS.2007.4318091

[76] Gajendra Doniparthi and Timo Mühlhaus and Stefan Deßloch, "A Hybrid Data Model and Flexible Indexing for Interactive Exploration of Large-Scale Bio-science Data," in *New Trends in Database and Information Systems - ADBIS 2021 Short Papers, Estonia, August 24-26, Proceedings*, 2021.

[77] Online, "Proteomics identifications database," https://www.ebi.ac.uk/pride/, webpage; Accessed: 2025-02-14.

[78] S. Chaudhuri and V. R. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, Eds. ACM, 2007, pp. 3–14. [Online]. Available: http://www.vldb.org/conf/2007/papers/special/p3-chaudhuri.pdf

[79] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, 2007, pp. 68–78.

[80] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "The uncracked pieces in database cracking," *Proc. VLDB Endow.*, vol. 7, no. 2, pp. 97–108, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p97-schuhknecht.pdf

[81] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe, "Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 585–597, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p586-idreos.pdf

[82] G. Graefe and H. A. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, ser. ACM International Conference Proceeding Series, I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, Eds., vol. 426. ACM, 2010, pp. 371–381. [Online]. Available: https://doi.org/10.1145/1739041.1739087

[83] F. M. Schuhknecht, J. Dittrich, and L. Linden, "Adaptive adaptive indexing," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 665–676. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00066

[84] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores," *Proc. VLDB Endow.*, vol. 5, no. 6, pp. 502–513, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p502_felixhalim_vldb2012.pdf

[85] T. Gündem, "Near optimal multiple choice index selection for relational databases," vol. 37, no. 2, pp. 111–120. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0898122198002569

[86] A. Caprara, M. Fischetti, and D. Maio, "Exact and approximate algorithms for the index selection problem in physical database design," vol. 7, no. 6, pp. 955–967. [Online]. Available: http://ieeexplore.ieee.org/document/476501/

[87] Z. Sadri, L. Gruenwald, and E. Leal, "Online index selection using deep reinforcement learning for a cluster database," in *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, pp. 158–161. [Online]. Available: https://ieeexplore.ieee.org/document/9094124/

[88] M. Valavala and W. Alhamdani, "Automatic database index tuning using machine learning," in *2021 6th International Conference on Inventive Computation Technologies (ICICT)*. IEEE, pp. 523–530. [Online]. Available: https://ieeexplore.ieee.org/document/9358646/

[89] A. Beutel, T. Kraska, E. Chi, J. Dean, and N. Polyzotis, "A machine learning approach to databases indexes," in *ML Systems Workshop*. NIPS, 2017.

[90] X. Zhou, L. Liu, W. Li, L. Jin, S. Li, T. Wang, and J. Feng, "AutoIndex: An incremental index management system for dynamic workloads," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, pp. 2196–2208. [Online]. Available: https://ieeexplore.ieee.org/document/9835594/

[91] E. Petraki, S. Idreos, and S. Manegold, "Holistic indexing in main-memory column-stores," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 1153–1166. [Online]. Available: https://doi.org/10.1145/2723372.2723719

[92] B. W. Ballard, "The *-minimax search procedure for trees containing chance nodes," *Artif. Intell.*, vol. 21, no. 3, pp. 327–350, 1983. [Online]. Available: https://doi.org/10.1016/S0004-3702(83)80015-0

[93] R. E. Korf, "Multi-player alpha-beta pruning," *Artif. Intell.*, vol. 48, no. 1, pp. 99–111, 1991. [Online]. Available: https://doi.org/10.1016/0004-3702(91)90082-U

[94] S. J. Russell, P. Norvig, M.-w. Chang, J. Devlin, A. Dragan, D. Forsyth, I. Goodfellow, J. Malik, V. Mansinghka, J. Pearl, and M. J. Wooldridge, *Artificial intelligence: a modern approach*, fourth edition, global edition ed., ser. Pearson series in artificial intelligence. Pearson.

[95] G. Doniparthi, "Using a Key-Value Index-Store for Cross-Model Join Queries over Heterogeneous Data Sources," in *Advances in Databases and Information Systems - 27th European Conference, ADBIS 2023, Barcelona, Spain, September 4-7, 2023, Proceedings*, ser. Lecture Notes in Computer Science, A. Abelló, P. Vassiliadis, O. Romero, and R. Wrembel, Eds., vol. 13985. Springer, 2023, pp. 45–58. [Online]. Available: https://doi.org/10.1007/978-3-031-42914-9_4

[96] A. Viggiano, "Redis-roaring," https://github.com/aviggiano/redis-roaring, webpage; Accessed: 2024-08-30.

[97] Y. Cheng and F. Rusu, "Parallel in-situ data processing with speculative loading," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1287–1298. [Online]. Available: https://doi.org/10.1145/2588555.2593673

[98] E. Z. Gnimpieba, M. S. VanDiermen, S. M. Gustafson, B. Conn, C. M. Lushbough *et al.*, "Bio-TDS: bioscience query tool discovery system," *Nucleic Acids Research*, vol. 45, no. D1, pp. D1117–D1122, 10 2016. [Online]. Available: https://doi.org/10.1093/nar/gkw940

[99] "Built-in Functions, any() - Python Documentation," Sep. 2023, [Webpage ; accessed 20. Sep. 2023]. [Online]. Available: https://docs.python.org/3/library/functions.html#any

[100] C. Anderson, "Docker [software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[101] "DataPLANT documentation - ARC Commander," 09 2023, [Webpage; accessed 20. Sep. 2023]. [Online]. Available: https://www.nfdi4plants.de/nfdi4plants.knowledgebase/docs/implementation/ArcCommander.html

[102] "TPC-H Homepage," Sep. 2023, [Webpage; accessed 17. Sep. 2023]. [Online]. Available: https://www.tpc.org/tpch

[103] "IMDb Non-Commercial Datasets," Sep. 2023, [Webpage; accessed 17. Sep. 2023]. [Online]. Available: https://developer.imdb.com/non-commercial-datasets

[104] J. Lee, D. J. Scott, M. Villarroel, G. D. Clifford, M. Saeed, R. G. Mark *et al.*, "Open-access MIMIC-II Database for Intensive Care Research," *Conference proceedings : ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual Conference*, vol. 2011, p. 8315, 2011.

[105] F. M. Harper and J. A. Konstan, "The MovieLens Datasets: History and Context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 1–19, Dec. 2015.

[106] T. J. Skluzacek, R. Kumar, R. Chard, G. Harrison, P. Beckman, K. Chard, and I. T. Foster, "Skluma: An extensible metadata extraction pipeline for disorganized data," in *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, 2018, pp. 256–266.

[107] Apache Software Foundation, "Nifi." [Online]. Available: https://nifi.apache.org/

[108] "Apache nifi documentation," https://nifi.apache.org/docs/nifi-docs/html/developer-guide.html#flowfile, [Wepbage ; Accessed on: 2023-9-13].

[109] P. Belmann, B. Fischer, J. Krüger, M. Procházka, H. Rasche, M. Prinz, M. Hanussek, M. Lang, F. Bartusch, B. Gläßle, J. Krüger, A. Pühler, and A. Sczyrba, "de.NBI Cloud federation through ELIXIR AAI [version 1; peer review: 2 approved, 1 not approved] ," *F1000Research*, vol. 8, no. 842, 2019.

[110] G. Doniparthi, T. Mühlhaus, and S. Deßloch, "Integrating FAIR experimental metadata for multi-omics data analysis," *Datenbank-Spektrum*, vol. 24, no. 2, pp. 107–115, 2024. [Online]. Available: https://doi.org/10.1007/s13222-024-00473-6

[111] Gajendra Doniparthi and Timo Mühlhaus and Stefan Deßloch, "A Bloom Filter-Based Framework for Interactive Exploration of Large-Scale Research Data," in *New Trends in Databases and Information Systems - ADBIS 2020 Short Papers, Lyon, France, August 25-27, 2020, Proceedings*, ser. Communications in Computer and Information Science, J. Darmont, B. Novikov, and R. Wrembel, Eds., vol. 1259. Springer, 2020, pp. 166–176. [Online]. Available: https://doi.org/10.1007/978-3-030-54623-6_15

[112] J. Bauer, M. Tschöpe, J. Weidhase, T. Mühlhaus, C. Garth, G. Doniparthi, H. Gauza, L. Perelo, C. M. Rodrigues, and D. von Suchodoletz, "From DataPLANT's DataHUB to DataPUB (lication)," in *International Workshop on Science Gateways*, 2023.

[113] G. A. Schreiner, D. Duarte, and R. dos Santos Mello, "When relational-based applications go to nosql databases: A survey," *Information*, vol. 10, no. 7, p. 241, 2019. [Online]. Available: https://doi.org/10.3390/info10070241

[114] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019. [Online]. Available: https://doi.org/10.1109/COMST.2018.2889329

[115] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, Eds. ACM, 2007, pp. 411–422. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf

[116] G. Luo and L. J. Frey, "Efficient execution methods of pivoting for bulk extraction of entity-attribute-value-modeled data," *IEEE J. Biomedical and Health Informatics*, vol. 20, no. 2, pp. 644–654, 2016. [Online]. Available: https://doi.org/10.1109/JBHI.2015.2392553

[117] Z. H. Liu, B. C. Hammerschmidt, and D. McMahon, "JSON data management: supporting schema-less development in RDBMS," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 1247–1258. [Online]. Available: https://doi.org/10.1145/2588555.2595628

[118] C. Chasseur, Y. Li, and J. M. Patel, "Enabling JSON document stores in relational systems," in *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23, 2013*, A. Bonifati and C. Yu, Eds., 2013, pp. 1–6. [Online]. Available: http://webdb2013.lille.inria.fr/Paper%2010.pdf

[119] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton, "Extending rdbmss to support sparse datasets using an interpreted attribute storage format," in *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, L. Liu, A. Reuter, K. Whang, and J. Zhang, Eds. IEEE Computer Society, 2006, p. 58. [Online]. Available: https://doi.org/10.1109/ICDE.2006.67

[120] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999. [Online]. Available: http://www.dcc.ufmg.br/irbook/

[121] P. G. Selfridge, D. Srivastava, and L. O. Wilson, "IDEA: interactive data exploration and analysis," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 1996, pp. 24–34. [Online]. Available: https://doi.org/10.1145/233269.233315

[122] C. Mishra and N. Koudas, "Interactive query refinement," in *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, ser. ACM International Conference Proceeding Series, M. L. Kersten, B. Novikov, J. Teubner, V. Polutin, and S. Manegold, Eds., vol. 360. ACM, 2009, pp. 862–873. [Online]. Available: https://doi.org/10.1145/1516360.1516459

[123] E. J. Otoo and K. Wu, "Accelerating queries on very large datasets," in *Scientific Data Management - Challenges, Technology, and Deployment*, ser. Chapman and Hall / CRC computational science series. CRC Press / Taylor & Francis, 2009. [Online]. Available: https://doi.org/10.1201/9781420069815-c6

[124] G. Doniparthi, T. Mühlhaus, and S. Deßloch, "A bloom filter-based framework for interactive exploration of large scale research data," in *New Trends in Databases and Information Systems - ADBIS 2020 Short Papers, Lyon, France, August 25-27, 2020, Proceedings*, ser. Communications in Computer and Information Science, vol. 1259. Springer, 2020, pp. 166–176. [Online]. Available: https://doi.org/10.1007/978-3-030-54623-6_15

[125] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[126] Y. Nti-Addae, D. Matthews, V. J. M. Ulat, R. Syed, G. Sempéré, A. Pétel, J. Renner, P. Larmande, V. Guignon, E. Jones, and K. Robbins, "Benchmarking database systems for genomic selection implementation," *Database J. Biol. Databases Curation*, vol. 2019, p. baz096, 2019. [Online]. Available: https://doi.org/10.1093/database/baz096

[127] A. Celesti, M. Fazio, and M. Villari, "A study on join operations in mongodb preserving collections data models for future internet applications," *Future Internet*, vol. 11, no. 4, 2019. [Online]. Available: https://www.mdpi.com/1999-5903/11/4/83

[128] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "Scidb: A database management system for applications with complex analytics," *Computing in Science Engineering*, vol. 15, no. 3, pp. 54–62, 2013.

[129] V. Gadepally, K. O'Brien, A. Dziedzic, A. Elmore, J. Kepner, S. Madden, T. Mattson, J. Rogers, Z. She, and M. Stonebraker, "Bigdawg version 0.1," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[130] B. B. Misra, C. Langefeld, M. Olivier, and L. A. Cox, "Integrated omics: tools, advances and future approaches," *Journal of Molecular Endocrinology*, vol. 62, no. 1, pp. R21 – R45, 2019. [Online]. Available: https://jme.bioscientifica.com/view/journals/jme/62/1/JME-18-0055.xml

[131] W. Zhang, F. Li, and L. Nie, "Integrating multiple 'omics' analysis for microbial biology: application and methodologies," *Microbiology*, vol. 156, no. 2, pp. 287–301, 2010.

[132] A. Davoudian and M. Liu, "Big data systems: A software engineering perspective," *ACM Comput. Surv.*, vol. 53, no. 5, pp. 110:1–110:39, 2020. [Online]. Available: https://doi.org/10.1145/3408314

[133] K. Yu, V. Gadepally, and M. Stonebraker, "Database engine integration and performance analysis of the bigdawg polystore system," in *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*. IEEE, 2017, pp. 1–7. [Online]. Available: https://doi.org/10.1109/HPEC.2017.8091081

[134] J. Lu, I. Holubová, and B. Cautis, "Multi-model databases and tightly integrated polystores: Current practices, comparisons, and open challenges," ser. CIKM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2301–2302. [Online]. Available: https://doi.org/10.1145/3269206.3274269

[135] W. L. Schulz, B. G. Nelson, D. K. Felker, T. J. S. Durant, and R. Torres, "Evaluation of relational and nosql database architectures to manage genomic annotations," *J. Biomed. Informatics*, vol. 64, pp. 288–295, 2016. [Online]. Available: https://doi.org/10.1016/j.jbi.2016.10.015

[136] R. Lawrence, "Faster querying for database integration and virtualization with distributed semi-joins," in *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2017, pp. 1406–1410.

[137] L. G. Azevedo, E. F. de Souza Soares, R. Souza, and M. F. Moreno, "Modern federated database systems: An overview," in *Proceedings of the 22nd International Conference on Enterprise Information Systems, ICEIS 2020, Prague, Czech Republic, May 5-7, 2020, Volume 1*, J. Filipe, M. Smialek, A. Brodsky, and S. Hammoudi, Eds. SCITEPRESS, 2020, pp. 276–283. [Online]. Available: https://doi.org/10.5220/0009795402760283

[138] A. R. Joyce and B. Ø. Palsson, "The model organism as a system: integrating 'omics' data sets," *Nature Reviews Molecular Cell Biology*, vol. 7, no. 3, pp. 198–210, Mar 2006. [Online]. Available: https://doi.org/10.1038/nrm1857

[139] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Adaptive partitioning and indexing for in situ query processing," *VLDB J.*, vol. 29, no. 1, pp. 569–591, 2020. [Online]. Available: https://doi.org/10.1007/s00778-019-00580-x

[140] Y. Hasin, M. Seldin, and A. Lusis, "Multi-omics approaches to disease," *Genome Biology*, vol. 18, no. 1, p. 83, May 2017. [Online]. Available: https://doi.org/10.1186/s13059-017-1215-1

[141] P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Commun. ACM*, vol. 51, no. 9, pp. 72–79, 2008. [Online]. Available: https://doi.org/10.1145/1378727.1378745

[142] *E-Science-Tage 2021: Share Your Research Data.* heiBOOKS, Apr. 2022. [Online]. Available: https://books.ub.uni-heidelberg.de/heibooks/catalog/book/979

[143] C. Capitanchik, S. Ireland, A. Harston, C. Cheshire, D. M. Jones, F. C. Lee, I. R. de los Mozos, I. A. Iosub, K. Kuret, R. Faraway, O. G. Wilkins, R. Arora, M. Hallegger, M. Modic, A. M. Chakrabarti, N. M. Luscombe, and J. Ule, "Flow: a web platform and open database to analyse, store, curate and share bioinformatics data at scale," *bioRxiv*, 2023. [Online]. Available: https://www.biorxiv.org/content/early/2023/08/29/2023.08.22.544179

[144] K. Haug, R. M. Salek, and C. Steinbeck, "Global open data management in metabolomics," *Current Opinion in Chemical Biology*, vol. 36, pp. 58–63, 2017, omics. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1367593116302083

[145] M. Krantz, D. Zimmer, S. O. Adler, A. Kitashova, E. Klipp, T. Mühlhaus, and T. Nägele, "Data Management and Modeling in Plant Biology," *Frontiers in Plant Science*, vol. 12, 2021. [Online]. Available: https://www.frontiersin.org/journals/plant-science/articles/10.3389/fpls.2021.717958

[146] N. Rettberg and B. Schmidt, "OpenAIRE - Building a collaborative Open Access infrastructure for European researchers," *LIBER Quarterly: The Journal of the Association of European Research Libraries*, vol. 22, no. 3, p. 160–175, Nov. 2012. [Online]. Available: https://liberquarterly.eu/article/view/10641

[147] R. N. Smith, J. Aleksic, D. Butano, A. Carr, S. Contrino, F. Hu, M. Lyne, R. Lyne, A. Kalderimis, K. Rutherford, R. Stepan, J. Sullivan, M. Wakeling, X. Watkins, and G. Micklem, "InterMine: a flexible data warehouse system for the integration and analysis of heterogeneous biological data," *Bioinformatics*, vol. 28, no. 23, pp. 3163–3165, Sep. 2012.

[148] S. Ho Sui, E. Merrill, N. Gehlenborg, P. Haseley, I. Sytchev, R. Park, P. Rocca-Serra, S. Corlosquet, A. Gonzalez-Beltran, E. Maguire, O. Hofmann, P. Park, S. Das, S.-A. Sansone, and W. Hide, "The Stem Cell Commons: an exemplar for data integration in the biomedical domain driven by the ISA framework," *AMIA Jt Summits Transl Sci Proc*, vol. 2013, p. 70, Mar. 2013.

[149] S. J. Ho Sui, K. Begley, D. Reilly, B. Chapman, R. McGovern, P. Rocca-Sera, E. Maguire, G. M. Altschuler, T. A. A. Hansen, R. Sompallae, A. Krivtsov, R. A. Shivdasani, S. A. Armstrong, A. C. Culhane, M. Correll, S.-A. Sansone, O. Hofmann, and W. Hide, "The Stem Cell Discovery Engine: an integrated repository and analysis system for cancer stem cell comparisons," *Nucleic Acids Res*, vol. 40, no. Database issue, pp. D984–91, Nov. 2011.

[150] A. González-Beltrán, P. Li, J. Zhao, M. S. Avila-Garcia, M. Roos, M. Thompson, E. van der Horst, R. Kaliyaperumal, R. Luo, T.-L. Lee, T.-W. Lam, S. C. Edmunds, S.-A. Sansone, and P. Rocca-Serra, "From Peer-Reviewed to Peer-Reproduced in Scholarly Publishing: The Complementary Roles of Data Models and Workflows in Bioinformatics," *PLoS One*, vol. 10, no. 7, p. e0127612, Jul. 2015.

[151] S.-A. Sansone, P. McQuilton, P. Rocca-Serra, A. Gonzalez-Beltran, M. Izzo, A. L. Lister, M. Thurston, and the FAIRsharing Community, "FAIRsharing as a community approach to standards, repositories and policies," *Nature Biotechnology*, vol. 37, no. 4, pp. 358–367, Apr 2019. [Online]. Available: https://doi.org/10.1038/s41587-019-0080-8

[152] C. L. Borgman, "The conundrum of sharing research data," *Journal of the American Society for Information Science and Technology*, vol. 63, no. 6, pp. 1059–1078, 2012. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.22634

[153] L. Bornmann and R. Mutz, "Growth rates of modern science: A bibliometric analysis based on the number of publications and cited references," *Journal of the Association for Information Science and Technology*, vol. 66, no. 11, pp. 2215–2222, 2015. [Online]. Available: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.23329

[154] F. Zhu, W. Wen, Y. Cheng, S. Alseekh, and A. R. Fernie, "Integrating multiomics data accelerates elucidation of plant primary and secondary

metabolic pathways," *aBIOTECH*, vol. 4, no. 1, pp. 47–56, Mar 2023. [Online]. Available: https://doi.org/10.1007/s42994-022-00091-4

[155] I. D. Dinov, "Volume and value of big healthcare data," vol. 4, no. 1, p. 3. [Online]. Available: http://www.hoajonline.com/medicalstat/2053-7662/4/3

[156] S. Wang, D. Maier, and B. C. Ooi, "Fast and adaptive indexing of multi-dimensional observational data," vol. 9, no. 14, pp. 1683–1694. [Online]. Available: https://dl.acm.org/doi/10.14778/3007328.3007334

[157] A. Doufene and D. Krob, "Pareto optimality and nash equilibrium for building stable systems," in *2015 Annual IEEE Systems Conference (SysCon) Proceedings*. IEEE, pp. 542–545. [Online]. Available: http://ieeexplore.ieee.org/document/7116808/

[158] D. Michie, "GAME-PLAYING AND GAME-LEARNING AUTOMATA," in *Advances in Programming and Non-Numerical Computation*. Elsevier, pp. 183–200. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780080113562500112

[159] D. Fudenberg and J. Tirole, "Game theory," *Cambridge, MA*, vol. 86, 1991.

[160] M. Shor. Dictionary of game theory terms. [Online]. Available: https://www.gametheory.net/dictionary

[161] M. R. Frank, E. R. Omiecinski, and S. B. Navathe, "Adaptive and automated index selection in RDBMS," in *Advances in Database Technology — EDBT '92*, A. Pirotte, C. Delobel, and G. Gottlob, Eds. Springer-Verlag, vol. 580, pp. 277–292, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/BFb0032437

[162] S. Huang, K. Chaudhary, and L. X. Garmire, "More Is Better: Recent Progress in Multi-Omics Data Integration Methods," *Frontiers in Genetics*, vol. 8, 2017.

[163] J. Yan, S. L. Risacher, L. Shen, and A. J. Saykin, "Network approaches to systems biology analysis of complex disease: integrative methods for multi-omics data," *Briefings in Bioinformatics*, vol. 19, no. 6, pp. 1370–1381, 06 2017. [Online]. Available: https://doi.org/10.1093/bib/bbx066

[164] Akhmedov, Murodzhon and Martinelli, Axel and Geiger, Roger and Kwee, Ivo, "Omics Playground: a comprehensive self-service platform for visualization, analytics and exploration of Big Omics Data," *NAR Genomics and Bioinformatics*, vol. 2, no. 1, p. lqz019, 12 2019. [Online]. Available: https://doi.org/10.1093/nargab/lqz019

[165] Bernasconi, Anna and Canakoglu, Arif and Masseroli, Marco and Ceri, Stefano, "META-BASE: A Novel Architecture for Large-Scale Genomic Metadata Integration," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 19, no. 1, pp. 543–557, 2022.

[166] Dai, Chengxin and Füllgrabe, Anja and Pfeuffer, Julianus and Solovyeva, Elizaveta M and Deng, Jingwen and Moreno, Pablo and Kamatchinathan,

Selvakumar and Kundu, Deepti Jaiswal and George, Nancy and Fexova, Silvie and others, "A proteomics sample metadata representation for multiomics integration and big data analysis," *Nature Communications*, vol. 12, no. 1, p. 5854, 2021.

[167] Neuweger, Heiko and Albaum, Stefan P. and Dondrup, Michael and Persicke, Marcus and Watt, Tony and Niehaus, Karsten and Stoye, Jens and Goesmann, Alexander, "MeltDB: a software platform for the analysis and integration of metabolomics experiment data," *Bioinformatics*, vol. 24, no. 23, pp. 2726–2732, 09 2008. [Online]. Available: https://doi.org/10.1093/bioinformatics/btn452

[168] Kuo, Tien-Chueh and Tian, Tze-Feng and Tseng, Yufeng Jane, "3Omics: a web-based systems biology tool for analysis, integration and visualization of human transcriptomic, proteomic and metabolomic data," *BMC systems biology*, vol. 7, pp. 1–15, 2013.

[169] Perez-Riverol, Yasset and Zorin, Andrey and Dass, Gaurhari and Vu, Manh-Tu and Xu, Pan and Glont, Mihai and Vizcaíno, Juan Antonio and Jarnuczak, Andrew F and Petryszak, Robert and Ping, Peipei and others, "Quantifying the impact of public omics data," *Nature communications*, vol. 10, no. 1, p. 3512, 2019.

[170] G. Doniparthi, T. Mühlhaus, and S. Deßloch, "Integrating FAIR Experimental Metadata for Multi-omics Data Analysis," *Datenbank-Spektrum*, pp. 1–9, 2024.

[171] S. Bechhofer, I. Buchan, D. De Roure, P. Missier, J. Ainsworth, J. Bhagat, P. Couch, D. Cruickshank, M. Delderfield, I. Dunlop, M. Gamble, D. Michaelides, S. Owen, D. Newman, S. Sufi, and C. Goble, "Why linked data is not enough for scientists," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 599–611, 2013, special section: Recent advances in e-Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X11001439

[172] M. Stonebraker, "Technical perspective - one size fits all: an idea whose time has come and gone," *Commun. ACM*, vol. 51, no. 12, p. 76, 2008. [Online]. Available: https://doi.org/10.1145/1409360.1409379

[173] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[174] P. Fawaz, S. Challita, Y. al dhuraibi, and P. Merle, "Model-driven management of docker containers," 07 2016.

[175] "Isa-tools specification," https://isa-tools.org/format/specification.html, [Webpage; Accessed on: 2023-9-12].

[176] "High-level overview of docker architecture," https://www.researchgate.net/figure/High-level-overview-of-Docker-architecture_fig1_308050257, [Webpage; Accessed on: 2023-09-12].

[177] "Docker Inc. docker images and containers," https://docs.docker.com/get-started/, webpage; Accessed on: 2023-09-11.

[178] "Docker hub," https://hub.docker.com/, webpage; Accessed on 2023-9-11.

[179] C. Quix, R. Hai, and I. Vatov, "Gemms: A generic and extensible metadata management system for data lakes." in *CAiSE forum*, vol. 129, 2016.

[180] M. Poess and R. Nambiar, "TPC Benchmark H Standard Specification," *ResearchGate*, Jan. 2010.

[181] M. Saeed, C. Lieu, G. Raber, and R. Mark, "Mimic ii: a massive temporal icu patient database to support research in intelligent patient monitoring," in *Computers in Cardiology*, 2002, pp. 641–644.

[182] S. Triesch, A. K. Denton, J. P. Buchmann, V. Reichel-Deland, U. Schlüter, and A. P. Weber, "Transposable elements contribute to the establishment of the glycine shuttle in brassicaceae species," *bioRxiv*, 2022. [Online]. Available: https://www.biorxiv.org/content/early/2022/12/09/2022.12.06.519256

[183] B. Spaniol, J. Lang, B. Venn, L. Schake, F. Sommer, M. Mustas, S. Geimer, F.-A. Wollman, Y. Choquet, T. Mühlhaus, and M. Schroda, "Complexome profiling on the Chlamydomonas lpa2 mutant reveals insights into PSII biogenesis and new PSII associated proteins," *Journal of Experimental Botany*, vol. 73, no. 1, pp. 245–262, 08 2021. [Online]. Available: https://doi.org/10.1093/jxb/erab390

[184] S. Schweikert, A. Kranz, T. Yakushi, A. Filipchyk, T. Polen, H. Etterich, S. Bringer, and M. Bott, "FNR-Type Regulator GoxR of the Obligatorily Aerobic Acetic Acid Bacterium Gluconobacter oxydans Affects Expression of Genes Involved in Respiration and Redox Metabolism," *Appl. Environ. Microbiol.*, vol. 87, no. 11, Jun. 2021.

[185] B. Venn., "Lpa2 complexome profiling," https://git.nfdi4plants.org/venn/LPA2_ComplexomeProfiling_Chlamy, 2023.

[186] S. Saurin, M. Meineck, M. Rohr, W. Roth, T. Opatz, G. Erkel, A. Pautz, and J. Weinmann-Menke, "The macrocyclic lactone oxacyclododecindione reduces fibrosis progression," *Front. Pharmacol.*, vol. 14, p. 1200164, Jun. 2023.

[187] B. Venn, "The macrocyclic lactone oxacyclododecindione reduces fibrosis progression," https://git.nfdi4plants.org/venn/Antifibrotic-effects-of-Oxa, 2023.

[188] K. Wolstencroft, S. Owen, O. Krebs, Q. Nguyen, N. J. Stanford, M. Golebiewski, A. Weidemann, M. Bittkowski, L. An, D. Shockley, J. L. Snoep, W. Mueller, and C. Goble, "SEEK: a systems biology data and model management platform," *BMC Syst. Biol.*, vol. 9, no. 1, pp. 1–12, Dec. 2015.

[189] "nfdi4plants - Annotated Research Context ," Sep. 2023, [Webpage; accessed 18. Sep. 2023]. [Online]. Available: https://nfdi4plants.org/content/learn-more/annotated-research-context.html

[190] N. Sewelam, D. Brilhaus, A. Bräutigam, S. Alseekh, A. R. Fernie, and V. G. Maurino, "Molecular plant responses to combined abiotic stresses put a spotlight on unknown and abundant genes," *Journal of*

Experimental Botany, vol. 71, no. 16, pp. 5098–5112, 05 2020. [Online]. Available: https://doi.org/10.1093/jxb/eraa250

[191] J. A. Novella, P. Emami Khoonsari, S. Herman, D. Whitenack, M. Capuccini, J. Burman, K. Kultima, O. Spjuth *et al.*, "Container-based bioinformatics with Pachyderm," *Bioinformatics*, vol. 35, no. 5, pp. 839–846, 08 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty699

[192] B. Venn., "Reversible burst of transcriptional changes during induction of crassulacean acid metabolism in talinum triangulare," https://gitdev.nfdi4plants.org/doniparthi/samplearc_rnaseq, 2023.

[193] S. Triesch *et al.*, "Hhu plant biochemistry - brassicaceae transposons," https://git.nfdi4plants.org/hhu-plant-biochemistry/triesch2023_brassicaceae_transposons.git, 2023.

[194] H. Fang, "Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem," in *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, 2015, pp. 820–824.

[195] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Rec.*, vol. 26, no. 1, p. 65–74, mar 1997. [Online]. Available: https://doi.org/10.1145/248603.248616

[196] "Designing data marts for data warehouses," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 4, p. 452–483, oct 2001. [Online]. Available: https://doi.org/10.1145/384189.384190

[197] "Consortia DataPLANT | NFDI," Sep. 2023, [Webpage ; accessed 18. Sep. 2023]. [Online]. Available: https://www.nfdi.de/consortia-dataplant/?lang=en

[198] D. von Suchodoletz, T. Mühlhaus, J. Krüger, B. Usadel, and C. M. Rodrigues, "Dataplant–ein nfdi-konsortium der pflanzen-grundlagenforschung," *Bausteine Forschungsdatenmanagement*, no. 2, pp. 46–56, 2021.

[199] A. Gorelik, *Introduction to Data Lakes*. O'Reilly Media, Inc., p. 1–224.

[200] Embl-Ebi, "PRIDE - Proteomics Identification Database," Sep. 2023, [Online; accessed 20. Sep. 2023]. [Online]. Available: https://www.ebi.ac.uk/pride

[201] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

Gajendra Doniparthi

Education

2020 – 2025 **Doctoral Studies**, *RPTU Kaiserslautern Landau*
Thesis: An Approach to Data Integration and Explorative Query Processing in Scientific Data Management Platforms

2017 – 2019 **M.Sc. Computer Science**, *TU Kaiserslautern*
Thesis: CromixDB - A Cross-omics Data Management System

1999 – 2003 **Bachelors in Engineering in Computer Science & Engineering**, *University of Madras, Tamil Nadu, India*

Work Experience

2019–2025 **Research Associate**, *RPTU Kaiserslautern-Landau*
Employment within the framework of the doctorate.
DataPLANT Project.

2010–2017 **Staff Programmer Analyst**, *SanDisk Corporation*, India, and USA.

2003–2010 **Associate**, *Cognizant Technology Solutions, India, and USA.*

gajendra.doniparthi@cs.rptu.de