University of Louisville

ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

5-2024

Using dynamic schemas for query optimization over JSON data.

Tomas Felipe Llano-Rios
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Part of the Databases and Information Systems Commons

Recommended Citation

Llano-Rios, Tomas Felipe, "Using dynamic schemas for query optimization over JSON data." (2024). *Electronic Theses and Dissertations.* Paper 4308.
https://doi.org/10.18297/etd/4308

This Doctoral Dissertation is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

USING DYNAMIC SCHEMAS FOR QUERY OPTIMIZATION OVER JSON DATA

By

Tomás Felipe Llano-Ríos

MSc. in Computer Science, 2020

A Dissertation

Submitted to the Faculty of the

J.B. Speed School of Engineering of the University of Louisville

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in Computer Science and Engineering

Computer Science and Engineering

University of Louisville

Louisville, Kentucky

May 2024

Copyright 2024 by Tomás Felipe Llano-Ríos

All rights reserved

USING DYNAMIC SCHEMAS FOR QUERY OPTIMIZATION OVER JSON DATA

By

Tomás Felipe Llano-Ríos
MSc. in Computer Science, 2020

Dissertation approved on

April 19, 2024

by the following Dissertation Committee:

Antonio Badia

Mohamed Khalefa

Nihat Altiparmak

Hichem Frigui

ABSTRACT

USING DYNAMIC SCHEMAS FOR QUERY OPTIMIZATION OVER JSON DATA

Tomás Felipe Llano-Ríos

April 19, 2024

Query optimization in document stores has traditionally relied on rule-based approaches, but recent research advocates for a shift towards cost-based optimization. However, this transition is hindered by the fragmented nature of existing approaches, stemming from the early development stage of cost-based query optimization for document databases. A key challenge lies in the absence of a standardized query language and semantics, exacerbated by the diverse and schema-less nature of JSON document collections. To tackle these challenges, the literature has proposed dynamic schemas, primarily utilized at parsing time. However, these schemas lack a formal foundation that describes meaningful semantics for query optimization. This thesis proposes a novel framework based on a relational-like plan, employing an algebra to internally represent queries. By manipulating algebra expressions, multiple plans are generated and subsequently evaluated for cost. Specifically tailored to JSON data, the thesis introduces a document algebra designed to accommodate JSON characteristics. Additionally, it formalizes a dynamic schema concept termed Data Pilot, inspired by XML DataGuides. An algebra over Data Pilots is presented, facilitating cardinality estimation without executing operations, aiding in query optimization. Furthermore,

the thesis proposes a strategy to determine when query rewriting using Document Algebra properties may be advantageous. Experimental validation demonstrates the feasibility of the proposed framework and showcases the construction of Data Pilot structures. Through this research, a step towards standardized, cost-based query optimization in document stores is taken, paving the way for more efficient and scalable query processing in the future.

TABLE OF CONTENTS

Abstract iii

Chapter I Introduction 1

Chapter II Background 5

Introduction 5

Relational Databases 7

NoSQL Databases 14

Document Stores 26

Chapter III Proposed Approach 36

The Problem: Query Optimization 36

Proposed framework 75

Thesis contribution 81

Chapter IV Documents 83

Notation and terminology 83

JSON Document Algebra 93

JSON Document Algebra Properties 110

Chapter V Data Pilots 142

Collection summarization 142

Data Pilot creation and maintenance 153

Data Pilot algebra 159
Data Pilot algebra properties 191
Optimization hints 199

Chapter VI Experimental Analysis 214
Chapter Overview 214
Proof of Concept: Data Pilot Creation 219
Proof of Concept: Query Optimization Framework 223

Chapter VII Related research 237
Introduction 237
JSON Data Model 237
DataPilot and Query Optimization 246

REFERENCES 257

Curriculum Vita 267

CHAPTER I

INTRODUCTION

In the ever-evolving realm of data management, traditional relational database systems have long held a position as the industry's gold standard. Nevertheless, the limitations of these systems in handling the intricacies of modern data have become glaringly evident, as they grapple with the relentless surge in data volume, diversity, and velocity.

To address these formidable limitations, two distinct categories of solutions have emerged over the years:

Specialized Database Systems These systems are finely tuned to cater to specific data types, with document databases, such as MongoDB[1] and Couchbase[2], leading the charge. Additionally, vector databases like Elasticsearch's vector database[3], time series databases such as InfluxDB[4] and Prometheus[5], and graph databases like Neo4j[6], have become prominent players in this landscape.

Multi-Paradigm Database Systems This category encompasses databases that embrace a multi-faceted approach, incorporating support for various data types. Notably, within the realm of relational databases, there have been groundbreaking pro-

[1] https://www.mongodb.com/
[2] https://www.couchbase.com/
[3] https://www.elastic.co/elasticsearch/vector-database
[4] https://www.influxdata.com/
[5] https://prometheus.io/
[6] https://neo4j.com/

posals such as SQL/JSON, enabling the storage and querying of JSON data using SQL within a relational database management system (RDBMS). More recently, SQL/PGQ has emerged, extending SQL to incorporate graph pattern matching capabilities into existing RDBMSs, a proposal put forth by the SQL standardization committee.

Remarkably, the lines between specialized and multi-paradigm systems have begun to blur, with specialized systems evolving to support diverse data types. For instance, MongoDB has introduced specialized storage support for time series data. In parallel, established relational systems like PostgreSQL and MariaDB offer extensibility via plugins and various storage engines. For instance, MariaDB introduced a column storage engine with a massively parallel processing shared-nothing architecture, facilitating distributed data warehousing, a feat that traditional relational systems have grappled with for years.

However, it is essential to recognize that these solutions are not without their trade-offs. Specialized systems often encounter scenarios where users require features from other data paradigms. For example, document databases excel in storing semi-structured data in JSON format, avoiding the need for a normalized schema found in traditional relational databases. Users can embed data within a single record that would typically reside in separate tables in a relational database. This simplifies and expedites data read and write operations by eliminating the necessity for transforming semi-structured data into a structured format and eradicating the need for complex joins.

MongoDB's evolution is illustrative; it initially lacked an operator analogous to relational joins, as users were encouraged to perform in-depth analyses in an OLAP database and utilize MongoDB exclusively for OLTP. However, the burgeoning demand for data analytics and OLAP use cases necessitated the addition of a join-like operator and the development of a comprehensive query optimizer. Similarly,

Couchbase, another document database, initially lacked a query optimizer capable of handling OLAP queries but later introduced a relational-like cost-based query optimizer[7].

Conversely, multi-paradigm databases often grapple with integrating common features across data paradigms, such as query optimization and statistics collection. For instance, PostgreSQL faced criticism after introducing JSON support due to the lack of statistics and optimization capabilities for this data type[77].

In many instances, the integration of these features becomes challenging due to the relative novelty of the paradigm and its original intended use case. An illustrative case in point is the realm of NoSQL document stores. These systems, beyond their proficiency in handling semi-structured data compared to relational databases, possess the unique trait of requiring no upfront design efforts. They are often architected from the ground up for distribution, a characteristic shared by many other NoSQL systems. This inherent feature allows document stores to be rapidly deployed, particularly in environments that do not favor centralized architectures. Consequently, these systems, along with other members of the NoSQL family, present themselves as viable alternatives to traditional RDBMSs for specific problem domains[32].

However, to achieve some of these advantages, NoSQL systems often find themselves compelled to deviate from the traditional relational model[68]. This deviation can entail limitations on optimization or require the management of data in superset formats such as JSON, a domain where research is still maturing. These trade-offs, naturally, carry over to Multi-Paradigm systems that incorporate these paradigms.

This thesis centers its focus on the JSON document model. JSON has not only gained prominence as one of the most prevalent data formats in recent years, thanks to its widespread use in web environments and data interchange, but also presents ample room for improvement within the scope of database research. Specifically, it offers promising avenues for enhancement in the domain of query optimization.

[7] https://www.couchbase.com/blog/cost-based-optimizer-for-couchbase-n1ql-sql-for-json/

In the forthcoming chapters, Chapter II will commence by providing essential background information about NoSQL databases in a general context. This chapter will introduce various common data models and query languages while delving into the specific nuances associated with document databases. It will also address the typical challenges encountered when optimizing queries over JSON data for two leading (formerly) specialized systems, MongoDB and Couchbase, as well as for one of the most prominent multi-paradigm RDBMSs, PostgreSQL. To illustrate and analyze the complexities involved in query optimization for JSON, an experimental comparison will be conducted.

Following that, Chapter III will outline the proposed approach to query optimization. It will dissect each component of a theoretical cost-based optimization framework designed for the JSON document model, elucidating the rationale behind its inclusion in the framework.

Chapter V will delve into a theoretical data structure known as a Data Pilot and its pivotal role in query optimization, particularly when indexes are either unusable or unavailable.

Chapter VI will present the results of a simulation of the approach proposed in Chapter III, substantiating its feasibility through empirical evidence.

Lastly, Chapter VII will provide an exploration of research related to the Data Pilot structure, the proposed cost-based query optimization approach, and theoretical formalizations pertinent to the JSON model. This chapter will also offer insights into avenues for future research.

CHAPTER II

BACKGROUND

1 Introduction

The term “NoSQL” can be perplexing for readers due to its various interpretations over the years [51][68]. Thus, it is imperative to establish a precise definition within the context of this thesis. Sharp-eyed readers may have observed its casual usage in the previous chapter without prior clarification. This was intentional because the interpretations of “NoSQL” that were implied in Chapter I encompass all the systems discussed.

Originating back to 1998, Carlo Strozzi coined the term "NoSQL" for his relational database, which is still available for download[1]. Unlike conventional databases, this system didn’t employ SQL queries; instead, it used shell commands inspired by the UNIX file structure, where commands resembled UNIX utilities. For example, to create a table, a user would first craft a template in a text file (e.g., `template.tpl`), specifying the table’s column names and descriptions. Subsequently, the user would deploy commands like `nosql maketable < template.tpl | nosql write -s table.rdb` to create the new table as a file. Here, `nosql write` functioned akin to the UNIX `tee` utility, with the `-s` option suppressing STDOUT. In this instance, “NoSQL” pertained to a specific system rather than a category, implying that a NoSQL database referred to any database devoid of a SQL query interface, irrespective of whether it adhered to the relational model or not.

[1] `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home%20Page`

Conversely, when Eric Evans reintroduced the term in 2009, it referred to the burgeoning array of specialized, non-relational distributed databases of that era[51]. Under this interpretation, a NoSQL database was one specialized in managing non-relational data, signifying a departure from the relational model but not necessarily from the SQL language itself. Accordingly, a NoSQL database, under this definition, could offer a variety of query interfaces, including SQL. This interpretation is now commonly referred to as "Not only SQL".

While the second interpretation has gained widespread acceptance, it still occasionally leads to confusion:

- From an implementation perspective, relational database management systems (RDBMSs) are assumed to store table records row-wise, but they can also store them column-wise, a fundamental principle of column-oriented databases. While the relational model is conceptually independent of implementation specifics, it is often associated with row-wise data storage. Consequently, column-oriented databases are sometimes categorized as NoSQL databases, even though their theoretical foundation remains relational.

- Systems that initially operated as RDBMSs are progressively evolving into multi-paradigm databases as they adopt the new SQL standard and incorporate JSON data. Thus, systems like PostgreSQL or MariaDB, which already support JSON, could be seen as "Not only SQL" systems. Nonetheless, they are still primarily regarded as relational databases.

Within the context of this thesis, a NoSQL system aligns with the "Not only SQL" interpretation but also maintains a history of being non-relational and lacking SQL support from its inception. Therefore, systems traditionally classified as RDBMSs, such as MariaDB and PostgreSQL, retain their classification in this thesis, even as they incorporate features from non-relational systems.

This chapter serves as an introduction to contemporary NoSQL data paradigms and the popular query languages associated with the data types they handle. It also delves into the transformative journey of JSON document stores, as they evolve into OLAP-capable systems, shedding light on the myriad challenges they encounter. Of particular focus is the central issue that forms the core of this thesis.

Special emphasis is placed on the structural aspects of the data these databases store and how these structural nuances significantly impact the key metrics used to assess their performance in specific use cases, which are explored within this thesis.

To provide readers with a solid foundation, Section II.2 offers a concise overview of pertinent information from relational databases. This overview serves as a baseline, enabling readers to discern the primary motivations and differentiating factors between NoSQL systems and their relational counterparts.

Section II.3 dives into various NoSQL paradigms, elucidating their core features and data management approaches. This section provides essential context for understanding the diversity and nuances of NoSQL databases.

Section II.4 takes a deep dive into document databases, offering a comprehensive understanding of their unique characteristics and addressing the specific challenges they encounter when tasked with data analytics responsibilities.

Lastly, Section III.1, elucidates on the central problem that forms the focal point of this thesis. This section is complemented by a series of illustrative experiments that substantiate and reinforce the core issue at hand.

2 Relational Databases

Before the relational model, data management was characterized by the use of network databases, which essentially comprised a collection of interconnected records, or hierarchical databases, where data resided in tree-like structures, with each record connected to one or more parent records. In these models, programmers were tasked with determining the navigational paths between various entities during the database

design phase. Think of it as envisioning a directed graph, with entities as nodes. Consequently, if a particular navigation path was absent, programmers were compelled to either overhaul the entire database design or implement query logic at the application level. This phenomenon, aptly named "path dependency" by [21], gave rise to three primary challenges:

- **Hierarchical Ambiguity:** Data represented in tree or network structures could assume multiple hierarchies. Deciding which hierarchy to adopt was far from straightforward, as certain queries would require significantly more effort in some hierarchies than in others.
- **Access Path Reliance:** Applications were heavily reliant on specific access paths to reach the data, making them susceptible to disruptions if the structure of the data changed.
- **Redundancy:** In hierarchical systems, particularly when many elements of one entity were related to many elements of another (and vice versa), hierarchies often necessitated the storage of multiple redundant copies of certain data.

The inception of the relational model was driven by the need for efficient data storage that minimized redundancy. This was of paramount importance at a time when storage resources were scarce and costly. Storing each piece of data just once translated to significant cost reductions in an era when storage was at a premium. [21, 69, 3].

Furthermore, the relational model was meticulously crafted with a paramount focus on data independence. This design principle empowered developers to write queries in a declarative manner, effectively shielding them from the labyrinthine intricacies of low-level implementation and performance concerns. This was in stark contrast to the procedural query construction necessary in network and hierarchical databases.

These remarkable achievements were made possible through the introduction of a fundamental concept known as the "schema" [1]. This schema serves as the archi-

tectural blueprint for a database, defining its structure at two essential levels: the logical and the physical.

The logical schema provides a high-level, conceptual framework for defining data structure. It outlines how data is modeled into tables, specifying their attributes and the intricate relationships between them. In more precise terms, it articulates that within a relational data perspective, a relation (in the mathematical sense) denoted as R is a set of n-tuples, where the first element of each tuple belongs to set S_1, the second to S_2, and so forth. In essence, this translates to R being a subset of $S_1 \times S_2 \times \cdots \times S_n$. When visually represented as a table, each row embodies one of these n-tuples, rendering their order inconsequential (in the same manner as tuples), and necessitating that every row remains distinct (akin to each tuple). Each column, in turn, embodies one of the domains, such as S_i, maintaining alignment with the ordering $S_1, S_2, \ldots S_n$ of R's domains.

Conversely, the physical schema plunges into the granular depths of data storage, encompassing both disk and memory configurations, indexing mechanisms, and optimization strategies, including the vital realm of statistics. It is within this schema layer that all performance and efficiency-related intricacies come to fruition.

The concept of a schema has wielded and continues to wield profound implications for both database theory and the development of database applications.

The logical schema enables applications to seamlessly manipulate data through operations designed over the relational model. These operations, capable of deriving relations from other relations, provide a powerful and abstract means of interacting with data. Crucially, any low-level data management concerns are deftly handled by the database system at the physical level. This means that a logical operation can find myriad physical implementations, with the system selecting the most suitable one based on the specific query at hand. Notably, modifications at the physical level, whether involving data representation on disk or the intricate implementations of relational operators, have no bearing on how queries are formulated.

The division of the schema into logical and physical tiers has been instrumental for researchers, affording them the opportunity to delve into each level independently [1]. At the logical level, database theory researchers have dedicated substantial effort to the exploration of query languages. They have scrutinized their complexities and expressive capabilities, forging close ties with complexity theory and logic—a trifecta of intertwined subjects. Noteworthy among these languages are relational algebra, relational calculus, and nonrecursive datalog with negation, forming the bedrock for SQL. For this thesis, the most pertinent of these languages is relational algebra (RA), including the following operations over relations: projection (π), selection (σ), join ($\bowtie$), union ($\cup$), intersection ($\cap$), difference ($-$), Cartesian product ($\times$), and rename (ρ).

On the flip side, at the physical level, researchers have devoted their energies to the efficient storage and the nimble implementation of algorithms. These endeavors are aimed at delivering the expected functionality of relational operators established at the logical level.

Eventually, the proliferation of tools tailored for the analysis of vast datasets charted a course for this research to give birth to the SQL language. SQL was crafted for query-intensive applications and a query evaluation framework, comprising distinct phases: (1) Parsing and translation, (2) optimization, and (3) evaluation [3]. In the first phase, the query is translated into a relational algebra expression, which is subsequently passed to the optimizer. The second phase leverages collected statistics about the data to reconfigure the relational algebra expression into a more efficient counterpart. In this process, the system catalog provides semantic information used to rewrite the query based on algebraic-rules (i.e. rule-based optimization) and statistics used to produce feasible logical and execution plans (i.e. cost-based optimization). Each operation is annotated with instructions on its evaluation—this entails specifying the algorithm for the operation or stipulating a particular index or set of indices to utilize. This process culminates in the construction of a query-execution plan.

Finally, in the third phase, the executor, also known as the query-execution engine, takes the query-execution plan, executes it, and furnishes the answer to the query. This query evaluation framework is depicted in Figure 1.

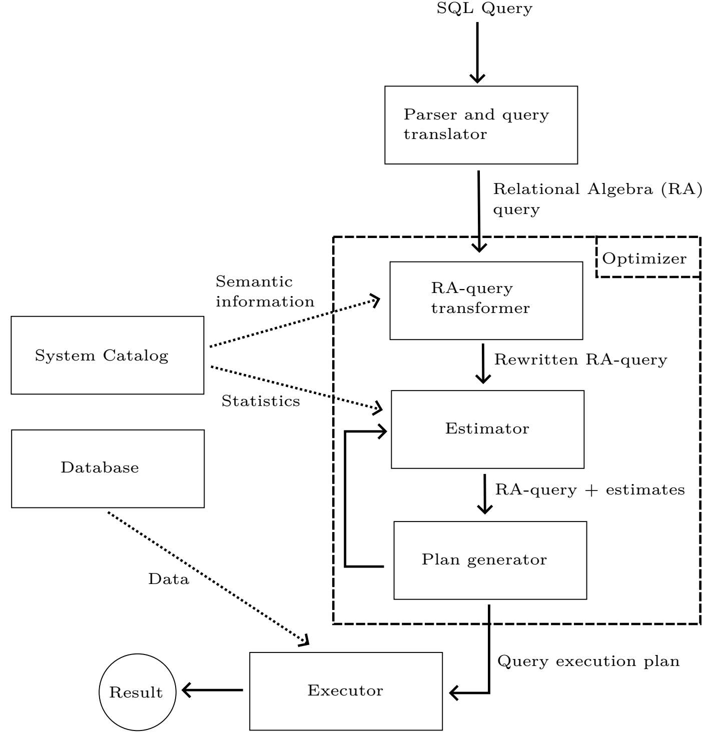


Figure 1. SQL query processing workflow

Example 2.1. Consider a database comprising two tables, with column names enclosed in parentheses: *Employee(E-ID, E-Name, E-LastName, E-Age, E-DeptID)* and *Department(D-ID, D-Name)*. Assume that the Employee table contains 50 rows, while the Department table contains 3 rows. Now, consider the following relational algebra queries:

$$Q1 = \pi_{\text{E-ID,E-Name,D-Name}} \left(\sigma_{\text{E-DeptID=D-ID} \wedge \text{E-Age}>25} \left(\text{Employee} \times \text{Department}\right)\right)$$

$$Q2 = \pi_{\text{E-ID,E-Name,D-Name}} \left(\sigma_{\text{E-Age}>25} \left(\text{Employee}\right) \bowtie_{\text{E-DeptID=D-ID}} \text{Department}\right)$$

At the logical level, both $Q1$ and $Q2$ aim to answer the same question: *What are the names and departments of employees over the age of 25?* Consequently, $Q1$ and

$Q2$ are equivalent as they yield identical results. However, if an optimizer is presented with $Q1$, it is highly likely to transform it into $Q2$, or even further optimize it. Here is why:

$Q1$ computes all conceivable combinations of records between the Employee and Department tables, resulting in a table with $3 * 50 = 150$ rows when evaluating Employee $\times$ Department. Conversely, $Q2$ initially filters the Employee table to select a subset of rows containing employees older than 25. These selected n rows are subsequently joined with Department rows having matching department IDs (E-DeptID $=$ D-ID). This operation generates a table with a maximum of n rows when computing: $\sigma_{\text{E-Age}>25}$ (Employee) $\bowtie_{\text{E-DeptID=D-ID}}$ Department. In the worst-case scenario, where all employees are over 25, n equals 50. Consequently, $Q2$ requires the storage of a maximum of 50 rows in memory, while $Q1$ necessitates three times as much space (i.e., a 200% increase in memory usage).

At the physical level, once the optimizer has selected $Q2$, it proceeds to construct the query-execution plan. The choice of join algorithm depends on the specific database system but typically includes options like “Nested Loop”, “Hash Join”, and “Merge Join”. Given the relatively small size of the tables in this example, the optimizer is likely to employ the “Nested Loop” implementation for the join operation. For the selection on the Employee table, it will either conduct an index scan if a suitable index exists or resort to a full table scan. $\square$

As relational databases began to grapple with more intricate and hierarchical data structures, such as arrays and records, it became increasingly clear that the traditional relational model alone would not suffice. This was primarily because data structured in such hierarchical ways did not seamlessly fit into the flat, tabular representations of tables, often leading to redundancy—something the relational model was initially designed to minimize.

To address these challenges, the Nested Relational Model was introduced, allowing relations to have attributes with relation values. In this model, queries were formu-

lated using various forms of nested relational algebra (NRA). Some of these algebras extended the traditional relational algebra by introducing additional operators such as *nest* and *unnest*. These operators were used to transform nested relations, bringing relevant attributes to the outermost level, where regular relational algebra operators could be applied. Once the necessary operations were completed, attributes could be nested back to replicate the original nested structure. An influential example of such an algebra is the Thomas and Fischer NRA[31, 30].

D-ID	**D-Name**	**Employee**			
		E-ID	**E-Name**	**E-LastName**	**E-Age**
1	Human Resources	2	Jane	Doe	30
		5	Hunter	Smith	40
2	Information Technology	1	Jhon	Doe	25
		3	Levi	Smith	45
		4	Sean	McCormick	20

Figure 2. Example of relation *Department* with nested relation *Employee*

D-ID	**D-Name**	**E-ID**	**E-Name**	**E-LastName**	**E-Age**
1	Human Resources	2	Jane	Doe	30
1	Human Resources	5	Hunter	Smith	40
2	Information Technology	1	Jhon	Doe	25
2	Information Technology	3	Levi	Smith	45
2	Information Technology	4	Sean	McCormick	20

Figure 3. Example of unnested relation *Employee* from Figure 2

Example 2.2. Consider the nested relation shown in Figure 2. Writting a query in Thomas and Fischer NRA equivalent to those of Example 2.1 would look as follows:

$$\pi_{\text{E-ID,E-Name,D-Name}}(\sigma_{\text{E-Age}>25}(\mu_{\text{Employee}}(\text{Department})))$$

the query reads from right to left: unnest (μ) the attribute *Employee* from relation *Department*, which produces the table shown in Figure 3. Then filter out and retain employees whose age is greater than 25. Finally, output the id and name of such employees alongside the name of the department they belong to.

It is worth mentioning that traditional relational operators alone would not suffice to express the query from this example, as they can only operate on flat relations. Thus, the introduction of unnest becomes necessary. □

Other approaches focused on defining operators that could manipulate nested attributes directly, eliminating the need for prior flattening. These extended the selection operator to support subrelation constructs, enabled navigation (traversal of nested relations), and facilitated modification of nested relations. The Deshpande and Larson NRA[25] is a notable example of this category.

Yet another approach, exemplified by the Latha S. Colby NRA[23], redefined relational operators and the nest and unnest operators originally introduced in the Thomas and Fischer NRA. This redefinition allowed these operators to operate recursively, enabling them to work on subrelations without the requirement of flattening beforehand.

To conclude this concise overview of relational databases, it is essential to underscore the pivotal roles played by the schema and the optimization phase in the query evaluation process. These two foundational concepts have been instrumental in allowing relational query optimizers to achieve the remarkable robustness and efficiency they exhibit today. Interestingly, this level of robustness is a benchmark that NoSQL databases are now actively striving to incorporate into their systems, as they seek to tackle similar challenges in the world of modern query-intensive applications.

3 NoSQL Databases

The 1990s witnessed the explosive growth of the World Wide Web, leading to a significant increase in the deployment of relational databases[3]. Relational databases not only had to handle this surge in demand but also adhere to high standards of reliability and availability. Over the years, as the influx of data continued to swell, these databases evolved to support not only high transaction-processing rates, crucial

for update-intensive applications, but also a robust analytics engine.

However, with the realization that web logs held valuable insights into user behavior, companies sought to leverage this data to enhance marketing and advertising campaigns, driving the need for a different approach. It became evident that traditional RDBMSs were ill-equipped to handle the rapidly accumulating textual and semi-structured data, now known as Big Data. Additionally, as applications became increasingly interconnected, data of various types and formats arrived from multiple sources, necessitating extensive data transformations to conform to the rigid constraints imposed by the relational model. These transformations, often managed at the application level, added significant time to data insertion and updating processes.

Big Data is typically distinguished from relational databases across several key dimensions: Volume, Velocity, Variety, Value, and Complexity[43]. Volume pertains to the sheer quantity of data, which even parallel relational databases struggled to manage, as they were originally designed to operate across tens to a few hundred machines in parallel[3]. Big Data use cases often necessitated thousands of machines. Velocity relates to the speed at which data arrives, with streaming services being a prime example. This metric encompasses not only data transmission speed but also the time it takes to ingest (i.e., extract-transform-load) data. Variety reflects the diversity of data formats, including text, video, and audio, used by applications. Value indicates the usefulness of data in decision-making, which can be challenging to discern, particularly with unstructured data, often requiring data exploration as a preliminary step. Complexity denotes the interconnectedness of data structures and the potential impact of minor changes.

From an analytics perspective, these metrics represent challenges to efficient data utilization. Notably, Data Variety emerged as a top concern for analytic sprawl due to the loss of advantages offered by schema-based relational databases. These advantages included data integrity, consistency, and abstraction, the flexibility to modify the logical schema without affecting the physical one, and robust query optimization.

To address these challenges, NoSQL systems have progressively evolved. In the 2000s, early NoSQL systems offered lightweight data management without a high-level query language like SQL and relied on "eventual consistency," where distributed copies of data could be inconsistent but would eventually reconcile. This schema-less approach allowed rapid data ingestion and granted applications greater flexibility when dealing with distributed data stores. While these relaxed data management schemes offered scalability and availability benefits highly valued by applications, they also brought the challenge of complex system maintenance for database administrators.

In response to these challenges, modern NoSQL systems have incorporated stricter notions of consistency. Furthermore, they have started developing declarative query languages tailored to certain semi-structured data formats, exemplified by the SQL++ query language, which operates seamlessly on JSON data, and are currently working towards strengthening their query optimizers to meet the demands of query-intensive applications.

One of the primary challenges faced by NoSQL database systems in terms of optimization is the absence of a clear and universally applicable logical data model with well-defined mathematical foundations, as noted by [68]. In contrast to the early days of relational databases, where significant research efforts were dedicated to developing the relational model and its associated algebraic frameworks, the initial focus of NoSQL databases was predominantly on the physical properties of data, their storage, and their distribution across multiple systems.

Legacy NoSQL Systems

Object-Oriented Database Systems (OODBMS)

As mentioned in Section II.2, relational databases primarily focused on simple or atomic data types and their structural representation within the data. However, during the rise of the object-oriented movement in the 1990s, there was a growing

demand for support for more complex data types and nested structures. This was particularly evident because the object-oriented data model was closely aligned with popular programming languages of that era, such as Simula 67, Smalltalk, C++, and Java. Programmers saw significant advantages in directly mapping classes, objects, and attributes to databases due to the striking similarities between the object-oriented and relational models. In this mapping, classes resembled relations, objects or class instances corresponded to tuples, and class attributes were analogous to columns[50].

However, certain concepts from the object-oriented model, such as inheritance, did not have a straightforward translation to the relational model. For instance, in object-oriented programming, an attribute a of some class C could be of an abstract class type A. In this scenario, two different objects, O_1 and O_2, both of type C, might have attributes $O_1.a$ and $O_2.a$ holding instances I_1 and I_2 of concrete subclasses of A, respectively. In the relational model, a would typically be represented as a domain (i.e., a column) in the relation C. However, I_1 and I_2 cannot coexist in the same domain because they are fundamentally of different types, and RDBMSs lack the concept of inheritance[50].

Today, the common approach to model such cases in RDBMSs is to use an Object-Relational Mapping (ORM) framework like Hibernate. These frameworks handle the translation between the object-oriented and relational models. In this approach, the framework maps concrete classes to new relations and establishes relationships between them and class C based on the specified relationship type (1-1, 1-M, M-N) indicated by the user at the application level. ORM tools also provide an API for performing CRUD operations, making it easy to create, insert, delete, or update records that are distributed across multiple tables. These tools abstract complex operations like joins by providing programmers with predefined functions that execute queries at the database level to populate objects with information stored in multiple tables. In contrast, back in the 1990s, this translation process was performed manually by programmers, often involving extensive schema planning and meticulous manipulation

of indexes[50].

On the other hand, Object-Oriented Database Management Systems (OODBMSs) allowed the storage of complex, nested data types without requiring prior knowledge of how user-defined abstract data types (ADTs) functioned[50]. This flexibility had the advantage of eliminating the need for joins in some cases, resulting in faster queries. Furthermore, OODBMSs enabled the sharing of inherited objects across programs and supported computationally complete operations on data, in contrast to the distinctions between class methods and stored procedures in RDBMSs[50].

As mentioned in [50], queries in OODBMSs were expressed in the Object Query Language (OQL), a SQL-like query language that, at the time, extended the SQL-92 `select-from-where` syntax with additional features for path expressions, complex objects, and inheritance. Notably, OQL was not a complete language like SQL, as it relied on object-defined methods to handle updates and invoked state-altering methods such as create(), add(), delete(), etc., closely bound to the programming language. Path expressions in OQL made it a navigational language, enabling the traversal of references from one retrieved object to another.

However, despite the existence of a standard, most products at the time only partially implemented it or had their own dialects. Unlike SQL, which also has its share of dialects, the close integration with programming languages and object-oriented features made the divergence of internal models handled by OODBMSs a significant concern. Moving to another system or migrating data often posed substantial challenges[50]. Another issue was the management of primitive (non-complex) data types, which OODBMSs typically handled less effectively compared to RDBMSs[50].

XML (document-oriented) Database Systems

An attempt was made to create a unified data model that could accommodate various types of data in the form of XML documents. Back then, and as of today, XML data can be well-structured or loosely-structured[33]. A well-structured XML document

follows a relatively fixed schema and is typically used for communication between applications. In contrast, a loosely-structured XML document usually prioritizes human readability and does not have a well-stablished schema. Hence, well-structured XML documents are generally easier to parse than loosely-structured XML documents. An example of these two types of XML documents is shown in Figure 4.

```
<Message>
  <Talk date="2023-01-10" time="20:00">
    The Future of Databases
  </Talk>
  <Location> Jhon Doe Hall </Location>
</Message>
```

(a) well-structured XML

```
<Message>
  All students, please attend to the Talk
  <Talk> The Future of Databases </Talk> given at
  <TalkLocation> Jhon Doe Hall </TalkLocation>
  <TalkDate> 2023-01-10 </TalkDate>
  <TalkTime> 20:00 </TalkTime>
</Message>
```

(b) loosely-structured XML

Figure 4. Example of XML document types

The query evaluation process within early XML systems, most notably the Lore system, bore a resemblance to RDBMSs' query evaluation. This architecture closely adhered to the schematic representation illustrated in Figure 1. Notable distinctions included the use of "Lorel" (Lore Language) for query composition, an Object Query Language (OQL)-based language featuring a SQL-like `select-from-where` syntax[2]. The internal data model employed for XML was the Object Exchange Model (OEM), which resembled a labeled directed graph[66]. The optimization process encountered familiar challenges from Relational DBMSs, along with additional complexities such as efficient traversal of subelements and reference links, often referred to as path traversals[55].

One of the primary concerns in this pioneering system revolved around adapting collected statistics to suit the graph-like data model. Among these statistics, a critical component was the description of the graph's shape, essentially an approximation of the logical schema. This proved immensely valuable for the optimizer as it facilitated the discovery of optimal or near-optimal navigation paths. The intricacies of navigating paths were so profound that the system resorted to the aggressive application of new heuristics, surpassing those typically utilized by traditional RDBMSs.

To illustrate the concept of an XML document visualized as an OEM graph, Figure 5 presents an adapted example from [20].

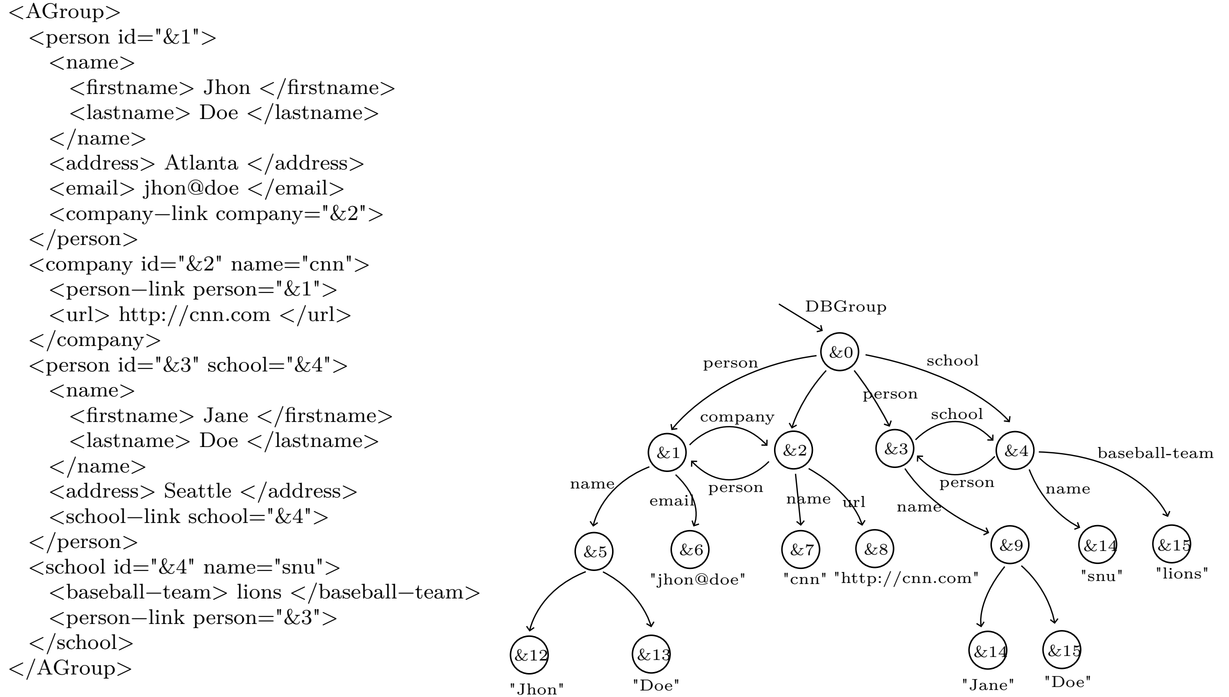


Figure 5. Example XML data and OEM graph

Apart from Lorel, the early days of XML databases saw the emergence of several other XML query languages, including XML-QL and YaTL. These languages shared commonalities in their querying approaches and expressive capabilities[29]. As XML gained prominence, a plethora of languages were crafted, some serving purposes beyond mere querying. Among these, three distinguished themselves as the most prominent: XSLT, XQuery, and XPath.

XSLT, or eXtensible Stylesheet Language - Transformation, was designed to perform transformations on XML data, converting it into various other formats such as HTML, RDBMS load files, plain text (including CSV), PDF, or even another XML document with a distinct tag set. Its primary role revolved around altering the display format of information contained within one or multiple XML documents.

On the other hand, XQuery was tailored explicitly for querying XML databases, akin to how SQL operates on Relational Database Management Systems (RDBMSs).

While XQuery possessed the capability to convert one XML document into another, it did not match the transformation prowess of XSLT.

XPath, in essence, functions as a subset of both XQuery and XSLT. It serves as the syntax for navigating through XML documents. These three languages were thoughtfully designed to work in concert. For instance, an application could extract data from an XML database using XQuery, articulating navigational queries via XPath, and finally present the results to users by employing XSLT transformations.

Example 3.1. Consider the following XML document:

```
<company>
  <employees>
    <employee id="1" age="25">Jhon Doe</employee>
    <employee id="2" age="30">Jane Doe</employee>
    <employee id="3" age="45">Levi Smith</employee>
    <employee id="4" age="20">Sean McCormick</employee>
    <employee id="5" age="40">Hunter Smith</employee>
  </employees>
  <department id="1">
    <name>Human Resources</name>
    <employeeRef>2</employeeRef>
    <employeeRef>5</employeeRef>
  </department>
  <department id="2">
    <name>Information Technology</name>
    <employeeRef>1</employeeRef>
    <employeeRef>3</employeeRef>
    <employeeRef>4</employeeRef>
  </department>
</company>
```

To retrieve the name of departments where at least one employee is older than 25, the following XPath query suffices:

```
//department[employeeRef=//employees/employee[@age>25]/@id]/name
```

it reads as follows: Identify all `<department>` elements within the hierarchical structure, regardless of their level, and filter these elements to include only those where

the `employeeRef` attribute matches the result of a subquery. This subquery entails selecting the `<empoyees>` element, which is a child of the top-level `<empoyees>` element. From these selected empoyees, only those whose age is greater than 25 are retained, and their corresponding `id` attributes are extracted.

On the other hand, to retrieve something slightly more complicated like the name of all employees older than 25 and their corresponding department (i.e. the equivalent of both queries from Example 2.1) the following XQuery FLWOR expression can be used:

```
let $root := .
for $d in $root//department, $e in $root//employees/employee
where $e/@age > 25 and $d/employeeRef = $e/@id
return <result>{data($e/@id)},{data($e)},{data($d/name)}</result>
```

the query starts with a `let` statement, which serves to define a variable. Then, it uses `for` to establish range variables, `d` and `e`, which iterate over departments and employees, respectively. The `where` clause filters and retains pairs of (department, employee) where the employee's age is greater than 25. Finally, the `return` statement specifies to retrieve the employee id, name and department.

□

XML gained popularity for a time, with notable RDBMS systems like PostgreSQL and SQL Server adopting it, a scenario referred to as an XML-enabled database (i.e., an RDBMS that offers XML support). In contrast to Native XML databases (NXDs), XML-enabled databases underwent more extensive processing steps. They either stored XML documents as opaque LOB (Large OBject) data types or engaged in data extraction and mapping to a relational schema, a process commonly known as "shredding."

In both cases, queries expressed in XQuery had to be translated into SQL. In the former approach, documents were parsed every time they were retrieved to validate

their XML schema. This strategy had its drawbacks, as elucidated in [63], including the need to load and re-parse entire documents into memory for conducting full-text searches, limited search capabilities on shredded XML documents, heavy dependence on indexes for search operations, and significant performance penalties during data writes if an index required modification.

In contrast, NXDs employed hierarchical storage techniques that preserved information about the structure of XML documents. Consequently, their search processes were more memory-efficient, as they could partially load documents, and eliminated the need for the document identity loss associated with "shredding". This highlights the advantages of NXD over XML-enabled DBMSs when querying loosely-structured XML documents compared to well-structured ones.

Eventually, XML's inherent complexity, verbosity, and lack of a clear standard led to a decline in its popularity. XML incurred substantially greater storage costs compared to more streamlined formats like JSON and posed challenges for optimization due to the intricate data hierarchies it introduced.

Modern NoSQL systems

As NoSQL systems evolved, they departed from the notion of a one-size-fits-all data format and instead embraced specific formats that align closely with the intended use cases of the respective databases. This evolution gave rise to various categories of NoSQL systems, each tailored to specific data models, as discussed in Section II.1. Some of the most prominent categories include:

- **key-value stores**: Often referred to as "big hash tables", these stores manage collections of tuples in the form of (key, value) pairs. Keys serve as unique identifiers similar to columns in a relational database, while values can be structured or unstructured, often stored as Binary Large Objects (BLOBs). Importantly, pairs can encompass different data types, unlike the strict typing of the rela-

tional model. Examples of key-value stores include Google's Bigtable, Amazon's DynamoDB, and Apache HBase.

- **document stores**: Document stores represent a subtype of key-value stores that permit records to have a semi-structured schema. These systems focus on the content and structure of data stored in (key, value) pairs rather than treating the "value" part as an opaque BLOB. However, there is no overarching schema for an entire set of records, in contrast to relational databases. MongoDB and Couchbase are well-known document stores that often store values in the JSON format.

- **Column-family stores:** In column-family stores, data is not constrained to individual rows; this stands in contrast to RDBMSs, where missing values are often represented as `NULL`. Instead, column-family stores employ a structure where data is organized into column families. These column families are essentially key-value pairs, where the key represents the name of the column family, and the value encompasses the associated columns. Columns that share the same column family are physically stored together on disk. Consequently, it is common practice to group columns that are frequently accessed or modified in the same column family. In column-family stores, the schema is highly flexible, allowing columns to be added to a column family at any point in time.

- **graph databases**: These databases excel in managing highly interconnected data structures, which can be viewed as nodes and edges in a graph. Although the relational model can represent graph data, graph databases like Neo4j offer more efficient and programmer-friendly implementations of path traversal queries.

The absence of a standardized and universally agreed-upon data model in NoSQL databases poses challenges for researchers. It restricts the exploration of schema properties through mathematical and complexity theory, a well-established practice in

the field of relational databases. Consequently, researchers often rely on entirely new algebraic frameworks or leverage existing query languages with sufficient expressive power to accommodate the specific logical data model of a given NoSQL system. However, the lack of a universally accepted data model means that logical properties discovered through research are often applicable exclusively to the specific model in question. This complicates efforts to exploit established optimization patterns in the broader context of NoSQL databases.

Moreover, since records within these systems can exhibit varying structures, the schema of a collection of records cannot be assumed to be fixed but rather is variable. This variability has notable implications:

- If an attribute (e.g., attribute x) is present in some but not all records within a collection, removing records without attribute x would effectively remove x from the schema. The optimizer often lacks access to the values of the data itself, making it uncertain about the removal of an attribute unless the filter explicitly requests the attribute's existence.

- Attributes may assume values of multiple types, which can either aid or complicate optimization efforts. The presence of various data types allows filters to more accurately estimate the number of retained records based on data heterogeneity. However, operations defined for multiple data types introduce complexity to the physical schema and may raise interpretability issues. To illustrate, consider a scenario where a collection comprises ten records, each having an attribute x. In two of these records, x is an integer, while in the remaining eight, x is a string. When applying a filter such as $x > 5$, it becomes evident that, at most, two records will satisfy this condition. This is because the filter is unequivocally false for any record where x is not an integer. When dealing with records where x is of string type, should the condition $x > 5$ be interpreted as $x >$ "5", with the comparison based on lexicographical order? Such a feature is

not unprecedented, as many programming languages have long supported implicit type conversion (e.g., arithmetic operations between operands of different types in C++). Nevertheless, adopting this approach could potentially compromise the advantages previously mentioned. It would prevent the optimizer from assuming that the condition $x > 5$ will always evaluate to false for records where x is a string.

In summary, NoSQL databases offer diverse data models tailored to specific use cases, challenging researchers to adapt established methodologies to these models while grappling with issues of schema variability and diverse data types.

4 Document Stores

The JSON Data Model

The JavaScript Object Notation (JSON) was conceived as a text format facilitating data interchange between programming languages[16, 24].

The format is built on top of two fundamental hierarchical structures: *objects* and *arrays*. Objects are a collection of key-value pairs and arrays are ordered lists of values. A value can be a number, string, true, false, null, object or array.

Because JSON has a language agnostic syntax, types are generic, or abstract, in the sense that they encompass many different concrete types defined within a particular language. For instance, the *number* type can be represented by a variety of concrete types (e.g. int32, int64) in multitude of programming languages. However, even if their internal representation differs, they all understand a sequence of digits to be a number and this is enough to interchange data. Objects can be represented as dictionaries, hash tables, etc., and arrays by vectors, lists, etc.

Format-wise, a string is a sequence of unicode points enclosed in quotation marks. *null*, *true* and *false* are reserved keywords. Object keys are strings, key-value pairs are enclosed in curly brackets and separated by a colon (:), and keys within the

same object must be unique. Array elements are all enclosed within square brackets. Listing II.1 shows the corresponding BNF grammar.

```
object ::= {} | { member }
member ::= pair | pair, member
pair ::= string:value
array ::= [] | [element]
element ::= value | value, element
value ::= object | array | string | number
string ::= "" | \"character string\"
number ::= + number | - number | digit number
```

Listing II.1. JSON BNF grammar

Numerous formalisms have been proposed in the literature for modeling JSON data[15, 52, 37, 39, 14]. While most of these formalisms recognize JSON's hierarchical nature and represent it as a labeled tree, there is not a unanimous consensus on the most effective way to handle arrays.

For instance, [39] highlights concerns with using array indexes as edge labels, particularly when certain operators require relabeling the edges. However, [15] and [37] assert that array indexes as edge labels are essential for JSON's navigational primitives when traversing the hierarchy. An example of the latter representation is shown in Figure 6.

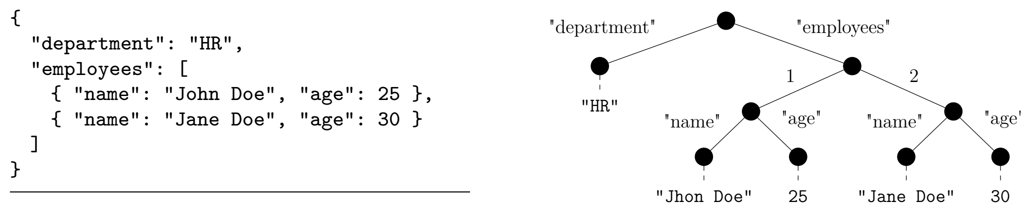


Figure 6. Example of a JSON object modeled as a labeled tree. Dotted lines represent the values leaf nodes point to.

It is worth noting that there are formalisms used for XML, such as the XML data-tree model, that share significant similarities with these JSON formalisms, mainly due to JSON's tree-like structure[15]. This resemblance creates opportunities for potentially reusing constructs and concepts from the realm of XML databases. However,

JSON and XML have inherent differences that render many of these XML constructs incompatible with JSON in their original form:

- JSON's deterministic nature arises from its requirement for distinct sibling edges within a JSON tree, while XML trees are inherently non-deterministic.
- Unlike JSON, XML lacks an equivalent construct to represent arrays.
- JSON objects can contain other JSON objects as values, introducing the necessity to consider subtree equality, whereas in XML, comparisons are typically made between atomic values.
- JSON objects have no inherent ordering, but arrays are ordered in JSON. This duality means that JSON accommodates both ordered and unordered data.

As a result, any attempt to repurpose XML constructs for JSON demands careful adjustments and considerations to account for these inherent distinctions.

A document database is designed to store and query self-defined, hierarchical (tree-like) data structures called *documents* often following the JSON specification, or an extension of it, and stored using a plethora of mechanisms. A document is recursively defined as a list of attributes and values, where each value can be simple (e.g. a string or number), a list of values, or a whole document. This creates a nested structure of documents and sub-documents, similar to nested relations ([22]). Documents are analogous to records (tuples) in RDBMSs, but are not tied to a rigid schema; that is, their attributes may have differences across documents logically having the same type. Document databases store sets (or multisets) of documents in a logical organization of typically, but not limited to, JSON data. MongoDB and Couchbase call this a collection, analogous to tables in relational databases (RDBMS). One or more collections make up a database (or Bucket as it is called in Couchbase).

Document-oriented databases typically adopt a different approach to data modeling compared to relational databases. In a document-oriented database, it's common

practice to aggregate all available data within a single collection. In contrast, relational databases organize data entities and their relationships across multiple tables, relying on foreign keys to establish connections between entities.

The document-oriented database model excels at handling *one-to-one* (1-1) and *one-to-many* (1-M) binary relationships through a technique known as *nesting* or *embedding*. This means that related data is included directly within a document, often in the form of arrays or sub-documents. For example, consider entities like `Customer`, `Order`, and `Lineitem` in the TPCH schema [73]. In a document database, you could store these entities in a single collection. Each document represents a customer and includes an attribute called "Orders," which contains an array or list of sub-documents representing orders. Each of these orders, in turn, contains a "Lineitems" attribute, an array or list of sub-documents representing line items. This hierarchical structure aligns well with the document model's capabilities.

However, challenges arise when dealing with *many-to-many* (M-N) relationships, which can introduce consistency problems and redundancy in document databases. For instance, entities like `Parts` and `Suppliers` have an M-N relationship. In a document database, representing this relationship directly can be problematic. To prevent redundancy, a common strategy is to employ *linking*. Each document in the collection is assigned a unique identifier by the database system. With this identifier, you can create two separate collections, one for parts and another for suppliers. Then, you can include an attribute in the parts documents containing a list or array of supplier IDs, and vice versa. However, this approach has significant implications for the query language and the way queries are constructed, as will be discussed in Section III.1.

MongoDB and Couchbase closely adhere to the document data model. Documents are typically formatted in JSON, and both databases offer the flexibility of using both embedding and linking to model relationships [19]. For instance, following the strategy outlined above, you can model relations like `Customer`, `Order`, and `Lineitem`

within a single collection. Alternatively, you have the option to represent them in three separate collections or explore other combinations. These modeling choices will be illustrated schematically in Section III.1.

At the physical level, various systems employ binary formats that enhance JSON's capabilities in distinct ways. For instance, MongoDB utilizes BSON, PostgreSQL employs JSONB, and MySQL adopts its unique binary representation. These binary extensions introduce their own primitive data types to JSON, expanding its range of functionalities.

In contrast, AsterixDB employs the Asterix Data Model (ADM), which incorporates certain OODBMS principles, such as Derived Types, into JSON.

Conversely, certain systems like MariaDB and SQL Server store JSON data as text and provide features to parse JSON records, similar to the approach adopted by XML-enabled databases. This parsing process allows these systems to extract the necessary content and convert it into specific data types as needed.

Query language

Most document-based systems provide a navigational query language, i.e. one where several operators are connected in a sequence to build queries. Typical operators in such languages allow operations in documents of a collection that are similar to selection, projection, grouping and aggregation in relational algebra. A query is given by a "chain" of operators: $OP_1.OP_2 \ldots OP_n$, where each operator may have additional parameters and the whole chain is "anchored" in a single database collection. Operators are executed in the order in which they appear in the chain unless an optimizer decides to change this order.

MongoDB follows this pattern. In MongoDB parlance, the query defines a data processing pipeline: documents from the collection are the initial input; they go through multiple pipeline stages (one per operator) and the final stage outputs a result. Operators are considered methods of the collection, invoked using the dot

notation. An schematic of this is shown in Figure 7, where each stage $Stage^i$ is comprised of a collection of operators ops^i and a buffer $buffer^i$ to store any temporary data produced.

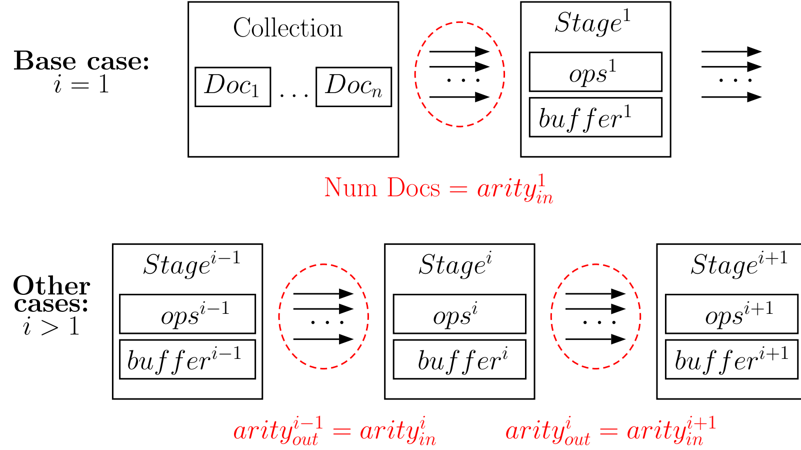


Figure 7. Pipeline-like query language schematic

MongoDB offers methods that fulfill the same role as relational operators. The input and the output for each all takes as input a collection of documents and returns a collection of documents:

- **`$match`: condition** filters out documents that do not meet a one or multiple conditions.
- **`$unwind`: array-name** outputs a new document per element in an array. This is an unnest operator ([22]).
- **`$project`: attribute-list** creates new documents by projecting attributes mentioned in the `attribute-list`.
- **`$group`: expression-list** creates groups of documents and outputs a document per group, applying any aggregates in `expression-list`. This acts like SQL's GROUP BY.

Conditions on nested attributes are expressed using the dot notation to reach such attributes. Unfortunately, this syntax provokes some ambiguity, since it can be

used to go inside an array, whence it is unclear whether the condition is meant for at least one, several, or all the elements of the array. A full list of pipeline stages and operators used within stages is provided at the MongoDB online manual[2,3]. The language's flexibility allows for queries to be written using different combinations of stages and methods. In fact, because documents can be nested the schema design may determine what is the best of such combinations. Example 4.1 shows common combinations to filter nested documents.

Example 4.1. Consider a collection with the document below:

```
{
  "department": "HR",
  "employees": [
    { "name": "John Doe", "age": 25 }, { "name": "Jane Doe", "age": 30 },
    { "name": "Carol Smith", "age": 26 }
  ]
}
```

Suppose that a query is issued to find the employees of each department whose age is greater than 25. In MongoDB, there are two ways to express such condition; both shown in Figure 8.

```
{'$project': {
  'department': 1,
  'employees': {
    '$filter': {
      'input':'$employees',
      'as': 'employee',
      'cond': { '$gt': ['$$employee.age',
           25] }
    }
  }
}},
{'$unwind': '$employees'}
```

(a) $project and $filter + $unwind

```
{'$unwind': '$employees'},
{'$match': {
  'employees.age': { '$gt': 25 }
}}
```

(b) $unwind + $match

Figure 8. Example of two different ways to filter nested documents

In Figure 8a, the `$filter` operator within the `$project` stage removes elements from the `employees` array without deconstructing it. The parameter `as` declares

[2]https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/
[3]https://www.mongodb.com/docs/manual/reference/operator/aggregation/

a variable named `employee` to reference in the condition parameter (`cond`). This variable is de-referenced (i.e. its value is accessed) by prepending `$$` to its name, i.e. `$$employee` and passed to the `$gt` (greater-than) operator. In essense, it behaves like the array `filter()` method in JavaScript:

```
function isOlderThanTwentyFive(employee) {
  return employee.age > 25;
}
let employees = employees.filter(isOlderThanTwentyFive);
```

after this stage, the document would look as follows:

```
{
  "department": "HR",
  "employees": [
    { "name": "Jane Doe", "age": 30 }, { "name": "Carol Smith", "age": 26 }
  ]
}
```

The `$unwind` stage deconstructs the filtered array afterwards, thus producing:

```
{ "department": "HR", "employees": { "name": "Jane Doe", "age": 30 } }
{ "department": "HR", "employees": { "name": "Carol Smith", "age": 26 } }
```

In Figure 8b, the array `employees` is deconstructed first, thus producing an intermediate result that includes employee John Doe:

```
  { "department": "HR", "employees": { "name": "Jhon Doe", "age": 25 } }
  { "department": "HR", "employees": { "name": "Jane Doe", "age": 30 } }
  { "department": "HR", "employees": { "name": "Carol Smith", "age": 26 } }
```

then, the `$match` stage filters out John Doe. □

When a database contains more than one collection, it may be necessary for a query to retrieve data from two or more of them. For such cases, MongoDB provides an analog to relational join, called **`$lookup:` from, localField, foreignField**. This operator performs a left outer join between the collection from which the operator is called (which plays the role of "left" relation) and a collection specified in the `from` field (which plays the role of "right" relation) on condition `localField`

`= foreignField`. The syntax can be extended to perform uncorrelated subqueries and/or specify multiple join conditions.

In contrast, some systems offer declarative query languages that, akin to SQL, encompass both a data definition language (DDL) designed for index creation, modification, and deletion, and a data manipulation language (DML) for actions such as selecting, inserting, updating, deleting, and upserting data into JSON documents. This approach is followed by platforms like Couchbase and Asterixdb, both of which employ the SQL++ language. SQL++ is a declarative SQL-like language that maintains significant compatibility with SQL.

SQL++ intentionally eliminates many of SQL's constraints to minimize the introduction of new features, instead opting to reuse much of SQL's underlying semantics[64]. Here are some notable distinctions:

- The *FROM* clause is no longer limited to binding with tuples but can be associated with any JSON element.
- SQL++ prioritizes composability, allowing subqueries to generate nested results when utilized in the *SELECT* clause.
- While SQL prohibits the correlation of subqueries within the same *FROM* clause, SQL++ enables the correlation of earlier defined variables in a *FROM* clause with subsequent subqueries in the same clause.
- In SQL, groups formed by a *GROUP BY* clause are exclusively employable in aggregate functions. In contrast, SQL++ permits their use in nested queries.
- SQL++ neither enforces nor demands data homogeneity.

A comprehensive survey[64] outlines a multitude of SQL++ features and highlights its compatibility with fifteen diverse data models. Additionally, it offers empirical validation of SQL++'s support for various query language features. This

information is succinctly presented through feature matrices, demonstrating the language's expressive capabilities.

CHAPTER III

PROPOSED APPROACH

1 The Problem: Query Optimization

The need to store and query semi-structured data has a long history, dating back to the emergence of XML databases as the earliest form of document stores. Research efforts in optimizing these systems mainly revolved around two primary objectives: (1) Developing navigational techniques for efficiently traversing hierarchical XML structures and (2) creating methods for storing and retrieving XML data efficiently, often involving compression due to XML's inherent verbosity.

In contrast, modern document stores have primarily focused on handling high-volume transaction processing and addressing challenges related to not being fully ACID-compliant. However, the recent shift in research priorities is towards supporting analytics. This shift is driven by the fact that analytics tends to follow the data, and the growing interest in near-real-time analytics, which eliminates the time for a costly ETL process, especially when dealing with heterogeneous data. Consequently, interest in query optimization for document stores has been on the rise, as evidenced by several recent papers ([42, 72, 67, 44, 49]).

Despite these recent advancements, query optimization for document stores is not yet fully developed:

- Optimization techniques from XML databases are not readily applicable to JSON databases due to several key differences:

  - *Deterministic Structure:* JSON trees have a deterministic structure, which restricts navigation compared to the more flexible navigation in XML. In XML, you can easily navigate between sibling elements, while JSON's requirement for distinct keys at the same level simplifies value retrieval. These distinctions render some XML-specific navigational optimizations ineffective in the context of JSON, thus influencing the design of query languages.
  - *Mixed Data Order:* JSON mixes ordered and unordered data, which is not native to XML. While this behavior can be emulated in XML, it's not inherently enforced. As a result, optimization techniques for XML often assume all elements are unordered, an assumption that doesn't hold in JSON.
  - *Subtree Comparisons:* JSON commonly involves comparisons between JSON subtrees, whereas XML primarily focuses on value comparisons within attributes of nodes, rather than comparing entire subtrees.

- Optimization techniques for any Nested Relational Algebra (NRA) face challenges when applied to JSON databases:

  - *Unknown Schema:* In JSON, the schema of nested relations is often unknown, unlike in NRA where it is well-defined. This creates complications when using operations like unnesting in JSON, as incorrectly defined unnesting operations can lead to documents that violate the rule of distinct object keys.
  - *Heterogeneous Arrays*: JSON arrays can be heterogeneous, containing elements of various types. NRA's unnest operator is designed to bring attributes of nested relations to the root level, but JSON arrays don't have attributes in the same way. This creates ambiguity, particularly when

dealing with arrays of primitive types. For example, unnesting an array of integers doesn't have a straightforward counterpart in NRA.

Many approaches still rely on rule-based methods, and the development of a comprehensive approach is hindered by the diversity of systems with varying query languages and back-ends. Additionally, JSON's unique features, such as heterogeneity and the combination of ordered and unordered elements, pose challenges for clean and efficient query processing.

However, a common thread among many recent proposals for optimizing JSON data queries is the adoption of the relational framework. This framework involves creating an internal declarative query representation, transforming this representation into equivalent forms using formal equivalences, and then converting each resulting representation into a query plan by implementing the corresponding operators. Some proposals, particularly those focusing on cost-based optimization ([44, 58]), also incorporate the ability to estimate the cost of each plan.

While this framework provides a foundation for general cost-based optimization for JSON data, it demands significant customization at each step to accommodate the specific characteristics of the JSON data model and leverage new techniques. For instance, similar to RDBMSs, capturing statistics about a dataset plays a crucial role in query processing, as demonstrated in Example 1.1 for estimating a physical plan.

Example 1.1. Consider again the relations *Department* and *Employee* from Example 2.1, but this time assume that there are 500000 employees instead of just 50. A common statistic maintained by relational systems is a histogram, which gives information about the distribution of data in a column. Suppose a histogram is created for column *E-Age* of table *Employee* and it finds that there are 20 distinct values (ages 20,21,. . . , 40), 96% of employees are between the ages of 31 and 40, and 4% between 20 and 29. If an user queries for the employees older than 30, an optimizer without a histogram assumes an even distribution, thus estimating a cardinality of

500000/20 = 25000 rows (i.e. 5% employees). With this estimate, the optimizer chooses to perform an index scan of the table. With the histogram, the optimizer estimates that 96% of the employees are older than 30, thus choosing a most appropriate alternative: a full table scan. □

In the case of JSON data, statistics gathering and use in a RDBMS-fashion is especially complicated due to the lack of schema enforcement and data heterogeneity. Example 1.2 illustrates this.

Example 1.2. Consider the tables from Example 2.1, but with employee ages stored as birthdays in the *E-Birthday* attribute. These tables can be modeled in JSON in two ways:

1. **Two Collections:** Similar to the relational case, two separate collections are used, one for employees and another for departments.

2. **Single Collection:** Here, a single collection is used where each document represents a department, and employees are nested within an array under their respective departments. This structure resembles the JSON document in Figure 6.

In the first model, building a histogram, as shown in Example 1.1, could be straightforward if the employee birthdays were always of a numeric type. However, a document store faces challenges due to:

- **Lack of Schema Knowledge:** Some RDBMSs, like Oracle[6], auto-create histograms for columns with high query workloads based on past queries. However, this feature isn't feasible when the system lacks schema knowledge and cannot track attribute references.

- **Heterogeneity:** Documents in the collection may have birthdays represented as strings, integers, or other types. For instance:

```
{ "name": "Jhon Doe", "birthday": "1998-03-10" }
{ "name": "Jane Doe", "birthday": 19931010 }
```

- **Lack of Integrity Constraints:** Even if the documents are homogeneous at one point, future insertions may introduce heterogeneity.

Maintaining a histogram per type or a single histogram that accommodates all types becomes significantly more complex in a document store compared to maintaining histograms in RDBMSs.

In the second model, the same problems as the first model apply. Additionally, even if a histogram exists, it becomes challenging for the system to use it reliably. For instance, if the query language requires flattening nested data, a query like "select all employees older than 30 from the HR department" would utilize the histogram effectively in RDBMSs and the first model but not in the second. This is because when querying at the top level (i.e., departments), the removal of non-HR departments automatically eliminates the associated employees, rendering the histogram ineffective. □

Some concepts from XML statistics gathering have found utility in the context of JSON data processing. Notably, [46] and [10] adapted an early proposal on XML statistics gathering, introducing a dynamic schema called DataGuide[35]. This (dynamic) schema serves as a repository for statistics, sample values, and structural summaries for XML collections. Most reuse efforts have focused on enhancing efficient storage ([26]) or integrating JSON data with relational systems ([52]).

Another approach known as the "structural index" compiles metadata for query optimization during parsing. These indexes are particularly suitable for DBMSs that store JSON as text, like the systems mentioned in Section II.4. They assume JSON documents are parsed when processing a query. For instance, the Mison parser[49] employs a structural index in the form of a bitmap, designed to navigate structural

characters within JSON, such as ":" for child relationships and "," for siblings. The Pison structural index constructor[41] provides techniques for efficiently constructing structural indexes, including parallelism within records and memory-saving mechanisms for processing deeply nested records.

However, it is important to note that these approaches primarily address optimization at the physical schema level, with less emphasis on the logical schema. They assume the presence of logical operators akin to those from Thomas and Fischer NRA (i.e., relational algebra operators along with nest and unnest) as these operators are found in the query languages of popular JSON document stores and RDBMSs that support JSON.

Furthermore, JSON document stores heavily rely on indexing for optimization. Over-reliance on indexes necessitates meticulous upfront planning and complicates the migration process to other JSON DBMSs, as they may not support the same types of indexes. Consequently, migrating data to a new system requires a fresh round of planning to determine the appropriate indexing strategy.

However, the need for modern document stores to handle high ingestion rates often forces system administrators into a reactive stance rather than a proactive one. This thesis identifies this situation as an opportunity to enhance the utilization of readily collectable statistics for query optimization at the logical level when a suitable index is unavailable.

Finally, to evaluate the optimization challenges outlined earlier in JSON document stores, an experimental validation was carried out. Detailed information regarding this validation is presented in Section III.1. This validation offers valuable insights into the performance of popular and widely adopted JSON Document Stores in comparison to traditional RDBMSs concerning the optimization of analytical queries within a Decision Support environment.

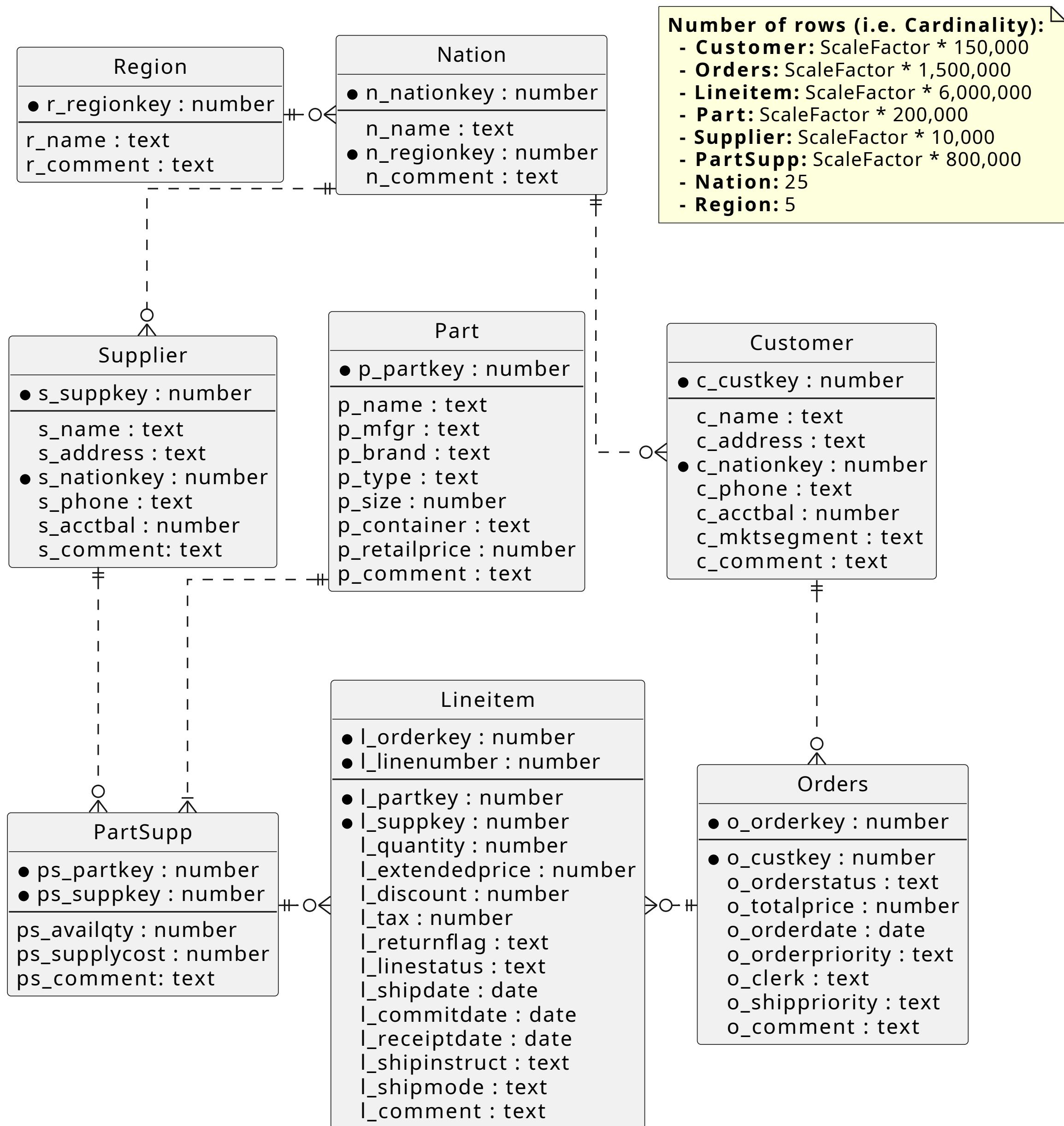


Figure 9. ER diagram of the TPC-H Benchmark's schema

Figure 10. Nested models of the TPC-H schema analogous to the schemas of data used during experimentation. Attributes in bold are used to link the relations

Experimental validation

Two comprehensive experiments were conducted, encompassing popular RDBMSs and JSON DBMSs. These experiments encompassed diverse database designs and query formulations to assess the impact of schema design and query optimization in document-based stores. The comparison was drawn against RDBMSs, which served as a baseline for evaluation. Both experiments employed the well-established TPC-H benchmark, renowned as a Decision Support System (DSS) Benchmark, across various scale factors (1G, 10G, 25G, 50G, 100G). The schema of this benchmark is shown in Figure 9.

For the JSON DBMS scenario, the dataset was initially generated using the dedicated TPC-H benchmark generation tool known as DBGEN. It was then mapped to JSON in the following schemas:

- **Denormalized Schema (S1):** This schema consolidates data into a single collection named $\mathcal{COL}$. Here, customers are endowed with an array containing zero or more orders, with these orders, in turn, housing arrays of line items.

- **Normalized Schema (S2):** This schema mirrors the relational model, dispersing documents across three distinct collections, namely $\mathcal{C}$, $\mathcal{O}$, and $\mathcal{L}$, which correspond to the tables Customer, Orders, and Lineitem in a relational context. The 1-M relationships between collections are depicted through attributes such as `o_custkey` in $\mathcal{O}$, referencing customers, and `l_orderkey` in $\mathcal{L}$, linking to orders.

- **Hybrid Schema (S3):** This schema entails documents residing in two collections, $\mathcal{C}$ and $\mathcal{OL}$. $\mathcal{C}$ is identical to its counterpart in S2, whereas $\mathcal{OL}$ incorporates the embedding of line items into orders while preserving a reference link represented by the attribute `o_custkey` to link back to customers. This approach maintains symmetry in embedding orders into customers and forming connections from line items to orders. Each variant is tailored to enhance

specific queries at the expense of others. Thus, evaluating the performance of one schema provides insights into the others.

For the many-to-many (M-N) relationship between Supplier and Part, two collections, $\mathcal{S}$ (for Supplier) and $\mathcal{P}$ (for Part), were employed using a two-way embedding technique. Within $\mathcal{S}$, the attribute `s_parts` stores a list of IDs corresponding to all associated parts. Within $\mathcal{P}$, the attribute `p_suppliers` maintains a list of documents that encapsulate all relevant suppliers along with attributes from the PartSupp table. Furthermore, $\mathcal{S}$ incorporates attributes from the Nation table within the `s_nation` field and attributes from the Region table within the nested field: `s_nation.n_-region`. Notably, attributes from $\mathcal{PS}$ can be embedded within Parts, Suppliers, or both.

The nested relational model of these schemas is shown in Figure 10 for illustrative purposes.

Experiment 1: Rich querying does not imply effective querying

This experiment examines the query capabilities of MongoDB and Couchbase, focusing intently on query re-ordering and the selection of optimal indexes. Particular emphasis is placed on MongoDB due to its rich aggregation query framework, boasting a plethora of operators that can effectively emulate the functionality of various operators from distinct nested relational algebras. MongoDB is also renowned for its heavy reliance on indexes to enhance the efficiency of join and selection operations.

It's worth noting that this experiment exclusively addresses one-to-many relationships for data modeling. Consequently, the Part and Supplier tables from TPC-H are not within the scope of consideration.

All queries are run on an IBM System x3650 M2 server with two Intel Xeon X5672 processors at 3.20GHz, 16 virtual cores, and 46GB of RAM. The server runs CentOS Linux version 7.6.1810, MongoDB version 4.0.6, Couchbase version 6.5, and PostgreSQL version 10.6. The operating system and all databases are tuned according

to best practices. All queries are run five times and the average running time is reported. After every single execution, the cache of the operating system and both databases is cleared.

The TPC-H data is mapped to PostgreSQL in plain relational following the TPC-H schema and an extra table (called $\mathcal{JCOL}$) is created using the JSONB datatype to mimic schema S1. To distinguish between the collection schemas S1, S2 and S3 on the document stores, the collections are named as indicated in Table 1.

	Collection Schemas		
JSON DBMS	**S1 (Denormalized)**	**S2 (Normalized)**	**S3 (Hybrid)**
MongoDB	$\mathcal{MCOL}$	$\mathcal{MC}, \mathcal{MO}, \mathcal{ML}$	$\mathcal{MC}, \mathcal{MOL}$
Couchbase	$\mathcal{CBCOL}$	$\mathcal{CBC}, \mathcal{CBO}, \mathcal{CBL}$	$\mathcal{CBC}, \mathcal{CBOL}$
PostgreSQL with JSONB	$\mathcal{JCOL}$	N/A	N/A

Table 1. Collections per schema and JSON DBMS on experiment 1

Queries The study focuses on specific TPC-H queries, namely 1, 3, 4, 12, 13, and 22. These queries exclusively operate on the Customer, Orders, and Lineitem tables. Their parameters are strategically modified to enhance selectivity and facilitate index performance analysis. Additionally, a micro-benchmark featuring ad-hoc queries is introduced to investigate various issues identified during experimentation. Detailed discussions on these ad-hoc queries will follow.

To explore the effects of employing different operators, the experiment generates multiple query versions for document stores, simply referred to as “versions”. MongoDB and Couchbase’s optimizers generate query plans that closely mirror the original query structure, with minimal reordering of certain operations. Different combinations of operators are explored within MongoDB and Couchbase to assess their impact on query ordering. It should be noted that correlated sub-queries can be challenging, and sometimes impossible, to implement in Couchbase without representing the correlation as a join. Couchbase suggests utilizing a single bucket with

multiple data groups, each identified by selections over a "type" attribute. However, this approach introduces ambiguity since every join becomes a self-join.

To distinguish between these versions, a naming convention is adopted. For PostgreSQL, queries are denoted as `q<num>_psql` or `q<num>_psql_json`, where `<num>` corresponds to the TPC-H query number, and `json` indicates the use of the JSONB data type. For document stores, queries are named as `q<num>[v<ver>]_<engine>_<schema>`, where `<num>` represents the TPC-H query number, `<ver>` denotes the version number, `<engine>` is either `cb` for Couchbase or `mongo` for MongoDB, and `<schema>` specifies whether the query runs on S1, S2, or S3.

MongoDB versions serve the following purposes:

- Demonstrating the superior filtering method for arrays, comparing combinations like `$unwind` and `$match` versus `$project`, `$filter` and `$unwind`. Query versions 1 and 2 on S1 provide insights into this aspect.

- Assessing the reliability of multi-key indexes (i.e. indexes over array fields) in scenarios where documents are nested more than one level. Query version 3 on S1 serves this purpose.

- Investigating whether the "direction" of a join influences execution time. In MongoDB's aggregation framework, the starting collection of a pipeline defines the direction of a join. Reversing this direction requires re-creating the entire query. Query versions 1, 2, and 3 on S2, along with versions 1 and 2 on S3, help measure the impact of joining from different directions.

Couchbase exhibits documented limitations, such as the absence of join reordering[59], which historically lacked a cost-based optimizer until its introduction in the enterprise edition of version 6.5. While Couchbase's query versions, outlined in Table 3, don not directly address these issues, they aim to tackle undocumented challenges related to query predicate reordering for enhanced performance. These challenges encompass:

Schema	Version	Meaning
S1	1	Use $filter to filter objects within an array and $unwind to retrieve elements from within it.
S1	2	Use $unwind to retrieve elements from within an array and $match to filter them.
S1	3	Use an extra $match meant to trigger MongoDB's optimizer to use an index, if any.
S2	1	Start pipeline at $\mathcal{MO}$, lookup from $\mathcal{ML}$, and $\mathcal{MC}$ thereafter.
S2	2	Start pipeline at $\mathcal{MC}$, lookup from $\mathcal{MO}$, and $\mathcal{ML}$ thereafter.
S2	3	Start pipeline at $\mathcal{ML}$, lookup from $\mathcal{MO}$, and $\mathcal{MC}$ thereafter.
S3	1	Start pipeline at $\mathcal{MOL}$ and lookup from $\mathcal{MC}$.
S3	2	Start pipeline at $\mathcal{MC}$ and lookup from $\mathcal{MOL}$.

Table 2. Description of MongoDB's query versions

Schema	Query	Version	Meaning
S2	3	1	No explicit selection push-down on $\mathcal{CBC}$.
S2	3	2	Explicit selection push-down on $\mathcal{CBC}$.
S1	13	1	Filter orders using LEFT OUTER UNNEST and WHERE.
S1	13	2	Filter orders using the ARRAY operator.
S1	13	3	Explicit projection of orders after LEFT OUTER UNNEST and later filter using WHERE.
S2/S3	22	1	Compute customers without orders using an uncorrelated sub-query, store result set `LET` and match in `WHERE` using `IN`.
S2/S3	22	2	Compute customers without orders in the `WHERE` using an uncorrelated sub-query and match using `IN`.
S2/S3	22	3	Similar to version 2, but the average account balances are saved using `LET` prior to the `WHERE` clause and the selection over `c_phone` is included when computing the set of customers without orders.
S2/S3	22	4	Filter $\mathcal{CBC}$ based on `c_phone` predicate, then left join with $\mathcal{CBO}$ and select customers without orders. Here, the combination of the uncorrelated sub-query and `IN` operator present in previous versions is replaced by the left join and further selection described earlier.

Table 3. Description of Couchbase's query versions

- **Selection Push-Down within Join Predicate (Query 3):** This query involves straightforward aggregations, selections, and joins among the three tables Customer, Orders, and Lineitem. The primary bottleneck resides in the join operation. An effective optimizer should push down selections within the WHERE clause to select subsets of data from each table before performing the join. To validate Couchbase's optimization in this aspect, two versions of the query are created, one with selections pushed down and one without.

- **Filtering in Nested Structures (Query 13):** Query 13 evaluates conditions exclusively on table Orders, with its bottleneck in S1 being the `c_orders` array. Couchbase provides various methods to assess conditions within this structure, including flattening the array using `UNNEST` and filtering using `WHERE` or looping through the array and applying `ARRAY` filtering. To explore the efficiency of these alternatives, two versions of the query are devised. Version 3 additionally examines the performance of unnesting `c_orders` within a sub-query, followed by projection and filtering using `WHERE`, in comparison to Version 1.

- **Correlated Sub-Query (Query 22):** Couchbase's N1QL query language cannot express correlated sub-queries referencing documents from a different bucket. Consequently, such sub-queries need to be restructured as joins or employ `IN` with an uncorrelated sub-query. Four versions of Query 22 are designed to assess the performance of different operator combinations when expressing correlated sub-queries.

Results The graphs in Figures 12, 14, 15, 18, 19, and 20 depict the runtimes of MongoDB, PostgreSQL (both relational and JSONB versions) for six TPC-H queries across five different scale factors. Each figure contains two parts: a line chart and a bar chart. The line chart displays the running times for each implementation at all five scale factors, while the bar chart focuses on the 100G scale factor, providing a detailed comparison among query versions.

Couchbase versions are tested solely at the 1G scale factor due to performance limitations. Results presented for Couchbase represent the best outcomes achieved after extensive manual query rewriting. Figures 11, 13, 16, 17, 21, and 22 compare Couchbase's running times with other databases. The relational SQL version of each query serves as a baseline and is represented by a horizontal black line parallel to the x-axis. The white bar corresponds to the PostgreSQL JSONB version, while bars with similar gray tones represent versions in the same schema (S1,S2,S3). Bars with matching hatch patterns denote versions in the same database (star for Couchbase, inverted diagonal line for MongoDB, crossed diagonals for PostgreSQL). The y-axis is presented in logarithmic scale to accommodate the significant differences in runtimes, with Couchbase often exhibiting runtimes several orders of magnitude higher than MongoDB and PostgreSQL.

Throughout the experiments, a time limit of 24 hours was imposed, and queries exceeding this limit were terminated, with their runtimes not recorded. For versions on schemas S2 and S3 in MongoDB, where queries employing the `$lookup` operator tended to surpass the time limit, indexes were created on `o_custkey` (for $\mathcal{MO}$ and $\mathcal{MOL}$) and `l_orderkey` (for $\mathcal{ML}$) to enhance the operator's performance. All collections had an index on the `_id` field by default. Although this optimization reduced runtimes, some queries still exceeded the time limit. Furthermore, in PostgreSQL and Couchbase, indexes on foreign keys and linking attributes were created to ensure a fair comparison, referred to as "key-only indexes". Alternative strategies involving additional indices are discussed separately.

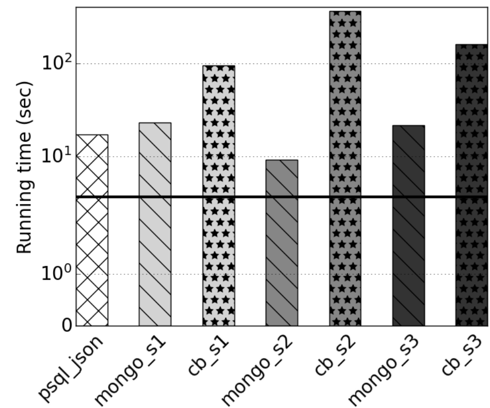


Figure 11. Running time (in log scale) of Q1 on all databases using scale factor of 1G

Below is a summary of the observed results for the tested queries:

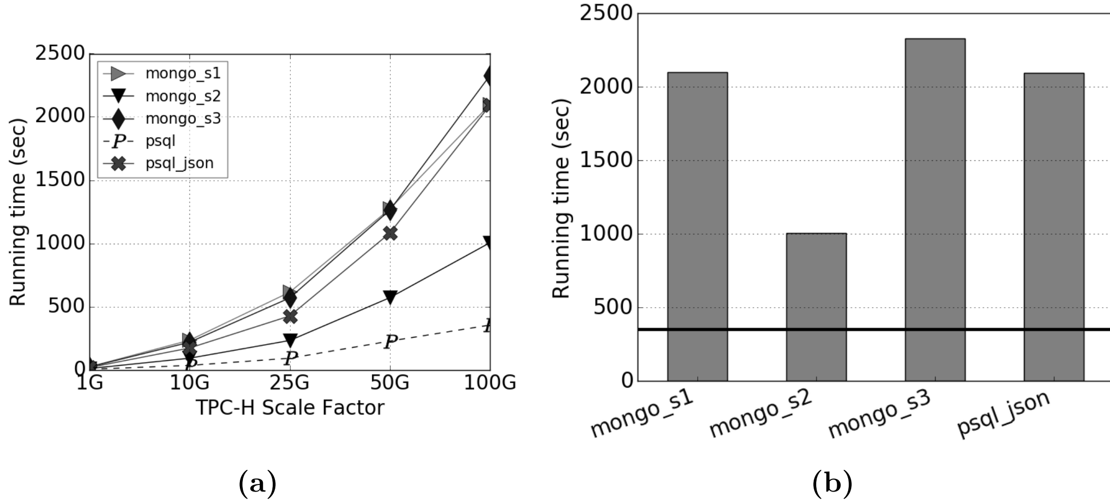


Figure 12. Running times of Q1 on MongoDB and PostgreSQL

Query 1 Since the query exclusively relies on attributes from the Lineitem table, the initial hypothesis is that versions on schemas S1 and S3 may be disadvantaged. In both scenarios, the target documents (lineitems) are stored within arrays as nested objects, necessitating the use of additional operators for retrieval. In MongoDB, these operators would include stages `$project` (with `$filter`) or `$unwind` followed by `$match`. In Couchbase, the `UNNEST` operator is employed. Conversely, queries executed on S2 work with smaller documents and do not require these supplementary stages. This accounts for their superior performance in MongoDB. However, it is worth noting that `q1_mongo_s2` is still over 50% slower than `q1_psql`, even in this scenario marked by efficiency.

Surprisingly, in Couchbase (as shown in Figure 11), this hypothesis does not hold. Depending on the intricacies of storage, scanning a larger number of documents can actually be more costly, especially when most of them need to be fetched from disk. Additionally, since the `UNNEST` operator operates in-memory, it does not significantly contribute to the overall cost. This phenomenon might explain why the Couchbase version on S2 is slower than on S3, and the version on S3 is slower than on S1.

Query 3 Schema S2 exhibits highly variable running times, both at the best and worst ends of the spectrum. Query plan analysis elucidates this behavior. Versions 2 and 3 on MongoDB perform notably poorly, with the latter failing to complete even at the 25G scale factor. In these versions, the joins are translated into two lookup (i.e. join) operations: the first scans $\mathcal{MO}$, and the second scans $\mathcal{ML}$ and $\mathcal{MC}$, respectively. In contrast, version 1 looks up $\mathcal{MC}$ first and $\mathcal{ML}$ second.

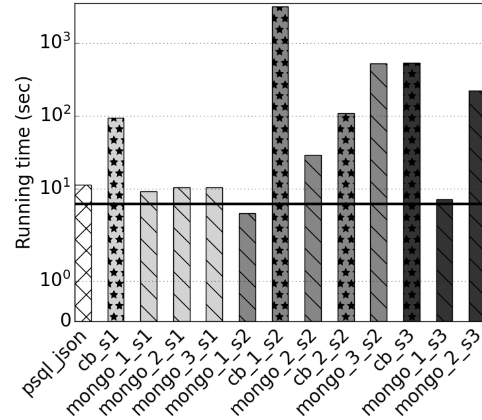


Figure 13. Running time (in log scale) of Q3 on all databases using scale factor of 1G

This divergence in performance can be attributed to the substantial size difference between $\mathcal{MO}$ and $\mathcal{MC}$, with $\mathcal{MO}$ being roughly 82.55% larger. Consequently, the scanning of $\mathcal{MO}$ consumes considerably more time, resulting in versions 2 and 3 spending a disproportionate amount of time on the first lookup compared to version 1. In fact, query 3 experiences a bottleneck when scanning any MongoDB equivalent of the Orders table, as evidenced by the same behavior in query `q3v2_mongo_s3`. However, due to the embedding of lineitems into orders in $\mathcal{MOL}$, a second lookup is unnecessary in this query, resulting in reduced running time compared to version 3 on S2. Nevertheless, `q3v1_mongo_s3` highlights that scanning $\mathcal{MC}$ is still more economical.

The observation is that `q3v1_mongo_s2` outperforms `q3v1_mongo_s3` because the cost of joining $\mathcal{MOL}$ with $\mathcal{MC}$ is significantly higher than joining $\mathcal{MO}$ with $\mathcal{MC}$, primarily due to the significantly larger document size in $\mathcal{MOL}$ compared to $\mathcal{MO}$.

This pattern appears to hold for the `$unwind` operator on S1 as well. The cost of unwinding documents with substantial nested documents is greater than joining two collections with smaller documents. In summary, `q3v1_mongo_s2` emerges as the fastest version of query 3 in MongoDB, owing to the order in which collections are

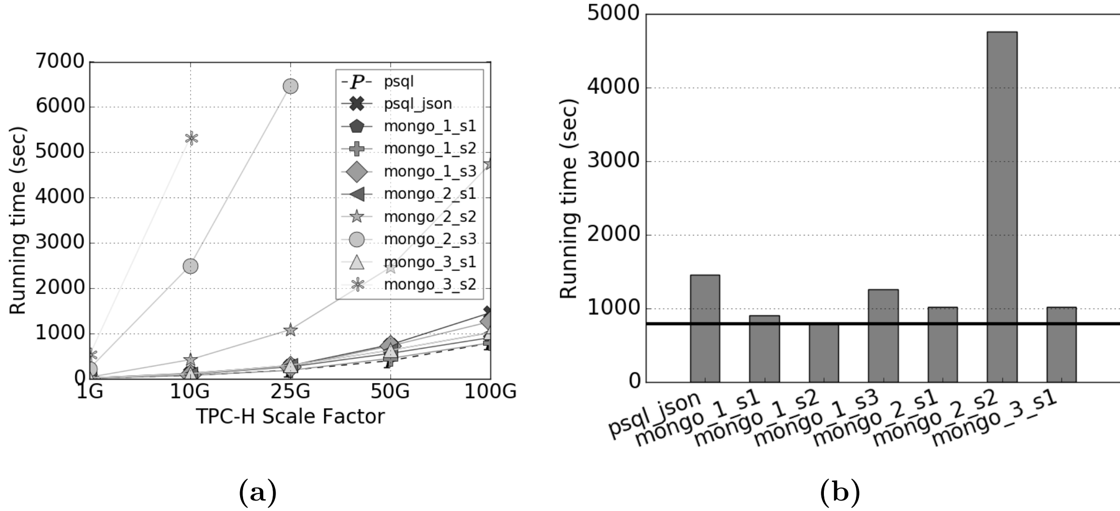


Figure 14. Running times of Q3 on MongoDB and PostgreSQL

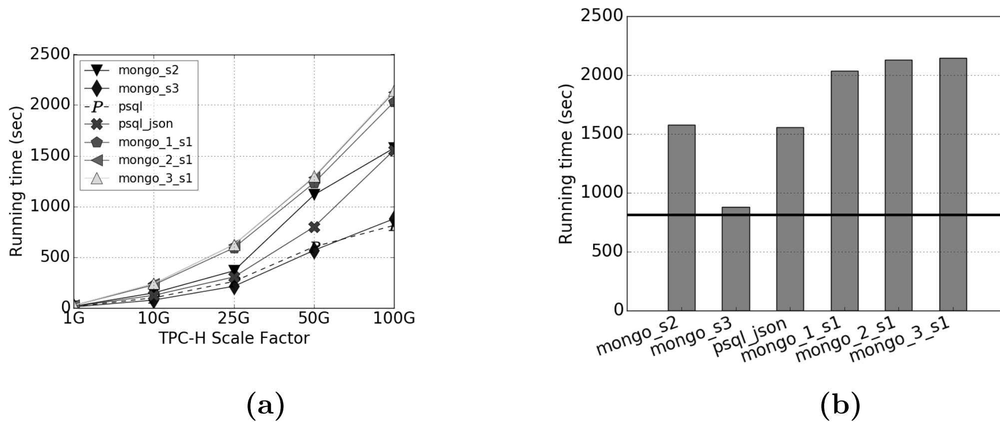


Figure 15. Running times of Q4 on MongoDB and PostgreSQL

joined and the document size of each one.

In Couchbase, Figure 13 illustrates that version 1 on S2 is slower than on S3 and S1. This is because the system retrieves all documents in the respective buckets of each schema. As noted earlier for query 1, it seems that scanning a larger number of documents in Couchbase has a considerably greater impact than the in-memory operators. Additionally, the system joins all documents before filtering, without pushing down any selection, which severely impacts performance. For version 2 on S2, all selections were explicitly pushed down, reducing the number of documents to fetch and join. Consequently, running time decreased to almost match that of the version on S1.

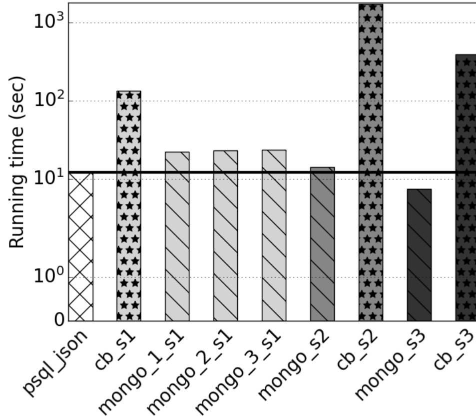


Figure 16. Running time (in log scale) of Q4 on all databases using scale factor of 1G

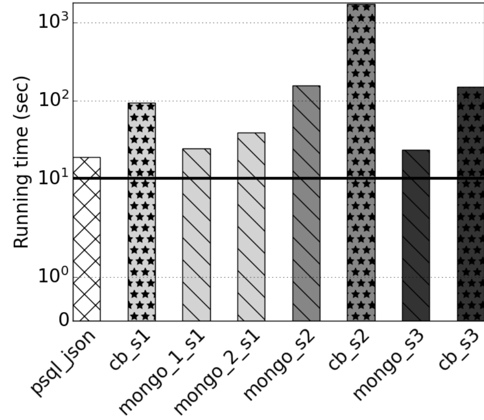
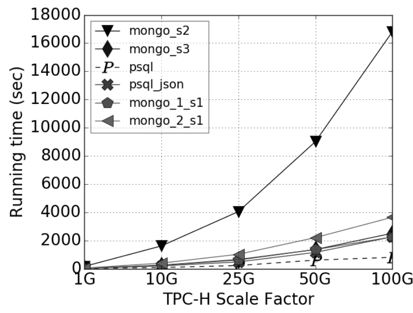


Figure 17. Running time (in log scale) of Q12 on all databases using scale factor of 1G

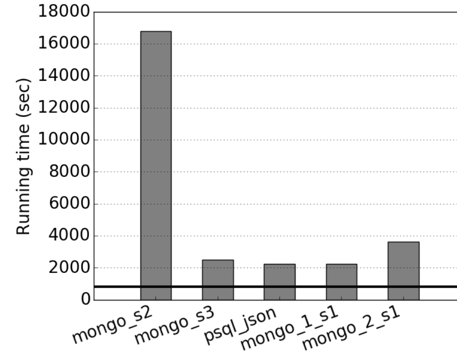
Query 4 In the case of `q4_mongo_s2`, a join is necessary because the query does not align with the schema $\mathcal{MO}$. Conversely, `q4_mongo_s3` benefits significantly from the design of S3 as it fits well within $\mathcal{MOL}$.

The query also fits within $\mathcal{MCOL}$, but queries on S1 require additional, seemingly costly steps: (1) the use of `$unwind` to deconstruct each array of orders per customer, and (2) the use of `$group` to re-group tuples of `o_orderkey` and `o_orderpriority` after unwinding `o_lineitems`. To elaborate, by deconstructing `o_lineitems`, new documents are generated per element in the array, each retaining all other fields. Consequently, tuples `o_orderkey` and `o_orderpriority` are no longer unique after this operation. The final result hinges on the count of such unique tuples, necessitating a re-grouping operation.

Couchbase tends to favor S1 due to the smaller number of documents to scan. Analogous to MongoDB, the query aligns with $\mathcal{CBOL}$ and $\mathcal{CBCOL}$, with the latter containing an order of magnitude more documents. This discrepancy results in `q4_cb_s3` being slower than `q4_cb_s1`. However, due to Couchbase's incomplete support for correlated subqueries, `q4_cb_s2` was modeled as a join between $\mathcal{CBO}$ and $\mathcal{CBL}$. Consequently, the system scans all buckets, performs the join, and filters afterward, elucidating its subpar performance.

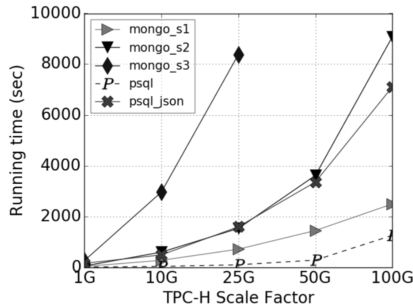


(a)

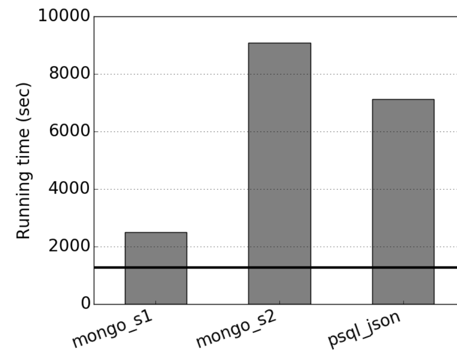


(b)

Figure 18. Running times of Q12 on MongoDB and PostgreSQL



(a)



(b)

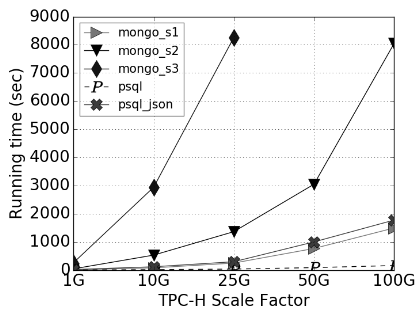
Figure 19. Running times of Q13 on MongoDB and PostgreSQL

Query 12 The MongoDB and Couchbase query versions exhibit notably slower performance on S2 compared to other versions. This outcome is expected since the act of joining $\mathcal{MO}$ and $\mathcal{ML}$, as well as $\mathcal{CBO}$ and $\mathcal{CBL}$, demands significant time due to their respective sizes.

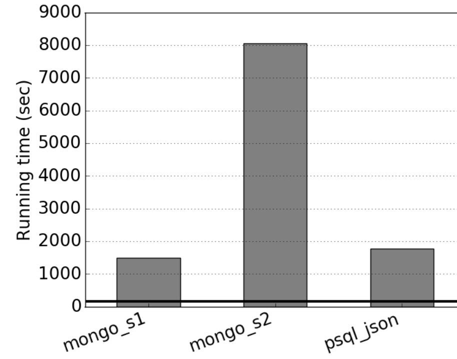
For MongoDB queries running on S1 and S3, they align with $\mathcal{MCOL}$ and $\mathcal{MOL}$, respectively. However, the former requires an additional unwind operation on the array of orders per customer, which contributes to its slower execution compared to the latter. The same rationale applies to JSONB.

In the case of Couchbase, scanning $\mathcal{CBCOL}$ proves to be more cost-effective than scanning $\mathcal{CBOL}$, resulting in faster query performance on S1.

Query 13 Due to the left outer join, the MongoDB pipeline must commence from



(a)



(b)

Figure 20. Running times of Q22 on MongoDB and PostgreSQL

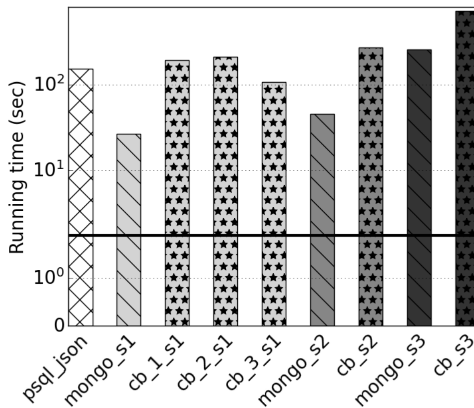


Figure 21. Running time (in log scale) of Q13 on all databases using scale factor of 1G

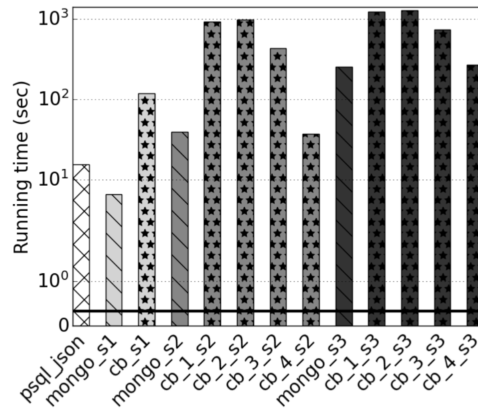


Figure 22. Running time (in log scale) of Q22 on all databases using scale factor of 1G

collection $\mathcal{MC}$ and perform lookups from $\mathcal{MO}$ for queries on S2, and $\mathcal{MOL}$ for queries on S3. As noted in Query 3, adopting this sequence results in suboptimal performance, as scanning any MongoDB equivalent of the Orders table in the `$lookup` operation proves to be slower than scanning $\mathcal{MC}$. A similar pattern emerges in Couchbase, as illustrated in Figure 21.

Notably, versions executed on S1 demonstrate shorter running times since they align with $\mathcal{JCOL}$, $\mathcal{MCOL}$, and $\mathcal{CBCOL}$, thereby bypassing the need for `$lookups` in MongoDB and left joins in Couchbase and PostgreSQL.

Additionally, Figure 21 underscores that filtering nested structures using `UNNEST` in

conjunction with WHERE in Couchbase outperforms the use of the ARRAY operator. Furthermore, the practice of projecting specific fields from orders in q13v3_cb_s1 is shown to be beneficial in reducing running times for subsequent aggregation steps.

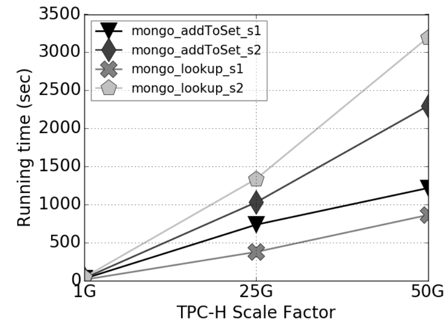


Figure 23. Running times of Q22 on MongoDB using the $lookup stage or the $group stage + the $addToSet operator

Query 22 As with Query 1, query versions operating over $\mathcal{JCOL}$ and $\mathcal{ACOL}$ face disadvantages compared to the plain relational approach. In MongoDB, all these versions necessitate a self-join at a specific juncture in the pipeline. However, after documents have traversed multiple stages, the $lookup stage must redundantly retrace the same steps leading to this juncture, resulting in considerable inefficiency. Specifically, while computing the average account balance, essential fields such as _id, c_acctbal, and c_cntrycode that are required later in the pipeline are lost due to the $group stage. Versions on S2 and S3 bear the brunt of this inefficiency since they must navigate through $lookup before self-joining, incurring the overhead of a nested $lookup stage.

Given this suboptimal performance, an alternative approach is devised to potentially enhance efficiency. This approach entails using the $addToSet operator instead of $lookup to retain the fields that would otherwise be lost.

For the queries conducted on schemas S1 and S2, the $lookup stage, meant to simulate a self-join, is replaced with a combination of the $group stage and the $addToSet operator. Figure 23 presents the running times of these queries. Versions employing $lookup include the keyword "lookup" in their names, while those utilizing $group + $addToSet are denoted with "addToSet".

The results underscore that the use of $addToSet reduces running times on S2 and S3 but increases them on S1. It is worth noting that using the $push operator instead

Query id	Objective	What the query does
a	Evaluate the performance of `$match` plus `$unwdind` against `$project` and `$filter` plus `$unwind`.	Select all lineitems where `o_orderkey` is equal to 7.
b	Evaluate how fast small and big documents are filtered	Select customer with `c_custkey` = 7 from $\mathcal{MC}$, and $\mathcal{MCOL}$.
c	Evaluate if the performance of joining a big collection with a small collection is affected by the direction of the join	Join pairs of collections in S2: $\mathcal{MC} \wedge \mathcal{MO}$ (version 1), $\mathcal{MO} \wedge \mathcal{MC}$ (version 2), $\mathcal{MO} \wedge \mathcal{ML}$ (version 3), $\mathcal{ML} \wedge \mathcal{MO}$ (version 4).
d	Evaluate performance of coalescing an `$unwind` within a lookup and then filtering vs projecting a subset of the array created by `$lookup` and then using `$unwind`	Select customer id and order id for orders where `o_totalprice` > 37500.

Table 4. Description of point queries

of `$addToSet` is also a feasible option. During experimentation, both approaches yielded similar performance when compared against `$lookup`.

The bottleneck observed in the first two versions appears to be primarily caused by the nested `$lookup` stage. Replacing it with `$addToSet` led to decreased running times. In the third version, while there is no nested `$lookup` stage, it involves filterings (`$match`) and a projection (`$project`). This combination appears to be less computationally intensive than using the `$addToSet` operator.

In Couchbase, due to the system's limitations, the correlated sub-query within the `EXISTS` clause had to be modeled as a join. Figure 22 illustrates that version 4 on S2 is the fastest among all Couchbase queries. In this version, the selection over `c_phone` was integrated into the uncorrelated sub-query, along with the predicate filtering `c_acctbal` based on the average of positive account balances. As all selections have been pushed down in this version, it retrieves fewer documents, resulting in improved performance.

Micro-benchmark In most queries, with the exception of 3 and 4, the PostgreSQL (relational) implementation consistently outperformed both MongoDB and Couch-

base by a margin exceeding 50% at all scale factors. To elucidate this result, a set of hypotheses is formulated and subsequently tested through a custom micro-benchmark comprising the queries outlined in Table 4. These specific queries are referred to as "point queries" and are described below:

- In the case of Q3, Q4, and Q12, Query version 1 on S1 outpaces version 2. It is postulated that projecting a subset of an array based on a condition and then deconstructing it is faster than first deconstructing it and then filtering the result. Point query a is devised to explore this hypothesis. In version 1, documents are filtered after unwinding field `c_lineitems`, while in version 2, a subset of the array is projected before unwinding it. The results, as depicted in Figure 24a, affirm the hypothesis by demonstrating that version 2 is faster than version 1.

- Drawing insights from the outcomes of Q3, it is conjectured that retrieving larger documents is slower than retrieving smaller ones. Point query b is crafted to test this hypothesis, measuring the filtering speed of attributes at the top level of both large and small documents. Figure 24b supports this assertion by revealing that scanning $\mathcal{MC}$ is faster than $\mathcal{MCOL}$, confirming the hypothesis as $\mathcal{MCOL}$ contains substantially larger documents.

- Q3 showcases a conspicuous difference in running time when comparing versions 1, 2, and 3 over S3. This leads to the belief that the order in which collections are joined significantly affects performance, particularly whether the order of collections used for an equijoin (or equality match) with `$lookup` has a tangible impact. Point query c is devised to investigate this notion. Figure 24c illustrates that initiating the pipeline at $\mathcal{MO}$ is faster than starting at $\mathcal{MC}$ or $\mathcal{ML}$. Although this observation may be influenced by collection size, our experiments cannot definitively confirm it. Nevertheless, it is evident that the join's direction plays a pivotal role in running time.

- Results from both the TPC-H queries and point query *a* indicate that the combination of `$project`, `$filter`, and `$unwind` (approach 1) is more efficient than `$unwind` and `$match` (approach 2). However, when filtering elements of an array generated by the `$lookup` stage, both approaches yield similar performance because the optimizer consolidates `$unwind` (from approach 2) into `$lookup`. Point query *d* is introduced to verify this, evaluating both approaches after a `$lookup`. Figure 24d demonstrates that there is no substantial difference between the two approaches.

Indexes To evaluate the influence of indexing, additional indices are incorporated into the dataset. Specifically, attributes employed in query selections are identified, leading to the creation of indices on `l_shipdate`, `l_commitdate`, `l_receiptdate`, `o_orderdate`, `c_mktsegment`, and `c_acctbal`. This approach is termed "extended indexing". However, it's important to note that the TPC-H Benchmark experiences limited indexing impact due to the presence of many attributes with low cardinality, resulting in reduced selectivity.

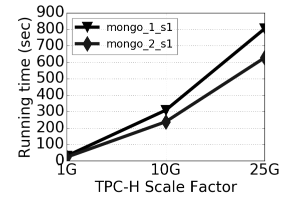
Extended indexes are established for relational and JSONB queries, as well as MongoDB and Couchbase queries on S1 (representing a typical schema for document stores). Notably, in MongoDB, only the first stage and the `$lookup` stage can benefit from indexing since they directly ingest documents from a collection. Consequently, these indexes are primarily effective for queries where one of these stages is used, specifically, queries 3 and 22.

The index selections made by each database are compared as follows:

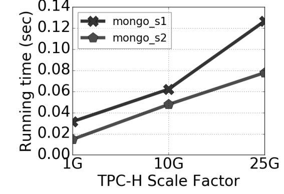
In the case of query 3, parameters `[DATE]=1992-01-02` and `[SEGMENT]= 'AUTOMOBILE'` are employed. The former is part of a condition involving `o_orderdate` and another involving `l_shipdate`. The first condition selects a mere 0.0414% of orders, while the second encompasses 99% of lineitems. The latter parameter relates to a condition involving `c_mktsegment` and selects approximately 19.83% of customers.

PostgreSQL relational queries opt for the index on `o_-orderdate`, while MongoDB and JSONB queries utilize the index on `c_mktsegment`. Table 5 reflects that PostgreSQL achieves superior speed-up as data size increases, underscoring the impact of better selectivity on performance. In contrast, MongoDB employs an available index indiscriminately, leading to suboptimal query plans. Indices in JSONB queries contribute to decreased performance, potentially attributed to PostgreSQL's inability to gather statistics on the JSONB type without the use of a functional index. In the case of Couchbase, all three indexes are scanned, with only documents matching the most selective predicate being fetched. This technique, known as "intersect scan", results in a significant speed-up by minimizing disk access.

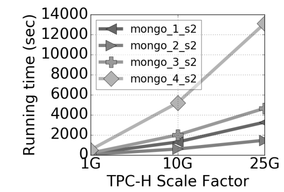
For query 22, the sole viable index selection pertains to `c_acctbal`. Two conditions are associated with this attribute: one filtering positive account balances (selecting around 90% of customers) and another filtering account balances greater than the average of positive account balances (selecting roughly 45% of customers). PostgreSQL queries universally employ the index related to the second condition, while MongoDB utilizes it for the first condition. This index usage pattern significantly impacts MongoDB's performance, as indicated in Table 5, further underscoring MongoDB's tendency to employ an index without considering selectivity.



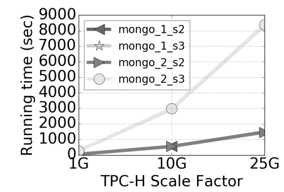
(a) Point query a



(b) Point query b



(c) Point query c



(d) Point query d

Figure 24. Running times of queries in the microbenchmark

Analysis Experimental results support the following points:

Query	SF: 1G	SF: 10G	SF: 25G
q3_psql	1.60%	26.71%	27.95%
q3_psql_json	0.78%	-11.15%	-6.05%
q3v1_mongo_s1	4.93%	9.16%	9.37%
q3v2_mongo_s1	2.43%	9.29%	9.62%
q3v3_mongo_s1	4.55%	10.41%	9.61%
q22_mongo_s1	-2944.67%	-2291.34%	-1992.62%
q22_psql	-1.66%	7.50%	1.27%
q22_psql_json	0.62%	23.69%	24.12%

Table 5. Speedup gained by using indexes. Negative percentages mean using indexes made the implementation slower

The performance of an arbitrary query, denoted as q, is not consistently superior when executed over a single collection compared to multiple collections. In the experimentation, queries over S1 typically do not necessitate joins, yet the absence of joins does not invariably translate to faster execution, as these queries often entail scanning a larger volume of data. Additionally, q may require the selection of data buried deep within a complex document structure, incurring the cost of deconstructing arrays.

A discernible trade-off becomes evident when contemplating database design, weighing the merits of employing a smaller number of intricate, deeply nested objects against multiple, albeit smaller, data units. This trade-off is substantiated by the results: when q comfortably fits within a collection in S2 or S3, its performance tends to be sluggish in S1. This phenomenon is exemplified in query 1, where versions executed on S2 outpace their S1 counterparts by a notable 50% margin across various scales. This result aligns with expectations, as q traverses a more compact dataset in S2.

In scenarios where q employs the `UNNEST` operation to extract fields from the deepest objects within S1 documents, its execution tends to be swifter in S2 or S3. This effect is substantiated by the outcomes of query 4, wherein the utilization of S2 resulted in a reduction in execution time by up to 22.4%, and S3 achieved an even more substantial reduction of up to 56% (both observed at the 100GB scale).

Notably, Couchbase's performance bottleneck primarily resides in the retrieval of

documents, a trend observable in all Couchbase queries lacking manual optimization, where the pattern of S1 surpassing S3, which in turn surpasses S2 in terms of speed prevails. Consequently, employing in-memory operators such as `UNNEST` proves advantageous, as these operators harness nested structures to accelerate performance by minimizing the need for document retrieval.

In a purely relational approach, the order of selections is contingent upon the sequence of joins. Conversely, in the document paradigm, it hinges on the structure of the document itself. While one can enforce a “top to bottom” data traversal pattern within the document structure without apparent performance degradation, this typically necessitates manual query optimization.

The selection between employing nesting (embedding) or linking in a database design hinges upon the database’s specific workload. Utilizing simpler objects distributed across multiple collections is advantageous for queries that exclusively access one collection but can lead to suboptimal performance when joins become necessary. Conversely, consolidating complex objects within a single collection eliminates the need for joins but can incur performance penalties, particularly when dealing with selective access to only specific segments of the object, particularly nested portions. In either case, meticulous schema design plays a pivotal role in optimizing performance.

In a database designed with multiple collections, it’s essential to accommodate join-like operations. This approach offers the advantage of enabling document-based databases to handle many-to-many relationships without excessive data duplication. However, it also introduces the challenge of efficiently supporting such operations. Unlike relational databases, document stores have inherent limitations and sometimes even bugs when it comes to joins (see `https://forums.couchbase.com/t/correlated-subquery-with-two-buckets/25160/12`). Thus, the choice of schema should carefully consider these factors. In a navigational query language, the order of joins is often dictated by the query’s structure, resulting in significant challenges when attempting to optimize queries with reordered operations.

For databases designed around a single collection of complex objects, particular attention must be paid to how the query language facilitates access and manipulation of object components.

In situations where a collection contains documents with one or more arrays of JSON objects, projecting a subset of these objects and then flattening them can be approached in two ways:

1. Deconstruct the array field(s) and discard documents that don't meet a specified condition, denoted as c.

2. Evaluate c against the objects within the array field during projection, creating new documents that include an array containing only the elements that satisfy the condition. Subsequently, deconstruct this refined array.

Option 2 offers an advantage by reducing the load during the array deconstruction phase, as it passes subsets of the original arrays. The experiment found that option 2 outperformed option 1.

When dealing with complex object selections, such as choosing customers based on specific embedded order conditions, there can be ambiguity. Depending on the query, two distinct output types may be desired: the customer object as-is (with all embedded orders) or the customer object with only those orders that meet the condition. Implementing separate operators for each case could potentially improve performance. In document stores, optimization largely relies on the query writer, as MongoDB and Couchbase, both representative of document stores, illustrate the lack of an effective optimizer capable of reordering predicates.

The optimal sequence of operations for collections is heavily contingent on the specific schema of those collections, as exemplified by Query 22. Interestingly, one particular ordering of operations led to reduced query runtimes on S2 but had the opposite effect on S1. Two potential optimizations, although not currently employed, could significantly enhance query performance:

1. In instances where the query selects elements after their deconstruction from an array, the query planner could potentially reorganize the sequence. It might transform this sequence into a projection that first filters the elements within the array, followed by the deconstruction process.

2. When a query involves a join, the query planner could explore the possibility of changing the direction of the join operation. However, it's important to note that transforming a left outer join into a right outer join (or vice versa) can be complex, often necessitating a complete query rewrite in the worst-case scenario.

Certain query patterns pose unique challenges in a document-oriented context. Consequently, advanced operators without direct SQL equivalents become indispensable. Queries originating from S1, S2, or S3 may require self-joins; for example, Query 22 in MongoDB involves an uncorrelated sub-query that essentially translates into a self-join. In Couchbase's recommended setup, every join operation is essentially a self-join. The experimental results have demonstrated that this approach negatively impacts schemas S2 and S3.

These observations highlight two key points:

1. The concept of a "self-join", a well-known technique in relational databases, does not seamlessly translate to document-oriented systems, particularly within navigational query languages.

2. The process of translating complex SQL queries into these document-oriented systems can be fraught with challenges.

Experiment 2: Schema-less does not imply schema-free

This experiment underscores the importance of database design in enhancing performance, even when working with schema-less data. It is evident that not all designs

are universally optimal for complex, ad-hoc queries. Rather, a specific schema can improve the performance of one query at the cost of another, as implied in Section III.1.

The prior experiment provided insights into how MongoDB performs compared to traditional relational databases in an analytical query environment, a domain traditionally dominated by RDBMSs. However, the rise of column-oriented RDBMSs for analytical querying, with their efficient access to data when only a subset of columns is needed, merits exploration. MongoDB, Couchbase, and other prominent JSON DBMSs typically store records contiguously, akin to the row-wise storage of traditional RDBMSs. This experiment also shows the performance gap that document stores like MongoDB need to bridge compared to column-oriented RDBMSs.

In this context, MongoDB is compared against MariaDB Column Store, with PostgreSQL also included for comprehensive evaluation. Notably, Couchbase is excluded from this experiment due to its subpar performance in the previous analysis.

This experiment retains the three schemas from the previous study and additionally addresses the M-N case (TPC-H queries involving tables Part, Supplier, and PartSupp).

Consequently, the analysis encompasses the following queries:

- Queries 1, 3, 4, 6, 12, 13, 18, and 22, examining the design of hierarchical data (i.e., 1-M relationships).
- Queries 2 and 11, delving into design choices for M-N data.
- Queries 5 and 9, offering insights into a combination of both 1-M and M-N scenarios.

As in the previous experiment, query parameters are adjusted to enhance selectivity. The queries are named following the convention `q<num>[v<ver>]_<engine>[_<schema>]`:

- `<num>` corresponds to the relevant TPC-H query number.

- `<ver>` represents the query version.
- `<engine>` is either “mgo” (MongoDB), “psql” (PostgreSQL), or “mcs” (MariaDB Column Store).
- `<schema>` refers to queries executed on S1, S2, or S3.

Results The results are categorized into three groups: queries involving hierarchical data (Customers, Orders, and Lineitems), queries pertaining to M-N data (Suppliers, Parts, and PartSuppliers), and mixed queries that encompass both hierarchical and M-N data. Each query description corresponds to its original SQL formulation.

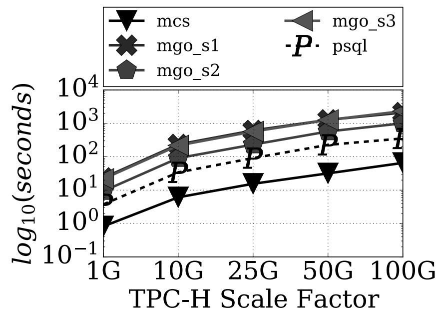
It is worth noting that the test design deliberately pushed the data to sizes where it no longer fit into memory, albeit at varying points for different systems. MariaDB, thanks to its columnar approach, exhibited consistent performance across all sizes. In contrast, data stored in JSON format, as utilized by MongoDB, proved notably larger than the standard row-based relational storage in PostgreSQL. Consequently, it necessitated disk spillover in some of the tests.

For a detailed presentation of the results, please refer to Figures 25 and 26.

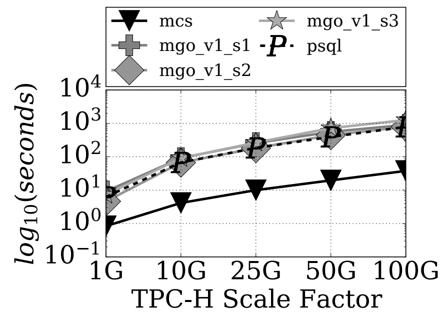
Hierarchical data Some of these results have been previously discussed in the earlier experiment, but the present study expands upon them by introducing additional queries (6 and 18) and a new system (MariaDB with columnar storage). A summary of these findings is provided here.

In this summary, MongoDB query versions retain the same meaning as in Table 2, but only the fastest query version for each schema is presented in Figure 25. It is essential to note that, due to time constraints, the newly included queries were executed with a time limit of 10 hours instead of the 24-hour limit applied in the previous experiment. Consequently, MongoDB versions at the 50G and 100G scale factors are not depicted in Figures 25d and 25g.

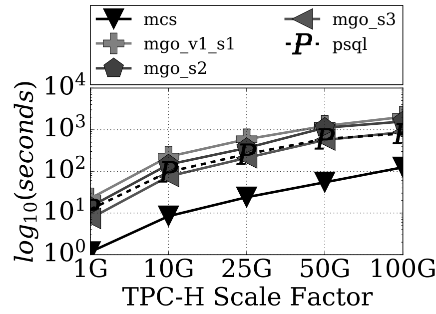
The columnar approach, represented by MariaDB, outperformed other systems in all cases. This outcome aligns with expectations, as the TPC-H queries exhibit



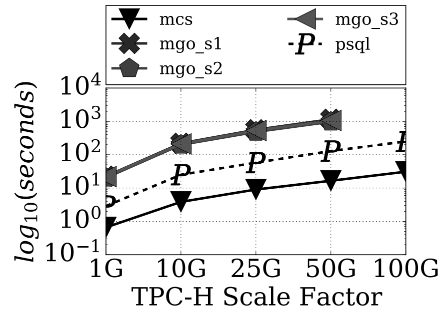
(a) Running time query 1



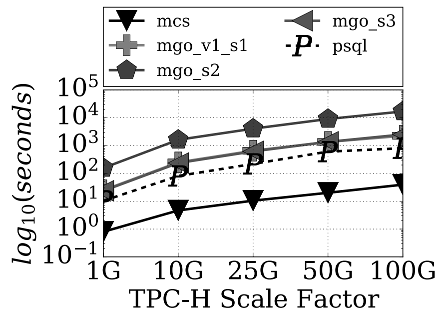
(b) Running time query 3



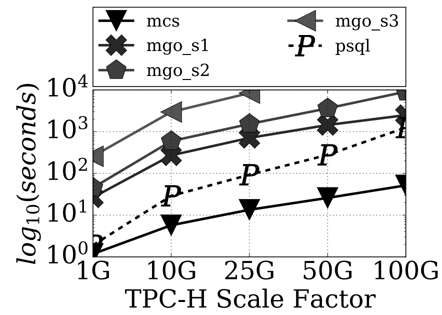
(c) Running time query 4



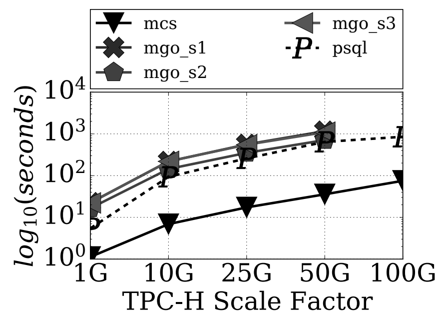
(d) Running time query 6



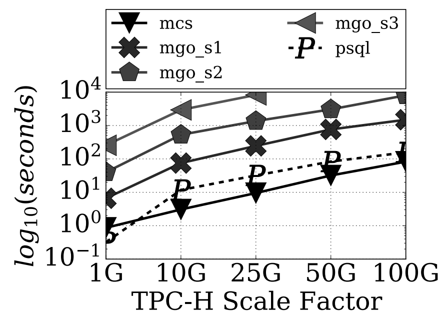
(e) Running time query 12



(f) Running time query 13



(g) Running time query 18



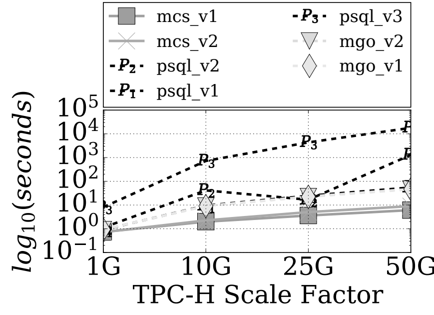
(h) Running time query 22

Figure 25. Running times (in log scale) of queries over 1-M data

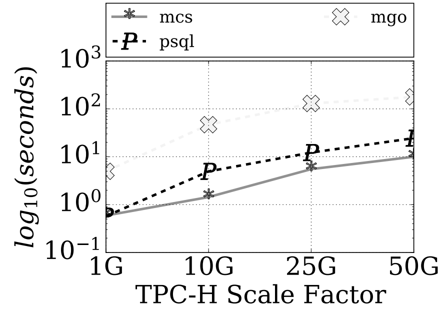
a relatively "narrow" nature, i.e., they utilize only a few attributes from each referenced table. This characteristic benefits the columnar approach, as it circumvents the retrieval of entire records when only a subset of values is required. In contrast, MongoDB's performance consistently fell behind that of both MariaDB and PostgreSQL. As discussed in the subsequent analysis, the experiment has highlighted the cost associated with avoiding joins by consolidating all data into a single collection. This approach results in increased scanning, even when only a small portion of the data is needed. Moreover, accessing the nested components of complex objects, and potentially restructuring them, proved to be an expensive operation for MongoDB. This is vividly illustrated in Figures 25a and 25d, where queries 1 and 6, operating primarily on deeply nested schemas (S1 and S3), encountered performance issues due to their extensive interaction with lineitems located at the deepest level. Additionally, the MongoDB query optimizer missed several opportunities to enhance performance, especially in cases involving left joins and sub-queries, as exemplified by queries 13 and 22 (Figures 25f and 25h).

A noteworthy observation pertains to MariaDB's query processing approach. It systematically executes sub-queries within the `WHERE` clause before assessing selections. Furthermore, when multiple sub-queries are present within the `WHERE` clause, they are evaluated in the order they appear.

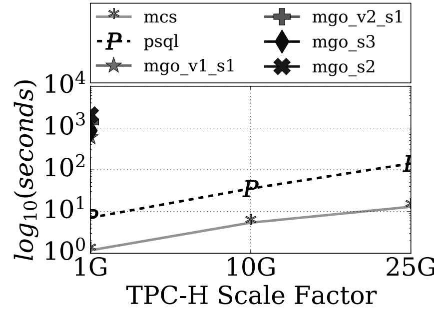
M-N data Queries 2 and 11 are illustrated in Figures 26a and 26b. Query 2 is presented in various versions. Notably, MariaDB necessitates a distinct approach due to its inability to perform a semi-join and scalar filter from different tables (or joined tables). As a result, the query is redefined using two approaches: one version unnests the subquery in the `WHERE` clause (`q2v1_mcs`), and the other version defines a common table expression (CTE) in the unnested version (`q2v2_mcs`). Note that MariaDB does not inherently unnest this query. Furthermore, the unnested and CTE versions of query 2 are also provided for PostgreSQL: `q2v1_psql` (original TPC-H query), `q2v2_-psql` (unnested), and `q2v3_psql` (CTE). In the case of MongoDB, two versions of



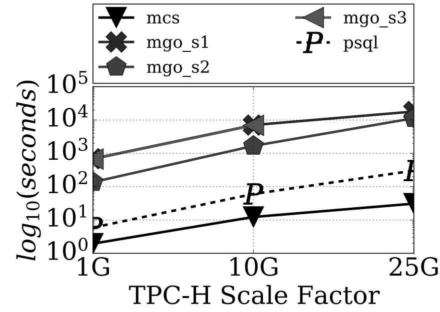
(a) Running time query 2



(b) Running time query 11



(c) Running time query 5



(d) Running time query 9

Figure 26. Running times (in log scale) of queries over M-N data (Q2, Q11) and mixed; 1-M and M-N (Q5, Q9)

query 2 are provided. One “navigates” from Supplier to Part (initiating the query at collection $\mathcal{MS}$), while the other navigates from Part to Supplier (initiating the query at collection $\mathcal{MP}$). After determining the more efficient version, an additional iteration is created that introduces extra projections to reduce the document size before each `$unwind` and `$lookup` operation.

As usual, the columnar approach proves to be the fastest. Notably, `q2v2_mcs` showcases that utilizing the CTE in MariaDB enhances performance as less work is involved. The CTE’s impact on PostgreSQL is adverse because (1) the query planner does not extend optimizations to the CTE, and (2) the system materializes it, subsequently applying the filters. `q2v1_mongo` and `q2v2_mongo` compete with `q2v1_psql` (the fastest PostgreSQL version of query 2). This can be attributed to several factors. Firstly, the sizes of $\mathcal{MS}$ and $\mathcal{MP}$ are sufficiently small to be stored in memory

across all scale factors, reducing the size disparity between datasets in relational and JSON formats. Secondly, the schema allows for the selective push of $\mathcal{MP}$'s selection up in the pipeline (i.e., pushing down the selection over table Parts). Both of MongoDB's top-performing versions of this query commence from $\mathcal{MP}$, utilizing a projection with the `$min` operator to identify the minimum supply cost per part without performing an unnest operation. This strategy diminishes the number of documents to process and circumvents a costly `$group` operation that might otherwise exceed the 100MB RAM limit per operation as the dataset size expands. It was confirmed that the performance substantially deteriorates if attributes from $\mathcal{PS}$ are absent in $\mathcal{MP}$, necessitating additional lookups for the aggregated attribute. It is worth noting that `q2v2_psql` exhibits anomalous behavior between the 10G and 25G scale factors because the query planner utilizes a resource-intensive sequential scan over Supplier at 10G, but transitions to an index scan on Supplier's primary key at 25G, resulting in an improved runtime. This was rectified by enforcing the mentioned index scan, making this query version faster than MongoDB's and `q2v1_psql`.

Query 11 originates from $\mathcal{MS}$ in MongoDB and filters documents based on the nation name constraint from the original query. It subsequently joins $\mathcal{MP}$ and employs the `$facet` operator to process multiple aggregations from the same input documents, one for the uncorrelated subquery in the `HAVING` clause and another for the outer query. An alternative approach would involve using `$lookup`'s special "pipeline" argument, designed for such cases. However, this approach currently exhibits performance issues, as of the time of this writing[1]. Unlike query 2, the schema choice hinders query 11 as it necessitates the merging of $\mathcal{MS}$ and $\mathcal{MP}$, which would be unnecessary if attributes from $\mathcal{PS}$ had been embedded in $\mathcal{MS}$. It was confirmed that transferring attributes from $\mathcal{PS}$ to $\mathcal{MS}$ can significantly enhance query performance. Nonetheless, this would involve altering the database's schema to accommodate each query's requirements.

[1] `https://jira.mongodb.org/browse/SERVER-41171`

Mixed 1-M and M-N data Queries 5 and 9 are depicted in Figures 26c and 26d, respectively. In the case of MongoDB, schemas S1, S2, and S3 were explored within the hierarchical component, while the same schema was used for M-N data.

In the context of MariaDB, a modification was made to the join condition of query 5, specifically, the condition was adjusted as "`and c_nationkey = s_nationkey + 0`" due to a restriction disallowing circular joins[2].

Across both queries, columnar storage consistently exhibits the best performance, closely followed by PostgreSQL.

Query 5 involves the computation of an aggregate over table Lineitems, with filtering based on `o_orderdate` in the Orders table and `r_name` in the Region table. This structure benefits `q5v[1,2]_mongo_s1` in MongoDB because the join between $\mathcal{MS}$ and $\mathcal{MCOL}$ can be facilitated through the `c_nation` attribute. However, initiating the pipeline from $\mathcal{MS}$ instead of $\mathcal{MCOL}$ (`q5v2_mongo_s1`) results in performance degradation, due to the substantial document size in the latter. `q5_mongo_s2` and `q5_mongo_s3` apply a filtering step to orders but need to perform lookups for $\mathcal{MC}$ before $\mathcal{MS}$—an additional join compared to the S1 versions. The queries on S2 and S3 encompass the same steps as `q5v1_mongo_s1`, plus the added lookups, rendering them slower. The rationale for this performance difference lies in the fact that S1 allows for the traversal of all hierarchical data from outer to innermost nested documents without unnecessary intermediary steps.

Query 9 bears resemblance to query 5 in that it computes aggregates over table Lineitem, but with the sole selection on `p_name` over table Parts. Consequently, `q9_mongo_s1` necessitates an additional unnest operation. Furthermore, as there is no join condition based on customer attributes, the pipeline is compelled to commence from $\mathcal{MCOL}$. In the case of `q9_mongo_s3`, the pipeline must initiate from $\mathcal{MOL}$ as lineitems need to be unnested for the $\mathcal{MS}$ join. Notably, `q9_mongo_s2` outperforms the other two versions due to its initiation from $\mathcal{MP}$ and preliminary part filtering,

[2] `https://jira.mariadb.org/browse/MCOL-1205`

effectively reducing the documents to be joined from $\mathcal{ML}$ and $\mathcal{MS}$.

Analysis The database's design significantly impacts performance and can substantially constrain query composition depending on the query language's design.

Queries involving hierarchical data dispel the notion that an arbitrary query will always perform more efficiently in a single collection compared to multiple collections, despite the absence of joins. In both document stores and traditional RDBMSs, the storage is row-wise, meaning that queries necessitating only a subset of attributes at any hierarchy level must retrieve all nested data from disk, leading to unnecessary data retrieval. This problem is exacerbated in document stores when dealing with queries that access deeply nested attributes and require multiple unnest operations. These unnests can generate numerous intermediate results that quickly overwhelm memory. In contrast, column-based storage, where data is decomposed into minimal units (single attribute values) and assembled as needed by the query[75], exhibits superior performance.

Queries 2 and 11 illustrate the considerable performance gains when attributes are accessible in a navigational data path. For example, the storage location of attributes from PartSupp (analogous to either the Part or Supplier table) makes a significant difference: when navigating from Part to Supplier, the aggregated attribute (supply-cost) is available for computing aggregation and further constraining data retrieval, leading to better MongoDB performance than Postgres (the only scenario where MongoDB outperforms Postgres). However, the performance significantly degrades when queries navigate from Supplier to Part. Since different queries favor different access patterns, no single design can be deemed superior to others.

Irrespective of the chosen design, operators required for navigating any design must be efficiently supported. Specifically, if a database is designed using several collections, efficient support for join-like operations is necessary, encompassing full support for all types of operations. Unlike joins in relational databases, CouchBase

ANSI JOINs exhibit several limitations, including system bugs[3]. MongoDB's `$lookup` operations must be performed in specific orders. In a navigational language, the query structure's order is constrained by how the query is written, making order inversion necessitate a total query structure reconstruction, which can lead to missed optimization choices like selection push-down.

CouchBase's case is particularly instructive; despite offering a declarative, SQL-like language, it performed significantly worse than other systems. The analysis reveals that CouchBase's query optimizer struggles to translate SQL-like queries into optimal access paths, requiring efficient algorithms for inter-document and intra-document operations as well as their efficient combination.

This also elucidates why "no join" is not always the best strategy. When a database is designed using a single collection of complex objects, careful consideration must be given to how the query language allows access and manipulation of object "parts". The experiment demonstrates that selecting a subset of JSON Objects from an arbitrary collection containing documents with one or more arrays can be achieved through two approaches: (1) Deconstructing the array field, generating a new document for each object within it, followed by a selection to discard non-compliant documents. (2) Evaluating the given condition against the objects within the array field in a projection and returning new documents with an array containing only matching elements, followed by deconstructing the projected array. Option (2) outperformed option (1) in MongoDB, reducing running times by more than 35%, as exemplified by query 12 over S1. Ideally, the system should determine which plan best suits a particular query, as both options implement the same conceptual operator.

It is worth noting that in several instances MongoDB can achieve good performance when everything is "lined up" its way, like in query 2. However, in general its optimizer is not flexible enough to take advantage of possible optimizations: it does not achieve optimal ordering of operations (which may depend on the schema of the

[3] `https://forums.couchbase.com/t/correlated-subquery-with-two-buckets/25160/12`

collection); in its navigational language, choosing the order in which to access the data is crucial, as query 2 and query 5 results show. In a complex data model (like the one used by document stores), designing an optimization engine that can consider several alternative plans for a query is very challenging, but also (as our results show) very necessary.

Based on this analysis, query optimization for document databases can be advanced by improving:

- the ability to rewrite entire queries to select the most efficient data access path in navigational languages (e.g., MongoDB).
- the mapping of relational operators to object and collection-based operations for declarative languages (e.g., Couchbase).

To support both of these objectives, it is advisable to develop a formally defined algebra for objects. This algebra should encompass intra and inter-object operations, and its properties, such as operator commutativity, should be formally studied. This foundation would enable the establishment of legal optimization strategies for the query optimizer. Existing research, like that of [28], can offer valuable insights in this context.

Additionally, it is important to acknowledge that schema design in document stores directly impacts physical data layout. This introduces a challenge, as the chosen storage structure may not be optimal for certain queries. Considering whether the storage engine should be granted more flexibility is a thought-provoking perspective. For instance, breaking down complex objects with deeply nested components into separate storage units may enhance query performance.

2 Proposed framework

As previously noted, many approaches to query optimization in document stores follow the traditional relational framework. This framework includes the conversion

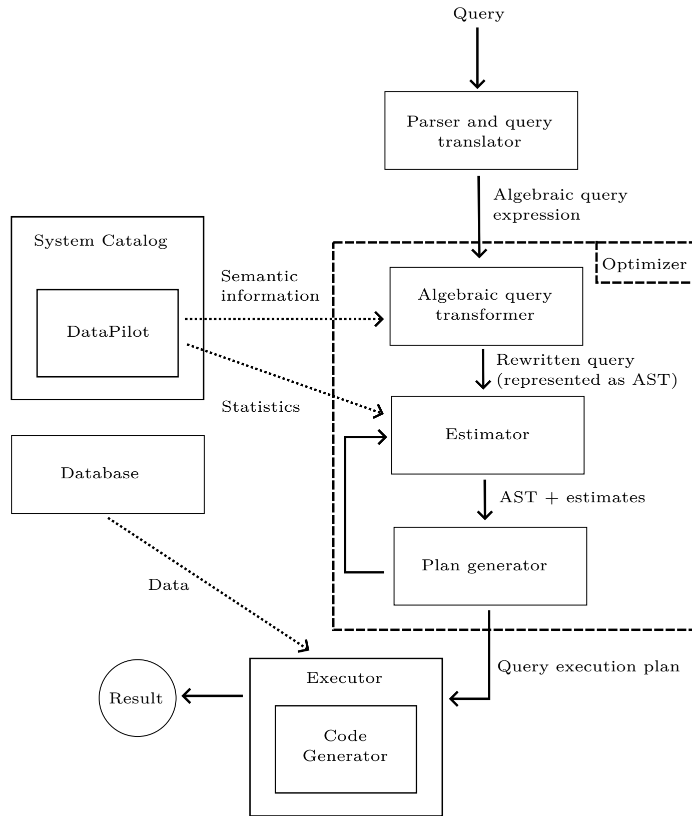


Figure 27. Proposed framework's query processing workflow

of queries into an internal representation, the establishment of transformation rules or equivalences, the development of query plans for these expressions using abstract operators, and for systems employing cost-based optimization, assigning associated costs.

This thesis proposes an instance of this framework tailored to JSON data, with the query processing flow illustrated in Figure 27. Each component of the flow is described as follows:

Parser and query translator: In an RDBMS, the parser generally performs minor query adjustments. This includes rewriting table names in the format `<database>.<schema>.table`, replacing table aliases with the rewritten name, and validating references to tables and attributes. The system catalog helps the parser check if a referenced table exists and if the attributes referenced are correct. It also ensures schema compatibility between tables for set operations like union or

intersection.

For JSON databases, the existence of attributes in a table is determined by the presence of paths in the structural summary provided by the DataPilot structure inside the Catalog. Schema compatibility for operations like joins is checked based on the DataPilot. Unlike RDBMSs, JSON databases typically do not enforce schema compatibility for operations like union.

After these checks, the query is converted into a parse tree structure. For example, an SQL parse tree structure is illustrated in Figure 28. The query is pre-optimized through techniques such as constant arithmetic simplification (e.g., `WHERE age > 25 + 10` is converted to `WHERE age > 35`), subquery flattening, where the query is rewritten to eliminate subqueries if possible (e.g. see Figure 29), and other heuristic-based optimizations. In JSON DBMSs, the specific syntactical optimizations depend on the query language used. While some techniques developed for SQL can be adapted for SQL-like languages like SQL++, navigational languages like MongoDB's aggregation pipeline may require approaches more akin to XML's Xpath.

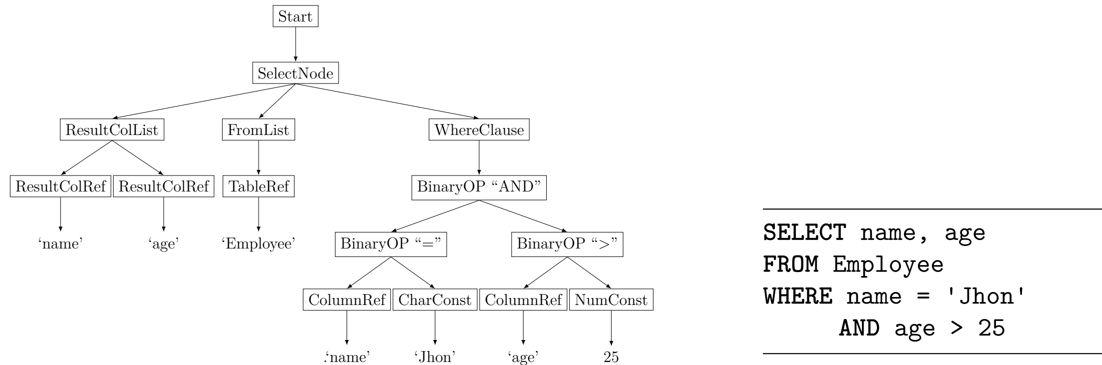


Figure 28. Example of a SQL parse tree

This parse tree is then translated into an Abstract Syntax Tree (AST) for an algebra belonging to the system's data model and passed down to the algebraic query transformer.

Algebraic query transformer: The AST is transformed by applying optimization techniques based on semantic rules (i.e. rule-based optimization). In RDBMSs,

```sql
SELECT Employee.*
FROM Employee
WHERE Employee.E-Dept IN (
  SELECT Department.D-ID
  FROM Department
  WHERE Department.D-Name = 'HR'
)
```

(a) Before subquery flattening

```sql
SELECT Employee.*
FROM Employee, Department
WHERE Employee.E-Dept = Department.D-ID
      AND Department.D-Name = 'HR'
```

(b) After subquery flattening

Figure 29. Example of subquery flattening in SQL

these semantic rules are properties of relational algebra. For the case of JSON, this thesis proposes a specialized algebra designed for JSON data, featuring operators structured into multiple levels for independence from data representation. The first level focuses on individual JSON trees, enabling the manipulation of these trees, and the conversion of trees into an Abstract Data Type to decouple them from the data storage representation. The second level extends nested relational algebra, adapted for sets, bags (i.e. multi-sets), and their ordered counterparts, to effectively handle collections of documents. This level aims to provide a robust foundation for document collection manipulation, establishing operator properties, and ensuring compatibility with various query languages. The third level introduces macro-operators, combinations of operators that can be efficiently implemented and are utilized in optimization. A relational algebra equivalent of these macro-operators are derived operators like `JOIN`, which is equivalent to a `CARTESIAN PRODUCT` followed by a `SELECTION`. An example of some rule-based transformations in RA is shown in Figure 30 for illustrative purposes. This separation into primitive (second level) and derived (third level) operators simplifies property verification and expression manipulation.

System Catalog Module: In RDBMSs, this module stores essential table and index metadata, such as schema details and statistical information, including summary structures like histograms. The algebraic query transformer and estimator modules request metadata as needed.

For JSON databases, this thesis introduces the concept of a DataPilot. Essentially,

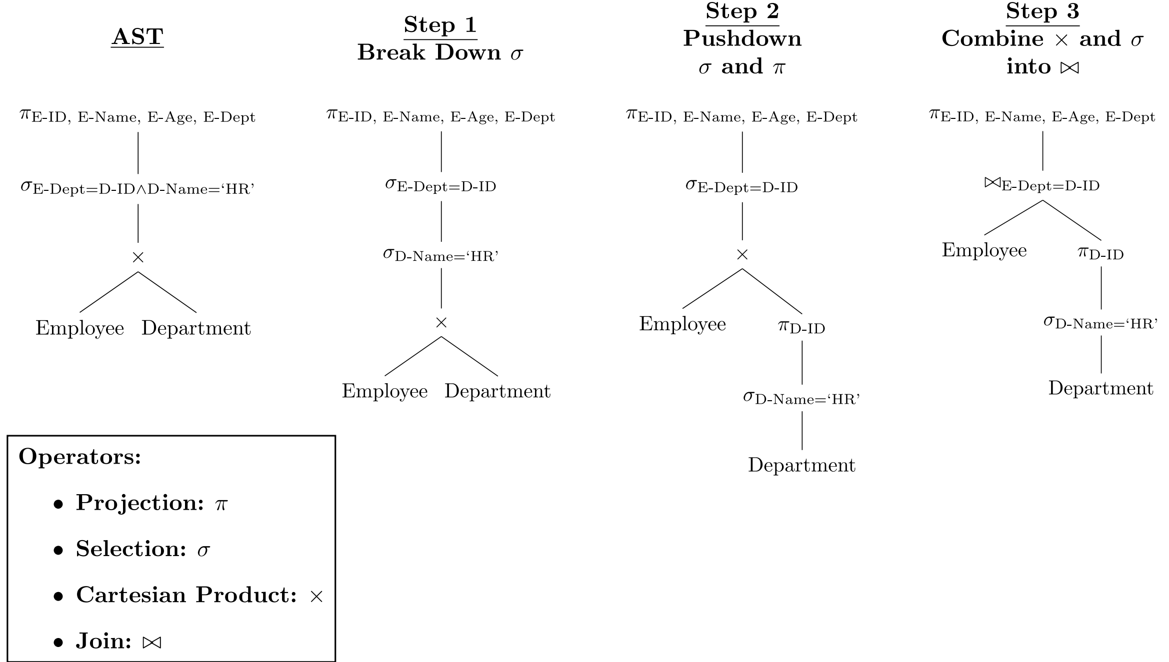


Figure 30. Example of SQL rule-based transformations

it is an adaptation of an XML DataGuide structure tailored to JSON, serving as a repository for structural insights about collections and associated statistics. While previous work by Klettke ([46]) and Baazizi ([10]) has explored the utilization of DataGuides adapted for JSON, this thesis employs this metadata for both rule-based and cost-based query optimization.

To enable this, an algebra specifically for DataPilots is formulated. This algebra features operators analogous to those in the second and third level algebras mentioned earlier. These DataPilot operators work in conjunction with the structure and metadata of the DataPilot, offering estimations of the DataPilot generated from the collection after applying the corresponding collection operator to the data.

For instance, consider an arbitrary collection, denoted as $\mathcal{C}$, the DataPilot computed from this collection as $dp^{\mathcal{C}}$, and OP representing a unary algebraic operator applied to $\mathcal{C}$, then within the DataPilot algebra, $OP(dp^{\mathcal{C}})$ approximates $dp^{OP(\mathcal{C})}$. This

ensures that the following diagram commutes:

$$\begin{array}{ccc} \mathcal{C} & \longrightarrow & dp^{\mathcal{C}} \\ {\scriptstyle OP}\downarrow & & \Downarrow{\scriptstyle OP} \\ OP(\mathcal{C}) & \longrightarrow & dp^{OP(\mathcal{C})} \end{array}$$

Estimator and Plan generator: The estimator leverages DataPilot statistics and available indexes to predict the cardinality of each operator in the AST, providing an estimate of the expected number of records returned by each operator. If a relevant index is available, it takes precedence over the DataPilot.

The estimator can source information from both indexes and the DataPilot. Once it receives the AST from the algebraic query transformer, it augments the AST with appropriate estimates and passes it to the Plan Generator. The Plan Generator then explores various logical and physical plan alternatives using plan enumeration algorithms, primarily focused on join reordering and often employing dynamic programming.

For each operator, there exist multiple physical-level code implementations that realize the operator's logical-level behavior. Each implementation is associated with its own cost function, which the plan enumeration algorithm employs to select the optimal plan. In the selected plan, the algebraic operators in the AST are substituted with nodes representing the specific physical implementations for each operator.

Executor: The plan generator's chosen optimal plan is forwarded to the executor, which consolidates all the physical operator templates from the AST into a compilable program using its code generator module. This program is then compiled and executed to perform the query. In systems with multiple storage engines, the storage module typically offers an API to access the underlying data and collect information regarding the capabilities of the storage backend (e.g., scanning, range access, point access, indices) where the data is located. The executor leverages this information to customize the code accordingly. Take, for instance, the two physical plans exemplified in Figure 31. These plans illustrate two distinct (pseudo-code) im-

plementations of the join logical operator: Hash Join and Nested-loop Join. Note how various logical operators on the right side of the join are coalesced together: Selections are transmuted into `if` statements nested within `for` loops that iterate through the in-memory representation of tables. Implicitly intertwined in these `for` loops is the access method employed to fetch each record from the disk. This access method, along with the functions `getAttribute` and `getValue`, is part of the API provided by the storage module.

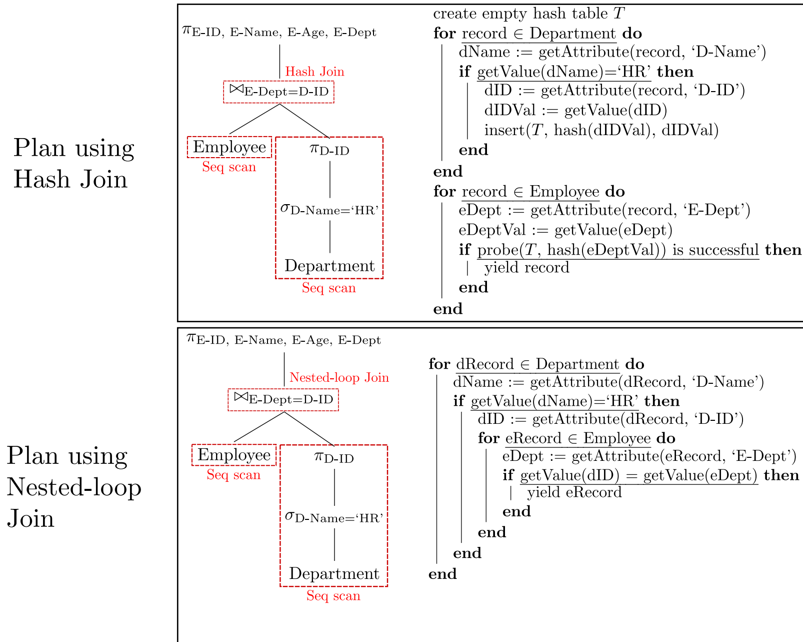


Figure 31. Example of two SQL physical plans derived from Figure 30

3 Thesis contribution

The undertaking of this framework is ambitious. The thesis primarily focuses on the DataPilot module, specifically its statistics and their application in query optimization. In subsequent chapters, the thesis presents formal models for JSON trees and the DataPilot module.

In summary, the thesis:

- Develops an algebra for JSON trees and collections, comprising operators dis-

tributed across three levels. (1) Operators that manipulate individual JSON trees, (2) operators that manipulate collections of JSON trees, and (3) operators derived from level (2) that can be efficiently executed at the code level.

- Introduces a method to generate indices over heterogeneous data by extending the idea of XML DataGuides ([35]) to determine when attributes have different data types and the cardinality of each. It further demonstrates how to incorporate this information into the optimization process.

- Establishes a formal model for the DataPilot structure and the structural and statistical data it encompasses.

- Introduces an algebra for DataPilots, where each algebraic expression yields a DataPilot approximation. This approximation mirrors the DataPilot resulting from the collection produced following the execution of the corresponding collection algebraic expression.

- Explores properties in the DataPilot algebra for rule-based query optimization over collections of JSON trees.

- Investigates the use of DataPilot statistics and algebraic properties for cost-based query optimization over collections of JSON trees.

- Introduces a two-step approach, encompassing both rule-based and cost-based methods, to convert and enrich a logical query plan derived from a given algebraic expression into an "optimization hint". This hint serves as the foundation for query optimization within a JSON document store when a more suitable index is absent.

- Conducts an experimental investigation to confirm the viability of the proposed framework.

CHAPTER IV

DOCUMENTS

1 Notation and terminology

This chapter commences with an introduction to the pertinent terminology and notation employed for elucidating the concept of a data pilot.

Recalling nomenclature from E.F. Codd's relational model: Data finds its representation in the form of tables, also referred to as relations. The designations of columns within a table are denoted as attributes, which, when aggregated into a set, constitute what is termed a *relation schema*, encapsulating the structure of the relation. Rows within a table are commonly known as *tuples*, and their cardinality must align with that of the relation schema.

In contrast, within the domain of a JSON document database, JSON objects find their analogue in tuples, collections in tables, and columns correspond to sequences of JSON fields. A comprehensive definition of each of these concepts is subsequently presented at the end of this section.

Diverging from relations, documents within a collection are not bound by a predetermined schema. Consequently, a distinction emerges between a JSON document schema and a collection schema: The schema of a JSON document encompasses sets of field sequences, where each sequence signifies a parent-child relationship. On the other hand, the collection schema is an amalgamation of the schemas of all JSON documents housed within the collection. A conspicuous disparity surfaces in the nature

of these schemas. A relational schema remains fixed and ascertainable even before any data is introduced to a relation. In contrast, a collection schema is dynamic, capable of changing with each novel document incorporation.

Having (approximately) delineated the parallelism between key concepts within relational algebra and the foundational notions explored in this chapter, the subsequent portion furnishes formalisms that lay the groundwork for the theoretical model expounded upon in subsequent sections.

The symbol $\mathbb{N}^0$ denotes the set of natural numbers with zero. i.e., $\mathbb{N}^0 = \mathbb{Z}^+ \cup \{0\}$.

For a function $f : A \to B$ and a subset $C \subseteq A$, the notation $f \restriction_C$ denotes the function f restricted to the elements of C. When referring to partial functions, the notation $f(a) \downarrow$ indicates that f is defined at a and that $f(a)$ exists, while $f(a) \uparrow$ indicates that f is not defined at a.

An alphabet Σ is a non-empty set of symbols, which are called labels. A labeling function is a function whose range is an alphabet.

A rooted tree $T = (N, E)$ is a connected acyclic graph identified by its set of nodes N and leaves $Lf(T) \subseteq N$, its set of edges $E = N \times N$, and a distinguished node $r(T)$ called the root. As usual, the notation $N(T)$, $E(T)$ and $Lf(T)$ refers to the nodes, edges and leaves of T respectively, any two nodes are adjacent if they are connected by and edge, and any two edges are adjacent if they are connected by a node.

A path refers to a sequence p of adjacent nodes or edges in T (e.g. a, b, c, d, or $(a, b), (b, c), (c, d)$). p is simple if each edge appears only once and elementary if each node appears only once. Paths are always assumed to be simple or elementary unless stated otherwise, which implies there is a unique path from $r(T)$ to any other node. p is said to *lead to* a node x if p starts at $r(T)$ and ends at x. p is said to be complete, denoted $p \overset{c}{\backsim} T$, if it leads to a leaf.

A labeled tree (N, E, κ, ϕ) is a rooted tree with mappings $\kappa : E' \subseteq E \to \Sigma_E$ and $\phi : N' \subseteq N \to \Sigma_N$, where (N, E, κ) is an edge-labeled tree and (N, E, ϕ) a

node-labeled tree.

A type is fundamentally a classification or categorization of a variable. In mathematics, the term "type" often denotes the *domain of discourse* pertaining to a mathematical object or value. For instance, if a variable x is confined to integer values, then x belongs to the type of integers (i.e., $x \in \mathbb{Z}$). In this context, a type can be conceptualized as a set of values. However, the concept of type in this chapter alludes to *data types* in the programming domain, distinguishing it from purely mathematical objects.

Formally, a type system, as relevant to this thesis, is characterized by the 4-tuple $(\mathfrak{A}, \mathfrak{C}, V^{\text{all}}, \Xi)$, where $\mathfrak{A}$ and $\mathfrak{C}$ represent disjoint sets of labels denoted as types, V^{all} constitutes a set encompassing sets of values, and Ξ constitutes a bijection from $\mathfrak{A} \cup \mathfrak{C}$ to V^{all}. Specifically, set $\mathfrak{A}$ encompasses the names of data types regarded as atomic by a given system, while set $\mathfrak{C}$ includes names of data types considered complex by the same system. The comprehensive set of types within a datbase type system is denoted as $\mathfrak{T} = \mathfrak{A} \cup \mathfrak{C}$.

In a JSON Type System, each type in $\mathfrak{T}$ corresponds to a data type in a JSON or JSON-like specification. Every type in the specification is represented by a type in $\mathfrak{T}$, and the classification into $\mathfrak{A}$ and $\mathfrak{C}$ aligns with the distinction between atomic and complex types in the specification.

Definition 1.1 (Document)**.** Let Σ_E be an alphabet and $V^{\text{atoms}} = \bigcup_{t \in \mathfrak{A}} \Xi(t)$. A document is the triplet (N, E, M) such that $M = \{\kappa : E \to \Sigma_E, \phi : Lf(T) \to V^{\text{atoms}}, \tau : N \to \mathfrak{T}\}$ and (N, E, κ, τ) is a labeled tree. Functions κ, τ and ϕ are called key function, type function and value function respectively. □

Theorem 1.1. Let T be a document, z an arbitrary node other than $r(T)$, and $r(T), y_1, y_2, \ldots, y_n, z$ a path. If for all nodes $u, v \in \text{children}(r(T), T)$, $\kappa(r(T), u) \neq \kappa(r(T), v)$, then the sequence $\kappa(r(T), y_1) \cdot \kappa(y_1, y_2) \cdot \ldots \cdot \kappa(y_n, z)$ is unique.

Proof. Assume a document T where for all nodes $u, v \in \text{children}(r(T), T)$, $\kappa(r(T), u) \neq$

$\kappa(r(T), v)$. Suppose Theorem 1.1 is false, then there exists two distinct nodes z, w with sequences $r(T), y_1, y_2, \ldots, y_n, z$ and $r(T), x_1, x_2, \ldots, x_n, w$ respectively such that $\kappa(r(T), y_1) \cdot \ldots \cdot \kappa(y_n, z) = \kappa(r(T), x_1) \cdot \ldots \cdot \kappa(x_n, z)$. This implies that $\kappa(r(T), y_1) = \kappa(r(T), x_1)$, which is a contradiction. □

Definition 1.2 (JSON Array)**.** Let A be a document with key function κ and type function τ such that $r(A)$ has k children. A is a JSON array if $\tau(r(A)) = \text{arr} \in \mathfrak{C}$ and for all children x of $r(A)$, the edge $(r(A), x)$ is labeled with a number in the interval $[0, k) \in \mathbb{N}^0$ such that $\kappa(r(A), x) \neq \kappa(r(A), y)$ for all siblings y of x. □

Definition 1.3 (Empty JSON Array)**.** Let A be a document with type function τ. A is an empty JSON Array if $\tau(r(A)) = \text{arr}$ and $r(A)$ has no children. □

Definition 1.4 (JSON Array concatenation)**.** Let A_1 and A_2 be JSON arrays with key functions κ_1 and κ_2 respectively, and $\text{children}(D, x)$ denote the set of children of a node x in the underlying tree of a document D. The concatenation of A_1 and A_2 denoted $A_1 \cdot A_2$ is a document with nodes $N(A_1) \cup N(A_2)$, edges $E(A_1) \cup (E(A_2) - \{(r(A_2), x) \mid x \in \text{children}(r(A_2), A_2)\}) \cup \{(r(A_1), x) \mid x \in \text{children}(r(A_1), A_1)\}$ and key function κ, type function τ and value function ϕ defined as follows:

$$\tau(x) = \begin{cases} \tau_1(x) & \text{if } x \in N(A_1) \\ \tau_2(x) & \text{if } x \in N(A_2) \end{cases}, \qquad \phi(x) = \begin{cases} \phi_1(x) & \text{if } x \in Lf(A_1) \\ \phi_2(x) & \text{if } x \in Lf(A_2) \end{cases},$$

$$\kappa(x, y) = \begin{cases} \kappa_1(x, y) & \text{if } (x, y) \in E(A_1) \\ \kappa_2(x, y) + \max\{\kappa_1(r(A_1), z) \mid z \in \text{children}(r(A_1))\} + 1 & \text{otherwise} \end{cases}$$

□

Definition 1.5 (JSON Document)**.** A JSON document $D = (N, E, M)$ adheres to the following conditions:

1. All types in the range of the mapping τ correspond to the set $\mathfrak{T}$ of a JSON type system. The range of the mapping ϕ corresponds to $\bigcup_{t \in \mathfrak{A}} \Xi(t)$, with set $\mathfrak{A}$ in the same type system.

2. $\tau(r(D)) =$ obj, where "obj" refers to the equivalent of a JSON Object in any JSON or JSON-like specification.

3. For any pair of children u, v of a node x with $\tau(x) =$ obj, $\kappa(x, u) \neq \kappa(x, v)$. That is, $\bigcap_{z \in \text{children}(D,x)} \{\kappa(x, z)\} = \emptyset$.

4. If a node x has $\tau(x) =$ arr, then the subtree rooted at x is a JSON Array as per Definition 1.2.

5. For any leaf $x \in Lf(D)$, if $\tau(x) \notin \{\text{arr,obj}\}$ then $\phi(x){\downarrow}$ else $\phi(x){\uparrow}$.

□

Definition 1.6 (Empty JSON Document)**.** Let D be a document with type function τ. D is an empty document if $\tau(x) =$ obj and $r(D)$ has no children. □

A path expression of length n is a finite sequence of n labels from a finite alphabet Σ. The notation p_i is used to refer to the i-th label in a path expression p, and $|p|$ denotes the length of p (i.e., $|p| = n$). When p is referred to explicitly by its labels, the notation $p_1 \cdot p_2 \cdot \ldots \cdot p_n$ is used, where $\cdot$ denotes concatenation. An empty path expression with no labels is denoted by ϵ.

Let p and q be two path expressions. q is said to be a prefix of p if either q is the empty path ϵ, or q is a shorter sequence than p and each label in q matches the corresponding label in p. The relation is denoted as $q \preceq p$. $\prec$ is defined as usual.

For convenience, any node x of a JSON document $D = (N, E, \{\kappa, \phi, \tau\})$ is often referenced to by the path expression formed by concatenating the labels of edges in the path leading to x. By addressing nodes this way, the hierarchical structure of D can be described under a new context (distinguished by using serif font), in terms of path expressions and the prefix relation, as follows:

1. Each node $x \in N$ is assigned a unique *identifier* through the bijection id_D: Let z be an arbitrary node other than $r(D)$ and $r(D), u_1, u_2 \ldots, u_n, z$ a path, then

$id_D(r(D)) = \epsilon$ and $id_D(z) = \kappa(r(D), u_1) \cdot \kappa(u_1, u_2) \cdot \ldots \cdot \kappa(u_n, z)$; note that $id_D(z)$ is guaranteed to be unique by Theorem 1.1.

2. Let $N' = \{id_D(x) \mid x \in N\}$, $E' = \prec$, $\tau' : x \in N' \mapsto \tau \circ id_D^{-1}(x)$, $\phi' : x \in N' \mapsto \phi \circ id_D^{-1}(x)$, and $\mathsf{D} = (N', E', \{\phi', \tau'\})$ where id_D^{-1} is the inverse of id_D. Then, D is equivalent to D in this new context.

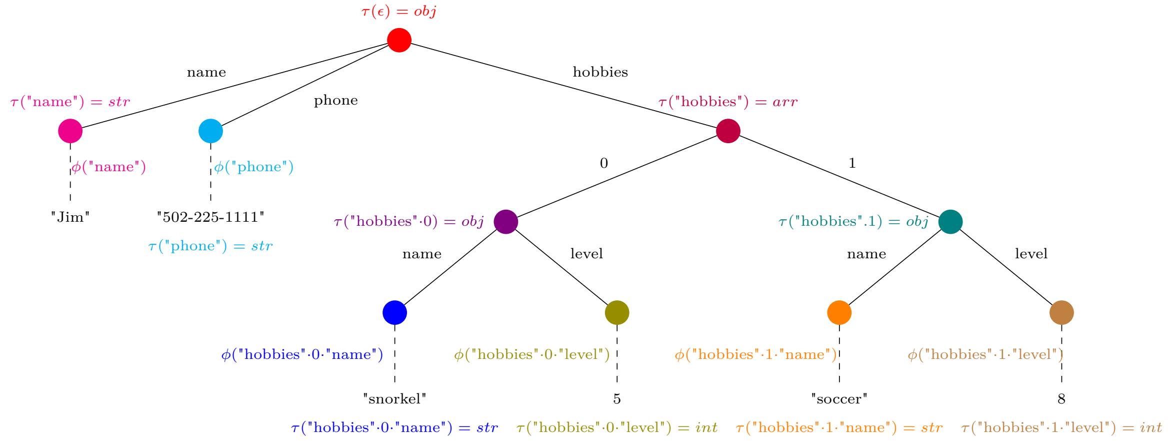
When explicitly denoting a JSON array as a set of paths, the notation $[\![p_1 : v_1, p_2 : v_2, \ldots, p_n : v_n]\!]$ is employed, where $v_1, v_2, \ldots, v_n$ represent the values that paths $p_1, p_2, \ldots, p_n$ evaluate to in the tree, respectively. It's required that the first labels of all paths are unique and fall within the interval $[0, n) \in \mathbb{N}^0$.

When explicitly denoting a JSON document as a set of paths, the notation $\{\!|p_1 : v_1, p_2 : v_2, \ldots, p_n : v_n|\!\}$ is used, with $v_1, v_2, \ldots, v_n$ being the corresponding values of paths $p_1, p_2, \ldots, p_n$ in the tree.

Notably, when a path p consists of more than one label, the presence of nested documents and arrays at each prefix of p (excluding p itself) is implicit. Alternatively, a path can be decomposed into its constituent labels to make the presence of these nested documents and arrays explicit. For example, consider a path $l_1 \cdot l_2 \cdot \ldots \cdot l_k$ evaluating to a value v. If each label in the path is not in $\mathbb{N}^0$, then the expressions $\{\!|l_1 \cdot l_2 \cdot \ldots \cdot l_k : v|\!\}$ and $\{\!|l_1 : \{\!|l_2 : \{\!|\ldots\{\!|l_k : v|\!\}\ldots|\!\}|\!\}|\!\}$ are equivalent. However, if there exists a label l_j in $\mathbb{N}^0$, then $\{\!|l_1 \cdot l_2 \cdot \ldots \cdot l_k : v|\!\}$ and $\{\!|l_1 : \{\!|l_2 : \{\!|\ldots\{\!|l_{j-1} : [\![l_j : \{\!|\ldots\{\!|l_k : v|\!\}\ldots|\!\}]\!]|\!\}\ldots|\!\}|\!\}|\!\}$ are equivalent.

Figures 32a and 32b show an example of a JSON document as a tree and as defined in the JSON specification respectively; types mapped to by τ and declarations of ϕ are colored according to the color of the respective node, dotted lines represent the value in V^{all} mapped to by ϕ.

Furthermore, the parents of a node x in D can be determined from x itself and the entire tree structure from its leaves, thus allowing for a recursive definition of the tree consisting of $r(\mathsf{D})$ and a regression rule explaining how, given a node, to derive



(a) JSON Document as a tree

```
{"name":"Jim",phone:"502-225-1111",
  "hobbies": [
    {"name":"snorkel","level":5},
    {"name":"soccer","level":"
        amateur"}
  ]
}
```

(b) JSON document as indicated in the JSON specification

$\{\!\![$ "name":"jim", "phone":"502-225-111",
"name"·"hobbies"·1·"name":"snorkel",
"name"·"hobbies"·1·"level":5,
"name"·"hobbies"·2·"name":"soccer"
"name"·"hobbies"·2·"level":"amateur" $]\!\!\}$

(c) JSON document using generic document notation

Figure 32. Example of a JSON document viewed as a tree and as indicated in the JSON specification

its parent:

$$\text{root: } x_0 = \epsilon$$

$$\text{rule: } (x)_n \xrightarrow{parent} x_0 \cdot \ldots \cdot x_{n-1}$$

Equipped with this recursive definition, any operation or map referring to nodes in D can be expressed in terms of $Lf(\mathsf{D})$[1]. Commonly used operations throughout the chapter are:

Definition 1.7 (Path suffix extraction)**.** Let T be a document and p a path expression. This operation treats p as a prefix of one or more paths in T to extract the suffix

[1]This recursive definition leverages the notion of a *generating tree*, employing a *regression rule* instead of the more customary succession rule found in generating tree definitions. The rule constitutes the inductive step, with the root serving as the foundation for induction.

q of all paths of the form $p.q$ in T. The restriction $q \neq \epsilon$ ensures the result equals $\emptyset$ instead of $\{\epsilon\}$ when $p \in Lf(\mathsf{T})$ and thus $suf(\mathsf{T}, p) \neq \emptyset$ ii and only if $p \in N(\mathsf{T}) - Lf(\mathsf{T})$. Note that when $p = \epsilon$, $suf(Lf(\mathsf{T}), p) = Lf(\mathsf{T})$.

$$suf(\mathsf{T}, p) = \{q \mid \exists\, q \neq \epsilon \wedge p.q \in Lf(\mathsf{T})\}$$

Definition 1.8 (Path projection)**.** Let T be a document and p a path expression. This operator filters out paths in $Lf(\mathsf{T})$ where p is not a prefix. In other words, it projects all branches of T where node p is present.

$$proj(\mathsf{T}, p) = \{p.q \mid \exists_q\, p.q \in Lf(\mathsf{T})\}$$

Definition 1.9 (Children extraction)**.** Let T be a document and p a path expression.

$$\text{children}(\mathsf{T}, p) = \{p.l \mid p.l \in N(\mathsf{T}) \wedge |l| = 1\}$$

Definition 1.10 (Subtree extraction)**.** Let T be a document and p a path expression. subtree$(\mathsf{T}, p) = \mathsf{T}'$ such that $Lf(\mathsf{T}') = suf(\mathsf{T}, p)$, $id^{-1}_{\mathsf{T}'} : q \in N(\mathsf{T}') \mapsto id^{-1}_{\mathsf{T}}(p.q)$ and for all $\lambda \in M(\mathsf{T})$ there exists a $\iota \in M(\mathsf{T}')$ such that $\iota : q \in N(\mathsf{T}') \mapsto \lambda(p.q)$

Definition 1.11 (Branch extraction)**.** Let T be a document and p a path expression. branch$(\mathsf{T}, p) = \mathsf{T}'$ such that $Lf(\mathsf{T}') = proj(\mathsf{T}, p)$, $id^{-1}_{\mathsf{T}'} = id^{-1}_{\mathsf{T}} \restriction_{N(\mathsf{T}')}$ and for all $\lambda \in M(\mathsf{T})$ there exists an $\iota \in M(\mathsf{T}')$ such that $\iota = \lambda\restriction_{N(\mathsf{T}')}$.

Definition 1.12 (Tree difference)**.** Let T_1 and T_2 be documents. The difference between T_1 and T_2, denoted $\mathsf{T}_1 - \mathsf{T}_2$, is the document T with $Lf(\mathsf{T}) = Lf(\mathsf{T}_1) - Lf(\mathsf{T}_2)$, $id^{-1}_{\mathsf{T}} = id^{-1}_{\mathsf{T}_1}\restriction_{N(\mathsf{T})}$ and for all $\lambda_1 \in M(\mathsf{T}_1)$ there exists a $\lambda \in M(\mathsf{T})$ such that $\lambda = \lambda_1\restriction_{N(\mathsf{T})}$.

Definition 1.13 (Subtree replacement)**.** Let T_1, T_2 be documents and p a path expression. replace$(\mathsf{T}_1, \mathsf{T}_2, p) = \mathsf{T}$ such that $Lf(\mathsf{T}) = (Lf(\mathsf{T}_1) - proj(\mathsf{T}_1, p)) \cup \{p.q \mid q \in \mathsf{T}_2\}$ and for all $\lambda_1 \in M(\mathsf{T}_1)$ and $\lambda_2 \in M(\mathsf{T}_2)$, there exists a $\lambda \in M(\mathsf{T})$ that maps paths as follows: for any path $x \in \mathsf{T}_2$, $\lambda(p.x) = \lambda_2(x)$, and for any path $y \in \mathsf{T}_1 - \text{branch}(\mathsf{T}_1, p)$, $\lambda(y) = \lambda_1(y)$.

Definition 1.14 (Union compatible documents)**.** Two documents T_1, T_2 are said to be union compatible if $id^{-1}_{\mathsf{T}_1}(\epsilon) = id^{-1}_{\mathsf{T}_2}(\epsilon)$, $id^{-1}_{\mathsf{T}_1}\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)} = id^{-1}_{\mathsf{T}_2}\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)}$, and there exists a bijection $\iota : M(\mathsf{T}_1) \to M(\mathsf{T}_2)$ such that for all $\lambda_1 \in M(\mathsf{T}_1)$, $\lambda_1\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)} = \iota(\lambda_1)\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)}$

Definition 1.15 (Document union)**.** Let T_1, T_2 be union compatible documents and $\iota : M(\mathsf{T}_1) \to M(\mathsf{T}_2)$ a bijection such that for all $\lambda_1 \in M(\mathsf{T}_1)$, $\lambda_1\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)} = \iota(\lambda_1)\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)}$. $\mathsf{T}_1 \cup \mathsf{T}_2 = \mathsf{T}$ such that $Lf(\mathsf{T}) = Lf(\mathsf{T}_1) \cup Lf(\mathsf{T}_2)$, for all λ_1 there exists a $\lambda \in M(\mathsf{T})$ such that $\lambda = \lambda_1 \cup \iota(\lambda_1)$, and

$$id^{-1}_{\mathsf{T}}(x) = \begin{cases} id^{-1}_{\mathsf{T}_2}(x) & \text{if } x \in N(\mathsf{T}_2) - (N(\mathsf{T}_1) \cap N(\mathsf{T}_2)) \\ id^{-1}_{\mathsf{T}_1}(x) & \text{otherwise} \end{cases}$$

Corollary 1.16. Let T_1, T_2 be union compatible documents. For any $\lambda \in M(\mathsf{T}_1 \cup \mathsf{T}_2)$, λ is always a function.

Proof. Let $\lambda_1 \in M(\mathsf{T}_1)$. Since T_1 and T_2 are union compatible, there exists a mapping $\lambda_2 \in M(\mathsf{T}_2)$ such that $\lambda_1\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)} = \lambda_2\restriction_{N(\mathsf{T}_1)\cap N(\mathsf{T}_2)}$. Thus, by the pasting lemma, $\lambda_1 \cup \lambda_2$ is a function. $\square$

Definition 1.17 (Path expression realization)**.** Let T be a document and p a path expression. p is realized in T, denoted $p \overset{r}{\backsim} \mathsf{T}$, if and only if $p \in N(\mathsf{T})$

Definition 1.18 (Path expression completeness)**.** Let T be a document and p a path expression. p is complete in T, denoted $p \overset{c}{\backsim} \mathsf{T}$, if and only if $p \in Lf(\mathsf{T})$.

Definition 1.19 (Value extraction of a JSON Document)**.** Let T be a JSON document, ϕ its value function, τ its type function, and p a path expression.

$$\mathsf{T} \hookrightarrow p = \begin{cases} \phi\,(p) & \text{if } p \in Lf(\mathsf{T}) \text{ and } \phi(p)\downarrow \\ subtree\,(\mathsf{T}, p) & \text{if } p \notin Lf(\mathsf{T}) \text{ and } \phi(p)\downarrow \\ \{\!\![\ ]\!\!\} & \text{if } p \in Lf(\mathsf{T}),\ \phi(p)\uparrow, \text{ and } \tau(p)\text{=obj} \\ [\![\ ]\!] & \text{if } p \in Lf(\mathsf{T}),\ \phi(p)\uparrow, \text{ and } \tau(p)\text{=arr} \end{cases}$$

Moreover, for any type t, $\mathsf{T} \overset{t}{\hookrightarrow} p = \mathsf{T} \hookrightarrow p$ if $\tau(p) = t$.

Example 1.1 (subtree)**.** Assume edge labeled trees A and B as depicted in Figure 33, such that $\mathsf{B} = \text{subtree}(\mathsf{A}, \text{"hobbies"})$. A has mapping $\lambda^{\mathsf{A}} : N(\mathsf{A}) \rightarrow \mathbb{N}^0$ while B has $\lambda^{\mathsf{B}} : N(\mathsf{B}) \subset N(\mathsf{A}) \rightarrow \mathbb{N}^0$. Paths colored light blue in $N(\mathsf{A})$ mark nodes making the subtree rooted at "hobbies", the symbol **X** indicates paths in $N(\mathsf{A})$ that no longer exist in $N(\mathsf{B})$, the diagonal lines drawn through labels within paths in $N(\mathsf{B})$ indicate labels no longer in the path—that is, ~~"hobbies"~~ becomes ϵ, ~~"hobbies"~~$\cdot$ 0 becomes 0 and so on; the end result being $N(\mathsf{B}) = \{\epsilon, 0, 0\cdot\text{"name"}, 0\cdot\text{"level"}\}$. Observe that:

- $\lambda^{\mathsf{B}}(\epsilon) = \lambda^{\mathsf{A}}(\text{"hobbies"}.\epsilon)$
- $\lambda^{\mathsf{B}}(0) = \lambda^{\mathsf{A}}(\text{"hobbies"}\cdot 0)$
- $\lambda^{\mathsf{B}}(0\cdot\text{"name"}) = \lambda^{\mathsf{A}}(\text{"hobbies"}\cdot 0\cdot\text{"name"})$
- $\lambda^{\mathsf{B}}(0\cdot\text{"level"}) = \lambda^{\mathsf{A}}(\text{"hobbies"}\cdot 0\cdot\text{"level"})$

which complies with the definition of *subtree* as $\lambda^{\mathsf{B}}(q) = \lambda^{\mathsf{A}}(\text{"hobbies"} \cdot q) \mid \forall q \in N(\mathsf{B})$. $\square$

Definition 1.20 (Collection)**.** A collection $\mathcal{C} = (\mathfrak{K}, \mathit{mult}^{\mathfrak{K}})$ is a multiset of JSON documents where $\mathfrak{K}$ is the underlying set of JSON documents and function $\mathit{mult}^{\mathfrak{K}} : \mathfrak{K} \rightarrow \mathbb{Z}^+$ gives the multiplicity (i.e. number of occurrences) of any document $\mathsf{D} \in \mathfrak{K}$ as the number $\mathit{mult}^{\mathfrak{K}}(\mathsf{D})$ such that $\mathit{mult}^{\mathfrak{K}}(\mathsf{D}) > 0$. The cardinality of $\mathcal{C}$ is expressed as $|\mathcal{C}| = \sum \mathit{mult}^{\mathfrak{K}}(\mathsf{D}) \mid \mathsf{D} \in \mathfrak{K}$ and its support as $\text{supp}(\mathcal{C}) = \mathfrak{K}$.

Definition 1.21 (Collection Cover)**.** Given a collection $\mathcal{C}$, the cover of $\mathcal{C}$ is the set of paths present in any document of $\mathcal{C}$. Formally,

$$\text{cover}(\mathcal{C}) = \bigcup_{\mathsf{d} \in \mathcal{C}} N(\mathsf{d})$$

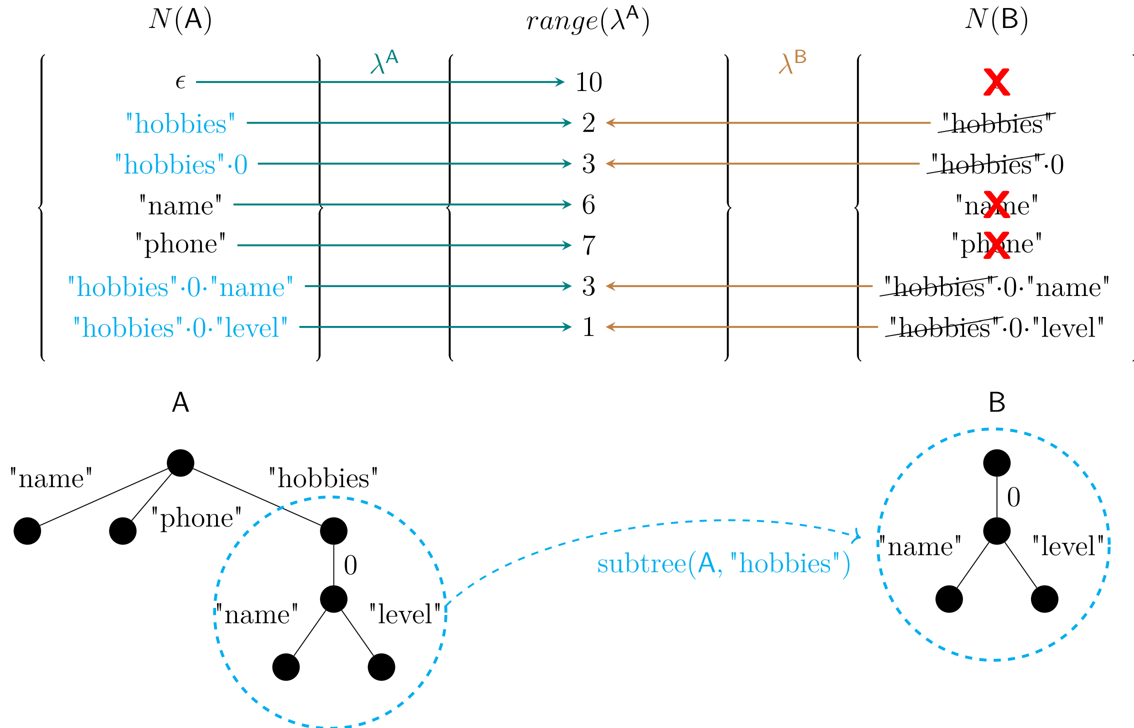


Figure 33. Example of the *subtree* function applied to a tree A with node labeling function $\lambda^{\mathsf{A}} : \mathsf{A}(N) \to \mathbb{N}^0$. The new tree and node labeling function are B and λ^{B} respectively.

2 JSON Document Algebra

Given the inherent capacity of JSON documents to facilitate nested structures, it becomes apparent that the ambit of operations performed on collections cannot be entirely encapsulated by conventional relational algebra. The prospect of nested relational algebra, though considered, is inadequate in addressing both the multifaceted diversity inherent in JSON arrays and the dynamic nature of collection schemas. There are multiple JSON formalisms in the literature, each offering a distinct avenue for articulating JSON queries. Within the scope of this manuscript, a specific formalism is adopted, one that takes the fundamental tenets of relational algebra and extends them, while mirroring the behavior of some operators from [14].

Selection

Represented as $\sigma_{\varphi}(\mathcal{C})$, a selection filters out documents in $\mathcal{C}$ for which the expression φ is false.

An expression φ follows the semantics below, where "Expr" denotes an expression itself, "Const" a constant (or literal), "PathExpr" a path expression as defined in Section IV.1 and θ a comparison operator that may be defined for multiple types, but always assumes operands of the same type.

$\texttt{Expr := SimpleExpr | (Expr) | Expr} \wedge \texttt{Expr | Expr} \vee \texttt{Expr}$

$\texttt{SimpleExpr := PathExpr } \theta \texttt{ PathExpr | PathExpr } \theta \texttt{ Const | } \exists\texttt{PathExpr}$

$\qquad\qquad \texttt{| isType(PathExpr, } t \in \mathfrak{T}\texttt{)}$

---

expressions are of types: Path-Path (PathExpr θ PathExpr), Path-Const (PathExpr θ Const), existence ($\exists$PathExpr), data type inquiry (isType(PathExpr, $t \in \mathfrak{T}$)), conjunctive (Expr $\wedge$ Expr), disjunctive (Expr $\vee$ Expr) and prioritized ((Expr)). The first four are of the simple expression kind and the last three of the complex expression kind.

Throughout the rest of the thesis, selection expressions are referred to as *selection predicates*, or just *predicates* when context allows, as a way to keep the concept familiar to relational database theory.

Projection

project is a unary operator written as $\pi_P(\mathcal{C})$, where P is a set of path expressions. Let d be an arbitrary JSON document in $\mathcal{C}$; the operator filters out paths in $Lf(\mathsf{d})$ that do not have as prefix at least one path in P. Any path having a prefix in P is said to be *projection idempotent* with respect to P, which is shown in more formal terms in Definition 2.1.

Definition 2.1 (Projection idempotent path)**.** For any path $q \in Lf(\mathsf{d})$, q is *projection idempotent* with respect to P if there exists a path $p \in P$ such that $p \preceq q$.

Definition 2.2 (Projection idempotent set)**.** All projection idempotent paths in $Lf(\mathsf{d})$ with respect to P form a projection idempotent set with respect to d and P. Elements of such set become the new leaves of d after a projection over $\mathcal{C}$.

Lemma 2.1 (Projection idempotence)**.** $\pi_P(\pi_P(\mathcal{C})) = \pi_P(\mathcal{C})$

Proof. Let d be an arbitrary document in collection $\mathcal{C}$ and S be a projection idempotent set with respect to d and P. Suppose d becomes d' after $\pi_P(\mathcal{C})$, which means $Lf(\mathsf{d}') = S$. Then, by definition, all paths in S are projection idempotent with respect to P, thus a second projection over $\pi_P(\mathcal{C})$ leaves d' unchanged as $Lf(\mathsf{d}')$ is already projection idempotent with respect to P. □

For any path s resolving to an array, projecting specific array elements of s in the data by using positional indexes (e.g. $s \cdot 0$) filters out any element whose index is not specified. Projecting fields of objects within arrays (e.g. $s \cdot 0 \cdot$"c") has a similar effect, but removes array instances of s where the specified object field does not exist. An example of both cases is shown in Examples 2.1 and 2.2.

Example 2.1 (Projection of array element in the data)**.** Consider a collection $\mathcal{K}$ with the following documents in JSON format:

```
1 {"a":1,"b":[{"c":1,"d":2},{"e":3,"f":4}]}
2 {"a":2,"b":[{"e":5,"d":6},{"c":7,"g":8}]}
3 {"a":3,"b":[5]}
4 {"b":{"0":{"c":1,"d":2},"1":{"e":3,"f":4}}}
```

The projection $\pi_{\text{"b"}\cdot 0}(\mathcal{K})$ removes all paths but "b" from all documents as long as "b" resolves to an array. Only the first element of "b" is kept. This is depicted in Figure 34, where the structure of each document is shown and tree branches in red are the only ones remaining after project.
The resulting collection after the projection is:

Figure 34. Projection $\pi_{\text{"b"}\cdot 0}(\mathcal{K})$ over collection $\mathcal{K}$ with documents d1, d2, d3, and d4

```
1 {"b":[{"c":1,"d":2}]}
2 {"b":[{"e":5,"d":6}]}
3 {"b":[5]}
4 {}
```

Notice that "b" resolves to an object in the document at line 4 and is thus removed. The key distinction between the path to project ("b"·0) and path "b"·"0" from the document at line 4 is that the symbols 0 and "0" are different as the former belongs to $\mathbb{N}^0$, while the latter does not. □

Example 2.2 (Projection of object field within array in the data)**.** Consider the same collection $\mathcal{K}$ from Example 2.1. The projection $\pi_{\text{"b"}\cdot 0\cdot\text{"c"}}(\mathcal{K})$ removes all paths but "b" from all documents as long as "b" resolves to an array and the first position of such array contains an object with field "c". This is depicted in Figure 34, where the structure of each document is shown and tree branches in red are the only ones remaining after project.

Figure 35. Projection $\pi_{\text{"b"}\cdot 0\cdot\text{"c"}}(\mathcal{K})$ over collection $\mathcal{K}$ with documents d1, d2, d3, and d4

Unlike Example 2.1, all documents except d1 are empty in the resulting collection:

```
1 {"b":[{"c":1}]}
```

```
2 {}
3 {}
4 {}
```

□

A variation of project called a non-preserving project is shown in Equation 1. Their behavior is similar, but the latter filters out empty documents. If applied to the resulting collection in Example 2.1, document d4 is removed, while documents d2 to d4 in Example 2.2 are removed.

$$\pi_P^{\oslash}(\mathcal{C}) = \sigma_{\exists p_1 \vee \cdots \vee \exists p_k}(\pi_P(\mathcal{C})) \tag{1}$$

Cartesian Product

Let $\mathcal{C}_1$,$\mathcal{C}_2$ be collections and l be the empty path or a label not realized in neither collection. The cartesian product of $\mathcal{C}_1$ and $\mathcal{C}_2$, denoted as $\mathcal{C}_1 \times^l \mathcal{C}_2$, is the merging of all pairs of documents $\mathsf{c}_1, \mathsf{c}_2$ such that $\mathsf{c}_1 \in \mathcal{C}_1$, $\mathsf{c}_2 \in \mathcal{C}_2$ and all keys at the first level of both documents are different. Merging occurs in one of two ways:

- if $l = \epsilon$, a new document is created with copies of all first level fields (and their values) from c_1 and c_2.
- else a new document is created with copies of all first level fields (and their values) from c_1 and the extra field l added such that the value of l is c_2.

A formal definition is given in Definition 2.3.

Definition 2.3. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and l a label such that either $l = \epsilon$ or $l \notin \text{cover}(\mathcal{C}_1)$.

$$\mathcal{C}_1 \times^l \mathcal{C}_2 = \begin{cases} \langle \mathsf{c}_1 \cup \mathsf{c}_2 \mid \mathsf{c}_1 \in \mathcal{C}_1 \wedge \mathsf{c}_2 \in \mathcal{C}_2 \rangle & \text{if } l = \epsilon \\ \langle \mathsf{c}_1 \cup \{\!\{ l : \mathsf{c}_2 \}\!\} \mid \mathsf{c}_1 \in \mathcal{C}_1 \wedge \mathsf{c}_2 \in \mathcal{C}_2 \rangle & \text{otherwise} \end{cases}$$

□

When $l = \epsilon$, the cartesian product of $\mathcal{C}_1$ and $\mathcal{C}_2$ is written as $\mathcal{C}_1 \times \mathcal{C}_2$ instead of $\mathcal{C}_1 \times^{\epsilon} \mathcal{C}_2$ for simplicity.

Unnest

Given a path expression p, unnesting p can be understood as unwinding the array p. This process comprises two steps: filtering—all documents where p does not exist, resolves to a non-array type or an empty array are removed—and deconstruction—on all documents where p resolves to an array a, a is deconstructed such that for each element $x \in a$, unnest outputs a new document where p resolves to x.

Let d be an arbitrary document in some collection $\mathcal{C}$. unnest has 3 invariants d must comply with to be deconstructed:

1. p must be realized in d.
2. p must point to an array.
3. the array p points to must not be empty.

There are two distinct unnest operators: preserving (Definition 2.5) and non-preserving (Definition 2.4). Unnest is preserving when only deconstruction is applied on $\mathcal{C}$ and non-preserving when filtering and deconstruction are (both) applied consecutively on $\mathcal{C}$.

Definition 2.4 (Non-preserving Unnest)**.** Let $\mathcal{C}$ be a collection and p a path. The non-preserving unnest of documents in $\mathcal{C}$ regarding to p is defined as follows:

$$\mu_p^{\oslash}(\mathcal{C}) = \langle \text{replace}(\mathsf{d}, \text{subtree}(\mathsf{d}, p \cdot x)\,, p) \mid \forall d \in \mathcal{C}\, \forall x \in \text{children}(\mathsf{d}, p) \,\wedge \tau(\mathsf{d}, p) = \text{arr} \rangle$$

□

Definition 2.5 (Preserving Unnest)**.** Let $\mathcal{C}$ be a collection and p a path. The preserving unnest of documents in $\mathcal{C}$ regarding p is defined as follows:

$$\mu_p(\mathcal{C}) = \mu_p^{\varnothing}(\mathcal{C}) \cup \langle \mathsf{d} \in \mathcal{C} \mid \tau(\mathsf{d}, p) \neq \text{arr} \rangle$$

□

Aggregate functions

Within the manuscript, aggregate functions, akin to those found in relational databases, are mechanisms that process multiple values concurrently to yield a single result. In the context of documents, these functions operate on three inputs: a label, a path, and a collection. Definition 2.6 provides a generalized abstraction of aggregate functions.

Definition 2.6 (Aggregate function)**.** Let $\mathcal{C}$ represent a collection, p a path, $(\odot, S)$ denote a monoid derived from elements in a generic set S with an operation $\odot$, and δ be a mapping assigning a value within the domain of $\odot$ to each document based on p. The aggregate function f, which reduces all values p refers to in documents from $\mathcal{C}$, is defined as:

$$f(\mathcal{C}, p) = \bigodot_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, p)$$

□

In Definition 2.6, the usage of a monoid implies the existence of an identity element. This identity is employed in the definition of specific aggregate functions to substitute any non-existent or incompatible values extracted from a JSON document. For instance, the well-known sum aggregate function, restricted to numbers, utilizes 0 as the identity. When encountering a non-numerical element, it simply replaces it with this identity (e.g., 1 + "a" is treated as 1 + 0). The extraction of these numbers from each document and the appropriate use of the monoid's identity element are delegated to the δ mapping for each particular aggregate function.

Aggregate functions are categorized in two types based on how their result is affected by the presence of duplicates. If the result of the aggregate function agg

does not change in the presence of duplicates (e.g. min and max), agg is called *duplicate-agnostic*, else, if the result does change, agg is called *duplicate-sensitive* ([27]).

Examples of specific aggregate functions based on Definition 2.6 include:

$$\operatorname{sum}(\mathcal{C}, p) = \sum_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, p) \text{ with } \delta(\mathsf{d}, p) = \begin{cases} \mathsf{d} \hookrightarrow p & \text{If } \mathsf{d} \xhookrightarrow{\text{num}} p \downarrow \\ 0 & \text{otherwise} \end{cases}$$

$$\operatorname{count}(\mathcal{C}, p) = \sum_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, p) \text{ with } \delta(\mathsf{d}, p) = \begin{cases} 1 & \text{If } p \overset{r}{\backsim} \mathsf{d} \\ 0 & \text{otherwise} \end{cases}$$

$$\operatorname{avg}(\mathcal{C}, p) = \frac{\operatorname{sum}(\mathcal{C}, p)}{\operatorname{count}(\mathcal{C}, p)}$$

It is noteworthy that there exist operators over numbers, like min and max, that do not constitute a monoid on the set of real numbers $\mathbb{R}$. However, this is not problematic as database systems handle subsets of $\mathbb{R}$ that are bounded below and above, primarily due to hardware limitations. For example, 64-bit integers have lower and upper bounds of -2^{63} and $2^{63} - 1$, respectively.

In traditional SQL-based systems, non-aggregate functions can operate within aggregate functions, thereby transforming the input to the aggregate function on a per-document basis. To support such behavior, Definition 2.6 can be extended as follows:

Definition 2.7 (Higher order aggregate function)**.** Let $\mathcal{C}$ represent a collection, p a path, $(\odot, S)$ denote a monoid derived from elements in a generic set S with an operation $\odot$, δ be a mapping assigning a value within the domain of $\odot$ to each document based on p, and g a non-aggregate function whose domain matches that of $\odot$. The higher-order aggregate function f is defined as:

$$f(\mathcal{C}, p, g) = \bigodot_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, g, p)$$

□

In Definition 2.7, the non-aggregate function g is responsible for transforming the individual values extracted from a document within the specific δ mapping of a particular aggregate function. For instance, defining the *sum* aggregate function as a higher-order aggregate function would look as follows:

$$\text{sum}\,(\mathcal{C}, p, g) = \sum_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, p, g) \text{ with } \delta(\mathsf{d}, p, g) = \begin{cases} g\,(\mathsf{d} \hookrightarrow p) & \text{If } \mathsf{d} \xhookrightarrow{\text{num}} p \downarrow \\ 0 & \text{otherwise} \end{cases}$$

This allows for representations of operations akin to SQL's $\text{sum}\,(2 * p)$ by defining the g function as $g : x \mapsto 2 * x$.

Compared to aggregate functions in a purely relational context, Definitions 2.6 and 2.7 are sufficiently general to allow aggregate functions in this context to generate not only scalars but also other types of mathematical objects. While these may not have an explicitly defined data type in a JSON-like standard, their utilization can aid in replicating the behavior of SQL's *json_arrayagg* or MongoDB's *$push* aggregate functions, which entail introducing order to an unordered collection of values.

Consider the aggregate function msetagg, which simply utilizes the multiset union operator and the empty multiset as the identity element to form a single multiset comprising all values a path p resolves to in documents from a collection $\mathcal{C}$:

$$\text{msetagg}\,(\mathcal{C}, p, g) = \bigcup_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, p) \text{ with } \delta(\mathsf{d}, p, g) = \begin{cases} \{g\,(\mathsf{d} \hookrightarrow p)\} & \text{If } p \overset{r}{\frown} \mathsf{d} \\ \emptyset & \text{otherwise} \end{cases}$$

If there exists a bijection $h : \text{msetagg}\,(\mathcal{C}, p) \rightarrow [0, k) \subset \mathbb{N}^0$ that can be declared on-demand in the database system whenever the *msetagg* aggregate function is used, with $k = |\,\text{msetagg}\,(\mathcal{C}, p, g)|$, then the values in $\text{msetagg}\,(\mathcal{C}, p, g)$ and h can be employed to construct an array. This can be achieved by pairing values from $\text{msetagg}\,(\mathcal{C}, p, g)$ into a tree and utilizing h to label its edges. For instance:

$$\text{arragg}\,(\mathcal{C}, p, g) = \bigcup_{x \in \text{msetagg}(\mathcal{C}, p, g)} [\![h(x) : x]\!]$$

Aggregate functions can be decomposed like they are in relational databases ([78],[27]). In this context, the property of *decomposability* is described in Definition 2.8.

Definition 2.8 (Decomposability)**.** Let $\mathcal{C}$, $\mathcal{C}_1$ and $\mathcal{C}_2$ be a collections such that $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$, p a path and g a non-aggregate function. An aggregate function agg is decomposable if there exist two aggregate functions agg_1 and agg_2 such that $\text{agg}(\mathcal{C}, p, g) = \text{agg}_2(\langle \{\!\{p : \text{agg}_1(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \text{agg}_1(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g)$.

Amongst aggregate functions, some of the most notorious decompositions listed in [27] are adapted to JSON documents as follows:

$$\begin{aligned}
&\min(\mathcal{C}_2 \cup \mathcal{C}_2, p, g) = \min(\langle \{\!\{p : \min(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \min(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g) \\
&\max(\mathcal{C}_2 \cup \mathcal{C}_2, p, g) = \max(\langle \{\!\{p : \max(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \max(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g) \\
&\text{count}(\mathcal{C}_2 \cup \mathcal{C}_2, p, g) = \text{sum}(\langle \{\!\{p : \text{count}(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \text{count}(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g) \\
&\text{sum}(\mathcal{C}_2 \cup \mathcal{C}_2, p, g) = \text{sum}(\langle \{\!\{p : \text{sum}(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \text{sum}(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g) \\
&\text{avg}(\mathcal{C}_2 \cup \mathcal{C}_2, p, g) = \frac{\text{sum}(\langle \{\!\{p : \text{sum}(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \text{sum}(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g)}{\text{sum}(\langle \{\!\{p : \text{count}(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \text{count}(\mathcal{C}_2, p, g)\}\!\}\rangle, p, g)}
\end{aligned}$$

Note that the *arragg* aggregate function introduced earlier is not easily decomposable as it would create an array per collection and then nest these arrays inside another array. To solve this, a new aggregate function that concatenates arrays using the operator in Definition 1.4can be defined as follows:

$$\text{arrconcat}(\mathcal{C}, p) = \mathop{\cdot}_{\mathsf{d} \in \mathcal{C}} \delta(\mathsf{d}, p) \text{ with } \delta(\mathsf{d}, p) = \begin{cases} \mathsf{d} \hookrightarrow p & \text{If } \mathsf{d} \xhookrightarrow{\text{arr}} p \downarrow \\ [\![]\!] & \text{otherwise} \end{cases}$$

Note that commutativity is sacrificed with arrconcat, but associativity is maintained and thus the monoidal property is preserved. The decomposition would then look as follows:

$$\text{arragg}(\mathcal{C}_2 \cup \mathcal{C}_2, p, g) = \text{arrconcat}(\langle \{\!\{p : \text{arragg}(\mathcal{C}_1, p, g)\}\!\}, \{\!\{p : \text{arragg}(\mathcal{C}_2, p, g)\}\!\}\rangle, p)$$

Group

A group operation consists of five elements: (1) a collection $\mathcal{C}$ to partition, (2) a set P of j path expressions used to partition $\mathcal{C}$ so that documents sharing the same values for all $p \in P$ are placed in the same partition, (3) a set Q of k paths to combine, (4) a proper class F of k aggregate functions used to combine values from paths in Q, and (5) a set R of k labels. Elements (1) and (2) constitute the *partitioning step*, and elements (3), (4), and (5) form the *merging step*.

Let $P = \{p_1, p_2, \ldots, p_j\}$ (where indexes 1 through j are used only to differentiate elements and have no inherent order). During the partitioning step, all documents in $\mathcal{C}$ with the same values for path expressions $p_1, p_2, \ldots, p_j$ are placed in the same partition. For any document, the values of these path expressions can be of some type in $\mathfrak{T}$ or *null*. For any two documents, if a path expression $p \in P$ is not realized, then p is considered *undefined* for both and assumed to resolve to the same value, although one that is different from any type in $\mathfrak{T}$ or *null*.

Let $Q = \{q_1, q_2, \ldots, q_k\}$, $R = \{r_1, r_2, \ldots, r_k\}$ and $f_1, f_2, \ldots f_k$ be elements of F. During the merging step, documents in the same partition are combined into a single document as follows: f_i combines all values of q_i based on some database-defined behavior. The new document has all realized path expressions in P, each one evaluating to the same value they evaluate to in all documents in the partition, and path r_i resolves to the result of the combination f_i produces.

Group has the form shown in Formula 2, where each term $r_i : f_i(q_i)$ on the left-hand side of the equality is called an *aggregate*.

$$
\begin{aligned}
&{}_{p_1,\ldots,p_j}\gamma_{r_1:f_1(q_1),r_2:f_2(q_2),\ldots,r_k:f_k(q_k)}(\mathcal{C}) = \\
&\quad \Big\langle \{\!\!\{ p_1 : v_1, \ldots, p_j : v_j \}\!\!\} \cup \bigcup_{1 \le i \le k} \{\!\!\{ r_i : f_i \left(\sigma_{p_1=v_1 \wedge \cdots \wedge p_j=v_j} \left(\mathcal{C} \right), q_i \right) \}\!\!\} \\
&\qquad \Big| \, \{\!\!\{ p_1 : v_1, \ldots, p_j : v_j \}\!\!\} \in \operatorname{supp} \left(\pi_{p_1,\ldots,p_j} \left(\mathcal{C} \right) \right) \Big\rangle
\end{aligned}
\tag{2}
$$

Group supports the fundamental scalar aggregate functions in SQL, including avg, count, max, min, and sum. Alternative query languages such as SQL++ and

MongoDB's aggregation pipeline introduce compatibility for various other aggregate functions that can also be supported, but are not covered. Standardization of array aggregate functions is less prevalent; however, the prevalent one, albeit named differently in the aforementioned query languages, is arragg. This aggregate function behaves analogously to SQL's json_arrayagg, MongoDB's $push, and is akin to a non-scalar subquery integrated within the select clause in SQL++.

In various query languages, multiple aggregates can often be employed. To simplify the explanation of certain concepts and relationships in subsequent sections, the adopted approach follows the use of aggregate vectors as outlined in [27]. An aggregate vector is represented as $\tilde{F} = (\!(r_1 : f_1(q_1), r_2 : f_2(q_2), \ldots, r_k : f_k(q_k))\!)$. The set of paths that are referenced within $\tilde{F}$ is denoted as $paths(\tilde{F})$. The combination of two distinct aggregate vectors, $\tilde{F}_1$ and $\tilde{F}_2$, is indicated by $\tilde{F}_1 \cdot \tilde{F}_2$. Formula 2 can then be reformulated in the following manner:

$$_P\gamma_{\tilde{F}}(\mathcal{C}) \text{ with } \tilde{F} = (\!(r_1 : f_1(q_1), r_2 : f_2(q_2), \ldots, r_k : f_k(q_k))\!) \tag{3}$$

Aggregate vectors support the splittability property outlined in [27]:

Definition 2.9 (Aggregate vector splittability)**.** Let $\mathcal{A}$,$\mathcal{B}$ be collections and $\tilde{F}$ a vector of aggregates. $\tilde{F}$ can be split into vectors $\tilde{F}_\mathcal{A}$ and $\tilde{F}_\mathcal{B}$ with respect to $\mathcal{A}$ and $\mathcal{B}$ if $\tilde{F} = \tilde{F}_\mathcal{A} \cdot \tilde{F}_\mathcal{B}$, $paths(\tilde{F}_\mathcal{A}) \cap \text{cover}\,(\mathcal{B}) = \emptyset$, and $paths(\tilde{F}_\mathcal{B}) \cap \text{cover}\,(\mathcal{A}) = \emptyset$.

Aggregate vector splittability together with the decomposability of aggregate functions are particularly useful properties when rewitting queries as they allow to break down aggregations and perform groupings earlier in a query when doing so benefits performance.

Example 2.3. Assume collections $\mathcal{O}$ and $\mathcal{L}$ analogous to tables Orders and Lineitem in the TPC-H Benchmark and the query: *Find the total loss of revenue on orders handled by each clerk due to parts being returned by customers.* This query looks as

follows in the algebra:

$$_{\text{o_clerk}}\gamma_{\text{loss:sum(l_extendedprice} * (1-\text{l_discount}))}(\sigma_{\text{l_returnflag='R'}}(\mathcal{O} \bowtie_{\text{o_orderkey=l_orderkey}} \mathcal{L}))$$

In the TPC-H schema, each order is handled by one clerk, thus the loss of revenue from each particular order can be eagerly computed and all losses subsequently aggregated to compute the total loss. This involves breaking the group operator into two as follows:

$$
\begin{aligned}
&_{\text{o_clerk}}\gamma_{\text{loss:sum(order_loss)}}(\\
&\quad \mathcal{O} \\
&\quad \bowtie_{\text{o_orderkey=l_orderkey}} \\
&\quad _{\text{l_orderkey}}\gamma_{\text{order_loss:sum(l_extendedprice} * (1-\text{l_discount}))}(\sigma_{\text{l_returnflag='R'}}(\mathcal{L})) \\
&)
\end{aligned}
$$

□

When rewriting queries, decomposition of duplicate-sensitive aggregate functions may require amendments to take into consideration any values grouped earlier in the decomposed aggregates.

Example 2.4. Assume collections $\mathcal{O}$ and $\mathcal{L}$ analogous to tables Orders and Lineitem in the TPC-H Benchmark and the query: *Find the total number of lineitems returned per customer.* The query can be written in the algebra as follows:

$$_{\text{o_custkey}}\gamma_{\text{returned:count}(*)}(\sigma_{\text{l_returnflag='R'}}(\mathcal{O} \bowtie_{\text{o_orderkey=l_orderkey}} \mathcal{L}))$$

In the TPC-H schema, an order can have multiple lineitems, thus the number of returned lineitems per order can be computer early on. Then, the grouping performed after the join would simply add up these numbers for orders that belong to the same

customer.

$$
\begin{aligned}
&{}_{\text{o_custkey}}\gamma_{\text{returned:sum(order_returned)}}(\\
&\quad \mathcal{O}\\
&\quad \bowtie_{\text{o_orderkey=l_orderkey}}\\
&\quad {}_{\text{l_orderkey}}\gamma_{\text{order_returned:count}(*)}(\sigma_{\text{l_returnflag='R'}}(\mathcal{L}))\\
&)
\end{aligned}
$$

Note that unlike with Example 2.3, here the aggregate used (i.e. count) must be *amended* in the outer group to keep the query coherent. □

In more general terms, this transformation, or amendment, is formalized as follows:

Definition 2.10 (Aggregation amendment)**.** Let agg be an aggregate, q a path and c a label. Amending agg by c is denoted $\text{agg} \otimes c$ and defined as follows:

$$
(\text{agg} \otimes c)(q) = \begin{cases} \text{agg}(q) & \text{if agg is duplicate-agnostic} \\ \text{agg}(q * c) & \text{if agg is sum} \\ \text{sum}\,(c) & \text{if agg}(q) = \text{count}\,(*) \end{cases}
$$

Definition 2.11 (Aggregate vector amendment)**.** Let $\tilde{F} = (\!(l_1 : \text{agg}_1(q_1), \ldots, l_k : \text{agg}_k(q_k))\!)$ be an aggregate vector and c a label. Amending $\tilde{F}$ by c, denoted $\tilde{F} \otimes c$, is defined as follows:

$$
\tilde{F} \otimes c = (\!(l_1 : (\text{agg}_1 \otimes c)(q_1), \ldots, l_k : (\text{agg}_k \otimes c)(q_k))\!)
$$

Inner Join

Let $\mathcal{A}$,$\mathcal{B}$ be collections, α a predicate and l the empty path or a label not realized in neither collection. The inner join of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \bowtie^l_\alpha \mathcal{B}$, is the merging of all pairs of documents a, b such that $\mathsf{a} \in \mathcal{A}$, $\mathsf{b} \in \mathcal{B}$, and all keys at the first level of both documents are different; if a and b satisfy condition α. Merging behaves as in the cartesian product.

Note that similarly to the relational case, the join of $\mathcal{A}$ and $\mathcal{B}$ when $l = \epsilon$ can be considered as a derived operator equivalent to a cartesian product followed by a selection: $\mathcal{A} \bowtie_\alpha \mathcal{B} = \sigma_\alpha(\mathcal{A} \times \mathcal{B})$. When $l \neq \epsilon$, α must be rewritten into some α' such that all paths from $\mathcal{B}$ involved in α are prepended l. Formally, $\mathcal{A} \bowtie_\alpha^l \mathcal{B} = \sigma_{\alpha'}(\mathcal{A} \times^l \mathcal{B})$ if $\{l \cdot p | p \in \text{exprpaths}(\alpha) \land p \overset{r}{\backsim} dp^{\mathcal{B}}\} \subseteq \text{exprpaths}(\alpha')$.

Other relational join variants are supported as well:

Definition 2.12 (Left Semijoin)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and α a predicate.

$$\mathcal{C}_1 \ltimes_\alpha \mathcal{C}_2 = \langle \mathsf{d} \in \mathcal{C}_1 \mid \langle \mathsf{d} \rangle \bowtie_\alpha \mathcal{C}_2 \neq \emptyset \rangle$$

□

Corollary 2.13 (Projection of Inner Join as Left Semijoin)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and α a predicate.

$$\mathcal{C}_1 \ltimes_\alpha \mathcal{C}_2 = \pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2)$$

□

Definition 2.14 (Left Antijoin)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and α a predicate.

$$\mathcal{C}_1 \overline{\ltimes}_\alpha \mathcal{C}_2 = \mathcal{C}_1 - (\mathcal{C}_1 \ltimes \mathcal{C}_2)$$

□

Definition 2.15 (Left Outer Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and α a predicate.

$$\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2 = (\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) \cup (\mathcal{C}_1 \overline{\ltimes}_\alpha \mathcal{C}_2)$$

□

Generalized Outer Join (GOJ)

The definition of the generalized outer join (GOJ), first introduced in [34] by Galindo-Legaria and Rosenthal, is adapted to the case of document databases in Definition 2.16. In the original paper, the formal definition of the GOJ involved the use of

the *outer-union* operator to introduce null-padding when uniting two tables with incompatible schemas, but since collections do not enforce schema, the (regular) union of collections with different schemas is valid for this case.

Definition 2.16 (Generalized Outer Join)**.** The GOJ on predicate α of collections $\mathcal{A}$ and $\mathcal{B}$ preserving paths $P \subseteq \text{cover}(\mathcal{A})$ is defined as:

$$\mathcal{A} \overset{\text{goj}}{\bowtie} [\alpha, P]\, \mathcal{B} = (\mathcal{A} \bowtie_\alpha \mathcal{B}) \cup (\pi^{\oslash}_{P}(\mathcal{A}) - \pi^{\oslash}_{P}(\mathcal{A} \bowtie_\alpha \mathcal{B}))$$

□

Nest Join and Lookup

The nest join is, essentially, a join followed by a nest operation where all matching documents from the collection to the right-hand side of the join are nested within matching documents from the collection to the left-hand side of the join. Definition 2.17 describes this formally.

Definition 2.17 (Nest Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, α a predicate and l the empty path or a label not realized in neither collection. The nest join of $\mathcal{C}_1$ and $\mathcal{C}_2$, denoted $\mathcal{C}_1 \hat{\bowtie}^l_\alpha \mathcal{C}_2$, is defined as follows: $\mathcal{C}_1 \hat{\bowtie}^l_\alpha \mathcal{C}_2 = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{p:\text{arragg}(\text{cover}(\mathcal{C}_2))}(\mathcal{C}_1 ⟗_\alpha \mathcal{C}_2)$ □

Similar to the regular join operator, other nest join variants exist. For any two collections $\mathcal{C}_1$ and $\mathcal{C}_2$:

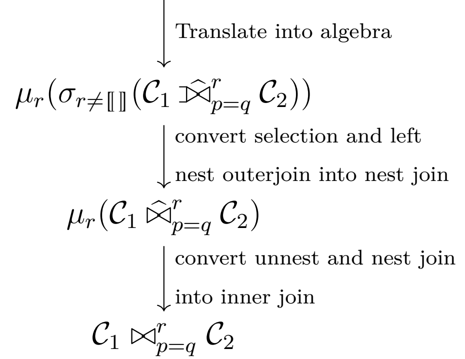
Definition 2.18 (Left Nest Semijoin)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, α a predicate and l the empty path or a label not realized in neither collection. The left nest semijoin is formally defined as: $\mathcal{C}_1 \hat{\ltimes}^l_\alpha \mathcal{C}_2 = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_2))}(\mathcal{C}_1 \ltimes_\alpha \mathcal{C}_2)$. □

Definition 2.19 (Left Nest Outerjoin)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, α a predicate and l the empty path or a label not realized in neither collection. The left nest outerjoin is formally defined as: $\mathcal{C}_1 \hat{⟕}^l_\alpha \mathcal{C}_2 = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_2))}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2)$. □

Of particular interest is the *nest left outerjoin*, a variant of the nest join that behaves like MongoDB's `$lookup` stage from the aggregation pipeline query language. Since `$lookup` is the only means MongoDB has to join collections, all inner joins in MongoDB are typically emulated through the use of three pipeline stages: `$lookup`, `$match` and `$unwind`. Figure 36a shows their interaction in more detail. These stages

```
db.C1.aggregate([
    {$lookup:{
        from: C2
        localfield: p
        foreignfield: q
        as: r
    }},
    {$match:{r:{$ne:[]}}},
    {$unwind:r}
])
```

(a) Typical MongoDB pipeline stages emulating an inner join.



(b) Simplification of MongoDB's emulated inner join into an algebraic inner join.

Figure 36. Structure of inner join emulated in MongoDB and its algebraic equivalence. Paths p and q appear in documents from $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively.

are roughly equivalent to the proposed algebra operators: *nest left outerjoin*, *select* and *unnest* respectively. Thus, a one-to-one translation is feasible. Furthermore, algebraic properties discussed later in the chapter allow the simplification of this one-to-one translation into a simpler algebraic expression, enabling more room for optimization. Figure 36b illustrates this.

In other words, *nest left outerjoin* is essentially a collection lookup on the right-hand side collection. It identifies whether one or more documents from this collection can be grouped together based on a specific document from the left-hand side collection. This operation can be conceptualized as categorizing documents from the right-hand side collection, where each document from the left-hand side collection acts as a category. Henceforth, this thesis adopts the term *lookup* to refer to the *nest left outerjoin*, as it better encapsulates its behavior.

Multiset operations

Definition 2.20 (Union)**.** Let $\mathcal{C}_1 = (\mathfrak{K}_1, mult^{\mathfrak{K}_1})$ and $\mathcal{C}_2 = (\mathfrak{K}_2, mult^{\mathfrak{K}_2})$ be collections, and $mult^{\mathfrak{K}_1 \cup \mathfrak{K}_2}(\mathsf{d}) = \max\left(mult^{\mathfrak{K}_1}(\mathsf{d}), mult^{\mathfrak{K}_2}(\mathsf{d})\right)$ for all d in $\mathfrak{K}_1 \cup \mathfrak{K}_2$ be a multiplicity function. The union between $\mathcal{C}_1$ and $\mathcal{C}_1$ is defined as $\mathcal{C}_1 \cup \mathcal{C}_2 = (\mathfrak{K}_1 \cup \mathfrak{K}_2, mult^{\mathfrak{K}_1 \cup \mathfrak{K}_2})$. □

Definition 2.21 (Intersection)**.** Let $\mathcal{C}_1 = (\mathfrak{K}_1, mult^{\mathfrak{K}_1})$ and $\mathcal{C}_2 = (\mathfrak{K}_2, mult^{\mathfrak{K}_2})$ be collections, and $mult^{\mathfrak{K}_1 \cap \mathfrak{K}_2}(\mathsf{d}) = \min\left(mult^{\mathfrak{K}_1}(x), mult^{\mathfrak{K}_2}(\mathsf{d})\right)$ for all d in $\mathfrak{K}_1 \cap \mathfrak{K}_2$ be a multiplicity function. The intersection between $\mathcal{C}_1$ and $\mathcal{C}_1$ is defined as
$\mathcal{C}_1 \cap \mathcal{C}_2 = (\mathfrak{K}_1 \cap \mathfrak{K}_2, mult^{\mathfrak{K}_1 \cap \mathfrak{K}_2})$. □

Definition 2.22 (Difference)**.** Let $\mathcal{C}_1 = (\mathfrak{K}_1, mult^{\mathfrak{K}_1})$ and $\mathcal{C}_2 = (\mathfrak{K}_2, mult^{\mathfrak{K}_2})$ be collections, and $mult^{\mathfrak{K}_1 - \mathfrak{K}_2}(\mathsf{d}) = \max\left(mult^{\mathfrak{K}_1}(\mathsf{d}) - mult^{\mathfrak{K}_2}(\mathsf{d}), 0\right)$ for all d in $\mathfrak{K}_1 - \mathfrak{K}_2$ be a multiplicity function. The difference between $\mathcal{C}_1$ and $\mathcal{C}_1$ is defined as
$\mathcal{C}_1 - \mathcal{C}_2 = (\mathfrak{K}_1 - \mathfrak{K}_2, mult^{\mathfrak{K}_1 - \mathfrak{K}_2})$. □

Rename

Definition 2.23 (Rename)**.** Let $\mathcal{C}$ be a collection, p and q path expressions such that $p \overset{r}{\backsim} \operatorname{cover}(\mathcal{C})$, $\neg q \overset{r}{\backsim} \operatorname{cover}(\mathcal{C})$ and p and q are not array path expressions. Renaming p to q across documents in $\mathcal{C}$ is defined as:

$$\rho_{q \leftarrow p}(\mathcal{C}) = \langle \delta(\mathsf{d}) \mid \mathsf{d} \in \mathcal{C} \rangle \text{ with}$$

$$\delta = \begin{cases} (\mathsf{d} - \operatorname{branch}(\mathsf{d}, p)) \cup \{\!\!\{ q : \operatorname{subtree}(\mathsf{d}, p) \}\!\!\} & \text{if } p \in N(\mathsf{d}) \\ \mathsf{d} & \text{otherwise} \end{cases}$$

□

3 JSON Document Algebra Properties

Later in the section, some equivalences heavily rely on the exclusion of documents from the result of certain operations based on the truth value of predicates involved in the respective operator or the behavior of such operator itself when predicates do

not apply. Specifically, Let $\mathcal{C}$ be a collection, α be a predicate, and OP_1 and OP_2 be operators such that OP_1 relies on α, and OP_2 does not involve any predicate. If α is guaranteed to be false for a sub-multiset $\mathcal{G}$ of $\mathcal{C}$ where documents in $\mathcal{G}$ share some property $prop_1$, α is said to *reject* $prop_1$. Consequently, OP_1 excludes documents in $\mathcal{G}$ from its result and OP_1 is said to reject $prop_1$ under α. If OP_2 rejects some property $prop_2$, the operator itself (i.e. OP_2) is said to reject $prop_2$. For instance, $\pi_P^{\oslash}(\mathcal{C})$ excludes all empty documents from $\mathcal{C}$, thus the operator $\pi^{\oslash}$ rejects empty documents. The following types of rejections are particularly useful later on when re-ordering and transforming operators:

Path absense rejection A predicate α is said to *reject absent paths* in a set of paths P if:

- α is a simple or conjunctive expression and its truth value is false on every document in which at least one path in P is absent.
- α is disjunctive expression and its truth value is false on every document in which all paths in P are absent.

An operator rejects absent paths if documents where at least one path in P is absent do not affect the operator's result. For example, Let $\mathcal{B}$ be a collection with information about employees of a company with path "dept" evaluating to the name of the department an employee belongs to. In $\sigma_{\text{dept="HR"}\wedge\text{age}>20}(\mathcal{B})$ the predicate dept="HR" $\wedge$ age > 20 rejects absent paths "dept", "age", any subset and superset of $N\left(\text{branch}\left(dp^{\mathcal{B}}, \text{"dept"}\right)\right) \cup N\left(\text{branch}\left(dp^{\mathcal{B}}, \text{"age"}\right)\right)$. On the other hand, in $\sigma_{\text{dept="HR"}\vee\text{age}>20}(\mathcal{B})$, the predicate dept="HR" $\vee$ age > 20 rejects absent paths on $\{\text{"dept"}, \text{"age"}\}$ and any superset of $N\left(\text{branch}\left(dp^{\mathcal{B}}, \text{"dept"}\right)\right) \cup N\left(\text{branch}\left(dp^{\mathcal{B}}, \text{"age"}\right)\right)$.

Empty object/array rejection A predicate is said to *reject empty objects/arrays* in a set of paths P if its truth value is false on every document in which at least one

path in P evaluates to an empty object/array. An operator rejects empty objects/arrays if documents in which at least one path in P evaluates to an empty object/array do not affect the operator's result.

Emptiness rejection A predicate is said to *reject emptiness* in a set of paths P if it rejects absent paths, empty objects and empty arrays in P. An operator rejects emptiness if it rejects absent paths, empty objects and empty arrays.

For (non-outer) joins and selection, the operator rejects emptiness if the predicate rejects emptiness. A left outer join $\mathcal{A} ⟕_{\alpha} \mathcal{B}$ rejects emptiness on P if α rejects emptiness on P and $P \subseteq dp^{\mathcal{B}}$, as does lookup. A full outer join does not reject emptiness as all inputs are let through.

Additionally, the following lemmas are useful to define and prove some equivalences later on:

Definition 3.1 (Document extension)**.** Given two documents a and b with value functions ϕ^a and ϕ^b respectively, a is inclusively extended by b (represented as $a \lessdot\!\!\!\!\!_\;\, b$) if $Lf(a) \subseteq Lf(b)$ and for every path x in $Lf(a)$ the value of x in a and b is the same. Formally,

$$a \leqslant b \leftrightarrow Lf(a) \subseteq Lf(b) \wedge \forall x(x \in Lf(a) \rightarrow \phi^a(x) = \phi^b(x)) \tag{4}$$

In contrast, a is exclusively extended by b (represented as $a \lessdot b$) when $Lf(a) \subset Lf(b)$:

$$a \lessdot b \leftrightarrow Lf(a) \subset Lf(b) \wedge \forall x(x \in Lf(a) \rightarrow \phi^a(x) = \phi^b(x)) \tag{5}$$

Lemma 3.1 (Reflexivity of $\leqslant$)**.** For any document d, $d \leqslant d$.

Proof. Let ϕ^d denote the value function of d. Then, it is always the case that $Lf(d) \subseteq Lf(d)$ and for any path $p \in Lf(d)$, $\phi^d(p) = \phi^d(p)$. Therefore $d \leqslant d$. $\square$

Lemma 3.2 (Transitivity of $\lesssim$)**.** For any triplet of documents a, b and c, if $a \lesssim b$ and $b \lesssim c$, then $a \lesssim c$.

Proof. Let ϕ^a, ϕ^b and ϕ^c be the value functions of a, b and c respectively. It is known by the premise that (1) $Lf(a) \subseteq Lf(b) \wedge Lf(b) \subseteq Lf(c)$ and (2) $\forall p\,(p \in Lf(a) \rightarrow \phi^a(p) = \phi^b(p) \wedge \phi^b(p) = \phi^c(p))$. Thus, (1) implies $Lf(a) \subseteq Lf(c)$ (by transitivity of $\subseteq$) and (2) implies $\phi^a(p) = \phi^c(p)$ (by transitivity of $=$). Therefore, $a \lesssim c$. □

Lemma 3.3 (Antisymmetry of $\lesssim$)**.** For any pair of documents a and b, if $a \lesssim b$ and $b \lesssim a$ then $a = b$.

Proof. Let ϕ^a and ϕ^b be the value functions of a and b. It is known by the premise that (1) $Lf(a) \subseteq Lf(b) \wedge Lf(b) \subseteq Lf(a)$, (2) $\forall p\,(p \in Lf(a) \rightarrow \phi^a(p) = \phi^b(p))$, and (3) $\forall p\,(p \in Lf(b) \rightarrow \phi^b(p) = \phi^a(p))$. Thus, (1) implies $Lf(a) = Lf(b)$ (by antisymetry of $\subseteq$) and (2) and (3) imply $\phi^a = \phi^b$. Therefore, $a = b$. □

Definition 3.2 (Collection extension)**.** Given two collections $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ is extended by $\mathcal{B}$ (represented as $\mathcal{A} \subseteq \mathcal{B}$) if for every document $a \in \mathcal{A}$ there exists a document $b \in \mathcal{B}$ such that $a \lesssim b$. Formally,

$$\mathcal{A} \subseteq \mathcal{B} \leftrightarrow \forall x(x \in \mathcal{A} \rightarrow \exists y(y \in \mathcal{B} \wedge x \lesssim y)) \tag{6}$$

Lemma 3.4 (Reflexivity of $\subseteq$)**.** Let $\mathcal{A}$ be a collection, then $\mathcal{A} \subseteq \mathcal{A}$.

Proof. Let x be an arbitrary element of $\mathcal{A}$, then prove $x \in \mathcal{A} \rightarrow \exists x(x \in \mathcal{A} \wedge x \lesssim x)$:

1.	$x \in \mathcal{A}$	premise
2.	$x \lesssim x$	reflexivity of $\lesssim$
3.	$x \in \mathcal{A} \wedge x \lesssim x$	conjunction
4.	$\exists x(x \in \mathcal{A} \wedge x \lesssim x)$	existential generalization

therefore $x \in \mathcal{A} \rightarrow \exists x(x \in \mathcal{A} \wedge x \lesssim x)$ and, by universal generalization, $\forall x\,(x \in \mathcal{A} \rightarrow \exists x(x \in \mathcal{A} \wedge x \lesssim x))$, which is the definition of $\mathcal{A} \subseteq \mathcal{A}$. □

Lemma 3.5 (Transitivity of $\subseteq$)**.** Let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be collections. If $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{C}$ then $\mathcal{A} \subseteq \mathcal{C}$.

Proof. Let x, y and z be arbitrary elements from $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ respectively, then prove $x \in \mathcal{A} \to \exists z(z \in \mathcal{C} \wedge x \lessdot z)$:

1.	$x \in \mathcal{A}$	premise
2.	$\forall x\,(x \in \mathcal{A} \to \exists y(y \in \mathcal{B} \wedge x \lessdot y))$	def. $\mathcal{A} \subseteq \mathcal{B}$
3.	$\forall y\,(y \in \mathcal{B} \to \exists z(z \in \mathcal{C} \wedge y \lessdot z))$	def. $\mathcal{B} \subseteq \mathcal{C}$
4.	$\exists y\,(y \in \mathcal{B} \wedge x \lessdot y)$	modus ponens (1,2)
5.	$\exists y\,(\exists z(z \in \mathcal{C} \wedge y \lessdot z) \wedge x \lessdot y)$	modus ponens (3,4)
6.	$\exists y \exists z\,(z \in \mathcal{C} \wedge y \lessdot z \wedge x \lessdot y)$	prenex form of 5
7.	$\exists y \exists z\,(z \in \mathcal{C} \wedge (x \lessdot y \wedge y \lessdot z))$	associativity and commutativity of $\wedge$
8.	$\exists z\,(z \in \mathcal{C} \wedge \exists y(x \lessdot y \wedge y \lessdot z))$	Aristotelian form *Some P's are Q's* of 7
9.	$\exists z\,(z \in \mathcal{C} \wedge x \lessdot z)$	Transitivity of $\lessdot$

therefore $x \in \mathcal{A} \to \exists z(z \in \mathcal{C} \wedge x \lessdot z)$ and, by universal generalization, $\forall x\,(x \in \mathcal{A} \to \exists z(z \in \mathcal{C} \wedge x \lessdot z))$, which is the definition of $\mathcal{A} \subseteq \mathcal{C}$. □

Lemma 3.6 (Antisymmetry of $\subseteq$)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be collections. If $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{A}$, then $\mathcal{A} = \mathcal{B}$.

Proof. Let a be an arbitrary document from $\mathcal{A}$. Then, there exists some document $b \in \mathcal{B}$ such that $a \lessdot b$ and $b \lessdot a$ (by assumption). Therefore, $a = b$ (by antisymmetry of $\lessdot$). □

Corollary 3.3. $\subseteq$ is a partial ordering.

Proof. $\subseteq$ is reflexive, transitive and antisymmetric. □

Projection transformations

Projections can be pushed down or introduced in a query to reduce the amount of data that needs to be processed, particularly on these scenarios:

- **Selective projection:** If a projection removes significantly big subtrees of a sub/document, it can lead to substantial savings in memory consumption, processing time and, depending on how a document is stored in disk, I/O.
- **Before joins and unwinds:** Pushing down projections before a join or unwind operation allows the optimizer to work with smaller intermediate results by reducing the size of elements to unnest in the case of unwind, the size of documents to join in the case of join and sometimes even the number of input documents passed to either operator.
- **Before aggregations:** Introducing projections before a group operation prunes unnecessary subtrees from documents, which reduces the size of data being aggregated.

A list of projection transformations considered in this manuscript is presented next.

Projection simplification

Lemma 3.7. Let P and Q be sets of paths, $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ be collections, such that $\mathcal{R}, \mathcal{S} \subseteq \mathcal{C}$, $\mathcal{S} = \pi^{\oslash}_{P}(\mathcal{C})$, and $\mathcal{R} = \pi^{\oslash}_{Q}(\mathcal{C})$. Then $\mathcal{S} \subseteq \mathcal{R}$ if and only if $P \subseteq Q$.

Proof. First, assume $\mathcal{S} \subseteq \mathcal{R}$ and prove $P \subseteq Q$. Suppose, for the sake of contradiction, that $P \nsubseteq Q$, then there exists at least one path p in P that is not in Q. Consider the case of some document $c \in \mathcal{C}$ having path p, but no other path in P nor Q. Then,

1. There exists some document $s \in \mathcal{S}$ such that $s \leq c$ (by def. $\mathcal{S}$).
2. s has path p (by def. $\mathcal{S}$).

3. There exists some document $r \in \mathcal{R}$ such that $s \lessdot r$ (by assumption).

4. r has path p (by 2 and 3).

5. Contradiction: No document in $\mathcal{R}$ can extend s because $p \notin Q$.

Second, assume $P \subseteq Q$ and prove $\mathcal{S} \subseteq \mathcal{R}$. Let s be an arbitrary document of $\mathcal{S}$. Then,

1. Any arbitrary path p from s is in P (by def. $\mathcal{S}$).

2. $p \in Q$ (by 1 and assumption).

3. All paths from s are in Q (generalization of 2).

4. There exists some document $r \in \mathcal{R}$ such that $s \lessdot r$ (by 3 and def. $\mathcal{R}$)

□

Lemma 3.8. Let P and Q be sets of paths, $\mathcal{C}$, $\mathcal{R}$ and $\mathcal{S}$ be collections, such that $\mathcal{R}, \mathcal{S} \subseteq \mathcal{C}$, $\pi_P^{\oslash}(\mathcal{C}) = \mathcal{S}$ and $\pi_Q^{\oslash}(\mathcal{C}) = \mathcal{R}$. Then $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) = \pi_P^{\oslash}(\mathcal{C})$ if and only if $P \subseteq Q$.

Proof. First, assume $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) = \pi_P^{\oslash}(\mathcal{C})$ and prove $P \subseteq Q$:

1. $\pi_P^{\oslash}(\mathcal{R}) = \pi_P^{\oslash}(\mathcal{C})$ (by def. $\mathcal{R}$ and assumption).

2. For all document $s \in \pi_P^{\oslash}(\mathcal{C})$ there exists a document $r \in \mathcal{R}$ such that $s \lessdot r$ (by 1).

3. There exists a document $c \in \mathcal{C}$ such that $s \lessdot r \lessdot c$ (by 2 and def. $\mathcal{R}$).

4. Suppose, for the sake of contradiction, that $P \nsubseteq Q$. Then there is at least one path $p \in P$ not in Q. Consider a document $d \in \mathcal{C}$ with path p, but no path in Q.

5. There exists a document $x \in \pi_P^{\oslash}(\mathcal{C})$ such that $x \lessdot d$ (by 4 and def. π)

6. There is no document $y \in \mathcal{R}$ such that $y \lessdot d$ (by 4).

7. Contradiction: Since $x \lessdot d$, (3) states there must be at least one document in $\mathcal{R}$ that extends x and is, in turn, extended by d.

Second, assume $P \subseteq Q$ and prove $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) = \pi_P^{\oslash}(\mathcal{C})$. The following proof shows that $\pi_P^{\oslash}(\mathcal{C}) \subseteq \pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C}))$:

1. Let s be an abitrary document from $\mathcal{S}$

2. $\mathcal{S} \lessdot\!\!\!\subseteq \mathcal{R}$ (by lemma 3.7 and assumption).

3. There exists a document $r \in \mathcal{R}$ such that $s \lessdot r$ (by 1,2).

4. s only has paths in P, thus a projection of P over $\mathcal{R}$ transforms r into s. Therefore $s \in \pi_P^{\oslash}(\mathcal{R})$ (by 3 and def. π).

The following proof shows that $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) \subseteq \pi_P^{\oslash}(\mathcal{C})$:

1. Let s be an abitrary document from $\pi_P^{\oslash}(\mathcal{R})$.

2. There exist two documents r and c from $\mathcal{R}$ and $\mathcal{C}$ respectively such that $s \lessdot r \lessdot c$ (by assumption and def. π).

3. $s \lessdot c$ (by 2 and transitivity of $\lessdot$).

4. s only has paths in P, thus a projection of P over $\mathcal{C}$ transforms c into s. Therefore $s \in \pi_P^{\oslash}(\mathcal{C})$.

$\square$

Lemma 3.9 (Projection simplification)**.** Let P and Q be sets of paths, $\mathcal{C}$, $\mathcal{R}$ and $\mathcal{S}$ be collections, such that $\mathcal{R}, \mathcal{S} \lessdot\!\!\!\subseteq \mathcal{C}$, $\pi_P^{\oslash}(\mathcal{C}) = \mathcal{S}$ and $\pi_Q^{\oslash}(\mathcal{C}) = \mathcal{R}$. Then $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) = \pi_P^{\oslash}(\mathcal{C})$ if and only if $\mathcal{S} \lessdot\!\!\!\subseteq \mathcal{R}$.

Proof. By lemma 3.8, $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) = \pi_P^{\oslash}(\mathcal{C})$ iff $P \subseteq Q$ and by lemma 3.7, $P \subseteq Q$ iff $\mathcal{S} \lessdot\!\!\!\subseteq \mathcal{R}$. Therefore, $\pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C})) = \pi_P^{\oslash}(\mathcal{C})$ iff $\mathcal{S} \lessdot\!\!\!\subseteq \mathcal{R}$. $\square$

Given a collection $\mathcal{C}$ and an operation O, a preserving projection in a expression of the form $O(\pi_P(\mathcal{C}))$ can be transformed into an un-preserving projection if O rejects absent paths. Most notably,

Lemma 3.10 (Selection makes projection non-preserving on absent path rejection)**.** Let P be a set of paths and α a selection predicate. If α rejects absent paths in P, then $\sigma_\alpha(\pi_P(\mathcal{C})) = \sigma_\alpha(\pi_P^{\oslash}(\mathcal{C}))$.

Proof. After the projection $\pi_P(\mathcal{C})$, any document where at least one path from exprpaths(α) is missing is filtered out by the subsequent selection. Let $\mathcal{R}$ be a submultiset of $\mathcal{C}$ where documents have at least one path from P. The output of $\pi_P(\mathcal{C})$ contains $|\mathcal{C}|$ documents: those in $\mathcal{R}$, plus $|\mathcal{C} - \mathcal{R}|$ empty documents. Note that $\mathcal{R} = \emptyset$ if no path in P is present in any document within $\mathcal{C}$, in which case $|\mathcal{C} - \emptyset| = |C|$ empty documents would be passed to the subsequent selection. The output of $\sigma_\alpha(\pi_P(\mathcal{C}))$ is the multiset $\mathcal{S}$ consisting of documents from $\mathcal{R}$ that have all paths in exprpaths(α). Similarly, the output of $\pi_P^{\oslash}(\mathcal{C})$ contains documents from $\mathcal{R}$, excluding all $|\mathcal{C} - \mathcal{R}|$ empty documents. Thus, $\sigma_\alpha(\pi_P^{\oslash}(\mathcal{C}))$ outputs documents from $\mathcal{R}$ with all paths in exprpaths(α), i.e., $\mathcal{S}$ (which is empty when $\mathcal{R}$ is empty). Therefore, $\sigma_\alpha(\pi_P(\mathcal{C})) = \mathcal{S} = \sigma_\alpha(\pi_P^{\oslash}(\mathcal{C}))$. □

Example 3.1 (Selection makes projection non-preserving on absent path rejection)**.** non-preserving projections can be formed by breaking down a selection through a conjunctive expression. For instance,

$$\begin{aligned}
\sigma_{p\cdot x\theta q}(\pi_{p\cdot x}(\mu_p^{\oslash}(\mathcal{C}))) &= \sigma_{p\cdot x\theta q \wedge \exists p\cdot x}(\pi_{p\cdot x}(\mu_p^{\oslash}(\mathcal{C}))) \\
&= \sigma_{p\cdot x\theta q}(\sigma_{\exists p\cdot x}(\pi_{p\cdot x}(\mu_p^{\oslash}(\mathcal{C})))) \\
&= \sigma_{p\cdot x\theta q}(\pi_{p\cdot x}^{\oslash}(\mu_p^{\oslash}(\mathcal{C})))
\end{aligned}$$

□

Lemma 3.11 (Unnest makes projection non-preserving based on prefixes)**.** Let P be a set of path expressions and q be a path expression used as a parameter for the unnest operator. If $q \in P$, then $\mu_q^{\oslash}(\pi_P(\mathcal{C})) = \mu_q^{\oslash}(\pi_P^{\oslash}(\mathcal{C}))$.

Proof. First, proof that $\mu_q^{\oslash}(\pi_P(\mathcal{C})) \subseteq \mu_q^{\oslash}(\pi_P^{\oslash}(\mathcal{C}))$. Let d_1 be a document in $\mu_q^{\oslash}(\pi_P(\mathcal{C}))$.

1. Assume $\mathsf{d}_1 = \{\!\!\{q : u, p_1 : v_1, p_2 : v_2, \ldots, p_n : v_n\}\!\!\}$, where $p_1, \ldots, p_n$ are paths in P.

2. By the definition of unnest, there exists a document $\mathsf{d}_2 \in \pi_P(\mathcal{C})$ such that $\mathsf{d}_2 \overset{\text{arr}}{\hookrightarrow} q \downarrow$, for all $j \in [1, n]$ $\mathsf{d}_2 \hookrightarrow p_j = \mathsf{d}_1 \hookrightarrow p_j$ and exists a label $i \in \mathbb{N}^0$ for which $\mathsf{d}_2 \hookrightarrow q \cdot i = \mathsf{d}_1 \hookrightarrow q = u$.

3. By the definition of projection, there exists a document $\mathsf{d}_3 \in \mathcal{C}$ such that $\mathsf{d}_2 \lessdot \mathsf{d}_3$.

4. After operation $\pi_P^{\oslash}(\mathcal{C})$, d_3 is transformed into a new document d_4 such that $\mathsf{d}_4 \lessdot \mathsf{d}_3$.

5. By steps 1-4, it is known that $\mathsf{d}_4 \hookrightarrow p_j = \mathsf{d}_3 \hookrightarrow p_j = \mathsf{d}_2 \hookrightarrow p_j = \mathsf{d}_1 \hookrightarrow p_j = v_j$ for all $j \in [1, n]$.

6. By the assumption, $q \in P$, thus, by step 4, $\mathsf{d}_4 \hookrightarrow q = \mathsf{d}_3 \hookrightarrow q$.

7. After operation $\mu_q^{\oslash}(\pi_P(\mathcal{C}))$, multiple documents are produced from d_4. Amongst these new documents, there exists at least one, referred to as d_5, for which there exists a label $i \in \mathbb{N}^0$ such that $\mathsf{d}_4 \hookrightarrow q \cdot i = \mathsf{d}_3 \hookrightarrow q \cdot i = \mathsf{d}_1 \hookrightarrow q = u$ and, thus, $\mathsf{d}_5 = \{\!\!\{q : \mathsf{d}_4 \hookrightarrow q \cdot i, p_1 : \mathsf{d}_4 \hookrightarrow p_1, \ldots, p_n : \mathsf{d}_4 \hookrightarrow p_n\}\!\!\} = \{\!\!\{q : u, p_1 : \mathsf{d}_4 \hookrightarrow p_1, \ldots, p_n : \mathsf{d}_4 \hookrightarrow p_n\}\!\!\}$.

8. By steps 5 and 7, $\mathsf{d}_5 = \{\!\!\{q : u, p_1 : v_1, \ldots, p_n : v_n\}\!\!\} = \mathsf{d}_1$. Therefore, $\mathsf{d}_1 \in \mu_q^{\oslash}(\pi_P^{\oslash}(\mathcal{C}))$.

Second, proof that $\mu_q^{\oslash}(\pi_P^{\oslash}(\mathcal{C})) \subseteq \mu_q^{\oslash}(\pi_P(\mathcal{C}))$. Let d_1 be a document in $\mu_q^{\oslash}(\pi_P^{\oslash}(\mathcal{C}))$.

1. By the definition of projection and unnest, it is known that $d_1 = \{\!\!\{ q : u, p_1 : v_1, p_2 : v_2, \ldots, p_n : v_n \}\!\!\}$, where paths $p_1, p_2, \ldots, p_n$ and (by the assumption) q are in P.

2. By definition of unnest, there exists a document d_2 in $\pi_P(\mathcal{C})$ and a label $i \in \mathbb{N}^0$ such that $d_2 \hookrightarrow p_j = d_1 \hookrightarrow p_j$ for all $j \in [1, n]$ and $d_2 \hookrightarrow q \cdot i = d_1 \hookrightarrow q = u$.

3. By definition of projection, there exists a document $d_3 \in \mathcal{C}$ such that $d_2 \leq d_3$.

4. When applying operation $\pi_P(\mathcal{C})$, a new document d_4 is created such that $d_4 = \{\!\!\{ q : d_3 \hookrightarrow q, p_1 : d_3 \hookrightarrow p_1, \ldots, p_n : d_3 \hookrightarrow p_n \}\!\!\}$.

5. When applying operation $\mu_q^{\oslash}(\pi_P(\mathcal{C}))$, new documents are produced by unnest. By steps 2 to 4, it is known that of all documents produced from d_4, there is a document $d_5 = \{\!\!\{ q : u, p_1 : d_4 \hookrightarrow p_1, \ldots, p_n : d_4 \hookrightarrow p_n \}\!\!\}$.

6. By steps 1 to 5, $d_5 = \{\!\!\{ q : u, p_1 : d_3 \hookrightarrow p_1, \ldots, p_n : d_3 \hookrightarrow p_n \}\!\!\} = \{\!\!\{ q : u, p_1 : d_2 \hookrightarrow p_1, \ldots, p_n : d_2 \hookrightarrow p_n \}\!\!\} = \{\!\!\{ q : u, p_1 : d_1 \hookrightarrow p_1, \ldots, p_n : d_1 \hookrightarrow p_n \}\!\!\} = \{\!\!\{ q : u, p_1 : v_1, \ldots, p_n : v_n \}\!\!\} = d_1$.

□

Lemma 3.12 (Non-preserving projection dominates preserving projection)**.** Let P and Q be sets of paths. If $Q \subseteq P$, then $\pi_Q^{\oslash}(\pi_P(\mathcal{C})) = \pi_Q^{\oslash}(\pi_P^{\oslash}(\mathcal{C}))$

Projection push-down

An unpreserving projection following an unnest can be performed beforehand if the path (or paths) to be projected refers to a field within array elements of the array to unnest as shown in lemma 3.13.

Lemma 3.13 (Projection push-down on unnest)**.** Let p and x be paths. Then

$$\pi_{p \cdot x}^{\oslash}(\mu_p^{\oslash}(\mathcal{C})) = \mu_p^{\oslash}(\pi_{p \cdot * \cdot x}^{\oslash}(\mathcal{C}))$$

Proof. First, proof that $\pi^{\oslash}_{p\cdot x}(\mu^{\oslash}_{p}(\mathcal{C})) \subseteq \mu^{\oslash}_{p}(\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C}))$. Let d_1 be an arbitrary document in $\pi^{\oslash}_{p\cdot x}(\mu^{\oslash}_{p}(\mathcal{C}))$. The goal is to proof that $\mathsf{d}_1 \in \mu^{\oslash}_{p}(\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C}))$:

1. $\mathsf{d}_1 = \{\!\!\{p \cdot x : v\}\!\!\}$, where v is some value.

2. By definition of projection, there exists a document $\mathsf{d}_2 \in \mu^{\oslash}_{p}(\mathcal{C})$ such that $\mathsf{d}_1 \preceq \mathsf{d}_2$. Thus, $\mathsf{d}_2 \hookrightarrow p \cdot x = d_1 \hookrightarrow p \cdot x = v$.

3. By definition of unnest, there exists a label $i \in \mathbb{N}^0$ and a document $\mathsf{d}_3 \in \mathcal{C}$ such that $d_3 \hookrightarrow p \cdot i \cdot x = \mathsf{d}_2 \hookrightarrow p \cdot x = v$.

4. After the operation $\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C})$, a new document d_4 is created from d_3 such that $\mathsf{d}_4 = \{\!\!\{p \cdot i \cdot x : \mathsf{d}_3 \hookrightarrow p \cdot j \cdot x \mid \exists j\, p \cdot j \cdot x \overset{r}{\backsim} \mathsf{d}_3\}\!\!\}$.

5. From steps 3 and 4 it can be inferred that $\mathsf{d}_4 \hookrightarrow p \cdot i \cdot x = v$.

6. After the operation $\mu^{\oslash}_{p}(\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C}))$, new documents are created from d_4. From the definiton of unnest and step 5, it can be inferred that one of such documents is $\{\!\!\{p \cdot x : \mathsf{d}_4 \hookrightarrow p \cdot i \cdot x\}\!\!\} = \{\!\!\{p \cdot x : v\}\!\!\} = \mathsf{d}_1$. Therefore, $\mathsf{d}_1 \in \mu^{\oslash}_{p}(\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C}))$.

Second, proof that $\pi^{\oslash}_{p\cdot x}(\mu^{\oslash}_{p}(\mathcal{C})) \supseteq \mu^{\oslash}_{p}(\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C}))$. Let d_1 be an arbitrary document in $\mu^{\oslash}_{p}(\pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C}))$. The goal is to proof that $\mathsf{d}_1 \in \pi^{\oslash}_{p\cdot x}(\mu^{\oslash}_{p}(\mathcal{C}))$:

1. By definition of unnest, there exists a label $i \in \mathbb{N}^0$ and a document $\mathsf{d}_2 \in \pi^{\oslash}_{p\cdot*\cdot x}(\mathcal{C})$ such that $\mathsf{d}_1 \hookrightarrow p = \mathsf{d}_2 \hookrightarrow p \cdot i$.

2. From the projection, it can be inferred that $p \cdot i \cdot x \overset{r}{\backsim} \mathsf{d}_2$.

3. From step 2 and definition of unnest, it can be inferred that $\mathsf{d}_1 = \{\!\!\{p \cdot x : v\}\!\!\}$, for some value v.

4. By definition of projection, there exists a document $\mathsf{d}_3 \in \mathcal{C}$ such that $\mathsf{d}_2 \preceq \mathsf{d}_3$.

5. By steps 1 and 4, it follows that there exist a label $j \in \mathbb{N}^0$ for which $\mathsf{d}_3 \hookrightarrow p \cdot j \cdot x = \mathsf{d}_2 \hookrightarrow p \cdot i \cdot x$.

6. After the operation $\mu_p^{\oslash}(\mathcal{C})$, multiple documents are produced from d_3. Amongst these new documents, there exists a document d_4 for which $\mathsf{d}_4 \hookrightarrow p \cdot x = \mathsf{d}_3 \hookrightarrow p \cdot j \cdot x = \mathsf{d}_2 \hookrightarrow p \cdot i \cdot x = \mathsf{d}_1 \hookrightarrow p \cdot x$.

7. After the operation $\pi_{p \cdot x}^{\oslash}(\mu_p^{\oslash}(\mathcal{C}))$, a new document d_5 is produced, such that $\mathsf{d}_5 = \{\!\{p \cdot x : \mathsf{d}_4 \hookrightarrow p \cdot x\}\!\} = \{\!\{p \cdot x : \mathsf{d}_1 \hookrightarrow p \cdot x\}\!\} = \{\!\{p \cdot x : v\}\!\} = \mathsf{d}_1$. Therefore, $\mathsf{d}_1 \in \pi_{p \cdot x}^{\oslash}(\mu_p^{\oslash}(\mathcal{C}))$.

□

For the case of joins, a projection push-down is done similarly to relational algebra, but instead of relying on the fixed schemas of tables, it relies on the collection cover.

Lemma 3.14 (Projection push-down on join)**.** Let P be a set of paths. If there exists a subset Q of $\text{cover}(\mathcal{A})$ such that $Q \subseteq P$ then

$$\pi_P(\mathcal{A} \bowtie_\alpha \mathcal{B}) = \pi_P(\pi_{Q \cup \text{exprpaths}(\alpha)}(\mathcal{A}) \bowtie_\alpha \mathcal{B})$$

Likewise, if $Q \subseteq \text{cover}(\mathcal{B})$ and $Q \subseteq P$, then

$$\pi_P(\mathcal{A} \bowtie_\alpha \mathcal{B}) = \pi_P(\mathcal{A} \bowtie_\alpha \pi_{Q \cup \text{exprpaths}(\alpha)}(\mathcal{B}))$$

Lemma 3.15 (Projection is distributive over join)**.** Let $\mathcal{C}_1, \mathcal{C}_2$ be collections, α a join predicate and P, Q, R sets of paths such that $P \subseteq \text{cover}(\mathcal{C}_1) \cup \text{cover}(\mathcal{C}_2)$. If $Q = (P \cap \text{cover}(\mathcal{C}_1)) \cup (\text{cover}(\mathcal{C}_1) \cap \text{cover}(\mathcal{C}_2))$ and $R = (P \cap \text{cover}(\mathcal{C}_2)) \cup (\text{cover}(\mathcal{C}_1) \cap \text{cover}(\mathcal{C}_2))$, then

$$\pi_P^{\oslash}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) = \pi_P^{\oslash}(\pi_Q^{\oslash}(\mathcal{C}_1) \bowtie_\alpha \pi_R^{\oslash}(\mathcal{C}_2)) \text{ and } \pi_P(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) = \pi_P(\pi_Q(\mathcal{C}_1) \bowtie_\alpha \pi_R(\mathcal{C}_2))$$

Lemma 3.16 (Projection push-down on lookup)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ be collections, α a predicate, l a label and P a set of paths. If there exists some subset Q of $\text{cover}(\mathcal{C}_2)$ such that $\{l \cdot * \cdot q \mid q \in Q\} \subseteq P$, then

$$\pi_P^{\oslash}(\mathcal{C}_1 \bowtie_\alpha^l \mathcal{C}_2) = \pi_P^{\oslash}(\mathcal{C}_1 \bowtie_\alpha^l \pi_{Q \cup \text{exprpaths}(\alpha)}^{\oslash}(\mathcal{C}_2))$$

Lemma 3.17 (Projection is distributive over lookup)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ be collections, α a predicate, l a label and P, Q and S sets of paths such that $Q \subseteq \text{cover}\,(\mathcal{C}_1)$, $S = \text{exprpaths}\,(\alpha)$, and S can be split into $S_1 = \{s \in S \mid s \in \text{cover}\,(\mathcal{C}_1)\}$ and $S_2 = \{s \in S \mid s \in \text{cover}\,(\mathcal{C}_2)\}$ such that $S_1 \sqcup S_2$. If there exists some subset R of $\text{cover}\,(\mathcal{C}_2)$ such that $P = \{l \cdot i \cdot r \mid \forall r \in R\, \exists i \in \mathbb{N}^0\} \sqcup Q$, then

$$\pi^{\oslash}_{P}\left(\mathcal{C}_1 \mathbin{\hat{⟕}}^{l}_{\alpha} \mathcal{C}_2\right) = \pi^{\oslash}_{P}\left(\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \mathbin{\hat{⟕}}^{l}_{\alpha} \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2)\right)$$

Proof. First, proof that $\pi^{\oslash}_{P}\left(\mathcal{C}_1 \mathbin{\hat{⟕}}^{l}_{\alpha} \mathcal{C}_2\right) \subseteq \pi^{\oslash}_{P}\left(\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \mathbin{\hat{⟕}}^{l}_{\alpha} \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2)\right)$. Let d_1 be a document in $\pi^{\oslash}_{P}(\mathcal{C}_1 \mathbin{\hat{⟕}}^{l}_{\alpha} \mathcal{C}_2)$.

1. By definition of projection, it is known that $\mathsf{d}_1 = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots, p_n : v_n\}\!\!\}$, where $p_1, \ldots, p_n \in P$ and $v_1, \ldots v_n$ are values.

2. By assumption, there exist paths $p_k, p_{k+1}, \ldots, p_m$, all realized in d_1, such that $1 \leq k \leq m \leq n$ and $l \preceq p_k, l \preceq p_{k+1}, \ldots, l \preceq p_m$.

3. By assumption, paths not preceeded by l must be in Q and since every suffix of l starts with a number in $\mathbb{N}^0$, it must be true that $\mathsf{d}_1 \overset{\text{arr}}{\hookrightarrow} l \downarrow$.

4. Let $q_1, q_2, \ldots, q_h$ be paths such that $\{q_1, q_2, \ldots, q_h\} = \{p_1, \ldots, p_n\} - \{p_k, p_{k+1}, \ldots, p_m\}$. Then, by steps 2-3, $q_1, q_2, \ldots, q_h \in Q$.

5. By steps 1 and 4, $\mathsf{d}_1 = \{\!\!\{q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, p_k : v_k, p_{k+1} : v_{k+1}, \ldots, p_m : v_m\}\!\!\}$, where $w_1, \ldots, w_h$ are values.

6. By step 5 and definition of JSON array, paths $\mathsf{d}_1 = \{\!\!\{q_1 : w_1, \ldots, q_h : w_h, l : [\![\mathsf{u}_1, \ldots, \mathsf{u}_z]\!]\}\!\!\}$ such that $\mathsf{u}_2, \ldots, \mathsf{u}_z$ are documents and for each u_i, with $1 \leq i \leq z$, exists at least one $r \in R$ for which $r \overset{r}{\backsim} \mathsf{u}_i$.

7. By definition of projection, there exists a document d_2 in $\mathcal{C}_1 \mathbin{\hat{⟕}}^{l}_{\alpha} \mathcal{C}_2$ such that $\mathsf{d}_1 \leq \mathsf{d}_2$.

8. By the definition of lookup and steps 6-7, there must exist a document $\mathsf{a}_1 \in \mathcal{C}_1$ and multiple documents $\mathsf{b}_1, \mathsf{b}_2, \ldots, \mathsf{b}_z \in \mathcal{C}_2$ such that for all $q \in Q$, if $q \overset{r}{\curvearrowleft} \mathsf{a}_1$, then $\mathsf{a}_1 \hookrightarrow q = \mathsf{d}_2 \hookrightarrow q$; and for every u_i, with $i \in [1, z]$, $\mathsf{u}_\mathsf{i} \preceq \mathsf{b}_i$.

9. When applying the operation $\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1)$, a_1 is transformed into a new document a_2 such that $\mathsf{a}_2 = \{\!\!\{ q_1 : \mathsf{a}_1 \hookrightarrow q_1, \ldots, q_h : \mathsf{a}_1 \hookrightarrow q_h \}\!\!\} \cup \{\!\!\{ s : \mathsf{a}_1 \hookrightarrow s \mid s \in S_1 \wedge s \overset{r}{\curvearrowleft} \mathsf{a}_1 \}\!\!\}$.

10. By steps 8-9, $\mathsf{a}_2 = \{\!\!\{ q_1 : \mathsf{d}_2 \hookrightarrow q_1, \ldots, q_h : \mathsf{d}_2 \hookrightarrow q_h \}\!\!\} \cup \{\!\!\{ s : \mathsf{a}_1 \hookrightarrow s \mid s \in S_1 \wedge s \overset{r}{\curvearrowleft} \mathsf{a}_1 \}\!\!\} = \{\!\!\{ q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h \}\!\!\} \cup \{\!\!\{ s : \mathsf{a}_1 \hookrightarrow s \mid s \in S_1 \wedge s \overset{r}{\curvearrowleft} \mathsf{a}_1 \}\!\!\}$.

11. When applying the operation $\pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2)$, $\mathsf{b}_1, \ldots, \mathsf{b}_z$ are transformed into new documents $\mathsf{c}_1, \ldots, \mathsf{c}_z$ such that for all $i \in [1, z]$, $\mathsf{c}_i = \{\!\!\{ r : b_i \hookrightarrow r \mid r \in R \wedge r \overset{r}{\curvearrowleft} b_i \}\!\!\} \cup \{\!\!\{ s : \mathsf{b}_i \hookrightarrow s \mid s \in S_2 \wedge s \overset{r}{\curvearrowleft} b_i \}\!\!\}$.

12. By steps 6-8 and 11, it is known that for all $i \in [1, z]$, $\mathsf{c}_i = \mathsf{u}_i \cup \{\!\!\{ s : \mathsf{b}_i \hookrightarrow s \mid s \in S_2 \wedge s \overset{r}{\curvearrowleft} b_i \}\!\!\}$.

13. When applying the operation $\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \widehat{\bowtie}^l_\alpha \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2)$, documents a_2 and $\mathsf{c}_1, \ldots, \mathsf{c}_z$ are merged into a new document $\mathsf{a}_3 = \{\!\!\{ q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, l : [\![\mathsf{c}_i \mid 1 \leq i \leq z]\!] \}\!\!\} \cup \{\!\!\{ s : \mathsf{a}_1 \hookrightarrow s \mid s \in S_1 \wedge s \overset{r}{\curvearrowleft} \mathsf{a}_1 \}\!\!\}$.

14. When applying the operation $\pi^{\oslash}_P \left(\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \widehat{\bowtie}^l_\alpha \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2) \right)$, a_3 is transformed into document $\mathsf{a}_4 = \{\!\!\{ q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, l : [\![\mathsf{c}_i - \{\!\!\{ s : \mathsf{b}_i \hookrightarrow s \mid s \in S_2 \wedge s \overset{r}{\curvearrowleft} b_i \}\!\!\} \mid 1 \leq i \leq z]\!] \}\!\!\}$.

15. By steps 13-14, $\mathsf{a}_4 = \{\!\!\{ q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, l : [\![\mathsf{u}_1, \ldots, \mathsf{u}_z]\!] \}\!\!\} = \mathsf{d}_1$. Therefore, $\mathsf{d}_1 \in \pi^{\oslash}_P \left(\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \widehat{\bowtie}^l_\alpha \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2) \right)$.

Second, proof that $\pi^{\oslash}_P \left(\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \widehat{\bowtie}^l_\alpha \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2) \right) \subseteq \pi^{\oslash}_P \left(\mathcal{C}_1 \widehat{\bowtie}^l_\alpha \mathcal{C}_2 \right)$. Let d_1 be a document in $\pi^{\oslash}_P \left(\pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \widehat{\bowtie}^l_\alpha \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2) \right)$

1. By definition of projection, it is known that $\mathsf{d}_1 = \{\!\!\{ p_1 : v_1, p_2 : v_2, \ldots, p_n : v_n \}\!\!\}$, where $p_1, \ldots, p_n \in P$ and $v_1, \ldots v_n$ are values.

2. By assumption, there exist paths $p_k, p_{k+1}, \ldots, p_m$, all realized in d_1, such that $1 \leq k \leq m \leq n$ and $l \preceq p_k, l \preceq p_{k+1}, \ldots, l \preceq p_m$.

3. By assumption, paths not preceeded by l must be in Q and since every suffix of l starts with a number in $\mathbb{N}^0$, it must be true that $\mathsf{d}_1 \overset{\text{arr}}{\hookrightarrow} l \downarrow$.

4. Let $q_1, q_2, \ldots, q_h$ be paths such that $\{q_1, q_2, \ldots, q_h\} = \{p_1, \ldots, p_n\} - \{p_k, p_{k+1}, \ldots, p_m\}$. Then, by steps 2-3, $q_1, q_2, \ldots, q_h \in Q$.

5. By steps 1 and 4, $\mathsf{d}_1 = \{\!\!\{ q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, p_k : v_k, p_{k+1} : v_{k+1}, \ldots, p_m : v_m \}\!\!\}$, where $w_1, \ldots, w_h$ are values.

6. By step 5 and definition of JSON array, paths $\mathsf{d}_1 = \{\!\!\{ q_1 : w_1, \ldots, q_h : w_h, l : [\![\mathsf{u}_1, \ldots, \mathsf{u}_z]\!] \}\!\!\}$ such that $\mathsf{u}_2, \ldots, \mathsf{u}_z$ are documents and for each u_i, with $1 \leq i \leq z$, exists at least one $r \in R$ for which $r \overset{r}{\frown} \mathsf{u}_i$.

7. By the definition of projection, there exists a document $\mathsf{d}_2 \in \pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1) \bowtie^{l}_{\alpha} \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2)$ such that $\mathsf{d}_1 \leq \mathsf{d}_2$.

8. By definition of lookup, there exists a document $\mathsf{a}_1 \in \pi^{\oslash}_{Q \cup S_1}(\mathcal{C}_1)$ such that $\mathsf{a}_1 \leq \mathsf{d}_2$ and $\mathsf{a}_1 \hookrightarrow q_i = \mathsf{d}_2 \hookrightarrow q_i$ for all $i \in [1, h]$.

9. By steps 7-8, it is implied that $\mathsf{a}_1 = \{\!\!\{ q_1 : \mathsf{d}_2 \hookrightarrow q_1, q_2 : \mathsf{d}_2 \hookrightarrow q_2, \ldots, q_h : \mathsf{d}_2 \hookrightarrow q_h \}\!\!\} \cup \{\!\!\{ s : \mathsf{d}_2 \hookrightarrow s \mid \exists s \in S_1 \}\!\!\} = \{\!\!\{ q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h \}\!\!\} \cup \{\!\!\{ s : \mathsf{d}_2 \hookrightarrow s \mid \exists s \in S_1 \}\!\!\}$.

10. By definition of projection and step 9, there exists a document $\mathsf{a}_2 \in \mathcal{C}_1$ such that $\mathsf{a}_1 \leq \mathsf{a}_2$.

11. By step 6 and the definition of lookup and projection, there exist documents $\mathsf{b}_1, \ldots, \mathsf{b}_z \in \pi^{\oslash}_{R \cup S_2}(\mathcal{C}_2)$ such that for all $i \in [1, z]$, $u_i \leq b_i$.

12. By step 7 and the definition of projection, there exist documents $\mathsf{c}_1, \ldots, \mathsf{c}_z \in \mathcal{C}_2$ such that for all $i \in [1, z]$, $b_i \leq c_i$. This implies that for all $s \in S_2$, if $s \overset{r}{\frown} \mathsf{b}_i$ then $\mathsf{c}_i \hookrightarrow s = \mathsf{b}_i \hookrightarrow s$; and for all paths $u \in N(\mathsf{u}_i)$, $\mathsf{c}_i \hookrightarrow u = \mathsf{u}_i \hookrightarrow u$.

13. When applying the operation $\mathcal{C}_1 \mathbin{\hat{\bowtie}}^{l}_{\alpha} \mathcal{C}_2$, documents a_2 and $\mathsf{c}_1, \ldots, \mathsf{c}_k$ are merged into a new document a_3 such that for all $i \in [1, h]$, $\mathsf{a}_3 \hookrightarrow q_i = \mathsf{a}_2 \hookrightarrow q_i$, for all $s \in S_1$, if $s \in \mathsf{a}_2$, $\mathsf{a}_3 \hookrightarrow s = \mathsf{a}_2 \hookrightarrow s$, and $\mathsf{a}_3 \hookrightarrow l = [\![\mathsf{c}_i \mid 1 \leq i \leq z]\!]$.

14. By steps 6-10 and 13, it is implied that for all $i \in [1, h]$, $\mathsf{a}_3 \hookrightarrow q_i = w_i$.

15. When applying the operation $\pi^{\oslash}_{P}\left(\mathcal{C}_1 \mathbin{\hat{\bowtie}}^{l}_{\alpha} \mathcal{C}_2\right)$, document a_3 is transformed into a document $\mathsf{a}_4 = \{\!| q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, l : [\![\{\!| r : \mathsf{b}_i \hookrightarrow r \mid \exists r \in R \wedge r \overset{r}{\backsim} \mathsf{b}_i |\!\} \mid 1 \leq i \leq z]\!] |\!\} = \{\!| q_1 : w_1, q_2 : w_2, \ldots, q_h : w_h, l : [\![\mathsf{u}_1, \ldots, \mathsf{u}_z]\!] |\!\} = \mathsf{d}_1$. Therefore, $\mathsf{d}_1 \in \pi^{\oslash}_{P}\left(\mathcal{C}_1 \mathbin{\hat{\bowtie}}^{l}_{\alpha} \mathcal{C}_2\right)$.

□

Lemma 3.18 (Projection push-down on union). Let $\mathcal{C}_1$, $\mathcal{C}_2$ be collections and P a set of paths. Then,

$$\pi^{\oslash}_{P}(\mathcal{C}_1 \cup \mathcal{C}_2) = \pi^{\oslash}_{P}(\mathcal{C}_1) \cup \pi^{\oslash}_{P}(\mathcal{C}_2) \text{ and } \pi_{P}(\mathcal{C}_1 \cup \mathcal{C}_2) = \pi_{P}(\mathcal{C}_1) \cup \pi_{P}(\mathcal{C}_2)$$

Lemma 3.19 (Projection push-down on selection). Let $\mathcal{C}$ be a collection, P a set of paths and α a selection predicate. Then, $\pi^{\oslash}_{P}(\sigma_{\alpha}(\mathcal{C})) = \pi^{\oslash}_{P}\left(\sigma_{\alpha}\left(\pi^{\oslash}_{P \cup \text{exprpaths}(\alpha)}(\mathcal{C})\right)\right)$ and $\pi_{P}(\sigma_{\alpha}(\mathcal{C})) = \pi_{P}\left(\sigma_{\alpha}\left(\pi_{P \cup \text{exprpaths}(\alpha)}(\mathcal{C})\right)\right)$ if α is not a simple or conjuctive expression that selects documents with absent paths.

Projection introduction

Lemma 3.20 (Projection introduction on group). Let P be a set of paths, $q_1, q_2, \ldots q_k$ paths from *cover* $(\mathcal{C})$, $r_1, r_2, \ldots, r_k$ labels and $f_1, f_2, \ldots, f_k$ aggregates. Then

$${}_{P}\gamma_{r_1:f_1(q_1), r_2:f_2(q_2), \ldots, r_k:f_k(q_k)}(\mathcal{C}) = {}_{P}\gamma_{r_1:f_1(q_1), r_2:f_2(q_2), \ldots, r_k:f_k(q_k)}(\pi_{P \cup \{q_1, \ldots, q_k\}}(\mathcal{C}))$$

Selection transformations

Lemmas 3.21,3.22,3.25 and 3.26, presented below, are typical selection transformations found in relational algebra and commonly used by RDBMSs.

Lemma 3.21 (Implied existence of paths from selection predicate)**.** Let α be a selection predicate with exprpaths $(\alpha) = \{p_1, p_2, \ldots, p_k\}$. Then, $\alpha = \alpha \wedge \exists p_1 \wedge \exists p_2 \wedge \cdots \wedge \exists p_k$.

Proof. The truth value of α depends on the existence of paths within exprpaths (α). Thus, if any of $p_1, p_2, \ldots, p_k$ is missing from a document, α is guaranteed to be false. If all paths exist, α can either be true or false depending on what each path evaluates to and the comparison operators involved. As such, path existance is required for (but does not guarantee) α to be true. The condition $\exists p_1 \wedge \exists p_2 \wedge \cdots \wedge \exists p_k$ only checks for the existence of each path involved in α and does, therefore, not affect its truth value:

α	$\exists p_1 \wedge \exists p_2 \wedge \cdots \wedge \exists p_k$	$\alpha \wedge (\exists p_1 \wedge \exists p_2 \wedge \cdots \wedge \exists p_k)$
T	T	T
F	T	F
F	F	F

□

Lemma 3.22 (Selection simplification)**.** Let α and β be match expressions, and $\mathcal{C}$, $\mathcal{R}$ and $\mathcal{S}$ be collections, such that $\mathcal{R}, \mathcal{S} \subseteq \mathcal{C}$, $\sigma_\alpha(\mathcal{C}) = \mathcal{S}$ and $\sigma_\beta(\mathcal{C}) = \mathcal{R}$. Then $\sigma_\alpha(\sigma_\beta(\mathcal{C})) = \sigma_\alpha(\mathcal{C})$ if and only if $\mathcal{S} \subseteq \mathcal{R}$.

Proof. First, assume $\sigma_\alpha(\sigma_\beta(\mathcal{C})) = \sigma_\alpha(\mathcal{C})$ and prove $\mathcal{S} \subseteq \mathcal{R}$. Suppose, for the sake of contradiction, that $\mathcal{S} \nsubseteq \mathcal{R}$. Then, there is at least one document d in $\mathcal{S}$ that is not in $\mathcal{R}$. However, since $\mathcal{S} = \sigma_\alpha(\mathcal{C}) = \sigma_\alpha(\sigma_\beta(\mathcal{C})) = \sigma_\alpha(\mathcal{R})$ (by assumption), this is a contradiction because d is not within the input (i.e. $\mathcal{R}$) passed to the selection in the expression $\sigma_\alpha(\mathcal{R})$ and thus cannot be part of $\mathcal{S}$. Second, assume $\mathcal{S} \subseteq \mathcal{R}$ and prove $\sigma_\alpha(\sigma_\beta(\mathcal{C})) = \sigma_\alpha(\mathcal{C})$. Note that $\sigma_\alpha(\sigma_\beta(\mathcal{C})) = \sigma_\alpha(\mathcal{R})$ (by def. $\sigma_\beta(\mathcal{C})$) and since $\mathcal{R} \supseteq \mathcal{S}$ (by assumption), $\sigma_\alpha(\mathcal{R}) = \sigma_\alpha(\mathcal{S})$. Thus, $\sigma_\alpha(\mathcal{S}) = \mathcal{S}$ (by idempotency of selection) and $\mathcal{S} = \sigma_\alpha(\mathcal{C})$ (by def. $\sigma_\alpha(\mathcal{C})$). □

Lemma 3.23 (Commutativity of selection)**.** Let φ and Ψ be selection predicates. The composition of two selections over a collection is commutative:

$$\sigma_{\varphi}\left(\sigma_{\Psi}(dp^{\mathcal{C}})\right) = \sigma_{\Psi}\left(\sigma_{\varphi}\left(\mathcal{C}\right)\right)$$

Lemma 3.24 (Selection is conjuction collapsible)**.** Let φ and Ψ be selection predicates, then

$$\sigma_{\varphi}\left(\sigma_{\Psi}(dp^{\mathcal{C}})\right) = \sigma_{\Psi\wedge\varphi}(\mathcal{C})$$

Lemma 3.25 (Selection push-down on join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and α, ψ_1 and ψ_2 be predicates. If $\text{exprpaths}\left(\psi_1\right) \subseteq \mathit{cover}\left(\mathcal{C}_1\right)$ and $\text{exprpaths}\left(\psi_2\right) \subseteq \mathit{cover}\left(\mathcal{C}_2\right)$, then

$$\sigma_{\psi_1\wedge\psi_2}(\mathcal{C}_1 \bowtie_{\alpha} \mathcal{C}_2) = \sigma_{\psi_1}(\mathcal{C}_1) \bowtie_{\alpha} \sigma_{\psi_2}(\mathcal{C}_2)$$

Lemma 3.26 (Selection push-down on set operators)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ be collections, ψ a predicate, and $\odot$ an arbitrary operator from the set $\{\cup, \cap, -\}$. Then

$$\sigma_{\psi}(\mathcal{C}_1 \odot \mathcal{C}_2) = \sigma_{\psi}(\mathcal{C}_1) \odot \sigma_{\psi}(\mathcal{C}_2)$$

In certain scenarios, the interplay between two operations, such as the unpreserving project and group-by operators illustrated in Example 3.2, provides an opportunity to minimize the input documents for either operation. Introducing a selection in such cases can effectively reduce the number of input documents for these operations. Lemma 3.27 formalizes this notion of introducing a selection before a group-by.

Example 3.2. Consider a collection $\mathcal{C}$ with documents:

⦃"a":1, "b":⦃"c":1⦄⦄, ⦃"a":1, "b": null ⦄, ⦃"a":1, "b":⦃"c":2⦄⦄, ⦃"a":2, "g":3⦄, ⦃"a":2, "g":4⦄, ⦃"b": ⦃"c":2⦄⦄, ⦃"b": ⦃"c":5⦄⦄

In the query $\pi^{\oslash}_{r\cdot*\cdot c}({}_{a}\gamma_{r:\text{arragg}(b)}(\mathcal{C}))$, the projection serves to filter out elements of array r within each group that are either not subdocuments or are subdocuments lacking the field "c". Moreover, it eliminates groups where all elements of r are candidates

for exclusion. By introducing a selection immediately preceding the group operation, the same desired behavior is maintained, simultaneously reducing the size of the input passed to the grouping process. Consequently, the query can be restructured as $\pi^{\oslash}_{r\cdot*\cdot c}({}_{a}\gamma_{r:\text{arragg}(b)}(\sigma_{\exists b.c}(\mathcal{C})))$. The outcome of each operation is presented in Table 6.

$\sigma_{\exists b.c}(\mathcal{C})$	${}_{a}\gamma_{r:\text{arragg}(b)}(\sigma_{\exists b.c}(\mathcal{C}))$	$\pi^{\oslash}_{r\cdot*\cdot c}({}_{a}\gamma_{r:\text{arragg}(b)}(\sigma_{\exists b.c}(\mathcal{C})))$
⦃"a":1, "b":⦃"c":1⦄⦄, ~~⦃"a":1, "b": null ⦄~~, ⦃"a":1, "b":⦃"c":2⦄⦄, ~~⦃"a":2, "g":3⦄~~, ~~⦃"a":2, "g":4⦄~~, ⦃"b": ⦃"c":2⦄⦄, ⦃"b": ⦃"c":5⦄⦄	⦃"a":1, "r":⟦⦃"c":1⦄, ⦃"c":2⦄ ⟧⦄, ⦃"r":⟦⦃"c":2⦄, ⦃"c":5⦄ ⟧⦄	⦃"r":⟦⦃"c":1⦄, ⦃"c":2⦄ ⟧⦄, ⦃"r":⟦⦃"c":2⦄, ⦃"c":5⦄ ⟧⦄

Table 6. Example selection introduction on group operation based on the interaction between unpreserving project and group

If the query has a preserving projection instead, such as $\pi_{r\cdot*\cdot c}({}_{a}\gamma_{r:\text{arragg}(b)}(\mathcal{C}))$, groups are retained even if they lack nested documents with the field "c"; subsequently, these groups are transformed into empty documents. However, the removal of non-document elements from r still persists. As a result, the query can be restated as $\pi_{r\cdot*\cdot c}({}_{a}\gamma_{r:\text{arragg}(b)}(\sigma_{\exists b.c\vee(\neg\exists b)}(\mathcal{C})))$. The consequences of each operation are visually summarized in Table 7.

$\sigma_{\exists b.c\vee(\neg\exists b)}(\mathcal{C})$	${}_{a}\gamma_{r:\text{arragg}(b)}(\sigma_{\exists b.c}(\mathcal{C}))$	$\pi_{r\cdot*\cdot c}({}_{a}\gamma_{r:\text{arragg}(b)}(\sigma_{\exists b.c}(\mathcal{C})))$
⦃"a":1, "b":⦃"c":1⦄⦄, ~~⦃"a":1, "b": null ⦄~~, ⦃"a":1, "b":⦃"c":2⦄⦄, ⦃"a":2, "g":3⦄, ⦃"a":2, "g":4⦄, ⦃"b": ⦃"c":2⦄⦄, ⦃"b": ⦃"c":5⦄⦄	⦃"a":1, "r":⟦⦃"c":1⦄, ⦃"c":2⦄ ⟧⦄, ⦃"r":⟦⦃"c":2⦄, ⦃"c":5⦄ ⟧⦄, ⦃"a":2, "r":⟦⟧⦄	⦃"r":⟦⦃"c":1⦄, ⦃"c":2⦄ ⟧⦄, ⦃"r":⟦⦃"c":2⦄, ⦃"c":5⦄ ⟧⦄, ⦃⦄

Table 7. Example selection introduction on group operation based on the interaction between preserving project and group

□

Lemma 3.27 (Selection introduction on group)**.** Let $\mathcal{C}$ be a collection, $\tilde{F}_1 = ⦅l^1_1 : \text{arragg}\,(q^1_1)\,, \ldots, l^1_k : \text{arragg}\,(q^1_k)⦆$ and $\tilde{F}_2 = ⦅l^2_1 : f^2_1(q^2_1), \ldots, l_j : f_j(q_j)⦆$ vec-

tors of aggregates, and P, L, R sets of paths such that $L = \{l_1^1, \ldots, l_k^1\} \cup \{l_1^2, \ldots, l_j^2\}$, $L \subseteq P$, $paths(\tilde{F}_1) \cap paths(\tilde{F}_2) = \emptyset$, and
$R = \{q.x \mid (\exists l \in L)(\exists q \in \{q_1^1, \ldots, q_k^1\})(\exists x \neq \epsilon)[l.*.x \in P \wedge (\!(l : \text{arragg}\,(q))\!) \in \tilde{F}_1]\} = \{r_1, \ldots, r_h\}$. Then

$$\pi_P^{\oslash}({}_G\gamma_{\tilde{F}_1 \cdot \tilde{F}_2}(\mathcal{C})) = \pi_P^{\oslash}({}_G\gamma_{\tilde{F}_1 \cdot \tilde{F}_2}(\sigma_{\exists r_1 \vee \cdots \vee \exists r_h}(\mathcal{C})))$$

Join transformations

Lemma 3.28 shows that, similar to traditional extensions to relational algebra, semi-joins can be simulated by using a projection over a join.

Lemma 3.28 (Convertion from join to semijoin)**.** Let α be a selection predicate, and $\mathcal{C}_1$, $\mathcal{C}_2$ be collections. Then

$$\pi_{\text{cover}(\mathcal{C}_1)}^{\oslash}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) = \mathcal{C}_1 \ltimes_\alpha \mathcal{C}_2 \text{ and } \pi_{\text{cover}(\mathcal{C}_2)}^{\oslash}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) = \mathcal{C}_1 \rtimes_\alpha \mathcal{C}_2$$

Other common transformations like converting outer joins to non-outer joins are possible as well. This is shown in Lemma 3.29.

Lemma 3.29 (Conversion of Left Outer Join to Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, ψ and α predicates, and $Q = \text{exprpaths}(\alpha) \cap \text{cover}(\mathcal{C}_2)$. If ψ rejects absent paths in Q, then $\sigma_\psi(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) = \mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2$.

Proof. Let $\mathcal{R}$ be the output of $\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2$, $\mathcal{S}$ be the output of $\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2$, and d be an arbitrary document from $\sigma_\psi(\mathcal{R})$. The goal is to prove that $\mathsf{d} \in \mathcal{S}$ using the hypothesis that ψ rejects absent paths in Q, implying that all paths in Q must be present in d (i.e., $Q \subseteq N(\mathsf{d})$).

Assume, for the sake of contradiction, that $\mathsf{d} \notin \mathcal{S}$. By the definition of left outer join, all documents from $\mathcal{C}_1$ without a matching document from $\mathcal{C}_2$ are in $\mathcal{R} - \mathcal{S}$ and lack any path from $\text{cover}\,(\mathcal{C}_2)$. Therefore, any document in $\mathcal{R} - \mathcal{S}$ lacks all paths in Q. Since ψ rejects absent paths in Q, no document from $\mathcal{R} - \mathcal{S}$ is in $\sigma_\psi(\mathcal{R})$. Hence,

$\sigma_\psi(\mathcal{R}) = \mathcal{R} - (\mathcal{R} - \mathcal{S}) = \mathcal{S}$. This, however, is a contradiction because it implies that if $\mathsf{d} \in \sigma_\psi(\mathcal{R})$, then $\mathsf{d} \in \mathcal{S}$. $\square$

Lemma 3.30 (Conversion of Extended Left Outer Join to Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, ψ and α predicates, and l a non-empty label. If ψ rejects empty objects in $\{l\}$, then $\sigma_\psi(\mathcal{C}_1 ⟕^l_\alpha \mathcal{C}_2) = \mathcal{C}_1 \bowtie^l_\alpha \mathcal{C}_2$.

Proof. Let $\mathcal{R}$ be the output of $\mathcal{C}_1 ⟕^l_\alpha \mathcal{C}_2$, $\mathcal{S}$ be the output of $\mathcal{C}_1 \bowtie^l_\alpha \mathcal{C}_2$, and d be an arbitrary document from $\sigma_\psi(\mathcal{R})$. The goal is to prove that $\mathsf{d} \in \mathcal{S}$ using the hypothesis that ψ rejects empty objects in $\{l\}$, implying that l must be present in $N(\mathsf{d})$ and resolve to a non-empty object.

Assume, for the sake of contradiction, that $\mathsf{d} \notin \mathcal{S}$. By the definition of the extended left outer join, all documents from $\mathcal{C}_1$ without a matching document from $\mathcal{C}_2$ are extended by a document in $\mathcal{R} - \mathcal{S}$ in which path l resolves to an empty document. Therefore, path l resolves to an empty document in any document from $\mathcal{R} - \mathcal{S}$. Since ψ rejects empty documents in $\{l\}$, no document from $\mathcal{R} - \mathcal{S}$ is in $\sigma_\psi(\mathcal{R})$. Hence, $\sigma_\psi(\mathcal{R}) = \mathcal{R} - (\mathcal{R} - \mathcal{S}) = \mathcal{S}$. This, however, is a contradiction because it implies that if $\mathsf{d} \in \sigma_\psi(\mathcal{R})$, then $\mathsf{d} \in \mathcal{S}$. $\square$

Lemma 3.31 (Conversion of Lookup to Nest Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, ψ and α predicates, and l a non-empty label. If ψ rejects empty arrays in $\{l\}$, then $\sigma_\psi(\mathcal{C}_1 \mathbin{\widehat{⟕}}^l_\alpha \mathcal{C}_2) = \mathcal{C}_1 \mathbin{\widehat{\bowtie}}^l_\alpha \mathcal{C}_2$.

Proof. Let $\mathcal{R}$ be the output of $\mathcal{C}_1 \mathbin{\widehat{⟕}}^l_\alpha \mathcal{C}_2$, $\mathcal{S}$ be the output of $\mathcal{C}_1 \mathbin{\widehat{\bowtie}}^l_\alpha \mathcal{C}_2$, and d be an arbitrary document from $\sigma_\psi(\mathcal{R})$. The goal is to prove that $\mathsf{d} \in \mathcal{S}$ using the hypothesis that ψ rejects empty arrays in $\{l\}$, implying that l must be present in $N(\mathsf{d})$ and resolve to a non-empty array.

Assume, for the sake of contradiction, that $\mathsf{d} \notin \mathcal{S}$. By the definition of lookup, all documents from $\mathcal{C}_1$ without a matching document from $\mathcal{C}_2$ are extended by a document in $\mathcal{R} - \mathcal{S}$ in which path l resolves to an empty array. Therefore, path l resolves to an empty array in any document from $\mathcal{R} - \mathcal{S}$. Since ψ rejects empty arrays

in $\{l\}$, no document from $\mathcal{R}-\mathcal{S}$ is in $\sigma_\psi(\mathcal{R})$. Hence, $\sigma_\psi(\mathcal{R}) = \mathcal{R}-(\mathcal{R}-\mathcal{S}) = \mathcal{S}$. This, however, is a contradiction because it implies that if $\mathsf{d} \in \sigma_\psi(\mathcal{R})$, then $\mathsf{d} \in \mathcal{S}$. $\square$

Lemma 3.32 (Inner Join push-down on Nest Join)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ be collections, and $\alpha_{1,2}$ and $\alpha_{2,3}$ be join predicates.

$$\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \hat{\ltimes}^l_{\alpha_{2,3}} \mathcal{C}_3\right) = {}_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_3))} \left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)\right)$$

Proof. First, prove that
$\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \hat{\ltimes}^l_{\alpha_{2,3}} \mathcal{C}_3\right) \subseteq {}_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_3))} \left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)\right)$. Let d be a document from $\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \hat{\ltimes}^l_{\alpha_{2,3}} \mathcal{C}_3\right)$.

1. By definition of join, there exist two documents $\mathsf{d}_1 \in \mathcal{C}_1$ and $\mathsf{d}_2 \in \mathcal{C}_2 \hat{\ltimes}^l_{\alpha_{2,3}} \mathcal{C}_3$ such that $\mathsf{d} = \mathsf{d}_1 \cup \mathsf{d}_2$.

2. By definion of nest join, there exist a document $\mathsf{d}_3 \in \mathcal{C}_2$ and several documents $\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_k \in \mathcal{C}_3$ such that $\mathcal{C}_3 \ltimes_{\alpha_{2,3}} \mathcal{C}_2 = \langle \mathsf{c}_2, \ldots, \mathsf{c}_k \rangle$ and $\mathsf{d}_2 = \mathsf{d}_3 \cup \{\!\{ l : [\![\mathsf{c}_1, \ldots, \mathsf{c}_k]\!] \}\!\}$.

3. By step 2 and definition of join, when performing operation $\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3$, several new documents will be produced. Amongst them, there exist $\mathsf{a}_1 = \mathsf{d}_3 \cup \mathsf{c}_1, \mathsf{a}_2 = \mathsf{d}_3 \cup \mathsf{c}_2, \ldots, \mathsf{a}_k = \mathsf{d}_3 \cup \mathsf{c}_k$.

4. By steps 1, 3 and definition of join, when performing operation $\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)$, several new documents will be produced. Amongst them, there exist $\mathsf{b}_1 = \mathsf{d}_1 \cup \mathsf{a}_1, \mathsf{b}_2 = \mathsf{d}_1 \cup \mathsf{a}_2, \ldots, \mathsf{b}_k = \mathsf{d}_1 \cup \mathsf{a}_k$.

5. By steps 3-4, $\mathsf{b}_1 = \mathsf{d}_1 \cup \mathsf{d}_3 \cup \mathsf{c}_1, \mathsf{b}_2 = \mathsf{d}_1 \cup \mathsf{d}_3 \cup \mathsf{c}_2, \ldots, \mathsf{b}_k = \mathsf{d}_1 \cup \mathsf{d}_3 \cup \mathsf{c}_k$.

6. By step 5 and definition of group, when performing operation ${}_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_3))} \left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)\right)$, documents $\mathsf{b}_1, \ldots, \mathsf{b}_k$ are merged into the new document $\mathsf{d}_1 \cup \mathsf{d}_3 \cup \{\!\{ l : [\![\mathsf{c}_1, \ldots, \mathsf{c}_k]\!] \}\!\} = \mathsf{d}_1 \cup \mathsf{d}_2 = \mathsf{d}$. Therefore, $\mathsf{d} \in {}_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_3))} \left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)\right)$.

Second, prove that ${}_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_3))}\left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)\right) \subseteq \mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \hat{\bowtie}^{l}_{\alpha_{2,3}} \mathcal{C}_3\right)$. Let d be a document from ${}_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_3))}\left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)\right)$.

1. By definition of group, $\mathsf{d} = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n, q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m, l : [\![\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_k]\!]\}\!\!\}$, where $p_1, \ldots, p_n \in \text{cover}(\mathcal{C}_1)$, $q_1, \ldots, q_m \in \text{cover}(\mathcal{C}_2)$, $v_1, \ldots, v_n, u_1, \ldots, u_m$ values and for all $i \in [1, k]$, $N(\mathsf{c}_i) \subseteq \text{cover}(\mathcal{C}_3)$.

2. By definition of group and step 1, there exist multiple documents $\mathsf{d}_1, \mathsf{d}_2, \ldots, \mathsf{d}_k \in \mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3\right)$ such that $\mathsf{d}_1 = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n, q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\} \cup \mathsf{c}_1, \mathsf{d}_2 = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n, q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\} \cup \mathsf{c}_2, \ldots, \mathsf{d}_k = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n, q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\} \cup \mathsf{c}_k$.

3. By definition of join and step 2, there exists a document $\mathsf{a} \in \mathcal{C}_1$ such that $\mathsf{a} = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n\}\!\!\}$.

4. By definition of join and step 2, there exist multiple documents $\mathsf{b}_1, \ldots, \mathsf{b}_k \in \mathcal{C}_2 \bowtie_{\alpha_{2,3}} \mathcal{C}_3$ such that $\mathsf{b}_1 = \{\!\!\{q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\} \cup \mathsf{c}_1, \mathsf{b}_2 = \{\!\!\{q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\} \cup \mathsf{c}_2, \ldots, \mathsf{b}_k = \{\!\!\{q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\} \cup \mathsf{c}_k$.

5. By definition of join and steps 1, 4, there exists a document $\mathsf{e} \in \mathcal{C}_2$ such that $\mathsf{e} = \{\!\!\{q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m\}\!\!\}$ and it can be inferred that $\mathsf{c}_1, \ldots, \mathsf{c}_k \in \mathcal{C}_3$

6. When performing operation $\mathcal{C}_2 \hat{\bowtie}^{l}_{\alpha_{2,3}} \mathcal{C}_3$, new documents are produced. By steps 4-5, out of these documents, there exists a document $\mathsf{e}_1 = \mathsf{e} \cup \{\!\!\{l : [\![\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_k]\!]\}\!\!\}$.

7. When performing operation $\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \hat{\bowtie}^{l}_{\alpha_{2,3}} \mathcal{C}_3\right)$ new documents are produced. By steps 2,3 and 6, out of these documents, there exists a document $\mathsf{e}_2 = \mathsf{a} \cup \mathsf{e}_1 = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n\}\!\!\} \cup \mathsf{e}_1 = \{\!\!\{p_1 : v_1, p_2 : v_2, \ldots p_n : v_n\}\!\!\} \cup \{\!\!\{q_1 : u_1, q_2 : u_2, \ldots, q_m : u_m, l : [\![\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_k]\!]\}\!\!\} = \mathsf{d}$. Therefore, $\mathsf{d} \in \mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \hat{\bowtie}^{l}_{\alpha_{2,3}} \mathcal{C}_3\right)$.

$\square$

Lemma 3.33 (Projection eliminates Left Outer Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, an α a predicate. If $\mathcal{C}_1$ does not have empty documents, then $\pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) = \mathcal{C}_1$.

Delay nesting

Fundamentally, a nest join is equivalent to an inner join followed by a group operation that uses the arragg aggregate function to nest matching documents from the right-hand-side of the inner join. Thus, for any two collections $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{C}_1 \hat{\bowtie}^{l}_{\alpha} \mathcal{C}_2 = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_2))}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2)$. In a more complex expression with two or more nest joins forming a right-deep nest join tree, the same principle can be applied to convert the expression into a right-deep inner join tree. For example,

Example 3.3. Assume collections: $\mathcal{C}$, $\mathcal{O}$ and $\mathcal{L}$ analogous to tables Customer, Orders and Lineitem from the TPC-H benchmak. A typical query on a document database joins these three collections to create documents where the top level JSON fields correspond to a customer from $\mathcal{C}$ and an array with all corresponding orders from $\mathcal{O}$ such that each order has an array with the corresponding lineitems from $\mathcal{L}$. Algebraically this query can be modeled as follows:

$$\mathcal{C} \hat{\bowtie}^{\text{orders}}_{\text{c_custkey=o_custkey}} (\mathcal{O} \hat{\bowtie}^{\text{lineitems}}_{\text{o_orderkey=l_orderkey}} \mathcal{L})$$

Alternatively, the two nest joins in the expression above can be decomposed into two inner joins followed by two nest operations:

$$
\begin{aligned}
&{}_{\text{cover}(\mathcal{C})}\gamma_{\text{orders:arragg}(\text{cover}(\mathcal{O})\cup\{\text{lineitems}\})}(\\
&\quad {}_{\text{cover}(\mathcal{C})\cup\text{cover}(\mathcal{O})}\gamma_{\text{lineitems:arragg}(\text{cover}(\mathcal{L}))}(\\
&\qquad \mathcal{C} \bowtie_{\text{c_custkey=o_custkey}} (\mathcal{O} \bowtie_{\text{o_orderkey=l_orderkey}} \mathcal{L}) \\
&\quad) \\
&)
\end{aligned}
$$

□

Note that as the query tree is traversed down, starting from the first group operation, the paths to group by accumulate at each group operation based on the order of appearance of collections in the original expression, when read from left to right. For instance, if $\mathcal{C}$ appears first, the set of paths to group by at the first group is denoted as $\mathrm{cover}\,(\mathcal{C})$. When $\mathcal{O}$ appears as the second collection, the set of paths to group by at the second group becomes $\mathrm{cover}\,(\mathcal{C}) \cup \mathrm{cover}\,(\mathcal{O})$.

For a query involving an arbitrary number of collections $\mathcal{C}_1, \ldots, \mathcal{C}_k$, along with $k-1$ nest joins arranged in a right-deep nest join tree, the decomposed query tree comprises $k-1$ group operations. At the inner-most group (the $k-1$-th group), the grouping is done based on $\mathrm{cover}\,(\mathcal{C}_1) \cup \mathrm{cover}\,(\mathcal{C}_2) \cup \cdots \cup \mathrm{cover}\,(\mathcal{C}_{k-1})$, while the $k-2$-th group groups by $\mathrm{cover}\,(\mathcal{C}_1) \cup \mathrm{cover}\,(\mathcal{C}_2) \cup \cdots \cup \mathrm{cover}\,(\mathcal{C}_{k-2})$, and so on.

On the other hand, the set of paths to nest at the first $k-2$ group operations can be expressed as $\mathrm{cover}\,(\mathcal{C}_{i+1}) \cup \{l_{i+1,i+2}\}$, where $1 \leq i \leq k-2$, and $l_{i+1,i+2}$ denotes the label created by the nest join between collections $\mathcal{C}_{i+1}$ and $\mathcal{C}_{i+2}$. The $k-1$ group operation simply nests paths in $\mathrm{cover}\,(\mathcal{C}_k)$. A generalized version of this concept is provided in Lemma 3.34.

Lemma 3.34 (Delayed nesting on chained Nest joins)**.** Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ be collec-

tions, $l_{1,2}, l_{2,3}, l_{3,4}, \ldots, l_{n-1,n}$ be labels, and $\alpha_{1,2}, \alpha_{2,3}, \alpha_{3,4}, \ldots, \alpha_{n-1,n}$ be predicates.

$$\mathcal{C}_1 \,\hat{\bowtie}^{l_{1,2}}_{\alpha_{1,2}} \left(\mathcal{C}_2 \,\hat{\bowtie}^{l_{2,3}}_{\alpha_{2,3}} \left(\ldots \mathcal{C}_{n-2} \,\hat{\bowtie}^{l_{n-2,n-1}}_{\alpha_{n-2,n-1}} \left(\mathcal{C}_{n-1} \,\hat{\bowtie}^{l_{n-1,n}}_{\alpha_{n-1,n}} \mathcal{C}_n\right) \ldots\right)\right) =$$

$$_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2)\cup\{l_{2,3}\})}\Big($$

$$_{\text{cover}(\mathcal{C}_1)\cup\text{cover}(\mathcal{C}_2)}\gamma_{l_{2,3}:\text{arragg}(\text{cover}(\mathcal{C}_3)\cup\{l_{4,5}\})}\Big($$

$$\ldots$$

$$_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_{n-2})}\gamma_{l_{n-2,n-1}:\text{arragg}(\text{cover}(\mathcal{C}_{n-1})\cup\{l_{n-1,n}\})}\Big($$

$$_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_{n-1})}\gamma_{l_{n-1,n}:\text{arragg}(\text{cover}(\mathcal{C}_n))}\Big($$

$$\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\mathcal{C}_2 \bowtie_{\alpha_{2,3}} \left(\ldots \mathcal{C}_{n-2} \bowtie_{\alpha_{n-2,n-1}} \left(\mathcal{C}_{n-1} \bowtie_{\alpha_{n-1,n}} \mathcal{C}_n\right) \ldots\right)\right)$$

$$\Big)$$

$$\Big)$$

$$\ldots$$

$$\Big)$$

$$\Big)$$

Proof. Let $n \in \mathbb{N}$ be the number of nest joins in the expression $\mathcal{C}_1 \hat{\bowtie}^{l_{1,2}}_{\alpha_{1,2}} \left(\ldots \left(\mathcal{C}_n \hat{\bowtie}^{l_{n,n+1}}_{\alpha_{n,n+1}} \mathcal{C}_{n+1}\right) \ldots\right)$, and

$$P(n) = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2)\cup\{l_{2,3}\})}\Big(\ldots$$
$$_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_n)}\gamma_{l_{n,n+1}:\text{arragg}(\text{cover}(\mathcal{C}_{n+1}))}\left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \left(\ldots \left(\mathcal{C}_n \bowtie_{\alpha_{n,n+1}} \mathcal{C}_{n+1}\right) \ldots\right)\right) \ldots\Big)$$

for every $n \in \mathbb{N}$. Induction is used to prove that $\forall n > 0.P(n)$.

Base case:

By definition of nest join, $\mathcal{C}_1 \hat{\bowtie}^{l_{1,2}}_{\alpha_{1,2}} \mathcal{C}_2 = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2))}\left(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \mathcal{C}_2\right)$ and $P(n)$ evaluates to the same when $n = 1$. Hence $P(1)$ holds.

Induction step:

Let k be an arbitrary natural number such that $k > 1$ and assume that $P(i)$ holds

for all $i \in [1, k]$. Then, prove that $P(k+1)$ holds:

$$
\begin{aligned}
&\mathcal{C}_1 \,\widehat{\bowtie}^{l_{1,2}}_{\alpha_{1,2}}\, \big(\ldots \mathcal{C}_k \,\widehat{\bowtie}^{l_{k,k+1}}_{\alpha_{k,k+1}}\, \big(\mathcal{C}_{k+1} \,\widehat{\bowtie}^{l_{k+1,n+2}}_{\alpha_{k+1,k+2}}\, \mathcal{C}_{k+2}\big)\ldots\big) \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2))\cup\{l_{2,3}\}}\Big(\cdots\ {}_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_k)}\gamma_{l_{k,k+1}:\text{arragg}(\text{cover}(\mathcal{C}_{k+1}))}\Big(&& \text{(by hypothesis)} \\
&\qquad \mathcal{C}_1 \bowtie_{\alpha_{1,2}} \Big(\ldots \mathcal{C}_k \bowtie_{\alpha_{k,k+1}} \Big(\mathcal{C}_{k+1} \,\widehat{\bowtie}^{l_{k+1,k+2}}_{\alpha_{k+1,k+2}}\, \mathcal{C}_{k+2}\Big)\ldots\Big)\Big)\ldots\Big) \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2))\cup\{l_{2,3}\}}\Big(\cdots\ {}_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_k)}\gamma_{l_{k,k+1}:\text{arragg}(\text{cover}(\mathcal{C}_{k+1}))}\Big(&& \text{(by Def. group)} \\
&\qquad \mathcal{C}_1 \bowtie_{\alpha_{1,2}} \Big(\ldots \mathcal{C}_k \bowtie_{\alpha_{k,k+1}} {}_{\text{cover}(\mathcal{C}_{k+1})}\gamma_{l_{k+1,k+2}:\text{arragg}(\text{cover}(\mathcal{C}_{k+2}))}\Big(\\
&\qquad \mathcal{C}_{k+1} \bowtie_{\alpha_{k+1,k+2}} \mathcal{C}_{k+2}\Big)\ldots\Big)\Big)\ldots\Big) \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2))\cup\{l_{2,3}\}}\Big(\cdots\ {}_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_k)}\gamma_{l_{k,k+1}:\text{arragg}(\text{cover}(\mathcal{C}_{k+1}))}\Big(&& \text{(by join assoc)} \\
&\qquad \Big(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \ldots \bowtie_{\alpha_{k-1,k}} \mathcal{C}_k\Big) \bowtie_{\alpha_{k,k+1}} {}_{\text{cover}(\mathcal{C}_{k+1})}\gamma_{l_{k+1,k+2}:\text{arragg}(\text{cover}(\mathcal{C}_{k+2}))}\Big(\\
&\qquad \mathcal{C}_{k+1} \bowtie_{\alpha_{k+1,k+2}} \mathcal{C}_{k+2}\Big)\ldots\Big)\ldots\Big) \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2))\cup\{l_{2,3}\}}\Big(\cdots\ {}_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_k)}\gamma_{l_{k,k+1}:\text{arragg}(\text{cover}(\mathcal{C}_{k+1}))}\Big(&& \text{(by Lemma 3.32)} \\
&\qquad {}_{\text{cover}(\mathcal{C}_1\bowtie_{\alpha_{1,2}}\ldots\bowtie_{\alpha_{k-1,k}}\mathcal{C}_k)\cup\text{cover}(\mathcal{C}_{k+1})}\gamma_{l_{k+1,k+2}:\text{arragg}(\text{cover}(\mathcal{C}_{k+2}))}\Big(\\
&\qquad \Big(\mathcal{C}_1 \bowtie_{\alpha_{1,2}} \ldots \bowtie_{\alpha_{k-1,k}} \mathcal{C}_k\Big) \bowtie_{\alpha_{k,k+1}} \Big(\mathcal{C}_{k+1} \bowtie_{\alpha_{k+1,k+2}} \mathcal{C}_{k+2}\Big)\Big)\ldots\Big)\ldots\Big) \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l_{1,2}:\text{arragg}(\text{cover}(\mathcal{C}_2))\cup\{l_{2,3}\}}\Big(\cdots\ {}_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_k)}\gamma_{l_{k,k+1}:\text{arragg}(\text{cover}(\mathcal{C}_{k+1}))}\Big(&& \text{(by join assoc)} \\
&\qquad {}_{\text{cover}(\mathcal{C}_1)\cup\ldots\cup\text{cover}(\mathcal{C}_{k+1})}\gamma_{l_{k+1,k+2}:\text{arragg}(\text{cover}(\mathcal{C}_{k+2}))}\Big(\\
&\qquad \mathcal{C}_1 \bowtie_{\alpha_{1,2}} \Big(\ldots \mathcal{C}_k \bowtie_{\alpha_{k,k+1}} \Big(\mathcal{C}_{k+1} \bowtie_{\alpha_{k+1,k+2}} \mathcal{C}_{k+2}\Big)\Big)\Big)\ldots\Big)\ldots\Big)
\end{aligned}
$$

Therefore, for any number n of nest joins with $n > 0$, $P(n)$ holds. □

Outer Join reordering through the GOJ

When first devised in [34], Galindo-Legaria and Rosenthal introduced the generalized outer join to avoid wasteful materialization of intermediate results by changing the order of evaluation of join and outer join operations under certain circumstances. Example 3.4 illustrates one of such cases.

Example 3.4. Consider collections: $\mathcal{C}$, $\mathcal{O}$ and $\mathcal{L}$ analogous to tables Customer, Orders and Lineitem from the TPC-H benchmak respectively, and the query:

$$\sigma_{\text{c_nation}\cdot\text{name="UK"}}(\mathcal{C}) \;\text{⟕}_{\text{c_custkey=o_custkey}}\; (\mathcal{O} \bowtie_{\text{o_orderkey=l_orderkey}} \mathcal{L})$$

Evaluated as is, the query produces a big intermediate result: $\mathcal{O} \bowtie_{\text{o_orderkey=l_orderkey}}$ $\mathcal{L}$. If the predicate c_nation·name="UK" in the selection is highly selective, then most of the documents in this intermediate result are disposable. A better evaluation order would be to perform the left outer join between $\mathcal{C}$ and $\mathcal{O}$ first to remove any irrelevant orders, but as Galindo-Legaria and Rosenthal explain in their paper, such task is not trivial because the "intuitive" ways in which to rewrite the query are not equivalent to the original:

Way 1 Perform the left join first and the regular join last as follows:

$$(\sigma_{\text{c_nation·name="UK"}}(\mathcal{C}) ⟕_{\text{c_custkey=o_custkey}} \mathcal{O}) \bowtie_{\text{o_orderkey=l_orderkey}} \mathcal{L}$$

This, however, eliminates any customer without orders.

Way 2 Perform the left join between $\mathcal{C}$ and $\mathcal{O}$ first, then peform a left outer join with $\mathcal{L}$ as follows:

$$(\sigma_{\text{c_nation·name="UK"}}(\mathcal{C}) ⟕_{\text{c_custkey=o_custkey}} \mathcal{O}) ⟕_{\text{o_orderkey=l_orderkey}} \mathcal{L}$$

This, however, keeps orders for out-of-stock items, which would otherwise not have been preserved in the original expression. □

The order of evaluation of the original query from Example 3.4 can be transformed to compute the left outer join between $\mathcal{C}$ and $\mathcal{O}$ first and then use the GOJ to join the result with $\mathcal{L}$:

$$(\sigma_{\text{c_nation·name="UK"}}(\mathcal{C}) ⟕_{\text{c_custkey=o_custkey}} \mathcal{O}) \overset{\text{goj}}{\bowtie} [\text{o_orderkey=l_orderkey, cover}\,(\mathcal{C})]\, \mathcal{L}$$

A generalization of this property is provided in Lemma 3.35 and extended to the case of lookup in Lemma 3.36.

Lemma 3.35 (Left Outer Join - Inner Join Reordering with GOJ)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ be collections, and α and β be predicates. If β rejects absent paths on *cover* $(\mathcal{C}_2)$, then $\mathcal{C}_1 ⟕_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3) = (\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \overset{\text{goj}}{\bowtie} [\beta, \text{cover}\,(\mathcal{C}_1)]\, \mathcal{C}_3$.

Proof. The hypothesis for this proof is that β rejects absent paths on $\mathit{cover}\,(\mathcal{C}_2)$. The implication of this is that the expression $(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \bowtie_\beta \mathcal{C}_3$ (used in the proof) is equivalent to $(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) \bowtie_\beta \mathcal{C}_3$. This is because all documents in $\mathcal{C}_1 \overline{\ltimes} \mathcal{C}_2$ lack paths from $\mathcal{C}_2$, which means that β evaluates to false whenever these documents are considered for the inner join.

$$
\begin{aligned}
&(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \overset{\text{goj}}{\bowtie} [\beta,\, \text{cover}\,(\mathcal{C}_1)]\, \mathcal{C}_3 \\
&= ((\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \bowtie_\beta \mathcal{C}_3) \cup (\pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) - \pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}((\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \bowtie_\beta \mathcal{C}_3)) && \text{(by Def 2.16)} \\
&= ((\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) \bowtie_\beta \mathcal{C}_3) \cup (\pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) - \pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}((\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) \bowtie_\beta \mathcal{C}_3)) && \text{(by hypothesis)} \\
&= (\mathcal{C}_1 \bowtie_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3)) \cup (\pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) - \pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 \bowtie_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3))) && \text{(by join associativity)} \\
&= (\mathcal{C}_1 \bowtie_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3)) \cup (\pi^{\oslash}_{\text{cover}(\mathcal{C}_1)}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) - (\mathcal{C}_1 \ltimes_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3))) && \text{(by Corollary 2.13)} \\
&= (\mathcal{C}_1 \bowtie_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3)) \cup (\mathcal{C}_1 - (\mathcal{C}_1 \ltimes_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3))) && \text{(by Lemma 3.33)} \\
&= (\mathcal{C}_1 \bowtie_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3)) \cup (\mathcal{C}_1 \overline{\ltimes}_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3)) && \text{(by Def 2.14)} \\
&= \mathcal{C}_1 ⟕_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3) && \text{(by Def 2.15)}
\end{aligned}
$$

□

Lemma 3.36 (Lookup - Inner Join Reordering with GOJ)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ be collections, and α and β be predicates. If β rejects absent paths on $\mathit{cover}\,(\mathcal{C}_2)$, then

$$
\mathcal{C}_1 \,\widehat{⟕}^{l}_{\alpha}\, (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3) = {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_2) \cup \text{cover}(\mathcal{C}_3))}((\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \overset{\text{goj}}{\bowtie} [\beta,\, \text{cover}\,(\mathcal{C}_1)]\, \mathcal{C}_3)
$$

Proof.

$$
\begin{aligned}
&\mathcal{C}_1 \,\widehat{⟕}^{l}_{\alpha}\, (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3) \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_2) \cup \text{cover}(\mathcal{C}_3))}(\mathcal{C}_1 ⟕_\alpha (\mathcal{C}_2 \bowtie_\beta \mathcal{C}_3)) && \text{(by Def 2.19)} \\
&= {}_{\text{cover}(\mathcal{C}_1)}\gamma_{l:\text{arragg}(\text{cover}(\mathcal{C}_2) \cup \text{cover}(\mathcal{C}_3))}((\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) \overset{\text{goj}}{\bowtie} [\beta,\, \text{cover}\,(\mathcal{C}_1)]\, \mathcal{C}_3) && \text{(by Lemma 3.35)}
\end{aligned}
$$

□

Group transformations

Lemma 3.37 (Full Group push-down over Selection)**.** Let $\mathcal{C}$ be a collection, $l_1, l_2, \ldots, l_k$ labels not realized in cover $(\mathcal{C})$, $q_1, \ldots, q_k$ paths realized in cover $(\mathcal{C})$, and α a selection

predicate.

$$ {}_{\text{exprpaths}(\alpha)}\gamma_{l_1:f_1(q_1),\ldots,l_k:f_k(q_1)}(\sigma_\alpha(\mathcal{C})) = \sigma_\alpha({}_{\text{exprpaths}(\alpha)}\gamma_{l_1:f_1(q_1),\ldots,l_k:f_k(q_k)}(\mathcal{C})) $$

Lemma 3.38 (Partial Group push-down over Selection)**.** Let $\mathcal{C}$ be a collection, G a set of paths to group by, $l_1, \ldots, l_k, l'_1, \ldots, l'_k$ labels not in cover $(\mathcal{C})$, $q_1, \ldots, q_k$ paths in cover $(\mathcal{C})$, α a selection predicate and ξ the function:

$$ \xi(G, \alpha) = \begin{cases} \text{exprpaths}\,(\alpha) & \text{if } G \subset \text{exprpaths}\,(\alpha) \\ G \cup \text{exprpaths}\,(\alpha) & \text{if } G \not\subseteq \text{exprpaths}\,(\alpha) \end{cases} $$

then

$$ {}_G\gamma_{l_1:f_1(q_1),\ldots,l_k:f_k(q_k)}(\sigma_\alpha(\mathcal{C})) = {}_G\gamma_{l_1:f_1(l'_1),\ldots,l_k:f_k(l'_k)}(\sigma_\alpha({}_{\xi(G,\alpha)}\gamma_{l'_1:f_1(q_1),\ldots,l'_k:f_k(q_k)}(\mathcal{C}))) $$

Lemma 3.39 (Unwind simplification due to Group)**.** Let $\mathcal{C}$ be a collection, G a set of paths to group by, p a path in cover $(\mathcal{C})$, l a label not in cover $(\mathcal{C})$ and agg an aggregate function. If there exists a non-aggregate function f that treats constituents of arrays as a bag of values and performs the same operation as agg would over the same bag of values, then ${}_G\gamma_{l:agg(p)}(\mu_p^{\oslash}(\mathcal{C})) = {}_G\gamma_{l:agg(f(p))}(\mathcal{C})$.

Example 3.5. Let $\mathcal{C}$ be a collection, G a set of paths to group by, p a path in cover $(\mathcal{C})$ and l a label not in cover $(\mathcal{C})$. Consider the query ${}_G\gamma_{l:\text{sum}(p)}(\mu_p^{\oslash}(\mathcal{C}))$ and the non-aggregate function arr_sum(x), designed to ascertain the sum of all numerical constituents within the array indicated by x in cases where x resolves to an array. Then, by Lemma 3.39, the query can be reduced to ${}_G\gamma_{l:\text{sum}(\text{arr_sum}(p))}(\mathcal{C})$. $\square$

Lemma 3.40 (Partial Group push-down over Left Outer Join)**.** Let $\mathcal{C}_1$,$\mathcal{C}_2$ be collections, α a predicate, G a set of paths to group by, and $\tilde{F}$ a vector of scalar aggregate functions such that $\tilde{F}$ can be split into $\tilde{F}_1$ and $\tilde{F}_2$; and G can be split into $G_1 = G \cap \text{cover}\,(\mathcal{C}_1)$ and $G_2 = G \cap \text{cover}\,(\mathcal{C}_2)$. Then

$$ {}_G\gamma_{\tilde{F}}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2) = {}_G\gamma_{(\tilde{F}_1 \otimes (\exists c_2 ? c_2 : 1)) \cdot \tilde{F}'_2}(\mathcal{C}_1 ⟕_\alpha {}_{G_2 \cup \text{exprpaths}(\alpha)}\gamma_{\tilde{F}_2 \cdot (\!(c_2:\text{count}(*))\!)}(\mathcal{C}_2)) $$

Union centric push-downs

The practical utility of Lemmas 3.41, 3.42, 3.43, 3.43, and 3.44 becomes evident in scenarios where collections are fragmented into subcollections and distributed across multiple servers within a JSON Document Store's implementation. This partitioning strategy, commonly known as *sharding*, designates each subcollection as a *shard*. At a conceptual level, treating each shard as a subcollection enables the recreation of the original collection by consolidating all of its shards. These lemmas serve as foundational principles for a specific sharding technique known as *horizontal sharding*, where documents remain intact within each shard concerning the original collection.

Lemma 3.41 (Broadband Selection)**.** Let $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \ldots, \mathcal{C}_k$ be collections and ψ a predicate.

$$\sigma_\psi \left(\bigcup_{1 \leq i \leq k} \mathcal{C}_i \right) = \bigcup_{1 \leq i \leq k} \sigma_\psi \left(\mathcal{C}_i \right)$$

Lemma 3.42 (Broadband Projection)**.** Let $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \ldots, \mathcal{C}_k$ be collections and P a set of paths.

$$\pi_P \left(\bigcup_{1 \leq i \leq k} \mathcal{C}_i \right) = \bigcup_{1 \leq i \leq k} \pi_P \left(\mathcal{C}_i \right) \text{ and } \pi_P^\oslash \left(\bigcup_{1 \leq i \leq k} \mathcal{C}_i \right) = \bigcup_{1 \leq i \leq k} \pi_P^\oslash \left(\mathcal{C}_i \right)$$

Lemma 3.43 (Broadband Join)**.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be collections, and α a predicate. If $\mathcal{C}_2$ is split up into subcollections $\mathcal{C}_2^1, \mathcal{C}_2^2, \mathcal{C}_2^3, \ldots, \mathcal{C}_2^k$, then

$$\mathcal{C}_1 \bowtie_\alpha \bigcup_{1 \leq i \leq k} \mathcal{C}_2^i = \bigcup_{1 \leq i \leq k} \mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2^i$$

Lemma 3.44 (Broadband Unnest)**.** Let $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \ldots, \mathcal{C}_k$ be collections and p a path.

$$\mu_p \left(\bigcup_{1 \leq i \leq k} \mathcal{C}_i \right) = \bigcup_{1 \leq i \leq k} \mu_p \left(\mathcal{C}_i \right) \text{ and } \mu_p^\oslash \left(\bigcup_{1 \leq i \leq k} \mathcal{C}_i \right) = \bigcup_{1 \leq i \leq k} \mu_p^\oslash \left(\mathcal{C}_i \right)$$

Lemma 3.45 (Broadband Group)**.** Let $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \ldots, \mathcal{C}_k$ be collections, G a set of paths to group by, and $\tilde{F}$ a vector of aggregates such that $\tilde{F}$ can be split into vectors $\tilde{F}_1, \tilde{F}_2, \ldots, \tilde{F}_k$.

$$_G\gamma_{\tilde{F}} \left(\bigcup_{1 \leq i \leq k} \mathcal{C}_i \right) = {}_G\gamma_{(\tilde{F}_1' \otimes c_1) \cdot \ldots \cdot (\tilde{F}_k' \otimes c_k)} \left(\bigcup_{1 \leq i \leq k} {}_G\gamma_{\tilde{F}_i \cdot \langle\!\langle c_i : \text{count}(*) \rangle\!\rangle} \left(\mathcal{C}_i \right) \right)$$

CHAPTER V

DATA PILOTS

1 Collection summarization

This chapter commences by introducing the concept of a DataPilot, providing an initial intuitive understanding before delving into formal definitions. Consider a collection $\mathcal{C}$ of JSON Documents. A DataPilot of $\mathcal{C}$, denoted as $DP(\mathcal{C})$ or succinctly as $dp^{\mathcal{C}}$, takes form as a labeled tree. This tree serves as an abstract representation, encapsulating the overarching structure present across all documents within the underlying set $\mathfrak{K}$ of $\mathcal{C}$ (i.e. the collection schema), and keeping the frequency of appearances of every distinct pathway exhibited by JSON documents within $\mathfrak{K}$.

Intuitively, the reader may also visualize a DataPilot in a tabular format through the following steps:

1. Start with a generic collection of JSON Documents.

2. Represent each document as a JSON tree and list all distinct paths. For example, given the collection in Listing V.1:

```
{"name": {"first": "John", "last": "Doe" },"age": 20}
{"name": "Jane Doe", "age": 24}
{"name": {"first": "Chris", "last": "Smith" },"age": 30}
```

Listing V.1. Example collection of documents

Paths are: ϵ, "name", "name"·"first", "name"·"last", "age".

3. Construct a table where rows correspond to paths, columns to types, and the intersection of a path p and type t signifies the count of documents with p of type t. Notably, as all documents have the empty path ϵ as the tree's root, the intersection of ϵ with the *JSON Object* type column encapsulates the total document count in the collection. Following the example, Table 8 shows a tabular representation of the DataPilot for the collection in Listing V.1.

Path	obj	arr	int	str
ϵ	3	0	0	0
name	2	0	0	1
name·first	0	0	0	1
name·last	0	0	0	1
age	0	0	3	0

Table 8. DataPilot from Listing V.1

Regarding arrays, numeric labels are substituted with the special label $\star$ in the DataPilot, signifying the presence of an array. For example, in a collection $\mathcal{C}$ with the two documents:

```
{"hobbies":[{"name":"soccer"},"tennis"]}
{"hobbies":["tennis",{"name":"soccer"}]}
```

Paths "hobbies"·1·"name"from the first document and "hobbies"·2·"name"from the second document are both stored in $dp^{\mathcal{C}}$ as "hobbies"·$\star$·"name". Consequently, the tabular representation of $dp^{\mathcal{C}}$ would show the number 2 under the column referencing the string type and the row referencing "hobbies"·$\star$·"name". Additionally, since the array in each document contains a literal (i.e., values `"tennis"` and `"soccer"` respectively), the intersection between the column referencing the string type and the row referencing "hobbies"·$\star$would display the number 2.

By storing array information in this manner, for any path p, a DataPilot retains information about the number of nested elements observed across all documents where

p evaluates to an array, but not about the order of such nested elements or their distribution across documents. For instance, given the path "hobbies"$\cdot\star\cdot$"name"and the count of 2 for the string type, one can either infer that there are two documents in the data where the path `"hobbies"` points to an array, and each one has a nested document with the key `"name"`, or that there is one document where the path `"hobbies"` points to an empty array, and another document where the path `"hobbies"` points to an array with two nested objects, each having the key `"name"`.

A distinction is made between path expressions containing the label $\star$ and path expressions without it. Definitions 1.1 and 1.3 elucidate on this.

Definition 1.1 (Array path expression)**.** Let p be a path expression of length n. p is said to be an array path expression if there exists some $p_k = \star$ such that $p = p_1 \cdot p_2 \cdot \ldots \cdot p_{k-1} \cdot p_k \cdot p_{k+1} \cdot \ldots \cdot p_n$. That is, more succinctly, if $\star \in p$. □

Definition 1.2 (Set of array path expressions)**.** Let P be a set of path expressions. The set of array path expressions of P is $arrs(P) = \{p \in P \mid \star \in p\}$. □

Definition 1.3 (Non-array path expression)**.** A path expression p is said to be a non-array path expression if it is not an array path expression. That is, more succinctly, if $\star \notin p$. □

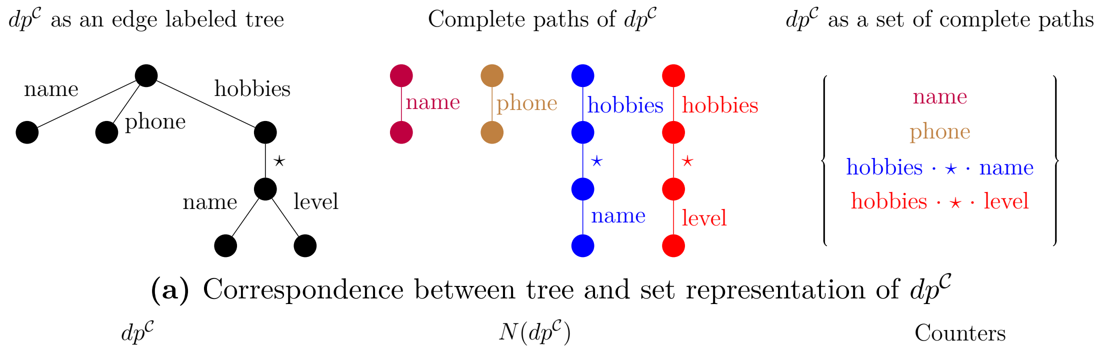
Definition 1.4 (Set of non-array path expressions)**.** Let P be a set of path expressions. The set of non-array path expressions of P is $noarrs(P) = \{p \in P \mid \star \notin p\}$. □

The primary purpose of a DataPilot is to establish a data structure that records the frequency counts of paths in $\mathcal{C}$. This structure facilitates the emulation of the behavior of operators and properties defined in Sections IV.2 and IV.3. The goal is to mimic the manipulation and transformation of documents in $\mathcal{C}$ so as to estimate the structure and frequency counts of documents within the collection resulting from the application of such operators to $\mathcal{C}$. The encoding of these two estimations is expounded upon in Definition 1.5, which is elaborated upon with an illustrative instance in Example 1.1.

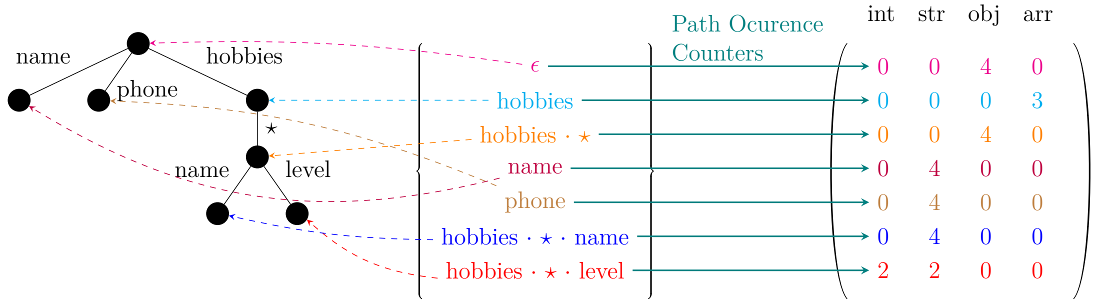
Definition 1.5 (DataPilot)**.** Let $\mathcal{C}$ be a collection, and p a path expression. A data pilot $dp^{\mathcal{C}}$ of $\mathcal{C}$ comprises an edge-labeled tree with a special label $\star$ not found in the alphabet of the labels from the JSON documents in $\mathcal{C}$ and the two mappings:

- **path occurrence counter**, denoted as $c_t(dp^{\mathcal{C}}, p) = x$, where x is a number from $\mathbb{N}^0$ that estimates the occurrences of p in $\mathcal{C}$ with type t.
- **empty array counter**, denoted as $\varepsilon(dp^{\mathcal{C}}, p) = x$, where x is a number from $\mathbb{N}^0$ that estimates the number of documents where p resolves to an empty array in $\mathcal{C}$.

□



(a) Correspondence between tree and set representation of $dp^{\mathcal{C}}$



(b) Path ocurrence counters of $dp^{\mathcal{C}}$ in a system with 4 types: int, str, obj, arr

Figure 37. Example of a DataPilot $dp^{\mathcal{C}}$ of a collection with 4 documents on a DB system with 4 types

Example 1.1. Assume a collection $\mathcal{C}$ with the JSON Documents in Listing V.2 on a DB system with the following 4 types: integer (int), string (str), object (obj) and array (arr).

```
{"name":"Jim","phone":"502-225-1111","hobbies":[
   {"name":"snorkel","level":5},{"name":"soccer","level":"amateur"}
        ]}
{"name":"Tom","phone":"502-335-1321","hobbies":[
   {"name":"swimming","level":3},{"name":"german","level":"C1"} ]}
{"name":"Carl","phone":"502-335-1321", "hobbies": []}
{"name":"Sam","phone":"502-335-1321"}
```

Listing V.2. Documents in collection $\mathcal{C}$

Since $dp^{\mathcal{C}}$ is an edge-labeled tree, it can be conceptualized as a set of complete paths, where each complete path references a leaf of the tree, as illustrated in Figure 37a. The recursive generating tree definition outlined in Section IV.1 is then employed to construct the tree based on its leaves. Similarly, each path in the set $N(dp^{\mathcal{C}})$ corresponds to a node in the tree. Figure 37b depicts this relationship and how each path maps to a vector of counters. For instance, the path p = hobbies·*·level is mapped to the vector $(2, 2, 0, 0)$. This indicates that there are 2 documents where p resolves to an integer ($c^p_{\text{int}} = 2$), such as {["name":"snorkel","level":5]} and {["name":"swimming","level":3]} from Listing V.2. Additionally, there are 2 documents where p resolves to a string ($c^p_{\text{str}} = 2$), such as {["name":"soccer","level":"amateur"]} and {["name":"german","level":"C1"]} from Listing V.2. □

In the context of a DataPilot, similar to the relational case, the term *collection cardinality* refers to an estimate of the number of elements in a collection. This estimate is structurally stored at the root node of the DataPilot, as explained in Definition 1.6.

Definition 1.6 (Estimated collection cardinality)**.** Let $\mathcal{C}$ a collection and $dp^{\mathcal{C}}$ a DataPilot of $\mathcal{C}$. The estimated number of documents within $dp^{\mathcal{C}}$, also called the cardinality of $\mathcal{C}$, corresponds to $Card(dp^{\mathcal{C}}, \epsilon) = c_{obj}(dp^{\mathcal{C}}, \epsilon)$. □

Additionally, various types of cardinalities are defined. The estimated frequency counts, or simply *counters*, serve to estimate the number of documents in a collection that possess a specific path, irrespective of its type, or the total number of elements observed within all arrays a path points to. Both cases involve adding up all counters, a concept termed *path cardinality.* However, they differ in the type of path used. The former case exclusively utilizes non-array paths, leading to *non-array path cardinality* (Definition 1.8), while the latter case employs only array paths, resulting in *array cardinality* (Definition 1.9) and *array-path cardinality* (Definition 1.10).

Definition 1.7 (Estimated path cardinality)**.** Let p be a path, $\mathcal{C}$ a collection and $dp^{\mathcal{C}}$ a DataPilot of $\mathcal{C}$. The estimated cardinality of p based on $dp^{\mathcal{C}}$ is defined as:

$$Card(dp^{\mathcal{C}}, p) = \sum_{t \in A} c_t(dp^{\mathcal{C}}, p)$$

□

Definition 1.8 (Estimated Non-array-path Cardinality)**.** For a non-array path p, a collection $\mathcal{C}$, and a DataPilot $dp^{\mathcal{C}}$ of $\mathcal{C}$. The cardinality $Card(dp^{\mathcal{C}}, p)$ represents the occurrences of p in the documents within $\mathcal{C}$. □

Definition 1.9 (Estimated array cardinality)**.** For a path p of length n with $p_n = \star$, $\mathcal{C}$ a collection, and $dp^{\mathcal{C}}$ a DataPilot of $\mathcal{C}$, the cardinality $Card(dp^{\mathcal{C}}, p)$ represents the cumulative number of elements within the array $p_1 \cdot \ldots \cdot p_{n-1}$ points to in documents from $\mathcal{C}$. Each counter $c_t(dp^{\mathcal{C}}, p)$ estimates the number of elements with type t across all arrays resolved by $p_1 \cdot \ldots \cdot p_{n-1}$ in $\mathcal{C}$. Similarly, $\varepsilon(dp^{\mathcal{C}}, p)$ estimates the cumulative number of (nested) empty arrays within the arrays resolved by $p_1 \cdot \ldots \cdot p_{k-1}$. □

Definition 1.10 (Estimated array-path cardinality)**.** For an array path $p = p_1, \cdot \ldots \cdot p_{k-1} \cdot p_k \cdot p_{k+1} \cdot \ldots \cdot p_n$ of length n with $p_k = \star$ and $p_n \neq \star$, a collection $\mathcal{C}$, and a DataPilot $dp^{\mathcal{C}}$ of $\mathcal{C}$. The cardinality $Card(dp^{\mathcal{C}}, p)$ represents the occurrences of the path $p_{k+1} \cdot \ldots \cdot p_n$ within the documents nested in the array that $p_1, \cdot \ldots \cdot p_{k-1}$ points to. □

For a DataPilot to consistently emulate the behavior of collections, it must adhere to fundamental properties that characterize collections. One such property regards collection cardinality, which, in terms of the data, represents the total number of documents in the collection. Consequently, if the counter of a non-array path in the DataPilot surpasses the estimated collection cardinality stored in the DataPilot's root node, it indicates an inconsistency. This is because a non-array path cannot be present in more documents than the total number of documents in a collection.

However, this constraint does not bind array paths in the same way. Their occurrence is contingent upon the total number of elements within arrays across all documents. In the case of array paths referring to a document nested within an array, nested into another array, their occurrence is contingent upon the total number of elements within the innermost array

These constraints imply that the cardinalities of non-array paths are linked to the DataPilot's root node, identified by path ϵ, while the cardinalities of array paths are tied to DataPilot nodes identified by paths whose last label is $\star$. Thus, certain prefixes of an array path are designated as *relative roots* of such path, and among these, there is an *immediate relative root*. These concepts are detailed in Definitions 1.11 and 1.12.

Definition 1.11 (Immediate relative root of a path expression)**.** For a path expression $p = p_1 \cdot p_2 \cdot \ldots \cdot p_n$ the immediate relative root of p, denoted as $relroot(p)$, is ϵ if p is a non-array path, else if p is an array path whose last label differs from $\star$ and there exists two paths q and r such that $p = q \cdot \star \cdot r$ and $\star \notin r$, then $relroot(p) = q \cdot \star$. Succinctly,

$$relroot(p) = \begin{cases} p_1 \cdot \ldots \cdot p_k & \text{if } (\exists p_k \in p)\, (p_k = \star \wedge \star \notin p_{k+1} \cdot \ldots \cdot p_n) \\ \epsilon & \text{otherwise} \end{cases}$$

□

Definition 1.12 (Relative roots of a path expression)**.** For a path expression p, the

relative roots of p are given by the expression below:

$$relroots(p) = \begin{cases} \{relroot(p)\} \cup relroots(q_1 \cdot \ldots \cdot q_{|q|-1}) & \text{if } q = relroot(p) \wedge q \neq \epsilon \\ \emptyset & \text{otherwise} \end{cases}$$

□

These fundamental properties, essential for a DataPilot to emulate a collection, are elucidated in Invariants 1.1, 1.2, and 1.3.

Invariant 1.1. In a DataPilot $dp^{\mathcal{C}}$ for a collection $\mathcal{C}$, the counters for the empty path ϵ (root of the tree) represent the number of elements in $\mathcal{C}$. Since collections can only hold JSON documents, the inferred invariant is: $c_t(dp^{\mathcal{C}}, \epsilon) = 0$ for all types $t \in \mathfrak{T} - \{obj\}$. Note that this also implies $\varepsilon(dp^{\mathcal{C}}, \epsilon) = 0$. □

Invariant 1.2. In a DataPilot $dp^{\mathcal{C}}$ for a collection $\mathcal{C}$, the ocurrence of non-array paths must be less than or equal to the total number of documents. Likewise, the ocurrence of array paths whose last label differs from $\star$ must be less than or equal to the total number of elements estimated to be in the array containing them. Thus, the inferred invariant is: $Card(dp^{\mathcal{C}}, p) \leq Card(dp^{\mathcal{C}}, relroot(p))$ for any path $p \in N(dp^{\mathcal{C}})$. □

Corollary 1.13. Let $\mathcal{C}$ be a collection and $dp^{\mathcal{C}}$ a DataPilot of $\mathcal{C}$. Then, Invariant 1.2 implies $Card(dp^{\mathcal{C}}, p) \leq Card(dp^{\mathcal{C}}, \epsilon)$ for any path $p \in noarrs(N(dp^{\mathcal{C}}))$. □

Corollary 1.14. Let $\mathcal{C}$ be a collection and $dp^{\mathcal{C}}$ a DataPilot of $\mathcal{C}$. Then, Invariant 1.2 implies $Card(dp^{\mathcal{C}}, p) \leq Card(dp^{\mathcal{C}}, relroot(p))$ for any path $p \in arrs(N(dp^{\mathcal{C}}))$ □

Definition 1.15 (Set of explicit path expressions)**.** Let P be a set of path expressions. A set of explicit path expressions of P is a subset where none of the paths within are considered as relative roots. Formally,

$$explicit(P) = P - \bigcup_{p \in P} relroots(p)$$

□

Definition 1.16 (Estimated number of non-empty documents)**.** Let $dp^{\mathcal{C}}$ be a DataPilot of collection $\mathcal{C}$. The estimated number of non-empty documents in $\mathcal{C}$ can be computed as follows:

$$\text{NumNED}\Big(dp^{\mathcal{C}}\Big) = \max_{p \in noarrs(explicit(N(dp^{\mathcal{C}})))} \Big(Card(dp^{\mathcal{C}}, p)\Big)$$

□

Definition 1.17 (Estimated Number of Empty Documents)**.** Let $dp^{\mathcal{C}}$ be a DataPilot of collection $\mathcal{C}$. The estimated number of empty documents in $\mathcal{C}$ can be computed as follows:

$$\text{NumED}\Big(dp^{\mathcal{C}}\Big) = Card(dp^{\mathcal{C}}, \epsilon) - \text{NumNED}\Big(dp^{\mathcal{C}}\Big)$$

□

Invariant 1.3. In a DataPilot $dp^{\mathcal{C}}$ for a collection $\mathcal{C}$, the presence of empty documents in $\mathcal{C}$ implies that the estimated collection cardinality must be greater than or equal to the maximum path cardinality of non-array paths in $explicit\Big(N\Big(dp^{\mathcal{C}}\Big)\Big)$. Thus, the inferred invariant is: $Card(dp^{\mathcal{C}}, \epsilon) \geq \text{NumNED}\Big(dp^{\mathcal{C}}\Big)$. □

DataPilots leverage *path estimation* (Definition 1.18) trough the *back* and *forth* properties (Definitions 1.19 and 1.20 respectively) to encapsulate document structure. Path estimation involves establishing an approximation to the existence of a path in a collection based on the existance of a path in said collection's DataPilot. For example, paths a·1·c and a·2·c are both estimated by a·⋆·c. A DataPilot has the back property if all its paths estimate a path in the cover of its collection, and it has the forth property if all paths in its collection's cover are estimated by a path in the DataPilot.

The forth property guarantees the presence of every path in a DataPilot in at least one document from its collection, while the back property ensures that all paths from documents in its collection are included in the DataPilot. A DataPilot is required to uphold the forth property, but when it adheres to both, it is labeled as *faithful* to its collection (Definition 1.21). Conversely, if a DataPilot lacks both the back

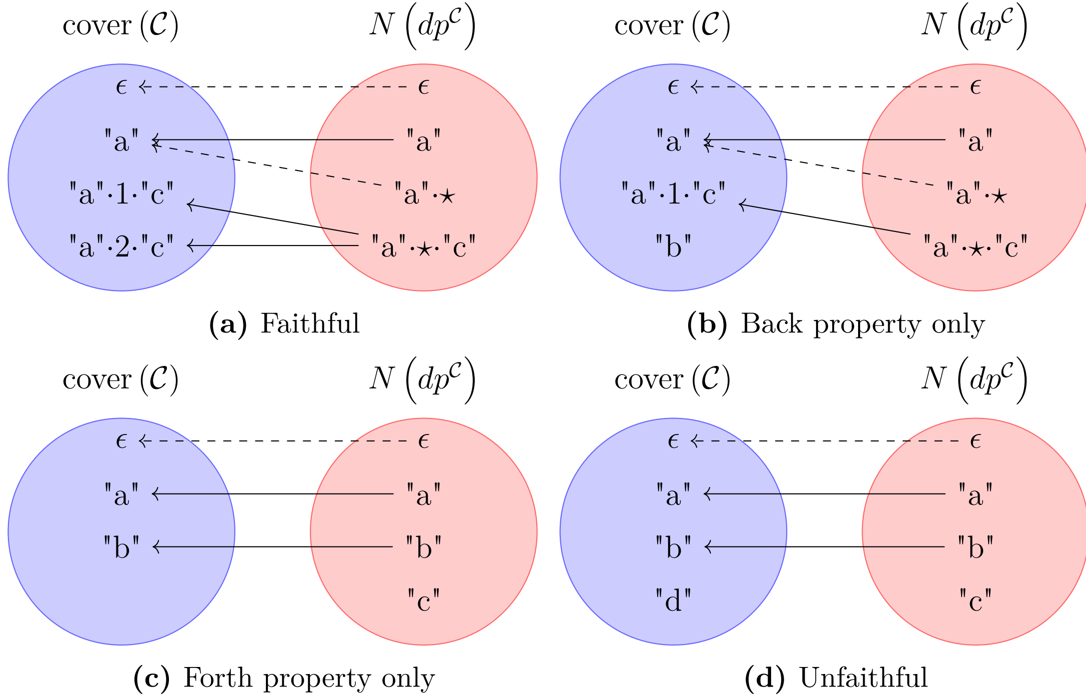


Figure 38. Example of DataPilots categorized by adherence to the Back and Forth properties. Paths indicated by arrows are estimated by paths at the arrow sources. Dashed arrows originate from relative roots.

and the forth property, it is classified as *unfaithful* to its collection (Definition 1.22). Figure 38 shows examples of DataPilots categorized by adherence to the Back and Forth properties.

Definition 1.18 (Structural path estimation)**.** A path $p = p_1 \cdot p_2 \cdot \ldots \cdot p_n$ in $dp^{\mathcal{C}}$ is said to structurally estimate (or just estimate when context allows) a path $q = q_1 \cdot q_2 \cdot \ldots \cdot q_n$ in cover $(\mathcal{C})$ if $p_i = q_i$ except that p has label $\star$ in terms where q has labels from $\mathbb{N}^0$. □

Definition 1.19 (Back property)**.** A DataPilot $dp^{\mathcal{C}}$ of collection $\mathcal{C}$ has the back property if for each path p in $N(dp^{\mathcal{C}})$, there exists at least one document d in $\mathcal{C}$ with a path q such that p estimates q. □

Definition 1.20 (Forth property)**.** A DataPilot $dp^{\mathcal{C}}$ of collection $\mathcal{C}$ has the forth property if for each path p in a document d in $\mathcal{C}$, there exists a path q in $N(dp^{\mathcal{C}})$ such that q estimates p. □

Definition 1.21 (Faithful DataPilot)**.** A DataPilot $dp^{\mathcal{C}}$ of collection $\mathcal{C}$ is said to be faithful to $\mathcal{C}$ if it has both the back and forth properties. □

Definition 1.22 (Unfaithful DataPilot)**.** A DataPilot $dp^{\mathcal{C}}$ of collection $\mathcal{C}$ is deemed unfaithful to $\mathcal{C}$ if it lacks both the back and forth properties. □

Definition 1.23 (Overlap)**.** Given two paths: p, q, an overlap $Overlap^{\mathcal{C}}(p,q)$ is a subset of $\mathcal{C}$ based on the existence of p and q; for any document *doc* in a $\mathcal{C}$, if p and q exist in *doc*, then *doc* is part of $Overlap^{\mathcal{C}}(p,q)$.

Documents not in $Overlap^{\mathcal{C}}(p,q)$ belong to the subset $Overlap^{\mathcal{C}}(p,q)^{\complement}$. Note that $Overlap^{\mathcal{C}}(p,q)^{\complement}$ acts as the *(set) complement* of $Overlap^{\mathcal{C}}(p,q)$: $\mathcal{C} = Overlap^{\mathcal{C}}(p,q) \cup Overlap^{\mathcal{C}}(p,q)^{\complement}$. The number of documents in $Overlap^{\mathcal{C}}(p,q)$, or $|Overlap^{\mathcal{C}}(p,q)|$, cannot be accurately inferred from the DP, but represented as an interval on a 2-case basis:

- **Case 1:** $Card(dp^{\mathcal{C}},p) + Card(dp^{\mathcal{C}},q) > n$, then

$$Card(dp^{\mathcal{C}},p) + Card(dp^{\mathcal{C}},q) - n \leq |Overlap^{\mathcal{C}}(p,q)| \leq \min\Big(Card(dp^{\mathcal{C}},p), Card(dp^{\mathcal{C}},q)\Big)$$

- **Case 2:** $Card(dp^{\mathcal{C}},p) + Card(dp^{\mathcal{C}},q) \leq n$, then
  $0 \leq |Overlap^{\mathcal{C}}(p,q)| \leq \min\Big(Card(dp^{\mathcal{C}},p), Card(dp^{\mathcal{C}},q)\Big)$

Both intervals can be collapsed into

$$\max\Big(0, Card(dp^{\mathcal{C}},p) + Card(dp^{\mathcal{C}},q) - n\Big) \leq |Overlap^{\mathcal{C}}(p,q)| \leq \min\Big(Card(dp^{\mathcal{C}},p), Card(dp^{\mathcal{C}},q)\Big)$$

2 Data Pilot creation and maintenance

When implementing a Data Pilot, its tree structure and counters are decoupled. This is achieved by storing paths and counters in separate data structures. At a more detailed level, this decoupling involves two main components: the *counter store*, a data structure holding vectors of counters per path; for instance, a matrix where columns represent types, and rows are assigned to paths; and the *path store*, a key-value container, such as a trie or a hash table, to manage paths and connect them with their corresponding vectors of counters in the counter store. This separation offers flexibility, allowing specialization based on space and time complexity requirements. Furthermore, it facilitates the application of Data Pilot algebra operators, as transformations can be separated based on their impact on tree structure and counters.

To illustrate the benefits of this decoupling, consider two scenarios:

1. **High Homogeneity of Types:** Assume the path store is to be implemented as a matrix of counters. In cases where a collection exhibits high homogeneity of types per path in its cover, resulting in a matrix with numerous zero counters, a sparse representation may be more efficient. Implementation options include using a 2D array with rows indicating the indices of non-zero elements, columns indicating the indices of non-zero columns, and the values of non-zero counters. Other alternatives involve linked lists or hash tables. At parsing, the matrix could be treated as sparse and transformed into a dense representation if its size exceeds a fixed threshold.

2. **Bulk Document Parsing:** When parsing a bulk of documents, each one in a depth-first manner, representing the path store as a trie can outperform its implementation as a hash table in insertion and lookup speed. If the trie keeps track of the node where a label was last inserted, it reduces the overhead of repeated concatenations and hashing. In contrast, a hash table requires

multiple concatenations and hash computations for each child in a depth-first traversal.

These cases are independent choices, enabling the implementation of the path store as a trie or hash table without affecting the implementation of the counter store as a sparse or dense matrix and vice versa.

Pseudocode 1: Example implementation of a DataPilot's counter store

```
1  struct {
2  |   uint nPaths;
3  |   uint nTypes;
4  |   uint[ ] data;
5  } CMatrix;
6  Function InitCounters(CMatrix counters, uint nPaths, uint nTypes):
7  |   counters.nPaths ← nPaths;
8  |   counters.nTypes ← nTypes;
9  |   counters.nInsertedPaths ← 0;
10 |   counters.data ← [0, 0, ..., 0]   (counters.nPaths ∗ counters.nTypes)
11 end
12 Function ResizeCounters(CMatrix counters, uint nPaths):
13 |   CMatrix newCounters;
14 |   InitCounters(newCounters, counters.nPaths + nPaths, counters.nTypes );
15 |   idx ← (counters.nPaths ∗ counters.nTypes) − 1;
16 |   for uint i=0 to idx do
17 |   |   newCounters.data[i] ← counters.data[i];
18 |   end
19 |   counters ← newCounters;
20 end
21 Function CounterInc(CMatrix counters, uint pathIdx, uint typeIdx):
22 |   if pathIdx = counters.nPaths then
23 |   |   ResizeCounters(counters, counters.nPaths )
24 |   end
25 |   uint idx ← typeIdx + (pathIdx ∗ counters.nTypes);
26 |   counters.data[idx ] ← counters.data[idx ] + 1;
27 end
```

In Scenario 1, the choice between a sparse or dense matrix representation for the counter store involves various considerations, encompassing ease of implementation, size, lookup time, and expansion.

From a complexity standpoint, consider the space complexity of the counter store within a Data Pilot $dp^{\mathcal{C}}$ of a collection $\mathcal{C}$. Represented as a dense matrix, the counter store exhibits a space complexity of $O(|N(dp^{\mathcal{C}})| * |\mathfrak{T}|)$. It is crucial to note that each path p in cover $(\mathcal{C})$ resolving to an array is estimated by two paths in $N(dp^{\mathcal{C}})$—namely, p and $p{\cdot}\star$. Proper Data Pilots may harbor more paths than cover $(\mathcal{C})$ (excluding

relative roots), adhering strictly to the forth property. Consequently, for disjoint subsets S_1 and S_2 of $\text{cover}(\mathcal{C})$, where all paths resolving to an array are in S_1 and those not resolving to an array are in S_2, the space complexity can be expressed as $\Theta(|\mathfrak{T}| * (2 * |S_1| + |S_2|))$, or more simply, $\Theta(|\mathfrak{T}| * |\text{cover}(\mathcal{C})|)$. In this scenario, space consumption scales linearly with both the number of paths and types.

Contrastingly, a sparse matrix, utilizing the 2D array representation, incurs space in the order of $\Theta(3 * m)$, where m denotes the count of non-zero counters, or more succinctly, $\Theta(m)$. Thus, space consumption grows proportionally with the number of non-zero counters. In instances where the collection is entirely homogeneous, including the content types of arrays, there can only be one non-zero counter per path in the Data Pilot. However, if heterogeneity exists, some paths may boast more than one non-zero counter. Consequently, the more homogeneous a collection, the sparser the counter matrix becomes.

Pseudocode 2: Example implementation of Data Pilot

```
1  struct {
2      Trie paths;
3      CMatrix counters;
4      uint nInsertedPaths;
5  } DataPilot;
6  Function InitDataPilot(DataPilot dp, uint nPaths, uint nTypes):
7      InitEmptyTrie(dp.paths );
8      InitCounters(dp.counters, nPaths, nTypes );
9      dp.nInsertedPaths ← 0;
10 end
11 Function DataPilotAdd(DataPilot dp, Path path, uint type):
12     // assume function TrieSearchAndRetrieve returns a struct
13     // with members "found" and "value" of types boolean and uint
14     res ← TrieSearchAndRetrieve(dp.paths, path );
15     pathIdx ← res.value;
16     if res.found = false then
17         pathIdx ← dp.nInsertedPaths;
18         TrieAdd(dp.paths, path, pathIdx);
19         dp.nInsertedPaths ← dp.nInsertedPaths + 1;
20     end
21     CounterInc(dp.counters, pathIdx, type );
22 end
```

Opting for a dense matrix offers simplicity in implementation and interaction, as depicted in Pseudocode 1. Counter incrementation entails basic arithmetic operations to locate the counter in the matrix and execute the increment itself. However, its

memory footprint is significantly higher the more homogeneus a collection is. For instance, assuming counters are stored as unsigned integers occupying 4 bytes each, and considering the 8 types in the BSON specification (i.e., byte, int32, int64, uint64, double, decimal128, array, and object), a row in the dense matrix necessitates $4 \times 8 = 32$ bytes. Consequently, to accommodate $\left\lfloor \frac{1024}{32} \right\rfloor = 32$ paths, 1KiB is required, and for $\left\lfloor \frac{1049000}{32} \right\rfloor = 32781$ paths, 1MiB is necessary. In a collection with homogeneous path types, a sparse matrix, utilizing a 2D array representation, needs only $4 \times 3 = 12$ bytes per path, as there is only one non-zero counter per path. Therefore, the sparse matrix stores the path (row) index, type (column) index, and the non-zero counter value, all as unsigned integers, requiring $12 \times 32 = 384$ bytes to store the same 32 paths as the dense matrix. This results in a reduction of memory consumption by $100\% \times \left(\frac{1024-384}{1024}\right) = 62.5\%$.

In a collection with heterogeneous path types, the sparse matrix would demand $3 \times 4 \times m_i = 12m_i$ bytes per path, where m_i is the number of non-zero counters for path i. In such cases, it may be a less favorable choice than the dense matrix.

In scenario 2, choosing a suitable container data structure, such as a hash table or trie, for the tree representation of the Data Pilot involves careful consideration. Taking the tree representation into account, consider a path p that resolves to an object with k children.

If a hash table is used, p must be retained in memory. When the parser iterator reaches child i of the object, l_i, the label resolving to that child, is concatenated with p. Subsequently, path $p{\cdot}l_i$ is hashed and inserted into the hash table. This process involves k concatenations and k hash computations.

In contrast, employing a trie that keeps track of the last insertion involves retaining the iterator to the last inserted node instead of maintaining p. If the iterator is, for instance, a pointer, only 8 bytes per trie iterator need to be stored. This allows for the straightforward insertion of label l_i into the trie as a new node without the need for any preceding search. It is worth noting that, while the size of p can vary and

Pseudocode 3: Example implementation of a Data Pilot's creation function using a recursive approach that traverses documents in a depth-first manner

```
 1 // iterator starts at opening bracket of the JSON array
 2 Function ProcessArr(DataPilot dp^C, Path path, Iterator it):
 3 |   key ← ⋆;
 4 |   depth ← IterDepth(it);
 5 |   // move iterator to first element of the array
 6 |   IterNext(it);
 7 |   do
 8 |   |   valueType ← IterType(it);
 9 |   |   DataPilotAdd(dp^C, path · key, valueType);
10 |   |   if valueType = obj then
11 |   |   |   ProcessDoc(dp^C, path · key, it);
12 |   |   else if valueType = arr then
13 |   |   |   ProcessArr(dp^C, path · key, it);
14 |   |   end
15 |   |   IterNext(it);
16 |   while IterDepth(it) < depth;
17 end
18 // iterator starts at opening brace of JSON object
19 Function ProcessDoc(DataPilot dp^C, Path path, Iterator it):
20 |   depth ← IterDepth(it);
21 |   // move iterator to first key of JSON object
22 |   IterNext(it);
23 |   do
24 |   |   key ← IterGet(it);
25 |   |   // move iterator to value
26 |   |   IterNext(it);
27 |   |   valueType ← IterType(it);
28 |   |   DataPilotAdd(dp^C, path · key, valueType);
29 |   |   if valueType = obj then
30 |   |   |   ProcessDoc(dp^C, path · key, it);
31 |   |   else if valueType = arr then
32 |   |   |   ProcessArr(dp^C, path · key, it);
33 |   |   end
34 |   |   // move iterator to next key
35 |   |   IterNext(it);
36 |   while IterDepth(it) < depth;
37 end
38 // dp^C must have already been initialized by calling the InitDataPilot function
39 Function CreateDataPilot(Collection C, DataPilot dp^C):
40 |   foreach document d of C do
41 |   |   it ← NewJsonIter(d);
42 |   |   // ϵ refers to the empty path
43 |   |   ProcessDoc(dp^C, ϵ, it);
44 |   end
45 end
```

might be less than 8 bytes (e.g., a·b·c takes 3 bytes, assuming each character occupies 1 byte), this scenario is unlikely in JSON datasets designed for human readability. Furthermore, even in such cases, p would still necessitate concatenation with children labels and subsequent hashing.

Pseudocode 2 shows an example of a Data Pilot implementation that exposes

functions `InitDataPilot` and `DataPilotAdd`, and interacts with the counter store implemented in Pseudocode 1 by using the function `CounterInc`. The Pseudocode assumes the existence of a path store implemented as a Trie, where keys are paths and values are unsigned integers that act as references linking a path to its corresponding row in the counter store that exposes functions `InitEmptyTrie` to initialize an empty counter store instance, `TrieSearchAndRetrieve` to find a path's reference to its vector of counters, and `TrieAdd` to add a path and its reference to the appropriate counter vector.

A Data Pilot can be developed concurrently with parsing and ingesting data into a collection or constructed from an existing collection. The flexibility to add paths later simplifies the process when new documents are inserted. In the context of parsing substantial datasets, an iterator over a JSON document should ideally refrain from retaining unnecessary data, freeing resources or marking them for potential release as the iterator progresses. This approach becomes particularly relevant for performance.

Pseudocode 3 illustrates the implementation of a function to construct a Data Pilot from a collection, adhering to the *move forward and forget* iterator approach. The pseudocode assumes the existence of an Iterator interface over JSON documents that facilitates a depth-first traversal. The function `NewJsonIter`, when provided with a document `d` as an argument, creates an iterator positioned at the opening brace of `d`. The `IterNext` function transitions the iterator from key to value if currently pointing to a key, from value to key if pointing to a value, and from array element to array element if currently positioned as such. During document processing, the iterator always begins at the opening brace (`{`) and, when handling an array, starts at the opening bracket (`[`). The iterator keeps track of the current depth, retrievable using the `IterDepth` function.

Theorem 2.1 (Creation of a faithful Data Pilot)**.** Given a collection $\mathcal{C}$ and an empty Data Pilot $dp^{\mathcal{C}}$, function `CreateDataPilot` from Pseudocode 3 populates $dp^{\mathcal{C}}$ so that it is faithful to $\mathcal{C}$.

3 Data Pilot algebra

In describing operations over a Data Pilot $dp^{\mathcal{C}}$, this section addresses two components independently: structural—how the tree structure is altered—and counting—how path occurrences are altered.

Each Data Pilot operator is homonymic to an operator from the algebra of documents; this intends to (1) outline a one-to-one correspondence between summary and data operations, which aids in the further implementation of a query planner, and (2) ease characterization of the estimates in collection structure and cardinality that a Data Pilot provides.

For any collection $\mathcal{C}$, the objective is for each Data Pilot operator to produce an approximation of the faithful Data Pilot obtained from the collection resulting from applying the corresponding document algebra operator to $\mathcal{C}$. In other words, for an arbitrary operation Ψ, a faithful Data Pilot $dp^{\mathcal{C}}$ of $\mathcal{C}$, a collection $\mathcal{C}' = \Psi(\mathcal{C})$, and a faithful Data Pilot $dp^{\mathcal{C}'}$ of $\mathcal{C}'$, the Data Pilot resulting from $\Psi\left(dp^{\mathcal{C}}\right)$ should approximate $dp^{\mathcal{C}'}$. The primary distinctions between these two Data Pilots are in their counters and structure. Structurally, $\Psi\left(dp^{\mathcal{C}}\right)$ maintains the forth property but forfeits the back property, making it non-faithful. Both differences arise because $dp^{\mathcal{C}'}$ is computed based on the data in $\mathcal{C}'$, while $\Psi\left(dp^{\mathcal{C}}\right)$ relies solely on the information in $dp^{\mathcal{C}}$.

To illustrate, consider $\mathcal{C}$ with documents $\mathsf{d}_1 =$ `{"name": "John Doe", "age": 20, "hobby": "soccer" }` and $\mathsf{d}_2 =$ `{"name": "Jane Doe", "age": 30 }`, and let Ψ be the selection of documents in $\mathcal{C}$ where "age"> 25. Therefore, $\mathcal{C}' = \sigma_{\text{age}>25}(\mathcal{C})$, and only document d_2 remains in $\mathcal{C}'$. This means that $dp^{\mathcal{C}'}$ lacks the path "hobby" since it is only present in d_1, absent from $\mathcal{C}'$. On the other hand, $\sigma_{\text{age}>25}\left(dp^{\mathcal{C}}\right)$ is only aware that path "age" evaluates to a number in the two documents present in $\mathcal{C}$ and that path "hobby" is present in one of those two documents. Because $dp^{\mathcal{C}}$ is unaware of the actual values in the data, Data Pilot $\sigma_{\text{age}>25}\left(dp^{\mathcal{C}}\right)$ provides an upper bound,

assuming that at most both documents in $\mathcal{C}$ are filtered by the selection predicate and, consequently, that path "hobby" remains as well.

The term *approximation*, loosely used in previous paragraphs, denotes the connection between a document operation Ψ applied to a collection with the data and a corresponding Data Pilot operation Ψ' applied to a Data Pilot, ensuring the commutativity of the following diagram:

$$\begin{array}{ccc} \mathcal{C} & \longrightarrow & dp^{\mathcal{C}} \\ {\scriptstyle\Psi}\big\downarrow & & \big\Downarrow{\scriptstyle\Psi'} \\ \Psi(\mathcal{C}) & \longrightarrow & dp^{\Psi(\mathcal{C})} \end{array}$$

The description of operators below starts then by outlining the behavior of the respective Data Pilot operation and how it affects $\mathcal{C}$, while describing the accurate (or *faithful*) changes in $dp^{\mathcal{C}}$ had perfect knowledge of structure and cardinality be available. Estimation of structure and cardinality are subsequently addressed in a separated manner.

Selection

The behavior of a selection has two implications on a data pilot: (1) path cardinalities either remain the same or decrease after a *select*, but can never increase; (2) *faithfulness* is not guaranteed (and oftentimes lost).

Structural estimation

A data pilot does not have access to the data; thus, branches in $dp^{\mathcal{C}}$ that should be absent in $\sigma_{\varphi}(dp^{\mathcal{C}})$ are unknown. Due to this uncertainty *select* refrains from deleting any path p (even if $Card(\sigma_{\varphi}(dp^{\mathcal{C}}), p) = 0$) to keep the forth property, but at the cost of losing the back property.

cardinality estimation

Suppose φ is an arbitrary select expression and exprpaths$\,(\varphi)$ the set of path expressions in φ. Cardinality estimates rely on the following two quantities:

- document retention factor. Denoted $drf(dp^{\mathcal{C}}, \varphi)$, this quantity estimates the percentage of documents to keep.

- type density factor. Denoted $tdf_b(dp^{\mathcal{C}}, \varphi, p)$, this quantity estimates the percentage of documents where path p is of type b out of all documents remaining where path p exists.

Both factors are calculated based on the type of φ.

In a Path-Path expression $p\theta q$, the operator θ is defined for a subset (often a proper subset) B of the types in $\mathfrak{T}$. For example, the operator $<$ is frequently defined for integers and strings in database systems, with the commonly accepted behaviors of integer and lexicographic orders, respectively. However, it is not clear how $<$ behaves for arrays or JSON objects as inputs.
As a result, documents where p and q are of any type in $\mathfrak{T} - B$ are filtered out from $\mathcal{C}$ based on this fact alone. Therefore, estimating the number of documents kept depends on the probability of p and q being of any type in B. Assuming that both events are independent, this probability is given by Equation 7.

$$\begin{aligned} drf(dp^{\mathcal{C}}, p\theta q) &= \left(\sum_{b \in B} \frac{c_b(dp^{\mathcal{C}}, p)}{Card(dp^{\mathcal{C}}, \epsilon)}\right) \times \left(\sum_{b \in B} \frac{c_b(dp^{\mathcal{C}}, q)}{Card(dp^{\mathcal{C}}, \epsilon)}\right) \\ &= \frac{1}{Card(dp^{\mathcal{C}}, \epsilon)^2} \times \left(\sum_{b \in B} c_b(dp^{\mathcal{C}}, p)\right) \times \left(\sum_{b \in B} c_b(dp^{\mathcal{C}}, q)\right) \end{aligned} \tag{7}$$

Likewise, the type density factor for any type of p and q given θ depends on B as shown in Equation 8. Take operator $<$ for example, if p is of type object in 5 documents, integer in 10 and string in 5, then all 5 documents where p is an object are not retained in $\mathcal{C}$ because obj $\notin B$ for $<$. Thus, the percentage of documents

where p is an object, integer and string after *select* are 0, $10/(10+5)$ and $5/(10+5)$ respectively.

$$tdf_a(dp^{\mathcal{C}}, p\theta q, s) = \begin{cases} 0 & \text{if } a \notin B \\ \dfrac{c_a(dp^{\mathcal{C}}, s)}{\sum_{b \in B} c_b(dp^{\mathcal{C}}, s)} & \text{if } s \in \text{exprpaths}\,(p\theta q) \\ \dfrac{c_a(dp^{\mathcal{C}}, s)}{Card(dp^{\mathcal{C}}, \epsilon)} & \text{if } s \notin \text{exprpaths}\,(p\theta q) \end{cases} \tag{8}$$

In a Path-Const expression $p\theta d$, one of the operands of θ (in this case, a constant d) has a single type t, unlike in a Path-Path expression. Therefore, estimating the number of documents kept depends on the probability of p and d being of the same type, as shown in Equation 9.

$$drf(dp^{\mathcal{C}}, p\theta d) = \frac{c_t(dp^{\mathcal{C}}, p)}{Card(dp^{\mathcal{C}}, \epsilon)} \tag{9}$$

The type density factor of path p with type t is 1 and 0 with any other type because documents where p is of a type other than t are not retained. For any path other than p, the density is computed based on the collection's estimated cardinality. This is shown in Equation 10.

$$tdf_a(dp^{\mathcal{C}}, p\theta d, s) = \begin{cases} 1 & \text{if } s = p \text{ and } a = t \\ \dfrac{c_a(dp^{\mathcal{C}}, s)}{Card(dp^{\mathcal{C}}, \epsilon)} & \text{if } s \neq p \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

In an existence expression $\exists p$, an estimate of the number of documents kept relies on the probability of a document having path p. This probability is given by Equation 11.

$$drf(dp^{\mathcal{C}}, \exists p) = \frac{Card(dp^{\mathcal{C}}, p)}{Card(dp^{\mathcal{C}}, \epsilon)} \tag{11}$$

All documents where p exists are kept and thus the percentage of documents where p is of any type $a \in \mathfrak{T}$ remains the same as in $dp^{\mathcal{C}}$. This is shown in Equation 12.

$$tdf_a(dp^{\mathcal{C}}, \exists p, s) = \frac{c_a(dp^{\mathcal{C}}, s)}{Card(dp^{\mathcal{C}}, \epsilon)} \tag{12}$$

In a grouped expression (α), the probability of finding a document in $\mathcal{C}$ that matches a (group of) expression(s) wrapped in parentheses is equal to the probability of α itself as shown in Equation 13. Similarly, any type density associated with (α) is also associated with α as shown in Equation 14.

$$drf(dp^{\mathcal{C}}, (\alpha)) = drf(dp^{\mathcal{C}}, \alpha) \tag{13}$$

$$tdf_a(dp^{\mathcal{C}}, (\alpha), s) = tdf_a(dp^{\mathcal{C}}, \alpha, s) \tag{14}$$

In a conjunctive expression $\alpha \wedge \beta$, the probability of finding a document in $\mathcal{C}$ that matches both α and β is the product of the probabilities of finding a document that matches each expression independently as given by Equation 15. Conversely, type density depends exclusively on the path expressions explicitely involved in α and β as shown in Equation 16.

$$drf(dp^{\mathcal{C}}, \alpha \wedge \beta) = drf(dp^{\mathcal{C}}, \alpha) \times drf(dp^{\mathcal{C}}, \beta) \tag{15}$$

$$tdf_a(dp^{\mathcal{C}}, \alpha \wedge \beta, s) = \begin{cases} tdf_a(dp^{\mathcal{C}}, \alpha, s) & \text{if } s \in \text{exprpaths}(\alpha) - \text{exprpaths}(\beta) \\ tdf_a(dp^{\mathcal{C}}, \beta, s) & \text{if } s \in \text{exprpaths}(\beta) - \text{exprpaths}(\alpha) \\ tdf_a(dp^{\mathcal{C}}, \alpha, s) \times tdf_a(dp^{\mathcal{C}}, \beta, s) & \text{if } s \in \text{exprpaths}(\alpha) \cap \text{exprpaths}(\beta) \end{cases} \tag{16}$$

In a disjunctive expression $\alpha \vee \beta$, the probability of finding a document in $\mathcal{C}$ that matches either α or β is calculated as the sum of the probabilities of finding a document that matches each expression independently, minus the probability of finding a document that matches both expressions. This calculation assumes that α and β are not mutually exclusive as show in Equations 17 and 18.

$$drf(dp^{\mathcal{C}}, \alpha \vee \beta) = drf(dp^{\mathcal{C}}, \alpha) + drf(dp^{\mathcal{C}}, \beta) - drf(dp^{\mathcal{C}}, \alpha \wedge \beta) \tag{17}$$

$$tdf_a(dp^{\mathcal{C}}, \alpha \vee \beta, s) = \begin{cases} tdf_a(dp^{\mathcal{C}}, \alpha, s) & \text{if } s \in \text{exprpaths}(\alpha) - \text{exprpaths}(\beta) \\ tdf_a(dp^{\mathcal{C}}, \beta, s) & \text{if } s \in \text{exprpaths}(\beta) - \text{exprpaths}(\alpha) \\ \begin{array}{l} tdf_a(dp^{\mathcal{C}}, \alpha, s) + tdf_a(dp^{\mathcal{C}}, \beta, s) \\ -tdf_a(dp^{\mathcal{C}}, \alpha \wedge \beta, s) \end{array} & \text{if } s \in \text{exprpaths}(\alpha) \cap \text{exprpaths}(\beta) \end{cases} \tag{18}$$

Estimating counters of $\sigma_{\varphi}(dp^{\mathcal{C}})$ is a two-step process:

Step 1 estimate the number of documents kept after a *select*, given by Equation 19.

$$Card(\sigma_{\varphi}(dp^{\mathcal{C}}), \epsilon) = \left\lfloor Card(dp^{\mathcal{C}}, \epsilon) \times drf(dp^{\mathcal{C}}, \varphi) \right\rfloor \tag{19}$$

When φ belongs to the simple expression kind, Equation 19 can be transformed into a more descriptive form given by Equation 20.

$$Card(\sigma_{\varphi}(dp^{\mathcal{C}}), \epsilon) = \begin{cases} \left\lfloor \frac{1}{Card(dp^{\mathcal{C}}, \epsilon)} \times \left(\sum_{b \in B} c_b(dp^{\mathcal{C}}, p) \right) \times \left(\sum_{b \in B} c_b(dp^{\mathcal{C}}, q) \right) \right\rfloor & \varphi = p\theta q \\ \left\lfloor c_t(dp^{\mathcal{C}}, p) \right\rfloor & \varphi = p\theta d \\ \left\lfloor Card(dp^{\mathcal{C}}, p) \right\rfloor & \varphi = \exists p \end{cases} \tag{20}$$

Step 2 estimate the counters of paths in $\sigma_{\varphi}(dp^{\mathcal{C}})$. Let s be a path such that $s \overset{r}{\leadsto} \sigma_{\varphi}(dp^{\mathcal{C}})$, then the counter of s for any type $a \in \mathfrak{T}$ is given by Equation 21.

$$c_a(\sigma_{\varphi}(dp^{\mathcal{C}}), s) = \left\lceil Card(\sigma_{\varphi}(dp^{\mathcal{C}}), \epsilon) \times tdf_a(dp^{\mathcal{C}}, \varphi, s) \right\rceil \tag{21}$$

Example 3.1 (Path-Path comparison using an operator defined for a single type)**.** Assume the data pilot $dp^{\mathcal{C}} = \{$"a","b","d","e"$\}$ with counters:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"a"}) = 25$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"b"}) = 30$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"a"}) = 15$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"b"}) = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = 5$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = 6$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = 10$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = 8$

Let θ be a comparison operator defined in type *int* only. Then, the counters for data pilot $dp^{\mathcal{C}'} = \sigma_{a\theta b}(dp^{\mathcal{C}})$ are computed as follows:

$$Card(dp^{\mathcal{C}'}, \epsilon) = \left\lfloor \frac{1}{40} \times 25 \times 30 \right\rfloor = 18$$

Counters of paths directly involved in the condition:

- $c_{\text{int}}(dp^{\mathcal{C}'}, \text{"a"}) = \left\lceil 18 \times \frac{25}{25} \right\rfloor = 18$
- $c_{\text{int}}(dp^{\mathcal{C}'}, \text{"b"}) = \left\lceil 18 \times \frac{30}{30} \right\rfloor = 18$
- $c_{\text{str}}(dp^{\mathcal{C}'}, \text{"a"}) = 0$
- $c_{\text{str}}(dp^{\mathcal{C}'}, \text{"b"}) = 0$

Other counters:

- $c_{\text{int}}(dp^{\mathcal{C}'}, \text{"d"}) = \left\lceil 18 \times \frac{5}{40} \right\rfloor = 2$
- $c_{\text{int}}(dp^{\mathcal{C}'}, \text{"e"}) = \left\lceil 18 \times \frac{6}{40} \right\rfloor = 3$
- $c_{\text{str}}(dp^{\mathcal{C}'}, \text{"d"}) = \left\lceil 18 \times \frac{10}{40} \right\rfloor = 5$
- $c_{\text{str}}(dp^{\mathcal{C}'}, \text{"e"}) = \left\lceil 18 \times \frac{8}{40} \right\rfloor = 4$

□

Example 3.2 (Path-Path comparison using an operator defined for multiple types)**.** Assume the same initial data pilot and counters from Example 3.1. Let α be a comparison operator defined for types *int* and *str*.Then, the counters for a data pilot $dp^{\mathcal{C}'} = \sigma_{d\alpha e}(dp^{\mathcal{C}})$ are computed as follows:

$$Card(\epsilon, dp^{\mathcal{C}'}) = \left\lfloor \frac{1}{40} \times (5 + 10) \times (6 + 8) \right\rfloor = 5$$

Counters of paths directly involved in the condition:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 5 \times \frac{5}{5+10} \right\rfloor = 2$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 5 \times \frac{6}{6+8} \right\rfloor = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 5 \times \frac{10}{5+10} \right\rfloor = 3$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 5 \times \frac{8}{6+8} \right\rfloor = 3$

Other counters:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"a"}) = \left\lceil 5 \times \frac{25}{40} \right\rfloor = 3$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"b"}) = \left\lceil 5 \times \frac{30}{40} \right\rfloor = 4$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"a"}) = \left\lceil 5 \times \frac{15}{40} \right\rfloor = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"b"}) = \left\lceil 5 \times \frac{0}{40} \right\rfloor = 0$

□

Properties

Property 3.1 (Commutativity of *select*)**.** Let φ and Ψ be *select* expressions. The composition of two selects over a data pilot is commutative:

$$\sigma_{\varphi}\left(\sigma_{\Psi}(dp^{\mathcal{C}})\right) = \sigma_{\Psi}\left(\sigma_{\varphi}(dp^{\mathcal{C}})\right)$$

□

Example 3.3 (Commutativity of *select*)**.** Assume the same initial data pilot and counters from Example 3.1. Let θ and α be comparison operators such that θ is defined for type *int* only and α for types *int* and *str*. The counters of data pilots $\sigma_{d\alpha e}\left(\sigma_{a\theta b}(dp^{\mathcal{C}})\right)$ and $\sigma_{a\theta b}\left(\sigma_{d\alpha e}(dp^{\mathcal{C}})\right)$ are computed as follows:

For $dp^{\mathcal{C}'} = \sigma_{d\alpha e}\left(\sigma_{a\theta b}(dp^{\mathcal{C}})\right)$, Example 3.1 shows how the counters change after $\sigma_{a\theta b}(dp^{\mathcal{C}})$. Using such information as a starting point, the new counters of the entire algebraic expressions are:

$$Card(\epsilon, dp^{\mathcal{C}'}) = \left\lfloor \frac{1}{18} \times (2+5) \times (3+4) \right\rfloor = 2$$

Counters of paths directly involved in the condition:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 2 \times \frac{2}{2+5} \right\rfloor = 1$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 2 \times \frac{3}{3+4} \right\rfloor = 1$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = \lceil 2 \times (5/(2+5)) \rfloor = 1$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 2 \times \frac{4}{3+4} \right\rfloor = 1$

Other counters:

- $c_{\text{int}}(dp^{\mathcal{C}}, "a") = \left\lceil 2 \times \frac{18}{18} \right\rceil = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, "a") = \left\lceil 2 \times \frac{0}{18} \right\rceil = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, "b") = \left\lceil 2 \times \frac{18}{18} \right\rceil = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, "b") = \left\lceil 2 \times \frac{0}{18} \right\rceil = 0$

For $dp^{\mathcal{C}'} = \sigma_{a\theta b}\left(\sigma_{d\alpha e}(dp^{\mathcal{C}})\right)$, Example 3.2 shows how the counters change after $\sigma_{d\alpha e}(dp^{\mathcal{C}})$. Using such information as a starting point, the new counters of the entire algebraic expressions are:

$$Card(\epsilon, dp^{\mathcal{C}'}) = \left\lfloor \frac{1}{5} \times 3 \times 4 \right\rfloor = 2$$

Counters of paths directly involved in the condition:

- $c_{\text{int}}(dp^{\mathcal{C}}, "a") = \left\lceil 2 \times \frac{3}{3} \right\rceil = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, "a") = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, "b") = \left\lceil 2 \times \frac{4}{4} \right\rceil = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, "b") = 0$

Other counters:

- $c_{\text{int}}(dp^{\mathcal{C}}, "d") = \left\lceil 2 \times \frac{2}{5} \right\rceil = 1$
- $c_{\text{str}}(dp^{\mathcal{C}}, "d") = \left\lceil 2 \times \frac{3}{5} \right\rceil = 1$
- $c_{\text{int}}(dp^{\mathcal{C}}, "e") = \left\lceil 2 \times \frac{2}{5} \right\rceil = 1$
- $c_{\text{str}}(dp^{\mathcal{C}}, "e") = \left\lceil 2 \times \frac{3}{5} \right\rceil = 1$

□

Property 3.2 (*select* is conjuction collapsible)**.** Let φ and Ψ be select expressions, then

$$\sigma_{\varphi}\left(\sigma_{\Psi}(dp^{\mathcal{C}})\right) = \sigma_{\Psi \wedge \varphi}(dp^{\mathcal{C}})$$

□

Example 3.4 (*select* is conjuction collapsible)**.** Assume the same initial data pilot and counters from Example 3.1. Let θ and α be comparison operators such that θ is defined for type *int* only and α for types *int* and *str*. Consider the equations: (1) $\sigma_{d\alpha e}\left(\sigma_{a\theta b}(dp^{\mathcal{C}})\right)$, presented in Example 3.3, and (2) $\sigma_{d\alpha e \wedge a\theta b}(dp^{\mathcal{C}})$ obtained after applying Property 3.2 to (1). The counters for (2) are computed as follows:

$$Card(\epsilon, dp^{\mathcal{C}'}) = \left\lfloor \frac{1}{40} \times 25 \times 30 \times (5 + 10) \times (6 + 8) \right\rfloor = 2$$

Counters of paths directly involved in condition $d\alpha e$:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 2 \times \frac{5}{15} \right\rfloor = 1$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 2 \times \frac{6}{14} \right\rfloor = 1$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 2 \times \frac{10}{15} \right\rfloor = 1$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 2 \times \frac{8}{14} \right\rfloor = 1$

Counters of paths directly involved in condition $a\theta b$:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"a"}) = \left\lceil 2 \times \frac{25}{25} \right\rfloor = 2$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"b"}) = \left\lceil 2 \times \frac{30}{30} \right\rfloor = 2$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"a"}) = 0$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"b"}) = 0$

$\square$

Projection

Structural estimation

For two arbitrary documents d_1 and d_2 in $\mathcal{C}$ can have totally different projection idempotent sets given the schemaless nature of $\mathcal{C}$. The union of all these sets makes up the leaves of the data pilot of $\pi_P(\mathcal{C})$, which ensures the *forth* property is maintained. For a data pilot, this means that any leaf in $dp^{\pi_P(\mathcal{C})}$ is part of at least one projection idempotent set of documents in $\mathcal{C}$ and, by definition, has a prefix in P. Therefore, removing any path $q \in Lf(dp^{\mathcal{C}})$ with no prefix in P—which is precisely what function *branch* does—captures the structural behavior of a projection over the data.

Let $K = \{p \mid \forall p \in P \wedge p \overset{r}{\backsim} dp^{\mathcal{C}}\}$. That is, the set of path expressions in P that exist in $dp^{\mathcal{C}}$. Then,

$$\pi_P(dp^{\mathcal{C}}) = \bigcup_{p \in K} \text{branch}\Big(dp^{\mathcal{C}}, p\Big) \text{ and } \pi_P^{\oslash}(dp^{\mathcal{C}}) = \bigcup_{p \in K} \text{branch}\Big(dp^{\mathcal{C}}, p\Big)$$

When projecting array elements, all array indexes are seen as $\star$ from a Data Pilot perspective. For instance, the result of $\pi_{\text{"b"}\cdot 0}(\mathcal{K})$ in Example 2.1 is estimated by $\pi_{\text{"b"}\cdot\star}(dp^{\mathcal{K}})$. Therefore, the estimations of $\pi_{s\cdot x}(\mathcal{C})$ for any path s realized in $dp^{\mathcal{C}}$ and $x \in \mathbb{N}$ are indistinguishable from one another.

Cardinality estimation

Counters remain the same for paths kept by $\pi_P(dp^{\mathcal{C}})$. Note that there could be documents in $\mathcal{C}$ whose projection idempotent set with respect to P is empty, in which case such documents become empty (i.e. $\{\!\!\{\}\!\!\}$) after $\pi_P(dp^{\mathcal{C}})$. As a consequence, $Card(\pi_P(dp^{\mathcal{C}}), \epsilon) > Card(\pi_P(dp^{\mathcal{C}}), q)$ for any path q other than ϵ in $N\left(\pi_P(dp^{\mathcal{C}})\right)$, and $Card(\pi_P(dp^{\mathcal{C}}), \epsilon) = Card(dp^{\mathcal{C}}, \epsilon)$. In contrast, for a non-preserving projection, $Card(\pi_P^{\oslash}(dp^{\mathcal{C}}), \epsilon) = \max_{q \in N(dp^{\mathcal{C}}) \cap P} Card(dp^{\mathcal{C}}, q)$.

Cartesian product

Structural estimation

Let $dp^{\mathcal{A}}$ and $dp^{\mathcal{B}}$ be data pilots of $\mathcal{A}$ and $\mathcal{B}$ respectively (note this implies that for any path p in $dp^{\mathcal{A}}$ and any path q in $dp^{\mathcal{B}}$, $p_1 \neq q_1$, by definition of $\mathcal{A}$ and $\mathcal{B}$). Then,

$$dp^{\mathcal{A}} \times dp^{\mathcal{B}} = dp^{\mathcal{A}} \cup dp^{\mathcal{B}}$$

Cardinality estimation

Let $dp^{\mathcal{C}}$ be data pilots such that $dp^{\mathcal{C}} = dp^{\mathcal{A}} \times dp^{\mathcal{B}}$. Then, the number of documents in $dp^{\mathcal{C}}$ is estimated through Equation 22 and the number of documents where a path $p \in dp^{\mathcal{C}}$ is of type $t \in \mathfrak{T}$ is given by Equation 23.

$$Card(dp^{\mathcal{C}}, \epsilon) = Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon) \tag{22}$$

$$c_t(dp^{\mathcal{C}}, p) = \begin{cases} c_t(dp^{\mathcal{A}}, p) \times Card(dp^{\mathcal{B}}, \epsilon) & \text{if } p \overset{r}{\frown} dp^{\mathcal{A}} \\ c_t(dp^{\mathcal{B}}, p) \times Card(dp^{\mathcal{A}}, \epsilon) & \text{if } p \overset{r}{\frown} dp^{\mathcal{B}} \end{cases} \tag{23}$$

Example 3.5 (Cardinality estimation of cartesian product)**.** Assume data pilots $dp^{\mathcal{A}}$ and $dp^{\mathcal{B}}$ with the cardinalities and counters shown in Table 9. Let $dp^{\mathcal{C}} = dp^{\mathcal{A}} \times dp^{\mathcal{B}}$. The counters of $dp^{\mathcal{C}}$ are computed as follows:

$$Card(dp^{\mathcal{C}}, \epsilon) = Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon) = 40 \times 15 = 600$$

DataPilot	Path	int	str	obj	Cardinality
$dp^{\mathcal{A}}$	ϵ	0	0	40	40
$dp^{\mathcal{A}}$	a	25	15	0	40
$dp^{\mathcal{A}}$	b	30	0	0	30
$dp^{\mathcal{B}}$	ϵ	0	0	15	15
$dp^{\mathcal{B}}$	d	5	10	0	15
$dp^{\mathcal{B}}$	e	6	8	0	12

Table 9. Data pilots from Example 3.5

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"a"}) = 25 \times 15 = 375$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"a"}) = 15 \times 15 = 225$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"b"}) = 30 \times 15 = 450$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"b"}) = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = 5 \times 40 = 200$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = 10 \times 40 = 400$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = 6 \times 40 = 240$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = 8 \times 40 = 320$

□

Inner Join

Cardinality estimation

Let $\mathcal{A}$, $\mathcal{B}$ be collections and α be a predicate of the form $p\theta q$. The process to estimate the cardinality of operation $\mathcal{A} \bowtie_\alpha \mathcal{B}$ is explained below.

The number of all possible document merges that can occur between both collections is $Card(dp^{\mathcal{A}}, \epsilon) * Card(dp^{\mathcal{B}}, \epsilon)$ or, equivalently, $Card(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, \epsilon)$.

Suppose θ is defined for any type in some set B, then tuples are not merged if (1) p or q are of any type in $A - B$, or (2) p and q are of different types. Therefore, estimating the number of merges depends on the probability of p and q being of any type in B over all possible merges. Assuming independence of events, this probability is given by Equation 24.

$$\left(\sum_{b\in B} \frac{c_b(dp^{\mathcal{A}}, p) \times Card(dp^{\mathcal{B}}, \epsilon)}{Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon)}\right) \times \left(\sum_{b\in B} \frac{c_b(dp^{\mathcal{B}}, q) \times Card(dp^{\mathcal{A}}, \epsilon)}{Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon)}\right)$$
$$= \left(\sum_{b\in B} \frac{c_b(dp^{\mathcal{A}}, p)}{Card(dp^{\mathcal{A}}, \epsilon)}\right) \times \left(\sum_{b\in B} \frac{c_b(dp^{\mathcal{B}}, q)}{Card(dp^{\mathcal{B}}, \epsilon)}\right) \tag{24}$$
$$= \frac{1}{Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon)} \times \left(\sum_{b\in B} c_b(dp^{\mathcal{A}}, p)\right) \times \left(\sum_{b\in B} c_b(dp^{\mathcal{B}}, q)\right)$$

Lemma 3.3. Equation 24 is equivalent to $drf(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q)$.

Proof.

$$\frac{1}{(Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon))^2} \left(\sum_{b\in B} c_b(dp^{\mathcal{A}}, p) \times Card(dp^{\mathcal{B}}, \epsilon)\right) \left(\sum_{b\in B} c_b(dp^{\mathcal{B}}, q) \times Card(dp^{\mathcal{A}}, \epsilon)\right)$$
$$= \frac{1}{Card(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, \epsilon)^2} \times \left(\sum_{b\in B} c_b(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p)\right) \left(\sum_{b\in B} c_b(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, q)\right)$$
$$= drf(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q)$$

□

Then, the number of documents produced after a join operation is computed as indicated in Equation 25.

$$Card(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, \epsilon) = \left\lfloor \underbrace{Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon)}_{\text{All possible document merges}} \times \underbrace{\sum_{b\in B} \frac{c_b(dp^{\mathcal{A}}, p)}{Card(dp^{\mathcal{A}}, \epsilon)} \times \sum_{b\in B} \frac{c_b(dp^{\mathcal{B}}, q)}{Card(dp^{\mathcal{B}}, \epsilon)}}_{\text{Probability computed in Equation 24}} \right\rfloor$$
$$= \left\lfloor \frac{Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon)}{Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon)} \times \left(\sum_{b\in B} c_b(dp^{\mathcal{A}}, p)\right) \times \left(\sum_{b\in B} c_b(dp^{\mathcal{B}}, q)\right) \right\rfloor$$
$$= \left(\sum_{b\in B} c_b(dp^{\mathcal{A}}, p)\right) \times \left(\sum_{b\in B} c_b(dp^{\mathcal{B}}, q)\right) \tag{25}$$

Alternatively, Equation 25 can also be written as $Card(\sigma_{p\theta q}(dp^{\mathcal{A}} \times dp^{\mathcal{B}}), \epsilon)$ as shown in Lemma 3.4.

Lemma 3.4. $Card(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, \epsilon) = Card(\sigma_{p\theta q}(dp^{\mathcal{A}} \times dp^{\mathcal{B}}), \epsilon)$

Proof.

$$
\begin{aligned}
Card(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, \epsilon) &= \left\lfloor Card(dp^{\mathcal{A}}, \epsilon) \times Card(dp^{\mathcal{B}}, \epsilon) \times \sum_{b \in B} \frac{c_b(dp^{\mathcal{A}}, p)}{Card(dp^{\mathcal{A}}, \epsilon)} \times \sum_{b \in B} \frac{c_b(dp^{\mathcal{B}}, q)}{Card(dp^{\mathcal{B}}, \epsilon)} \right\rfloor \\
&= \left\lfloor Card(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, \epsilon) \times drf(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q) \right\rfloor \\
&= Card(\sigma_{p\theta q}(dp^{\mathcal{A}} \times dp^{\mathcal{B}}), \epsilon)
\end{aligned}
$$

□

To compute the new counters after a join, first consider all possible merges of documents. Out of the resulting $Card(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, \epsilon)$ documents, the operation on the data only keeps those where p and q are equal. For a data pilot, this translates to the likelihood of p and q having the same type given the occurrences of each path segregated by type.

Let s be an arbitrary path in $dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}$, then if s is either p or q, the percentage of documents where s is of type $t \in B$ is

$$
\frac{c_t(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p)}{\sum_{b \in B} c_b(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p)} = \frac{c_t(dp^{\mathcal{A}}, p) \times Card(dp^{\mathcal{B}}, \epsilon)}{\sum_{b \in B} c_b(dp^{\mathcal{A}}, p) \times Card(dp^{\mathcal{B}}, \epsilon)} = \frac{c_t(dp^{\mathcal{A}}, p)}{\sum_{b \in B} c_b(dp^{\mathcal{A}}, p)}
$$

If s is a path other than p and q, the percentage of document retention per type is

$$
\frac{c_t(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, q)}{\sum_{b \in B} c_b(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, q)} = \frac{c_t(dp^{\mathcal{B}}, q) \times Card(dp^{\mathcal{A}}, \epsilon)}{\sum_{b \in B} c_b(dp^{\mathcal{B}}, q) \times Card(dp^{\mathcal{A}}, \epsilon)} = \frac{c_t(dp^{\mathcal{B}}, q)}{\sum_{b \in B} c_b(dp^{\mathcal{B}}, q)}
$$

If s has some type in $A - B$, then document retention is 0 because θ is not defined for such type.

Altogether, the percentage of document retention for a path s with arbitrary type $a \in \mathfrak{T}$ is equivalent to $tdf_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q, s)$ as shown below:

$$
tdf_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q, s) = \begin{cases} 0 & \text{if } a \notin B \\ \dfrac{c_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, s)}{\sum_{b \in B} c_b(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, s)} & \text{if } s \in \text{exprpaths}\,(p\theta q) \\ \dfrac{c_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, s)}{Card(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, \epsilon)} & \text{if } s \notin \text{exprpaths}\,(p\theta q) \end{cases} \qquad (26)
$$

Equation 26 can be written in simpler terms by eliminating its dependence on *exprpaths*, which results in Equation 27.

$$tdf_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q, s) = \begin{cases} 0 & \text{if } a \notin B \\ \dfrac{c_t(dp^{\mathcal{A}}, p)}{\sum_{b \in B} c_b(dp^{\mathcal{A}}, p)} & \text{if } s = p \\ \dfrac{c_t(dp^{\mathcal{B}}, q)}{\sum_{b \in B} c_b(dp^{\mathcal{B}}, q)} & \text{if } s = q \\ \dfrac{c_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, s)}{Card(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, \epsilon)} & \text{otherwise} \end{cases} \tag{27}$$

This type density factor is used to estimate counters as indicated by Equation 28.

$$c_a(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, s) = \left\lceil Card(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, \epsilon) \times tdf_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q, s) \right\rfloor \tag{28}$$

Lemma 3.5. For any path s in either $dp^{\mathcal{A}}$ or $dp^{\mathcal{B}}$, $c_a(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, s) = c_a(\sigma_{p\theta q}(dp^{\mathcal{A}} \times dp^{\mathcal{B}}), s)$.

Proof.

$$\begin{aligned} c_a(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, s) &= \left\lceil Card(dp^{\mathcal{A}} \bowtie_{p\theta q} dp^{\mathcal{B}}, \epsilon) \times tdf_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q, s) \right\rfloor \\ &= \left\lceil Card(\sigma_{p\theta q}(dp^{\mathcal{A}} \times dp^{\mathcal{B}}), \epsilon) \times tdf_a(dp^{\mathcal{A}} \times dp^{\mathcal{B}}, p\theta q, s) \right\rfloor \quad \text{(by Lemma 3.4)} \\ &= c_a(\sigma_{p\theta q}(dp^{\mathcal{A}} \times dp^{\mathcal{B}}), s) \quad \text{(by Equation 21)} \end{aligned}$$

□

Example 3.6. Assume the same data pilots $dp^{\mathcal{A}}$ and $dp^{\mathcal{B}}$ from Example 3.5. Further, let θ be a comparison operator defined for type *int* and p and q paths such that $p \overset{r}{\backsim} dp^{\mathcal{A}}$ and $p \overset{r}{\backsim} dp^{\mathcal{B}}$. The counters for data pilots $\sigma_{a\theta d}(dp^{\mathcal{A}} \times dp^{\mathcal{B}})$ and $dp^{\mathcal{A}} \bowtie_{a\theta d} dp^{\mathcal{B}}$ are computed as follows:

For $dp^{\mathcal{C}} = \sigma_{a\theta d}(dp^{\mathcal{A}} \times dp^{\mathcal{B}})$, Example 3.5 shows how the counters change after $dp^{\mathcal{A}} \times dp^{\mathcal{B}}$. Using such information as a starting point, the new counters of the entire algebraic expressions are:

$$Card(dp^{\mathcal{C}}, \epsilon) = \left\lfloor \frac{1}{600} \times 375 \times 200 \right\rfloor = 125$$

Counters of paths directly involved in the condition:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"a"}) = \left\lceil 125 \times \frac{375}{375} \right\rfloor = 125$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"a"}) = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 125 \times \frac{200}{200} \right\rfloor = 125$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = 0$

Other counters:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"b"}) = \left\lceil 125 \times \frac{450}{600} \right\rfloor = 94$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"b"}) = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 125 \times \frac{240}{600} \right\rfloor = 50$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 125 \times \frac{320}{600} \right\rfloor = 67$

For $dp^{\mathcal{C}} = dp^{\mathcal{A}} \bowtie_{a\theta d} dp^{\mathcal{B}}$, the new counters are:

$$Card(dp^{\mathcal{C}}, \epsilon) = 25 \times 5 = 125$$

Counters of paths directly involved in the join condition:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"a"}) = \left\lceil 125 \times \frac{25}{25} \right\rfloor = 125$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"a"}) = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"d"}) = \left\lceil 125 \times \frac{5}{5} \right\rfloor = 125$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"d"}) = 0$

Other counters:

- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"b"}) = \left\lceil 125 \times \frac{30 \times 15}{40 \times 15} \right\rfloor = 94$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"b"}) = 0$
- $c_{\text{int}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 125 \times \frac{6 \times 40}{40 \times 15} \right\rfloor = 50$
- $c_{\text{str}}(dp^{\mathcal{C}}, \text{"e"}) = \left\lceil 125 \times \frac{8 \times 40}{40 \times 15} \right\rfloor = 67$

□

Unnest

Structural estimation

Definition 3.1 (non-preserving unnest)**.** Denoted as $\mu_p^{\oslash}(dp^{\mathcal{C}})$, a non-preserving unnest of $dp^{\mathcal{C}}$ produces a new data pilot where the subtree rooted at p is equal to the subtree from $dp^{\mathcal{C}}$ rooted at $p.\star$. Formally,

$$\mu_p^{\oslash}(dp^{\mathcal{C}}) = \text{replace}\Big(dp^{\mathcal{C}}, \text{subtree}\Big(dp^{\mathcal{C}}, p.\star\Big), p\Big) \quad (29)$$

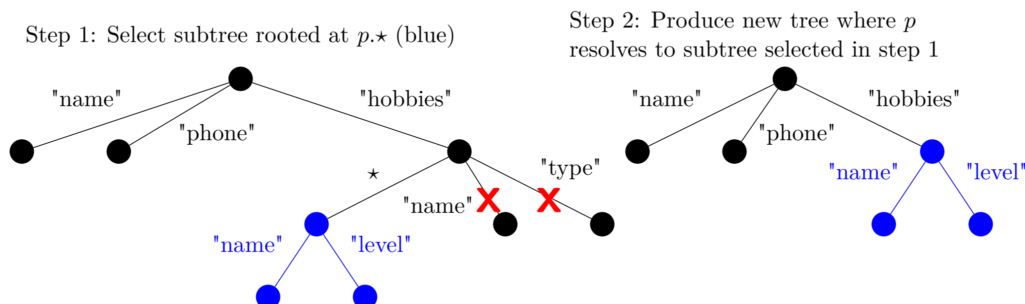


Figure 39. Example of a non-preserving unnest: $\mu_{\text{hobbies}}^{\oslash}(dp^{\mathcal{C}})$

Definition 3.2 (preserving unnest)**.** Denoted as $\mu_p(dp^{\mathcal{C}})$, a preserving unnest of $dp^{\mathcal{C}}$ produces a new data pilot where the subtree rooted at p is the union of the subtrees from $dp^{\mathcal{C}}$ rooted at non-array children of p and the subtrees from $dp^{\mathcal{C}}$ rooted at children of $p.\star$. Formally,

$$\begin{aligned}\mu_p\left(dp^{\mathcal{C}}\right) = \text{replace}\Big(& \\ & dp^{\mathcal{C}}, \text{subtree}\Big(dp^{\mathcal{C}}, p\cdot\star\Big) \cup \\ & \quad \Big(\text{subtree}\Big(dp^{\mathcal{C}}, p\Big) - \text{branch}\Big(\text{subtree}\Big(dp^{\mathcal{C}}, p\Big), \star\Big)\Big), p\Big)\end{aligned}$$

cardinality estimation

To understand how the unnest operation affects counters, first consider how the deconstruction process affects arrays. Given a collection $\mathcal{C}$, suppose a document $doc \in \mathcal{C}$ with a path p that meets all deconstruction invariants. This means that p resolves to an array with some number x of elements. By deconstructing doc based on p, x new documents are created such that path p resolves to an array element on each new document. For instance, if the array in question has 2 integers and 1 string, then 3 almost-exact copies of doc are created, except that p resolves to an integer in two of them and a string in the remaining one.

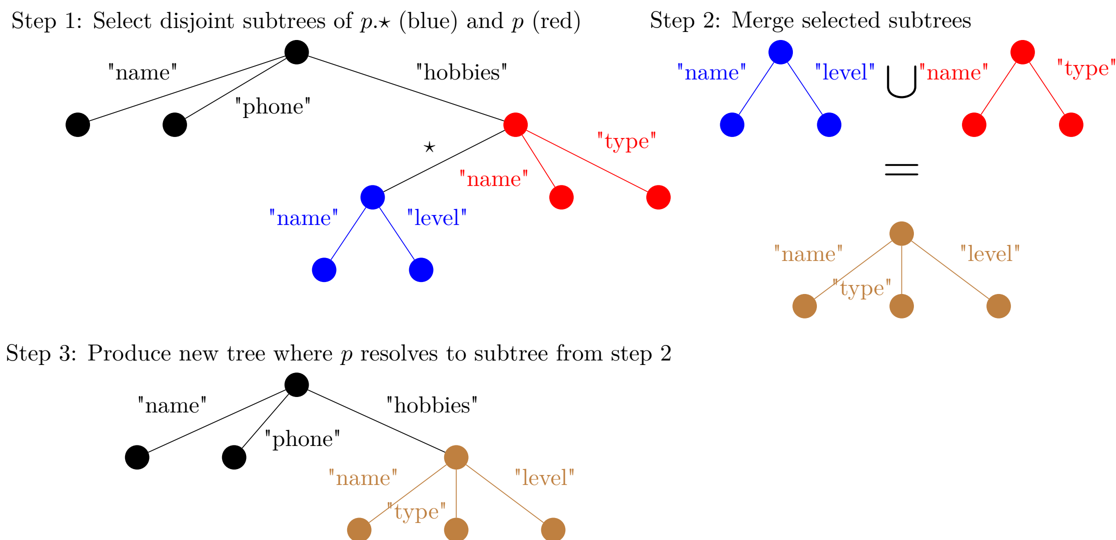


Figure 40. Example of a preserving unnest: $\mu_{\text{hobbies}}(dp^{\mathcal{C}})$

On a collection level, the number of new documents produced after deconstruction based on p is the cumulative length of the arrays that p resolves to on each document meeting all deconstruction invariants. On $dp^{\mathcal{C}}$, this cumulative length can be computed as $Card(dp^{\mathcal{C}}, p.\star)$. When grouping these new documents based on the type of p, the number of documents in each group corresponds to $c_t(dp^{\mathcal{C}}, p.\star)$, where $t \in \mathfrak{T}$.

If no document in $\mathcal{C}$ meets all the invariants, then deconstruction based on p is not possible. That is, if either of the following is true:

- p is not realized in $dp^{\mathcal{C}}$ (Invariant 1 is false on all documents).
- p is realized in $dp^{\mathcal{C}}$, but $c_{\text{arr}}(dp^{\mathcal{C}}, p) = 0$ (Invariant 2 is false on all documents).
- p is realized in $dp^{\mathcal{C}}$, but $c_{\text{arr}}(dp^{\mathcal{C}}, p) = \varepsilon(dp^{\mathcal{C}}, p)$ (Invariant 3 is false on all documents).

For any such case, a non-preserving unnest removes all documents from $\mathcal{C}$ (i.e. $Card(\mu_p^{\oslash}(dp^{\mathcal{C}}), \epsilon) = 0$) and a preserving one keeps them all (i.e. $Card(\mu_p(dp^{\mathcal{C}}), \epsilon) = Card(dp^{\mathcal{C}}, \epsilon)$).

Otherwise, when at least one document in $\mathcal{C}$ meeets all invariants, a non-preserving unnest creates as many new documents as the cummulative number of elements within documents where p is an array. Therefore,

$$Card(\mu_p^{\oslash}(dp^{\mathcal{C}}), \epsilon) = Card(dp^{\mathcal{C}}, p.\star) \tag{30}$$

A preserving unnest also produces $Card(dp^{\mathcal{C}}, p.\star)$ new documents, while keeping documents where the unnest invariants are not met (i.e. an extra $Card(dp^{\mathcal{C}}, \epsilon) - c_{\text{arr}}(dp^{\mathcal{C}}, p) + \varepsilon(dp^{\mathcal{C}}, p)$ documents are included in the output). Thus,

$$Card(\mu_p(dp^{\mathcal{C}}), \epsilon) = Card(dp^{\mathcal{C}}, p.\star) + Card(dp^{\mathcal{C}}, \epsilon) - c_{\text{arr}}(dp^{\mathcal{C}}, p) + \varepsilon(dp^{\mathcal{C}}, p) \tag{31}$$

Since p resolves to an array element after deconstruction, then for a non-preserving unnest, $c_t(\mu_p^{\oslash}(dp^{\mathcal{C}}), p)$ is equal to the number of array elements with type $t \in \mathfrak{T}$:

$$c_t(\mu_p^{\oslash}(dp^{\mathcal{C}}), p) = c_t(dp^{\mathcal{C}}, p.\star) \tag{32}$$

for a preserving unnest, the number of array elements with a given type is added to the counters of p in $dp^{\mathcal{C}}$. In the case of arrays, however, because documents with empty arrays are kept and the ones deconstructed removed, $c_{arr}(dp^{\mathcal{C}}, p)$ must be substracted and $\varepsilon(dp^{\mathcal{C}}, p)$ added:

$$c_t(\mu_p^{\oslash}(dp^{\mathcal{C}}), p) = \begin{cases} c_t(dp^{\mathcal{C}}, p.\star) + \varepsilon(dp^{\mathcal{C}}, p) & \text{if } t = arr \\ c_t(dp^{\mathcal{C}}, p) + c_t(dp^{\mathcal{C}}, p.\star) & \text{otherwise} \end{cases} \tag{33}$$

For any path q other than p, both variations of unnest re-distribute counters for any path $q \neq p$ based on the percentage of documents held on $dp^{\mathcal{C}}$:

$$c_t(\mu_p^{\oslash}(dp^{\mathcal{C}}), q) = c_t(\mu_p^{\oslash}(dp^{\mathcal{C}}), \epsilon) \times \frac{c_t(dp^{\mathcal{C}}, q)}{Card(dp^{\mathcal{C}}, \epsilon)} \tag{34}$$

$$c_t(\mu_p(dp^{\mathcal{C}}), q) = c_t(\mu_p(dp^{\mathcal{C}}), \epsilon) \times \frac{c_t(dp^{\mathcal{C}}, q)}{Card(dp^{\mathcal{C}}, \epsilon)} \tag{35}$$

Group

Let $dp^{\mathcal{C}}$ be a Data Pilot of collection $\mathcal{C}$, G a set of paths to group by, $l_1, l_2, \ldots, l_k$ labels not in $N\left(dp^{\mathcal{C}}\right)$, $q_1, \ldots, q_k$ paths in $N\left(dp^{\mathcal{C}}\right)$, and $f_1, \ldots, f_k$ the name of aggregate

functions. Group has the form:

$$_P\gamma_{r_1:f_1(q_1),r_2:f_2(q_2),\ldots,r_k:f_k(q_k)}(dp^{\mathcal{C}})$$

Note that a data pilot aggregate function f in references an aggregate function in a database system (e.g. *sum*), but does not operate on the data. Rather, operations are carried out on paths and counters. That is, f reflects the structural and path occurrence implications for a collection when using the homonymic aggregate named f on the data.

The result of each data pilot aggregate function f_i evaluated at q_i dangles from branch l_i in the new tree generated by the f_i.

structural estimation

In each path of G, there exists at least one document in $\mathcal{C}$. To ensure path completeness, the resulting Data Pilot after applying γ encompasses elements of $\pi_G(dp^{\mathcal{C}})$. Each triplet $l_i, f_i, q_i \mid 1 \leq i \leq k$ is utilized to construct a tree T_i such that $Lf(\mathsf{T}_i) = \{l_i \cdot s \mid \forall s \in Lf(f_i(q_i))\}$. The structure of each tree generated by an aggregate function varies depending on the its type:

- **Scalar aggregates** applied to the data yield literals, essentially forming a tree with a single node (ϵ) from a structural standpoint. Therefore, for all scalar aggregates, a triplet l_i, f_i, q_i generates a tree T_i such that $Lf(\mathsf{T}_i) = \{l_i\}$.

- **Array aggregates** applied to the data result in an array, often by appending values from a path. Structurally, this entails that any array aggregate f produces a tree where at least ϵ and $\star$ are included as nodes. Thus, a triplet l_i, f_i, q_i, with f_i representing an array aggregate, generates a T_i with $Lf(\mathsf{T}_i) = \{\star\}$ at the very least.

The structure of the Data Pilot created by the group operator is thus given by:

$$Lf\left(_P\gamma_{l_1:f_1(q_1),l_2:f_2(q_2),\ldots,l_k:f_k(q_k)}\left(dp^{\mathcal{C}}\right)\right) = Lf\left(\pi_G\left(dp^{\mathcal{C}}\right)\right) \cup \bigcup_{1\leq i\leq k} Lf(\mathsf{T}_i)$$

Cardinality estimation

Suppose an arbitrary collection $\mathcal{C}$ is devided into n partitions. After the merging step, the number of documents remaining is one per partition; that is, n.

Let $dp^{\mathcal{C}'}$ be the Data Pilot resulting from a group operation over $dp^{\mathcal{C}}$ and t_i the return type of an aggregate function f_i. Then,

- $Card(dp^{\mathcal{C}'}, \epsilon) = n$
- If f_i is a scalar aggregate, $c_{t_i}(dp^{\mathcal{C}'}, l_i) = n$ and $Card(dp^{\mathcal{C}'}, l_i) = c_{t_i}(dp^{\mathcal{C}'}, l_i)$.
- If f_i is an array aggregate, $c_{\text{arr}}(dp^{\mathcal{C}'}, l_i) = n$, $Card(dp^{\mathcal{C}'}, l_i) = c_{\text{arr}}(dp^{\mathcal{C}'}, l_i)$ and $Card(dp^{\mathcal{C}'}, l_i \cdot \star) = Card(dp^{\mathcal{C}}, q_i)$.
- If f_i is a document aggregate, then $c_{\text{obj}}(dp^{\mathcal{C}'}, l_i) = n$ and $Card(dp^{\mathcal{C}'}, l_i) = c_{\text{obj}}(dp^{\mathcal{C}'}, l_i)$.

Furthermore, the counters for any path $l_i \cdot s$ where s is produced by a non-scalar f_i are dependant on how the specific f_i works.

In practice, n can only be computed from the actual values in the data. To understand how a Data Pilot estimates n, first consider the case where there is only one path expression p in G: the biggest value n can take is $Card(dp^{\mathcal{C}}, \epsilon)$, when p is present in all documents and resolves to distinct values on all of them. The smallest value is 1, when either p is present on all documents and resolves to the same value on all of them, or p is absent on all documents. If $Card(dp^{\mathcal{C}}, p) < Card(dp^{\mathcal{C}}, \epsilon)$ then there is a partition with $Card(dp^{\mathcal{C}}, \epsilon) - Card(dp^{\mathcal{C}}, p)$ documents where p is undefined. If $c_{\text{null}}(dp^{\mathcal{C}}, p) > 0$ then there is a partition with $c_{\text{null}}(dp^{\mathcal{C}}, p)$ documents where p evaluates to null. As the cummulative number of documents in these two partitions increases relative to $Card(dp^{\mathcal{C}}, \epsilon)$, the number of possible partitions to estimate from the remaining documents where p does not evaluate to null decreases. Furthermore, without relying on the data, it is known with certainty that:

- If $c_{\text{null}}(dp^{\mathcal{C}}, p) = Card(dp^{\mathcal{C}}, \epsilon)$ or $Card(dp^{\mathcal{C}}, p) = 0$, then $n = 1$.
- If $c_{\text{null}}(dp^{\mathcal{C}}, p) = Card(dp^{\mathcal{C}}, p)$ and $Card(dp^{\mathcal{C}}, p) < Card(dp^{\mathcal{C}}, \epsilon)$, then n is exactly 2 as there is one partition for documents where p is undefined and another for documents where p evaluates to null.
- If $Card(dp^{\mathcal{C}}, p) < Card(dp^{\mathcal{C}}, \epsilon)$ and $Card(dp^{\mathcal{C}}, p) - c_{\text{null}}(dp^{\mathcal{C}}, p) > 0$ then n is at least 3.

Likewise, these three facts extend to the case when there are multiple path expressions in G. Each path expression can either be undefined, evaluate to null or a value other than null. Let j be the number of paths in G, then there are 3^j permutations of these three possibilities. For instance, if $G = \{p_1, p_2\}$ these permutations are: (undefined, undefined), (null, null), (nut-null, not-null), (undefined, not-null), (not-null, undefined), (null, not-null), (not-null, null), (undefined, null), and (null, undefined).

If all path expressions are null or undefined within all documents, then $n = 1$. In a Data Pilot, this happens when $c_{\text{null}}(dp^{\mathcal{C}}, p) = Card(dp^{\mathcal{C}}, p)$ for all $p \in P$ or, alternatively, $Card(dp^{\mathcal{C}}, p) = 0$ for all $p \in G$. For all other $3^j - 2^j$ permutations, excluding those containing only null and undefined paths, a Data Pilot is unable to guess the exact values that coincide within documents and can thus only assume the worst case of:

$$\min\left(Card(dp^{\mathcal{C}}, \epsilon), 2 + \sum_{p \in G} Card(dp^{\mathcal{C}}, p)\right)$$

To further investigate the impact of null and undefined paths on the number of groups, the following experiment is implemented:

Experimental Setup Generate a table with varying numbers of documents and paths per document. Each cell in the table represents a path evaluation and can have one of three string values: "value", "undefined", or "null". "null" indicates that the path evaluates to null, "undefined" indicates the path is not present, and "value"

indicates a distinct non-null value. The distribution of null, value, and undefined paths is controlled by the experimental variable `pvals`.

Experimental Variables

- **ndocs (Number of documents):** Determines the number of simulated documents (table rows).
- **ppd (Paths per document):** Determines the number of simulated paths per document (table columns).
- **npgb (Number of paths to group by):** Specifies the number of random paths chosen for grouping.
- **nruns:** Specifies the number of experimental runs.
- **pvals:** Represents the distribution probabilities as percentages, in the format `pvals=<percent of nulls>,<percent of values>,<percent of undefined>`.

Experiment Steps

1. Identify rows where none of the selected paths for grouping have a non-null value (i.e. the string "value").
2. Calculate the number of groups and documents per group based on the selected rows.
3. Determine the total number of documents in the selected rows from step 1.
4. Subtract the value obtained in step 3 from ndocs.
5. Add the number of groups obtained in step 2 to the value obtained in step 4.

The value obtained in step 4 represents the number of groups where at least one of the paths for grouping resolves to a non-null value.

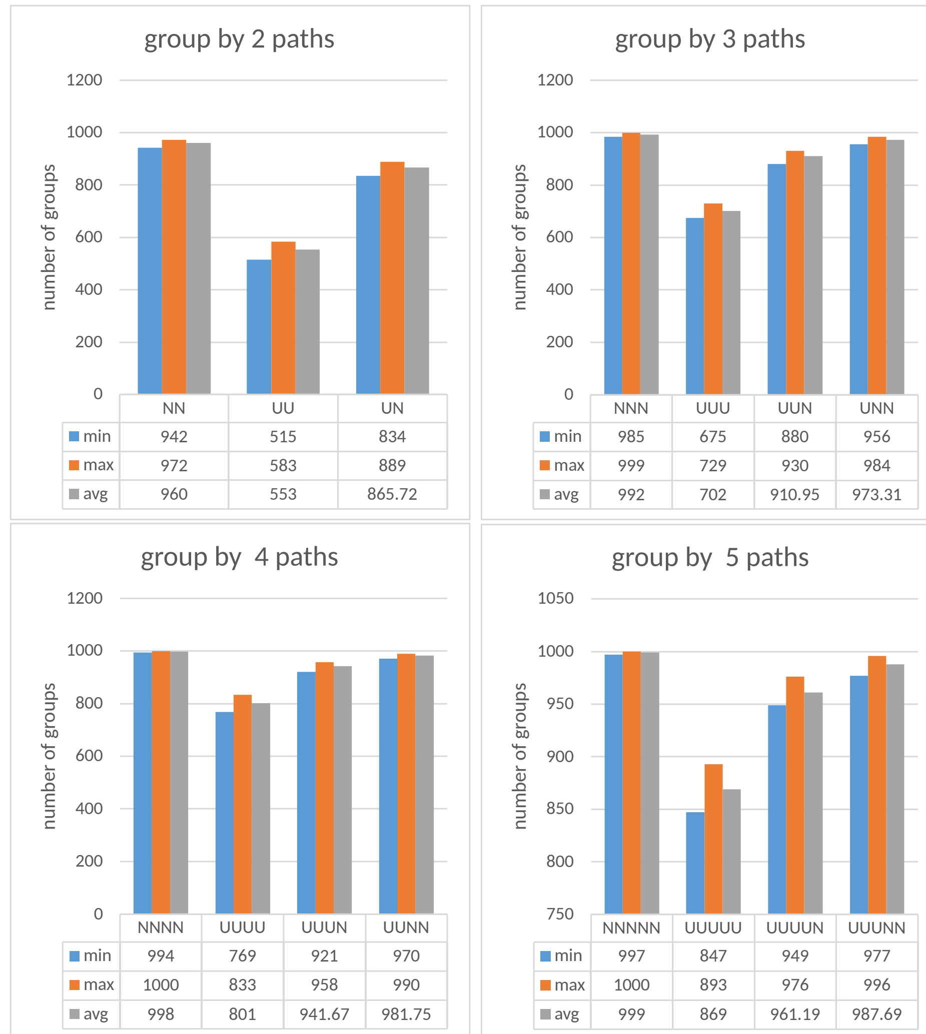


Figure 41. Impact of null and undefined paths on the number of groups

Experiment Results Figure 41 illustrates the number of groups obtained after grouping by 2, 3, 4, and 5 paths in 100 iterations. The X-axis is labeled based on the distribution of null, undefined, and value paths. The label "N" represents the normal distribution with pvals of 0.1, 0.8, and 0.1 for undefined, value, and null respectively. Similarly, the label "U" corresponds to the uniform distribution with a pval of 1/3 for all three types. The number of characters in a X-label indicates the number of paths to group by and each character the distribution type along with the minimum, maximum, and average number of groups obtained from the 100 runs.

The analysis reveals that when at least one path follows a normal distribution,

there is minimal reduction in the number of groups.

$$Card(dp^{\mathcal{C}'}, \epsilon) = \begin{cases} \max\limits_{p \in G}(Card(dp^{\mathcal{C}}, p) - c_{\text{null}}(dp^{\mathcal{C}}, p)) & \text{if } \frac{\max_{p\in G}(Card(dp^{\mathcal{C}},p) - c_{\text{null}}(dp^{\mathcal{C}},p))}{Card(dp^{\mathcal{C}},\epsilon)} < \frac{1}{3} \\ Card(dp^{\mathcal{C}}, \epsilon) & \text{otherwise} \end{cases}$$

group families

The cost to compute all possible groups with respect to types in $\mathfrak{A}$ is considerably high. To ease such computation and provide a better upper bound, the combinations of paths and types forming groups are derived from the number of ways a group can be formed. This concept is called group families, an example of which is provided in Example 3.7.

Example 3.7. Suppose G has 3 paths and the database system has 3 atomic types: $\mathfrak{A} = \{\text{int, str, date}\}$, then there are 10 different ways to form groups:

Type	GF_1	GF_2	GF_3	GF_4	GF_5	GF_6	GF_7	GF_8	GF_9	GF_{10}
int	3	0	0	2	2	1	0	0	1	1
str	0	3	0	1	0	2	2	1	0	1
date	0	0	3	0	1	0	1	2	2	1

Table 10. Number of paths with a specific type in a group family

□

Group families are rationalized through a generating function and its properties. First, consider that any path $p \in G$ can have at most 1 type on each group; this is rather obvious, but it is important when defining the function's coefficients. A group can have all paths be the same type or a combination of types and paths as seen in Example 3.7, which poses the question "Given a set of paths to group by, how many group families are there?". Consider the same setup as in Example 3.7, the information $|\mathfrak{A}| = 3$ and $|G| = 3$ can be encoded in the following polynomial:

$$\underbrace{(x^0 + x^1 + x^2 + x^3)}_{\text{int}} \times \underbrace{(x^0 + x^1 + x^2 + x^3)}_{\text{str}} \times \underbrace{(x^0 + x^1 + x^2 + x^3)}_{\text{date}}$$

where the nature of x is immaterial, and the exponents read as follows: 0 means a group family where no paths have type t, 1 means a group family where 1 path has type t, etc. For instance, in the first term of the polynomial (the set of parentheses marked as int), x^0 encodes the information that a group family may have 0 paths be of type int, x^1 that it may have 1 path of type *int*, and so on. When generalized to arbitrary G and $\mathfrak{A}$, the expression becomes:

$$\underbrace{(x^0 + x^1 + x^2 + x^3 + \cdots + x^{|G|})}_{t_1} \times \underbrace{(x^0 + x^1 + x^2 + x^3 + \cdots + x^{|G|})}_{t_2} \times \cdots \times \underbrace{(x^0 + x^1 + x^2 + x^3 + \cdots + x^{|G|})}_{t_{|\mathfrak{A}|}} = \Big(\sum_{k=0}^{|G|} x^k\Big)^{|\mathfrak{A}|} \tag{36}$$

After pluggin in the values $|G| = 3$ and $|\mathfrak{A}| = 3$, the new expression expands to the polynomial:

$$x^9 + 3x^8 + 6x^7 + 10x^6 + 12x^5 + 12x^4 + 10x^3 + 6x^2 + 3x + 1$$

where the relevant terms are those with exponent equal to or less than $|G|$ (more about this later). They read as:

- $10x^3$: "Given 3 paths to group by, **each with an arbitrary type**, there are 10 group families"; which is exactly the number group families shown in Example 3.7.

- $6x^2$: "Given 3 paths to group by, where **2 of them have the same type**, there are 6 group families"; these correspond to GF_4, GF_5, GF_6, GF_7, GF_8, and GF_9.

- $3x$: "Given 3 paths to group by, where **all of them have the same type**, there are 3 group families"; these correspond to GF_1, GF_2, GF_3.

- $1 = x^0$: "Given 3 paths to group by, where **all of them have different type**, there is 1 group family"; this is GF_{10}.

Any term of the form cx^n, where c is a coefficient and $n > |P|$ violates the initial conditions set in the problem.

From an implementation perspective, the number of types in a system are fixed, thus computing the polynomial only depends on $|G|$ in practice. When designing an algorithm to determine the number of group families, the polynomial coefficients can be computed ahead of time and fixed in the code. Aditionally, newer terms in the polynomial can be derived from previous ones as seen in Example 3.8.

Example 3.8. Consider $|\mathfrak{A}| = 3, 4$, $|G| = 0, 1, 2, \ldots, 5$, and let C denote the table shown in Table 11, where each cell $C_{i,j}$ is a coefficient from the expanded polynomial produced by the generating function evaluated in particular choosings of $|G|$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	**1**															
1	1	**3**	3	1												
2	1	3	**6**	7	6	3	1									
3	1	3	6	**10**	12	12	10	6	3	1						
4	1	3	6	10	**15**	18	19	18	15	10	6	3	1			
5	1	3	6	10	15	**21**	25	27	27	25	21	15	10	6	3	1

(a) $|\mathfrak{A}| = 3, |G| = 0, 1, 2, \ldots, 5$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	**1**																
1	1	**4**	6	4	1												
2	1	4	**10**	16	19	16	10	4	1								
3	1	4	10	**20**	31	40	44	40	31	20	10	4	1				
4	1	4	10	20	**35**	52	68	80	85	80	68	52	35	20	10	4	1

(b) $|\mathfrak{A}| = 4, |G| = 0, 1, 2, \ldots, 4$

Table 11. Expansion of generating function with $|\mathfrak{A}| = 3, 4$ and $|G| = 0, 1, 2, \ldots, 5$ encoded in a tabular format. Rows are values of $|G|$, columns are exponents, and any cell at row i and column j is a coefficient. Red cells denote the total number of group families per number of group attributes.

It can be seen, for instance, that cells $(3, 0), (3, 1), (3, 2)$, and $(3, 3)$ in Table 11a correspond to terms $1, 3x, 6x^2$, and $10x^3$ in Example 3.7. Additionally, when $|G| = 1$,

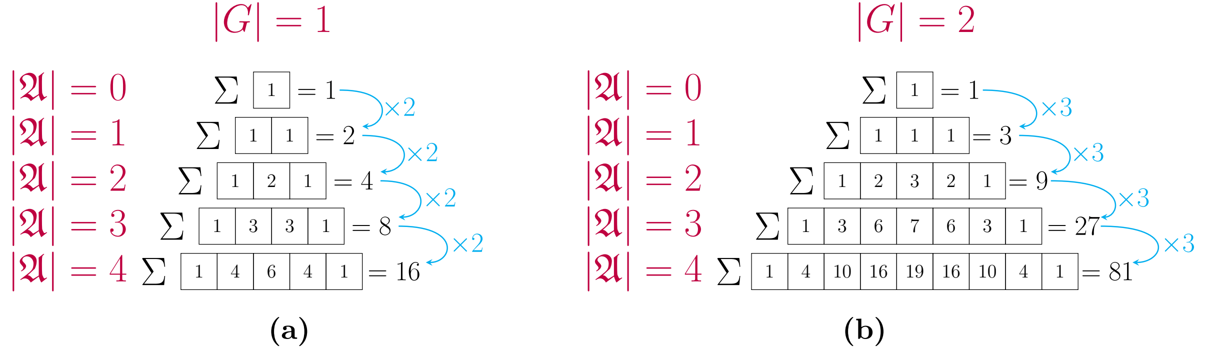


Figure 42. Rows of Pascal's triangle computed from the generating function's expansion after pluggin in $|G| = 1, 2$, and $|\mathfrak{A}| = 0, 1, 3, 4$.

the rows in both tables correspond to rows of Pascal's triangle. For instance, row 1 of Table 11a makes up the Pascal's triangle row corresponding to $|\mathfrak{A}| = 3$ in Figure 42a.

When $|G| > 1$, a variation of Pascal's triangle is constructed; an example with $|G| = 2$ is shown in Figure 42b.

A well-known property of Pascal's triangle is that each number is the sum of the two numbers above. This is true for $|G| = 1$, but for $|G| = 2$ each number is the sum of the 3 numbers above, except for the numbers next to the first and last number of the row.

□

Intersection

Unlike other DataPilot operators that compute structural and cardinality estimations independently, the intersection's structural and cardinality estimation are interdependent. While a straightforward definition might entail $N(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}) = N(dp^{\mathcal{C}_1}) \cap N(dp^{\mathcal{C}_2})$, consider a scenario where a path p exists in both DataPilots. If one DataPilot records only non-zero counters for type t and the other for a different type t', p should not be included in the result as it points to values with different types. Example 3.9 elucidates this, along with insights into computing the resulting counters and cardinalities post-intersection.

Path	obj	arr	int	str	Cardinality
ϵ	25	0	0	0	25
a	0	0	25	0	25
b	0	0	0	6	6
d	0	0	3	1	4

(a) $dp^{\mathcal{C}_1}$

Path	obj	arr	int	str	Cardinality
ϵ	15	0	0	0	15
a	0	0	10	5	15
b	0	0	4	0	4
e	0	0	4	2	6

(b) $dp^{\mathcal{C}_2}$

Table 12. DataPilots from Example 3.9

Example 3.9. Consider DataPilots $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ with the counters and cardinalities in Tables 12a and 12b. While paths "a" and "b" are present in both collections, the intersection operation considers paths and their corresponding values. Thus, no document with path "b" will belong to $\mathcal{C}_1 \cap \mathcal{C}_2$ as the non-zero counters for path "b" in $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ correspond to types *str* and *int*, respectively. For path "a", the non-zero counters for type *int* suggest a potential $\min(25, 10) = 10$ documents in the intersection. However, considering path "e" present in most documents of $\mathcal{C}_2$ but absent in $\mathcal{C}_1$, it is apparent that at least $Card(dp^{\mathcal{C}_2}, \text{e}) = 6$ documents from $\mathcal{C}_2$ will not be in the intersection. Thus, only $Card(dp^{\mathcal{C}_2}, \epsilon) - Card(dp^{\mathcal{C}_2}, \text{e}) = 15 - 6 = 9$ documents from $\mathcal{C}_2$ can potentially be in the intersection. Following a similar logic, out of the 25 documents in $\mathcal{C}_1$, only $Card(dp^{\mathcal{C}_1}, \epsilon) - Card(dp^{\mathcal{C}_1}, \text{d}) = 25 - 4 = 21$ can potentially be in the intersection. Thus, an approximation of the number of documents with path "a" pointing to a value of type *int* in the intersection is $\min(25, 10, 9, 21) = 9$.

□

Let $D = N(dp^{\mathcal{C}_1}) \triangle N(dp^{\mathcal{C}_2})$, where $\triangle$ denotes symmetric difference, be the set of paths outside the intersection (from a structural point of view). The expression $\max_{q \in N(dp^{\mathcal{C}_1}) \cap D} \left(Card(N\left(dp^{\mathcal{C}_1}\right), q)\right)$ provides the estimated number of documents in $dp^{\mathcal{C}_1}$ where there is at least one path outside the intersection. It is essential to note that this expression does not capture co-occurrences in the actual data, and therefore, it may not accurately detect cases where the intersection is empty. For instance, when

all documents in a collection have at least one path from D; refer to Example 3.10 for clarification.

Example 3.10. Consider the following collections:

Collection $\mathcal{C}_1$:

```
{a:1,b:2,d:4}
{a:2,d:3}
{a:3,f:5}
```

Collection $\mathcal{C}_2$:

```
{a:1,b:2}
{a:2,e:3}
{a:3}
```

When computing the intersection between $\mathcal{C}_1$ and $\mathcal{C}_2$, it is evident that there are no common documents, resulting in an empty intersection. Furthermore, all 3 documents in $\mathcal{C}_1$ have one path outside of the intersection, but from the DataPilot perspective this is only the case for 2 documents: path "d" is present in 2 documents and path "f" in 1, thus the number of paths from $\mathcal{C}_1$ outside of the intersection is estimated to be $\max(2, 1) = 2$. For Collection $\mathcal{C}_2$, only path "e" is outside the intersection. Thus, the number of documents from $\mathcal{C}_2$ outside of the intersection is estimated to be 1. Consequently, the estimated integer counter for paths "a" and "b" would be 1. □

The expression $\text{NumNED}\Big(dp^{\mathcal{C}_1}\Big) - \left(\max\limits_{q \in N(dp^{\mathcal{C}_1}) \cap D} \Big(Card(N(dp^{\mathcal{C}_1}), q)\Big)\right)$ estimates the number of documents in $\mathcal{C}_1$ where there is no path outside of the intersection, and similarly, $\text{NumNED}\Big(dp^{\mathcal{C}_2}\Big) - \left(\max\limits_{q \in N(dp^{\mathcal{C}_2}) \cap D} \Big(Card(N(dp^{\mathcal{C}_2}), q)\Big)\right)$ estimates the number of documents in $\mathcal{C}_2$ where there is no path outside of the intersection. Consequently, the counters of the DataPilot $dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}$ are computed through the following steps:

1. Compute the structural estimation of the intersection between DataPilots as shown in Equation 37.

$$N(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}) = N(dp^{\mathcal{C}_1}) \cap N(dp^{\mathcal{C}_2}) \tag{37}$$

2. Use Equation 38 to compute the counters of all paths in $N(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}) - \{\epsilon\}$.

$$c_t\left(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}, p\right) = \begin{cases} c_t^{\text{non-array}}\left(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}, p\right) & \text{if } \star \notin p \\ c_t^{\text{array}}\left(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}, p\right) & \text{otherwise} \end{cases} \tag{38}$$

$$\begin{aligned} & c_t^{\text{non-array}}\left(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}, p\right) = \min\Bigg(\\ & \left\lceil \frac{c_t(dp^{\mathcal{C}_1}, p)}{Card(dp^{\mathcal{C}_1}, p)} * \left(\text{NumNED}\left(dp^{\mathcal{C}_1}\right) - \max_{q \in N\left(dp^{\mathcal{C}_1}\right) \cap D} \left(Card(dp^{\mathcal{C}_1}, q)\right)\right)\right\rceil, \\ & \left\lceil \frac{c_t(dp^{\mathcal{C}_2}, p)}{Card(dp^{\mathcal{C}_2}, p)} * \left(\text{NumNED}\left(dp^{\mathcal{C}_2}\right) - \max_{q \in N\left(dp^{\mathcal{C}_2}\right) \cap D} \left(Card(dp^{\mathcal{C}_2}, q)\right)\right)\right\rceil \\ & \Bigg) \end{aligned} \tag{39}$$

$$c_t^{\text{array}}\left(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}, p\right) = \min\left(c_t\left(dp^{\mathcal{C}_1}, p\right), c_t\left(dp^{\mathcal{C}_2}, p\right)\right) \tag{40}$$

3. Compute the cardinality of the DataPilot using Equation 41.

$$\begin{aligned} Card(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}, \epsilon) = \text{NumNED}\left(dp^{\mathcal{C}_1} \cap dp^{\mathcal{C}_2}\right) + \\ \min\left(\text{NumED}\left(dp^{\mathcal{C}_1}\right), \text{NumED}\left(dp^{\mathcal{C}_2}\right)\right) \end{aligned} \tag{41}$$

Difference

The behavior of the Difference operator has a straightforward structural definition: $N\left(dp^{\mathcal{C}_1} - dp^{\mathcal{C}_2}\right) = N\left(dp^{\mathcal{C}_1}\right) - N\left(dp^{\mathcal{C}_2}\right)$.

Cardinality estimation follows similar steps to those of the intersection's cardinality estimation.

1. Compute the estructural estimation of the union between Data Pilots using the formula: $N\left(dp^{\mathcal{C}_1} - dp^{\mathcal{C}_2}\right) = N\left(dp^{\mathcal{C}_1}\right) - N\left(dp^{\mathcal{C}_2}\right)$.
2. Use Equation 42 to compute the counters of all paths in $N\left(dp^{\mathcal{C}_1} - dp^{\mathcal{C}_2}\right) - \{\epsilon\}$.

$$\begin{aligned} & c_t\left(dp^{\mathcal{C}_1} - dp^{\mathcal{C}_2}, p\right) = \max\Bigg(\\ & 0, c_t\left(dp^{\mathcal{C}_1}\right) - c_t\left(dp^{\mathcal{C}_2}\right), \left\lceil \frac{c_t\left(dp^{\mathcal{C}_1}\right)}{Card(dp^{\mathcal{C}_1}, \epsilon)} * \max_{q \in N(dp^{\mathcal{C}_1}) - N(dp^{\mathcal{C}_2})} \left(Card(dp^{\mathcal{C}_1}, q)\right)\right\rceil \\ & \Bigg) \end{aligned} \tag{42}$$

3. Compute the cardinality of the Data Pilot using Equation 43.

$$\begin{aligned} Card(dp^{\mathcal{C}_1} - dp^{\mathcal{C}_2}, \epsilon) = \text{NumNED}\Big(dp^{\mathcal{C}_1} - dp^{\mathcal{C}_2}\Big) + \\ \max\left(\text{NumED}\Big(dp^{\mathcal{C}_1}\Big) - \text{NumED}\Big(dp^{\mathcal{C}_2}\Big), 0\right) \end{aligned} \tag{43}$$

Union

The behavior of the union operator has a straightforward structural definition: $N\left(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\right) = N\left(dp^{\mathcal{C}_1}\right) \cup N\left(dp^{\mathcal{C}_2}\right)$.

Cardinality estimation follows similar steps to those of the intersection's cardinality estimation. Updating counters is, however, simpler as the usual definition of union of multisets can be used. The steps are as follows:

1. Compute the estructural estimation of the union between Data Pilots using the formula: $N\left(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\right) = N\left(dp^{\mathcal{C}_1}\right) \cup N\left(dp^{\mathcal{C}_2}\right)$.

2. Use Equation 44 to compute the counters of all paths in $N\left(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\right) - \{\epsilon\}$.

$$c_t\left(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\right) = \max\left(c_t\left(dp^{\mathcal{C}_1}\right), c_t\left(dp^{\mathcal{C}_2}\right)\right) \tag{44}$$

3. Compute the cardinality of the Data Pilot using Equation 45.

$$\begin{aligned} Card(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}, \epsilon) = \text{NumNED}\Big(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\Big) + \\ \max\left(\text{NumED}\Big(dp^{\mathcal{C}_1}\Big), \text{NumED}\Big(dp^{\mathcal{C}_2}\Big)\right) \end{aligned} \tag{45}$$

Join variants

Similarly to the algebra of documents, join variants are derived operators defined as follows:

$$dp^{\mathcal{C}_1} \ltimes_\alpha dp^{\mathcal{C}_2} = \pi^{\oslash}_{dp^{\mathcal{C}_1}}\left(dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}\right) \qquad \text{(Left semijoin)}$$

$$dp^{\mathcal{C}_1} \overline{\ltimes}_\alpha dp^{\mathcal{C}_2} = dp^{\mathcal{C}_1} - \left(dp^{\mathcal{C}_1} \ltimes dp^{\mathcal{C}_2}\right) \qquad \text{(Left antijoin)}$$

$$dp^{\mathcal{C}_1} ⟕_\alpha dp^{\mathcal{C}_2} = \left(dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}\right) \cup \left(dp^{\mathcal{C}_1} \overline{\ltimes}_\alpha dp^{\mathcal{C}_2}\right) \qquad \text{(Left outer join)}$$

Lookup

Similarly to the algebra of documents, lookup is a derived operator defined as follows:

$$dp^{\mathcal{C}_1} \mathbin{\widehat{⟕}}_l^{\alpha} dp^{\mathcal{C}_2} = {}_{dp^{\mathcal{C}_1}}\gamma_{l:\text{arragg}\left(dp^{\mathcal{C}_2}\right)} \left(dp^{\mathcal{C}_1} ⟕_{\alpha} dp^{\mathcal{C}_2}\right)$$

4 Data Pilot algebra properties

This section explores properties of the operators from the Data Pilot algebra that intend to mimic the behavior of properties from the Document algebra. However, certain properties from the document algebra that depend on the data itself do not have a parallel in the algebra of Data Pilots. An example of this is Lemma 3.22. Others like Lemma 3.21, although not explicitely mentioned in the Data Pilot algebra, still apply.

Lemma 4.1 alludes to Lemma 3.9. Lemma 4.2 alludes to Lemma 3.10. Lemma 4.3 alludes to Lemma 3.11. Lemma 4.4 alludes to Lemma 3.12. Lemma 4.6 alludes to Lemma 3.14. Lemma 4.7 alludes to Lemma 3.15. Lemma 4.8 alludes to Lemma 3.16. Lemma 4.9 alludes to Lemma 3.17. Lemma 4.10 alludes to Lemma 3.18. Lemma 4.11 alludes to Lemma 3.20. Lemma 4.12 alludes to Lemma 3.25. Lemma 4.13 alludes to Lemma 3.26. Lemma 4.14 alludes to Lemma 3.27. Lemma 4.15 alludes to Lemma 3.28. Lemma 4.16 alludes to Lemma 3.29. Lemma 4.17 alludes to Lemma 3.30. Lemma 4.18 alludes to Lemma 3.31. Lemma 4.19 alludes to Lemma 3.33. Lemma 4.20 alludes to Lemma 3.34. Lemma 4.21 alludes to Lemma 3.35. Lemma 4.22 alludes to Lemma 3.36. Lemma 4.23 alludes to Lemma 3.37. Lemma 4.24 alludes to Lemma 3.38. Lemma 4.25 alludes to Lemma 3.39. Lemma 4.26 alludes to Lemma 3.40.

Lemma 4.1 (Projection simplification)**.** Let $\mathcal{C}$ be a collection and P and Q be sets of paths. If $P \subseteq Q$, then $\pi_P^{\oslash}(\pi_Q^{\oslash}(dp^{\mathcal{C}})) = \pi_P^{\oslash}(dp^{\mathcal{C}})$.

Lemma 4.2 (Selection makes projection non-preserving on absent path rejection)**.** Let P be a set of paths and α a selection predicate. If α rejects absent paths in P, then $\sigma_\alpha(\pi_P(dp^{\mathcal{C}})) = \sigma_\alpha(\pi_P^{\oslash}(dp^{\mathcal{C}}))$.

Lemma 4.3 (Unnest makes projection non-preserving based on prefixes)**.** Let P be a set of paths and q be a path other than ϵ. If there is at at least one path $p \in P$ for which $q \preceq p$, then $\mu_q^{\oslash}(\pi_P(dp^{\mathcal{C}})) = \mu_p^{\oslash}(\pi_P^{\oslash}(dp^{\mathcal{C}}))$.

Lemma 4.4 (Non-preserving projection dominates preseving projection)**.** Let P and Q be sets of paths. If $Q \subseteq P$, then $\pi_Q^{\oslash}(\pi_P(dp^{\mathcal{C}})) = \pi_Q^{\oslash}(\pi_P^{\oslash}(dp^{\mathcal{C}}))$.

Lemma 4.5 (Projection push-down on unwind)**.** Let p and x be paths. Then $\pi_{p\cdot x}^{\oslash}(\mu_p^{\oslash}(dp^{\mathcal{C}})) = \mu_p^{\oslash}(\pi_{p\cdot *\cdot x}^{\oslash}(dp^{\mathcal{C}}))$

Lemma 4.6 (Projection push-down on join)**.** Let P be a set of paths. If there exists a subset Q of $N\left(dp^{\mathcal{A}}\right)$ such that $Q \subseteq P$ then

$$\pi_P(dp^{\mathcal{A}} \bowtie_\alpha dp^{\mathcal{B}}) = \pi_P(\pi_{Q\cup \text{exprpaths}(\alpha)}(dp^{\mathcal{A}}) \bowtie_\alpha dp^{\mathcal{B}})$$

Likewise, if $Q \subseteq N\left(dp^{\mathcal{B}}\right)$ and $Q \subseteq P$, then

$$\pi_P(dp^{\mathcal{A}} \bowtie_\alpha dp^{\mathcal{B}}) = \pi_P(dp^{\mathcal{A}} \bowtie_\alpha \pi_{Q\cup \text{exprpaths}(\alpha)}(dp^{\mathcal{B}}))$$

Lemma 4.7 (Projection is distributive over join)**.** Let $dp^{\mathcal{C}_1}, dp^{\mathcal{C}_2}$ be Data Pilots, α a predicate and P, Q, R sets of paths such that $P \subseteq N\left(dp^{\mathcal{C}_1}\right) \cup N\left(dp^{\mathcal{C}_2}\right)$. If $Q = \left(P \cap N\left(dp^{\mathcal{C}_1}\right)\right) \cup \left(N\left(dp^{\mathcal{C}_1}\right) \cap N\left(dp^{\mathcal{C}_2}\right)\right)$ and $R = \left(P \cap N\left(dp^{\mathcal{C}_2}\right)\right) \cup \left(N\left(dp^{\mathcal{C}_1}\right) \cap N\left(dp^{\mathcal{C}_2}\right)\right)$, then $\pi_P^{\oslash}\left(dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}\right) = \pi_P^{\oslash}\left(\pi_Q^{\oslash}\left(dp^{\mathcal{C}_1}\right) \bowtie_\alpha \pi_R^{\oslash}\left(dp^{\mathcal{C}_2}\right)\right)$ and $\pi_P\left(dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}\right) = \pi_P\left(\pi_Q\left(dp^{\mathcal{C}_1}\right) \bowtie_\alpha \pi_R\left(dp^{\mathcal{C}_2}\right)\right)$.

Lemma 4.8 (Projection push-down on lookup)**.** Let $dp^{\mathcal{C}_1}$, $dp^{\mathcal{C}_2}$ be Data Pilots, α a predicate, l a label and P a set of paths. If there exists some subset Q of $N\left(dp^{\mathcal{C}_2}\right)$ such that $\{l \cdot * \cdot q \mid q \in Q\} \subseteq P$, then

$$\pi_P^{\oslash}(dp^{\mathcal{C}_1} \mathbin{\widehat{⟕}}_\alpha^l dp^{\mathcal{C}_2}) = \pi_P^{\oslash}(dp^{\mathcal{C}_1} \mathbin{\widehat{⟕}}_\alpha^l \pi_{Q\cup \text{exprpaths}(\alpha)}^{\oslash}(dp^{\mathcal{C}_2}))$$

Lemma 4.9 (Projection is distributive over lookup)**.** Let $dp^{\mathcal{C}_1}$, $dp^{\mathcal{C}_2}$ be Data Pilots, α a predicate, l a label and P, Q sets of paths such that $Q \subseteq N\left(dp^{\mathcal{C}_1}\right)$. If there exists some subset R of $N\left(dp^{\mathcal{C}_2}\right)$ such that $\{l \cdot * \cdot r \mid r \in R\} \cup Q \subseteq P$, then

$$\pi^{\oslash}_{P}(dp^{\mathcal{C}_1} \hat{\bowtie}^{l}_{\alpha} dp^{\mathcal{C}_2}) = \pi^{\oslash}_{Q \cup \mathrm{exprpaths}(\alpha)}(dp^{\mathcal{C}_1}) \hat{\bowtie}^{l}_{\alpha} \pi^{\oslash}_{R \cup \mathrm{exprpaths}(\alpha)}(dp^{\mathcal{C}_2})$$

Lemma 4.10 (Projection push-down on union)**.** Let $dp^{\mathcal{C}_1}$, $dp^{\mathcal{C}_2}$ be Data Pilots and P a set of paths. Then,

$$\pi^{\oslash}_{P}\left(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\right) = \pi^{\oslash}_{P}\left(\mathcal{C}_1\right) \cup \pi^{\oslash}_{P}\left(dp^{\mathcal{C}_2}\right) \text{ and } \pi_{P}\left(dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}\right) = \pi_{P}\left(dp^{\mathcal{C}_1}\right) \cup \pi_{P}\left(dp^{\mathcal{C}_2}\right)$$

Lemma 4.11 (Projection introduction on group)**.** Let P be a set of paths, $q_1, q_2, \ldots q_k$ paths from $N\left(dp^{\mathcal{C}}\right)$, $r_1, r_2, \ldots, r_k$ labels and $f_1, f_2, \ldots, f_k$ Data Pilot aggregates. Then,

$${}_{P}\gamma_{r_1:f_1(q_1), r_2:f_2(q_2), \ldots, r_k:f_k(q_k)}\left(dp^{\mathcal{C}}\right) = {}_{P}\gamma_{r_1:f_1(q_1), r_2:f_2(q_2), \ldots, r_k:f_k(q_k)}(\pi_{P \cup \{q_1, \ldots, q_k\}}(dp^{\mathcal{C}}))$$

Lemma 4.12 (Selection push-down on join)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be Data Pilots, and α, ψ_1 and ψ_2 be predicates. If $\mathrm{exprpaths}(\psi_1) \subseteq N\left(dp^{\mathcal{C}_1}\right)$ and $\mathrm{exprpaths}(\psi_2) \subseteq N\left(dp^{\mathcal{C}_2}\right)$, then

$$\sigma_{\psi_1 \wedge \psi_2}(dp^{\mathcal{C}_1} \bowtie_{\alpha} dp^{\mathcal{C}_2}) = \sigma_{\psi_1}(dp^{\mathcal{C}_1}) \bowtie_{\alpha} \sigma_{\psi_2}(dp^{\mathcal{C}_2})$$

Lemma 4.13 (Selection push-down on set operators)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be collections, ψ a predicate, and $\odot$ an arbitrary operator from the set $\{\cup, \cap, -\}$. Then

$$\sigma_{\psi}\left(dp^{\mathcal{C}_1} \odot dp^{\mathcal{C}_2}\right) = \sigma_{\psi}\left(dp^{\mathcal{C}_1}\right) \odot \sigma_{\psi}\left(dp^{\mathcal{C}_2}\right)$$

Lemma 4.14 (Selection introduction on group)**.** Let $dp^{\mathcal{C}}$ be a collection, $\tilde{F}_1 = (l^1_1 : \mathrm{arragg}(q^1_1), \ldots, l^1_k : \mathrm{arragg}(q^1_k))$ and $\tilde{F}_2 = (l^2_1 : f^2_1(q^2_1), \ldots, l_j : f_j(q_j))$ vectors of aggregates, and P, L, R sets of paths such that $L = \{l^1_1, \ldots, l^1_k\} \cup \{l^2_1, \ldots, l^2_j\}$, $L \subseteq P$, $paths(\tilde{F}_1) \cap paths(\tilde{F}_2) = \emptyset$, and
$R = \{q \cdot x \mid (\exists l \in L)(\exists q \in \{q^1_1, \ldots, q^1_k\})(\exists x \neq \epsilon)[l \cdot * \cdot x \in P \wedge (l : \mathrm{arragg}(q)) \in \tilde{F}_1]\} = \{r_1, \ldots, r_h\}$. Then

$$\pi^{\oslash}_{P}({}_{G}\gamma_{\tilde{F}_1 \cdot \tilde{F}_2}(dp^{\mathcal{C}})) = \pi^{\oslash}_{P}({}_{G}\gamma_{\tilde{F}_1 \cdot \tilde{F}_2}(\sigma_{\exists r_1 \vee \cdots \vee \exists r_h}(dp^{\mathcal{C}})))$$

Lemma 4.15 (Convertion from join to semijoin)**.** Let α be a predicate, and $dp^{\mathcal{C}_1}$, $dp^{\mathcal{C}_2}$ be Data Pilots. Then

$$\pi^{\oslash}_{N\left(dp^{\mathcal{C}_1}\right)}\left(dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}\right) = dp^{\mathcal{C}_1} \ltimes_\alpha dp^{\mathcal{C}_2} \text{ and } \pi^{\oslash}_{N\left(dp^{\mathcal{C}_2}\right)}\left(dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}\right) = dp^{\mathcal{C}_1} \rtimes_\alpha dp^{\mathcal{C}_2}$$

Lemma 4.16 (Conversion of Left Outer Join to Join)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be collections, ψ and α predicates, and $Q = \text{exprpaths}\,(\alpha) \cap N(dp^{\mathcal{C}_2})$. If ψ rejects absent paths in Q, then $\sigma_\psi\left(dp^{\mathcal{C}_1} ⟕_\alpha dp^{\mathcal{C}_2}\right) = dp^{\mathcal{C}_1} \bowtie_\alpha dp^{\mathcal{C}_2}$.

Lemma 4.17 (Conversion of Extended Left Outer Join to Join)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be Data Pilots, ψ and α predicates, and l a non-empty label. If ψ rejects empty objects in $\{l\}$, then $\sigma_\psi\left(dp^{\mathcal{C}_1} ⟕^l_\alpha dp^{\mathcal{C}_2}\right) = dp^{\mathcal{C}_1} \bowtie^l_\alpha dp^{\mathcal{C}_2}$.

Lemma 4.18 (Conversion of Lookup to Nest Join)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be Data Pilots, ψ and α predicates, and l a non-empty a label. If ψ rejects empty arrays in $\{l\}$, then $\sigma_\psi\left(dp^{\mathcal{C}_1} \widehat{⟕}^l_\alpha dp^{\mathcal{C}_2}\right) = dp^{\mathcal{C}_1} \widehat{\bowtie}^l_\alpha dp^{\mathcal{C}_2}$.

Lemma 4.19 (Projection eliminates Left Outer Join)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be collections, an α a predicate. If $\text{NumED}\left(dp^{\mathcal{C}_1}\right) = 0$, then $\pi^{\oslash}_{N\left(dp^{\mathcal{C}_1}\right)}\left(dp^{\mathcal{C}_1} ⟕_\alpha dp^{\mathcal{C}_2}\right) = dp^{\mathcal{C}_1}$.

Lemma 4.20 (Delayed nesting on chained Nest joins)**.** Let $dp^{\mathcal{C}_1}, dp^{\mathcal{C}_2}, \ldots, dp^{\mathcal{C}_k}$ be Data Pilots, $l_{1,2}, l_{2,3}, l_{3,4}, \ldots, l_{k-1,k}$ be labels, and $\alpha_{1,2}$, $\alpha_{2,3}$, $\alpha_{3,4}$, $\ldots$, $\alpha_{k-1,k}$ be predicates.

$$dp^{\mathcal{C}_1} \widehat{\bowtie}^{l_{1,2}}_{\alpha_{1,2}} \left(dp^{\mathcal{C}_2} \widehat{\bowtie}^{l_{2,3}}_{\alpha_{2,3}} \left(\ldots dp^{\mathcal{C}_{k-2}} \widehat{\bowtie}^{l_{k-2,k-1}}_{\alpha_{k-2,k-1}} \left(dp^{\mathcal{C}_{k-1}} \widehat{\bowtie}^{l_{k-1,k}}_{\alpha_{k-1,k}} dp^{\mathcal{C}_k}\right) \ldots\right)\right) =$$

$$\begin{aligned}
&{}_{N\left(dp^{\mathcal{C}_1}\right)}\gamma_{l_{1,2}:\mathrm{arragg}\left(N\left(dp^{\mathcal{C}_2}\right)\cup\{l_{2,3}\}\right)}\Bigg(\\
&\quad {}_{N\left(dp^{\mathcal{C}_1}\right)\cup N\left(dp^{\mathcal{C}_2}\right)}\gamma_{l_{2,3}:\mathrm{arragg}\left(N\left(dp^{\mathcal{C}_3}\right)\cup\{l_{4,5}\}\right)}\Bigg(\\
&\qquad \ldots\\
&\qquad {}_{N\left(dp^{\mathcal{C}_1}\right)\cup\ldots\cup N\left(dp^{\mathcal{C}_{k-2}}\right)}\gamma_{l_{k-2,k-1}:\mathrm{arragg}\left(N\left(dp^{\mathcal{C}_{k-1}}\right)\cup\{l_{k-1,k}\}\right)}\Bigg(\\
&\qquad\quad {}_{N\left(dp^{\mathcal{C}_1}\right)\cup\ldots\cup N\left(dp^{\mathcal{C}_{k-1}}\right)}\gamma_{l_{k-1,k}:\mathrm{arragg}\left(N\left(dp^{\mathcal{C}_k}\right)\right)}\Bigg(\\
&\qquad\qquad dp^{\mathcal{C}_1} \bowtie_{\alpha_{1,2}} \left(dp^{\mathcal{C}_2} \bowtie_{\alpha_{2,3}} \left(\ldots dp^{\mathcal{C}_{k-2}} \bowtie_{\alpha_{k-2,k-1}} \left(dp^{\mathcal{C}_{k-1}} \bowtie_{\alpha_{k-1,k}} dp^{\mathcal{C}_k}\right) \ldots\right)\right)\\
&\qquad\quad \Bigg)\\
&\qquad \Bigg)\\
&\qquad \ldots\\
&\quad \Bigg)\\
&\Bigg)
\end{aligned}$$

Lemma 4.21 (Left Outer Join - Inner Join Reordering with GOJ)**.** Let $dp^{\mathcal{C}_1}$, $dp^{\mathcal{C}_2}$ and $dp^{\mathcal{C}_3}$ be collections, and α and β be predicates. If β rejects absent paths on $N\left(dp^{\mathcal{C}_2}\right)$, then

$$\begin{aligned}
&dp^{\mathcal{C}_1} ⟕_{\alpha} \left(dp^{\mathcal{C}_2} \bowtie_{\beta} dp^{\mathcal{C}_3}\right) =\\
&\quad \left(dp^{\mathcal{C}_1} ⟕_{\alpha} dp^{\mathcal{C}_2}\right) \overset{\mathrm{goj}}{\bowtie} [\beta,\, N\left(dp^{\mathcal{C}_1}\right)]\, dp^{\mathcal{C}_3}
\end{aligned}$$

Lemma 4.22 (Lookup - Inner Join Reordering with GOJ)**.** Let $dp^{\mathcal{C}_1}$, $dp^{\mathcal{C}_2}$ and $dp^{\mathcal{C}_3}$ be collections, and α and β be predicates. If β rejects absent paths on $N\left(dp^{\mathcal{C}_2}\right)$, then

$$\begin{aligned}
&dp^{\mathcal{C}_1} \widehat{⟕}^{l}_{\alpha} \left(dp^{\mathcal{C}_2} \bowtie_{\beta} dp^{\mathcal{C}_3}\right) =\\
&\quad {}_{N\left(dp^{\mathcal{C}_1}\right)}\gamma_{l:\mathrm{arragg}\left(N\left(dp^{\mathcal{C}_2}\right)\cup N\left(dp^{\mathcal{C}_3}\right)\right)} \left(\left(dp^{\mathcal{C}_1} ⟕_{\alpha} dp^{\mathcal{C}_2}\right) \overset{\mathrm{goj}}{\bowtie} \left[\beta,\, N\left(dp^{\mathcal{C}_1}\right)\right]\, dp^{\mathcal{C}_3}\right)
\end{aligned}$$

Lemma 4.23 (Full Group push-down over Selection)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot of some collection $\mathcal{C}$, $l_1, l_2, \ldots, l_k$ labels not realized in $dp^{\mathcal{C}}$, $q_1, \ldots, q_k$ paths realized in $dp^{\mathcal{C}}$, and α a selection predicate.

$$_{\text{exprpaths}(\alpha)}\gamma_{l_1:f_1(q_1),\ldots,l_k:f_k(q_1)}\left(\sigma_\alpha\left(dp^{\mathcal{C}}\right)\right) = \sigma_\alpha\left(_{\text{exprpaths}(\alpha)}\gamma_{l_1:f_1(q_1),\ldots,l_k:f_k(q_k)}\left(dp^{\mathcal{C}}\right)\right)$$

Lemma 4.24 (Partial Group push-down over Selection)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot, G a set of paths to group by, $l_1, \ldots, l_k, l'_1, \ldots, l'_k$ labels not realized in $dp^{\mathcal{C}}$, $q_1, \ldots, q_k$ paths realized in $dp^{\mathcal{C}}$, α a selection predicate and ξ the function:

$$\xi(G, \alpha) = \begin{cases} \text{exprpaths}\,(\alpha) & \text{if } G \subset \text{exprpaths}\,(\alpha) \\ G \cup \text{exprpaths}\,(\alpha) & \text{if } G \not\subseteq \text{exprpaths}\,(\alpha) \end{cases}$$

then

$$\begin{aligned} &_G\gamma_{l_1:f_1(q_1),\ldots,l_k:f_k(q_k)}\left(\sigma_\alpha\left(dp^{\mathcal{C}}\right)\right) = \\ &\quad _G\gamma_{l_1:f_1(l'_1),\ldots,l_k:f_k(l'_k)}\left(\sigma_\alpha\left(_{\xi(G,\alpha)}\gamma_{l'_1:f_1(q_1),\ldots,l'_k:f_k(q_k)}\left(dp^{\mathcal{C}}\right)\right)\right) \end{aligned}$$

Lemma 4.25 (Unnest simplification due to Group)**.** Let $dp^{\mathcal{C}}$ be a collection, G a set of paths to group by, p a path in $N\left(dp^{\mathcal{C}}\right)$, l a label not in $N\left(dp^{\mathcal{C}}\right)$ and agg an aggregate function. If there exists a non-aggregate function f that treats constituents of arrays as a bag of values and performs the same operation as agg would over the same bag of values, then $_G\gamma_{l:agg(p)}\left(\mu_p^{\oslash}\left(dp^{\mathcal{C}}\right)\right) = {_G\gamma_{l:agg(f(p))}}\,(\mathcal{C})$.

Lemma 4.26 (Partial Group push-down over Left Outer Join)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be Data Pilots, α a predicate, G a set of paths to group by, and $\tilde{F}$ a vector of scalar aggregate functions such that $\tilde{F}$ can be split into $\tilde{F}_1$ and $\tilde{F}_2$; and G can be split into $G_1 = G \cap N\left(dp^{\mathcal{C}_1}\right)$ and $G_2 = G \cap N\left(dp^{\mathcal{C}_2}\right)$. Then

$$_G\gamma_{\tilde{F}}(dp^{\mathcal{C}_1} ⟕_\alpha dp^{\mathcal{C}_2}) = {_G\gamma_{(\tilde{F}_1 \otimes (\exists c_2 ? c_2 : 1)) \cdot \tilde{F}'_2}}(dp^{\mathcal{C}_1} ⟕_\alpha {_{G_2 \cup \text{exprpaths}(\alpha)}\gamma_{\tilde{F}_2 \cdot (\!(c_2 : \text{count}(*))\!)}}(dp^{\mathcal{C}_2}))$$

It is important to note that all operations performed on Data Pilots using operators from the Data Pilot algebra adhere to a conservative approach. This approach

ensures that paths are not removed from or added to the input collection's estimated cover unless it is 100% certain that they should.

This concept is formalized in Definition 4.1.

Definition 4.1 (Conservative operation)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot, Ω an operation over collections, and Ψ the corresponding operation of Ω over Data Pilots. Ψ is considered conservative regarding $dp^{\mathcal{C}}$ if the Data Pilot $\Psi\left(dp^{\mathcal{C}}\right)$ satisfies either of the following conditions:

1. If $dp^{\mathcal{C}}$ is faithful, any path in $N\left(\Psi\left(dp^{\mathcal{C}}\right)\right)$ that does not approximate a path in $\text{cover}\left(\Omega\left(\mathcal{C}\right)\right)$ must exist in $N\left(dp^{\mathcal{C}}\right)$.
2. If $dp^{\mathcal{C}}$ is not faithful, there must exist a faithful Data Pilot $dp^{\mathcal{C}'}$ and a sequence of conservative Data Pilot operations $\Psi_1, \Psi_2, \ldots, \Psi_n$ such that $dp^{\mathcal{C}} = \Psi_n\left(\ldots \Psi_2\left(\Psi_1\left(dp^{\mathcal{C}'}\right)\right)\ldots\right)$, and for any path p in $N\left(\Psi\left(dp^{\mathcal{C}}\right)\right)$ that does not approximate a path in $\text{cover}\left(\Omega\left(\mathcal{C}\right)\right)$, there exists an $i \in [1, n]$ such that $p \overset{r}{\backsim} \Psi_i\left(\Psi_{i-1}\left(\ldots \Psi_1\left(dp^{\mathcal{C}'}\right)\ldots\right)\right)$.

□

Conservative operations ensure the maintenance of the forth property, indicating that while there may be "false positives" (i.e., estimating that a path exists when it does not), "false negatives" (i.e., estimating that a path does not exist when it does) are prevented. This is behavior is similar to that of "Bloom Filters". Avoiding false negatives is crucial for estimates, as a false negative could prematurely suggest that a query will result in an empty collection.

Moreover, conditions 1 and 2 outlined in Definition 4.1 serve to minimize false positives by prohibiting operations that forfeit the back property from generating any path not approximating a path in the collection cover of the pertinent Data Pilot's collection. Essentially, a conservative operation on a Data Pilot removes paths with absolute certainty if they are to be eliminated from the respective collection cover

(e.g., projection), retains paths with absolute certainty if they should be retained or if uncertainty exists regarding their removal (e.g., selection), and introduces paths only if there is absolute certainty of their creation in the corresponding collection cover (e.g., unnest).

Theorem 4.27 (Every Data Pilot operator constitutes a conservative operation)**.** Let $dp^{\mathcal{C}}$ be a faithful Data Pilot and Ψ an operation with an underlying operator from the Data Pilot algebra, then Ψ is conservative with respect to $dp^{\mathcal{C}}$.

Under specific conditions, there are operations whose results respect the back property when processing a faithful Data Pilot. This is further explained in Lemmas 4.28, 4.30, and 4.29.

Lemma 4.28 (Preserving projection is faithful)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot of collection $\mathcal{C}$, and P a set of path expressions. If $dp^{\mathcal{C}}$ is faithful and no path $p \in arrs\left(P\right)$ exists such that $p \overset{r}{\backsim} N\left(dp^{\mathcal{C}}\right)$, then $\pi_P\left(dp^{\mathcal{C}}\right)$ is faithful.

Lemma 4.29 (Cartesian product is faithful)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be Data Pilots of collections $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. If both $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ are faithful, then $dp^{\mathcal{C}_1} \times dp^{\mathcal{C}_2}$ is faithful.

Lemma 4.30 (Union is faithful)**.** Let $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ be Data Pilots of collections $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. If both $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ are faithful, then $dp^{\mathcal{C}_1} \cup dp^{\mathcal{C}_2}$ is faithful.

Lemma 4.31 (Unnest is faithful)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot of collection $\mathcal{C}$ and p a path expression. If $dp^{\mathcal{C}}$ is faithful, then $\mu_p\left(dp^{\mathcal{C}}\right)$ is faithful.

Lemma 4.32 (Group is faithful)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot of collection $\mathcal{C}$, G a set of path expressions realized in $dp^{\mathcal{C}}$ and $\langle\!\langle F \rangle\!\rangle$ an aggregate vector whose non-scalar aggregate functions comprise arragg and arrconcat, then ${}_G\gamma_{\langle\!\langle F \rangle\!\rangle}\left(dp^{\mathcal{C}}\right)$.

There are also conservative operations that are guaranteed to violate the back property. These are discussed in Lemma 4.33.

Lemma 4.33 (Non-faithful operators preserve forth property)**.** Let $dp^{\mathcal{C}}$ be a Data Pilot of collection $\mathcal{C}$, and Ψ an operation over Data Pilots. If the underlying operator of Ψ is a selection, join (or any of its derivatives), difference, or intersection, then $\Psi\left(dp^{\mathcal{C}}\right)$ loses the back property but retains the forth property.

5 Optimization hints

This section introduces the dynamics (join-simplifaction, reordering of operators, etc.) used to pass down *advices*, or optimization hints, to the optimizer. Intuitevely, the idea is simple: Given a query q, modify q when convenient based on the cardinality estimates produced by the data pilots of collections involved in q aiming to reduce the size of intermediate results and provide the optimizer with more flexibility to choose candidate implementations of an operator. For instance, an optimizer has fixed outer and inner collections in a left outer join, but is free to choose the inner and outer collections in a regular join.

An optimization hint is more formally described in Definition 5.1.

Definition 5.1 (Optimization hint)**.** An optimization hint is a pair (H, E), where H is an algebraic equivalence referring to a property of the document algebra and E a number from $\mathbb{N}^0$ referring to a cardinality estimate computed from the information stored in the Data Pilot. □

Throughout the section, this manuscript incorporates multiple diagrams, each representing a property from the document algebra. These visual aids, denoted as *cardinality estimation diagrams*, provide a visual depiction of the anticipated number of documents entering and exiting operators before and after the application of a specific document algebra property. These diagrams closely resemble query plans at the logical level, but instead of showing a query tree, they aim to illustrate the direction in which data flows together with its expected quantity. The source of each arrow marks the data source and the arrow label the expected number of documents

the target is to receive. The word *access* marks the retrieval of documents from a collection (what a physical plan would typically name a scan, be it through an index or not).

Commencing with Figure 43, the diagram is based on Lemma 3.13. By performing an non-preserving projection beforehand, the count of elements to unnest changes from $Card(dp^{\mathcal{C}}, \epsilon)$ to $c_{\text{arr}}(dp^{\mathcal{C}}, p)$, potentially benefiting the unwind operation in two ways:

$$\text{(Before push-down)} \qquad \text{access } \mathcal{C} \xrightarrow{Card(dp^{\mathcal{C}},\epsilon)} \mu_p^{\oslash} \xrightarrow{Card(dp^{\mathcal{C}},p\cdot\star)} \pi_{p\cdot x}^{\oslash} \xrightarrow{Card(dp^{\mathcal{C}},p\cdot\star\cdot x)}$$

$$\text{(After push-down)} \qquad \text{access } \mathcal{C} \xrightarrow{Card(dp^{\mathcal{C}},\epsilon)} \pi_{p\cdot\star\cdot x}^{\oslash} \xrightarrow{c_{\text{arr}}(dp^{\mathcal{C}},p)} \mu_p^{\oslash} \xrightarrow{Card(dp^{\mathcal{C}},p\cdot\star\cdot x)}$$

Figure 43. Cardinality estimation diagram of Lemma 3.13

1. Since the projection removes any document where the path $p \cdot \star \cdot x$ is missing, all documents where p is not an array are removed. Thus, unnest can assume that p exists on all its input and must only check if p is an empty array or not.

2. The input size decreases, reducing the size of intermediate results.

Moreover, if $Card(dp^{\mathcal{C}}, p \cdot \star \cdot x)$ is considerably smaller than $Card(dp^{\mathcal{C}}, \epsilon)$, introducing a selection before the projection and transforming the un-preserving operators into preserving ones can be beneficial from an implementation standpoint. These transformations reduce conditionals at the code level, thereby minimizing branching. For instance, consider two paths p and q, and the query described in Equation 46.

$$\begin{aligned} \pi_{p\cdot q}^{\oslash}(\mu_p^{\oslash}(\mathcal{C})) &= \mu_p^{\oslash}(\pi_{p\cdot*\cdot q}^{\oslash}(\mathcal{C})) \\ &= \mu_p^{\oslash}(\pi_{p\cdot*\cdot q}(\sigma_{\exists p\cdot*\cdot q}(\mathcal{C}))) \\ &= \mu_p(\pi_{p\cdot*\cdot q}(\sigma_{\exists p\cdot*\cdot q}(\mathcal{C}))) \end{aligned} \tag{46}$$

By ensuring that the selection with an existential predicate is the first operator evaluated in the query, the optimizer can take advantage of a faithful data pilot of $\mathcal{C}$.

Assuming one is available, the optimizer can declare an early-stopping criterion on the loop that scans either an index or a collection itself. This criterion would consider that a maximum of $Card(dp^{\mathcal{C}}, p)$ documents are to be retrieved and a maximum of $Card(dp^{\mathcal{C}}, p \cdot \star \cdot q)$ are to be projected.

For the case of joins, a projection push-down is done similarly to relational algebra, but instead of relying on the fixed schemas of tables, it relies on Data Pilots. If all paths within a set of projected paths following a join are present in the data pilot of one of the input collections, the projection can be performed on that specific collection beforehand as described in Lemma 3.14. Figure 44 shows the cardinality estimations at each operation.

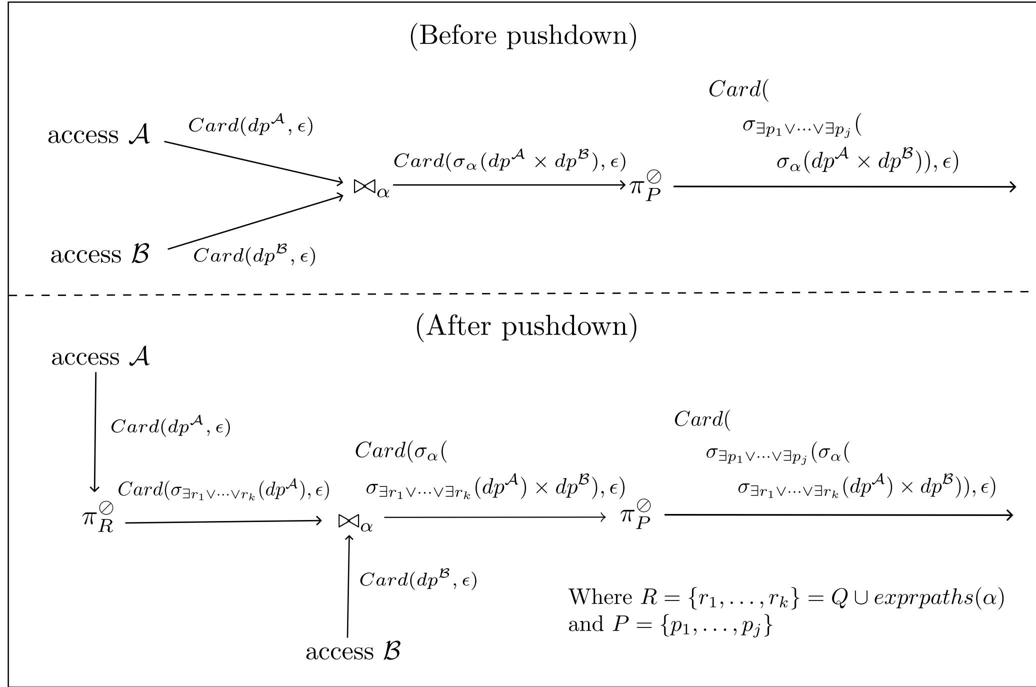


Figure 44. Cardinality estimation diagram of Lemma 3.14

A selection can be pushed down to one or both arguments of a join, multiset intersection or multiset difference. For the case of union, selections must be pushed down to both arguments.

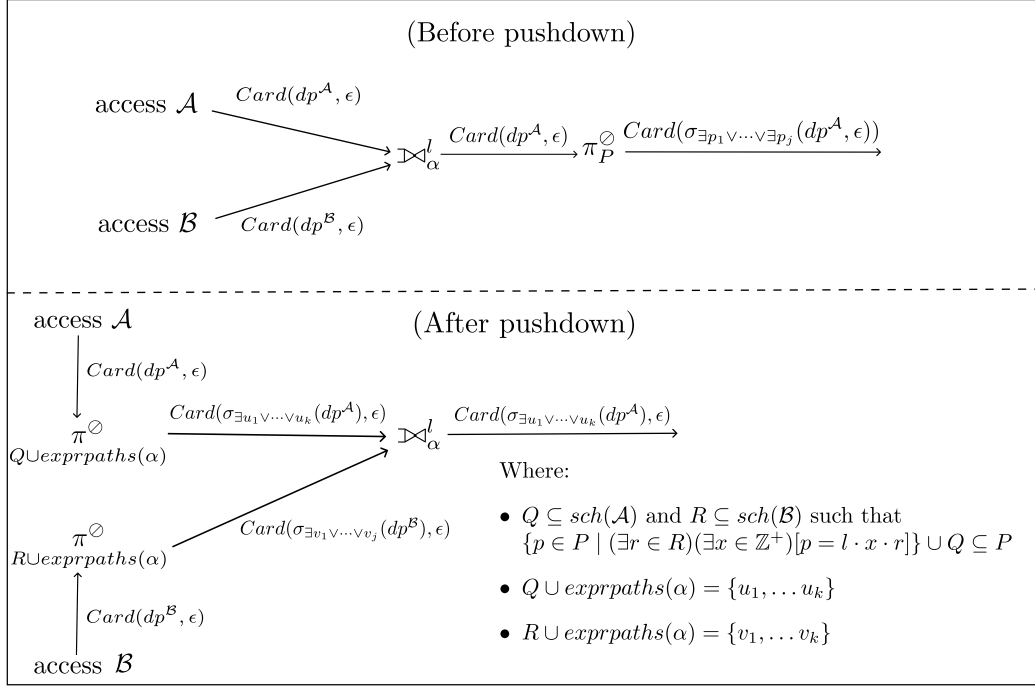


Figure 45. Cardinality estimation diagram of Lemma 3.16

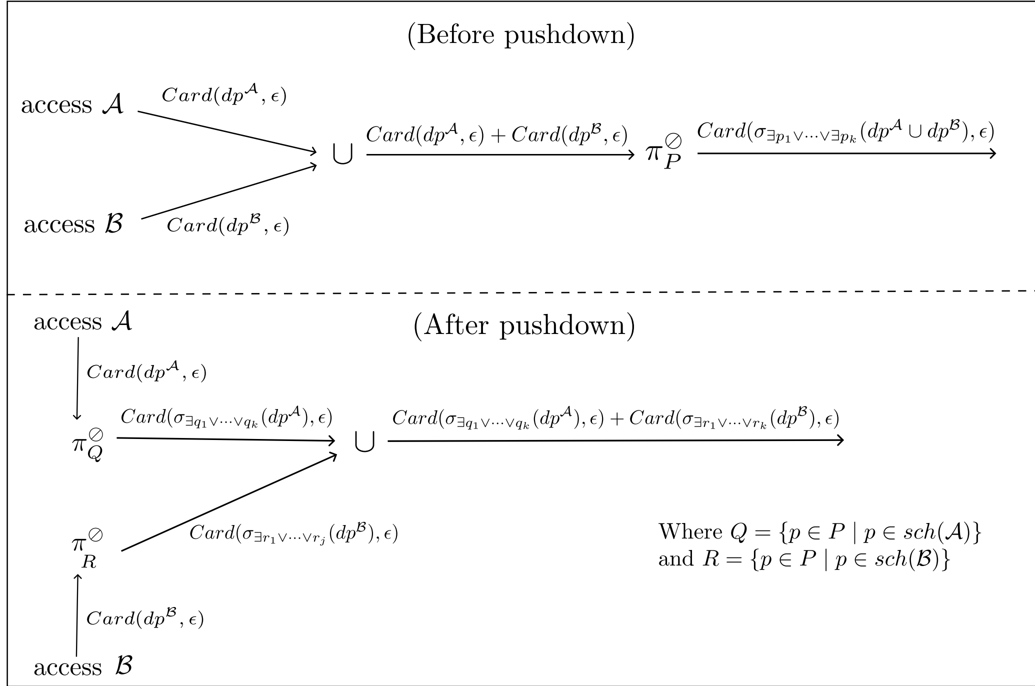


Figure 46. Cardinality estimation diagram of Lemma 3.18

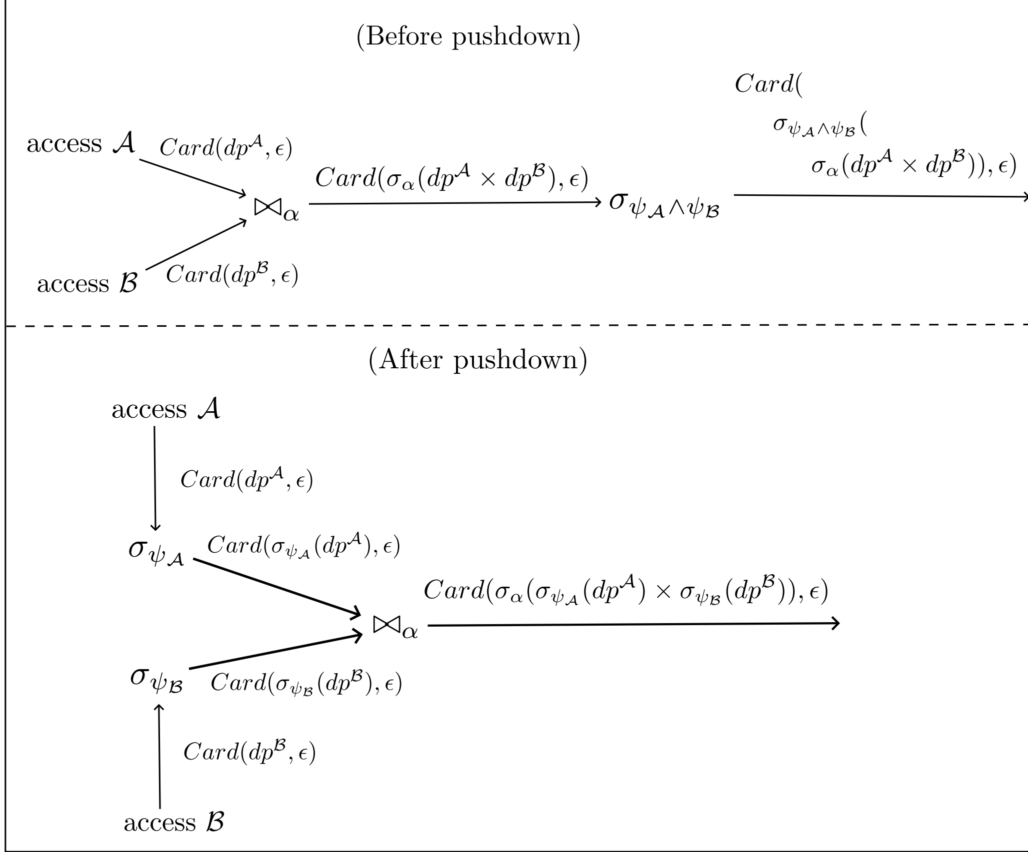


Figure 47. Cardinality estimation diagram of Lemma 3.25

Collection volume

When reordering operators, there are situations where the most significant performance improvements result from reducing the number of intermediate results generated at a specific operation. This section introduces a metric known as "Collection volume" and outlines its utility in assessing the viability of an optimization. Essentially, the collection volume represents the weighted sum of the cardinality of all paths within the collection's data pilot. The underlying idea behind this metric is that the more frequently a path appears in a collection's documents, the more significance or weight it carries. Think of it as an abstract representation of a collection's memory footprint in comparison to another collection. This abstraction does not consider the actual memory sizes of types since they can vary across different database management systems. Instead, when deciding whether to suggest an optimization to the

optimizer, the metric is computed both before and after applying the optimization to gauge how the collection's volume changes at a specific operation.

Definition 5.2 provides a method to calculate a collection's volume utilizing its data pilot, as expressed in Equation 47. This computation relies on the path weight, as defined recursively in Equation 48. In essence, this weight distinguishes between two scenarios: whether a path is within an array (indicated by the presence of the label $\star$ in the path) or not.

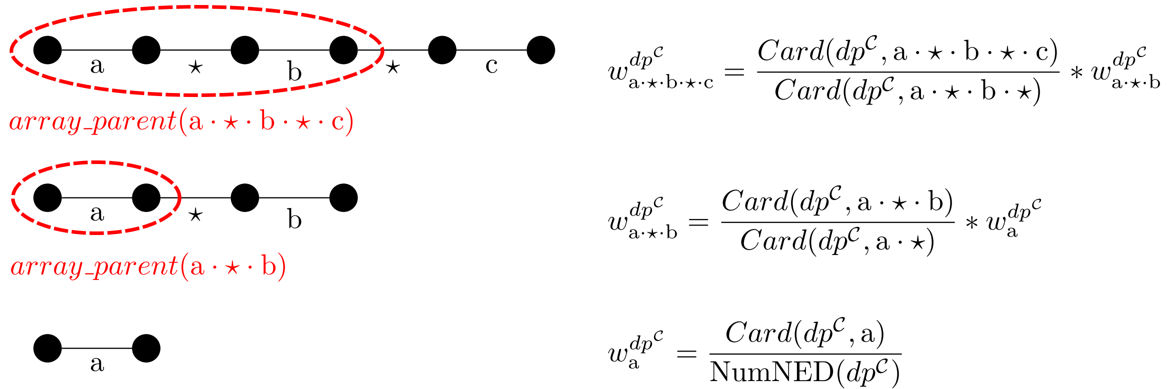
When the path is not within an array, the weight is determined by the percentage of non-empty documents in which the path is found. This percentage is calculated as the ratio between the cardinality of the path and the number of non-empty documents in the collection, denoted by $\text{NumNED}\left(dp^{\mathcal{C}}\right)$.

In situations involving arrays, some documents may contain a number of array elements nested to a greater extent than the total number of documents in the collection. In such instances, unnesting operations could become computationally expensive without a prior selection. Therefore, the weight of paths within arrays should account for both the number of elements within the parent array and the cardinality of the parent array concerning the entire collection. This concept is the base of the recursion in Equation 48, with an illustrative example provided in Figure 48.

Definition 5.2 (Collection Volume)**.** Let $dp^{\mathcal{C}}$ be a DataPilot of collection $\mathcal{C}$, p be a path of $dp^{\mathcal{C}}$ and $array_parent(p)$ denote a prefix q of p such that there exists some sequence of labels r with $\star \notin r$ that satisfies the equality $p = q.\star.r$. The volume of collection $\mathcal{C}$ based on $dp^{\mathcal{C}}$ is expressed by Equation 47.

$$vol(dp^{\mathcal{C}}) = \sum_{p \in explicit(N(dp^{\mathcal{C}}))} \left(w_p^{dp^{\mathcal{C}}} * Card(dp^{\mathcal{C}}, p) \right) \tag{47}$$

$$w_p^{dp^{\mathcal{C}}} = \begin{cases} \dfrac{Card(dp^{\mathcal{C}}, p)}{\text{NumNED}(dp^{\mathcal{C}})} & \text{if } \star \notin p \\ \dfrac{Card(dp^{\mathcal{C}}, p)}{Card(dp^{\mathcal{C}}, array_parent(p) \cdot \star)} * w_{array_parent(p)}^{dp^{\mathcal{C}}} & \text{otherwise} \end{cases} \tag{48}$$



$$w^{dp^{\mathcal{C}}}_{\text{a}\cdot\star\cdot\text{b}\cdot\star\cdot\text{c}} = \frac{Card(dp^{\mathcal{C}}, \text{a}\cdot\star\cdot\text{b}\cdot\star\cdot\text{c})}{Card(dp^{\mathcal{C}}, \text{a}\cdot\star\cdot\text{b}\cdot\star)} * w^{dp^{\mathcal{C}}}_{\text{a}\cdot\star\cdot\text{b}}$$

$$w^{dp^{\mathcal{C}}}_{\text{a}\cdot\star\cdot\text{b}} = \frac{Card(dp^{\mathcal{C}}, \text{a}\cdot\star\cdot\text{b})}{Card(dp^{\mathcal{C}}, \text{a}\cdot\star)} * w^{dp^{\mathcal{C}}}_{\text{a}}$$

$$w^{dp^{\mathcal{C}}}_{\text{a}} = \frac{Card(dp^{\mathcal{C}}, \text{a})}{\text{NumNED}(dp^{\mathcal{C}})}$$

Figure 48. An illustration of the weight computed from a path within an array, in accordance with Definition 5.2.

Definition 5.3 (Faithful Collection Volume)**.** The volume of a collection $\mathcal{C}$ is faithful if it is computed from a faithful DataPilot of $\mathcal{C}$.

To determine whether an optimization should be recommended to the optimizer, the percentage decrease in both input and output volumes associated with a specific operator is calculated as follows:

1. Select a candidate optimization for analysis, based on the given query. For instance, if a join precedes a projection, “projection push-down over join” (Lemma 3.14) is a candidate optimization.

2. Identify pipeline-breaker operators influenced by changes in volume either in the collections they receive as input or in the collection they produce as output. Following the example from the previous step, “join” is the operator affected by a change in the volume of the two collections passed to it.

3. Estimate the input and/or output volumes of pipeline-breaker operators affected by the optimization. For input collections, calculate the total input volume by summing their volumes. For output collections, calculate the volume as usual.

4. Apply the optimization to the query and estimate the input and output volumes of pipeline-breaker operators impacted by the optimization.

5. Calculate the percentage decrease in input and output volume for the pipeline-breaker operators influenced by the optimization.

The optimization is then recommended to the optimizer if any of the calculated percentages exceeds a user-defined threshold.

Example 5.1. Consider two data pilots $dp^{\mathcal{C}_1}$ and $dp^{\mathcal{C}_2}$ with the paths and counters shown in Table 13.

DataPilot	Path	Cardinality
$dp^{\mathcal{C}_1}$	ϵ	40
$dp^{\mathcal{C}_1}$	a	40
$dp^{\mathcal{C}_1}$	b	30
$dp^{\mathcal{C}_1}$	d	15
$dp^{\mathcal{C}_1}$	e	14
$dp^{\mathcal{C}_2}$	ϵ	20
$dp^{\mathcal{C}_2}$	f	5
$dp^{\mathcal{C}_2}$	g	15
$dp^{\mathcal{C}_2}$	h	20

Table 13. Data pilots from Example 5.1

Suppose an user wants to execute the query $Q1 = \pi_{e,f,d,g}(dp^{\mathcal{C}_1} \bowtie_{d=g} dp^{\mathcal{C}_2})$. Following the steps outlined above:

1. **Select a candidate optimization:** Based on Lemma 3.14, $Q1$ can be rewritten as $Q2 = \pi_{d,e}(dp^{\mathcal{C}_1}) \bowtie_{d=g} \pi_{g,f}(dp^{\mathcal{C}_2})$. The candidate optimization is then "projection push-down over join".

2. **Pipeline-breaker operators influenced by changes in volume:** For this case, only the join operator is affected.

3. **Estimation of input/output volumes before optimization:** For this case, the output volume of the join is the same whether or not the optimization is applied. Thus, the only determining factor is the input volume computed as

follows:

$$
\begin{aligned}
vol\left(dp^{\mathcal{C}_1}\right) &= \frac{40}{40} * 40 + \frac{30}{40} * 30 + \frac{15}{40} * 15 + \frac{14}{40} * 14 = 73.025 \\
vol\left(dp^{\mathcal{C}_2}\right) &= \frac{20}{20} * 20 + \frac{5}{20} * 5 + \frac{15}{20} * 15 = 32.5 \\
\text{input volume } &= vol\left(dp^{\mathcal{C}_1}\right) + vol\left(dp^{\mathcal{C}_2}\right) = 73.025 + 32.5 = 105.525
\end{aligned}
$$

4. **Estimation of input/output volumes after optimization:** $Q2$ is the optimized version of $Q1$ after applying the candidate optimization. The input volume of the join in $Q2$ is then:

$$
\begin{aligned}
vol\left(\pi_{d,e}\left(dp^{\mathcal{C}_1}\right)\right) &= \frac{15}{15} * 15 + \frac{14}{15} * 15 \approx 28.067 \\
vol\left(\pi_{g,f}\left(dp^{\mathcal{C}_2}\right)\right) &= \frac{15}{15} * 15 + \frac{5}{15} * 5 \approx 16.667 \\
\text{input volume } &= vol\left(dp^{\mathcal{C}_1}\right) + vol\left(dp^{\mathcal{C}_2}\right) = 28.067 + 16.667 = 44.734
\end{aligned}
$$

5. **Percentage decrease in input/output volume of operator:**

$$
\frac{105.525 - 44.734}{105.525} \approx 0.5761
$$

□

Example 5.2. Consider a collection $\mathcal{C}$ and its data pilot $dp^{\mathcal{C}}$, with the cardinality and counters shown in Table 14.

Path	obj	arr	int	str	Cardinality
ϵ	40	0	0	0	40
a	0	0	25	15	40
b	0	0	30	0	30
d	0	0	5	10	15
e	0	0	6	8	14

Table 14. Data pilot from Example 5.2

Assume the following queries are issued:

- $\pi_{a,b}(\sigma_{a\theta const}(\mathcal{C}))$
- $\pi_{a,b}(\sigma_{b\theta const}(\mathcal{C}))$
- $\pi_{a,b}(\sigma_{d\theta const}(\mathcal{C}))$
- $\pi_{a,b}(\sigma_{e\theta const}(\mathcal{C}))$
- $\pi_{d,e}(\sigma_{a\theta const}(\mathcal{C}))$
- $\pi_{d,e}(\sigma_{b\theta const}(\mathcal{C}))$
- $\pi_{d,e}(\sigma_{d\theta const}(\mathcal{C}))$
- $\pi_{d,e}(\sigma_{e\theta const}(\mathcal{C}))$

First, start by computing the volume of $dp^{\mathcal{C}}$ as follows:

$$vol\left(dp^{\mathcal{C}}\right) = \frac{40}{40} * 40 + \frac{30}{40} * 30 + \frac{15}{40} * 15 + \frac{14}{40} * 14 = 73.025$$

then, compute the volume of each projection and selection pushdown followed by the percentage decrease in volume using $vol\left(dp^{\mathcal{C}}\right)$ as the base.

For $\pi_{a,b}(\sigma_{a\theta const}(\mathcal{C}))$:

New counters and cardinality after selection:

$$Card(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), \epsilon) = c_{int}\left(dp^{\mathcal{C}}, a\right) = 25$$

$$c_{int}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 25 * 1 \rfloor = 25,\ c_{str}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), a\right) = 0$$

$$c_{int}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), b\right) = \lceil 25 * (30/40) \rfloor = 19,\ c_{str}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), b\right) = 0$$

$$c_{int}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 25 * (5/40) \rfloor = 3,\ c_{str}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 25 * (10/40) \rfloor = 6$$

$$c_{int}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 25 * (6/40) \rfloor = 4,\ c_{str}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 25 * (8/40) \rfloor = 5$$

Number of non-empty documents:

$$max\left(Card(\pi_{a,b}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right), a), Card(\pi_{a,b}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right), b)\right) = max\,(25, 19) = 25$$

New volume:

$$vol\left(\pi_{a,b}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{25}{25} * 25 + \frac{19}{25} * 19 = 39.44$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{a,b}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 39.44}{73.025} \approx 0.46$$

For $\pi_{a,b}(\sigma_{b\theta const}(\mathcal{C}))$:

New counters and cardinality after selection:

$Card(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), \epsilon) = c_{int}\left(dp^{\mathcal{C}}, b\right) = 30$

$c_{int}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), b\right) = \lceil 30 * 1 \rfloor = 30,\ c_{str}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), b\right) = 0$

$c_{int}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 30 * (25/40) \rfloor = 19,\ c_{str}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 30 * (15/40) \rfloor = 11$

$c_{int}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 30 * (5/40) \rfloor = 4,\ c_{str}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 30 * (10/40) \rfloor = 8$

$c_{int}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 30 * (6/40) \rfloor = 5,\ c_{str}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 30 * (8/40) \rfloor = 6$

Number of non-empy documents:

$max\left(Card(\pi_{a,b}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right), a), Card(\pi_{a,b}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right), b)\right) = max\,(30, 30) = 30$

New volume:

$$vol\left(\pi_{a,b}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{30}{30} * 30 + \frac{30}{30} * 30 = 60$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{a,b}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 60}{73.025} \approx 0.1784$$

For $\pi_{a,b}(\sigma_{d\theta const}(\mathcal{C}))$:

New counters and cardinality after selection:

$Card(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), \epsilon) = c_{int}\left(dp^{\mathcal{C}}, d\right) = 5$

$c_{int}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 5 * 1 \rfloor = 5,\ c_{str}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), d\right) = 0$

$c_{int}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 5 * (25/40) \rfloor = 3,\ c_{str}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 5 * (15/40) \rfloor = 2$

$c_{int}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), b\right) = \lceil 5 * (30/40) \rfloor = 4,\ c_{str}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), b\right) = 0$

$c_{int}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 5 * (6/40) \rfloor = 1,\ c_{str}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 5 * (8/40) \rfloor = 1$

Number of non-empy documents:

$max\left(Card(\pi_{a,b}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right), a), Card(\pi_{a,b}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right), b)\right) = max\,(5, 4) = 5$

New volume:

$$vol\left(\pi_{a,b}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{5}{5} * 5 + \frac{4}{5} * 4 = 8.2$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{a,b}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 8.2}{73.025} \approx 0.8877$$

For $\pi_{a,b}(\sigma_{e\theta const}(\mathcal{C}))$:

New counters and cardinality after selection:

$Card(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), \epsilon) = c_{int}\left(dp^{\mathcal{C}}, e\right) = 6$

$c_{int}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), e\right) = \lceil 6 * 1 \rfloor = 1,\ c_{str}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), e\right) = 0$

$c_{int}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 6 * (25/40) \rfloor = 4,\ c_{str}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), a\right) = \lceil 6 * (15/40) \rfloor = 2$

$c_{int}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), b\right) = \lceil 6 * (30/40) \rfloor = 5,\ c_{str}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), b\right) = 0$

$c_{int}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 6 * (5/40) \rfloor = 1,\ c_{str}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right), d\right) = \lceil 6 * (10/40) \rfloor = 2$

Number of non-empy documents:

$$max\left(Card(\pi_{a,b}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right), a), Card(\pi_{a,b}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right), b)\right) = max\left(6, 5\right) = 6$$

New volume:

$$vol\left(\pi_{a,b}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{6}{6} * 6 + \frac{5}{6} * 5 = 10.17$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{a,b}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 10.17}{73.025} \approx 0.8607$$

The cardinalities and counters previously calculated for the data pilots $\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)$, $\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)$, $\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)$, and $\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)$ are employed for the following volume computations and reductions.

For $\pi_{d,e}(\sigma_{a\theta const}(\mathcal{C}))$:

Number of non-empty documents:

$$max\left(Card(\pi_{d,e}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right), d), Card(\pi_{d,e}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right), e)\right) = max\left(9, 9\right) = 9$$

New volume:

$$vol\left(\pi_{d,e}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{9}{9} * 9 + \frac{9}{9} * 9 = 18$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{d,e}\left(\sigma_{a\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 18}{73.025} \approx 0.7535$$

<u>For $\pi_{d,e}(\sigma_{b\theta const}(\mathcal{C}))$:</u>

Number of non-empty documents:

$$max\left(Card(\pi_{d,e}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right), d), Card(\pi_{d,e}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right), e)\right) = max\left(12, 11\right) = 12$$

New volume:

$$vol\left(\pi_{d,e}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{12}{12} * 12 + \frac{11}{12} * 11 = 22.083$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{d,e}\left(\sigma_{b\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 22.083}{73.025} \approx 0.6976$$

<u>For $\pi_{d,e}(\sigma_{d\theta const}(\mathcal{C}))$:</u>

Number of non-empty documents:

$$max\left(Card(\pi_{d,e}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right), d), Card(\pi_{d,e}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right), e)\right) = max\left(5, 2\right) = 5$$

New volume:

$$vol\left(\pi_{d,e}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{5}{5} * 5 + \frac{2}{5} * 2 = 5.8$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{d,e}\left(\sigma_{d\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 5.8}{73.025} \approx 0.9206$$

<u>For $\pi_{d,e}(\sigma_{e\theta const}(\mathcal{C}))$:</u>

Number of non-empty documents:

$$max\left(Card(\pi_{d,e}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right), d), Card(\pi_{d,e}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right), e)\right) = max\left(6, 3\right) = 6$$

New volume:

$$vol\left(\pi_{d,e}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right)\right) = \frac{6}{6} * 6 + \frac{3}{6} * 3 = 7.5$$

Reduction in volume of:

$$\frac{vol\left(dp^{\mathcal{C}}\right) - vol\left(\pi_{d,e}\left(\sigma_{e\theta const}\left(dp^{\mathcal{C}}\right)\right)\right)}{vol\left(dp^{\mathcal{C}}\right)} = \frac{73.025 - 7.5}{73.025} \approx 0.8973$$

□

Example 5.3. Consider a collection $\mathcal{C}$ and its data pilot $dp^{\mathcal{C}}$, with the path, cardinalities and counters shown in Table 15a. Assume the query $\pi^{\oslash}_{\mathrm{a \cdot c, a \cdot d}}\left(\mu^{\oslash}_{\mathrm{a}}\left(dp^{\mathcal{C}}\right)\right)$ is

Path	obj	arr	int	str	Cardinality
ϵ	20	0	0	0	20
a	0	5	0	0	5
b	0	0	20	0	20
a·⋆	50	0	0	50	100
a·⋆·c	0	0	0	50	50
a·⋆·d	0	0	30	0	30

(a) $dp^{\mathcal{C}}$

Path	obj	arr	int	str	Cardinality
ϵ	20	0	0	0	20
a	0	5	0	0	5
a·⋆	50	0	0	0	50
a·⋆·c	0	0	0	50	50
a·⋆·d	0	0	30	0	30

(b) $\pi^{\oslash}_{\mathrm{a \cdot \star \cdot c, a \cdot \star \cdot d}}\left(dp^{\mathcal{C}}\right)$

Path	obj	arr	int	str	Cardinality
ϵ	100	0	0	0	100
a	50	0	0	50	100
b	0	0	100	0	100
a·c	0	0	0	50	50
a·d	0	0	30	0	30

(c) $\mu^{\oslash}_{\mathrm{a}}\left(dp^{\mathcal{C}}\right)$

Path	obj	arr	int	str	Cardinality
ϵ	50	0	0	0	50
a	50	0	0	50	50
a·c	0	0	0	50	50
a·d	0	0	30	0	30

(d) $\mu^{\oslash}_{\mathrm{a}}\left(\pi^{\oslash}_{\mathrm{a \cdot \star \cdot c, a \cdot \star \cdot d}}\left(dp^{\mathcal{C}}\right)\right)$

Table 15. Data pilots from Example 5.3

given to the system. This query can be rewritten by pushing down the projection as follows: $\mu_{\mathrm{a}}\left(\pi^{\oslash}_{\mathrm{a \cdot \star \cdot c,\ a \cdot \star \cdot d}}\left(dp^{\mathcal{C}}\right)\right)$. This is how both queries compare in terms of volume:

For unnest's input volume

1. Compute the input volume before the projection pushdown (i.e. The volume from the DataPilot in Table 15a):

$$vol(dp^{\mathcal{C}}) = \frac{20}{20}*20 + \frac{5}{20}*5 + \frac{20}{20}*20 + \frac{5}{20}*\frac{100}{100}*100 + \frac{5}{20}*\frac{50}{100}*50 + \frac{5}{20}*\frac{30}{100}*30 = 74.75$$

2. Compute the input volume after the projection pushdown (i.e. The volume from the DataPilot in Table 15b):

$$vol(\pi^{\oslash}_{\text{a}\cdot\star\cdot\text{c, a}\cdot\star\cdot\text{d}}\left(dp^{\mathcal{C}}\right)) = \frac{20}{20}*20+\frac{5}{20}*5+\frac{5}{20}*\frac{50}{50}*50+\frac{5}{20}*\frac{50}{50}*50+\frac{5}{20}*\frac{30}{50}*30 = 50.75$$

3. Compute the reduction in input volume: $(74.75 - 50.75)/74.75 \approx 0.3211$.

For unnest's output volume

1. Compute the output volume before the projection pushdown (i.e. The volume from the DataPilot in Table 15c):

$$vol(\mu^{\oslash}_{\text{a}}(dp^{\mathcal{C}})) = \frac{100}{100} * 100 + \frac{100}{100} * 100\frac{100}{100} * 100 + \frac{50}{100} * 50 + \frac{30}{100} * 30 = 334$$

2. Compute the output volume after the projection pushdown(i.e. The volume from the DataPilot in Table 15d):

$$vol(\mu_{\text{a}}(\pi^{\oslash}_{\text{a}\cdot\star\cdot\text{c, a}\cdot\star\cdot\text{d}}\left(dp^{\mathcal{C}}\right))) = \frac{50}{50} * 50 + \frac{50}{50} * 50\frac{50}{50} * 50 + \frac{30}{50} * 30 = 168$$

3. Compute the reduction in input volume: $(334 - 168)/334 \approx 0.4970$.

□

CHAPTER VI

EXPERIMENTAL ANALYSIS

1 Chapter Overview

This chapter presents two proof-of-concept (POC) experiments focusing on essential aspects of the envisioned query processing framework. Recognizing the comprehensive nature of the entire framework's development beyond the scope of the thesis, each POC selectively assumes the existence of components not directly relevant to the validated concept. The primary objectives of these POCs include the creation of a Data Pilot, as described in Section VI.2, and the execution of queries using "compiled queries" detailed in Section VI.3. In this approach, each query is translated into source code and subsequently compiled into a program for execution.

In the context of the query compilation and execution POC, an algebraic query translator is presumed to exist. Within this component, an input query undergoes translation into expressions at levels 2 and 3 of the document algebra, forming a query tree. The translation from the query language to the query tree depends on the specific query language used by a given JSON document store. For the conducted experiments, queries from the TPC-H benchmark, scripted in MongoDB's aggregation pipeline, are manually translated into the algebra.

Subsequently, query plans are generated in two phases. The first phase involves applying various transformations to the query tree based on algebraic properties of the operators. These transformations aim to simplify or enhance the tree. Noteworthy

transformations include the strategic pushing down of projections and selections, converting outerjoins into joins, and optimizing the elimination of unnestings and the delay of nests. In the second phase, each resulting tree from the first phase undergoes implementation through (a) providing an implementation for each operator in the tree and (b) exploring different join orders in case of multiple joins in the query, yielding a set of physical query plans.

The optimization process also identifies sequences of operations that correspond to a macro-operator, replacing them with the macro-operator. This introduces cross-operator optimization, a dimension enhancing the optimization process.

Example 1.1 (Demonstration of the Approach: Query 5)**.** The process of query translation and plan generation is exemplified here with query 5 from the TPC-H benchmark, implemented on a database where each relational table resides in a distinct collection. Figure 49 MongoDB's aggregate pipeline representation of Query 5, with multiple query versions considered for optimal performance. Its translation to the proposed algebraic framework is portrayed in Figure 50. Notably, this algebraic expression is structured as a tree and subjected to Phase I of query planning. This initial optimization stage involves pushing down projections and selections, eliminating matching nest and unnest pairs, and converting outerjoins to joins when feasible. The decision to transform outerjoins relies on insights from the DataPilot, which affirms that documents on one side of the outerjoin invariably have matches on the other side. The resulting optimized tree, showcased in Figure 51a, manifests a considerably streamlined algebraic expression. In Phase II, where several join orders are explored, the best-suited query plan is determined, as revealed in Figure 51b. The DataPilot again plays a pivotal role, providing estimates of selection size (distribution of attributes like `r_name` and `o_orderdate`) and collection size (e.g., `orders`, `customers`, `lineitem`, `supplier`). It is worth noting that the final plan does not explicitly exhibit the concluding grouping and ordering operations performed at the apex of the tree.

```
S1  {$match:{$expr:{$and:[
      {$gte:["$o_orderdate", ISODate("1992-01-01T00:00:00Z")]},
      {$lt:["$o_orderdate", ISODate("1993-01-01T00:00:00Z")]}]}}},
S2  {$lookup: {from: "lineitem",localField: "_id",
      foreignField: "_id.l_orderkey",as: "lineitems"}},
S3  {$unwind: "$lineitems"},
S4  {$lookup: {from: "customer",localField: "o_custkey",
      foreignField: "_id",as: "customers"}},
S5  {$unwind: "$customers"},
S6  {$lookup: {from: "supplier",localField: "customers.c_nationkey",
     foreignField: "s_nation.n_nationkey",as: "suppliers"}},
S7  {$unwind: "$suppliers"},
S8  {$project:{suppliers: 1,lineitems: 1}},
S9  {$match: {"suppliers.s_nation.n_region.r_name": "MIDDLE EAST"}},
S10 {$match: {$expr:{$eq:["$suppliers._id", "$lineitems.l_suppkey"]}}},
S11 {$project:{suppliers:1,
      l_extendedprice: "$lineitems.l_extendedprice",
      l_discount: "$lineitems.l_discount"}},
S12 {$group: {
       _id: "$suppliers.s_nation.n_name",
       revenue: {$sum: {$multiply:[
         "$l_extendedprice", {$subtract:[1, "$l_discount"]}]}}}}
```

Figure 49. TPC-H Query 5 in MongoDB's aggregate pipeline. Pipeline starts from collection `orders`.

$$S_1 = \sigma_{\text{o_orderdate} \geq \text{1992-01-01} \wedge \text{o_orderdate} < \text{1993-01-01}}(\texttt{orders})$$

$$S_2 = S_1 \,\widehat{\bowtie}^{\text{litems}}_{\text{_id=lid}\cdot\text{l_orderkey}}\, \rho_{\text{lid} \leftarrow \text{_id}}(\texttt{lineitem})$$

$$S_3 = \mu_{\text{litems}}(S_2)$$

$$S_4 = S_3 \,\widehat{\bowtie}^{\text{custs}}_{\text{o_custkey=cid}}\, \rho_{\text{cid} \leftarrow \text{_id}}(\texttt{customer})$$

$$S_5 = \mu_{\text{custs}}(S_4)$$

$$S_6 = S_5 \,\widehat{\bowtie}^{\text{supps}}_{\text{custs}\cdot\text{c_nationkey=s_nation}\cdot\text{n_nationkey}}\, \texttt{supplier}$$

$$S_7 = \mu_{\text{supps}}(S_6)$$

$$S_8 = \pi_{\text{supps,litems}}(S_7)$$

$$S_9 = \sigma_{\text{supps}\cdot\text{s_nation}\cdot\text{n_region.r_name='MIDDLE EAST'}}(S_8)$$

$$S_{10} = \sigma_{\text{supps}\cdot\text{_id=litems}\cdot\text{l_suppkey}}(S_9)$$

$$S_{11} = \pi_{\text{supps}\cdot\text{s_nation}\cdot\text{n_name,litems}\cdot\text{l_extendedprice,litems}\cdot\text{l_discount}}(S_{10})$$

$$S_{12} = \gamma_{\text{supps}\cdot\text{s_nation}\cdot\text{n_name,sum(litems}\cdot\text{l_extendedprice* (1 - litems}\cdot\text{l_discount))}}(S_{11})$$

Figure 50. Initial algebraic expression of TPC-H Query 5 derived from Figure 49

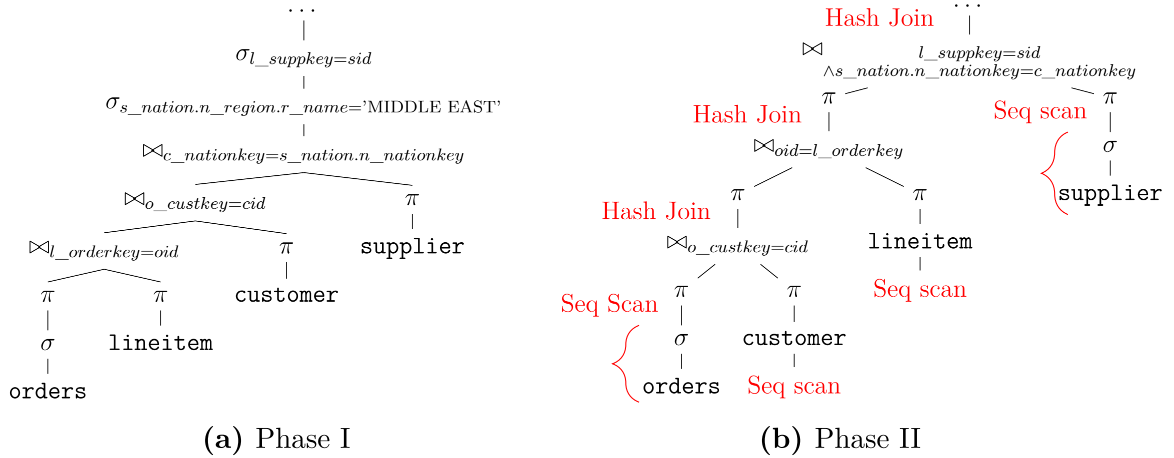


Figure 51. Query trees derived from Figure 50 after phases I and II of query planning

This example underscores the significance of algebraic manipulations in our methodology. Transformation of outerjoins into joins is instrumental in considering diverse join orders, a flexibility constrained by the non-associative nature of outerjoins ([71]). MongoDB's optimizer, in contrast, adheres to the query's specified order, necessitating manual optimization.

Remarkably, the resultant query tree closely aligns with a relational structure. Comparative analysis with plans produced by PostgreSQL indicates striking similarities, implying that the proposed approach, in a sense, discerns and simplifies the "relational nature" of the data. This intrinsic capability contributes to an optimized plan that mirrors traditional relational strategies. □

In the envisioned model, cost estimates are grounded in the work of [36], providing formulas to evaluate document access costs via indices or scanning. These formulas extend traditional approaches by incorporating factors like average document size (partially captured in the Data Pilot) and system-dependent elements such as cache size and eviction policies. These costs estimates are, however, not considered for the POCs as evaluation of system-dependent elements is outside the scope of this thesis.

The final stage of the process involves generating a program to be executed against the database back-end. The current approach involves compiling the query using

templates, akin to the methodology employed in the Hyper system ([61]). Each algebraic operator is associated with a code block containing parameters. These distinct blocks are assembled by traversing the query plan from the bottom up.

The query plan is divided into sections based on “blocking” operations, primarily joins and groupings. A code block or segment is generated for each section, with efforts made to amalgamate code for multiple operations within a tight loop. Thanks to the utilization of macro-operators, multiple operations can sometimes be consolidated into a single code block. For example, a sequence involving selections, projections, and joins might be executed in a singular loop, ultimately resulting in a hashmap probed later by the other collection in the join.

Similar to the approach detailed in [44], the code is tailored for diverse backends. This adaptation is crucial for JSON data, which may be stored as BSON or other formats. Leveraging information from the DataPilot and the query, an attempt is made to determine or narrow down possible data types. For instance, if projections involve attributes like `a·b·c` and `a·b·d`, and the DataPilot indicates that `a·b·c` is consistently an integer while `a·b·d` is consistently a string, a structure with two fields (an integer and a string) is created to capture these values. In cases where `a·b·c` might be either an integer or a float, a structure with a float is created, converting `a·b·c` as necessary. In scenarios where `a·b·c` is variably an integer or a string, a typed union with `std::variant` is used to store this value.

Additionally, the query itself aids in type inference. For instance, a condition like `order·discount= 0.5` implies that only numerical values of the attribute can satisfy the condition. If the type of `order·discount` has not been conclusively determined by the Data Pilot, the attribute is accessed, and non-numerical values (along with their respective documents) are discarded without further condition checks.

The current strategy employs an "eager but opportunistic" approach to format conversion. Data is converted upon access but only when required. For instance, in the case mentioned above, `order·discount` is converted to check the condition.

Only if the condition holds true are other attributes in the document accessed and converted. This approach is efficient as it aligns with the principle of pushing down projections, ensuring that only necessary attributes undergo processing.

2 Proof of Concept: Data Pilot Creation

This small experiment evaluates the runtime overhead of creating a Data Pilot during parsing using a *naive*, non-optimized implementation. The Data Pilot is constructed based in Pseudocodes 2, 1, and 3.

For Pseudocode 2, the experiment benchmarks two different data structures: the HAT-Trie[7] and the Adaptive Radix Tree (ART)[48]. Note that while the HAT-Trie and ART are designed with performance in mind, this experiment does not account for considerations like parallel (multi-core) execution or tracking the last inserted path.

In the case of Pseudocode 3, the experiment utilizes the SIMDJson on-demand parser ([47]). This parser efficiently handles the placement and movement of the iterator within and across documents. Additionally, serialization into BSON is integrated into the parsing process using the libbson library from MongoDB.

The experiment consists of three benchmarks:

1. **Path Insertion:** Reads a file of JSON documents, extracts their paths one document at a time, and inserts them into the designated data structure for the Data Pilot's pathstore. This benchmark solely measures the raw insertion time for both the HAT-Trie and ART.

2. **Data Pilot Creation:** Reads a file of JSON documents, extracts their paths one document at a time, and inserts them into the designated data structures for the Data Pilot's pathstore and counter store. Path types are counted during this process. The Data Pilot implementation involves using a Pathstore implementation (either a HAT-Trie or ART) to store the paths, and a vector of

counters to store the counters of each path. When a new path is encountered, a new counter struct is created and added to the vector, then the path is inserted into the pathstore's underlying data structure as a key, with its value being the position of the counter struct in the vector. This benchmark measures the raw creation time of the Data Pilot.

3. **BSON Serialization:** Reads a file of JSON documents and converts them into BSON. Notably, the libbson library provides a function to parse JSON into BSON, but this function is not utilized in the experiment. Instead, parsing is handled using SIMDJson, and the respective BSON documents are created using the relevant libbson directives. This benchmark measures the time taken to serialize JSON into BSON, both with and without constructing the Data Pilot.

The benchmarks utilize the Yelp 2019 Dataset, which serves as a subset of Yelp's businesses, reviews, and user data. This dataset comprises multiple text files, where each line represents a JSON document. Each text file essentially represents a collection in its own right. For the experiments, the two smallest and largest files from the dataset are selected. The smaller files pertain to businesses and tips, while the larger ones contain user and review data. Henceforth, these collections are referred to as Business, Tip, User, and Review. Table 16 provides pertinent details about each collection.

	Business	Tip	User	Review
Size (MiB)	114	173	3208	5095
Number of documents	150346	908915	1987897	6990280
Number of complete paths	60	5	22	9
Maximum depth	2	1	1	1
Average depth	1.972	1.000	1.000	1.000

Table 16. Yelp collections used for the Data Pilot creation POC

Benchmark	Implementation	Yelp Dataset (time in ms)			
		Business	Tip	User	Review
Path Insertion	HAT-Trie	187	173	1542	2193
	ART	183	120	1385	1661
Data Pilot Creation	HAT-Trie	176	162	1466	2095
	ART	265	204	2311	2330
BSON Serialization	HAT-Trie	417	437	4539	7128
	ART	520	541	5653	7860
	No Data Pilot	330	373	3668	6125

Table 17. Result of Data Pilot feasibility experiment

Results and discussion

All benchmarks are run on a Lenovo Legion Pro 5 laptop equipped with an AMD Ryzen 7 5800H processor, 16GB of RAM, and running the Alma Linux 9.3 operating system. Each benchmark is executed 5 times, and the average time is reported. Table 17 summarizes the results of each benchmark and Figure 52 depicts them.

The results indicate that, on the whole, the HAT-Trie outperforms ART. However, it is evident that ART generally exhibits superior performance in raw path insertion. The observed performance degradation during Data Pilot creation and BSON Serialization benchmarks when using ART can likely be attributed to its interactions with the counter store implementation. It's essential to emphasize that this experiment does not delve into the intricacies of these issues, their underlying causes, or potential solutions. Instead, it serves as a proof of concept, illustrating the feasibility of creating a Data Pilot at parsing time through a naive implementation with a reasonable increase in running time. Specifically, when employing the HAT-Trie, a maximum overhead of approximately 21% was observed across the Yelp datasets.

In the BSON Serialization benchmark, the User and Business collections exhibit the most significant performance decline, with their running times increasing by 19.2% and 20.86%, respectively, compared to plain BSON serialization without Data Pilot

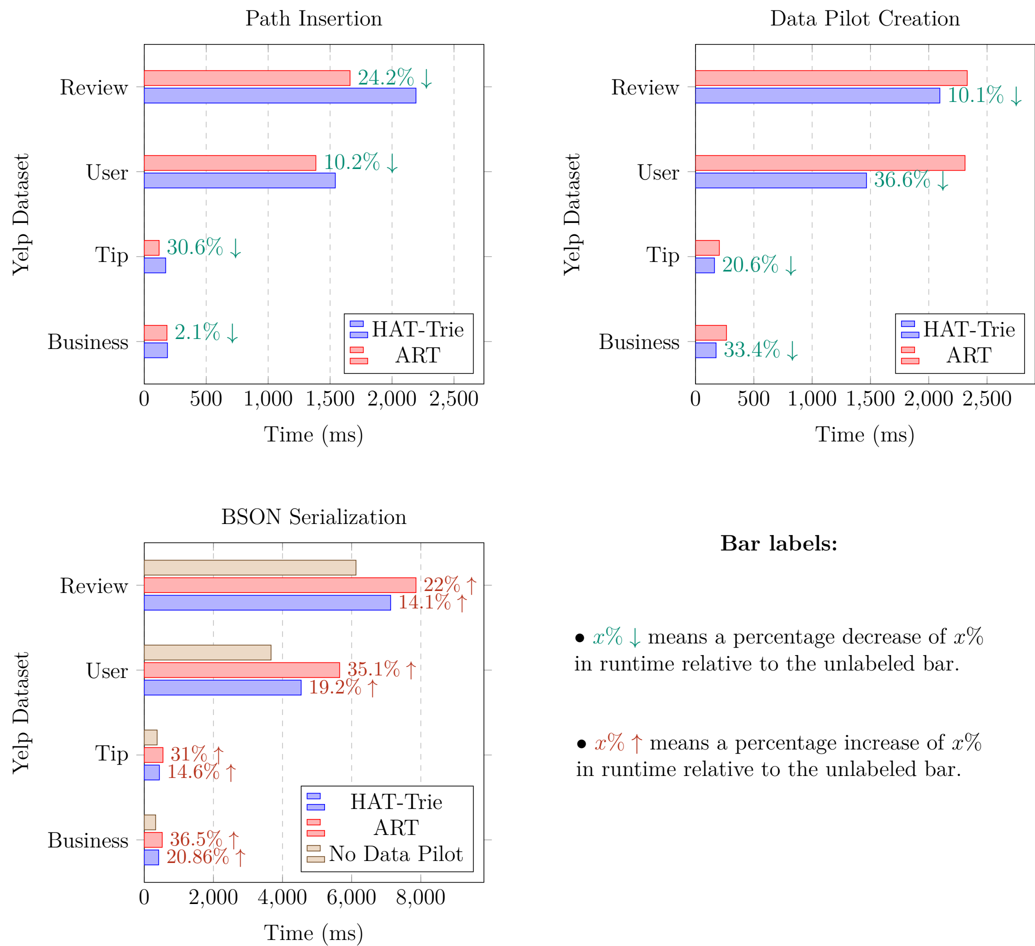


Figure 52. Results of benchmarks from the Data Pilot feasibility experiment

	Business	Tip	User	Review
Number of documents	150346	908915	1987897	6990280
Documents with at least one missing path	100%	0%	0%	0%
Number of complete paths	60	5	22	9
Maximum depth	2	1	1	1
Overhead	20.86%	14.6%	19.2%	14.1%

Table 18. Minum overhead of Data Pilot creation in BSON Serialization benchmark

creation. These collections share a common characteristic: they contain the highest number of complete paths among all four collections. In contrast, the Tip and Review collections, with fewer complete paths, show a lower overhead ranging between 14% and 15%, despite the Review collection occupying 4922MiB more space than Tip. This information is summarized in Table 18, which presents the minimum overhead recorded in the benchmark, observed in the HAT-Trie implementation. From the table, it is evident that the more heterogeneous a collection is, the greater the overhead incurred when creating a Data Pilot. Although heterogeneity is influenced by multiple factors, including depth, the number of paths across all documents, and documents with missing paths, the number of paths across all documents appears to be the most influential factor on the overhead. This is expected, as a larger number of paths increases the overall size of the Data Pilot.

3 Proof of Concept: Query Optimization Framework

The assessment methodology employs the TPC-H benchmark ([73]). Analogous to the adaptation of the relational-formatted TPC-H data to JSON in the experiments detailed in Section III.1, this experiment recreates the original data in JSON using two approaches, yielding two schemas: *Denormalized* (referred to as S1) and *Normalized* (S2). S2 comprises the collections Customer ($\mathcal{C}$), Orders ($\mathcal{O}$), Lineitem ($\mathcal{L}$), Supplier ($\mathcal{S}$), and Part ($\mathcal{P}$), thus mimicking the original relational schema. S1 consists of collections $\mathcal{COL}$—the amalgamation of $\mathcal{C}$, $\mathcal{O}$, and $\mathcal{L}$ through embedding—$\mathcal{P}$, and $\mathcal{S}$. The latter two are linked through two-way embedding—documents in $\mathcal{S}$ incorporate a list of ids from all associated parts, and in $\mathcal{P}$ a list of sub-documents modeled after matching tuples from the PartSupp table in the relational schema (noting that attributes from this table can be embedded in Parts, Suppliers, or both). Each document in $\mathcal{S}$ includes a (single) subdocument modeled after matching tuples from the junction of tables Nation and Region in the relational schema. The objective is to leverage naturally occurring hierarchies between Customer, Order, and Lineitem

to create a schema that exploits JSON's ability to represent nesting.

This experiment employs a subset of queries from the TPC-H benchmark—1, 3, 4, 6, 12, 13, 18, and 22 for hierarchical data (only tables Customer, Orders, and Lineitem are involved), 2 and 11 to address the M-N case (only tables Part, Supplier, and PartSupp are involved), and 5 and 9 to address the mixed (1-M and M-N) case. The experiments run on scale factors 1, 10, 25, and 50 using an IBM System x3650 M2 server with two Intel Xeon X5672 processors at 3.20GHz, 16 virtual cores, and 46GB of RAM running CentOS 7.6. MongoDB 4.4 and PostgreSQL 10.6 serve as the baselines, with both database systems tuned according to best practices. One database per system is created to hold the benchmark's data.

While the entire framework described in Section III is not fully implemented, the experiment aims to evaluate its potential before committing to a specific implementation and to gain a better understanding of its weaknesses. To "simulate" an algorithmic process, a translation from MongoDB's aggregation pipeline to the document algebra is defined by providing a translation for each operator involved in any query (`$lookup, $unwind, ...`). On each algebraic query tree, the following transformations are applied in the same order: pushing down selections, eliminating nest/unnest pairs (whenever possible), transforming outer joins into joins (whenever possible), and pushing down projections. Once the query tree is refined, plans are generated by considering join orders. To simplify query plan generation and evaluation and concentrate on the impact of algebraic optimization, no indices are used (all access is through scan), and only hashing is considered as an implementation for joins. As a result, only one join order is considered for plan generation, and cost estimation is greatly simplified. This, in turn, makes it possible to generate a final, compiled query via simple templates. Finally, it is assumed that a Data Pilot with relevant information is available for each collection, as well as metadata about the database structure. The ramifications of these simplifying choices are discussed in detail later in this section.

The queries are reformulated both in MongoDB's aggregation framework (referred to as *MongoDB queries* for brevity) and as C++ programs (referred to as *compiled queries*).

These compiled queries are constructed against MongoDB's source code, utilizing their catalog library to access data files in the Wired Tiger storage engine. The translation of MongoDB queries to the algebra, cost estimation, and query plans computation for the compiled queries is performed manually. Queries are versioned based on the schema they use (S1 or S2), except for 2 and 11 as they exclusively involve $\mathcal{P}$ and $\mathcal{S}$. Timing for MongoDB queries is accomplished using the aggregation framework's explain command running in "allPlansExecution" mode, compiled queries leverage MongoDB's function-level benchmarking library, and relational queries utilize PostgreSQL's explain analyze command. All queries are executed five times, and the cache of the operating system and the respective database system is cleared after each execution. The average running time is reported. Due to time constraints, there is a set time limit of 10 hours per query, and indexes are added to the linking fields of collections to favor MongoDB's $lookup operator. Indexes are also added to foreign keys on the tables of the relational database. During experimentation it is noted that these indexes are not utilized by PostgreSQL nor our approach (as indexing is not supported).

The results are presented on a per-query basis below, using the notation q<number>_[s1|s2]_[m|c] to distinguish MongoDB queries from compiled queries. Here, <number> corresponds to the TPC-H query number, [s1|s2] denotes the denormalized (s1) or normalized (s2) schema, and [m|c] indicates a MongoDB query (m) or a compiled query (c).

Results

Queries 1 and 6: Both queries exclusively operate on lineitems. Query 1 involves 1 selection and 8 aggregates, while query 6 has 4 selections and no group by. Query

Query	Schema	SF1	SF10	SF25	SF50
1	S1	46.20%	43.27%	43.85%	39.25%
	S2	38.19%	31.16%	32.97%	28.77%
	S2†	38.74%	31.36%	34.12%	32.11%
2	N/A‡	43.65%	32.70%	41.84%	36.94%
3	S1	45.15%	50.04%	42.34%	9.14%
	S2	-127.10%	-117.33%	-122.46%	-119.98%
4	S1	52.31%	56.27%	56.33%	37.94%
	S2	30.20%	27.72%	27.25%	25.95%
5	S1	97.06%	97.22%	96.82%	95.39%
	S2	93.98%	93.91%	93.69%	93.12%
6	S1	35.35%	44.62%	37.92%	30.77%
	S2	30.83%	32.12%	23.15%	20.36%
9	S1	99.21%	99.15%	99.12%	N/A§
	S2	91.34%	92.61%	92.63%	N/A§
11	N/A‡	81.11%	83.01%	N/A§	N/A§
12	S1	11.24%	23.78%	14.32%	5.90%
	S2	92.72%	92.72%	92.62%	92.38%
13	S1	15.16%	17.57%	19.36%	14.36%
	S2	78.31%	78.85%	78.89%	81.50%
18	S1	55.58%	58.87%	54.32%	47.79%
	S2	64.58%	64.50%	60.36%	56.68%
22	S1	49.48%	59.25%	65.51%	72.67%
	S2	89.63%	89.19%	88.59%	90.65%

S1: Denormalized schema, **S2:** Normalized schema, **S2†:** Query on schema S2 with extra features to handle data heterogeneity, **N/A‡:** Query only uses collections $\mathcal{P}$ and $\mathcal{S}$, **N/A§:** Query was killed

Table 19. Decrease in running time achieved by compiled queries compared to their MongoDB query counterparts. Higher is better.

1's selection based on `l_shipdate` retrieves between 95% and 97% of the rows in the table. In alignment with prior work ([54, 53]), the selectivity is enhanced to only return 0.00028% of the rows. Figure 53a illustrates that `q1_s1_c` and `q1_s1_m` are slower than their S2 counterparts. This is attributed to the former queries reading all customers, orders, and lineitems, whereas `q1_s2_c` and `q1_s2_m` exclusively read lineitems. Moreover, leveraging tight loops for processing extensive data enhances code locality, boosting performance on compiled queries and significantly reducing running time, as demonstrated in Table 19. A similar performance improvement is observed for query 6.

The compiled queries rely on statistics to infer a field's datatype. However, due to the homogeneity of the TPC-H data, the DataPilot consistently returns a field's exact type. Consequently, `q1_s1_c` and `q1_s2_c` are generated under this assumption, neglecting data heterogeneity. To address this, an alternative version of `q1_s2_c` named `q1v2_s2_c` is created. This version employs a hash table with BSON keys to handle cases where the same field may have different datatypes and introduces checks (thereby introducing branches) to verify the correct datatype of attributes before computing aggregates.

In Figure 54, the updated query demonstrates performance comparable to the original version on S2. However, it is acknowledged that a few branch mispredictions may occur due to the homogeneity of the dataset type. The investigation of performance degradation in the presence of heterogeneous data is deferred to future work.

Query 2: This query interacts with collections $\mathcal{P}$ and $\mathcal{S}$. The compiled query first constructs a supplier hashtable (first loop) and then iterates through documents in $\mathcal{P}$ (second loop). The second loop navigates nested documents, modeled after the PartSupp table in the relational schema, in the `p_suppliers` attribute to compute the correlated sub-query. It subsequently performs a second pass to compute the outer query. While optimizing time locality by having the second loop traverse `p_-`

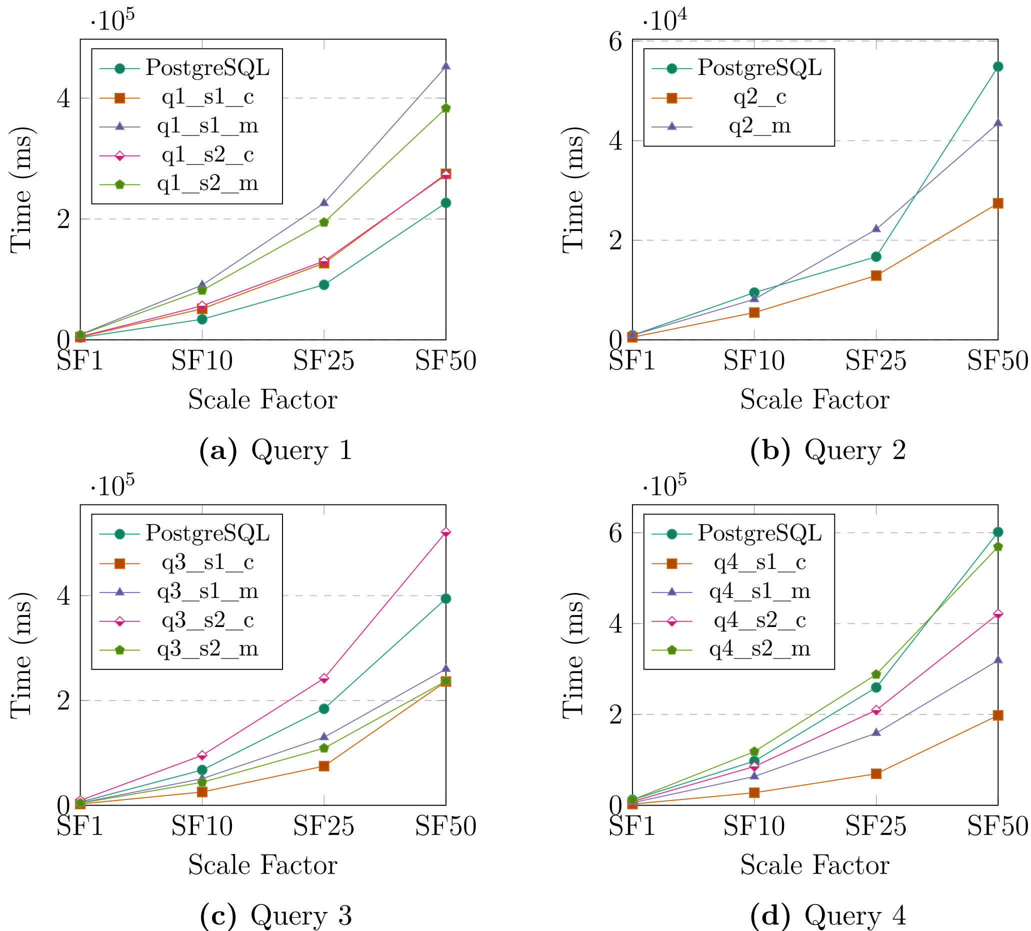


Figure 53. Running time of TPC-H queries 1-4 on PostgreSQL, MongoDB and our compiled approach

`suppliers` in the first loop's opposite direction could enhance performance, as shown in Figure 53b and Table 19, the approach still outperforms both the relational and MongoDB queries without such optimization.

Query 3: This query has one selection per table: Customer, Orders, and Lineitem, with selectivities of ~20%, ~0.041%, and ~99.99%, respectively. `q3_s2_m` starts by filtering out orders; thus, the aggregation pipeline starts from $\mathcal{O}$, then joins $\mathcal{O}$ with $\mathcal{C}$ and the result with $\mathcal{L}$. It performs the appropriate aggregations. Notably, (1) the system scans the index on $\mathcal{O}$'s `_id` field as the first join retrieves ~41.3% of customers, (2) unnesting of $\mathcal{O}$ after the first join is unnecessary since only one customer will be retrieved per order, (3) the second join triggers an index seek on $\mathcal{L}$'s `l_orderkey` field, thus retrieving 0.042% of lineitems. `q3_s2_c` follows the plan shown in Fig-

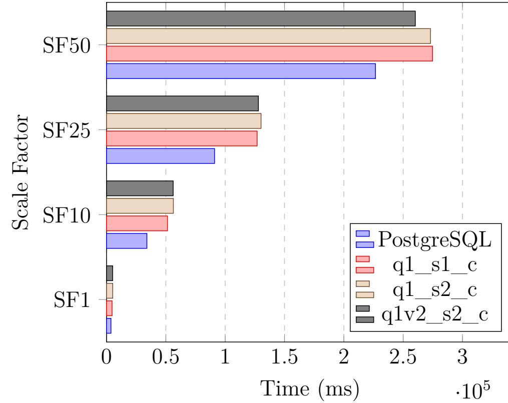


Figure 54. Q1 compiled queries vs PostgreSQL

ure 56. This plan is implemented using three loops. The first two scan $\mathcal{O}$ and $\mathcal{L}$, populate the hash tables, and filter the respective documents of these two collections. The third one scans $\mathcal{O}$, probes the hash tables created in the first two loops, and computes aggregates. Sorting is done afterwards. Scanning $\mathcal{L}$ is the most expensive step in this compiled query, taking ~86% of the time and being by itself slower than what it takes `q3_s2_m` to finish. Figure 53c and Table 19 clearly show such a performance gap and evidence one of the main weaknesses of the experimental setup for the proposed framework: lack of indexes. `q3_s1_m` starts by filtering customers from $\mathcal{COL}$, then unnesting $\mathcal{O}$, filtering orders, unnesting $\mathcal{L}$, filtering lineitems and, lastly, aggregating. `q3_s1_c` has three (nested) loops: The first one scans $\mathcal{COL}$, the second traverses orders from customers, and the third lineitems from orders. There are if-statements before the second and third loops to avoid traversing orders and lineitems not meeting the respective selection criterion. Because the selection on orders filters out most documents (less than 0.001% of orders qualifies), MongoDB can afford to access lineitem for the $lookup using the index and barely touch this table, while the compiled approach is forced to scan it in its entirety. This effect is reminiscent of sideways information passing ([40]), where information about selectivity in one table gets propagated to other tables through join; the compiled approach does not currently take advantage of this, but MongoDB does.

Queries 4 and 18: Query 4 features two selections over orders, one over lineitem,

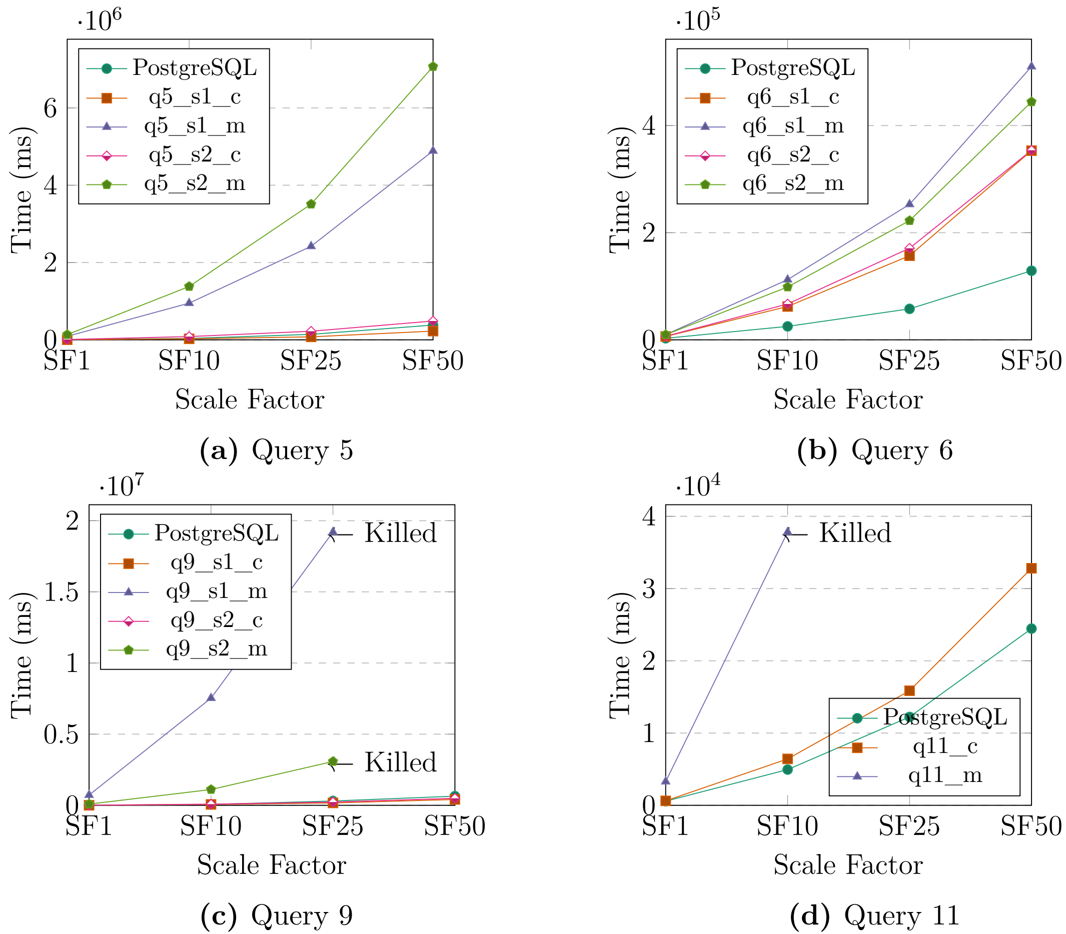


Figure 55. Running time of TPC-H queries 5, 6, 9 and 11 on PostgreSQL, MongoDB and our compiled approach

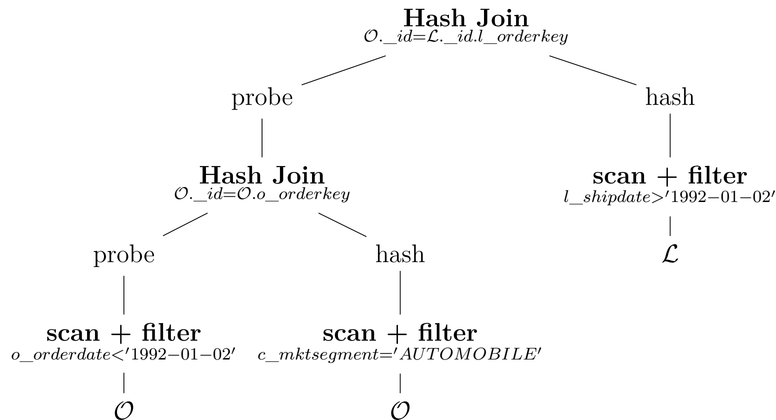


Figure 56. plan for compiled query 3 on normalized schema (S2)

and a correlated sub-query within SQL's exists operator that verifies if an order has at least one lineitem delivered to the respective customer later than its committed

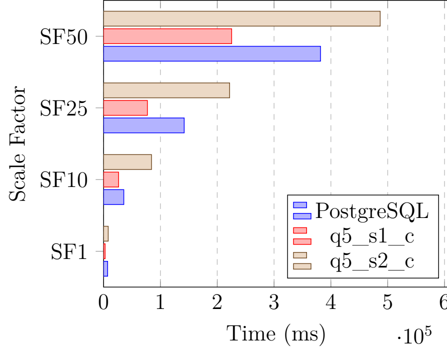
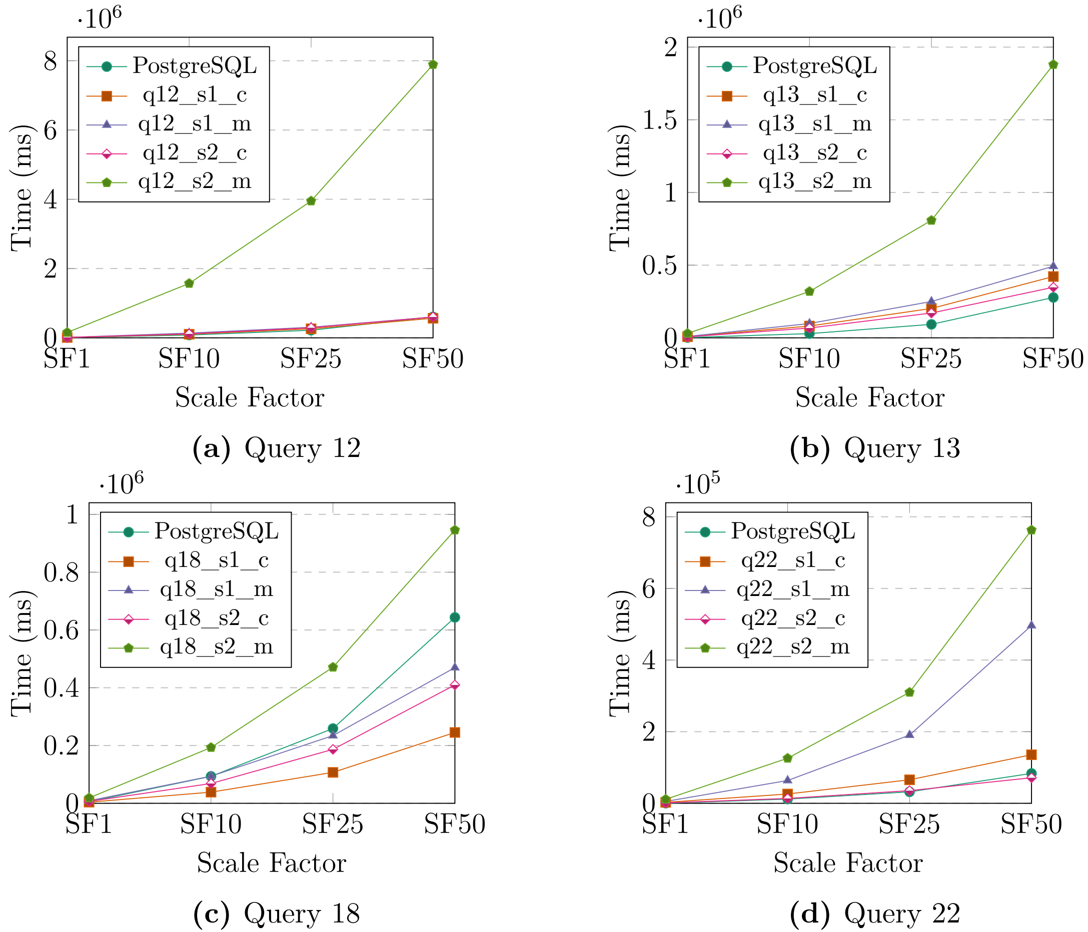


Figure 57. Q5 compiled queries vs PostgreSQL

date. `q4_s2_m`'s pipeline begins with $\mathcal{O}$, filters orders, and then joins with $\mathcal{L}$. MongoDB's $lookup operator nests the matching lineitems, emulating the behavior of SQL's `EXISTS` and the sub-query by (1) filtering out lineitems received by the customer earlier than its committed date and (2) removing orders left with an empty array of lineitems. `q4_s1_m` follows the same approach but unnests orders instead of using $lookup. `q4_s2_c` treats the query as an unnested version of the relational query (i.e., a semi-join). `q4_s1_c` follows the same approach as `q3_s1_c`, using one loop to traverse $\mathcal{COL}$ and two nested loops with if-statements in between. Figure 53d shows the compiled approach outperforms MongoDB queries on their respective schemas, and Table 19 demonstrates that the compiled approach achieves around a 2X consistent speedup on S1 and a 1.4X consistent speedup on S2. Query 18 behaves similarly to query 4 in both MongoDB and the compiled queries. However, a better speedup over S1 is achieved on query 18 than query 4.

Query 5: This query has a selection over region and orders. The query makes use of all tables in the relational schema and all collections of each of the schemas in MongoDB. `q5_s2_m` is a modified version of MongoDB's implementation of query 5 over S2 proposed in previous work ([54, 53]) that tries to mimic the plan in Figure 51b. In practical terms, this means values of `localField` and `foreignField` in S_6 of Figure 49 are replaced with `lineitems.l_suppkey` and `_id` respectively, and the value of `$expr` in S_{10} with



(a) Query 12

(b) Query 13

(c) Query 18

(d) Query 22

Figure 58. Running time of TPC-H queries 12, 13, 18 and 22 on PostgreSQL, MongoDB and our compiled approach

```
["$suppliers.s_nation.n_nationkey", "$customers.c_nationkey"]
```

This change allows MongoDB to use the index on `l_suppkey`, decrease the documents S_6 outputs (i.e. increase join selectivity) and decrease S_{10}'s input. `q5_s2_c` follows the plan in Figure 51b. It creates hash tables for $\mathcal{O}$, $\mathcal{L}$, $\mathcal{S}$ and then traverses $\mathcal{O}$ and probes them. The main bottleneck of this query is located in S_{10} for MongoDB queries, but the compiled queries amortize it through pipelining. This explains the gap between MongoDB queries and the compiled queries shown in Figure 55a. Furthermore, because `n_nationkey` has low cardinality, it is feasible to create an extra hash table with Boolean values when scanning $\mathcal{S}$ at a negligible cost to discard customers not being in the same nation as the supplier that filled their order of parts. This technique is incorporated in `q5_s1_c` and achieves an extra speedup. This is

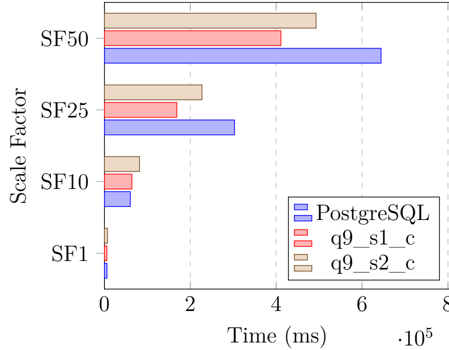


Figure 59. Q9 compiled queries vs PostgreSQL

shown more clearly in Figure 57 and Table 19.

Query 9: This query is similar to query 5. There is a selection over $\mathcal{P}$, but there are no customers. Thus, the pipeline of `q9_s1_m` can only start from $\mathcal{COL}$ to unnest orders, lineitems and then join with $\mathcal{P}$ and $\mathcal{S}$. These unnests increase the number of input documents to join and harm performance as shown in Figure 55c. `q9_s2_m` filters parts from $\mathcal{P}$ first and then joins with $\mathcal{L}$ and $\mathcal{S}$, thus avoiding such problem. `q9_s2_c` and `q9_s1_c` follow a similar approach, even outperforming the relational query as shown in Figure 59.

Query 11: As stated in [54, 53], the way the M-N relationship between parts and suppliers is modeled harms this query's performance in `q11_m` and `q11_c` compared to the relational schema as it forces joinining $\mathcal{P}$ and $\mathcal{S}$. Notable, this can be avoided by embedding PartSupp tuples as nested documents in $\mathcal{S}$ at the cost of more redundancy. During experimentation, `q11_m` reached the $facet operator's 100MB threshold on scale factors 25 and 50, which forcibly aborted the query's execution. Hence, their running times are not reported on Figure 55d, nor Table 19. On `q11_c`, $facet is treated as if joining two sub-queries in a relational schema, which is handled similarly to query 4 for this case. Results in Figure 55d show the compiled approach is close in performance to the relational query despite having to join $\mathcal{S}$ and $\mathcal{P}$.

Queries 12 and 13: Query 13 has a sub-query with a left outer join that counts orders per customer as long as they meet a condition set on attribute `o_comment`.

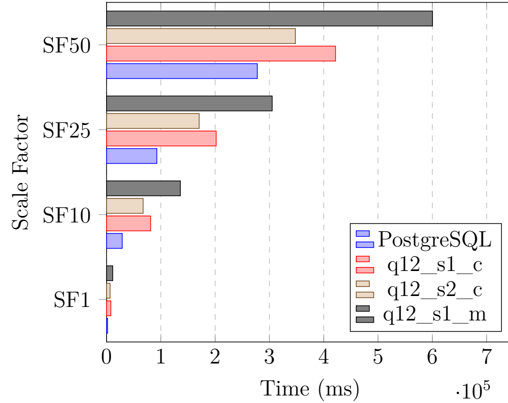


Figure 60. Q12 compiled queries vs PostgreSQL and MongoDB query on S1

`q13_s2_c` manages the left join by creating a hash table with all customer keys and a default counter value of 0, then traversing $\mathcal{O}$, probing the hash table and adding one to the counter matching the value of field `o_custkey`. `q13_s1_c` loops through nested orders inside documents of $\mathcal{COL}$, but it is slower due to the overhead of retrieving nested lineitems, which is unnecessary as the query only needs customers and orders. Figures 58a and 58b show that query 12 behaves similarly to query 13, but scales worse on MongoDB queries where a join is involved. This is expected as query 12 uses orders and lineitems, for which the join between $\mathcal{O}$ and $\mathcal{L}$ is much more expensive than $\mathcal{C}$ and $\mathcal{O}$. Furthermore, Figure 60 evidences that the extra overhead of reading customers on `q12_s1_c` is much less than joining $\mathcal{O}$ and $\mathcal{L}$ in `q12_s2_c`—small enough that `q12_s1_c` is ~8% faster than `q12_s2_c` on average.

Query 22: This query features one sub-query in the from-clause and two nested subqueries (at the same level). One is a scalar uncorrelated sub-query that computes customers' average positive account balance, and the other is an anti-join that ensures customers of interest have not placed any orders. Similar to `q11_s2_m`, `q22_s2_m` uses $facet to compute these sub-queries. This new version of the MongoDB query significantly improves performance compared to the previous work. `q22_s1_m` is kept as described in [54, 53] as it is still faster than using $facet on S1. `q22_s2_c` handles the anti-join by (1) creating a hash table based on the `o_custkey` field from $\mathcal{O}$ and (2) discarding customer documents later on a successful probe. From a relational per-

spective, it pushes down selections unrelated to the table in the outer query but *pulls up* the selection involving the uncorrelated sub-query, keeping track of the number of account balances and adding their values together to subtract from `numcust` and `totalaccountbal` at the end for every customer with an account balance less than or equal to the average. `q22_s1_c` simply checks the size of the array of orders in every document of $\mathcal{COL}$. Similar to other queries on S1, `q22_s1_m` and `q22_s1_c` read more information than they need, making them slower than queries on S2 due to the extra overhead. Figure 58d illustrates this.

Limitations and Analysis

This test acknowledges some critical limitations. A prominent one is the high homogeneity of the dataset, originating from a relational dataset, enabling the transformation of every outer join into an inner join and eliminating all nesting. This optimization aligns the plans closely with relational counterparts. Additionally, the ability to specialize code for a single data type per attribute contributes to simplicity, as the metadata requirements are akin to those of a relational database's system tables.

Despite these limitations, the test yields valuable insights. The summary, outlined in Table 19, illustrates that the proposed approach consistently outperformed MongoDB across all queries and schemas, except for one instance (query 3 on S2, discussed earlier). The improvements range from modest (5.9%) to substantial (97%), with an average improvement that is notably significant, approaching 50%. Furthermore, the observed improvement either sustains or increases as the database size grows in nearly all cases, indicating robust scalability compared to MongoDB. Notably, the proposed approach achieved these results without employing indices or additional performance-enhancing methods (caches, multi-threading, etc.), relying solely on the described optimizations. In contrast, indices were provided for MongoDB, and multiple versions of each query were executed to maximize their aggregated pipeline's

performance.

In comparison with PostgreSQL, a more mature system, the proposed approach demonstrates competitive performance. It outperforms PostgreSQL in some queries (2, 4, 5, 9, 18) and lags slightly in others (1, 11, 12, 13). Some queries show varying results between schemas (3, 22), but seldom does the proposed approach trail by a significant margin (query 6). Thus, despite acknowledged limitations, the results substantiate the viability and effectiveness of the proposed approach.

CHAPTER VII

RELATED RESEARCH

1 Introduction

The research discussed in this chapter is highly pertinent to the JSON data model, its formal representation as JSON Trees, and the intricacies of cost-based optimization within JSON document stores. The core of this thesis hinges on a JSON data model rooted in labeled trees. Notably, various researchers concur that JSON documents naturally manifest as labeled trees, although a unanimous consensus regarding the precise labeling scheme remains elusive. The comprehensive exploration of these researchers' findings is presented in Section VII.2.

It is important to note that while alternative formalisms for JSON exist, only those directly relevant to this thesis are referenced here. The thesis is centered on DataPilots, which draw inspiration from XML DataGuides. Consequently, the ensuing sections delve into a comparative analysis of how other initiatives have adapted DataGuides to suit the idiosyncrasies of JSON, all of which are detailed in Section VII.3.

2 JSON Data Model

[14] formalizes MongoDB's data model and a fragment of the aggregation framework query language called MQuery, comprised of stages *match, unwind, project, group* and *lookup*—each named after the corresponding pipeline stage in MongoDB's aggregation

framework. The authors outline the (rough) correspondence between MQuery stages and NRA operators *select, unnest, project, nest* and *left join* respectively, and claim MQuery and NRA to have equivalent expressive power. Furthermore, they analyze fragments of MQuery—each denoted by $\mathcal{M}^{\alpha}$, where α are the initials of the operator names in the fragment (e.g. $\mathcal{M}^{MUPG}$)—to determine their computational complexity.

In their formalization, the authors view a JSON Object as an unordered set of key-value pairs such that a key is always a unique string and a value is either a literal (i.e. any atomic value such as an string, number, date, etc.), object or an array of values. MongoDB documents are viewed as unordered, unranked, node-labeled and edge-labeled trees, where labels belong to one of the three disjoint sets: K of keys, I of (non-negative integer) array indexes and V of node labels to denote literals. A MongoDB document is then defined as a tuple (N, E, L_n, L_e) where N is a set of nodes, E a set of edges, (N, E) is a tree, $L_n : N \to V \cup \{'\{\!|\,|\!\}', '[\,]'\}$ is a node labeling function and $L_e : E \to K \cup I$ is an edge labeling function, such that outgoing edges from nodes labeled $'\{\!|\,|\!\}'$ must be labeled by keys, outgoing edges from nodes labeled by $'[\,]'$ must be labeled by distinct array indexes and any node labeled by a literal must be a leaf. A collection is viewed as a forest that can be transformed into another forest by any MQuery stage.

An MQuery (or MQuery pipeline) has the form $s_1 \rhd s_2 \rhd \ldots s_k$, where each $s_i | 1 \leq i \leq k$ is a stage in the pipeline and $s_a \rhd s_b$ denotes output documents from stage s_a are stage s_b's inputs. The authors define 5 stages:

- **Match stage (μ):**
    - μ_{φ}: Select tree based on criterion φ, which only supports path existence or path equality comparisons to constants (the authors claim that adding other order comparison operators besides equality will not affect their com-

plexity and expressivity analyses, thus they omit them for simplicity), thus the values of two paths cannot be compared to each other (note that this happens in MongoDB's \$match stage as well without using the `$expr` -operator). When using equality to compare a path p that evaluates to an array a with a value v, this stage checks (1) whether v is exactly a or (2) v is within a.

  - **divergence with MongoDB:** for MongoDB's `$match`, $p = null$ for some path p holds if p evaluates to `null` or is missing, whereas for μ it only holds in the former case.

- **Unwind stage (ω):**

  - ω_p: flatten array reached by path p and output a new tree per element in the array. For any tree t, if p is not defined in t or evaluates to a non-array value, t is discarded from the output.

  - ω_p^+: Same as ω_p, but preserves t if p is not defined in t or evaluates to a non-array value.

- **Project stage (ρ):**

  - ρ_P: project away, rename or create new paths. P is a sequence of the form p or q/d, where p is a path to be kept, and q a new path whose value is equal to the value d evaluates to. Unlike φ in stage μ, d can be a boolean expression β allowing the value comparison of two paths, or a conditional expression $(\beta?d_1 : d_2)$—read as: if β, then evaluate to d_1, else evaluate to d_2. The key *_id* is kept by default.

  - $\rho_P^{\not{id}}$: Same as ρ_P, but key *_id* is projected away by default. This would be (roughly) equivalent to using `$project` in MongoDB like: `{$project: {_id:0, ...}}`.

- **divergence with MongoDB:** Unlike MongoDB's `$project` behavior for equality comparisons between arrays and values, where only check (1) described in stage ω is performed, β performs both checks in this stage.

- **Group stage (γ):**
  - $\gamma_{G:A}$: group trees based on condition G and collect values for each group based on condition A.
  - **divergence with MongoDB:**
    * when grouping by one path, `$group` creates a single group for documents where the grouping path is missing or evaluates to `null`. e.g. for documents $doc_1, doc_2, \ldots doc_n$, the expression $\gamma_{g/y:\ldots}$ puts all doc_i where y is missing or evaluates to `null` in a group with $_id = \{\!\![g : null]\!\!\}$.
    * when grouping by multiple paths, `$group` creates two groups, one for documents where all grouping paths are missing and another for documents where all grouping paths are null. e.g. for documents $doc_1, doc_2, \ldots doc_n$, the expression $\gamma_{g_1/y1,\ldots,g_m/y_m:\ldots}$ puts documents where all $y_j | 1 \leq j \leq m$ are missing in a group with $_id = \{\!\![\,]\!\!\}$, while documents where all y_j evaluate to `null` are put in a group with $_id = \{\!\![g_1 : null, \ldots, g_m : null]\!\!\}$.

- **Lookup (λ):**
  - $\lambda_p^{p_1=C.p_2}$: joins input documents with documents from collection C using paths p_1 (evaluated on input documents) and p_2 (evaluated on documents in C) to express the join condition, then stores matching documents in an array under path p.

The semantics of these stages can, in some cases, abstract the use of MongoDB operators with MongoDB pipelines. For instance, when renaming paths inside arrays

one can use the combination of `$project` and `$map`, or just `$project`. For instance, the authors (in page 7) show the MQuery:

$$\rho_{awsName/awards.award, awsYear/awards.year}(\cdots)$$

which can be translated as:

```
{$project: {
  awsNames: {
    $map:{
      input:"$awards",
      as:"aws",
      in:"$$aws.name"
  } },
  awsYear: {
    $map:{
      input:"$awards",
      as:"aws",
      in:"$$aws.year"
  } } }
}
```

or the more simple pipeline:

```
{$project: {awsNames:"$awards.name", awsYear:"$awards.year"} }
```

Note that even if both MongoDB pipelines produce the same result, `$map` inherently produces a new array, which could be deemed as an incorrect translation of an algebraic operator depending on how such operator is defined. For this particular case, the authors mention that arrays get "disassembled" when renaming paths within, which is made explicit from the combination of `$project` and `$map` (implementation-wise,

using just `$project` performs better).

To perform their analyses of NRA and MQuery's expressive power, the authors define new higher order pipeline stages to "linearize" nested queries on (tree-shaped) NRA expressions. This is necessary because MQuery is based on MongoDB's aggregation pipeline, where the first stage fetches documents and each subsequent stage acts over the output of the previous stage, which makes it impossible to support nested queries. To this end, the authors define the new pipeline stages:

$$spec2 = \rho^{\not{\text{id}}}_{\text{origDoc}/\epsilon,\text{actRel}/[1,2]} \triangleright \omega_{\text{actRel}} \triangleright \rho_{\text{actRel},\{\text{rel}i/((\text{actRel}=i)?\text{origDoc:dummy})\}_{i=1,2}}$$

where:

1. $\rho^{\not{\text{id}}}_{\text{origDoc}/\epsilon,\text{actRel}/[1,2]}$ creates (and projects) the new path "origDoc" whose value is the input document itself and the path "actRel" with value [1, 2]. e.g.
   document ⦃ name: jhon, age: 20 ⦄
   becomes: ⦃ origDoc:⦃ name: jhon, age: 20 ⦄, actRel:[1,2] ⦄

2. ω_{actRel} unwinds "actRel". The purpose of this stage is to duplicate the original document so that two independent pipelines operate over different copies of the same document. e.g. from the previous stage, two new documents are created:
   ⦃ origDoc:⦃ name: jhon, age: 20 ⦄, actRel:1 ⦄
   ⦃ origDoc:⦃ name: jhon, age: 20 ⦄, actRel:2 ⦄

3. $\rho_{\text{actRel},\{\text{rel}i/((\text{actRel}=i)?\text{origDoc:dummy})\}_{i=1,2}}$ projects "actRel" and creates a new property called "rel$i$", where i is the same number as the value of "actRel". "dummy" is some non-existent path that causes ρ to ignore (project away?) "rel$i$" for $i \neq$ actRel.from the previous stage, documents become:
   ⦃ actRel:1, rel1: ⦃ name: jhon, age: 20 ⦄ ⦄
   ⦃ actRel:2, rel2: ⦃ name: jhon, age: 20 ⦄ ⦄

$$pipeline(q_1, q_2) = spec2 \triangleright subq_1(q_1) \triangleright subq_2(q2)$$

where q_1 and q_2 are two independent sequences of stages (i.e. two pipelines) that operate over the same input documents and $subq_j(q_j), j \in \{1, 2\}$ is defined as $subq_j(s_1) \triangleright \cdots \triangleright subq_j(s_n)$, for $q_j = s_1 \triangleright \cdots \triangleright s_n$. Because $spec2$ modifies the schema of documents, stages in q_1 and q_2 must be adapted through $subq_1$ and $subq_2$ respectively. The authors provide a table in page 10 that shows the translation of a MQuery stage s using $subq_j(s)$. For instance, $subq_j(\omega_p^+) = \omega_{\text{rel}j.p}^+$. Note the resemblance of this stage to MongoDB's `$facet` stage:

```
{ $facet: {
      <outputField1>: [ <stage1>, <stage2>, ... ],
      <outputField2>: [ <stage1>, <stage2>, ... ],
      ...
  }
}
```

which computes independent pipelines over the same set of input documents and stores the output of each one in a separate array. In `$facet`, the MongoDB equivalent of MQuery's $spec2$ is done by the system, thus there is no explicit document duplication.

The paper [37] introduces J-Logic, a logical framework rooted in Datalog. This framework is designed for the analysis of JSON querying, employing a Sequence Datalog language that operates on sets of sequences using paths as its foundation. In this formalism, JSON documents are conceptually represented as labeled trees, mirroring the approach outlined in this thesis.

The paper places particular emphasis on the generation of data elements, such as keys, edge labels, and nodes within a document, as outcomes of operations performed

between documents. This methodology results in the encapsulation of operation results within a single JSON document, as opposed to a collection of documents.

To address issues like naming conflicts or the necessity for creating new keys, the authors propose a technique called *packing*. Packing involves concatenating a key sequence to generate a new key. For instance, the authors generalize the cartesian product as follows:

Consider two documents, D_1 and D_2, defined as: $D_1 = \{\!\{k_1^1 : v_1^1, k_2^1 : v_2^1, ..., k_n^1 : v_n^1\}\!\}$, $D_2 = \{\!\{k_1^2 : v_1^2, k_2^2 : v_2^2, ..., k_m^2 : v_m^2\}\!\}$. Their cartesian product is represented as:

$$
\begin{aligned}
\{\!\{ \quad & \langle k_1^1 \cdot k_1^2 \rangle : \{\!\{d_1 : \{\!\{k_1^1 : v_1^1\}\!\}, d_2 : \{\!\{k_1^2 : v_1^2\}\!\}\}\!\}, \\
& \langle k_1^1 \cdot k_2^2 \rangle : \{\!\{d_1 : \{\!\{k_1^1 : v_1^1\}\!\}, d_2 : \{\!\{k_2^2 : v_2^2\}\!\}\}\!\}, \\
& \ldots \\
& \langle k_1^1 \cdot k_m^2 \rangle : \{\!\{d_1 : \{\!\{k_1^1 : v_1^1\}\!\}, d_2 : \{\!\{k_n^2 : v_n^2\}\!\}\}\!\}, \\
& \langle k_2^1 \cdot k_1^2 \rangle : \{\!\{d_1 : \{\!\{k_2^1 : v_2^1\}\!\}, d_2 : \{\!\{k_1^2 : v_1^2\}\!\}\}\!\}, \\
& \ldots \\
& \langle k_n^1 \cdot k_m^2 \rangle : \{\!\{d_1 : \{\!\{k_n^1 : v_n^1\}\!\}, d_2 : \{\!\{k_m^2 : v_m^2\}\!\}\}\!\} \quad \}\!\}
\end{aligned}
$$

Here, $\langle k_i^1 \cdot k_j^2 \rangle$ signifies a packed key resulting from the concatenation of keys k_i^1 and k_j^2.

Note that this cartesian product is non-commutative due to the nature of packed keys, which are formed by concatenating the key from the right-hand side document to that of the left-hand side document. Hence, swapping the documents would yield different packed keys.

To address this, the authors propose transitioning from packed keys to "Object Identifiers" (OIDs), as proposed in ILOGIC[38]. OIDs are independent of the order in which the arguments appear, relying instead on the key pairs themselves. This adjustment renders the cartesian product commutative.

In summary, the authors highlight the potency of J-Logic as a language, particularly in handling non-recursive queries over deeply nested data. The concept of

packing serves as a versatile tool, enabling the application of various common techniques used in traditional relational optimizers, such as duplicate elimination.

In [39], the authors introduce an innovative approach to compute the similarity between JSON documents, which they aptly term the JSON Edit Distance (JEDI). The authors utilize trees as their foundational structure for representing JSON data, but unlike the methodology presented in this manuscript literals are stored as node labels, and the value “null” is assigned when a node signifies an object or an array.

The authors also introduce a secondary node labeling function, which serves to assign types (namely, object, array, key, or literal) to each node within the JSON tree. Additionally, they introduce a strict, partial ordering mechanism, denoted as $<_S$, which they refer to as the "sibling ordering." This ordering applies exclusively to elements within arrays and proves to be especially beneficial for JSON similarity queries. It provides a significant advantage over the conventional representation of array indexes using numerical edge labels, as the latter approach introduces an error of magnitude $O(n)$.

To illustrate this point, the authors present a straightforward example: consider two JSON documents containing arrays, [‘A’, ‘B’, ‘C’, ‘D’] and [‘B’, ‘C’, ‘D’]. In this scenario, the array index of each element (represented by edge labels) differs due to the absence of ‘A’ in the second array, resulting in a substantial $O(n)$ error.

It is noteworthy, however, that the authors’ approach is not fundamentally incompatible with this manuscript’s model of JSON documents as sets of paths. In fact, the concept of sibling ordering can be seamlessly incorporated into this work’s framework:

For any two paths, p and q, the sibling order $p <_S q$ holds true if both paths share a common prefix, denoted as r, satisfying the following:

$$(\exists x \in \mathbb{Z}^+ \cup \{0\})(\exists y \in \mathbb{Z}^+ \cup \{0\})[r \cdot x \preceq p \wedge r \cdot y \preceq q \wedge x < y]$$

This adaptation allows for the integration of the sibling ordering concept while

preserving the compatibility between the two approaches.

Additionally, the authors allude to the potential utility of JEDI in identifying schemas for similar documents. However, they do not delve into further details or provide an in-depth exploration of this application.

3 DataPilot and Query Optimization

In [35], DataGuides were introduced as a concept for XML databases. Much like DataPilots, DataGuides serve the purpose of offering structural overviews of XML data, acting as dynamic schemas, storing statistics, and contributing to query optimization. The paper puts forth a formal definition of a DataGuide and outlines algorithms for its creation and maintenance.

The authors underline that DataGuides are rooted in the transformation of a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA), as mentioned in [60], with the NFA originating from the OEM data model. Consequently, an OEM model can give rise to multiple DataGuides, akin to the various equivalent DFAs derived from the NFA. Among these DataGuides, selecting one to maintain data summaries (both structural and statistical) is not a straightforward task. For instance, a minimal DataGuide, while the most concise, proves to be the most challenging to keep up-to-date in the face of updates. Notably, this issue does not apply to DataPilots, given that JSON data is inherently deterministic. However, it's worth noting that multiple collections may share the same DataPilot, as DataPilots discard array labels and consolidate them into the $\star$ label.

Additionally, the paper explores the utility of DataGuides as *path indexes*, where path existence is verified by referencing the DataGuide and subsequently extracting object IDs from other accessible indexes upon successful verification. This approach can also be adapted for DataPilots. It's worth noting that the paper does not delve into estimating data or detail how the statistics maintained in a DataGuide are updated based on a sequence of query operators.

[45] introduces a versatile system designed for querying a wide array of heterogeneous datasets stored in various formats. This system, named Proteus, serves as an analytical query engine with the adaptability of a general-purpose query engine. It achieves this by employing a modular architecture that relies on plug-ins to manage statistics and calculate query costs for different data types. Consequently, Proteus can be easily extended to support new data formats without the need for data conversion into a uniform representation.

One notable feature of Proteus is its ability to query datasets with varying formats directly, avoiding the necessity of transforming data into a standardized structure. This approach is especially advantageous when dealing with format-variable workloads.

The authors of the paper highlight a key distinction between Proteus and specialized data model systems like MongoDB or CouchDB. While the latter require users to handle data integration through middleware when conducting cross-dataset queries, Proteus offers a unified data model. This model employs monoid comprehension calculus as its primary query language, providing both flexibility and expressiveness. This flexibility allows for the mapping of other query languages as needed.

In terms of query optimization, the approach presented in this thesis shares some similarities with Proteus:

- Both systems employ nested relational algebra as an internal representation of queries. Proteus initially translates queries into calculus expressions and then restructures them as nested relational algebra trees.
- Both leverage the code compilation strategy introduced by Thomas Neumann in the Hyper system ([62],[61]).
- Both follow a multi-step approach for generating query plans, including rule-based rewriting followed by cost-based transformations.

However, it's important to note that the primary focus of Proteus is to serve as a general-purpose query engine adaptable to diverse data types and formats. In contrast, the approach described in this manuscript centers on a theoretical JSON-centric query engine. While both systems use structural indexes, they do so differently. Proteus employs structural indexes to optimize data access by navigating through data and extracting only necessary values. In contrast, a Data Pilot is utilized in this manuscript's approach to aid in cardinality estimation and provide optimization hints when more suitable indexes are unavailable.

In their work [52], the authors present an approach to representing JSON documents as tree structures. They introduce three distinct types of nodes within this tree, each representing JSON objects, JSON arrays, and JSON scalars. In essence, they construct JSON documents as a JSON DOM tree.

The DataGuide associated with this JSON DOM tree serves as its "skeleton". Here, leaf scalar values are replaced with data type and length information. To form a Collection DataGuide, the DataGuides from all documents in the collection are merged. During this merging process, duplicate tree paths are eliminated, provided they exhibit the same node type throughout the entire path. Tree paths where at least one node type differs at any point are considered distinct.

When merging two paths p and q, the authors employ two strategies when $p = q$ and node types differ only at the leaf nodes:

- If both p and q are scalars, then either one is retained in the DataGuide, but the type information in the leaf node is generalized. For example, if the conflicting types are string and number, the DataGuide stores the string type and the maximum length between both.

- Otherwise, both p and q are stored separately, each with its corresponding type information and length.

Notably, their DataGuide differs from the Data Pilot discussed in this manuscript

in several key aspects:

- A Data Pilot stores only one instance of each path, and array indexes on a path are consistently replaced by a special character ($\star$). Consequently, paths with different array indexes are considered equivalent.

- While a Data Pilot records information about path occurrences grouped by type, it extends its focus beyond type length.

- The authors leverage the Data Guide to establish a virtual relational view, allowing them to query relational and JSON data in tandem. Conversely, a Data Pilot serves as a tool to provide optimization hints to the query optimizer, primarily when no suitable index is available.

These distinctions underline the unique functionalities and applications of the DataGuide and Data Pilot in their respective contexts.

In their work, [76] introduces a comprehensive JSON taxonomy and conducts benchmarking exercises on various popular binary JSON specifications. Their evaluation employs the "SchemaStore open-source test suite" — a rich resource encompassing both schema-driven and schema-less JSON datasets. Notably, the primary focus of their research revolves around exploring novel space-efficient representations for JSON data in binary form.

Within the framework of their taxonomy, the authors introduce a set of statistics aimed at summarizing document structure and categorizing individual documents. To facilitate this, they have developed a tool called JSON Stats (accessible at `https://www.jsonbinpack.org/stats/`). These statistics are designed for analyzing individual documents and do not extend to the level of collections. Additionally, some of these metrics are tightly coupled with specific binary serialization formats, and their applicability to query optimization remains unaddressed. In contrast, a Data Pilot is serialization-agnostic and centers its focus on enhancing query optimization rather than document categorization.

By incorporating a Data Pilot approach, it becomes feasible to extend and adapt some of these statistics for collection-level analysis, offering approximate and repurposed insights tailored to query optimization. For instance, the authors' metric "document redundancy," which quantifies the percentage of duplicated values within a JSON document, can be re-envisioned to gauge the homogeneity within a collection. Indeed, a Data Pilot implicitly relies on this concept when estimating the cardinality of a selection. In cases where the values from a path referenced within a selection exhibit heterogeneity, it may lead to improved cardinality estimates, demonstrating the potential for enhancing query optimization strategies.

In their research presented in [17], the authors focus on visually representing the intricate relationships between schemas within different services operating within the same domain or web API. The motivation behind this endeavor stems from the challenge of dealing with data from disparate providers. Often, when publishing or consuming data from various sources, it becomes essential to navigate and extract the required information from service responses, even when those services lack comprehensive documentation.

To facilitate this exploration, the authors develop a dedicated plugin for the popular Eclipse Integrated Development Environment (IDE). This plugin empowers users to generate insightful visualizations that unveil the underlying relationships between services. This process unfolds across three distinct phases:

1. **Pre-discovery Phase:** Initially, the authors define the JSON grammar using the Eclipse Xtext framework. This step yields not only the JSON metamodel but also the essential tooling, known as the "injector", necessary for transforming documents into models conforming to the JSON metamodel. Consequently, every document can be seamlessly converted into an Ecore model.

2. **Single-Service Discoverer Phase:** In this stage, an Ecore model is generated, providing a general description of the models encompassing all injected

document models.

3. **Multi-Service Discoverer Phase:** Building upon the models from the previous phase, this step produces a composite model that amalgamates the various individual models.

Identifying the correspondences between these models, however, presents a computational challenge, as it is an NP-hard problem, reducible to graph isomorphism. Nevertheless, the authors posit that the complexity of finding an approximate solution remains manageable, particularly when there are numerous similarities between the models. This expectation holds since these models inherently belong to the same application domain.

Comparatively, this paper diverges from the focus of this manuscript, as it primarily seeks to enhance web API documentation by elucidating the relationships among documents returned by services. Nonetheless, an intriguing application of their work to query optimization emerges. This could involve reimagining what they term "services" as operations in a query and constructing a dependency graph. Within this graph, nodes would represent operations, while the edges' weights would signify cardinality estimations. The ultimate goal would be to minimize a cost function applied throughout a path from the query's input to the target output, thereby identifying the most efficient path to reach the desired query output.

In [9], an algorithm for schema inference is presented, which utilizes a map-reduce approach. The algorithm is executed within the Apache Spark framework and incorporates a parameter to control the precision and conciseness of the inferred schema. Regardless of the chosen precision level, the inferred schema includes information about mandatory fields (labels that appear in all records) and optional fields (labels that appear in some records) within JSON records. In the most precise schema, the algorithm identifies co-occurrences of fields. Notably, the focus of this work is on inferring the structural schema of extensive JSON datasets, without delving into the

potential applications of such schema for query optimization, although it underscores its significance for this purpose. Additionally, this work does not provide frequency counts of types, which are covered in a separate paper by the same authors ([8]).

In [18], a JSON to Relational mapping layer, named Algo, is introduced. It outlines two strategies for mapping a JSON collection into a relational schema: (1) employing a single table that retains type information, resulting in the relational schema *collection(objid, keystr, valstr, valnum, valbool)*, and (2) using one table per primitive type, resulting in separate relational schemas: *collection_str(objid, keystr, valstr)*, *collection_num(objid, keystr, valnum)*, and *collection_bool(objid, keystr, valbool)*. Queries in the Algo/SQL language described in the paper are adaptable to either strategy, as they are translated into SQL queries customized for the chosen approach.

Notably, the authors' work relies on the underlying RDBMS for query optimization and primarily focuses on the mapping layer, aligning with the approach used in XML-enabled databases. In contrast, this thesis explores optimization strategies within a JSON model based on labeled trees, utilizing a structural summary data structure inspired by XML DataGuides.

In [46], the authors introduce the concept of utilizing a miner, inspired by XML research[57], for schema extraction in JSON data. Their approach also considers the incorporation of statistics during schema construction. They employ a data structure termed the "Structure Identification Graph", where nodes and edges indicate the presence of structural properties in specific JSON documents, including references to the documents. Due to potential memory usage concerns, the authors present a more concise version of this structure, referred to as the "Reduced Structure Identification Graph", which records frequency of appearances only. The authors explore applications of the initial structure in detecting potential schema outliers and discuss how both versions can be employed for calculating similarity measures and gathering statistics, including frequency counts and derived information from these counts.

In comparison to this thesis, the authors' work primarily focuses on schema extraction and outlier detection. Notably, their work does not further delve into the application of these graphs for query optimization.

In [11], the authors employ a concept reminiscent of this thesis' Data Pilots, viewing them as sets of paths. They introduce a repository termed a *dictionary*, comprising key-value pairs or tuples. Each key represents a potential path existing within any document from a collection, while its corresponding value consists of a set of all *absolute paths* originating from the root node leading to that key. For example, a tuple from the paper might resemble (year, {year, info·year, film·details·year, description·year}).

The paper primarily focuses on utilizing reformulation rules derived from a collection's dictionary. These rules automatically rewrite queries to ensure that all absolute paths linked to every path used within the query are appropriately considered. However, unlike this thesis, the paper does not discuss considerations related to cost-based query optimization, nor does it address the utilization of path occurrences in query rewriting.

In the AsterixDB project ([5, 13, 67]), queries are processed using a query algebra known as Algebricks, with optimization revolving around algebraic expressions ([12]). While this methodology shares similarities with the framework proposed in this thesis, significant distinctions exist.

Firstly, AsterixDB operates as a standalone system, with Algebricks tailored to function alongside the SQL++ query language, executing query plans through Hyracks—a framework facilitating parallel execution of data-flow jobs ([13]). Conversely, the document algebra introduced in this thesis is agnostic to both system architecture and query language.

Secondly, optimization within Algebricks relies on rule-based mechanisms, contrasting with the proposed cost-based approach in this thesis. Here, paths and their occurrences are leveraged to generate cardinality estimations when suitable indices

are unavailable.

Thirdly, while Algebricks maintains a data-model-agnostic stance, the document algebra in this thesis specifically targets JSON data. As a result, Algebricks necessitates language developers to furnish families of operators to translate abstract query plans into physical operators, alongside providing data-model-specific rewrite rules for the optimizer and an implementation for a Metadata interface supplying data information such as schema and keys. Conversely, the proposed document algebra's JSON focus obviates the need for such extensive support. However, in pursuit of system independence, the proposed approach mandates translations from diverse query languages to the document algebra, along with specialized code generation for various back-ends.

Lastly, in AsterixDB, a tree-like structure akin to a Data Pilot is employed to store schema information. However, its primary purpose lies in tuple compaction, aiming to minimize storage requirements, particularly in highly heterogeneous collections, as elucidated in [4]. In contrast, this thesis leverages Data Pilots at the logical layer, rather than the physical one, to aid a query optimizer in making informed decisions regarding optimization strategies when suitable indices are unavailable.

In [70], the authors introduce a rule language for JSON, termed constrained tree-rules, grounded in automata theory. Their contribution lies in devising a rewriting-based reasoner tailored for JSON document stores, enabling query composition and rewriting. The paper's primary objective centers on empowering users to formulate queries that operate seamlessly across heterogeneous data, all without necessitating prior knowledge of the schema. In this regard, the authors prioritize flexibility over efficiency, diverging from the primary motivation of this thesis. Their rule-based query rewriting technique hinges on a dataguide-like structure akin to the Data Pilot proposed in this thesis, facilitating the generation of potential query rewritings.

In [74], the authors introduce the *JXPLAIN* algorithm, designed to extract schema information from a collection. Unlike the structure of Data Pilots, the author's

schema inference approach refrains from immediately assuming that every document mirrors a tuple, as typically observed in relational databases. Instead, it accounts for scenarios where a document might encapsulate an entire collection. To discern between these cases, documents undergo classification as tuple-like or collection-like based on various heuristics. While this approach may offer advantages over the simplistic assumption of document uniformity inherent in Data Pilots, it comes at the cost of slower extraction processes.

Moreover, a key disparity between the focus of the paper and this thesis lies in their respective research objectives. While the paper prioritizes the provision of highly accurate and precise schema descriptions, it does not delve into discussions regarding the utilization of such schemas and the information they contain to aid in query rewriting or to facilitate cost-based optimization.

In [56], the authors introduce JSONoid, a tool dedicated to distributed schema discovery using monoidal structures. Their approach to schema construction is straightforward and intuitive: they extract the schema of each individual document and merge them into a schema that encompasses the entire collection. This principle aligns with the methodology employed in this thesis for creating a Data Pilot. Similarly, both the paper and this thesis gather metadata about the collection and utilize it to generate cardinality estimations. However, the execution of this principle and the aim of utilizing the generated schema and gathered metadata differ significantly between the two approaches.

In this thesis, the Data Pilot is proposed as a logical data structure that, in conjunction with the Data Pilot algebra, enables the production of cardinality estimates at each stage of a query. It also provides hints to an optimizer regarding whether a specific optimization should be applied or not. Conversely, the paper utilizes a combination of multiple statistical data structures such as Bloom Filters and HyperLogLog to estimate the collection's structure and approximate cardinality. The objective is to furnish analysts with insightful information about the data, with discussions

on potential applications including outlier detection and constraint discovery (e.g., suggesting possible foreign-key-like constraints).

In summary, the paper primarily focuses on data exploration, whereas this thesis emphasizes aiding cost-based optimization through path occurrences and derived statistics from path occurrences.

In [65], the authors present the *Zed* data store, which harnesses the *Zed super-structured data model*, a comprehensive framework encompassing both JSON and relational data models. This data model encompasses a robust type system featuring primitive types (such as int32, string, etc.), complex types (including record and array), named types, and the null type. Each document in the Zed system is annotated with its corresponding type information, and the system relies solely on the type system to represent the schema of a collection.

Regarding analytics performance, the system adopts a dynamic columnar storage approach, wherein columns are constructed based on the types identified from individual documents. However, the specifics of this process are not elaborated upon. Furthermore, there is no explicit mention of how the schemas are utilized for query optimization, nor is there clarity on whether statistics are gathered as part of the system's functionality.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. Foundations of databases, volume 8. Addison-Wesley Reading, 1995.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L Wiener. The lorel query language for semistructured data. International journal on digital libraries, 1:68–88, 1997.

[3] Silberschatz Abraham, F Korth Henry, et al. Database system concepts. McGraw-Hill Education, 2019.

[4] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. An lsm-based tuple compaction framework for apache asterixdb (extended version). arXiv preprint arXiv:1910.08185, 2019.

[5] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source bdms. Proc. VLDB Endow., 7(14):1905–1916, oct 2014.

[6] Lance Ashdown. Oracle database sql tuning guide, 19c. https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/sql-tuning-guide.pdf, dec 2021.

[7] Nikolas Askitis and Ranjan Sinha. Hat-trie: a cache-conscious trie-based data structure for strings. In Proceedings of the thirtieth Australasian conference on Computer science-Volume 62, pages 97–105. Citeseer, 2007.

[8] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Counting types for massive json datasets. In Proceedings of The 16th International Symposium on Database Programming Languages, pages 1–12, 2017.

[9] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive json datasets. In The VLDB Journal, volume 28, pages 497–521, 2019.

[10] Mohamed-amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Carlo Sartiani, Mohamed-amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, Mohamed-amine Baazizi, Université Pierre, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schema Inference for Massive JSON Datasets. 2017.

[11] Hamdi Ben Hamadou, Faiza Ghozzi, André Péninou, and Olivier Teste. Schema-independent querying for heterogeneous collections in nosql document stores. Information Systems, 85:48–67, 2019.

[12] Vinayak Borkar, Yingyi Bu, E. Preston Carman, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J. Carey, and Vassilis J. Tsotras. Algebricks: A data model-agnostic compiler backend for big data languages. In Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, page 422–433, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive comput-

ing. In Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE '11, page 1151–1162, USA, 2011. IEEE Computer Society.

[14] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of mongodb queries. In 21st International Conference on Database Theory (ICDT 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[15] Pierre Bourhis, Juan L Reutter, and Domagoj Vrgoč. Json: Data model and query languages. Information Systems, 89:101478, 2020.

[16] Tim Bray. Rfc 8259: The javascript object notation (json) data interchange format, 2017.

[17] Javier Luis Cánovas Izquierdo and Jordi Cabot. Discovering implicit schemas in json data. In Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings 13, pages 68–83. Springer, 2013.

[18] Craig Chasseur, Yinan Li, and Jignesh Patel. Enabling json document stores in relational systems. In Proceedings of the WebDB Conference, 2013.

[19] Christian Kvalheim . The Little Mongo DB Schema Design Book. Leanpub, October 2015.

[20] Tae-Sun Chung and Hyoung-Joo Kim. Techniques for the evaluation of xml queries: a survey. Data & Knowledge Engineering, 46(2):225–246, 2003.

[21] Edgar F Codd. A relational model of data for large shared data banks. Communications of the ACM, 13(6):377–387, 1970.

[22] Latha S. Colby. A recursive algebra and query optimization for nested relations. In Proc. of the 1989 ACM SIGMOD International Conference on Management of Data, pages 273–283, New York, NY, USA, 1989. ACM.

[23] Latha S Colby. A recursive algebra for nested relations. Information Systems, 15(5):567–582, 1990.

[24] Douglas Crockford and Chip Morningstar. Standard ecma-404 the json data interchange syntax. Geneva: ECMA International, 2017.

[25] V Deshpande and Per-Åke Larson. An algebra for nested relations. University of Waterloo. Department of Computer Science, 1987.

[26] Dominik Durner, Viktor Leis, and Thomas Neumann. JSON Tiles: Fast Analytics on Semi-Structured Data. Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 445–458, 2021.

[27] Marius Eich, Pit Fender, and Guido Moerkotte. Efficient generation of query plans containing group-by, join, and groupjoin. The VLDB Journal, 27:617–641, 2018.

[28] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. ACM Transactions on Database Systems (TODS), 25(4):457–516, 2000.

[29] Mary Fernandez, Jérôme Siméon, Philip Wadler, S Cluet, A Deutsch, D Florescu, A Levy, D Maier, J McHugh, J Robie, et al. Xml query languages: Experiences and exemplars. Communication to the XML Query W3C Working Group, 256, 1999.

[30] Patrick C Fischer and Dirk Van Gucht. Determining when a structure is a nested relation. In VLDB, volume 85, pages 171–180, 1985.

[31] PC Fischer and SJ Thomas. Nested relational structures. The Theory of Databases, Advances in Computing Research III, pages 269–307, 1983.

[32] Avrilia Floratou, Nikhil Teletia, David J DeWitt, Jignesh M Patel, and Donghui Zhang. Can the elephants handle the nosql onslaught? Proceedings of the VLDB Endowment, 5(12):1712–1723, 2012.

[33] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the world-wide web: A survey. ACM Sigmod Record, 27(3):59–74, 1998.

[34] Cesar Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. ACM Transactions on Database Systems (TODS), 22(1):43–74, 1997.

[35] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical Report 1997-50, Stanford InfoLab, 1997.

[36] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. A cost model for random access queries in document stores. The VLDB Journal, pages 1–20, 2021.

[37] Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-logic: Logical foundations for json querying. In Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '17, page 137–149, New York, NY, USA, 5 2017. Association for Computing Machinery.

[38] Richard Hull and Masatoshi Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In VLDB, volume 90, pages 455–468, 1990.

[39] Thomas Hütter, Nikolaus Augsten, Christoph M Kirsch, Michael J Carey, and Chen Li. Jedi: These aren't the json documents you're looking for... In Proceedings of the 2022 International Conference on Management of Data, pages 1584–1597, 2022.

[40] Zachary G. Ives and Nicholas E. Taylor. Sideways information passing for push-style query processing. In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, ICDE '08, pages 774–783, Washington, DC, USA, 2008. IEEE Computer Society.

[41] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable structural index construction for json analytics. Proc. VLDB Endow., 14(4):694–707, dec 2020.

[42] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. Scalable processing of contemporary semi-structured data on commodity parallel processors - a compilation-based approach. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 79–92, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Stephen Kaisler, Frank Armour, J Alberto Espinosa, and William Money. Big data: Issues and challenges moving forward. In 2013 46th Hawaii international conference on system sciences, pages 995–1004. IEEE, 2013.

[44] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. Proc. VLDB Endow., 9(12):972–983, aug 2016.

[45] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. Proceedings of the VLDB Endowment, 9(12):972–983, 2016.

[46] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema extraction and structural outlier detection for JSON-based NoSQL Data Stores. Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI), 241:425–444, 2015.

[47] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. The VLDB Journal, 28(6):941–960, 2019.

[48] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In 2013 IEEE 29th International Conference on Data Engineering (ICDE), pages 38–49. IEEE, 2013.

[49] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast json parser for data analytics. Proc. VLDB Endow., 10(10):1118–1129, jun 2017.

[50] Caixue Lin. Object-oriented database systems: A survey. Online. Online, http, 2003.

[51] Adam Lith and Jakob Mattsson. Investigating storage solutions for large data-a comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data. 2010.

[52] Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Ying Liu, and Hui Joe Chang. Closing the functional and performance gap between sql and nosql. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, page 227–238, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Tomas Felipe Llano-Rios, Mohamed Khalefa, and Antonio Badia. Evaluation nosql systems for decision support: An experimental approach. In Proc. of the IEEE Conference on Big Data, pages 2802–2811, 2020.

[54] Tomas Felipe Llano-Rios, Mohamed Khalefa, and Antonio Badia. Experimental comparison of relational and nosql document systems: the case of decision support. In Proc. of the Twelfth TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC), 2020.

[55] Jason McHugh and Jennifer Widom. Query optimization for xml. In VLDB, volume 99, pages 315–326, 1999.

[56] Michael J Mior. Jsonoid: Monoid-based enrichment for configurable and scalable data-driven schema discovery. arXiv preprint arXiv:2307.03113, 2023.

[57] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. Dtd-miner: a tool for mining dtd from xml documents. In Proceedings Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems. WECWIS 2000, pages 144–151. IEEE, 2000.

[58] Keshav Murthy. Cost based optimizer for couchbase n1ql (sql for json). `https://blog.couchbase.com/cost-based-optimizer-for-couchbase-n1ql-sql-for-json/`, aug 2019.

[59] Keshav Murthy. Less is more: Why use a cost-based optimizer? `https://www.couchbase.com/blog/why-cost-based-optimizer-for-nosql-n1ql-couchbase/`, sept 2019.

[60] Svetlozar Nestorov, Jeffrey Ullman, Janet Wiener, and Sudarashan Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In Proceedings 13th International Conference on Data Engineering, pages 79–90. IEEE, 1997.

[61] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow., 4(9):539–550, June 2011.

[62] Thomas Neumann and Viktor Leis. Compiling database queries into machine code. IEEE Data Eng. Bull., 37(1):3–11, 2014.

[63] Matthias Nicola and Jasmi John. Xml parsing: a threat to database performance. In Proceedings of the twelfth international conference on Information and knowledge management, pages 175–178, 2003.

[64] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The sql++ query language: Configurable, unifying and semi-structured. arXiv preprint arXiv:1405.3631, 2014.

[65] Amy Ousterhout, Steve McCanne, Henri Dubois-Ferriere, Silvery Fu, Sylvia Ratnasamy, and Noah Treuhaft. Zed: Leveraging data types to process eclectic data. CIDR, 2023.

[66] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In Proceedings of the eleventh international conference on data engineering, pages 251–260. IEEE, 1995.

[67] Christina Pavlopoulou, Jr Preston E. Carman, Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. A parallel and scalable processor for JSON data. In Proceedings of the 21st International Conference on Extending Database Technology (EDBT), pages 576–587, 2018.

[68] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. In Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, pages 278–283, 2011.

[69] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. Database management systems, volume 3. McGraw-Hill New York, 2003.

[70] Olivier Rodriguez, Federico Ulliana, and Marie-Laure Mugnier. Scalable reasoning on document stores via instance-aware query rewriting. Proc. VLDB Endow., 16(11):2699–2713, jul 2023.

[71] Arnon Rosenthal et al. Outerjoin simplification and reordering for query optimization. ACM Transactions on Database Systems, 22(1):43–74, 1997.

[72] Nico Schäfer and Sebastian Michel. Joda: A vertically scalable, lightweight json processor for big data transformations. In 2020 IEEE 36th International Conference on Data Engineering (ICDE), pages 1726–1729, 2020.

[73] Standard Specification and Transaction Processing Performance Council TPC. TPC BENCHMARK TM H. Standard, San Francisco, CA, September 2017.

[74] William Spoth, Oliver Kennedy, Ying Lu, Beda Hammerschmidt, and Zhen Hua Liu. Reducing ambiguity in json schema discovery. In Proceedings of the 2021 International Conference on Management of Data, pages 1732–1744, 2021.

[75] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column oriented dbms. In Proc. of the VLDB Conference, pages 553–564, 2005.

[76] Juan Cruz Viotti and Mital Kinderkhedia. A benchmark of json-compatible binary serialization specifications. arXiv preprint arXiv:2201.03051, 2022.

[77] Tomas Vondra. Collecting statistics about contents of jsonb columns. https://www.postgresql.org/message-id/c9c4bd20-996c-100e-25e7-27e27bb1da7c@enterprisedb.com, 12 2021. Accessed on September 11, 2023.

[78] W Yan. Rewriting optimization of sql queries containing group-by. PhD thesis, PhD thesis, University of Waterloo, 1995.

CURRICULUM VITA

Tomas Felipe Llano-Rios

EDUCATION

M.S., Computer Science, 2019-2020

University of Louisville, 2301 S 3rd St, Louisville, KY, USA

B.S., Computer Science, 2014-2018

Universidad EAFIT, Carrera 49, Cl. 7 Sur #50, Medellín, Antioquia, Colombia

PROFESSIONAL POSITIONS

Enterprise Research Computer Analyst, 2023-Present, Full Time

University of Louisville, Louisville, USA

Graduate Assistant (HPC Linux Sysadmin), 2022-2023, Part Time

University of Louisville, Louisville, USA

Teaching Assistant, 2020-2022, Part Time

University of Louisville, Louisville, USA

Student Intern in HPC, 2017-2018, Full Time

ITAP Research Computing (RCAC) - Purdue University

West Lafayette, IN, USA

HPC Research Assistant, 2015-2017 and 2018, Part Time

Centro de Computación Científica APOLO - Universidad EAFIT

Medellín, Antioquia, Colombia

PUBLICATIONS

Tomas Felipe Llano-Rios, Mohamed Khalefa, and Antonio Badia. Experimental comparison of relational and nosql document systems: the case of decision support. In Proc. of the Twelfth TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC), 2020

Tomas Felipe Llano-Rios, Mohamed Khalefa, and Antonio Badia. Evaluation nosql systems for decision support: An experimental approach. In Proc. of the IEEE Conference on Big Data, pages 2802–2811, 2020