Physical database design in document stores

Moditha Lakshan Dharmasiri Hewasinghage

Physical Database Design in Document Stores

Ph.D. Dissertation
Moditha Lakshan Dharmasiri Hewasinghage

Abstract

NoSQL is an umbrella term used to classify alternate storage systems to the traditional Relational Database Management Systems (RDBMSs). At the moment of writing, there are more than 200 NoSQL systems available that can be classified into four main categories on the data storage model: key-value stores, document stores, column family stores, and graph stores. Document stores have gained popularity mainly due to the semi-structured data storage model and the rich query capabilities compared to the other NoSQL systems making them an ideal candidate for rapid prototyping. Document stores encourage users to use a data-first approach as opposed to a design-first one. Database design on document stores is mainly carried out in a trial-and-error or ad-hoc rule-based manner instead of a formal process such as normalization in an RDBMS. However, these approaches could easily lead to a non-optimal database design leading to additional costs in query processing, data storage, and redesigning.

This PhD thesis aims to provide a novel multi-criteria-based approach to database design in document stores. Most of the existing approaches of database design are based on optimizing query performance. However, other factors include storage requirement and complexity of the stored documents specific to each use case. Moreover, there is a large solution space of alternative designs due to the different combinations of referencing and nesting of data. Hence, we believe multi-criteria optimization is ideal with a proven track record of solving such problems in various domains. However, to achieve this, we need to address several issues that will enable us to apply multi-criteria optimization for the data design problem.

First, we evaluate the impact of alternate storage representations of semi-structured data. There are multiple and equivalent ways to physically represent semi-structured data, but there is a lack of evidence about the potential impact on space and query performance. Thus, we embark on the task of quantifying that precisely for document stores. We empirically compare multiple ways of representing semi-structured data, which allows us to derive a set of guidelines for efficient physical database design considering both JSON and relational options in the same palette.

Then, we need a formal canonical model that is capable of representing alternative designs. To this extent, we propose a hypergraph-based approach for representing heterogeneous datastore designs. Taking an existing common programming interface to NoSQL systems, we extend and formalize it as hypergraphs. Then, we define design constraints and query transformation rules for three representative data store types. Next, we propose a simple query rewriting algorithm from a generic one into underlying data stores specific one and provide a prototype implementation. Furthermore, we introduce a storage statistics estimator on the underlying data stores. Finally, we show the feasibility of our approach on a use case of an existing polyglot system and its usefulness in metadata and physical query path calculations.

Next, we require a formal query cost model to estimate and evaluate query performance on alternative document store designs. Document stores use primitive approaches to query processing, such as evaluating all possible query plans to find the winning one and using it in the subsequent similar queries or relying on the end-user to specify the usage of indexes instead of a formal cost model. However, we require a reliable approach to compare two alternative designs on how they perform on a specific query. For this, we define a generic storage and query cost model based on disk access and memory allocation that allows estimating the impact of design decisions. Since all document stores carry out data operations in memory, we first estimate the memory usage by considering the characteristics of the stored documents, their access patterns, and memory management algorithms. Then, using this estimation and metadata storage size, we introduce a cost model for random access queries. This is the first attempt at such an approach to the best of our knowledge. Finally, we validate our work on two well-known document store implementations: MongoDB and Couchbase. The results show that the memory usage estimates have an average precision of 91%, and predicted costs are highly correlated to the actual execution times. During this work, we have managed to suggest several improvements to document storage systems. Thus, this cost model also contributes to identifying discordance between document store implementations and their theoretical expectations.

Finally, we implement the automated database design solution using multi-criteria optimization. First, we introduce an algebra of transformations that can systematically modify a design of our canonical representation. Then, using these transformations, we implement a local search algorithm driven by a loss function that can propose near-optimal designs with high probability. Finally, we compare our prototype against an existing document store data design solution purely driven by query cost. Our proposed designs have better performance and are more compact with less redundancy.

Resum

NoSQL és un terme paraigua utilitzat per classificar sistemes d'emmagatzematge alternatius als sistemes tradicionals de gestió de bases de dades relacionals (RDBMS). En el moment d'escriure aquesta tesi, hi ha més de 200 sistemes NoSQL disponibles que es poden classificar en quatre categories principals segons el model d'emmagatzematge de dades: magatzems de clau-valor, magatzems de documents, magatzems de famílies de columnes i magatzems de grafs. Els magatzems de documents han guanyat popularitat principalment a causa del model d'emmagatzematge de dades semiestructurat i les riques capacitats de consulta en comparació amb els altres sistemes NoSQL, que els converteixen en un candidat ideal per al prototipat ràpid. Els magatzems de documents animen els usuaris a utilitzar un enfocament de "dades primer" en lloc d'un enfocament de "disseny primer". El disseny de bases de dades en magatzems de documents es porta a terme principalment d'una manera d'assaig i error o basat en regles ad-hoc en lloc d'un procés formal com ara la normalització en un RDBMS. Tanmateix, aquests enfocaments podrien conduir fàcilment a un disseny de base de dades no òptim que comportarà costos addicionals en el processament de consultes, l'emmagatzematge de dades i el redisseny.

Aquesta tesi doctoral pretén proporcionar un nou enfocament basat en diversos criteris per al disseny de bases de dades en magatzems de documents. La majoria dels enfocaments existents de disseny de bases de dades es basen en l'optimització del rendiment de les consultes. Tanmateix, altres factors inclouen l'espai requerit i la complexitat dels documents emmagatzemats específics per a cada cas d'ús. A més, hi ha un gran espai de solucions de disseny alternatives a causa de les diferents combinacions d'apuntadors i nidificació de dades. Per tant, creiem que l'optimització multicriteri és ideal amb un historial provat de resolució d'aquest tipus de problemes en diversos dominis. Tanmateix, per aconseguir-ho, hem d'abordar diversos problemes parcials que ens permetran aplicar l'optimització multicriteri per al problema del disseny de dades.

En primer lloc, cal estudiar l'impacte de les representacions d'emmagatzematge alternatives per a dades semiestructurades. Hi ha maneres múltiples i equiva-

lents de representar físicament dades semiestructurades, però hi ha una manca d'evidència sobre l'impacte potencial en l'espai requerit i el rendiment de les consultes. Així, ens embarquem en la tasca de quantificar-ho precisament per als magatzems de documents. Comparem empíricament múltiples maneres de representar dades semiestructurades, la qual cosa ens permet derivar un conjunt de directrius per a un disseny eficient de bases de dades físiques tenint en compte tant les opcions JSON com les relacionals alhora.

Després, necessitem un model canònic formal que sigui capaç de representar dissenys alternatius. Per aquesta tasca, proposem un enfocament basat en hipergrafs per representar dissenys heterogenis d'emmagatzemament de dades. Prenent una interfície de programació comuna existent als sistemes NoSQL, l'ampliem i la formalitzem com a hipergrafs. A continuació, definim restriccions de disseny i regles de transformació de consultes per a tres tipus de magatzem de dades representatius. A continuació, proposem un algorisme generic de reescriptura de consultes senzilles per a un magatzems de dades específic i proporcionem un prototipus. A més, introduïm un estimador d'estadístiques d'emmagatzematge sobre els magatzems de dades subjacents. Finalment, mostrem la viabilitat del nostre enfocament en un cas d'ús d'un sistema políglot existent i la seva utilitat en els càlculs de metadades i camins de consulta física.

A continuació, necessitem un model de costos de consulta formal per estimar i avaluar el rendiment de la mateixa consulta en dissenys alternatius de magatzem de documents. Els magatzems de documents utilitzen enfocaments primitius per al processament de consultes, com ara avaluar tots els plans de consulta possibles per trobar el guanyador i utilitzar-lo en consultes similars posteriors o confiar en l'usuari final per especificar l'ús d'índexs en lloc d'un model de costos formal. Tanmateix, necessitem un enfocament fiable per comparar com funciona una consulta específica en dos dissenys alternatius. Per a això, definim un model genèric de costos d'emmagatzematge i consulta basat en l'accés al disc i l'assignació de memòria que permet estimar l'impacte de les decisions de disseny. Com que tots els magatzems de documents duen a terme operacions de dades a memòria, primer estimem l'ús de la memòria tenint en compte les característiques dels documents emmagatzemats, els seus patrons d'accés i els algorismes de gestió de la memòria. A continuació, utilitzant aquesta estimació i la mida d'emmagatzematge de metadades, introduïm un model de costos per a consultes d'accés aleatori. Fins on sabem, aquest és el primer intent d'aquest enfocament. Finalment, validem el nostre treball en dues implementacions de magatzem de documents conegudes: MongoDB i Couchbase. Els resultats mostren que les estimacions d'ús de memòria tenen una precisió promig del 91% i els costos previstos estan altament correlacionats amb els temps d'execució reals. Durant aquest treball, hem aconseguit suggerir diverses millores als sistemes d'emmagatzematge de documents utilitzats als experiments. Així, aquest model de costos també

contribueix a identificar discordances entre les implementacions del magatzem de documents i les seves expectatives teòriques.

Finalment, implementem la solució de disseny automatitzat de bases de dades mitjançant optimització multicriteri. En primer lloc, introduïm una àlgebra de transformacions que pot modificar sistemàticament un disseny en la nostra representació canònica. A continuació, utilitzant aquestes transformacions, implementem un algorisme de cerca local impulsat per una funció de pèrdua que pot proposar dissenys gairebé òptims amb alta probabilitat. Finalment, comparem el nostre prototipus amb una solució existent de disseny de dades de magatzem de documents només impulsada pel cost de la consulta. Els nostres dissenys proposats tenen un millor rendiment i són més compactes, amb menys redundància.

Résumé

NoSQL est un terme générique utilisé pour classer les systèmes de stockage alternatifs aux systèmes de gestion de bases de données relationnelles (SGBDR) traditionnels. Au moment de la rédaction de cet article, il existe plus de 200 systèmes NoSQL disponibles qui peuvent être classés en quatre catégories principales sur le modèle de stockage de données : magasins de valeurs-clés, magasins de documents, magasins de familles de colonnes et magasins de graphiques. Les magasins de documents ont gagné en popularité principalement en raison du modèle de stockage de données semi-structuré et des capacités de requêtes riches par rapport aux autres systèmes NoSQL, ce qui en fait un candidat idéal pour le prototypage rapide. Les magasins de documents encouragent les utilisateurs à utiliser une approche axée sur les données plutôt que sur la conception. La conception de bases de données sur les magasins de documents est principalement effectuée par essais et erreurs ou selon des règles ad hoc plutôt que par un processus formel tel que la normalisation dans un SGBDR. Cependant, ces approches pourraient facilement conduire à une conception de base de données non optimale entraînant des coûts supplémentaires de traitement des requêtes, de stockage des données et de refonte.

Cette thèse de doctorat vise à fournir une nouvelle approche multicritères de la conception de bases de données dans les magasins de documents. La plupart des approches existantes de conception de bases de données sont basées sur l'optimisation des performances des requêtes. Cependant, d'autres facteurs incluent les exigences de stockage et la complexité des documents stockés spécifique à chaque cas d'utilisation. De plus, il existe un grand espace de solution de conceptions alternatives en raison des différentes combinaisons de référencement et d'imbrication des données. Par conséquent, nous pensons que l'optimisation multicritères est idéale par l'intermédiaire d'une expérience éprouvée dans la résolution de tels problèmes dans divers domaines. Cependant, pour y parvenir, nous devons résoudre plusieurs problèmes qui nous permettront d'appliquer une optimisation multicritère pour le problème de conception de données.

Premièrement, nous évaluons l'impact des représentations alternatives de

stockage des données semi-structurées. Il existe plusieurs manières équivalentes de représenter physiquement des données semi-structurées, mais il y a un manque de preuves concernant l'impact potentiel sur l'espace et sur les performances des requêtes. Ainsi, nous nous lançons dans la tâche de quantifier cela précisément pour les magasins de documents. Nous comparons empiriquement plusieurs façons de représenter des données semi-structurées, ce qui nous permet de dériver un ensemble de directives pour une conception de base de données physique efficace en tenant compte à la fois des options JSON et relationnelles dans la même palette.

Ensuite, nous avons besoin d'un modèle canonique formel capable de représenter des conceptions alternatives. Dans cette mesure, nous proposons une approche basée sur des hypergraphes pour représenter des conceptions de magasins de données hétérogènes. Prenant une interface de programmation commune existante aux systèmes NoSQL, nous l'étendons et la formalisons sous forme d'hypergraphes. Ensuite, nous définissons les contraintes de conception et les règles de transformation des requêtes pour trois types de magasins de données représentatifs. Ensuite, nous proposons un algorithme de réécriture de requête simple à partir d'un algorithme générique dans un magasin de données sous-jacent spécifique et fournissons une implémentation prototype. De plus, nous introduisons un estimateur de statistiques de stockage sur les magasins de données sous-jacents. Enfin, nous montrons la faisabilité de notre approche sur un cas d'utilisation d'un système polyglotte existant ainsi que son utilité dans les calculs de métadonnées et de chemins de requêtes physiques.

Ensuite, nous avons besoin d'un modèle de coûts de requêtes formel pour estimer et évaluer les performances des requêtes sur des conceptions alternatives de magasin de documents. Les magasins de documents utilisent des approches primitives du traitement des requêtes, telles que l'évaluation de tous les plans de requête possibles pour trouver le plan gagnant et son utilisation dans les requêtes similaires ultérieures, ou l'appui sur l'usager final pour spécifier l'utilisation des index au lieu d'un modèle de coûts formel. Cependant, nous avons besoin d'une approche fiable pour comparer deux conceptions alternatives sur la façon dont elles fonctionnent sur une requête spécifique. Pour cela, nous définissons un modèle de coûts de stockage et de requête générique basé sur l'accès au disque et l'allocation de mémoire qui permet d'estimer l'impact des décisions de conception. Étant donné que tous les magasins de documents effectuent des opérations sur les données en mémoire, nous estimons d'abord l'utilisation de la mémoire en considérant les caractéristiques des documents stockés, leurs modèles d'accès et les algorithmes de gestion de la mémoire. Ensuite, en utilisant cette estimation et la taille de stockage des métadonnées, nous introduisons un modèle de coûts pour les requêtes à accès aléatoire. Il s'agit de la première tentative d'une telle approche au meilleur de notre connaissance. Enfin, nous validons notre travail

sur deux implémentations de magasin de documents bien connues : MongoDB et Couchbase. Les résultats démontrent que les estimations d'utilisation de la mémoire ont une précision moyenne de 91% et que les coûts prévus sont fortement corrélés aux temps d'exécution réels. Au cours de ce travail, nous avons réussi à proposer plusieurs améliorations aux systèmes de stockage de documents. Ainsi, ce modèle de coûts contribue également à identifier les discordances entre les implémentations de stockage de documents et leurs attentes théoriques.

Enfin, nous implémentons la solution de conception automatisée de bases de données en utilisant l'optimisation multicritères. Tout d'abord, nous introduisons une algèbre de transformations qui peut systématiquement modifier une conception de notre représentation canonique. Ensuite, en utilisant ces transformations, nous implémentons un algorithme de recherche locale piloté par une fonction de perte qui peut proposer des conceptions quasi optimales avec une probabilité élevée. Enfin, nous comparons notre prototype à une solution de conception de données de magasin de documents existante uniquement basée sur le coût des requêtes. Nos conceptions proposées ont de meilleures performances et sont plus compactes avec moins de redondance.

Acknowledgements

I would like to take this opportunity to thank the many people who, directly or indirectly, have contributed to the realization of this thesis.

First and foremost, I am grateful to my home advisors Alberto Abelló and Jovan Varga for their endless patience, enthusiasm and guidance. This dissertation would have not been possible without them. Their door has been always open for long, technical and stimulating discussions. Second, I am indebted to my host advisor Esteban Zimányi. Besides making my visits at ULB very enjoyable, he has taught me that striving for formal elegance pays off. I look up to the three of them, both professionally and personally, and consider myself very lucky to have been their student.

I thank the members of the DTIM research group at UPC for creating an engaging working environment. Special thanks goes to Sergi Nadal for the fruitful collaborations on Chapters 2, 5, and the corresponding demo in Appendix B. I am grateful for your friendly advises, support, and technical discussions. I also would like to thank the members of the CoDE-WIT research group at ULB who made my time in Brussels memorable.

I am grateful to my family and many good friends for their unconditional and continuing support. I would like to express my gratitude to my lovely friends Larissa, Gledis, Phil, Ward, Olga, Anas, Katya, Jorge, Rediana, Maximiliano, Luciana, Eugenia, Jasmine, Juan, Viktor, Mariana, Ziyad, and Thao for their kindness, love and support through thick and thin.

There are three people who deserve a special mention, my father Sarath who has been a pillar of support throughout my journey for his unconditional love and understanding. Suela, for being there since we first met in Brussels during our masters for being the best partner to work with, for your kindness, advises and giving me strength to achieve my goals. Vipula, my best friend since grade 8, for always being there for me as a brother and making my visits to Sri Lanka always unforgettable.

Barcelona, March 2022

Contents

Contents

Contents

List of Figures

List of Tables

Thesis Details

Thesis Title: Physical Database Design in Document Stores
Ph.D. Student: Moditha Lakshan Dharmasiri Hewasinghage
Supervisors: Prof. Alberto Abelló Gamazo, Universitat Politècnica de Catalunya, BarcelonaTech, Spain (UPC supervisor)
Dr. Jovan Varga, Microsoft, Costa Rica (UPC co-supervisor)

Prof. Esteban Zimányi, Université Libre de Bruxelles, Brussels, Belgium (ULB supervisor)

The main body of this thesis consists of the following papers:

[1] On the Performance Impact of Using JSON, Beyond Impedance Mismatch. Moditha Hewasinghage, Sergi Nadal, Alberto Abelló. European Conference on Advances in Databases and Information Systems (ADBIS). 2020.

[2] Managing Polyglot Systems Metadata with Hypergraphs. Moditha Hewasinghage, Alberto Abelló, Jovan Varga, Esteban Zimányi. Data & Knowledge Engineering, 134, 101896. 2021.

[3] A Cost Model for Random Access Queries in Document Stores. Moditha Hewasinghage, Alberto Abelló, Jovan Varga, Esteban Zimányi. The VLDB Journal 30, 559–578. 2021.

[4] Automated Database Design for Document Stores with Multi-criteria Optimization. Moditha Hewasinghage, Sergi Nadal, Alberto Abelló, , Esteban Zimányi. Knowledge and Information Systems (KAIS) (submitted).

In addition to the main papers, the following peer-reviewed publications have also been made.

Conference articles.

[5] Managing Polyglot Systems Metadata with Hypergraphs. Moditha Hewasinghage, Jovan Varga, Alberto Abelló, Esteban Zimányi. International Conference on Conceptual Modeling (ER). 2018. (extended in [2])

Tool demonstrations.

[6] DocDesign: Cost-based Database Design for Document Stores. Moditha Hewasinghage, Alberto Abelló, Jovan Varga, Esteban Zimányi. International Conference on Scientific and Statistical Database Management (SSDBM). 2020

[7] DocDesign 2.0: Automated Database Design for Document Stores with Multi-criteria Optimization. Moditha Hewasinghage, Sergi Nadal, Alberto Abelló. International Conference on Extending Database Technology (EDBT). 2021

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the submitted or published scientific papers which are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis.

Chapter 1

Introduction

1 Background and Motivation

Traditional Relational Database Management Systems (RDBMSs) have been the go-to solution for data storage for the last couple of decades. RDBMSs have been around for about 50 years maturing and gaining popularity in storing tabular data. However, with the big data era, NoSQL systems were introduced as alternate storage solutions, giving rise to novel data storage paradigms [24, 74]. The difference between tabular data model of RDBMS to the object-oriented programming model is known as the impedance mismatch. This can affect the performance of programs especially in the case of large databases. Thus, NoSQL systems tend to minimize this performance impact by having object-oriented or similar data models. More than 200 NoSQL systems are currently available, catering to specific niches of modern data storage.[1] These NoSQL systems fall under four main categories: key-value stores, document stores, wide-column stores, and graph stores. Among these systems, document stores provide extended features such as complex query capabilities due to the flexible, semi-structured data model. JSON storage is now widely used in data analytics due to the fast serialization and ease of interchange between applications. This semi-structured data model of document stores allows them to handle the problem of data variety efficiently but introduces new challenges in database design.

Based on the relational algebra, normalization theory guides RDBMS database design [77, 94]. The database design obtained by reaching 3NF or BCNF can effectively and efficiently answer most typical queries while removing data redundancy. On the contrary, similar to most NoSQL systems, document stores encourage de-normalization and embedding of related data

---

[1]http://nosql-database.org

to avoid joins. Consequently, data redundancy is accepted or at times even encouraged in document store data designs. Although the flexibility allows faster deployment, poor design decisions can lead to incorrect results or negative performance impact. It has been shown that data design requires significant modeling decisions that impact important quality requirements, including scalability and performance [10, 12, 22]. Therefore, optimal data design is an essential part of any non-trivial data-driven application.

Document stores allow having fully redundant collections, each of which can answer specific queries efficiently. However, apart from query cost, other factors are affected by the database design, such as storage size and complexity of the stored documents. Thus, a redundant database design requiring considerably higher storage space is not ideal for most use cases. Hence, document store data design generally resorts to trial-and-error or ad-hoc rule-based approaches. For instance, the leading document store, MongoDB, presents common patterns for database design identified through typical use cases.[2] Although such an approach is capable of coming up with relatively optimal designs, the optimality of the design is highly dependant on the experience of the data designer. Thus, it is a common problem with document store designs that performance issues arise as the storage grows with the life time of a system. Addressing such issues in a production system by restructuring the storage can become costly, especially with massive data sets in the big data era. Therefore, a common practice is to scale out the document store by adding more nodes or adding more powerful hardware (memory, CPU, SSD storage), which can incur additional costs only to provide a temporary solution.

Little research has been conducted on modeling and systematical designing the data in document stores and NoSQL systems, in general. due to the novelty and the complexity of the problem. As mentioned before, NoSQL database design is limited to guidelines and best practices provided by the tools themselves, and most of the research is purely based on optimizing the query performance ignoring other factors that are affected by the design [58, 88, 95] such as storage space and complexity of the design. Thus, the objective of this thesis is to automate the database design for document stores given a use case consist of an entity-relationship design, a query workload, and user preference on multiple factors (e.g., query performance, storage space, the complexity of the stored documents) that are impacted by the data design. Moving away from ad-hoc rule-based approaches into a methodical one requires us to answer the following questions.

- What are the key data design concepts that could determine the optimal storage design for a given dataset in document stores?

---

[2]https://www.mongodb.com/blog/post/building-with-patterns-a-summary

- How do design decisions affect end-user requirements such as performance, storage space, and complexity of the stored documents, given a workload?

The answers to these questions are not trivial and involve choosing the optimal design out of a multitude of possible designs, each having its strengths and weaknesses. Therefore, a solution for the design decision problem can be modeled as a multi-criteria optimization problem. The available data designs become the search space, and the affected parameters become the objective functions that need to be optimized.

Multiple parameters are affected by the data design, including storage space, query execution time, depth of the stored documents, and the degree of heterogeneity within a collection. These parameters create contradicting cost functions that drive the optimization problem. For example, data duplication increases the storage size and the writing cost but reduces the query execution time. Therefore, it is also essential to analyze how each of these cost functions behaves and model them in the overall solution.

This work could greatly benefit different stages of a data-driven application from the development phase to maintenance. Moreover, a clear understanding of the data design will improve data quality aspects. Improving the query performance through the design itself would significantly reduce the data movement and maintenance concerning the changes in the data design. A formalized systematic approach to data design will give end-users, such as application developers and database administrators, the much-needed transparency and control over the data design rather than trial-and-error ad-hoc approaches.

2 The NoSQL Systems and Document Stores

Traditional RDBMSs have been a universal solution for data storage until the dawn of the BigData era, where specialized data storage solutions arose. Indeed, a single kind of storage system could not handle all the volume, velocity and variety of BigData [33]. Thus, NoSQL systems were born, an umbrella term used for alternative data stores from the traditional RDBMSs. These alternate storage systems can be divided into four main categories, as shown in Fig. 1.1.

1. **Key-value stores** are the simplest form of data storage among the NoSQL systems. The storage is similar to that of a hash table, where each data item is stored as a blob (schemaless) under a unique key. This allows fast lookups for data and can be easily scaled out. However, updating part of data and complex queries (apart from a simple get by the key) become inefficient. They require replacing the entire data

Key-value	Document	Wide-column	Graph
Key — Value Key — Value Key — Value Key — Value			

1 — Alice
2 — Bob
3 — Charlie
4 — Dan
	{
	"user":{
	  "id": 1
	  "name": "Alice"
	  }
	}
		1 Fruit A Foo B Bar
		2 City E DC D PLA D FLD
		3 State A NZ C CL
			Alice
			Bob

redis, apache Ignite riak	MongoDB., ArangoDB Couchbase, RAVENDB	cassandra, HBASE SCYLLA	neo4j AllegroGraph

Fig. 1.1: NoSQL data store types

and retrieving the stored data and processing them apriori by the client application respectively.

2. **Document stores** augment the key-value stores by allowing the stored data to be queried and updated utilizing nested values associated with each key. Their stored data are called documents and use semi-structured formats such as XML and JSON (widely used now). This enables efficient query capabilities to document stores through secondary indexes. Moreover, the document formats themselves provide efficient storage and compression mechanisms.

3. **Wide-column stores** use tables. These can be identified as two-dimensional key-value stores (for the row and the column). A column family consists of multiple arranged columns, and a piece of data can be identified by a key and a set of fixed column families. This allows to store and process substantial amounts of distributed data.

4. **Graph stores** use a flexible graph model from the traditional rigid structure of rows and columns, as the name suggests. This specialized storage structure enables storing inherently graph-like data such as social networks. In addition, they focus on data inter-connectivity allowing efficient graph processing algorithms compared to the traditional storage methods.

These NoSQL systems are created to cater specific storage needs of the end-user. Thus, it is essential to understand which data store is best suited for a particular task, and several works have been conducted on using NoSQL systems in different use-cases such as decision support systems [80, 81].

Nevertheless, document stores have gained popularity among the others due to several reasons:

- Semi-structured data storage allow users to maintain some structuring of data.

- Object-oriented JSON storage structure reduces the need of additional transformations reducing impedence mismatch.

- Complex query capabilities allows processing data within the data store instead of the client application.

- Maintaining a schema gives the end users a clear understanding of the stored data. This has encouraged mainstream RDBMS such as Oracle [79] and PostgreSQL[3] to incorporate document store-like capabilities with JSON storage.

- Data-first approach instead of design-first approach in RDBMS reduces the time to get started with a document store, making it a great tool for rapid prototyping [36].

3 Data Design for Document Stores

The semi-structured nature of document stores allows them to have database designs beyond traditional normalization theories. This makes the database design decisions more complicated with a myriad of possibilities, especially with tolerated data redundancy and encouraged nesting. Thus, the database design process for them has resorted to trial-and-error or ad-hoc rule-based approaches.[4] However, such approaches consider local optimizations and miss out on the bigger picture of global optimizations. Having a good database design is essential for any data storage system's performance, and bad design decisions cannot always be compensated by adding more powerful hardware. Let us take a simple example of two entities (*authors* and *books*) with a many-to-many relationship. If a typical RDBMS is used to store this information, we end up with three tables in 3NF, one per entity and another junction table as shown in Fig. 1.2. However, in a document store, there are multiple possibilities to store the same information depending on the choice of direction of the relationship, embedding or referring the relationship, and flattening or nesting the internal document or reference.

Entities are always stored in one or more collections. Thus, we can identify alternate design decisions as to how a relationship is stored in a document

---

[3]https://www.postgresql.org/docs/9.2/release-9-2.html
[4]https://www.mongodb.com/blog/post/building-with-patterns-a-summary
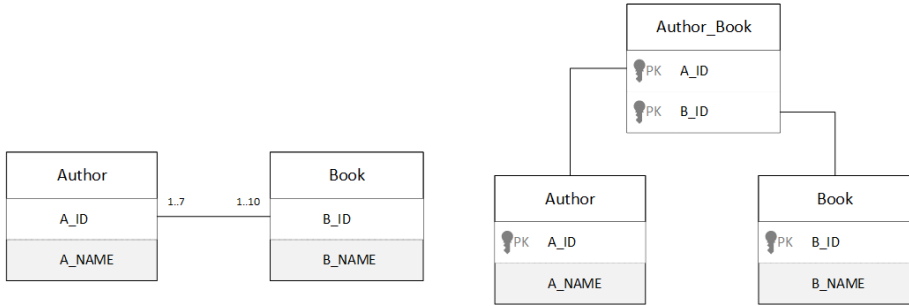
Fig. 1.2: Storing a many-to-many relationship in an RDBMS

store to ensure that all the entities are available. This storage is based on three choices as shown in Fig. 1.3.

- **Direction** determines which of the two participating entities is maintaining the information about the relationship. This can be either one of the entities or both.

- **Representation** ensures how the relationship information is maintained in the chosen entity either by referencing the identity of the other entity or by storing the whole entity as an embedded document.

- **Structure** governs how the relationship is structured either as a nested list or a flattened one. For example, if we decide to to store *books* inside an *author* it can be stored as a list ($books : [\{B\_ID : 1, ...\}, \{B\_ID : 2, ...\}, ...]$) or in a flattened manner ($book\_1 : ..., book\_2 : ..., ...$)
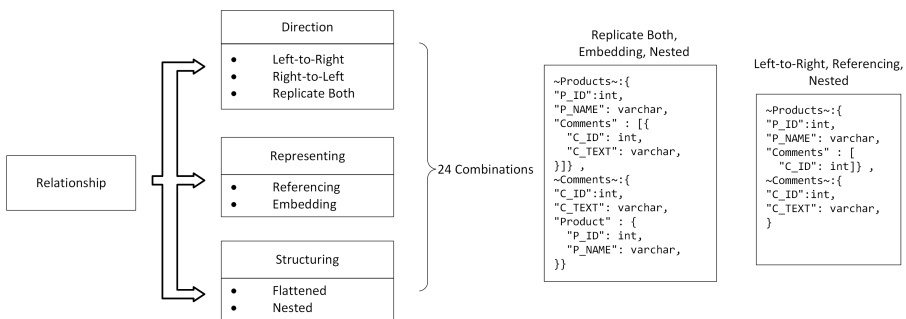
Fig. 1.3: Alternatives to storing a relationship

By enumerating all the design choices as shown in Table 1.1, we can identify that there are 24 possible design alternatives for a single relationship in a document store. Furthermore, several factors are affected by the design, such

as query performance, storage size, and complexity of the stored documents. Thus, each design has its own advantages and disadvantages, making them a potential optimal one depending on different end-user requirements.

Table 1.1: Enumeration of design choices for a relationship in a document store

Direction	Entity 1		Entity 2	
	Representation	Structure	Representation	Structure
left-to-right	reference	list		
	reference	flat		
	embed	list		
	embed	flat		
right-to-left			reference	list
			reference	flat
			embed	list
			embed	flat
both	reference	list	reference	list
	reference	list	reference	flat
	reference	list	embed	list
	reference	list	embed	flat
	reference	flat	reference	list
	reference	flat	reference	flat
	reference	flat	embed	list
	reference	flat	embed	flat
	embed	list	reference	list
	embed	list	reference	flat
	embed	list	embed	list
	embed	list	embed	flat
	embed	flat	reference	list
	embed	flat	reference	flat
	embed	flat	embed	list
	embed	flat	embed	flat

It is evident that the number of alternative designs grows exponentially with the number of relationships (24^r). Thus, it is impossible to iterate through all of them to find the optimal one, and the current design approaches resort to trial-and-error or ad-hoc rule-based processes. Our goal is to leave this beaten path and introduce a novel systematical approach for database design for document stores.

The workflow of our approach is depicted in Figure 1.4. Document stores (MongoDB is used as an example) do not maintain any schema, but extracting one has been already carried out in previous work [43, 116]. Therefore, we assume that the schema is available together with the data. Then, this schema

is represented in a canonical model which allows the representation of the data design and the query operations over it. Next, a set of transformations specified in the canonical model generates different data designs. Then, by using the query load, the objective functions, and the cost model, a multicriteria optimization approach is applied to identify the optimal design or a set of Pareto-optimal designs for the given use case. Finally, the chosen design can be implemented in the document store as collections.
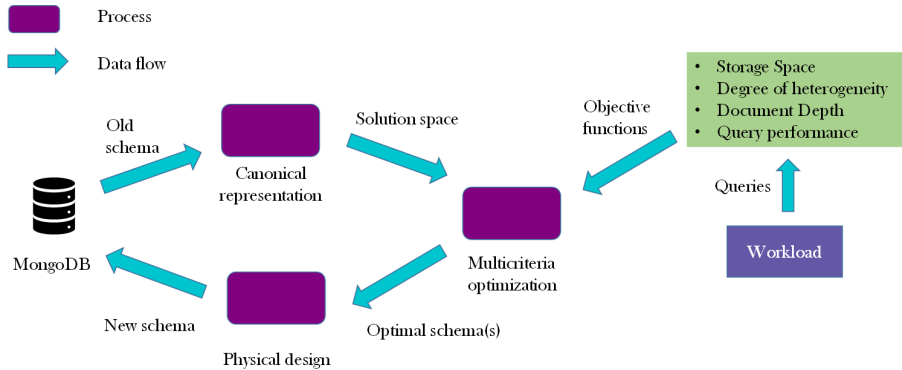
Fig. 1.4: Workflow of the approach

3.1 State of the Art and Challenges

In this subsection, we review the state of the art related to the automated database design depicted in Figure 1.4. Here, we also highlight related open research problems that drive the proposed contributions.

3.1.1 Canonical Model

Little research has been conducted on modeling and systematical designing the data in NoSQL data stores. Moreover, the schema management of document stores is handled differently from the typical RDBMS [42] due to the flexible schema. As a result, NoSQL design is limited to guidelines and best practices provided by the tools themselves. The semi-structured storage makes the schema management of document stores different from the typical RDBMS [42]. In [12], the authors introduce a high-level description of an interface for NoSQL systems based on a generic model. The main objective of this work is to formalize a common programming interface to NoSQL by hiding the specific implementations of the underlying data stores. They use a meta-modeling approach where a basic, standard structure is defined with the methods to access the system. The main component of the general

model are attributes, structs, and sets. An attribute is a simple element (a simple key-value pair or a single qualifier); a struct is a collection of attributes (column families, document); finally, a collection of elements is modeled by a set.

NoAm is a NoSQL abstract model designed to support scalability, performance, and consistency [10, 22]. The main aspect of this model is to organize the application data in aggregates. It defines the four main activities in developing a NoSQL conceptual schema which includes identifying the entities and the relationships between them. Then, the aggregates are designed upon those elements by grouping the related entities. Next, the aggregate storage in the target systems is done depending on the data access patterns and the scalability and consistency needs. NoAM also defines a set of different mapping strategies from the general data model to the physical layer of various NoSQL data stores.

A general approach for designing a NoSQL system for analytical workloads has been discussed in [58]. It adapts the traditional 3-phase design methodology of conceptual, logical, and physical design and integrates the relational and the co-relational models into a single quantitative method. First, at the conceptual level, the traditional ER diagram is used. Then, it is transformed into an directed graph where each of the entities is denoted by nodes and their relationships by the edges, tagged with the relationship type (specialization, composition, and association). Later, the rest of the components of the model are identified as nested entities. These steps group the entities into independent domain concepts and produce a hypergraph where the hypernode is a subgraph of the original node and the set of hyperedges among them. Finally, in cases where an entity can become a part of several heterogeneous hypernodes, the entity is replicated in each of them, and the hypernodes are identified as nested, heterogeneous, or homogeneous. A unified metamodel for NoSQL and relational databases including the notion of structural variability is introduced in the U-Schema unified metamodel [23]. This metamodel is capable of representing logical schemas for both relational and NoSQL systems. The authors introduce formal mapping between the U-schema and the data model for each data store, implement and validate these mappings through example applications.

The Concept and Object Modeling Notation (COMN) introduced covers the full spectrum of not only the data store but the software design process [59]. COMN is a graphical notation capable of representing the conceptual, logical, physical, and real-world design of an object. This helps to model the data in NoSQL systems where the traditional Entity-Relationship (ER) diagrams fail to represent specific concepts such as nesting. Moreover, the ability to represent the logical and the physical design on the same diagram enables a broader overall view of the underlying system.

Although there are several models that can represent NoSQL systems,

there is no single model that suits us to use in our automated database design approach. Most of them are oversimplified to support different NoSQL systems or over specified for a particular data store or usecase. We require a **formal** model that is capable of representing **alternative NoSQL systems** as well as **alternative designs** together with **storage metadata** and **query access plan** representation.

Thus, the first problem of interest in this thesis concerns providing a canonical model that is capable of representing alternate database designs. The model should be formal one with the ability to maintain metadata on the design as well as the transformation of one design to another in a systematic manner. Moreover, this canonical model should be extendable to represent any NoSQL and RDBMS and their designs.

3.1.2 Cost Model

Cost models or cost-based estimations for RDBMS systems are widely observed and utilized in the implementations of RDBMSs [47, 61, 75, 77]. Most of these cost models use the disk I/O as the key influential factor to the cost. [69] explores the possibility of a cost model for XML stores based on the tree structure of XML and the traversal of the tree from the root element, using the number of pages that need to be accessed in order to retrieve a particular element. Although these cost models are based on hard disk drives (HDDs), most of these systems are easily adapted for Solid State Drives (SSDs). Some specific work on SSD based optimization is also carried out [90].

In-memory data storage systems need to adopt new strategies to estimate costs and optimize performance due to the fact that the storage paradigm is not disk-based. [83] utilizes CPU usage as the key cost component for in-memory systems. Some other work also explore different concerns and possibilities of cost models for memory-based storage systems [68, 78].

Most of the NoSQL systems encourage heavy usage of memory to achieve faster response times and throughput but also utilize the non-volatile memory or the disk to store the data permanently and swap the data in and out that does not entirely fit in memory. This makes estimating the cost of querying and retrieval more complex. For example, MongoDB does not use a query estimation but rather execute all the possible query plans in parallel and picks a winner among those to answer a particular query[5]. This winning plan is then cached to be used on similar queries that follow. Unfortunately, there are no generic cost models available on document stores to compare the effectiveness of a particular design.

Hence, the second problem that we address in this thesis is the introduction of a cost model that can efficiently predict the query performance in a document store under a given query workload.

---

[5]https://docs.mongodb.com/manual/core/query-plans

3.1.3 Automated Database Design for Document Stores

Various performance comparisons have been carried out comparing document stores to other NoSQL data models in [6, 54] and RDBMS in [5, 7, 18]. They conclude that depending on the use case and the data model, different data stores have their own strengths and weaknesses. The applicability of document stores on different domains, such as spatial data, is discussed in [37, 117] and tree-like structures in [118]. The improvements in the physical data storage on document stores regarding the Solid State Drives compared to Hard Disk Drives are presented in [90].

When it comes to document stores, one should not forget about XML [19] which is a textual data format that is both human and machine-readable that enables simplicity, generality, and usability across the Internet. Xpath [28] and XQuery [17] are the two main standards for querying XML, and most of the research has been carried out on improving the query performance in the various, aforementioned storage methods utilizing query processing algorithms [26, 49], dynamic XML creation [2] and storage enhancements [41, 52]. The native XML data stores consider the query mechanism and use their own indexing and efficient storage methods.

Benchmarking on databases enables to evaluate the performance of different systems under different workloads. XMark [101] is a benchmarking framework that enables the assessment of different XML databases in performing a broad range of different queries that are typically encountered in the real world. YCSB [31] is an extension of TPC-H that is specifically designed for benchmarking NoSQL systems. This enables the end-users to fine-tune the read, write, and update loads on NoSQL data stores to compare relative performance. The work carried out in [109] introduces a benchmark of document stores of JSON as XML implementations.

Several work have been done on database design for NoSQL systems. NoSE [88] is a generic tool for generating a schema for a given use case on column family stores. This uses a cost-based approach to present a database schema from a conceptual schema. However, document store can have complex schema with multiple nesting levels compared to column family stores and the data design guidelines themselves differ from a column family store to a document store making NoSE limited to column family stores. Mortadello [34], is a framework devised for the automatic design of NoSQL databases. However, it only has a preliminary implementation for the document stores while column family stores are fully supported. Other approaches for database design for document stores [35, 95] only optimize the design on query performance.

The third and last problem that this thesis addresses is that of automatically proposing the optimal design in a document store for a given workload driven by user preferences of query performance, storage requirement, and complexity of stored

documents.

4 Structure of the Thesis

The results of this PhD thesis are reported in the four main chapters of the document (i.e., Chapters 2 to 5). Each chapter is self-contained, corresponding to an individual or a collection of research papers. Thus, they can be read in isolation as each chapter adheres to the same structure providing related work for the topic and concluding remarks. However, there might exist overlapping in concepts and examples given they were formulated in similar settings. Additionally, Appendices A and B refers to a published tool demonstration of our approach to automated data design for document stores.

The papers included in this thesis are listed below. Chapter 2 is based on Paper 1; Chapter 3 is based on Papers 2 and 5; Chapter 4 is based on Paper 3; Chapter 5 is based on current draft of Paper 4, and Appendix A and B are based on Papers 6 and 7 respectively.

1. On the Performance Impact of Using JSON, Beyond Impedance Mismatch. Moditha Hewasinghage, Sergi Nadal, Alberto Abelló. European Conference on Advances in Databases and Information Systems (ADBIS). 2020.

2. Managing Polyglot Systems Metadata with Hypergraphs. Moditha Hewasinghage, Alberto Abelló, Jovan Varga, Esteban Zimányi. Data & Knowledge Engineering, 134, 101896. 2021.

3. A Cost Model for Random Access Queries in Document Stores. Moditha Hewasinghage, Alberto Abelló, Jovan Varga, Esteban Zimányi. The VLDB Journal 30, 559–578 (2021).

4. Automated Database Design for Document Stores with Multi-criteria Optimization. Moditha Hewasinghage, Sergi Nadal, Alberto Abelló, , Esteban Zimányi. Under review in Knowledge and Information Systems (KAIS).

5. Managing Polyglot Systems Metadata with Hypergraphs. Moditha Hewasinghage, Jovan Varga, Alberto Abelló, Esteban Zimányi. International Conference on Conceptual Modeling (ER). 2018.

6. DocDesign: Cost-based Database Design for Document Stores. Moditha Hewasinghage, Alberto Abelló, Jovan Varga, Esteban Zimányi. International Conference on Scientific and Statistical Database Management (SSDBM). 2020.

7. DocDesign 2.0: Automated Database Design for Document Stores with Multi-criteria Optimization. Moditha Hewasinghage, Sergi Nadal, Alberto Abelló. International Conference on Extending Database Technology (EDBT) 2021.

5 Thesis Overview

In this section, we provide a brief overview of the results of this PhD thesis by discussing the contributions presented in each chapter.

5.1 On the Performance Impact of Using JSON, Beyond Impedance Mismatch

In Chapter 2, we study the impact on data design decisions in document stores. In this chapter, we identify the primary motivation behind choosing document stores compared to their RDBMS counterpart. Document stores adopt semi-structured data models, such as JSON, to easily accommodate schema evolution and overcome the overhead generated from transforming internal structures to tabular data (i.e., impedance mismatch). There exist multiple and equivalent ways to represent such semi-structured data physically. However, there is a lack of evidence about the potential impact on space and query performance of each design. To this end, in this chapter, we embark on the task of quantifying the performance impact of physical database design choices on document stores. We empirically compare multiple ways of representing JSON and relational data, which allows deriving a set of guidelines for efficient physical database design considering both JSON and relational options in the same palette.

Document stores are known to use semi-structured data, which implies that the data have some structure, but they are either irregular or not known beforehand. XML or JSON documents consists of a nested hierarchy of elements/key-value pairs with a single root. Child documents are an unordered sequence list of optional elements. This allows XML/JSON documents to be self-descriptive, eliminating the requirement of a schema. However, having a known structure facilitates optimal storage and query capabilities as in the case of RDBMS, where data are set of attributes, each with a concrete domain. Considering these differences, in this chapter, **we identify six data representational differences between document stores and RDBMS under three categories** as shown in Fig. 1.5.

Next, we **empirically quantify the impact of these design choices in semi-structured data**. Finally, in Chapter 2, we **evaluate the effect of these**
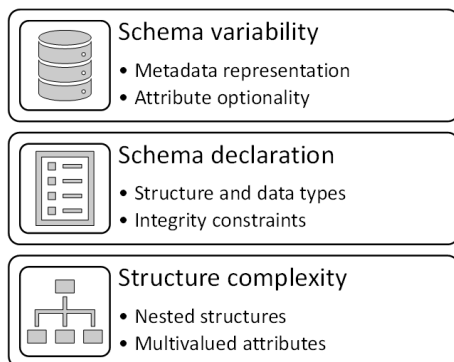
Fig. 1.5: Data representation difference between JSON and relational tables

different designs in an RDBMS and two alternative document store implementations. Thus, Chapter 2 is an introduction to data design choices in document stores and how each impacts the storage requirements and query performance.

5.2 Managing Polyglot System Metadata with Hypergraphs

In Chapter 3, we address the first problem of automated database design, the canonical model. This canonical model is necessary to represent the solution space for database design which consists of the alternate designs. Even though our focus is mainly on document stores, the canonical model we introduce is capable of representing other NoSQL data store categories as well as RDBMS. This results from document stores having characteristics of other database models and the canonical model being based on a graph. Furthermore, apart from the alternative designs, the canonical model maintains metadata of the underlying data such as cardinality of the relationships between entities, data type and the average size of stored attributes. Through this information, we are capable of calculating the storage requirements for a particular design.

The class diagram of the canonical model is shown in Fig. 1.6. First, we introduce the concept of a catalog (C) that represents the overall data storage design. This catalog can contain multiple data stores in order to support polyglot systems, which allows extending the work on this thesis (details in Chapter 3). Second, the canonical model is based on hypergraphs where an edge (referred to as hyperedge) can join any number of vertices instead of exactly two. In addition, we use a generalized hypergraph model where a hyperedge can contain other hyperedges. Thus, the canonical model constructs are of three levels.

- **Immutable** level of C contains information on the entities and the re-

lationships that are stored. This is represented as a graph and is a direct mapping from a typical ER diagram. Each vertex of this graph is identified as an *Atom (A)*. *Atoms* are of two types, *Class Atom (A_C)* represents an entity and *Attribute Atom (A_A)* represents attributes of an entity. The edges between the *Atoms (E_R)* represent the relationships between the entities and attributes as well as entities themselves. *Atoms* and *Relationships* also contain additional metadata. In the case of A_As, this will be the data type and the average size, A_Cs contain the represented entity's count, and E_Rs between A_Cs contain the corresponding cardinalitys.

- **Storage agnostic** level of C identifies the common constructs to any data store. These are *hyperedges (E_H)* that contain *A*s and *E_R*s as well as other *E_H*s. *Struct (E_{Struct})* represents the structure of the stored data. This could be a row of an RDBMS or a document in a document store. *Set (E_{Set})* represents a collection within the stored document for instance a table of an RDBMS or a collection or an array in a document store.

- **Document store-specific** constructs introduce specialized E_Hs for document stores. E_{Col}^{Doc} and E_{List}^{Doc} are specialized E_{Set}s representing document store collection and nested list/arrays respectively. E_{Top}^{Doc} and E_{Doc}^{Doc} are specialized E_{Struct}s that represent top level documents of a collection and internal documents (nested ones) of a document store collection.
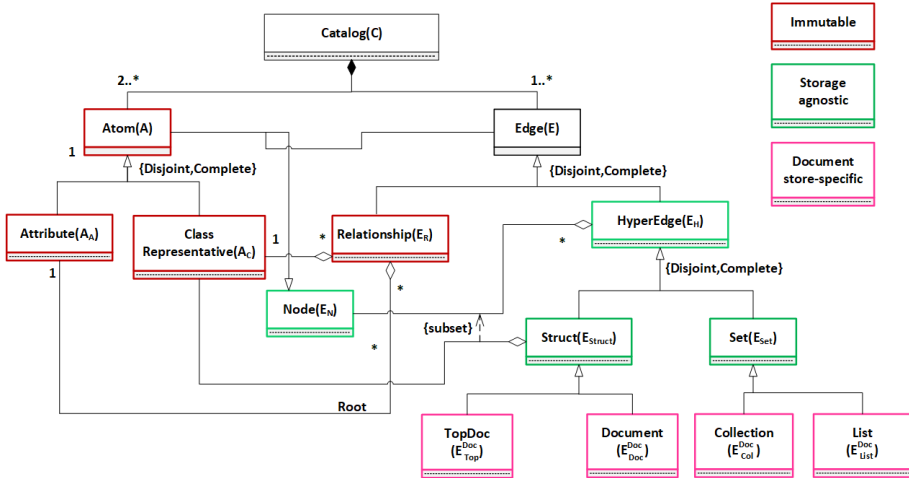
Fig. 1.6: Class diagram of the canonical model to represent heterogeneous data stores

Each of the E_H constructs adheres to eight definitions that help us guarantee a valid representation of a data store. Moreover, we introduce a set

of grammars specific to each data store type, representing the constraints within that data store storage system and its design constructs. For instance, traditional RDBMSs does not allow nested arrays. This can be defined as a rule by not allowing an E_{Set} within an E_{Struct} in an RDBMS setting. Next, we introduce simple query generation rules and algorithms capable of automatically generating data store-specific queries from the ones issued on the immutable data. Furthermore, we present algorithms to calculate storage statistics and physical access patterns for queries taking document stores as an example. Thus, this canonical model enables us to measure the essential attributes of a particular database design to evaluate it against another.

5.3 A Cost Model for Random Access Queries in Document Stores

Chapter 4 addresses the second problem of defining a cost model that can efficiently predict the query performance in a document store under a given query workload. Document stores have been widely adopted in different domains due to their ability to store semi-structured data and expressive query capabilities. However, implementations differ in terms of concrete data storage and retrieval. Unfortunately, a standard framework for data and query optimization for document stores is nonexistent, and only implementation-specific design and query guidelines are used. Hence, this chapter aims to aid in automating the data design for document stores based on query costs. For this, we define a generic storage and query cost model based on disk access and memory allocation that allows estimating the impact of design decisions.

We present a generic cost model for random access in document stores based on storage metadata and memory usage. Both these parameters are specific for different document store implementation decisions. We use the canonical model introduced in Chapter 3 to obtain the data storage metadata, and the physical storage metadata can be obtained from the disk storage structures. Since all document stores carry out data operations in memory, we first estimate the memory usage by considering the characteristics of the stored documents, their access patterns, and memory management algorithms. Memory usage depends on memory mapping, associativity, and cache eviction policies, and each of them consists of several possibilities (e.g., pre-determined or shared associativity). Therefore, we introduce formulas for different options and depending on the underlying document store; we pick the corresponding formulas to estimate the memory usage. Then, using this estimation and metadata storage size, we introduce a cost model for random access queries. Finally, we validate our work on two well-known document store implementations: MongoDB and Couchbase. The results show that the memory usage estimates have an average precision of 91%, and predicted costs are highly correlated with actual execution times. During this work, we have managed

to suggest several improvements to document storage systems. Thus, this cost model also contributes to identifying discordance between document store implementations and their theoretical expectations.

5.4   Automated Database Design for Document Stores

In Chapter 5, we introduce an automated database design methodology for document stores. The large number of potential designs together with multiple, often conflicting aspects such as storage space, query performance, and complexity of the documents makes finding the optimal design a complex one. To overcome these issues, in this chapter, we apply multicriteria optimization. Our approach is driven by a query workload and a set of optimization objectives. An overview of this approach is shown in Fig 1.7. The immutable level of the canonical model introduced in Chapter 3 is used to represent the entity-relationships as an immutable graph.
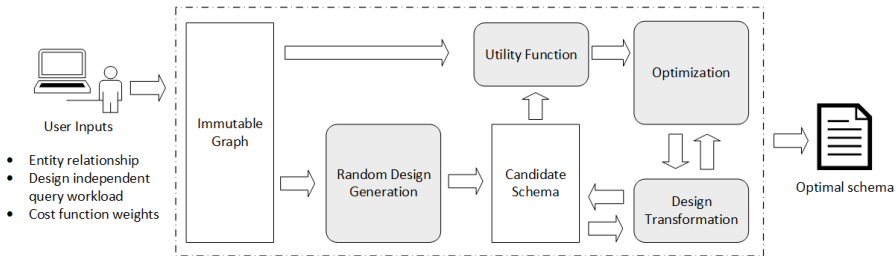
Fig. 1.7: Overview of automated schema design approach

We chose shotgun hill-climbing, also known as the random-restart hill-climbing algorithm, to obtain a Pareto-optimal design. In simple hill-climbing, an arbitrary solution is selected, and then a better solution is attempted by making incremental changes to the current solution. Shotgun hill-climbing iteratively does hill-climbing each time, starting with a random initial condition while maintaining the overall best solution. If a new run of a hill-climbing produces a better solution, it replaces the comprehensive best solution.

In order to apply shotgun-hill climbing for database design, first, we introduce a random design generation algorithm that produces a random initial design. Then, we need means of making incremental changes to a given design. To achieve this, we extend the canonical model described in Chapter 3 that is capable of representing alternating designs by introducing an algebra of transformations that can systematically modify a given design. Designs that produced through such transformations are shown in Fig. 1.8. Then, using these transformations, we implement a local search algorithm driven by a loss function that can propose near-optimal designs with high probability.

The loss function allows us to evaluate optimization criteria for a given
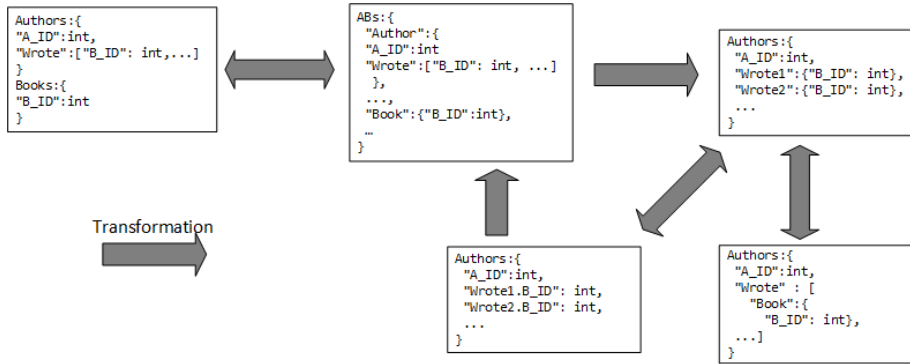
Fig. 1.8: Transforming document store designs

design. In our case, we evaluate four properties or cost functions for a particular design.

- The storage size

- The depth of the stored documents

- The heterogeneity of documents within a collection

- The query performance

The storage size, the depth of the documents, and the heterogeneity of documents within a collection can be easily obtained through the canonical model and the algorithms that accompany it, introduced in Chapter 3. Finally, the query performance is calculated with the cost model of Chapter 4 together with the storage metadata obtained through the canonical model.

Finally, we compare our prototype against an existing document store data design solution purely driven by query cost. Our proposed designs have better performance and are more compact with less redundancy.

6 Contributions

Figure 1.4 depicts an overall view of the contributions of this PhD thesis within the document store database design process. The contributions of this PhD thesis are summarized as follows:

- *Performance impact on semi-structured data design*. We systematically analyze the implications on the query performance introduced with semi-structured data storage. First, we identify six data representational differences between document stores and RDBMS. Next, we empirically

quantify the impact of these design choices. Finally, we evaluate the impact of these alternative designs in different document stores and RDBMS in terms of query performance and storage requirements. Thus, our contribution provides a broader and systematic evaluation of semi-structured database design, especially on JSON storage, beyond the obvious impedance mismatch advantage that it brings.

- *Canonical model for data stores*. We propose a canonical model based on hypergraphs to support representing alternative data stores, both RDBMS and NoSQL. The model consists of constructs that are immutable, storage agnostic, and document store specific. We can calculate metadata specific to each data storage system, such as storage size, with the information provided in the immutable layer. Moreover, we introduce a set of grammars specific to each data store and its design constructs. Through this, we generate data store-specific queries from ones issued on the immutable layer. Thus, the novelty of this model is its formalization and flexibility to represent storage systems and designs that can be used to represent the solution space of alternate database design choices.

- *Cost model for document stores.* We present a cost model for document stores that can predict relative query cost for random access queries. There are no formal cost models for document stores, mainly due to their semi-structured nature. Most document store implementations rely on primitive approaches to determine the query execution instead of formal query processing algorithms. Thus, our cost model is the first of its kind to the best of our knowledge. We propose a generic cost model based on storage metadata, disk access, and memory usage patterns. We estimate the memory usage of document stores based on memory mapping, associativity, and cache eviction policies and provide cost model components relevant to each choice. Then, depending on the underlying implementation of the document store, we combine the necessary components to obtain the overall cost model for a particular document store. This cost model allows us to estimate relative query performance on alternative designs and their behaviour instead of implementing the actual design in the document store and obtaining the query performance details. While implementing the cost model, we were able to find and suggest fixes to some issues in the caching algorithms in MongoDB. This strongly supports our claim on the necessity of having a formal cost model for query execution for document stores.

- *Automated database design for document stores.* Finally, we provide the capability to automatically propose the optimal design(s) for a document store. We propose a method that, given entity-relationships and query workload on top of the entities, selects the Pareto-optimal design(s). The

novelty of our approach is on the ability to fine-tune between alternate requirements of the end-user, such as the size of the final storage and the complexity of the documents as opposed to only relying on the query performance in the existing approaches, which might lead to missing out on certain alternatives.

Chapter 2

On the Performance Impact of Using JSON, Beyond Impedance Mismatch

Co-authoring declaration   This work has been done together with the post doctoral researcher Sergi Nadal, with an overall equal contribution from both.

1   Introduction

The relational model was defined as an abstraction level to gain independence of the file system and any internal storage structure [29]. Thus, we could gain flexibility and interoperability without losing efficiency by following a tabular representation and some normal forms [9]. Indeed, the first normal form (1NF) established that attribute domains had to be atomic (i.e., they could be neither compound-complex structures nor arrays). However, a rigid tabular structure is not adequate in modern agile software development, where the schema is under continuous evolution. Moreover, a well-known problem of RDBMS is the impedance mismatch, defined as the overhead generated by transformations from internal structures to tables, and then into programming structures [8].

The development of NoSQL systems, which adopt more flexible data representations, allowed to overcome the impedance mismatch [98]. Such data formats (e.g., JSON), are directly mapped from disk to memory.This is additionally achieved by breaking 1NF, allowing typical programming nested structures and arrays in the attribute values (e.g., MongoDB encourages denormalization[1]). Furthermore, such semi-structured formats, also allow to skip schema declaration, which is beneficial in highly evolving applications [97]. Nevertheless, it is not clear whether denormalization and schemaless is a conscious design choice, or merely a paradigm imposed by the limitations of NoSQL systems. Yet, the flexibility offered by NoSQL comes at a price, where each one of the associated design choices may widely change their physical representation, and thus profoundly impact performance. Practitioners have ignored this, and today make binary design decisions based on rules, and programming needs with no overall view of the system needs [89]. Thus, it is vital to consider the benefits and drawbacks posed by these different alternatives during the design process [13]. Relational and semi-structured data models, are not a simple binary choice, but a continuum of options with different degrees of (de)normalization. This idea is pursued by the co-relational model, which entails a wide range of complementary database design possibilities [86].

In this chapter, we quantify the performance impact of physical database design choices on NoSQL systems, focusing on the JSON data model. To this end, different design choices (i.e., equivalent representational differences) related to both metadata (i.e., schema), such as attribute embedding or optionality, and data, such as nested objects or arrays, are quantitatively scrutinized. We acknowledge that many DBMS features can affect performance (i.e., concurrency control and recoverability mechanism, distribution and parallelism

[1]`https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2`

management, connection pools and setup, etc.) [107]. Nevertheless, we only study the impact of design decisions on a semi-structured data model, being agnostic of the technological choice. Our main contributions are as follows:

1. We identify the main physical design characteristics of semi-structured data and compare them to their structured counterpart.
2. We empirically quantify the impact of design choices in semi-structured data.
3. We evaluate the different designs in a relational and NoSQL DBMS.

2   Related Work

The work carried out in [10] abstracts and homogenizes the modeling commonalities of NOSQL systems. It considers databases as sets of collections, which in turn are sets of blocks, finally represented by sets of entries.nDatabase design reduces to the identification of aggregates, driven by data access patterns and scalability needs. Experimental results show that following the proposed design method yields to worst query performance. Similarly, [58] proposes a subject-oriented methodology to design NOSQL databases. A conceptual model of the system is converted into an equivalent hypergraph representation, such that hyperedges identify specializations or aggregations among entities. For each hyperedge, an specific data model, either relational or co-relational. Then, based on the query workload, an affinity matrix determines which data are accessed together and, thus, must be allocated in the same vertical fragment. [34] proposes a method to generate NOSQL databases from a high-level conceptual model automatically. The authors propose the UML-like Generic Data Metamodel, integrating structural and data access patterns. Then, a set of transformation rules generate the specific constructs for the target model (e.g., document or column-family). Experimental results show improvements w.r.t. the state of the art on insertion and query performance for the generated databases.

Regarding performance, [57] benchmarks PostgreSQL and MongoDB. An OLAP-like workload is evaluated in both systems on real-world data from Github. The benchmark concludes that PostgreSQL yields higher performance results, but different design alternatives are not explored. The impact of normalized collections w.r.t. embedded objects in MongoDB is explored in [66], and it also empirically shows that querying embedded objects is orders of magnitude faster than their normalized counterpart using joins. Similarly, [110] benchmarks systems in the NOSQL realm (i.e., MongoDB and CouchDB) as well as RDBMSs with built-in JSON support (i.e., PostgreSQL and MySQL). This work differs from our setting, as it focuses on CRUD transactions for a simple document structure. Finally, [65] presents an adaptation of the TPC-C benchmark over MongoDB. One of the many adaptations performed to the

original schema is denormalizing structures, concluding that this improves runtime results w.r.t. the normalized version.

3 Representational Differences

The term semi-structured describes data that have some structure but is neither regular nor known a priori [1]. For example, a JSON document consists of a nested hierarchy of key-value pairs with a single root. Child documents are an unordered sequence list of pairs with optional presence. Hence, JSON documents are self-descriptive, and do not require a schema declaration, despite a known structure facilitates storage and encourages queries [3]. Conversely, a structured database distinguishes schema and instances [4]. The former is a set of attributes, each with a concrete domain, while the later is a tuple of values that belong to the corresponding domain in the previously declared schema. Hence, here, we present representational differences between semi-structured and structured data (i.e., equivalent alternatives to represent some datum exploiting the characteristics offered by each of both models), and discuss their potential impact on storage size, data insertion, and query performance. For each representational difference, we present patterns used in the empirical validation in Section 4.

3.1 Schema variability

A common schema is defined for all instances in structured databases, but in JSON, there may exist potentially different document schemata inside the same collection. Here, we focus on comparing alternative ways to represent the schema.

3.1.1 Metadata representation

Representing different schemata across JSON documents entails embedding their metadata into each instance (Fig. 2.1). This clearly impacts negatively the size of the database and consequently query performance. The more attributes are present, the more metadata (i.e., attribute names) will be embedded into each document. Additionally, the ratio between the size of data and metadata is clearly an important factor to consider (i.e., attribute name length w.r.t. its values). Thus, we need to consider (a) the absolute amount of metadata by analysing different number of attributes (from 1 to n), and (b) the relative amount of metadata by analysing different ratios (by increasing the value length from 1 to m, while at the same time that decreases the attribute name length in the same number of characters).

JSON
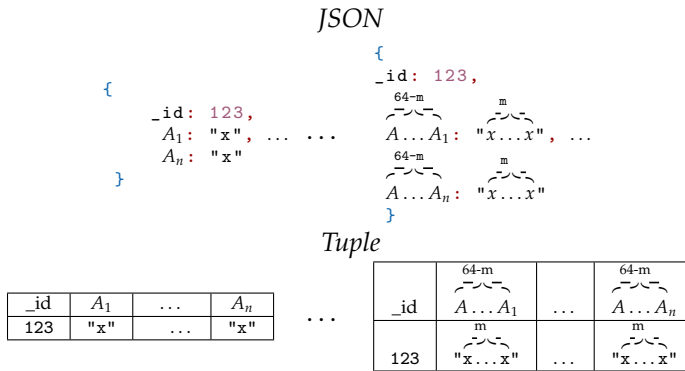
Fig. 2.1: Alternative representations of Metadata

3.1.2 Attribute optionality

Another feature of the semi-structured data model is the possibility to skip the representation on an attribute in the absence of its value (as the case of *J-Abs*, Fig. 2.2). However, it also supports to, either use a special value outside the domain (as in *J-NULL*) or use a specific value inside the attribute domain (as in *J-666*). Notice that in a relational representation, as the schema is fixed and common to all instances, only the last two options are possible (as in *T-NULL* and *T-666*, respectively). The impact on space and performance of these representations will vary depending on the percentage of absent/present values for the attribute.
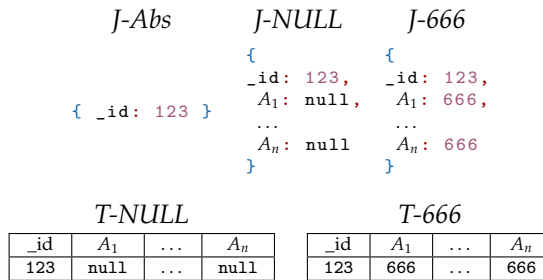
Fig. 2.2: Alternative representations for optional attributes

3.2 Schema declaration

In order to benefit from Schema declaration and validation in semi-structured databases, one must adopt additional constructs. JSONSchema is a JSON-based schema language that allows to constrain the shape, types and values of JSON

documents. Here, we will evaluate the impact of both structure plus data type declaration, and integrity constraint (IC) validation separately.

3.2.1 Structure and data types

To validate structure and data types, `JSONSchema` uses the `properties` key. For each attribute, it is possible to specify its data type, which can be either a primitive or complex object. Furthermore, the `required` key represents an array enumerating the list of expected attributes. Fig. 2.3 depicts the exemplary document patterns considered. Clearly, this declaration has no impact on database size, since it does not grow with instances. However, it has a cost on insertion, corresponding to validating presence and domain, and on the other hand, it could potentially benefit query time by saving an explicit casting and type conversion.

J-Typ   *T-Typ*

```
                    {
                    "type": "object",
{                   "properties": {
_id: 123,             "A₁": {"type": "number"},     _id   A₁    ...   A₆₄
A₁: k, ...                                          123   k     ...   k
A₆₄: k                ...
}                     "Aₙ": {"type": "number"},
                    required: ["A₁",...,"Aₙ"]
                    } }
```
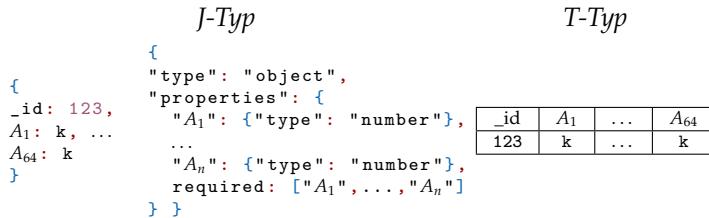
Fig. 2.3: Alternative representations of structure and data type validation

3.2.2 Integrity constraints

Besides the data type validation mechanisms, `JSONSchema` also offers means to represent integrity constraints for attributes. Here, as depicted in Fig. 2.4, we focus on enforcing ranges of values. In relational databases, this is achieved via `CHECK` constraints. As above, this has no impact on the size of the database but will have some on the insertion since it has to be checked before accepting the data. Despite this, it might also be used to perform some semantic optimization at query time; we consider this is technology-specific (i.e., not directly dependent on the data representation) and will not be evaluated in Section 4.

3.3 Structure complexity

An RDBMS conforms to 1NF, yet a semi-structured one relaxes such restriction, which allows storing nested and multi-valued data. Here, we study the impact of different complexity degrees on data according to that.

J-IC

```
{                                  {"type": "object",
 _id: 123,                          "properties": {
 A_1: k, ...                         "A_1": {
 A_64: k                              "type": "number",
}                                     "minimum":−k',"maximum: k'}, ...
                                      "A_n": {
                                       "type": "number",
                                       "minimum":−k',"maximum: k'
                                     }}
```

T-IC

```
                                   ALTER TABLE T ADD CONSTRAINT
                                   val_A_1 CHECK
                                   (A_1 BETWEEN −k' AND k');
| _id | A_1 | ... | A_64 |         ...
| 123 | k   | ... | k    |         ALTER TABLE T ADD CONSTRAINT
                                   val_A_n CHECK
                                   (A_n BETWEEN −k' AND k');
```
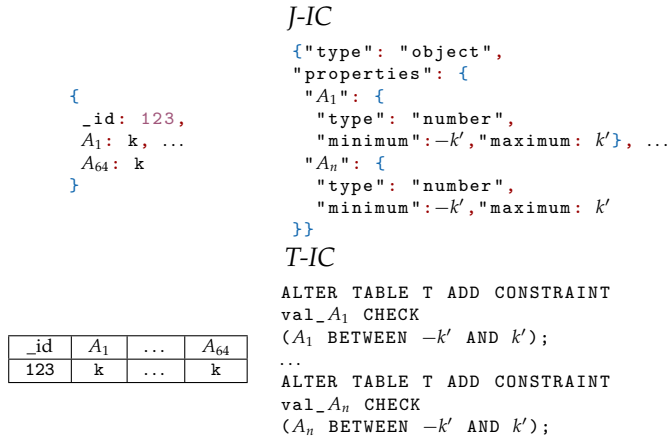
Fig. 2.4: Alternative representations of Integrity Constraints (IC) validation

3.3.1 Nested structures

Documents allow to explicit into a data structure conceptually independent objects, which are accessed using dot notation. Yet, it is unclear what is the impact regarding size (i.e., with an increasing number of brackets in the document), and on querying such structures. To explore this, we will experiment with a range of levels and attributes as shown in Fig. 2.5). Precisely, we will evaluate (a) increasing document sizes (i.e., *Nest-one*), and (b) constant document sizes (i.e., *Nest-all*); both w.r.t. the number of nesting levels. *Nest-1* indicates that there is only one attribute in the lowest level, while *Nest-all* contains less attributes the more levels we have. For instance, with 32 nesting levels, *Nest-one* has only A_{33}, while *Nest-all* has attributes A_{33} to A_{64}. Thus, in the latter, for every level we add together with the required extra characters (i.e., :, {, and }), we remove an attribute. Consequently, the overall size remains constant in terms of document length, but not in physical storage space due to the encoding of integer values being used.

Nest-one   *Nest-all*

```
{ _id: 123,                   { _id: 123,
  L_1: { ...                    L_1: { ...
    L_n: {                        L_n: {
      A_{n+1}: k }...               A_{n+1}: k, ...
}}                                  A_64: k }...
                              }}
```
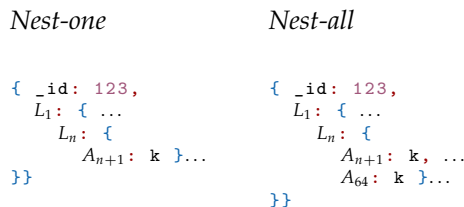
Fig. 2.5: Representation of nesting structures

3.3.2 Multi-valued attributes

Only modern object-relational DBMSs have adopted variable-length multidimensional arrays as data type, an aspect present in JSON by definition. Yet, it is unclear what is the impact of managing such types. On bounded arrays, one could argue that it might be better to store each position as an independent attribute, as depicted in Fig. 2.6, where we distinguish, for both JSON and tuples, *array* and *multi-attribute* alternatives. Multi-valued attributes could also be stored in a separate normalized table, however such independent structure would compete for resources, heavily impacting insertion and query (see Appendix A). We consider such eviction policies are technology-specific, thus they will not be evaluated.
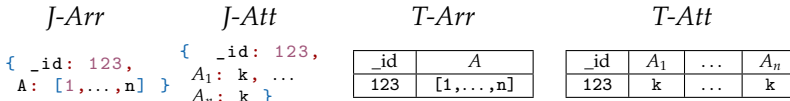
Fig. 2.6: Representation of multi-valued attributes

4 Experimental evaluation

We conducted experiments to evaluate the impact of the modeling choices discussed in Section 3, using PostgreSQL v12 (an open-source RDBMS supporting native JSON storage) to compare the differences between relational and JSON alternatives. Moreover, we also used MongoDB v4.2 (nowadays, the most popular choice for document stores) to validate the consistency of results. It is important to note that our objective is not to perform a technological comparison, but to evaluate the impact of document design choices.

Experimental setup. The experiments were conducted on a Debian 4.9 OS with Intel Xeon E5520 and 24GB of RAM. They were implemented on Java 8 using the PostgreSQL Java driver v42.2 and MongoDB Java driver v3.12. No specific tuning was performed for any system, using the default parameters. However, we disabled compression in MongoDB to facilitate its comparison with PostgreSQL. Also, we cleared the operating system cache and restarted the DBMS between each execution to ensure caching mechanisms were not affecting.

Thus, we got three metrics: (a) storage size in *MB* (given by, respectively, `pg_total_relation_size()` and `db.collection.status()`, for PostgreSQL and MongoDB); (b) overall runtime of insertions in seconds; and (c) median runtime to aggregate a numeric attribute in seconds over 20 repetitions. To store JSON in PostgreSQL, we created a table with two attributes: a `CHAR(24)`

to store the ID (equivalent to `Object_ID` in MongoDB) and a `JSONB` to store the document.

Then, we generated 1 million random documents according to each schema pattern in Section 3, over an exponentially increasing parameter, which were inserted in 100 batches of 10K documents.[2] Finally, to minimise the impact of impedance mismatch, queries retrieve only a single value by aggregating some numerical attributes in the data. Unfortunately, by default, MongoDB stores 32-bits integers [3], while PostgreSQL uses 64-bits[4], which in the end causes differences in storage size and consequently in insertion and query performance.

4.1 Schema variability

For schema variability, we conducted three experiments overall because we already had two patterns regarding metadata embedding (Section 3.1.1): (i) change the number of numeric attributes in a document; and (ii) change the data-metadata ratio, keeping a fixed number of attributes.

Varying document size. Fig. 2.7a depicts storage space growth as the number of attributes increases. JSON always requires more space than tuples, due to metadata being replicated in every document. We can observe the same trend in Fig. 2.7b, showing that more attributes lead to longer insertion times. However, although storage space for a tuple is smaller in all cases, insertion time is shorter only for few (i.e., four) attributes. Beyond that, JSON insertion is faster (due to no type checking, as shown later in Section 4.2). Again, at query time, runtime increases with the number of attributes, as seen in Fig. 2.7c. Nevertheless, oppositely to insertion, tuples perform faster (since they benefit from the work done at insertion time). In all cases, we can see that PostgreSQL and MongoDB follow the same trend on storing JSON. They only differ in the physical format, which requires less space in the latter (64-bit vs. 32-bit integers). Thus, MongoDB generates less I/O (roughly half), improving insertion and query time.

Constant document size. Aiming to stabilise the overall size of the document, we keep constant the sum of characters between attribute name and value. Thus, we have one numerical attribute for the queries and consider other nine string attributes, changing at once their data to metadata ratio by changing the length of attribute name and value keeping a constant of 64 characters for both together. The number is chosen based on PostgreSQL having a limit of 63 characters for attribute names, so the attribute name length ranges from 1 to 63 and the value length from 63 to 1. As shown in Fig. 2.8a, since attribute

---

[2]The source code is available in `https://github.com/dtim-upc/MongoDBTests`.
[3]`https://docs.mongodb.com/manual/reference/bson-types`
[4]`https://www.postgresql.org/docs/12/datatype-json.html`

(a) Storage   **(b)** Insertion time   **(c)** Query time
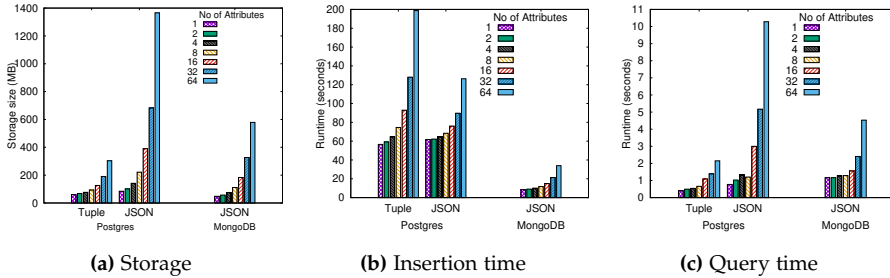
Fig. 2.7: Effect of metadata embedding with changing number of attributes

name is only stored once, independently of the number of tuples, the storage space taken by the tuples decreases with the growth of the attribute name length. Oppositely, attribute names are redundantly stored in all documents in JSON, so the overall size remains constant except for the last bar, seemingly due to the presence of a step function in physical storage allocation. This is confirmed in MongoDB, where the gradual growth in space is more apparent. Interestingly, PostgreSQL and MongoDB storage size for JSON is much closer in this experiment as most of the attributes are strings instead of integers. Insertion and query times, in Fig. 2.8b and 2.8c, respectively, follow the same trend as the attribute length grows in all cases, which indicates I/O is always the dominant factor.

(a) Storage   **(b)** Insertion time   **(c)** Query time
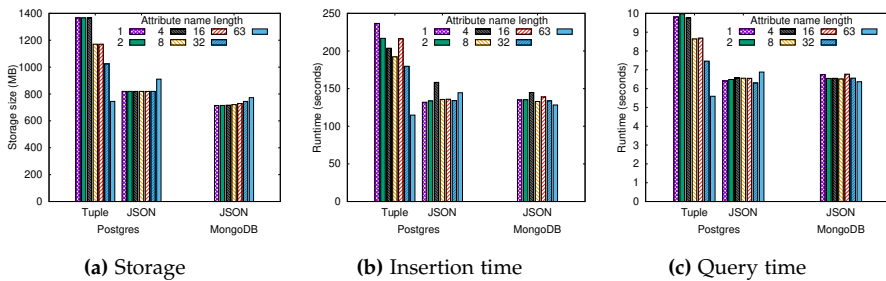
Fig. 2.8: Effect of metadata embedding with changing data-metadata ratio

Optional attributes. Regarding attribute optionality, we consider five alternatives to represent the absence of values in the attributes (Section 3.1.2). Thus, the pattern consists of 64 integer attributes (potentially removed all at once), and one fixed-length string of size 64 to guarantee a minimal document size when the former are removed. Thus, we vary in the experiment the percentage of documents without value for their integer attributes. Regarding storage space, as shown in Fig. 2.9a, the worst option to represent absence of data is

using a value inside the domain (i.e., *T-666* and *J-666*), which keeps a constant size. In both tuples and JSON, we can use a *null* special value (i.e., *T-NULL* and *J-NULL*), which clearly saves space as attribute values disappear. However, the complete absence of the attribute in JSON (namely *J-Abs*), reduces the storage space the most due to the saving also in the metadata. As before, storage space in MongoDB follows the same trend as in PostgreSQL, but with smaller values due to the different encoding of integers. Regarding insertion time (Fig. 2.9b), also the trend coincides with that of the storage used for JSON in both systems. However, tuples in PostgreSQL keep a constant insertion time, because the dominant factor is not I/O, but validation and formatting of data, which is not even compensated by the saving in metadata storage. When querying the data, we tested both summing and counting their presence (Fig. 2.9c and 2.9d, respectively) with similar results. We can see that, in all cases, the dominant factor of the query time is I/O, and consequently follows the trends and proportions of storage space.

(a) Storage

(b) Insertion time

(c) Query time

(d) Count nulls
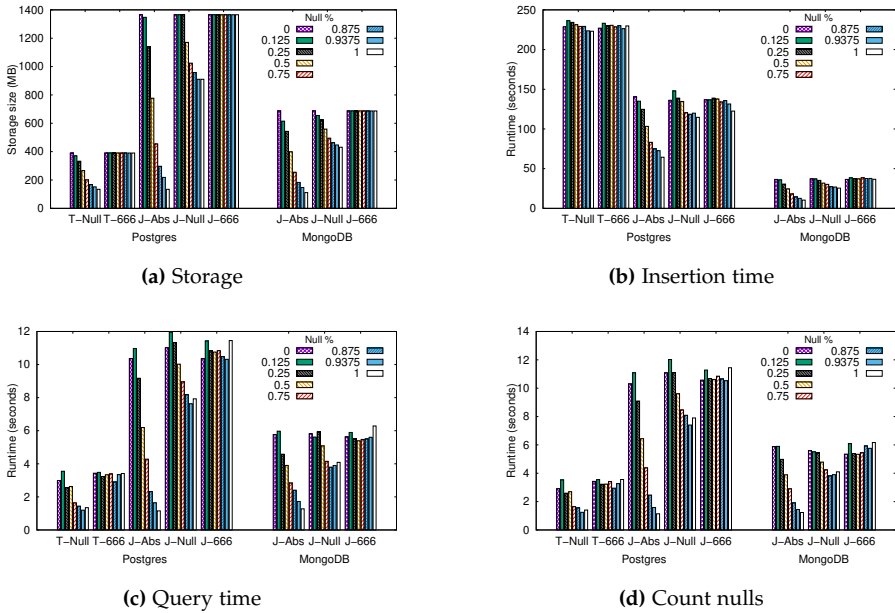
Fig. 2.9: Effect of optional values with different percentages

4.2 Schema declaration

As discussed, schema declaration does neither affect the overall storage size nor query time. Thus, we only report insertion time and do it together for

both data types and ICs.

Type and constraint validation. Regarding type and IC checking (Sections 3.2.1 and 3.2.2), we generated documents with 64 attributes and declared type and ICs in an incremental manner (from 1 to 64). To enforce JSON schema declaration in PostgreSQL, we used the *postgres-json-schema*[5] extension. In MongoDB, this is a built-in feature that can be simply enabled with the operator $jsonSchema, which is provided at creation time of the collection. For the sake of completeness, we show the insertion time of tuples (Fig. 2.10), however, in this case, all data types must always be declared, leading to constant insertion time. Oppositely, when inserting JSON, time increases with data types declaration, confirming the consequent overhead. Checking concrete ICs on top of data types, substantially increments the overhead. Both systems confirm trends, the only difference being that MongoDB mechanism is built-in, consequently faster.
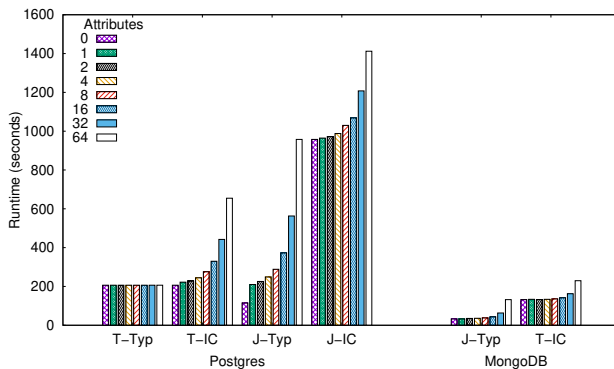
Fig. 2.10: Effect of schema declaration affecting different number of attributes

4.3 Structure complexity

Finally, we analyse the impact of breaking first normal form by either nesting documents (Section 3.3.1) or storing multi-valued attributes (Section 3.3.2). Notice that only the latter is available in relational implementations.

Nested structures. As depicted in Fig. 2.11a, the storage size of nesting one attribute increases the document size with the increasing number of levels. Nevertheless, when keeping the document size constant, PostgreSQL has an overall constant size in all cases except for 64 levels, where the storage size suddenly drops. Further investigation (using *pg_column_size*), revealed that the physical storage size of an individual document slightly decreases

---

[5]https://github.com/gavinwahl/postgres-json-schema

(a) Storage   **(b)** Insertion time   **(c)** Query time
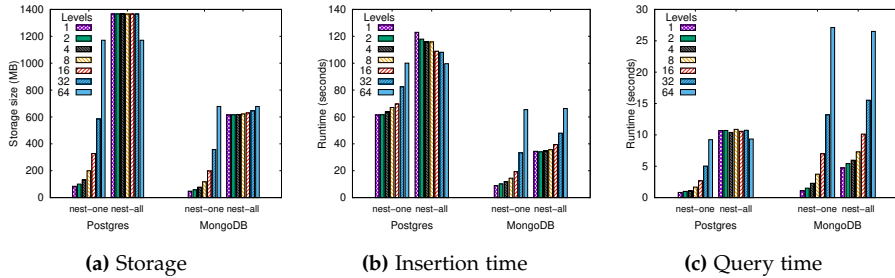
Fig. 2.11: Effect of nesting structure with varying number of levels

as the number of levels increases in PostgreSQL encoding. This is, however, considerably small, and the storage structures round up the total size. We do not appreciate such decrement in MongoDB, which slightly increases the physical storage when the number of levels increases, even with constant document size. The integer encoding difference (64-bits vs. 32-bits) explains this opposite behavior. Yet, Fig. 2.11b shows that despite insertion times follow the trend of the individual document storage sizes, we appreciate an extra overhead in MongoDB beyond that of purely I/O. Also Fig. 2.11c, where PostgreSQL performs better than MongoDB (despite having higher I/O), and MongoDB have a clear upward trend with the increasing number nesting levels as opposed to constant runtime in PostgreSQL confirms the overhead nesting generates in MongoDB.

Multi-valued attributes. Regarding the storage of multi-valued attributes (Section 3.3.2), we generated documents with the number of values per attribute ranging from 2 to 64 for the different options. For tuples, we used either PostgreSQL native array storage (*T-Arr*) or separate attributes for each value (*T-Att*), and similarly for JSON either as an array in the document (*J-Arr*), or as separate attributes (*J-Att*). Regarding storage size, Fig. 2.12a depicts that both systems take more space for JSON than tuples, because of the saving of tuples on metadata replication. While in tuples both options use the same space, in JSON arrays are clearly more efficient, since separate attributes require more characters (the same behavior is confirmed in MongoDB, but mitigated by its smaller encoding of integers). However, Fig. 2.12b shows that despite insertion time in JSON is dominated by I/O, in tuples inserting to an array is faster than inserting multiple attributes, due to the overhead of parsing and validating independent attributes in front of one single array. Nevertheless, the extra processing at insertion time pays off at query time (Fig. 2.12c), where processing the independent attributes is faster than digging inside the array. For JSON, we appreciate the same benefit of querying independent attributes in PostgreSQL, but surprisingly the opposite behavior in MongoDB, where

(a) Storage

(b) Insertion time

(c) Query time
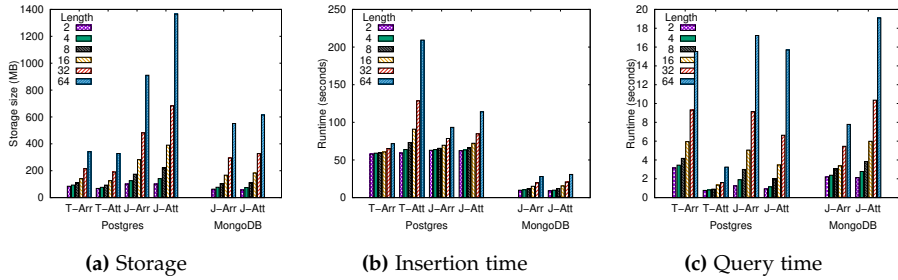
Fig. 2.12: Effect of multi-valued attributes with varying number of values

processing the array is systematically faster. Both MongoDB and PostgreSQL have to sum individual attributes in *J-Att*. However, MongoDB has a built-in function that sums the content of the array, which is more efficient, and on the contrary, PostgreSQL needs to unwind the array in order to calculate the sum, which is more expensive.

5 Discussion

Fig. 2.13 summarizes all results with regard to storage space, load time, and query time. For this, we calculated the average of all measurements per representational difference for each of the three options (i.e., Tuples, and JSON in both systems). Since data follows different patterns in each case, we separately min-normalize per case (e.g., divide the minimum of the three averages for nested data by the average for Tuples) and plot them all in the corresponding radar chart. This means values further away from the center of the radar are better than the ones closer, and the bigger the area of the polygon, the better the system performs.

(a) Storage

(b) Insertion time

(c) Query time
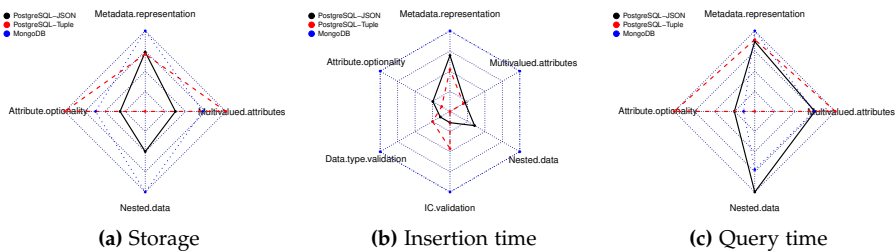
Fig. 2.13: Multidimensional view of experimental results

According to Fig. 2.13a, storing tuples takes the least amount of space in all

cases except metadata representation. On interpreting this, we acknowledge the impact of the ratio between metadata and data, which is fixed to be relatively high in all experiments. Thus, attribute names should always be encoded in JSON to shorten them as much as possible and improve that ratio. Obviously, this is more relevant, for example, if values are numeric than if they are strings (the former requiring less space, in general). Within JSON, PostgreSQL storage size is much larger than MongoDB in all the cases, due to the different encoding of integers (64-bits vs. 32-bits).

According to Fig. 2.13b, it is clear looking at PostgreSQL that loading JSON is faster than tuples, except for data type and integrity constraint validations. However, it is important to note that the validation of JSON was carried out through a third-party plugin, which definitely impacts the results. MongoDB being a native document store, has a clear advantage over PostgreSQL JSON storage in loading data (at the end of the day, JSON is stored as a column in a PostgreSQL table), beating even tuple storage in the validation dimensions. This, however, can come not only from using JSON format but from other DBMS characteristics (e.g., lack of ACID transactional support).

Finally, Fig. 2.13c depicts that tuples, in general, perform better in queries. This is so because they use less space, in general, and benefit from validation at insertion time. Thus, we can see that when the space-saving is lost depending on the data-metadata ratio, so the benefit is mostly lost at query time, as well. Nonetheless, JSON representation is at a disadvantage, as each of the documents needs to be parsed and processed on demand. Consequently, we should consider the trade-off between the pressure of fast ingestion and the long term benefit of recurring queries. It is also interesting to see that even though the storage size of JSON is larger in PostgreSQL, this is still faster than MongoDB. We believe this fact results from the differences in how query engines handle the calculations. PostgreSQL benefits here from the well-optimized aggregation operations in the relational engine, which data stored in JSON format also have access to.

Chapter 3

Managing Polyglot Systems Metadata with Hypergraphs

This chapter has been published as a paper and an extension in:

- Proceedings of the 37th International Conference on Conceptual Modeling (ER). 2018
  DOI: `https://doi.org/10.1007/978-3-030-00847-5_33`

- Data and Knowledge Engineering, 134, 101896. 2021
  DOI: `https://doi.org/10.1016/j.datak.2021.101896`

The layout of the papers has been revised.

1   Introduction

With the dawn of the big data era, the heterogeneity among the data storage models has expanded drastically, mainly due to the introduction of NoSQL. There are four primary data store models in NoSQL systems: (i) Key-value stores perform like a typical hashmap, where the data is stored and retrieved through a key and an associated value; (ii) Wide-column stores that manage the data in a columnar fashion; (iii) Document stores that represent data in a document-like structure, which can become increasingly complex with nested elements; (iv) Graph stores that are instance-based and store the relationships between those instances. The heterogeneity is not only limited to the data models but also various implementations of the same data model can be entirely different from one another due to the lack of a standard.

Heterogeneous systems can be useful in different scenarios because it is highly unlikely that a single data store can efficiently handle all the requirements of the end-user. Therefore, it is common to use different ones to manage different portions of the data. This allows controlling the storage and retrieval more efficiently for different requirements. Hence, polyglot systems were introduced, similar to traditional Federated Database Systems (FDBMS), but with more complexity considering the need to handle semistructured data models. Due to the heterogeneity at different levels, most of the work on polyglot systems [20, 38] suggests the implementation of wrappers or interfaces for each participating data store. However, this becomes more complex as the number of participating data store types grows.

The catalog (see [45]) maintains the meta-information of the data store. Having one for a polyglot system enables end-users to have a clear view of the complex system. Its metadata plays a significant role in understanding the overall picture of the underlying infrastructure. Moreover, it also helps to improve the design of the polyglot system and determine the statistics and the access patterns needed for different query requirements. It is essential to answer questions such as: What is the structure of the data being stored? Where is a piece of data stored? Is it duplicated in another store? What is the best way to retrieve this data? What would be the storage requirements for a particular data store design? What are the access patterns of a particular query over a particular datastore? Nevertheless, little research addresses the managing of metadata in polyglot systems. This is mainly due to the lack of a design construct that can represent heterogeneous, semistructured data. In this paper, we address the metadata management in polyglot systems by extending an already existing NoSQL design method [11, 12], and formalizing the constructs through hypergraphs.

The Save Our Systems (SOS) Model [11] claims to capture the NoSQL modeling structures in data design for key-value stores, document stores, and

wide-column stores utilizing three main constructs: attributes, structs, and sets. These constructs and their interactions allow representing the physical storage of above NoSQL systems. The fact that the model is simple makes it compelling in representability, but the lack of formalization leaves space for ambiguity and hinders the automation of metadata management in such settings. Instead, it is simply used as a common programming interface for data exchange.

In this Chapter, we formalize SOS using a hypergraph-based representation, defining a common conceptual model for the metadata of any NoSQL system, which we have formalized through definitions of the concepts based on logics. RDF is considered to be able to represent any kind of data and is often used as a data interchange format. Therefore, we make the assumption that we have exemplars of the data in the polyglot system in RDF. Then, we build a hypergraph that maps to different data design constructs, representing the SOS model over the information. We represent the catalog of the polyglot system using these constructs and introduce a simple query generation algorithm to show the usefulness of our approach. Next, we explore different data store models, identify their design constructs, introduce their design constraints, and define query generation rules for each of them. Afterwards, we introduce how our catalog can be used to calculate storage statistics and physical access patterns for queries using document stores as an example. Finally, we show the feasibility of our approach using a use case of an existing polyglot system by representing its metadata catalog through our constructs.

The simple, yet powerful hypergraph-based approach presented in this chapter is a step towards representing heterogeneous, semistructured data in a formal manner as well as managing the corresponding metadata of a polyglot system. It proves to be useful concerning (i) expressiveness: the ability to express different representations, regardless of their complexity and (ii) semantic relativism: the ability to accommodate different representations of the same data, as defined in [99].

2   Preliminaries

In this section, we introduce the basic concepts of Resource Description Framework (RDF) [72] and SOS Model [11, 12] that are used in our approach.

2.1   Resource Description Framework (RDF)

The Resource Description Framework [72] is a World Wide Web Consortium (W3C) specification for representing information on the Web. It is a graph-based data model that enables sharing of information and statements about available resources.

RDF represents data as triplets consisting of subject, predicate, and object (s, p, o). These can be resources that are identified by an Internationalized Resource Identifier (IRI), which is a unique Unicode string within the RDF graph. An object can also be a literal, which is a data value. An example of RDF is shown in Listing 3.1, written in Turtle notation. The example contains information about music albums, artists, and songs and is used throughout the paper.

```
@prefix foaf: <http://xmlns.com/foaf/0.1> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://dbtune.org/jamendo/artist/dylan> rdf:type mo:MusicArtist;
  foaf:name "Bob Dylan"^^xsd:string;
  foaf:made <http://dbtune.org/jamendo/record/emp>.

<http://dbtune.org/jamendo/record/emp> rdf:type mo:Record;
  dc:title "Empire Burlesque"^^xsd:string;
  foaf:maker <http://dbtune.org/jamendo/artist/dylan> ;
  mo:track <http://dbtune.org/jamendo/track/Seeing>, <http://dbtune.org/jamendo/track/Tight> .

<http://dbtune.org/jamendo/track/Tight> rdf:type mo:Track;
  dc:title "Tight Connection to My Heart"^^xsd:string .
<http://dbtune.org/jamendo/track/Seeing> rdf:type mo:Track;
  dc:title "Seeing the Real You at Last"^^xsd:string .
```

Listing 3.1: Example RDF dataset [1]

2.2 SOS Model

The high flexibility of NoSQL systems gives freedom to have multiple designs for the same data. A particular data design built focusing on a specific scenario can result in adverse performance when applied in a different context. Most of the data design for NoSQL is carried out based on concrete guidelines for different datastores and access patterns. Nevertheless, recent approaches propose generic design constructs for NoSQL systems. For our approach, we decided to use the SOS model [11, 12] as a starting point.

The SOS model introduces a basic common model (or a meta-layer), which is a high-level description of the data models of non-relational systems. This model helps to handle the vast heterogeneity of the NoSQL datastores and provides interoperability among them, easing the development process. The primary objective of the meta-layer is to generalize the data model of heterogeneous NoSQL systems. Thus, it allows standard development practices on a predefined set of generic constructs. The meta-layer reconciles the descriptive elements of key-value stores, document stores, and record stores. These different data models exposed by NoSQL datastores are effectively managed in the SOS data model with three major constructs: *Attribute*, *Struct*, and *Set* [12].

---

[1] http://dbtune.org

A name and an associated value characterize each of these constructs. The structure of the value depends on the type of construct. An *Attribute* can contain a simple value such as an Integer or String. *Struct*s and *Set*s are complex elements which can contain multiple *Attribute*s, *Struct*s, *Set*s or a combination of those. SOS Model mainly addresses data design on document stores, key-value stores, and wide-column stores [11]. Each of the datastore instances is represented as a set of collections. There can be any arbitrary number of *Set*s depending on the use case. Simple elements such as key-value pairs or single qualifiers can be modeled as *Attribute*s and groups of *Attribute*s, or a simple entity such as a document can be represented as a *Struct*. A collection of entities is represented in a *Set*, which can be a nested collection in a document store or a column family in a wide-column store. A possible SOS representation of the example is shown in Fig. 3.1.
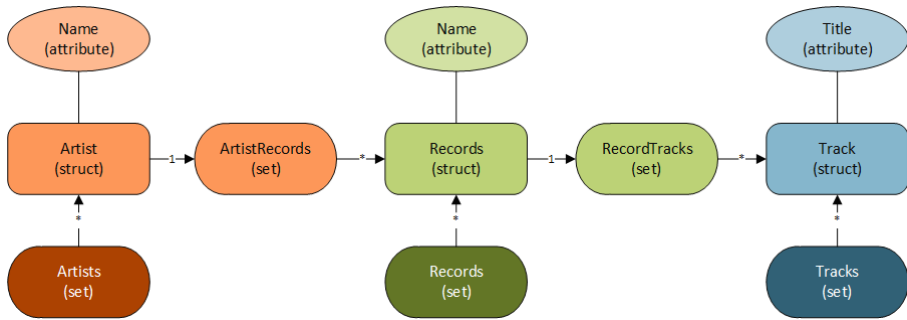
Fig. 3.1: SOS representation of the example

3   Formalization

In this section, we introduce and formalize our data model, which is based on representative exemplars in RDF format of each kind of instances in the underlying data stores. This RDF graph contains the classes and user-defined types of the polyglot system. Thus, having the schemas of the underlying data stores and having a global schema is important in our model. However, this is beyond the scope of the current work. The schema of a structured data store such as RDBMS can be extracted through the underlying DDL. Schema inference from semi-structured data has also been carried out [30]. Moreover, previous work has shown that this global schema can be obtained by extracting the schema of each data store and reconciling them [39, 103].

Building on top of the RDF data model discussed in Section 2.1, we introduce our design constructs based on the SOS Model. Figure 3.2 shows the overall class diagram of our constructs, where thicker lines represent the

elements already available in the SOS (namely *Set*s, *Struct*s, and *Attribute*s), and the relationship multiplicities defined in SOS are preserved. On top of that, we introduce additional constructs to aid the formalization process and manage metadata. From here on, we use letters in blackboard font to represent sets of elements (e.g., $\mathbb{A} = \{A_1, A_2...A_n\}$).

We rely on the concept of hypergraph, which is a graph where an edge (aka hyperedge) can relate any number of elements (not only two). This can be further generalized so that hyperedges can also contain other hyperedges (not only nodes).
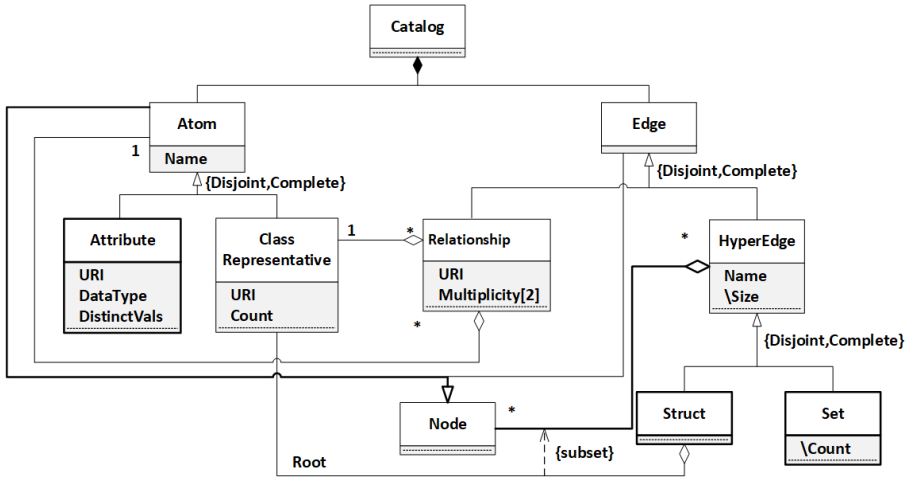
Fig. 3.2: Class diagram for the overall catalog

We define the overall polyglot system catalog as composed by the schema and the essential elements that support a uniquely accessible terminology for the polyglot system.

Definition 1
A polyglot catalog $C = \langle \mathbb{A}, \mathbb{E} \rangle$ is a generalized hypergraph where $\mathbb{A}$ is a set of atoms and $\mathbb{E}$ is a set of edges.

Definition 2
The set of all atoms $\mathbb{A}$ is composed of two disjoint subsets of class atoms $\mathbb{A}_C$ and attribute atoms $\mathbb{A}_A$.
Formally: $\mathbb{A} = \mathbb{A}_C \cup \mathbb{A}_A$

*Atom*s are the smallest constituent unit of the graph and carry a name. Moreover, every A_C contains a URI that represents the class semantics, while every A_A carries the datatype and a URI for the user-defined type semantics. Additionally, the information on the *Atom*s can be enhanced by the distinct values for A_As and the number of instances on A_Cs identified as *Count*.

Definition 3

The set of all edges $\mathbb{E}$ composed of two disjoints subsets of relationships $\mathbb{E}_R$ that denote the connectivity between $\mathbb{A}$, and hyperedges $\mathbb{E}_H$ that denotes connectivity between other constructs of C.
Formally: $\mathbb{E} = \mathbb{E}_R \cup \mathbb{E}_H$

Definition 4

A relationship $E_R^{x,y}$ is a binary edge between two atoms A_x and A_y and a URI u that represents the semantics of E_R. At least one of the atoms in the relationship must be an A_C.
Formally: $E_R^{x,y} = \langle A^x, A^y, u \rangle | A^x, A^y \in \mathbb{A} \wedge (A^x \in \mathbb{A}_C \vee A^y \in \mathbb{A}_C)$

The E_Rs that connect two A_Cs can include the multiplicities between the two classes. Since the relationships are bidirectional multiplicities are also diploid.

This graph $G = \langle \mathbb{A}, \mathbb{E}_R \rangle$ is a representation of the available data, i.e., an RDF translation of the original representatives of the data contained in the polyglot system, that we assume to be given. G us immutable as it contains the knowledge about the data. Figure 3.3 shows the graph G of the original RDF example in Listing 3.1. Here, we can assume that each artist can have 5 records and each record has 2.5 artists on average. A record has on average 15 tracks and each track is in 1 record. Finally, there are 20 *Artist*, 100 *Record*, and 1500 *Track* instances.
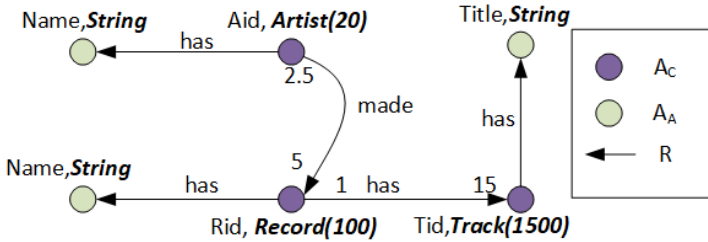
Fig. 3.3: Translated graph built from the RDF

We build our data design on top of G, based on the constructs introduced in SOS model. Thus, we make use of *Hyperedge*s and give rise to our hypergraph-based catalog C. An incidenceSet of an *Atom* or a *Hyperedge* contains the immediate set of E that *Atom* or *Hyperedge* is part of, respectively.

Definition 5

The transitive closure of an edge E is denoted as E^+, where $E \in E^+$, $\forall e \in E^+ :$
$e \in e'.incidenceSet \implies e' \in E^+$

Definition 6

A hyperedge E_H is a subset of atoms $\mathbb{A}$ and edges $\mathbb{E}$ and it cannot be transitively contained in itself.

Formally : $E_H \subseteq \mathbb{A} \cup \mathbb{E} \wedge E_H.incidenceSet \cap E_H^+ = \varnothing$

Definition 7

A struct E_{Struct} is a hyperedge that contains a set of atoms $\mathbb{A}$, relationships $\mathbb{E}_R$, and/or hyperedges $\mathbb{E}_H$ (a). Every struct has a special predefined root atom that enables to identify the struct noted as $O(E_{Struct})$. The root itself is a subset of the E_{Struct}s composition. All the As must have a unique path of relationships to its root which is also part of the struct (b). All the roots of the nested E_{Struct}s inside a parent E_{Struct} must have a unique path of relationships from the root of the parent, and this path must be inside the parent (c). All the E_{Set}s inside a parent E_{Struct} must contain a set of relationships that connects some A_C of the parent to the root of the child E_{Struct} or the A in the E_{Set} (d). All of the E_Rs inside a E_{Struct}, must be involved in a path that connects either the child As or the root of the child E_{Struct}s (e).

a) $E_{Struct} \subseteq \mathbb{A} \cup \mathbb{E}_R \cup \mathbb{E}_H$

b) $\forall a \in (self \cap \mathbb{A}) : \exists!\{E_R^{O(self),x_1}, \ldots, E_R^{x_n,a}\} \subseteq self$

c) $\forall s \in (self \cap \mathbb{E}_{Struct}) - O(self) : \exists!\{E_R^{O(self),x_1}, \ldots, E_R^{x_n,O(s)}\} \subseteq (self \cup \mathbb{A}_C)$

d) $\forall s \in (self \cap \mathbb{E}_{Set}), \forall t \in (s \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : \exists!\{E_R^{y,x_1}, \ldots, E_R^{x_n,z}\} \subseteq s \wedge y \in (self \cap \mathbb{A}) \wedge (t \in \mathbb{A}?z = t : z = O(t))$

e) $\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : (\exists y \in (self \cap \mathbb{A}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,y}\} \subseteq self) \vee (\exists y \in (self \cap \mathbb{E}_{Struct}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,O(y)}\} \subseteq self)$

Definition 8

A set E_{Set} is a hyperedge that contains a set of arbitrary E_{Struct}s, As, and specific relationships $\mathbb{E}_R$ (a). All the relationships in the set must originate from a class atom of the parent of the set and the destination should be the root of a child struct or A of the set (b).

a) $E_{Set} \subseteq \mathbb{E}_{Struct} \cup \mathbb{A} \cup \mathbb{E}_R$

b) $\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : \exists A_C^x \in self.parent, \exists y \in (self \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : E_R^{a,b} \in \{E_R^{x,x_1}, \ldots, E_R^{x_n,z}\} \subseteq self \wedge (y \in \mathbb{A}?z = y : z = O(y))$

4 Metadata Management

One crucial aspect of a metadata management system is the ability to represent different data store models. In our work, we exemplify it on traditional RDBMS, document stores, and wide-column stores. Figure 3.4 extends our original diagram of constructs to support those.
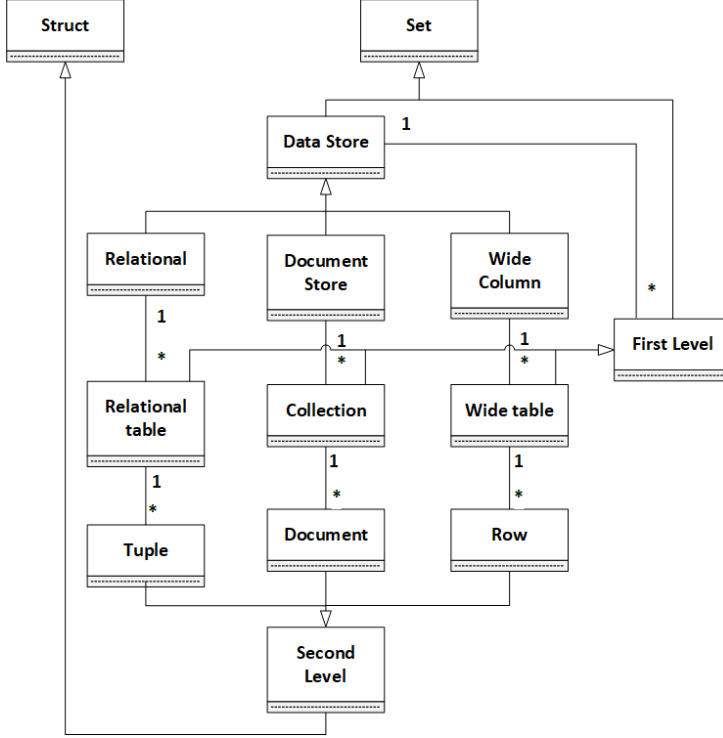
Fig. 3.4: Class diagram for *Hyperedge* hierarchy

The E_{Set} are specialized into two types: *Data Store* $\mathbb{E}_D$ and *First Level* $\mathbb{E}_F$. E_D represents a concrete data store of the polyglot system. E_F denotes a set of instances in the particular data store. All the allowed kinds of data stores are a subclass of $\mathbb{E}_D$. Moreover, $E_D.incidenceSet = \varnothing$. Thus, we define three kinds of E_D (namely *Relational* $\mathbb{E}_D^{Rel}$, *Document Store* $\mathbb{E}_D^{Doc}$, and *Wide − Column* $\mathbb{E}_D^{Col}$ in Figure 3.4), which are the participants of our polyglot system. There can be multiple E_F within each E_D adhering to the number of collections that participate in the polyglot system.

All the *Atom*s and *Edge*s of the polyglot system belong to the transitive closure of one or more of these E_D, $\mathbb{A} \cup \mathbb{E} = \bigcup E_D^+$. Therefore, we can deduce that the entire polyglot system catalog C can be represented by the

participating E_D. The *Second Level* (E_S), is a *Struct* that represents a kind of object residing directly in E_F. These E_S should align with the type of E_D where it is contained. It is a tuple for E_S^{Rel}, a document stored directly in the collection for E_S^{Doc}, and a row for E_S^{Col}. The specialized E_D, E_F, and E_S identify specific E_Hs in the data store constraints.

Each E_H carries a name which is interpreted depending on the context. In E_D, it represents the physical location of the underlying data store. In E_F, it is the collection name or table name. Depending on the type of E_D that represents the data store, we can identify specific constraints and transformation rules for the queries over the representatives.

The *Edge*s of the catalog can carry much more information than just the name. For example, an E_R can indicate the multiplicity between *Atom*s. Likewise, an E_H can carry information like the size of a collection, percentage of null values, maximum, minimum, and average of values. This catalog differs from a relational catalog in the expressiveness that allows to represent heterogeneous data models. By modeling the catalog of a polyglot system through a hypergraph, it is possible to retain the structural heterogeneity thanks to its high expressiveness and flexibility. Leveraging this information, it is interesting to see how we can retrieve the data from the polyglot system. Thus, our goal is to transform the formulation of a query over G into a query over the underlying data stores. Inter-data store data reconciliation or merges are out of the scope of this Chapter.

Algorithm 1 Query over polyglot system algorithm

Input: A query q
Output: A set of multi language queries Q corresponding to data store queries
 1: $Q \leftarrow \varnothing$
 2: $M \leftarrow newHashmap() < E_H, Set >$
 3: $Q \leftarrow newQueue()$
 4: **for each** *Atom* $a \in q$ **do**
 5:   **for** E_H $i \in a.incidenceSet$ **do**   ▷ hyperedges containing an *Atom*
 6:     $Q.enqueue(< i, a >)$
 7: **while** $Q \neq \varnothing$ **do**
 8:   $temp \leftarrow Q.dequeue$
 9:   $current \leftarrow temp.first$
10:   $M.addToSet(current, temp.second)$ ▷ adds the second parameter to the set
11:   **for each** E_H $j \in current.incidenceSet$ **do**
12:     $Q.enqueue(< j, current >)$
13: **for each** E_F $f \in M.keys$ **do**
14:   $Q.add(CreateQuery(f, ""))$
15: **return** Q

4.1 Query Representation

We assume that any query over the original RDF dataset or an equivalent query over the graph G corresponds to a query over the polyglot system. Hence, this query needs to be transformed into sub-queries that are executed on the relevant underlying data stores. For this, we introduce Algorithm 1 which builds an adjacency list for all the E_H (hash map M) whose closure contains *Atom*s of the query, aided by the incidence sets. First, all the E_Hs that contains *Atom*s in the input query q is added into a queue Q as a pair of $< E_H, A >$ (lines 4–8). Then, a pair *temp* is dequeued from Q and M is updated with the E_H in *temp.first* as the key and *temp.second* (can be an *Atom* or E_H) added to the corresponding value set with the help of *addToSet* in line 12. All the E_Hs in the *temp.first*'s incidence set is added to the queue Q (lines 13–15). This process is carried out until the queue Q has no more elements (lines 9–16). The generated adjacency list can be used to identify the E_Ds that corresponds to the relevant underlying data stores for the query. Once this adjacency list M is generated and corresponding E_D identified, a simple projection query can be composed recursively with Algorithm 2 for each of the E_F according to different rules depending on the kind of data store (lines 17–19). Algorithm 2 uses a prefix, a suffix and the path relevant for each of the constructs of the data stores.

Algorithm 2 Create Query algorithm

Input: *source* E_H, *path of* E_H (adjacency list M from Algorithm 1 is also available)
Output: A data store query q
1: $q \leftarrow prefixOf(source, path)$
2: **for each** *child* $\in M.get(source)$ **do**
3:    $q \leftarrow q + CreateQuery(child, pathOf(source))$
4: $q \leftarrow q + suffixOf(source)$
5: **Return** q

We are only generating projection queries, but selections can be considered a posteriori by pushing down the predicates over the query path. Also, there can be cases where the same information is available in multiple data stores. If this happens, this overlap can be easily identified as the considered *Atom* will be contained in more than one E_D^+. Then a join should be performed in the corresponding mediator.

Definition 9
A query q over the hypergraph G is a connected graph consisting of a selection atom *sel*, a set of projection atoms *proj*, a set of relationships *rel* and a frequency *freq*.

Formally : $q = \langle sel, proj, rel, freq \rangle | sel \in A, \forall a \in proj : a \in A, \forall A^x, A^y \in sel \cup proj : \exists \{E_R^{x,x_1}, E_R^{x_1,x_2}, \ldots, E_R^{x_n,y}\} \in rel, 0 < freq \leqslant 1$

4.2 Constraints and Transformation Rules on Data Stores

Considering the constructs and the query generation algorithm mentioned earlier, each of the data store models would have its own rules and constraints on the data. Therefore, in this section, we analyze the constraints and transformation rules for 3 of them: relational stores, document stores, and wide-column stores.

4.2.1 Relational Database Management Systems

A typical example of the type of data design in RDBMS is shown in Figure 3.5. The constraints and the mappings on E_H can be represented in a grammar as follows:

$$E_D^{Rel} \implies E_F^{Rel} *, \; E_F^{Rel} \implies E_S^{Rel}, \; E_S^{Rel} \implies A_C A *$$
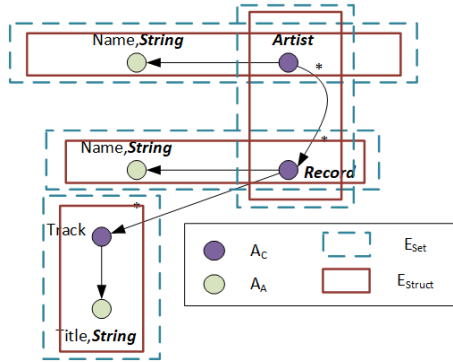
Fig. 3.5: An example data design for an RDBMS

A traditional RDBMS data storage system consists of tables, tuples, and simple attributes. The data store can have multiple tables, which are represented by E_F^{Rel}. Within a table, the schema of the tuple E_S^{Rel} is fixed. Therefore, there can only be a single E_S^{Rel} inside a E_F^{Rel}. Finally, the tuple contains at least one A_C, which is the primary key. The E_H containing an E_R that crosses two E_F^{Rel} corresponds to the relation that has the foreign key together with the relevant A_C.

The RDBMS design of Figure 3.5 represents the following tables:

$Artist[A\_id, name],$
$Record[R\_id, name],$
$Artist\_Record[A\_id(FK), R\_id(FK)],$
$Track[T\_id, title, R\_id(FK)].$

Symbol	prefix	suffix	path
E_F^{Rel}		"FROM"+ E_F^{Rel}.name	
E_S^{Rel}	"SELECT"	deleteComma()	
A	A.name	",”	

Table 3.1: Symbols for Algorithm 2 in RDBMS

Symbol	prefix	suffix	path
E_F^{Col}	"scan' " + E_F^{Col}.name + " ', {COLUMNS => ["		
E_S^{Col}		deleteComma() + "]}"	
E_{Struct}^{Col}			path + E_{Struct}^{Col}.name + "."
A	"'" + path + A.name + "'"	",”	

Table 3.2: Symbols for Algorithm 2 in Wide-Column Stores

The prefix and suffix in Table 3.1 are used in Algorithm 2 to generate the corresponding queries. Note that *deleteComma*() is an operation that deletes the trailing comma of a string.

4.2.2 Wide-Column Stores

In a wide-column store, the data is stored in vertical partitions. A key and fixed column families identify each piece of data. Inside a column family, there can be an arbitrary number of qualifiers which identify values.

$$E_D^{col} \implies E_F^{col} *, \; E_F^{col} \implies E_S^{col},$$
$$E_S^{col} \implies A_C E_{Struct}^{col} {}^+, \; E_{Struct}^{col} \implies A +$$
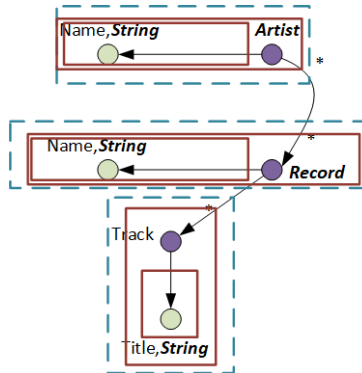
Fig. 3.6: An example data design for wide column store

The outer most E_F^{col} represents the tables. E_F^{col} contains an E_S^{col}, which represents the rows. The A_C inside this E_S^{col} becomes the row key. E_S^{col} contains several E_{Struct}^{col}, which represent the column families. The A inside the

E_{Struct}^{col} represents the different qualifiers. The relationships between A_C can be represented as reverse lookups. In our example scenario, the hypergraph in Figure 3.6 contains one column family per table. This can be mapped into

Artist[*A_id*, [*name*, {*R_id*}]],
Record[*R_id*, [*name*]],
Track[*T_id*[*title*, *R_id*]].

Wide-column stores generally support only simple get and put queries and require the row key to retrieve the data. We use HBase query structure to demonstrate the capability of simple query generation. Table 3.2 depicts the translation rules for simple queries in wide-column stores used in Algorithm 2.

4.2.3 Document Stores

Document stores have the least constraints when it comes to the data design. They enable multiple levels of nested documents and collections within. Figure 3.7 shows a document data store design of our example scenario. The
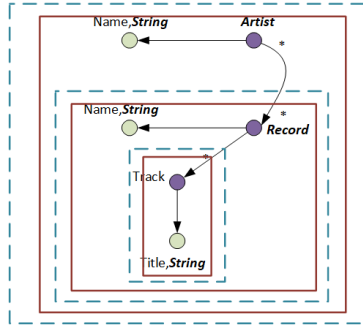
Fig. 3.7: An example data design for Document Store

constraints and mappings in a document store design are as follows:

$$E_D^{Doc} \implies E_F^{Doc} *, \ E_F^{Doc} \implies E_S^{Doc} +,$$
$$E_S^{Doc} \implies A_C(A|E_{Set}^{Doc}|E_{Struct}^{Doc})^*, E_{Set}^{Doc} \implies (A|E_{Struct}^{Doc})^+,$$
$$E_{Struct}^{Doc} \implies (A|E_{Set}^{Doc}|E_{Struct}^{Doc})^+$$

E_F^{Doc} represents the collections of the document store. E_S^{Doc} inside the E_F^{Doc} represents the documents within the collection, which must have an identifier A_C. Apart from that, E_S^{Doc} can have *Atom*s, or E_{Set}^{Doc}, which represents nested collections, or E_{Struct}^{Doc}, or a combination of any of them. E_{Set}^{Doc} represents a nested collection which contains documents E_{Struct}^{Doc}.

The design in Figure 3.7 can be mapped into a document store design as
Artist{*A_id* :, *name* :, *Records* : [{*R_id* :, *name* :, *Tracks* : [{*T_id* :, *title* :}]}]}

49

We use MongoDB syntax for the queries as it is one of the most popular document stores at the moment. Table 3.3 identifies the symbols for the queries.

Symbol	prefix	suffix	path
E_F^{Doc}	$\text{“db.”} + E_F^{Doc}.name + \text{“}.find(\{\}\text{”},\{$	$\text{”})\text{”}$	
E_S^{Doc}		$deleteComma()$	
$E_{Struct/Set}^{Doc}$			$path + E_{Struct/Set}^{Doc}.name + \text{“.”}$
$A(path \neq \varnothing)$	$\text{“”“”} + path + A.name + \text{“”: 1”}$	“,”	
$A(path = \varnothing)$	$A.name + \text{“ : 1”}$	“,”	

Table 3.3: Symbols for Algorithm 2 in Document Stores

4.2.4 Other Data Stores

As discussed above, we have managed to model and infer constraints and transformation rules for RDBMS, wide-column stores, and document stores which cover most of the use cases. However, it is also interesting to see the capability of the approach to represent other data stores. Since our model is based on graphs, we can simply conclude that it can express graph data stores. We only need to map the data into G, and define a single set with all *Atom*s. The key-value stores do not have sophisticated data structures, and it can be considered as a single column in a column family. Thus, since we are disregarding the storage of complex structures in the values that are not visible to the data store, we can state that our model covers key-value stores as well.

5 Calculating Statistical and Storage Metadata

We extend our metadata representation by including statistics on the *Hyperedge*s. For this, we consider the data type and the number of distinct values on the A_As, count on A_Cs, and the multiplicity on each end of the E_Rs. E_{Set} will also have the cardinality as a calculated value (see Fig. 3.2). Using the count and the multiplicity, we can calculate the total storage size for each of the storage structures (relational table, collection, or wide table). In document stores, if there are nested attributes within a collection (i.e., Records or Tracks inside an Artist in Fig. 3.7) it is important to calculate the multiplicity between the $O(E_{S/Struct})$ and the $O(E_{Struct})$ of the nested E_{Struct}. This information can be used to estimate the size of secondary indexes, especially with multiple nesting levels. Moreover, the same information can be used to determine the physical access patterns of a particular query over a certain datastore. Even though these calculations can be carried out on any datastore, it is particularly interesting in document stores, because they are more flexible on storing data

compared to RDBMS and column stores, mainly due to nesting. Thus, we focus on document stores to explain the algorithms.

5.1 Storage size estimation

We take our running example in Fig. 3.3 to illustrate how Algorithms 3 and 4 can be used to calculate the storage size and the multipliers between the *Root* atom and the rest of the Atoms in an E_{Struct}. The multiplicity between *Artist* and *Record* is 5 and between *Record* and *Track* is 15, and there are 20, 100, 1500 instances of *Artist*s, *Record*s, and *Track*s, respectively. Let's assume all A_Cs are integers of 4 bytes in size, and all the A_A Strings are 10 bytes.

Algorithm 3 works in a recursive manner going through each of the E_Hs, calculating the size of each of the *nodes* inside, and multiplying them by the number of instances of the E_H. In the case of an *Atom*, this will be the size of the stored data (i.e., 4 bytes for an integer). In case of a named E_{Struct} or E_{Set}, it will add the name length to the total size, and its content. E_{Set} represents a collection of elements that could be a list or an array. In this case, we need to calculate the size of this list/array. Algorithm 4 finds the number of instances of such complex objects by referring to the multiplicity of the relationship between parent and the child E_Hs.

If we want to calculate the storage size and the multipliers of the atoms of the schema in Fig. 3.7, the collection hyperedge will be the input to Algorithm 3. In the first iteration, the size will be set to 0 (Line 6). Then, at line 16, the top level document will be found, and the size will be increased by the length of the name of the collection (6 assuming it is "Artist"). When the embedded *Records* are reached, the E_{Set}^{Doc} will have size 7 (with the name "Records"). Then the *Records* will look into the embedded *Tracks*. The *Tracks* will have a size of 7 with the name, and then the T_ID and the T_NAME will be 4 and 10 bytes respectively, making an individual track 14 bytes overall. Moreover, the E_{Set}^{Doc} that contains the relationship between *Record* and *Track* will return the multiplier being 15. Therefore, the record size will be calculated as the sum of 15 tracks, the length of the text "Tracks", R_ID and the R_NAME adding up to 230 ($15 * 14 + 6 + 4 + 10$) bytes. The E_{Set}^{Doc} that contains the relationship between *Artist* and *Record* will return the multiplier 5. Hence, the *Records* inside the Artists will have a size of 1, 150 ($230 * 5$) bytes. The A_ID, A_NAME, and *Records* inside the *Artist* document will get a multiplier of 20. Since there are 20 *Artist*s, the total size of the collection will increase to 23, 426 ($6 + 20 * (4 + 10 + 7 + 1, 150)$) bytes.

Apart from the total size, Algorithm 3 also returns the number of instances of each of the *Atom*s within that collection. From the previous example, we get 20, 100, 1500 as the number of instances for A_ID, R_ID, and T_ID respectively. By dividing the number of instances by the number of instances of *Root*, we can obtain the multiplying factor between the root and the nested

---

Algorithm 3 CalculateSize algorithm

Input: *source* $\in A \cup E$
Output: Size *s*, Hashmap <A,multiplier> *map*
 1: $map \leftarrow newHashmap() < A, multiplier >$
 2: **if** *source* $\in A$ **then**
 3:    $s \leftarrow source.size$
 4:    $map.put(source, 1)$
 5: **else if** *source* $\in E_F^{Doc} || source \in E_S^{Doc}$ **then**
 6:    $s \leftarrow 0$
 7: **else if** [**then**Embeded list]*source* $\in E_{Set}$
 8:    $s \leftarrow source.name.length()$
 9: **else if** *source* $\in E_{Struct}$ **then**
10:    **if** *source.name* $= \varnothing$ **then**   ▷ Struct inside a set
11:      $s \leftarrow 0$
12:    **else**   ▷ Embedded struct
13:      $s \leftarrow source.name.length()$
14: **for each** *child* $\in source.getChildren()$ **do**
15:    $multiplier \leftarrow CalculateMultiplier(source, child)$
16:    $result \leftarrow CalculateSize(child)$
17:    $s \leftarrow s + result.s * multiplier$
18:    **for each** *key* $\in result.map.keys()$ **do**
19:      $map.add(key, result.map.get(key) * multiplier)$
20: **Return** $< s, map >$

---

*Atom*s. This value together with the distinct values of an A_A can be used to identify the potential secondary index sizes and their effectiveness. For example, in an alternative document store design where the tracks are the first level documents that embed the records and the record embed the artists, with the same calculations we would get 1500 as the number of *Author*s stored. However, since we know that there are only 20 different *Author*s, if we build a secondary index on *A_ID*, we can estimate that each *A_ID* index entry would point to $\frac{1500}{20} = 75$ top level documents (*Tracks*).

Algorithm 4 CalculateMultiplier algorithm

Input: *source* $\in E_H^{doc}$, *child* $\in M$
Output: *Multiplier m*
 1: **if** [**then**top level collection]*source* $\in E_S^{doc}$
 2:    $m \leftarrow source.root.count$
 3: **else if** *source* $\in E_{Set}^{doc}$ **then**
 4:    $relationship \leftarrow source.findRelationship(*, child.root)$ ▷ Set has one parent
 5:    $m \leftarrow relationship.Multiplicity$
 6: **else**
 7:    $m \leftarrow 1$
 8: **Return** *m*

5.2 Physical access patterns for workloads

A query workload is usually provided as the frequencies of the respective queries. However, the access patterns and the runtimes of the queries depend on the underlying schema design, as shown in works such as [12] and [10]. Therefore, we believe that it is of interest to determine how the underlying physical storage structures are used for a particular workload on different data stores and schema designs. We assume that the workload is given as a set of queries $\mathbb{Q}$ as stated in Definition 9, together with their access frequency. Thus, we use Algorithm 5 to calculate the access frequencies of the collections and their secondary indexes depending on the design choices.

Algorithm 5 CalculateFrequency algorithm

Input: $\mathbb{Q}$, $\mathbb{E}_S^{Doc}$
Output: Hashmap$<E_S^{Doc} \cup A_C, freq> m$
1: $m \leftarrow newHashmap() < E_S^{Doc} \cup A_C, freq >$
2: $winners \leftarrow newQueue()$
3: **for each** $query \in \mathbb{Q}$ **do**
4:   $remaining \leftarrow query.proj$
5:   $cands \leftarrow E_S^{Doc}.find(contains(query.sel))$
6:   **if** candidates.size = 1 **then**
7:     $winner \leftarrow cands[0]$
8:   **else**   ▷ winner is the smallest collection with the selection as root
9:     $winner \leftarrow cands.find(candidate.root = query.sel \wedge candidate.size = findMinSize(cands))$
10:   **if** [**then**no selection as the root]$winner = \varnothing$
11:     $remaining.add(query.sel)$
12:     $winner \leftarrow cands.find(candidate.size = findMinSize(cands.size))$
13:   $m.Aupdate(< winner,$
  $query.freq \cdot winner.getMultiplier(query.sel) >)$
14:   $m.Aupdate(< winner.get(query.sel), query.freq >)$
15:   $winners.enqueue(winner)$
16:   **while** $winners \neq \varnothing \wedge remaining = \varnothing$ **do**
17:     $main \leftarrow winners.dequeue()$
18:     $covered \leftarrow main^+.Atoms \cup remaining$
19:     $remaining \leftarrow remaining - main^+.Atoms$
20:     **for each** $A_c \in covered$ **do**
21:       **for each** $A \in remaining$ **do**
22:       $rel \leftarrow query.rel.findRelationship(A_c, A)$
23:       **if** $rel \neq \varnothing$ **then**   ▷ found a join
24:       $join \leftarrow G.findCollection(contains(A, A_c))$
25:       $m.Aupdate(< join, m.get(main)\cdot$
  $rel.Multiplicity \cdot join.getMultiplier(A_c) >)$
26:       $m.Aupdate(< join.get(A_c),$
  $m.get(main) \cdot rel.Multiplicity >)$
27:       $winners.enqueue(join)$
28:   $SumToUnity(m)$
29: **Return** m

We use a simple query structure with a single selection predicate and

multiple projections. Nevertheless, notice that multiple tables/collections within the data store may still be capable of answering the same query. In such cases, we use a greedy approach for selecting the winner as the smallest sized collection out of the candidates. Thus, the base case scenario is a single collection containing the selection predicate (Line 6), followed by the smallest collection containing the selection *as the root* if many contain it (Line 8), and finally, the smallest collection that contains the selection predicate if none of them have it has the *root* (Line 11).

For example, let us assume that we want to retrieve *R_ID*s and *T_ID*s by *A_ID* with a frequency of 1, the records embed the artists, and the *Tracks* have a separate collection with *R_ID* as the reference. We will have the *Record* as the winning collection. If there is a secondary index on a particular *A*, each query will use that index once, and the top level collection will be accessed *multiplier* times. Therefore, the frequency of the collection is higher than the actual query (Lines 15 and 16). Thus, we multiply the query frequency by the multiplier of the selection predicate as the current frequency of the collection ($1 * 5$ for *Record*) and the selection predicate *A_ID*, is accessed with the frequency of the query (1). The *getMultiplier(A)* method will return the number of instances calculated in Algorithm 3 divided by the number of root instances.

All the *A*s transitively contained in the document that belongs to the query are already retrieved by accessing it (*R_ID*). However, if there are other *A*s in different collections, they need to be joined outside of the data store (*T_ID*). Assuming that we use a row-nested loop join, the index of the joined collection needs to be accessed *multiplicity* times with regard to the original collection, as shown in line 28, and the collection is accessed according to the referred index as in line 27. Hence, the index on *R_ID* is accessed 5 times, and the collection is accessed 75 ($5 * 15$) times. The algorithm continues until all the *A*s in the query are covered by the data store. Finally, in line 33, we use a normalizing factor to sum to unity, giving us the relative access frequencies of different storage structures in the overall data store for this particular workload. In our example, the four physical structures of *Record* and *Track* collections and the indexes on *A_ID* in *Record* and *R_ID* on *Track* will be accessed with overall frequency of 0.059 ($\frac{5}{5+1+5+75}$), 0.87 ($\frac{375}{5+1+5+75}$), 0.012 ($\frac{1}{5+1+5+75}$), and 0.059 ($\frac{5}{5+1+5+75}$), respectively.

The Algorithms 3, 4, and 5 enable us to identify the storage requirements of data and physical access patterns of the queries. Thus, we can determine the storage sizes and the individual access frequencies of the collections and their indexes (both primary and secondary). Applying these frequencies and the storage sizes will allow us to estimate the query performance for different datastore designs. We have implemented the approach in Appendices A and B, to evaluate alternative schema designs on document stores using the

cost model in Chapter 4 on storage space and query performance.

6 Use Case
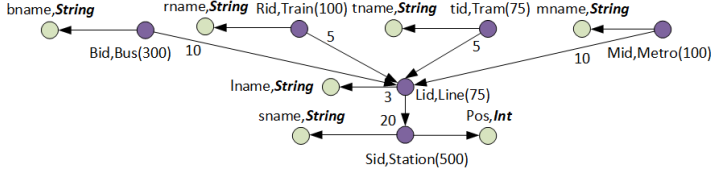
Fig. 3.8: Graph representation of ESTOCADA

In this section, we showcase our technique applied on an already available polyglot system. We base the example on the scenario used for ESTOCADA [21]. This involves a typical transportation data storage for a digital city open data warehousing. It uses RDBMS, document stores, and key-value stores. Figure 3.8 shows the corresponding graph G. The multiplicities are shown with the corresponding arrows (e.g. each line goes through 20 stations). We have omitted the multiplicity of each bus, train, tram, and metro being associated with one line for clarity of the figure.

The ESTOCADA system is used to store train, tram, and metro information in an RDBMS, the train and metro route information in a document store, and bus route together with the buses information in a key-value store (see [21] for more details). This information can be represented in our polyglot catalog[2] as follows (shown as containment sets):

$C = \{E_D^{Rel}, E_D^{Kv}, E_D^{Doc}\}$

$E_D^{Rel} = \{E_{F\_Train}^{Rel}, E_{F\_Merto}^{Rel}, E_{F\_Tstat}^{Rel}, E_{F\_Mstat}^{Rel}, E_{F\_Station}^{Rel}\}$,

$E_{F\_Train}^{Rel} = \{E_{S\_Train}^{Rel}\}$, $E_{S\_Train}^{Rel} = \{A_{C\_rid}, A_{A\_rname}\}$,

$E_{F\_Metro}^{Rel} = \{E_{S\_Metro}^{Rel}\}$, $E_{S\_Metro}^{Rel} = \{A_{C\_mid}, A_{A\_mname}\}$,

$E_{F\_Tstat}^{Rel} = \{E_{S\_Tstat}^{Rel}\}$, $E_{S\_Tstat}^{Rel} = \{A_{C\_rid}, A_{C\_sid}, A_{A\_pos}\}$,

$E_{F\_Mstat}^{Rel} = \{E_{S\_Mstat}^{Rel}\}$, $E_{S\_Mstat}^{Rel} = \{A_{C\_mid}, A_{C\_sid}, A_{A\_pos}\}$,

$E_{F\_Station}^{Rel} = \{E_{S\_Station}^{Rel}\}$, $E_{S\_Station}^{Rel} = \{A_{C\_sid}, A_{A\_sname}\}$,

$E_D^{Doc} = \{E_{F\_Metros.Trams}\}$,

$E_{F\_Metros.Trams}^{Doc} = \{E_{S\_Metros.Trams}^{Doc}\}$,

$E_{S\_Metros.Trams}^{Doc} = \{A_{C\_lid}, A_{A\_lname}, E_{Set\_route}^{Doc}\}$,

$E_{Set\_route}^{Doc} = \{E_{Struct\_Station}^{Doc}\}$, $E_{Struct\_Station}^{Doc} = \{A_{C\_sid}, A_{A\_sname}\}$,

---

[2]The implementation of the catalog is available in https://git.io/vxyHO

$$E_D^{Kv} = \{E_{F\_Station}^{Kv}, E_{F\_Bus}^{Kv}\}, \; E_{F\_Station}^{Kv} = \{E_{S\_Route}^{Kv}\},$$
$$E_{S\_Route}^{Kv} = \{A_{A\_lname}, E_{Set\_loc}^{Kv}\},$$
$$E_{Set\_loc}^{Kv} = \{A_{A\_sname}\}, \; E_{F\_Bus}^{Kv} = \{E_{S\_Bus}^{Kv}\},$$
$$E_{S\_Bus}^{Kv} = \{A_{C\_bid}, A_{A\_lname}\}$$

Our goal was to store the metadata of the ESTOCADA polyglot system with a hypergraph. Thus, we used HyperGraphDB[3] to save the entire catalog information including the *Atom*s, *Relationship*s, and *Hyperedge*s for the structures. With this catalog, one can quickly detect where each fragment of the polyglot system lies by merely referring to *Hyperedge*s and the content within.

Let us assume that the following queries are issued on the catalog with equal probability.

- **Q1:** Find information about trains on a given station (`sid, sname, rid, rname`).

- **Q2:** Find the metro lines on a given station (`sid, sname, lid, lname`)

By utilizing Algorithms 1 and 2, we can generate the following data store specific queries (selections are added a priori).

- ```
  SELECT rid, rname FROM Train
  SELECT sid, sname FROM Station WHERE sid = <>
  SELECT sid, rid FROM Tstat
  db.MetrosTrams.find({route.sid:<>},{route.sid:1,
  route.sname:1})
  ```

- ```
  db.MetrosTrams.find({route.sid:<>},{lid:1, lname:1,
  route.sid:1, route.sname:1})
  SELECT sid, sname FROM Station WHERE sid = <>
  ```

For the ease of demonstration, let us assume that all the identifiers are 4 bytes and the names are 20 bytes. Then, using the multiplicities provided in *G* (Fig. 3.8) we can calculate the storage sizes and the multipliers using Algorithms 3 and 4. Thus, we get a storage size of 54,150 bytes for a document store. Each nested station document is 34 bytes (20+4+4+6). Then, the route having list of 20 stations becomes 688 bytes (20*34+8). Next, the line contains the route and the names with 722 bytes (688+20+4+4+6). Finally, the total size is obtained by multiplying the line size by 75 (number of lines). This algorithm can be reused to calculate sizes of other data stores giving us 58,200 bytes in total on a RDBMS (75*(4+20) for Train, 100*(4+20) for Metro, 75*1*20*(4+4+4) for Tstat, 100*1*20*(4+4+4) for Mstat, and 500*(4+20) for Station) and 19,000

---

[3]http://www.hypergraphdb.org

bytes on the Key-value store 30*(20+(20*20)) for Route and 300*(4+20) for Bus). When it comes to multipliers, only the document store and the RDBMS tables with foreign keys should be considered. Thus, *sid* has a multiplier of 15 on *Tstat* (3*5) and 30 on *Mstat* (3*10) and 3 on document store collection due to each station belonging to 3 lines and each line having 5 trains and 10 metros. Finally, using Algorithm 5, we can calculate the access frequencies of *sid* and *MetrosTrams* collection (assuming both Q1 and Q2) as 0.25 and 0.75 as shown in Table 3.4. Similarly, the RDBMS tables Station, Tstat, and Train will be accessed with frequencies of 0.063, 0.4687, and 0.4687 respectively.

	Usage	
Query	*sid*	*MetrosTrams*
Q1 (p=0.5)	0.5	0.5*3 = 1.5
Q2 (p=0.5)	0.5	0.5*3 = 1.5
Total	1	3
Frequency	$\frac{1}{21}$=0.25	$\frac{20}{21}$=0.75

Table 3.4: Access frequencies of the document store storage structures

7 Related Work

There are few polyglot systems already available to support heterogeneous NoSQL systems. In BigDAWG [38], different data models, including relation, array, graph, stream, and text, are classified as islands. Each of the islands has a language to access its data, and the data stores provide a shim to the respective islands it supports for a given query. Cross-island queries are also allowed, provided appropriate query planning and workload monitoring. Contrastingly, ESTOCADA [20, 21] enables the end user to pose queries using the native format of the dataset. In this case, the storage manager fragments and stores the data in different underlying data stores by analyzing the access patterns. These fragments may overlap, but the query executor decides the optimal storage to be accessed. ODBAPI [102] introduces a unified data model and a general access API for NoSQL and heterogeneous NoSQL systems. This approach supports simple CRUD operations over the underlying systems, as long as they provide an interface adhering to the global schema.

Myria [115] is a federated data analytics system that allows expressing complex data analytic processes using its own hybrid language MyriaL. The Relational Algebra Compiler(RACO) is used as the federated query executor, which uses relational algebra extended with imperative constructs to capture the semantics of non-relational concepts such as arrays. It generates query plans for a specific array, graph, and key value engines. Apache Drill [55] is a distributed query engine for ad-hoc analysis. It supports file-based, document, relational, and columnar based storage systems. It used an in-memory columnar data representation based on JSON and Parquet, which

allows flexible schema management. CloudMdsQL [73] is a scalable SQL query engine with extended capabilities to query non-relational data stores. It uses a SQL based query language with embedded subqueries native to the underlying data stores. A comprehensive analysis of BigDawg, Myria, Apache Drill, and CloudMdsQL is carried out in [108] comparing the different systems in terms of heterogeneity, transparency, optimally, flexibility, and autonomy. The work concludes by stating that none of the systems nor approaches are better than the other. Different systems are performant on different aspects according to the design trade-offs that they have made.

SQL++ [91] introduces a unifying query interface for NoSQL systems as an extension of SQL to support complex constructs such as maps, arrays, and collections. This is used as the query language for the FORWARD middleware that unifies structured and non-structured data sources. A similar approach to SQL++ is carried out in [113]. Katpathikotakis et al. [67] introduces a monoid comprehension calculus-based approach which supports different data collections and arbitrary nestings of them. Monoid calculus allows transformations across data models and optimizable algebra. This enables the translation of queries into nested relational algebra that can be executed in different data stores through native queries.

Using different adapters or drivers for heterogeneous data stores is a common approach used in polyglot systems. The systems, as mentioned earlier, use the same principle. Liao et al. [76] use an adapter-based approach for RDBMS and HBase. The authors introduce a SQL interface to RDBMS and NoSQL system, a DB converter that transforms the information with table synchronization, and a three-mode query approach that provides different policies on how applications access the data. The Spring framework [64] is one of the most popular software used to access multiple data stores, as it supports different types by using specific drivers and a common access interface. Apache Gremlin [96] and Tinkerpop[4] follow a similar approach but particularly for graph data stores. The main drawback of this approach is that each and every implementation needs to adhere to a common interface, which is difficult due to the vast number of available data stores.

Standalone data stores have their own metadata catalogs. For example, HBase uses HCatalog[5] (for hive) to maintain the metadata. They are strictly limited to the respective data models involved. In our work, we introduce a catalog to handle heterogeneous data models. MongoDB, on the contrary, does not maintain any metadata by default but instead handles the documents themselves but, there is a built in schema and data type validation[6]. Some work also has been carried out in managing document store schema externally [116].

---

[4]http://tinkerpop.apache.org
[5]https://cwiki.apache.org/confluence/display/Hive/HCatalog
[6]https://docs.mongodb.com/manual/core/schema-validation

Several works have been carried out on data design methodologies for NoSQL systems. NoSQL abstract model (NoAM) [10, 22] is designed to support scalability, performance, and consistency using concepts of collections, blocks, and entries. This model organizes the application data in aggregates. It defines the four main activities: conceptual modeling, aggregate design, aggregate partitioning, and implementation. The aggregate storage in the target systems is done depending on the data access patterns, scalability, and consistency needs. Our metamodel is capable of representing the same data on different data stores having different schemas and their overlaps (essential for the catalog) which NoAM cannot directly represent. Moreover, NoAM does not discuss multiple levels of nesting which is essential in representing JSON. Similarly, a general approach for designing a NoSQL system for analytical workloads has been presented in [58]. It adapts the traditional 3-phase design methodology of conceptual, logical, and physical design, and integrates the relational and co-relational models into a single quantitative method. At the conceptual level, the traditional ER diagram is used and transformed into an undirected graph. Nodes denote the entities, and the edges represent their relationships, tagged with the relationship type (specialization, composition, and association). In cases where an entity can become a part of several different hyper nodes, it is replicated in each of them. Mortadelo [34] introduces a model-driven database design process to automatically generate a concrete NoSQL database system from a high-level conceptual model. The platform-independent Generic Data Metamodel (GDM) is used to represent not only structural data but also the data access patterns. Then, applying a set of transformation rules, the logical NoSQL specification is generated for specific data store models (column family and document stores). Next, a set of implementation scripts are generated for the target technology of the data store. Mortadelo shows improvements over state of the art by comparison on different use cases in document and column stores.

The Concept and Object Modeling Notation (COMN) introduced in [59] covers the full spectrum of not only the datastore but also the software design process. COMN is a graphical notation capable of representing the conceptual, logical, physical, and real-world design of an object. This helps to model the data in NoSQL systems where the traditional ER diagrams fail in representing certain situations, such as nesting.

Chapter 4

A cost model for random access queries in document stores

1   Introduction

In the last couple of decades, NoSQL systems were introduced as an alternative storage mechanism to traditional Relational Database Management Systems (RDBMS). As of today, there are more than 100 NoSQL implementations, categorized into four main types, namely, key-value stores, document stores, column stores, and graph stores [24]. Among these, document stores have been one of the most popular, mainly because of the schema flexibility they provide.

Traditional RDBMS arrange data in tables with a fixed schema, and each tuple within the table adheres to it. Similarly, document stores arrange data in collections, and as the name suggests, they use document formats such as XML or JSON as the unit of storage. However, in contrast to RDBMS tables, the schema of a collection is not fixed and allows semi-structured data. This means the documents within a collection need not be homogeneous, and in extreme cases, one might have a collection containing documents that are entirely different from one another.

Motivated by various commercial needs, the industry drives most of the document store development. The semi-structured storage nature of document stores enables end-users to follow a data-first approach rather than a schema-first one. This can reduce the time and effort of defining the schema, especially for cases where there is some uncertainty about the data structure and content. However, in contrast to RDBMS, there is no formal standard, such as SQL and normalization theory for document stores. Instead of a formal data and query design, existing document stores provide guidelines for implementation-specific optimizations [60]. Furthermore, all document stores use primitive approaches to determine the optimal query execution plan. For example, MongoDB tries to execute all viable query plans that use indexes in parallel to finally choose the winning option. Here, rather than performing all the query plans, some plans that take more work units (execution time or the number of documents examined) than the currently leading one are discarded prematurely as an optimization. The winning plan is cached and used on similar subsequent queries. [1] Therefore, it is crucial to have a better schema design to support this query execution model.

There are no existing methodologies to evaluate the viable document store schema designs (referred to as design from now on), but only through expensive trial and error one could determine which is the best design. Here, a formal cost model would allow end-users to estimate how different design decisions affect performance rather than relying on their intuition. Relational cost models [61, 77, 94] are per-query and based on the current state of the DBMS at the time of query execution. The primitive approaches of

---

[1]https://docs.mongodb.com/manual/core/query-plans

document store query planners do not even rely on such information. Existing cost models in RDBMS are based on disk access as the number of disk accesses will always dominate the execution time of a query. Moreover, as a result of hardware cost reduction in recent times, most of the data is pushed into the memory for faster performance, and the cost models have adjusted accordingly [83]. Thus, most document stores encourage having the working dataset in memory to achieve higher throughput and lower response time. Deciding on which fraction of the data should be in memory depends on access patterns and memory allocation policies. Nevertheless, both of these systems need to persist data into the disk (there are in-memory systems that are out of the scope of our paper), and in most of the cases during the actual operations, most of the data does not fit into the available memory. Therefore, it is highly probable that the data is accessed from both memory and disk simultaneously. Hence, in our work, *we aim to predict the behaviour of the document store memory and its effect on query execution depending on data design decisions*.

In this chapter, we present a generic cost model for random access in document stores based on storage metadata and memory usage. Both these parameters are specific for different document store implementation decisions. Metadata can be easily obtained from the data stores and depends on the disk storage structures. However, memory usage depends on memory mapping, associativity, and eviction policies, and each of them consists of several possibilities (e.g., pre-determined or shared associativity). Therefore, we introduce formulas for different possibilities and depending on the underlying document store, we pick the corresponding formulas to estimate the memory usage. Finally, we use these memory estimates in our generic cost model. We use Couchbase Server and MongoDB as two exemplars with different memory usage patterns to validate our cost model. Couchbase Server has pre-determined memory associativity per collection and maps individual documents into memory, whereas MongoDB has a single memory shared among the collections governed by a Least Recently Used (LRU) cache eviction policy and maps disk blocks to memory.

The objective of our work is not to predict an exact runtime, but rather to obtain a relative cost for a specific query under a fixed memory, workload, and varying design decisions (e.g., average document size, number of documents, and access frequency). Thus, the main contribution of this work is *a generic cost model for document stores for random access queries, which includes a detailed memory distribution estimation model for different memory management choices*. Based on experiments, we show that we are able to estimate memory distribution within an average precision of 91% and successfully predict the relative cost of queries concerning the most relevant parameters. To the best of our knowledge, this is the first attempt at a cost model, allowing us to predict the design impact in document stores. Even though it is possible to introduce this cost model

into the native query processing of the document store implementation, it may introduce significant overhead hindering the performance. The simplicity of the current primitive query processing approaches is one of the many reasons behind the performance gains in document store or NoSQL systems in general compared to traditional RDBMS. Thus, we instead intend to use this cost model as a tool to enhance the schema design process, which is mostly rule-based. Using our approach, we can calculate a relative cost for a particular design under a predefined workload. This value can then be used as a measure of optimality together with other contradicting requirements such as storage space of a collection on each of the viable designs to select the best one. Thus, this is an initial step towards optimizing resource usage using a systematic data design.

Document stores attract users who engage in rapidly changing requirements taking away the burden of fixing a database schema. Following this trend, the SQL:2016 standard has incorporated JSON into the specification allowing RDBMS to have unstructured data in a single attribute [87]. Therefore, data design for RDBMS also goes beyond strict normalization and might require a cost-based schema design and could benefit from our cost model as well. We were also able to identify several inconsistencies of existing document store implementations and propose how to improve them. Therefore, this work can also be used to *identify discordance of document store implementation behavior against a theoretical expectation*.

2   Background and Related Work

Most database system has its own cost model to determine query costs and make execution plans accordingly. RDBMS have been around for more than three decades and have well-established cost models and query planning capabilities [61, 77, 94]. These cost models mainly depend on the disk I/O as it is the most costly resource compared to other factors such as CPU calculations and memory access. The main considerations of these cost models are physical data structures, access paths, and the algorithms used.

Several works have been carried out on optimizations and cost models for XML databases [50, 69] as well as for object-oriented databases [16, 46]. Moreover, Manegold et al. proposed a generic technique to create cost functions for database operations in hierarchical memory systems in [83]. This approach claims to be extensible to include disk I/O. As this hierarchical memory model approach is most comparable to ours, we use it as a baseline for comparison with details in Sect. 5.4.

Contrary to classical RDBMS systems, in-memory databases do not depend on accessing data from the disk. Consequently, the cost models depend exclusively on CPU cycles, memory capacity, line size, and associativity [44].

As mentioned before, document stores utilize both disk and memory for optimized performance. Between the two, the disk access cost is several magnitudes higher than of the memory. Thus, the determining factor for the performance of a query will always be disk I/O when the data does not fit in memory (most of the cases). Hence, we base our cost model on the RDBMS approach due to the similarity in storage structures used in the observed document stores and the maturity of the approach. Furthermore, our cost model incorporates memory distribution in the calculation as needed for overall design optimization.

Data in a typical RDBMS is stored with a primary index that utilizes B-tree as the data structure when storing data in the disk. The internal nodes of the B-tree contain the indexed values, and the leaf nodes contain either data or pointers to data depending on the type of index (clustered and non-clustered). These nodes are stored as fixed-size blocks in the disk. The total size of a table depends on the record size, the number of records, and the indexing mechanism used. The data access paths can be identified as a table scan, random access, searching for one or several tuples, insertion, and deletion of a tuple.

A given query can have multiple execution plans subject to the access path to the tuples, index structures, and the order and execution algorithm for join operations. RDBMS use a cost-based estimation where alternative query plans are generated, and intermediate result sizes and cardinalities are estimated from the available statistics. Next, a cost is estimated for each plan based on blocks read and written to choose the best one. For example, in PostgreSQL, each of the operations has predefined cost value[2] and the alternatives are evaluated through genetic optimization.[3]

Caching is an essential concept in modern disk-based computer systems that refers to keeping already used data in memory so that the information can be served faster for future requests. The retained data originates from a prior request or calculation. When the data is fetched from the cache, it is considered a cache hit or, in the opposite case, a cache miss. In the case of a cache miss, the data needs to be fetched from the disk, which increases the latency. Thus, higher cache hit rates lead to better performance [104]. However, the cache is generally smaller than the information that needs to be retrieved by an application. Therefore, cached data needs to be replaced over time. There are several eviction policies such as first-in-first-out, last-in-first-out, least recently used, etc [85]. Different cache policies have their strengths and weaknesses, and they are used depending on the application requirements [105]. In any case, it is essential to know the hit rate to analyze the performance of end-user applications. However, this is not trivial due

---

[2]https://www.postgresql.org/docs/12/static/runtime-config-query.html
[3]https://www.postgresql.org/docs/12/geqo-pg-intro.html

to the complex nature of the cache, the replacement policies, and the access patterns. Frank King III introduced a Markov-chain-based cache hit rate approximation using the independent reference model [70]. Based on this approach, several other works have been conducted on approximating the cache miss and hit ratios for different cache policies [53, 63, 114]. These approaches provide an approximation of hit rate and cache usage not only for a single application but also for shared cache among several applications [40]. In our work, we followed a probability-based approach to have a simple model with lower runtime complexity.

The structure of the data also plays a vital role when implementing a cost model. Although document stores and RDBMS maintain structured data, they differ in several ways. Data in RDBMS is stored in and conforms to a user-defined structure (table), and a single real-world object may often span several tables. Document stores keep documents in a collection, but the structure of the documents within it can differ from one to another. Moreover, document stores support nested data structures and encourage denormalization, so that all of the information for a real-world object can be a single document in the database. Similar to key-value storage, data in document stores are kept with a primary identifier. However, unlike in key-value storage, the internal structure of data is not entirely hidden to the document store. Consequently, document stores provide more extensive query capabilities for the end-user.

RDBMS satisfy ACID properties and use approaches such as write-ahead logging to guarantee validity even in the event of errors. However, despite the fact that the data is still stored in the disk, document stores encourage keeping the working set in memory, so all the updates are done in transient storage and only periodically synchronized to the disk (most of the NoSQL systems use a similar approach to improve performance at the expense of reliability). Document stores have begun to include mechanisms such as logging protocols and transaction management to ensure reliability. How data is stored, what additional metadata is used, and the memory usage differs from one document store to another. Thus, we focus our study on two representative document store platforms: Couchbase Server and MongoDB.

Several works have been carried out on cost-based schema design for NoSQL systems. In NoSE, the authors evaluate alternative designs for column stores (Cassandra) using the cost of accessing different column families to answer the queries [88]. However, when it comes to document stores, estimating the cost becomes more complex due to the introduction of secondary indexes. The work by Vajk et al. [112] generates multiple alternative schema sketches using denormalization, starting with 3NF. These schemas are then evaluated based on the storage and the number of transactions to choose optimal cloud-based storage solutions. Mortadelo [34] uses a meta-modeling approach to generate database implementations from a high-level conceptual model for document and column stores. A query merging approach is used

Fig. 4.1: Overview of the cost model for document stores

to optimize the access patterns to determine the optimal designs.

Regarding document stores, some work has been carried out on optimizing the data storage in Solid State Drives [100], storage-specific data models [118], and use case comparisons [56], but to the best of our knowledge, not much work has been done regarding the cost models especially for JSON-based storage systems. For example, MongoDB has a query planner for optimizing complex queries that consider multiple execution paths as explained before, and caches the winning plan for subsequent queries. Nevertheless, a server restart or adding/deleting an index will clear the cached query plans. Therefore, the present paper proposes a cost model that predicts not only relative execution time but also memory usage patterns independent from query plan caching that can be used to guide document design.

3   Formalization of the Cost Model

In this section, we present our cost model based on the data storage and query mechanism of document stores. The cost model consists of a generic and a specific component. The generic component is based on three parameters, as shown in Fig. 4.1. First, the type of access affects cost. It can be random access through a primary or secondary index, or a sequential one where the entire collection is read. Second, the storage metadata characterizes the available data in terms of storage structure, indexes, and their sizes. Finally, the memory usage affects query performance, too. Among these parameters, the storage metadata and memory usage are directly affected by the underlying document store specifics. Therefore, we introduce the document store specific component consisting of three segments (storage structure, memory mapping, and memory associativity) to calculate and provide the two parameters to the generic component.

In our work, we focus on random access, which refers to accessing a

document in a collection via physical disk location as these are the majority of the queries used on document stores (owing to the support of secondary indexes). The capital letters in italic correspond to the concepts in Fig 4.1. The storage size of a collection and its indexes mainly depend on the *physical storage structures*, which is the first segment of the specific component. These could be B-trees (*T*), hash buckets (*B*), or heap files (*F*) [77, 94]. In our validation, we use MongoDB and Couchbase Server in which B-trees are used. When bringing the data from the disk, it is directly mapped (*D*) into the memory following the structure as in the disk (MongoDB). However, Couchbase Server uses a hash-based mapping (*H*) where each document in the disk is brought and identified in the memory through a hash value indicating an in-memory bucket. Hence, we introduce *memory mapping* as the second segment of the implementation-specific component. The *memory associativity* in the presence of different collections can be: Pre-determined if the amount of memory used by each of the collections is also predetermined (fixed by the user in Couchbase); or shared if the usage of each collection determines how much memory is devoted to it (MongoDB). The proportion of documents in memory can be easily calculated in the case of pre-determined associativity (*P*). However, with shared associativity (*S*), this proportion is affected by the probability of accessing a collection and its storage metadata as well as the *eviction policy* being used. The eviction policy can be LRU (*L*), FIFO (*Q*), Random (*R*), etc.

We introduce an encoding for the external parameter configurations for the sake of convenience. The encoding contains three to four letters, each representing a segment in the order of storage structure, memory mapping, memory associativity, and eviction policy (only under shared memory). In this work, we introduce cost formulas for a B-tree based disk storage (*T*) under direct memory mapping (*TD*) with pre-defined associativity (*TDP*) or shared associativity under an LRU eviction policy (*TDSL*), and hashed memory mapping (*TH*) with pre-defined associativity (*THP*) or shared associativity under an LRU eviction policy (*THSL*). We validate our model using Couchbase Server (*THP*) and MongoDB (*TDSL*), introducing specific details regarding their implementation. As such, we can include other document stores in our cost model, depending on their configuration.

3.1   Generic Component

We assume that the data is retrieved from the disk to the cache and served for processing from the cache. Furthermore, the disk and the cache are accessed in fixed-size blocks, and the costs of reading a block from the cache or the disk are different but constant. The main focus of the present work is to estimate the cost for data access via the primary or secondary index of a collection (random access). We exemplify our approach on B-tree storage

Table 4.1: Variables of the Cost Model

M	Total memory available for the document store	$B_{i_f}(C)$	Total index size on field f in blocks
f	Indexed field of a collection	$Req_f(C)$	Total number of requests to an indexed field
$Bsize_d$	Block size for data	$P_d(C)$	Probability of queried data block being in the cache
$Bsize_i$	Block size for index	$P_{i_f}(C)$	Probability of queried index block on field f being in the cache
$M_d(C)$	Memory blocks used for the data of a collection	$Cost_{Rand}$	Relative cost for a random read
$M_{i_f}(C)$	Memory blocks used for the index of a collection	$Rep_f(C)$	Number of repetitions of an indexed field
T_m	Time to read a block from cache	$M_{alloc}(C)$	Memory allocated to a collection
T_d	Time to read a block from disk	$P(C, q)$	Probability of querying a collection by a query q
N	Number of collections	$|Q|$	Overall number of queries in an eviction cycle
C	A collection	$SF_f(C)$	Probability of a document being requested using field f
$Size_d(C)$	Average document size of a collection	$P_d^{req}(C)$	Probability of data block in cache being requested
$Size_{i_f}(C)$	Average index entry size of a collection (on field f)	$P_{i_f}^{req}(C)$	Probability of index block in cache being requested
$|C|$	Number of documents of a collection	$M_d^{sat}(C)$	Memory blocks used for the documents of a collection at saturation point
F	Fill factor of the B-tree	$M_{i_f}^{sat}(C)$	Memory blocks used for the index of a collection at saturation point
$R_d(C)$	Average number of documents in a block	K	Total size of non-leaf nodes of all the B-trees
$E_f(C)$	Number of unique requests to an indexed field	P_d^{block}	The probability that any of the documents in a leaf block being requested
$R_{int}(C)$	The average number of reference entries in an internal B-tree block	P_d^{block}	The probability that any of the documents in a leaf block being requested
$R_{i_f}(C)$	Average number of entries in a block for an index over field f	$Shots_d^{in}(C)$	Number of queried data blocks that are in memory within a time window (hits)
$B_d(C)$	Total collection size in blocks	$Shots_d^{out}(C)$	Number of queried data blocks that are not in memory within a time window (misses)
$Bsize_{int}(C)$	Internal B-tree block size	$Size_{int}(C)$	Internal B-tree reference entry size
$Mult_{i_f}(C)$	Multiplying factor of a secondary index	$M_{user}(C)$	Memory allocated by the user for a collection C in pre-defined memory associativity

used by both Couchbase and MongoDB. For the sake of simplicity, we assume that the internal nodes of the B-trees are never removed from the cache in the calculations. Nevertheless, in case that the size of the internal nodes is relevant compared to the amount of memory available, extending the formulas to include them is straightforward (detailed explanation in Sect. 4).

Data distribution plays a vital role in document stores. Our estimations are intended for a single instance of a document store. However, they can be extended into a distributed environment. Assuming that the data has a uniform distribution among the nodes, we can estimate the size of the collections and indexes lying on each node by merely dividing the number of documents by the number of nodes. Therefore, the formulas can be directly applied to each of the nodes independently, just considering that distribution introduces an additional network cost. Existing work has added this cost together with the I/O cost as a weighted sum [92]. The network cost can be estimated by the size of data that needs to be moved between the nodes depending on the query. For example, if a query contains a shard key in MongoDB it will execute it only on the relevant nodes, but if it does not, the query needs to be performed on all the nodes, and the results need to be aggregated.[4] Random access queries can hardly benefit from data distribution, as the end result is accessing a single piece of data in a given machine through a given path. We compared the runtime of a sharded cluster against a single instance for the experiments carried out in this work. In all the cases, the runtime of the distributed system was more than that of the single instance due to the added network cost, except for those instances that can accommodate all the data in memory with more machines.

Table 4.1 lists the variables used in the cost model equations. We define the number of cached data blocks as $M_d(C)$ and cached index blocks as $M_i(C)$. These numbers and their behavior vary depending on the type of document store and the access pattern of a collection, but we assume that the index entry and the document sizes are smaller than the block size ($Size_{i_f}(C), Size_d(C) \ll Bsize_d, Bsize_i$) and the blocks are filled in the average up to a percentage F. Thus, the average number of documents in a document block of collection C, $R_d(C)$, and the average number of index entries in an index block (on a particular field f) $R_{i_f}(C)$ can be defined as follows.

$$R_d(C) = F \cdot \left\lfloor \frac{Bsize_d}{Size_d(C)} \right\rfloor \quad R_{i_f}(C) = F \cdot \left\lfloor \frac{Bsize_i}{Size_{i_f}(C)} \right\rfloor \tag{4.1}$$

Now, we can define the total data leaf blocks of a collection $B_d(C)$ and the total index leaf blocks of the collection $B_{i_f}(C)$ dividing the number of documents by the respective number of documents and index entries that fit in a block as follows. For a secondary index, the leaf level entries depend on

---

[4]https://docs.mongodb.com/manual/core/distributed-queries

how many documents are being pointed by a single index entry, specified by a multiplying factor $Mult_{i_f}(C)$.

$$B_d(C) = \left\lceil \frac{|C|}{R_d(C)} \right\rceil \quad B_{i_f}(C) = \left\lceil \frac{|C| * Mult_{i_f}(C)}{R_{i_f}(C)} \right\rceil \tag{4.2}$$

If there are $M_d(C)$ data blocks and $M_{i_f}(C)$ index blocks in memory for collection C, by using Eq. 4.3, we define the probabilities of the block containing the queried document and the block containing the index entry being in the cache ($P_d(C)$ and $P_{i_f}(C)$) as proportions of the total number of data and index blocks. In the case of hashed memory mapping, these proportions can be taken with the document or index sizes as there is no block structure in memory. (Detailed description in Sect. 3.2.1)

$$P_d(C) = \frac{M_d(C)}{B_d(C)} \qquad P_{i_f}(C) = \frac{M_{i_f}(C)}{B_{i_f}(C)} \tag{4.3}$$

Next, we define the cost function for random access through an indexed field using the above equations. First, the relevant block containing the index of the document needs to be fetched. This block could reside in the cache with probability $P_{i_f}(C)$ or should be retrieved from disk with probability $1 - P_{i_f}(C)$. Next, the block containing the document needs to be retrieved, and this could be from the cache with a probability of $P_d(C)$ or the disk with a probability of $1 - P_d(C)$. Thus, the total cost is the average of retrieving the index and the document blocks as follows.

$$\begin{aligned} Cost_{Rand} = {} & \frac{T_m * P_{i_f}(C) + T_d * (1 - P_{i_f}(C))}{2} \\ & + \frac{T_m * P_d(C) + T_d * (1 - P_d(C))}{2} \end{aligned} \tag{4.4}$$

If the indexed field is that of a typical primary index of a B-tree, the index components can be omitted as the index is contained in the internal nodes (this is not the case in the current MongoDB, refer Sect. 4.2). Now, if we assume that the cost of reading a block from the cache can be neglected compared to the cost of reading from the disk ($T_m \ll T_d$), we can simplify the cost to $\frac{T_d * (2 - (P_{i_f}(C) + P_d(C)))}{2}$. Moreover, considering that block sizes are constant in the system, by replacing $P_{i_f}(C)$ and $P_d(C)$ from Eq. 4.3, we can infer that the cost of random access is negatively correlated with the size of the memory allocated to the index and data, and positively on the collection size (i.e., the size of the B-tree). The collection size is a product of the number of documents and the average document size divided by the fill factor (F in Table 4.1). We define the memory allocated to a particular collection as the sum of memory

used by data and all the indexes (I).

$$M_{alloc}(C) = M_d(C) * Bsize_d + \sum_{k=0}^{I} M_{i_k}(C) * Bsize_i \qquad (4.5)$$

Finally, our generic cost model uses memory usage as an external parameter specific to the underlying technology.

3.2 Specific Component

The specific component consists of three segments, namely, storage structure, memory mapping, and memory associativity. Both Couchbase Server and MongoDB use a B-tree structure to store data and indexes. Since the estimation of B-tree size is a familiar process [77, 94], we focus on memory mapping and memory associativity in detail.

When the document store is started, the cache is assumed to be empty (i.e., cold start). Thus, all the requests sent involve fetching data from disk and caching them in memory. This will continue until the cache becomes full, and the cache eviction starts to release some of the blocks to allow new ones to be cached. As shown by previous work, the cache becomes stable in terms of the memory allocated to the different collections and indexes after a certain point, and its state can consequently be approximated [32, 53, 63, 114]. However, this approximation depends on the specific approaches used for managing the memory.

3.2.1 Memory Mapping

There are two forms of memory mapping that we explore in our work: direct and hashed. First, we define the unique queries issued in the workload $\mathbb{Q}$ as a set of triples. Each triple consists of a collection C, indexed field f, and the probability of using that indexed field $P(C, f)$.

Direct (*D*)   In direct memory mapping, both data and index are stored, retrieved, and managed in memory as blocks. Thus, the formulas used from this point onwards apply to both the data B-tree and the (secondary) index B-tree. We define the number of repetitions of the indexed field f of a collection C as the ratio between the total number of leaf level entries and the number of distinct values of f as follows. When the value is a primary index or has a unique constraint, $Rep_f(C) = 1$ (which is used in Eq. 4.11).

$$Rep_f(C) = \frac{|C| * Mult_{i_f}(C)}{distinct(f)} \qquad (4.6)$$

We assume that just before the eviction starts, there have been $|Q|$ issued queries and we define this state as the saturation point. These queries are from all the collections that are being accessed. Thus, each collection has $Req_f(C)$ number of document requests from each field f, which is proportional to its access frequency.

$$Req_f(C) = |Q| \cdot P(C, f) \tag{4.7}$$

However, the same document or the index can be requested more than once. Therefore, we estimate the number of unique requests $E_f(C)$ as the expected value after issuing $Req_f(C)$ requests out of the total number of distinct values with replacements.

$$E_f(C) = distinct(f) * \left(1 - \left(\frac{distinct(f) - 1}{distinct(f)}\right)^{Req_f(C)}\right) \tag{4.8}$$

Then, we define the selectivity factor in a collection C, with respect to a field f as $SF_f(C)$, which is the probability of a document being requested through the index on f. Using Eq. 4.8, we define this as $\frac{E_f(C)}{distinct(f)}$. However, there could be multiple queries that access the same collection through different indexes. Therefore, we aggregate the selectivity factor of a collection by using the formula for the probability of union on n events as follows. The number of all the queries issued on the document store is denoted by $|Q|$.

$$SF(C) = \sum_{i=1}^{Q} (-1)^{i+1} \left(\sum_{1 \leqslant < k_1 ... < k_i \leqslant |Q|} (SF_{f_{k_1}}(C) \wedge ... \wedge SF_{f_{k_i}}(C))\right) \tag{4.9}$$

If a data block is in the cache, at least one of the documents in the data block must have been requested. So, the probability of a document not being requested is $1 - SF(C)$, and the probability of none of the documents in a data block being requested is $(1 - SF(C))^{R_d(C)}$. Hence, the probability of a data block being requested by a query $P_d^{req}(C)$ is the complement of none of its documents being requested by that query. In turn, the index B-tree has the same $SF(C)$ as it needs to be accessed in order to access the document. Even though the selectivity factor of the secondary index and the data B-tree is the same, there is one crucial difference between the physical storage of the two structures. The secondary index B-tree is sorted by the indexed value while the data B-tree is not. On account of this, the probability of a leaf node of the secondary index not being requested is that of none of the unique index values within the index block being requested. The index block contains $R_{i_f}(C)$ index entries and the number of unique values within the block is $\frac{R_{i_f}(C)}{Rep_f(C)}$.

$$P_d^{req}(C) = 1 - (1 - SF(C))^{R_d(C)} \tag{4.10}$$

$$P_{i_f}^{req}(C) = 1 - (1 - SF_f(C))^{\frac{R_{i_f}(C)}{Rep_f(C)}} \tag{4.11}$$

Consequently, at the saturation point, just before the eviction starts, the size of cached data $M_d^{sat}(C)$ and index $M_i(C)$ can be stated as follows.

$$M_d^{sat}(C) = B_d(C) * P_d^{req}(C) \tag{4.12}$$

$$M_{i_f}^{sat}(C) = B_i(C) * P_{i_f}^{req}(C) \tag{4.13}$$

Hashed (H)   In a hashed memory mapping system, memory is managed per document. The document is brought into memory with its metadata. Here, when there is enough memory for all metadata, only documents are evicted while the metadata remains in memory. Since the relationship between the blocks and the documents are lost in hashing. Thus, we only estimate the size of documents in the cache instead with Eqs. 4.14 and 4.15.

$$M_{i_f}(C) = Size_{i_f}(C) * |C| \tag{4.14}$$

$$M_d(C) = M_{alloc}(C) - Size_{i_f}(C) * |C| \tag{4.15}$$

In the case of not having enough memory to allocate all metadata, a full eviction mode could be used. In this case, the metadata is evicted when the document is evicted from memory. Here, the total memory used by a collection is divided proportionately to the size of the document and metadata, as in Eqs. 4.16 and 4.17.

$$M_d(C) = M_{alloc}(C) \cdot \frac{Size_d(C)}{Size_{i_f}(C) + Size_d(C)} \tag{4.16}$$

$$M_{i_f}(C) = M_{alloc}(C) \cdot \frac{Size_{i_f}(C))}{Size_{i_f}(C) + Size_d(C)} \tag{4.17}$$

With regard to Eq. 4.3, hashed memory mapping loses the block information. Thus, we use values from Eqs. 4.14 to 4.17 as the numerator and the size of the collection as the denominator (e.g. $\frac{R_d(C)}{F}$) to get the proportion of what is in memory out of the overall collection/index.

3.2.2 Memory Associativity

Memory associativity describes how memory is allocated between the collections. Here, we allocate a constant overhead K as extra memory used for parameters that are not considered in the formula. For example, it could be the internal nodes of the B-trees. Moreover, not all the memory is used by the collections and indexes and an upper memory limit is set. The eviction takes place once this limit is reached. We introduce this upper limit as a percentage denoted by u. In **pre-determined** (P) associativity the memory is decided by the user.

$$M_{alloc}(C) + K = M_{user}(C) \tag{4.18}$$

In **shared** (*S*) associativity, the overall memory is shared between different collections which can be formalized as follows.

$$\sum_{i=1}^{N} M_{alloc}(C_i) + K = uM \tag{4.19}$$

3.2.3 Cache Eviction Policy

A Cache and its eviction policy is applicable when there is shared memory associativity. We introduce the formulas for a B-tree based, **LRU** (*L*) cache eviction policy as it is considered to be fair in most of the use cases.

When eviction cycles start, the least recently used blocks are removed from memory. Suppose a document (resp. index entry) is accessed with a probability P_{doc}. In that case, the likelihood of a leaf block in the data B-tree (resp. index B-tree) being accessed is the probability that some of the documents in that block are requested, which is $1 - (1 - P_{doc})^{R_d(C)}$, noted as P_d^{block}. To evict an internal block in the data B-tree (resp. index B-tree), all the leaf blocks pointed by that internal block need to be evicted. Hence, the probability of one of the leaf blocks not being referred is $1 - P_d^{block}$, and consequently the probability of some of these leaf blocks is referred is $1 - (1 - P_d^{block})^{R_{int}(C)}$ noted as P_d^{inter}. Since they depend one on another and $R_{int}(C) \gg R_d(C)$, it is clear that $P_d^{inter} \gg P_d^{block}$, and we can safely assume that the internal data blocks are hardly evicted (only in extreme cases). The same reasoning can be done for an index B-tree (notice that the probability of accessing a document is the same as the probability of accessing its corresponding index entry, so we would similarly obtain P_i^{block} and P_i^{inter}). Therefore, for the sake of simplicity, we only consider the eviction of leaf nodes and assume that all the internal nodes of the data and the index B-trees are pinned to the cache and take constant *K* memory as explained above. Their eviction will only become significant when there is a substantial number of blocks in the leaves. If so, refer to Appendix C for a detailed calculation and extension of Eq. 4.10 to include the eviction of internal B-tree blocks.

We use the term reference entry to name an entry of an internal block which points to a leaf block, and define the average number of reference entries in an internal block $R_{int}(C)$, in terms of internal block size $Bsize_{int}(C)$, reference entry size $Size_{int}(C)$ and the corresponding fill factor. For index B-trees, the reference entry size depends on the field *f*.

$$R_{int}(C) = F \cdot \left\lfloor \frac{Bsize_{int}}{Size_{int}(C)} \right\rfloor \tag{4.20}$$

Thus, the value of *K* can be easily obtained by iteratively moving up on the B-trees, starting from the leaves and calculating the number of blocks at each level by dividing the previous by $R_d(C)$ (or $R_i(C)$ or $Bsize_{int}(C)$ depending

on the kind of B-tree and level). Then, by solving the system of Eqs. 4.7 4.13 under the condition that the sum of memory used equals the total memory available as shown by Eq. 4.19, we obtain the memory distribution just before the eviction, $M_d^{sat}(C)$ and $M_{i_f}^{sat}(C)$.

When the cache is stable, the probability of bringing in a new data block of a collection $P_d^{in}(C)$ should be equal to the probability of evicting a data block from the same collection $P_d^{out}(C)$ and the same can be applied for the index B-tree. Thus, solving the following system of equations, together with Eq. 4.19 we can obtain the stable state of the memory.

$$\forall C_j : P_d^{in}(C_j) = P_d^{out}(C_j), \quad \forall f \in C_j : P_{i_f}^{in}(C_j) = P_{i_f}^{out}(C_j) \tag{4.21}$$

We define $Shots_d(C)$ as the number of queried data blocks at a given time window for a collection at the stable state. Among these queried blocks, $Shots_d^{in}(C)$ number of blocks are already residing in the memory and $Shots_d^{out}(C)$ blocks need to be fetched from the disk into the memory. Thus, the number of queries whose documents are found in the cache is proportional to the number of blocks already in the cache and the ratio of cached blocks.

$$Shots_d^{in}(C) = M_d^{sat}(C) \cdot \frac{M_d(C)}{B_d(C)} \tag{4.22}$$

$$Shots_{i_f}^{in}(C) = M_{i_f}^{sat}(C) \cdot \frac{M_{i_f}(C)}{B_{i_f}(C)} \tag{4.23}$$

The evictable data blocks of a collection $E_d(C)$ are those blocks that have not been accessed in the last eviction cycle (i.e., those least recently used).

$$E_d(C) = M_d(C) - Shots_d^{in}(C) \tag{4.24}$$

Therefore, the evictability of a certain block in memory is $\frac{E_d(C)}{M_d(C)}$. Now, we can define the probability of evicting a data block from a collection (similarly for the index) as being a weighted average as in Eq. 4.25.

$$P_d^{out}(C) = \frac{W_d(C) \cdot \frac{E_d(C)}{M_d(C)}}{\sum_{j=1}^{N} \left(W_d(C_j) \cdot \frac{E_d(C_j)}{M_d(C_j)} + \sum_{j=1}^{N} \left(W_i(C_j) \cdot \frac{E_{i_f}(C_j)}{M_{i_f}(C_j)} \right) \right)} \tag{4.25}$$

We introduce the weight $W_d(C)$ mainly due to the implementation specifics of the underlying document stores. For an ideal LRU cache eviction policy system where it can determine the exact least recently used blocks to be evicted, the value should be 1. Since tracking all the blocks in memory is expensive, different document store implementations enforce approximations of the least recently used blocks.

We define the probability of a block containing the requested document being in the cache as $P_d(C) = \frac{M_d(C)}{B_d(C)}$. Thus, we define the probability of bringing a new block of a collection to the cache with regard to all the collections that are being used.

$$P_d^{in}(C) = \frac{M_d^{sat}(C) \cdot (1 - P_d(C))}{\sum_{j=1}^{N} \left(M_d^{sat}(C_j) \cdot (1 - P_d(C_j)) + \left(\sum_{f=1}^{I} M_{i_f}^{sat}(C_j) \cdot (1 - P_{i_f}(C_j)) \right) \right)} \quad (4.26)$$

4 Applying the cost model

We introduced the generic and the specific cost model components in the previous section. Depending on the document store, the relevant specific component formulas can be used to determine the memory distribution. However, each document store can have its own implementation decisions that need to be taken into account. In this section, we take two document store implementations in detail and discuss how to apply the formulas introduced above.

4.1 Couchbase Server (*THP*)

Couchbase Server is a distributed multi-modal data store that provides scalability, low latency, and high throughput for key-value and JSON document storage. Couchbase Server manages data using buckets, which are a logical grouping of physical resources. It offers two types of buckets, namely Memcached and Couchbase, but we focus our work on the latter because it stores data both in memory and on disk (Memcached only uses memory).

The documents in the disk are stored in a B-tree structure (T). Buckets operate on these documents only when loaded into memory. If the requested document is not currently in memory, it is automatically brought in from the disk individually together with its metadata as a hash (H). A bucket has a quota of dedicated memory which is configured at creation time (P). When a bucket reaches 85% of the allocated memory, an item is evicted. Each document stored in Couchbase Server has a fixed metadata size (i.e., 56 bytes). If a document is being used, its metadata and id need to be in memory. By default, Couchbase recommends all the metadata to be in memory. In this scenario we can apply Eqs. 4.14 and 4.18 with $u = 0.85$.[5] However, this requires more memory when the number of documents grows. Fig. 4.2 shows the memory is allocated to data and metadata with the two different eviction approaches. Eviction of metadata is supported only from version 3.2 onwards.

---

[5]https://docs.couchbase.com/server/5.1/architecture/db-engine-architecture.html (High water mark)

4. Applying the cost model
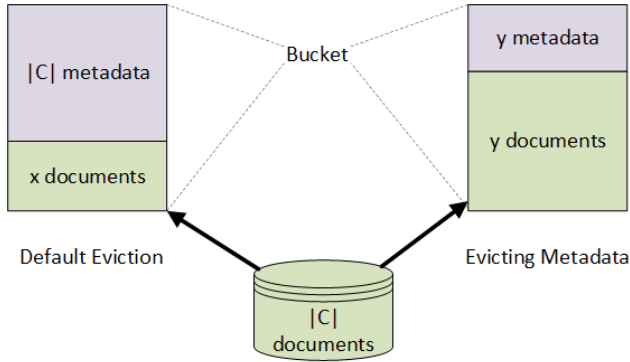
Fig. 4.2: Couchbase Server bucket usage

With default eviction policy, all metadata entries (i.e., $|C|$) will always be in memory, and x documents will use the rest. When evicting metadata is enabled, there will be y documents and the corresponding y metadata entries in memory ($x \leqslant y$). The metadata is evicted together with the document. Thus, we can apply Eqs. 4.16, 4.17 and 4.18 with $u = 0.85$.

Fig. 4.3 shows the distribution of the memory quota among metadata and the documents in five different buckets with the same memory quota but different document sizes. Each of the buckets' average document size increases by a factor, but documents in all the buckets have the same index entry size. Therefore, the metadata size per document is the same. The chart shows that the memory ratio between data and metadata is affected by the document size. When the document size grows, few documents fit in the memory. Since the metadata in memory is only those of the documents also in memory, fewer metadata entries are leading to smaller memory usage.
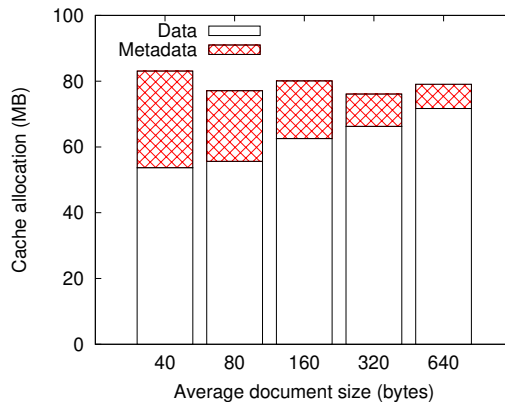
Fig. 4.3: Memory utilization in Couchbase Server

4.2 MongoDB (*TDSL*)

MongoDB stores data in BSON (binary JSON) format and supports ad-hoc queries such as field, range, regular expressions, and aggregation. The documents are stored in collections, have a primary identifier, and also support secondary indexes. It has a pluggable architecture where the end-user can select which storage engine to use. At the moment of writing, there are three main engines: MMAPv1, WiredTiger, and in-memory storage. From now on, we focus on WiredTiger as it is the default and more complex one.
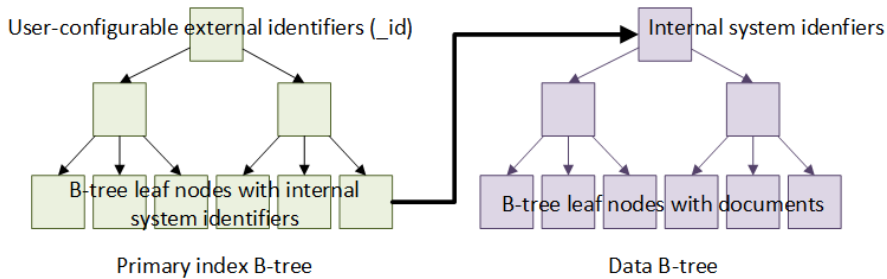
Fig. 4.4: MongoDB B-tree usage for primary key

The storage structure of WiredTiger is a B-tree or LSM-tree with a B-tree memory structure (*T*). However, as of MongoDB 4.2, only the B-tree storage structure is being used, and the LSM structure is not configurable. Moreover, in the current implementation of MongoDB, because of backward compatibility, the internal nodes of the B-tree do not contain the user-configurable external identifier (_id). Instead, as shown in Fig. 4.4, the documents are stored in a B-tree (*data B-tree* from now on) indexed by an internal system identifier. Then, there is a second B-tree (*index B-tree* from now on) where the leaf nodes contain the system identifiers of the data B-tree indexed by the user-configurable external identifier. Thus, the _id field behaves similar to a typical secondary index. The size of leaves is not fixed but capped with a maximum. All the collections and their indexes share a pre-defined cache memory zone (*S*), where all documents are brought in blocks (*D*). When the cache is full, the blocks are evicted to leave room for new blocks to be brought in. WiredTiger uses an LRU-like cache eviction policy (*L*) to evict under-used blocks. Since the index and the documents are in two different B-trees, they behave independently in the WiredTiger cache eviction policy. A running example of applying the formulas in MongoDB can be found in Appendix D.

We carried out different experiments on a single MongoDB instance, randomly accessing documents from various collections changing different parameters. However, tests revealed inconsistencies concerning MongoDB specification. In particular, we identified that the cache eviction policy implementation
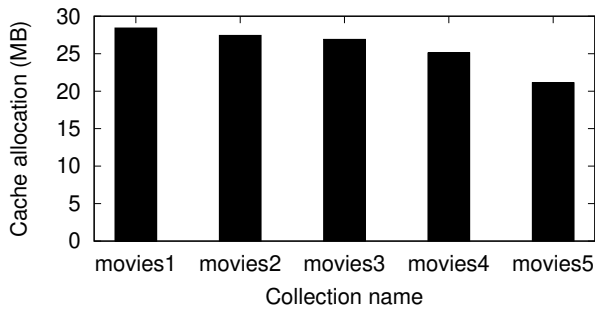
Fig. 4.5: MongoDB cache policy prioritizing the name

(a) Access frequency **(b)** Document size **(c)** Document count
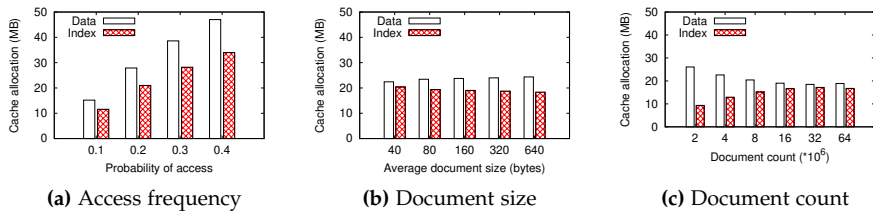
Fig. 4.6: Effect of different parameters on cache distribution in MongoDB

was surprisingly prioritizing the eviction based on the collection's name. This is shown in Fig. 4.5, which depicts memory allocation for five identical collections with the same access frequency, being collection name the only difference (the average cache distribution is measured after 50,000 queries). The authors informed MongoDB about this issue and proposed a bug fix.[6]

Once the bug was solved (fixed in WiredTiger Release 3.2.1), we found three factors that affect the distribution of the cache among the collections and their indexes, namely access frequency, average document size, and the number of documents. Fig. 4.6 shows the distribution of the cache among different collections and their primary indexes after several queries, once the cache is full and stabilized after several eviction cycles (we capped the memory of MongoDB to 256 MB, issued 50,000 random access queries on different collections and took the measures by reading the cache metadata of every collection at the end).

Figs. 4.6a, 4.6b, and 4.6c show the effects of the access frequency, document size, and count on the cache distribution, respectively. It is visible that the frequency of access affects the distribution of the cache the most (as expected), while the impact of the document size and count is smaller. As shown in Fig. 4.6b, the memory allocated to the index decreases compared to that

---

[6]https://jira.mongodb.org/browse/WT-4732

allocated to data when the documents get larger. On the contrary, when the document count changes, the memory used by the index increases while the memory of data drops as depicted by Fig. 4.6c.

We can apply Eqs. 4.7 to 4.13 under Eq. 4.19 with $u = 0.80$[7] for the saturation of the memory and Eqs. 4.21 to 4.26 together with Eq. 4.27 under Eq. 4.19 for the eviction in MongoDB. Yet, MongoDB only keeps track of 300 pages as eviction candidates in a queue. Each B-tree in use is walked to fill out this queue. The number of pages picked to fill this queue from a B-tree is proportional to the current memory occupation of the tree. Hence, $W_d(C)$ in Eq. 4.25 is proportional to the size of the memory occupation. A running example of applying these equations is presented in Appendix D.

$$W_d(C) = \frac{picks_d(C)}{queue\ size}$$

$$picks_d(C) = queue\ size \cdot \frac{M_d(C) \cdot Bsize_d}{M}$$

$$\therefore W_d(C) = \frac{M_d(C) \cdot Bsize_d}{M} \tag{4.27}$$

Other document stores can be included in our cost model with an analysis of their specific design decisions. For example, RethinkDB is a *TDSL* system similar to the one of MongoDB.[8]

5 Experiments

In this section, we validate our cost model through experiments with Couchbase Server and MongoDB. All experiments were carried out on a single node with Intel Xeon E5520, 24 GB of RAM running on Debian 4.9. Couchbase Server Community Edition version 5.1.1 was used with 1 GB dedicated to all the buckets. We used MongoDB Community Edition version 4.2, already modified to fix the bug explained above. We also disabled the parallel execution of the eviction policy to obtain more stable results with fewer repetitions of the experiments. All experiments were conducted using MongoDB Java driver 3.8.2 and Couchbase Java client version 2.6.2. We conducted the experiments with hot cache for both Couchbase Server and MongoDB varying the frequency of access for MongoDB, bucket memory quota for Couchbase Server, average document size, and the number of documents for both. We generated synthetic data with flat documents for our experiments.[9] Despite nesting being relevant in evaluating different designs (it would generate different

---

[7]`http://source.wiredtiger.com/3.2.1/tune_cache.html` (eviction_target)

[8]`https://rethinkdb.com/`

[9]The data generation and the experimental setup can be found in `https://github.com/modithah/MongoExperiments`

document sizes, access patterns and frequencies), it does not change the cost model itself, but only its parametrization. We used GEKKO Optimization Suite [15] to solve the systems of equations for cache distribution.

We measured the runtime individually for 50,000 random access queries (accessing documents through an index) in nanoseconds after the memory became stable for each of the experiments and took the average. Then, using the Pearson correlation coefficient, we measured how our estimates are related to the actual runtime values. However, the query cost estimation formulas introduced in Sect. 3 produce values without any unit. The actual runtime requires a multiplication factor which depends on external factors (i.e., hardware, operating system). Thus, to compare and plot the unitless estimated cost against the actual run time, we first do a min-max normalization on the two series separately. Then, the normalized series are shown in the same line chart. This is a common approach used to compare incomparable data by making them dimensionless [111]. The schema used for all of the experiments is shown in Listing 4.1. We used two integer fields for the primary (_id) and secondary index (s_index) fields. The range of the s_index values is used to change the repetitions with regard to Eq. 4.6, and the load field is used to adjust the size of the document depending on the experiment.

Listing 4.1: Schema used for experiments

```
{
  "_id":    <int> ,
  "s_index":   <int{range}> ,
  "load":   <String(n)>
}
```

5.1 Couchbase Server

The retrieval of the documents through the primary index is done using the java client's $bucket.get(\{randomid\})$ command. As discussed in Sect. 4, Couchbase Server has fix-sized memory quota per bucket. Therefore, the access frequency does not affect the memory distribution. According to Eqs. 4.1, 4.2, 4.16, and 4.17 the memory distribution within a bucket depends on the average size of the documents. Fig. 4.7a compares our estimate of the memory distribution within a bucket using Eqs. 4.16 and 4.17 to the actual one. It is visible that the memory used by the data increases as the size of the documents grows.

Next, we used the memory distribution values in our cost model in Eqs. 4.3 and 4.4. We changed the size of the bucket and the average size of the stored documents and measured the average runtime for queries with random access through the primary index. Fig. 4.7b plots the estimated cost against the actual run time for different bucket sizes. Our estimation shows that there is a linear decrement of the run time when the bucket size is increased. This is also visible through the trend obtained by the actual runtime values. As shown in

(a) Couchbase cache usage prediction

(b) Time estimate with different bucket size

(c) Time estimate with different document size
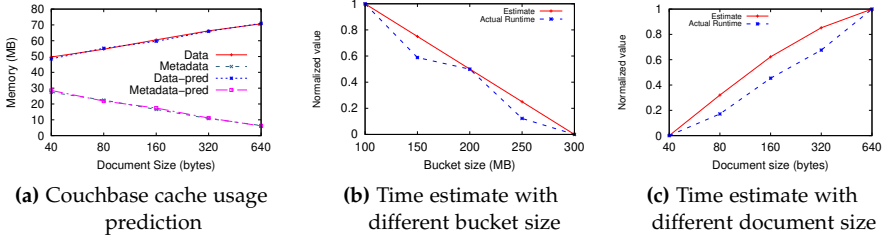
Fig. 4.7: Estimating the memory and time estimation in Couchbase Server

(a) Document size

(b) Document count

(c) Size and count
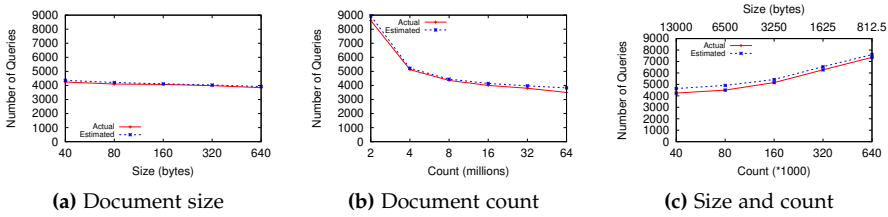
Fig. 4.8: Predicting saturation for a single collection with different parameters in MongoDB

Fig. 4.7c, the runtime gradually increases with the size of the documents.

5.2 MongoDB

Our formulas for predicting memory distribution in MongoDB involve estimating two key factors:

a) The number of queries required to saturate the cache (by solving the system of equations Eqs. 4.7 to 4.13 under Eq. 4.19)

b) The distribution of the cache among different collections and indexes (by solving the system of equations Eqs. 4.21 to 4.26 together with Eq. 4.27 under Eq. 4.19 replacing $|Q|$ from saturation formulas).

For all of the experiments, we executed 50,000 random access queries, measured the cache distribution after every 100 queries, and obtained the average of 10 runs (the system was restarted after each run to reset the cache). We had to measure after every 100 queries because more frequent cache status requests affected the cache policy and the final memory distribution. We scrutinize the values of accessing a single collection and two collections. We varied the number of documents N, average document size $Size_d(C)$, frequency of accessing a collection $P(C, q)$, and the repetitions of the indexed value $Rep_v(C)$ as parameters of concern.

(a) Document size   **(b)** Document count   **(c)** Access frequency
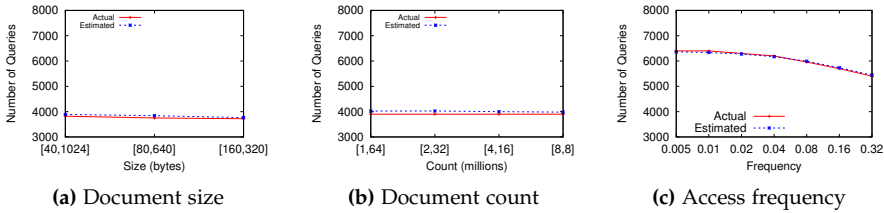
Fig. 4.9: Predicting saturation for two collections with different parameters for MongoDB

For a single collection, we included four tests:

Test 1 Fix the number of documents (13 million) and repetitions (1) while changing the average document size.

Test 2 Fix the average document size (80 B) and repetitions (1) while changing the number of documents.

Test 3 Fix the overall collection size and repetitions (1) while changing both document size and count at the same time.

Test 4 Fix both the average document size (80 B) and the number of documents (13 million) while changing the repetitions (secondary index).

For two collections, we conducted three other tests:

Test 5 Fix the document count (13 million), repetitions (1), and the frequency (50%) while changing the average size of the documents.

Test 6 Fix the average document size (320 B), repetitions (1), and the frequency (50%) while changing the number of documents.

Test 7 Fix the average document size (1 kB), repetitions (1), and the number of documents (1 million) while changing the frequency.

Using the java client we issued `collection.find -One(new BasicDBObject("_id", {random id}))` for tests using primary indexes (Tests 1, 2, 3, 5, 6, and 7) and `collection.findOne( new BasicDBObject ("s_index", {randomvalue}))` for Test 4 involving the secondary indexes.

Predicting Saturation We used Eqs. 4.7–4.13 under Eq. 4.19 to estimate the saturation point ($|Q|$) and compared it with the average number of queries (different runs) before eviction starts.

Fig. 4.8 illustrates the behavior of the saturation point of a single collection. Fig. 4.8a demonstrates that the saturation point is almost constant, with a slight decrease when the size of the documents grow on conducting Test 1.

(a) Document size   **(b)** Document count   **(c)** Size and count
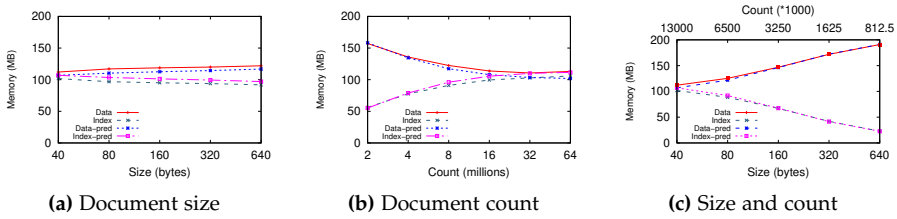
Fig. 4.10: Predicting cache distribution for a single collection with different parameters in MongoDB

(a) Document size   **(b)** Document count   **(c)** Access frequency
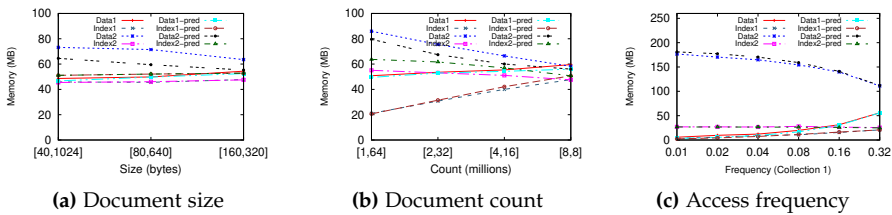
Fig. 4.11: Predicting cache distribution for two collections with different parameters in MongoDB

This is because the documents are accessed in blocks, and before saturation, there are many cache misses leading to bringing new blocks into memory. The number of requests remains almost constant because the probability of a miss is close to one in all cases, given the huge number of documents being used. As shown in Fig. 4.8b, with Test 2, it takes fewer queries to saturate the cache when the number of documents grows. This is due to both index and data B-trees being bigger with the higher number of documents, leading to fewer cache hits and resulting in fewer queries needed to saturate the cache. The impact of the document count is more significant than that of the document size as depicted in Fig. 4.8c, which shows that more queries are needed to saturate as the number of documents grows, and the document size shrinks in Test 3. This is because, the smaller the number of documents in the collection, the higher the hit rate, consequently higher the number of queries required.

We can also see how many queries are required to saturate the cache when accessing two collections (Fig. 4.9). The result of Test 5 is shown in Fig. 4.9a, where we clearly see that a few more queries are needed to saturate the cache (there is a slight downward trend) when the document size difference is higher to saturate the cache due to the higher hit rate of smaller document sizes. The saturation point is a bit lower than of the single collection, because of the space taken by the internal nodes pushing the memory to fill earlier (i.e., K is bigger). We noticed that the effect of the document size is more evident when

there is a smaller number of documents (1 million). The effect of the document count (Test 6) is also negligible, demonstrated through Fig. 4.9b, whose values are comparable to the ones of single collections (Fig. 4.8b) beyond 16 million documents (notice that the sum of both collections is always above that). The only remarkable difference is that the saturation point is lower than that of a single collection due to more internal nodes being pinned. Finally, Fig. 4.9c shows the results of the saturation point of Test 7. We can see that more queries are needed to saturate the cache when the access frequency is low. The real reason, however, is that there is an opposite collection which is accessed with complementary frequency for each of the points (e.g., 0.995 for 0.005). The opposite collection has obviously more documents in the cache due to the higher access frequency leading to higher hit rates, and as a result, more queries are required to saturate the cache. Thus, the more balanced the frequencies of collections, the fewer queries are needed to saturate the cache.

(a) Document size   **(b)** Document count   **(c)** Access frequency
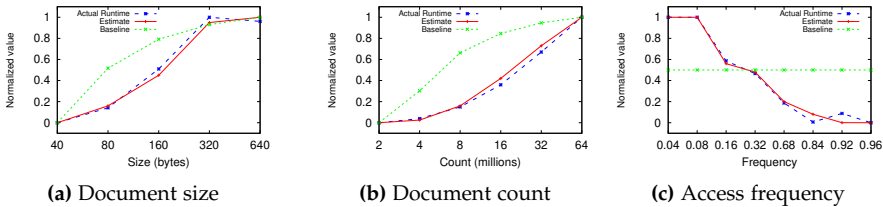
Fig. 4.12: Time estimation comparison for different parameters in MongoDB

Predicting the Cache Distribution   Once we know when the memory saturates and starts being stable, we can analyze how memory is distributed among collections and indexes.

The outcome of the memory distribution for Test 1 is shown in Fig. 4.10a. When increasing the document size, the amount of memory devoted to data blocks grows while the memory for index blocks decreases. This happens because the data B-tree becomes larger with larger document sizes, and it occupies more memory, but the index B-tree size remains the same. However, the effect is minimal. Fig. 4.10b shows the results of Test 2 in measuring the effect of the number of documents on the memory distribution. The size of both B-trees grows with the number of documents. However, the index B-tree grows slower than the data B-tree, resulting in higher hit rates and more memory. The internal blocks of the B-tree increases as the number of documents increase. These internal blocks could also be getting evicted in the experiments whereas we assume them to be pinned in the cache. Thus, our estimation error gets higher as the number of documents increase. When we change both the size and the number of documents, keeping constant the

collection size as per Test 3, we observe the trend augmented as shown in Fig. 4.10c (note that the axis of the number of documents is reversed due to the inverse relationship between the document count and the document size as we try to maintain the same overall collection size), because both factors favor the growth of memory used for the data (i.e., the less and bigger documents the more memory devoted to data and the less to index). When the repetitions increase in Test 4, a single index entry points to multiple documents, thus increasing the memory used by the data and decreasing the index. Our cost model is able to predict the memory distribution with a maximum error of 4%.

Fig. 4.11 illustrates in the corresponding subfigures the effect of the document size, number of documents, and the access frequency when accessing two collections. The gap between the memory used by the data decreases when the average document sizes are closer in Test 5, while the memory used by the index remains constant (Fig. 4.11a). Fewer documents fit in memory when the documents are larger, resulting in more cache misses and more data blocks need to be brought into the cache. When looking at Fig. 4.11b (Test 6), the memory used by the index of the first collection (*index-1*) increases while the one for data of the second collection (*data-2*) decreases when the difference between the number of documents of the two collections decreases. The memory used by *data-1* slightly increases and *index-2* decreases and align with *data-2* and *index-1* when the counts become identical. When a collection has more documents, there are more pinned internal nodes, and there are more cache misses requiring to bring more data and index blocks into the memory. When the collections are identical, the memory usage is similar. With regard to Fig. 4.11c (Test 7), the memory used by both data and index increases with the access frequency. When a collection is accessed with a higher rate, the evictability of a block becomes lower, resulting in more blocks residing in memory.

Query Cost Estimation   Finally, once we have the memory distribution, we can analyze the performance of the system in a stable state using the generic cost functions. As illustrated by Fig. 4.12a (Test 1), the runtime increases as the size of the documents increases. Since the memory size is fixed, the number of documents that fit in the cache gets smaller resulting in more cache misses, leading to fetching more documents from the disk and increasing the runtime. As shown in Fig. 4.12b (Test 2), in the case of fixed-document size increasing the document count, the probability of a cache hit is higher on smaller document counts, and the runtime rises with the number of documents. The runtime gets lower as the frequency of access gets higher, as demonstrated by Fig. 4.12c (Test 7). This is because the collections with higher access frequency have more memory, which creates higher hit

rates and lower runtimes. We calculated the miss rates of the hieracical cost model of Manegold et al. [83] and plotted the estimated costs as a baseline in Figs. 4.12a and 4.12b for different document sizes and counts, respectively. We also include a horizontal line at 0.5 in the approximation in Fig 4.12c as all the cost estimations of Manegold et al. coincide in this case. (an in depth discussion of that approach can be found in Sect. 5.4)

5.3 Accuracy of Prediction

Regarding our memory predictions, on looking at Fig. 4.7a, we see that our estimated trend is identical to the actual memory distribution of Couchbase Server with an average error of 3%. For MongoDB, this is a bit more complex, since we need the number of queries required to saturate the memory (Fig. 4.8 and 4.9), which is in general slightly overestimated, for an average error of 3% and a correlation of 0.995. By using it in solving the corresponding system of equations, we can predict memory usage of the data and indexes and accurately find the trend in all the cases with an under-estimation of the data while over-estimating the index, for an average error in all the scenarios of 6% for index and 5% for the data. As shown in Figs. 4.10, 4.11, and 4.11c, the estimates of the memory usage are highly correlated (0.995) with the actual values. With regard to Figs. 4.11a and 4.11b, prediction for *index-1* and *data-1* are almost perfect, but we underestimate *data-2* and overestimate *index-2*. The highest error we encountered is with the prediction of the *data-2* when changing the document size. When looking into Fig. 4.11c, we can see that the error increases for the number of data pages when increasing the probability of accessing the collection.

Overall, we have successfully predicted the allocation of the memory with a maximum error in all the experiments of 13% and an average error of 9% for the different number of documents, average document sizes, probability of access, and available memory. However, when it comes to the runtime estimates, the effect of this error becomes negligible. Regarding runtime predictions, we manage to find the runtime trend in Couchbase Server with a high correlation (0.93) between all our estimates and the actual values (values from Fig. 4.7b and 4.7c). The correlation is even higher (0.94) for MongoDB (values from Fig. 4.12a, 4.12b, and 4.12c). There is a slight difference between our estimate and the actual value when the access frequency is very low and very high (Fig. 4.12c), because the measured runtime values are very close to each other in extreme cases (three milliseconds). Thus, our approach enables us to identify the overall effect of the design decisions on runtime. This runtime, together with other parameters such as storage space and heterogeneity of a collection, can be used to evaluate design alternatives.

5.4 Comparison to Other Approaches

To the best of our knowledge, this work is the first generic cost model for document stores. However, we can compare our cost estimations against a generic cost model for hierarchical memory systems (hereinafter referred as the hierarchical cost model) [83]. This cost model is based on in-memory database systems and can be extended to the disk I/O layer. However, unlike the database system discussed in the hierarchical cost model, which relies only on the operating system cache, document stores have their own cache management system. Our cost model has the capacity to handle these implementation-specific cache policies, and we have shown it by applying them on two different document stores, Couchbase Server and MongoDB.

Random access in the hierarchical cost model is equivalent to the random access queries discussed in this chapter. Nevertheless, it requires the number of random accesses performed to model the cost, which is not required in our approach that only relies on the stability of the cache. Moreover, the Sterling numbers and the factorials used in the calculations quickly grow quite large making the calculations almost impossible for large numbers (the largest Sterling number the authors have used in their experiments is 1024 which corresponds to the number of L1 cache lines [82]) unless you make mathematical approximations on the formulas, thus increasing the error. Therefore, the approach used in the hierarchical cost model is not scalable to the experiments that we have carried out as database caches are larger than 1024 blocks. However, we substituted Eq. 4.8 for the expected value whose result for small values exactly coincide.

The concurrent execution formulas in the hierarchical cost model assume that the cache contains a fraction of data regions involved proportional to their footprint size (i.e., the size of the collection). However, our experimental results show otherwise, especially with different access frequencies (See Fig. 4.6) and eviction policies. The pattern is even more complex when it comes to secondary indexes where a sequential accesses on the index are followed by multiple random accesses to the documents. The comparison results in Figs. 4.12a, 4.12b, and 4.12c further confirm that a estimation specific for document stores, is clearly superior to a generic approach for simply caching.

Chapter 5

Automated Database Design for Document Stores with Multi-criteria Optimization

This chapter is under submission for Knowledge and Information Systems (KAIS).

The layout of the paper has been revised.

Co-authoring declaration   This work has been done together with the post doctoral researcher Sergi Nadal. Precisely, the introduction and problem formation were done jointly with equal contribution. The canonical model, random design generation, and design transformations were done by Moditha Hewasinghage, while the search algorithm was done by Sergi Nadal. The experimental evaluation was jointly developed, with Sergi Nadal focusing on search algorithm and Moditha Hewasinghage implementing the canonical model, transformations, and loss functions.

1 Introduction

In the last couple of decades, the data storage paradigm has shifted from traditional relational database management systems (RDBMSs) towards more flexible NoSQL engines [24]. Among these, the popularity of document stores has prevailed due to the adoption of semi-structured data models, which avoids the impedance mismatch between the data storage and applications. Document stores allow users to focus on rapid application development with a data-first approach instead of the traditional schema-first of RDBMSs. This has caused an increase in their popularity, especially in the startup ecosystem, where the goal is to rapidly deliver products into the competitive market [36]. As a consequence, database design (i.e., the design of database structures and their relationships) has been overlooked and not performed in a principled manner. However, it has been shown that the choice of database design plays a critical role in performance [10]. Database design for document stores is, in general, mostly carried out in a trial-and-error or ad-hoc rule-based manner. For instance, MongoDB, the leading document store, provides a set of design patterns that define certain guidelines on how to structure documents.[1] Nevertheless, even with these guidelines, it is still common to make bad design decisions, and issues tend to arise in the long run when the amount of data has grown considerably, and changing them incur additional cost, time, and money.

Let us consider an exemplary scenario of product reviews composed of the entities *Comments* and *Products*, with a one-to-many relationship from the latter to the former, as well as an equiprobable hypothetical workload defined as follows: *a)* given a Comment ID, find its text; *b)* given a Product ID, find its name; *c)* given a Comment ID, find the Product name; and *d)* given a Product ID, find all of its Comments. To illustrate the complexity of manually determining the optimal design even in such an oversimplistic scenario, we conducted a questionnaire to database experts on an equivalent setting.[2] 63% of the participants managed to identify only three potential designs for a document store, many of them overlooking the redundant nesting and referencing options. After comparing the results against an actual experimental setup performance on MongoDB, we concluded that only 9% of the participants managed to find the optimal design, while 40% guessed the fourth-best design as the optimal one (in terms of query performance). This evidences that the current way of database design does not yield the expected results, even for very limited scenarios like this. Indeed, real-world scenarios are far more complex involving multiple entities and relationships.

If we assume that all attributes of an entity are kept together within the

---

[1]https://www.mongodb.com/blog/post/building-with-patterns-a-summary
[2]https://moditha.typeform.com/to/NRTjEm
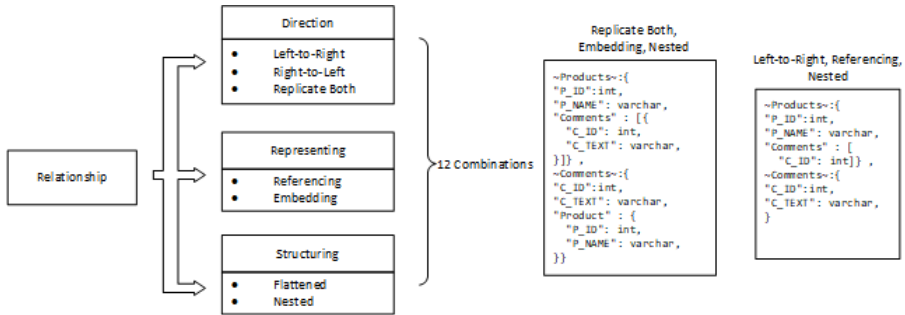
Fig. 5.1: Relationship storage choices for database design

same document, we are still left with the decision on where the relationships must be stored in the final design. Thus, database designs can be enumerated based on the alternatives to store the relationships, which depend on three independent choices: Direction, Representing, and Structuring, as shown in Fig. 5.1 together with two examples. **Direction** determines which entity keeps the information about the relationship. It can be any of the two entities or both. **Representation** affects how this relationship is stored by either keeping a reference or embedding the object. Finally, **Structuring** determines how we structure the relationship, either as a nested list or flattened (in the case of one-to-many relationships). For example, if we decide to keep the references to the comments in the product, they can be stored as a list of references (*comment:...*) or in a flattened manner (*comment_1:.., comment_2:..*). Hence, we end up with 24 possible designs for our running example. Depending on end-user preference, each one of these designs has the potential to become the optimal design choice. This trade-off between alternatives makes the process of finding the optimal design a complex one.

As a matter of fact, the number of relationships r determines the number of candidate designs, which is exponential (24^r), as the storage option of each relationship is independent of others. Note, however, that here we did not consider allowing heterogeneous collections/lists, which is possible in the context of schemaless databases, leading to a complexity increase. For example, collections at the top level could potentially contain different kinds of documents. In our running example, the product and comment documents could be stored in a single heterogeneous collection mixing both. Precisely, for a design with c top-level collections, the total number of combinations will be $\sum_{i=1}^{c} \left\{ {c \atop i} \right\}$, where $\left\{ {n \atop k} \right\}$ is the Stirling number of the second kind, corresponding to the number of ways to partition n distinct elements into k non-empty subsets [51]. Overall, such exponential growth makes it impossible to enumerate and evaluate all candidate designs.

To overcome such limitations, there exist several solutions in the literature

mainly relying on the query workload to propose a database design, such as DBSR [95], NoSE [88] and Mortadelo [34]. However, query performance is not the only factor affected by design choices. For instance, having redundant collections to support different queries will increase query performance but require larger storage capacity. Having complex document structures with multiple branching and nesting levels hinders the readability of documents. Similarly, having heterogeneous collections requires additional documentation to map where each piece of information is stored. We believe that it is essential to find an optimal design that considers a variety of the user's preferences automatically. To that end, we propose a novel automated database design method for document stores that considers all such factors. This is achieved by the adoption of well-known multicriteria optimization techniques, proven to efficiently tackle multiple and conflicting objectives [27]. Thus, we generate near-optimal database designs that balance the end-user's preferences regarding four quantifiable objectives: storage size, query performance, degree of heterogeneity and the average depth of documents. This work has been demonstrated as a tool in Appendix B. To show the effectiveness of our method, we compare it against an existing document store schema generation tool in terms of the quality of the generated design and its performance. Our experimental results show that we manage to present a design with less redundancy and offer better performance with the flexibility of catering to the end-user preference. We can also scale up when the number of entities and relationships grows.

Contributions. The main contributions of our work are as follows:

- We propose a novel loss function for multicriteria selection of the optimal database design for document stores.

- We devise an algebra of transformations that allow to systematically modify a database design while retaining all the information (i.e., no attributes or entities are lost).

- We present an implementation of a local search algorithm that, driven by the loss function, proposes with high probability near-optimal designs.

- We assess our method and prototype to show the scalability as well as the performance gain of our solution against competitors by using the RUBiS benchmark [25].

2   Related Work

JSON has gained popularity in recent years as an alternative storage format to XML. Although JSON is semi-structured, a schema can be defined [93],

and thus some approaches aim to extract such schema representation from heterogeneous JSON documents [71]. When it comes to data design for document stores, these mainly rely on finding a solution that balances the two extremes (i.e., normalized and embedded models) [66]. As shown in Chapter 2, the design decisions for JSON storage are not trivial and can affect the performance as storage requirements that also depends on the choice of the storage system. Moreover, some authors [48] discuss data quality aspects that could arise in JSON stores and how to measure them, which can help to evaluate the potential document designs.

It is well-known that design methods for NoSQL data stores must consider both relational and co-relational perspectives at once [58]. Nevertheless, although data modeling has played a significant role in RDBMS, little work has been carried out on data design methods for document stores. Different approaches have been employed to tackle such problems, including some for analytical workloads [106], as well as cost-based schema design evaluation (Appendix A) based on a hypergraph data model (Chapter 3). NoAM [10] uses three constructs (collection, block, and key) to introduce an abstract data model that can be mapped into heterogeneous NoSQL stores based on entity aggregates. Alternatively, the SOS platform [11] introduces three design constructs (attributes, structs, and sets), which are capable of representing relational, key-value, document, and column-family stores. Indeed, our approach builds on a hypergraph-based formalization of the SOS constructs, which we proposed in Chapter 3. This formalization enables the generation of native queries over the heterogeneous stores to manage the metadata of polyglot systems, thus representing the basis for defining design transformations.

Additionally, several work have been carried out specifically on automated schema design for NoSQL stores. We used the search terms *NoSQL design*, *document store schema*, *nosql schema* as the seach terms to select the initial set of articles and used a a snowballing approach to track down references related on automated schema design on document stores and NoSQL systems in general. However, there are only a few schema generation/suggestion tools available for NoSQL systems. DBSR [95] is a database schema recommender for document stores, which uses a search-based approach to navigate the data design space similar to the one of this paper. However, DBSR only considers the query workload in generating potential designs. Moreover, DBSR only considers nesting as opposed to referencing, avoiding joins altogether. However, in our work we have encountered instances where external joins perform better than certain embedded approaches (See Appendix A). Thus, DBSR can omit some optimal designs in its process. DBSR compares itself to other existing approaches that we found relevant for our work and we have compared our approach against it since it is the latest research carried out in automated schema design and specific to document stores. There are several similarities between DocDesign 2.0 and DBSR they are both based on

read-only workloads, allow data duplication, nested structures, secondary index usage, and query plan estimation on document stores. However, the cost model (Chapter 4) used by DocDesign 2.0 is more mature and robust to the underlying document store implementation as opposed to the simple access pattern based one used by DBSR. Moreover, DocDesign 2.0 uses a multi criteria-based approach with more fine tune capabilities to the user compared to only tuning the number of final collections in DBSR resulting a superior output.

NoSE [88] uses a cost-based schema design approach specific for column stores. In Mortadelo [34], a model-driven data design based on a generic data model is used to generate optimized datastore schemas and queries for document and column stores. Another approach [35] generates a physical schema from a logical one for document stores based on the workload by optimizing the query access patterns. Unlike rule-based design generation in these approaches that could omit certain designs, our work opens the door to potentially generate all possible combinations.

3 Overview

In this section, we provide a high-level overview of the components of our approach. In order to yield the most suitable database design for a given set of contradicting objectives, we adopt multicriteria optimization techniques. These have shown to be effective in obtaining near-optimal solutions out of a large search space in the presence of conflicting objectives [84]. In these scenarios, one can only aim to obtain a Pareto-optimal solution (a solution that, in the presence of multiple objectives, cannot improve one objective without worsening another). An overview of our approach is shown in Fig. 5.2, next in the following subsections, we briefly describe each of its components: Namely user inputs, the design process, loss function, and the search algorithm.
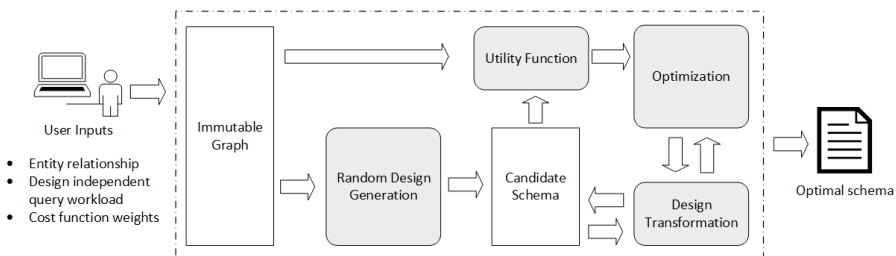
Fig. 5.2: High-level overview of our approach

Let us take the RUBiS benchmark [25] as a running example. As depicted in Fig. 5.3, RUBiS implements an online auction system (we also take the
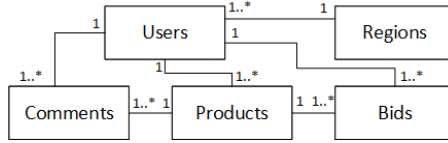
Fig. 5.3: ER diagram for RUBiS framework

same 11 queries used by the DBSR framework as our workload)[3]. If one were to design a traditional RDBMS data storage for this case, it would be natural to apply normalization and deploy a database schema in either third-normal form (3NF) or Boyce-Codd normal form (BCNF). It is well known from relational theory that such a design avoids update anomalies and would still efficiently execute many queries. However, this penalty is not acceptable for document stores, and they encourage denormalization and promote nesting to keep related pieces of information together, avoiding expensive joins even if increasing redundancy. It is thus unclear what would be the best database design, whether it could be a fully denormalized collection with `Regions` as the top-level document, a normalized approach similar to 3NF, or an in-between solution. With reference to our estimate in the introduction, there are 24^6 possible alternative designs to choose from.

3.1 User Inputs

The end-user must provide three inputs to initiate the optimization process: the Entity-Relationship (ER) diagram (enriched with some physical information like attribute sizes and cardinalities), a query workload (i.e., a set of design-independent queries together with their frequencies), and the weights of the four cost functions.

Entity-Relationship describes the domain in terms of a graph and determines the complexity of the optimization problem. The user has to identify the entities, the attributes of each entity, the average size of the attributes, the number of instances of the entities, and their relationships, including the multiplicities. Essentially, this depicts an ER diagram serialized in JSON format. As previously mentioned, we represent this information using the hypergraph-based canonical model introduced in Chapter 3. In our running example, these are the entities, relationships, and multiplicities shown in Fig. 5.3. By definition, this graph is considered immutable, and we carry out the database design on top of it by building hyperedges. We discuss the canonical model and the design process in detail in Sect. 4.

Query workload determines the user's query requirements on the underlying

---

[3]`https://github.com/vreniers/DBSR`

data. Since the input ER does not contain design information, the queries are also represented in a design-independent manner. Thus, a query is a set of interconnected *Atoms* with a specific one representing a selection criterion (as defined in the cost model in Chapter 4). Our approach works with a constant workload, including the frequency of each of the queries. In our running example, the workload would be the 11 queries in RUBiS together with their frequencies.

Cost function weights determine how the final design performs in four criteria, namely: query performance, storage size, degree of heterogeneity within a collection, and depth of the documents. Each of these has a cost function associated which will be weighted according to the user's needs to compose the loss function to be optimized.

3.2   Design Processes

Once the user has provided the ER diagram, it is stored in an immutable graph where the entities and attributes are represented as vertices (*Atoms*) and relationships as edges. Precisely, we perform all design operations on top of this immutable graph generating candidate designs. Two processes generate these candidate designs, namely: initial **random design generation** (through those relationship design choices shown in Fig. 5.1) and **design transformation** of an existing design (through the methods associated with the different classes in the canonical model). For instance, the random design generator could generate a design with five separate collections for each entity with their respective references (e.g., the user references the region). Then, through design transformations, we can generate an alternative design from the existing one by embedding the region inside the user. We introduce the formal canonical model, algorithms to generate random designs, and design transformations in Sect. 4.

3.3   Loss Function

We introduce four components of the loss function to be measured and optimized in DocDesign 2.0: query cost, storage cost, degree of heterogeneity, and the average depth of the documents (as a measure of their complexity). We chose these loss functions as a representative of different cost functions and not intended as an ultimate combination that would provide the optimal database design. These are defined as follows:

Query cost (CF_Q) is the weighted average of the relative query performance values estimated from the schema information using the cost model. This cost model is configurable according to the storage and the execution model of different document stores. Thus, it is possible to adapt DocDesign 2.0 to

alternate document store implementations. The cost model firstly takes the query workload as an input, calculates the distribution of the cache under a workload, and estimates a relative cost of accessing each of the collections and indexes. Then, the total relative cost of each of the queries (Q_q) that depends on the access patterns of the storage structures is also calculated. Thus, we can sum up the total query cost as $CF_Q = \sum_{k=1}^{N_q} f^k \cdot Q_q^k$ where N_q is the number of queries in the workload and f^k is the frequency of the Query.

Storage size (CF_S). Is the total storage size required by the collections and indexes, calculated using schema information in the canonical model. We define the size of a collection and an index as S_C and S_I, respectively. Thus, the total storage size $CF_S = \sum_{k=1}^{N_c} S_C^k + \sum_{k=1}^{N_i} S_I^k$ where N_c and N_i are the total number of collections and indexes in the design.

Degree of heterogeneity (CF_H) is the number of distinct types of documents in a collection/list. We use the weighted average over all the collections and lists of the schema. Each value is given a weight depending on which level the list/collection lies in the document. The higher the level, the higher the assigned weight, thus penalizing heterogeneities at higher levels of the document structure. If there are n heterogeneous collection/list at a given level lv, the degree of heterogeneity value $H_{lv} = \frac{n}{lv+1}$. Assuming there are N_h number of collections/lists in the design, the average degree of heterogeneity $CF_H = \frac{\sum_{k=1}^{N_h} H_{lv}^k}{N_h}$.

Depth of the documents (CF_D) is the average depth of the documents of the design. We can assume that each document at the top level of a collection has branches reaching from the root to the leaf level with a length ln, and there is N_b number of branches in all the documents in the top level. Thus, we define the average depth of the documents of the design $CF_D = \frac{\sum_{k=1}^{N_b} ln^k}{N_b}$.

3.4 Search Algorithm

Local search algorithms consist of the systematic modification of a given state, utilizing action functions, in order to derive an improved state. The intricacy of these algorithms consists of their parametrization, which is, at the same time, their key performance aspect. Due to the genericity of different use cases our method can tackle, we decided to choose *hill-climbing*, a non-parametrized search algorithm that can be seen as a local search, always following the path that yields lower loss values. Nevertheless, the cost functions we use are highly variable and non-monotonic, which can cause hill-climbing to provide different outputs depending on the initial state. To overcome this problem, we adopt a variant named *shotgun hill-climbing*, which consists of a hill-climbing with restarts using random initial states.

Algorithm 6 Shotgun Hill-Climbing

Input: [Initial design, number of non-improving iterations] *D*, *N*:
Output: [Solution design] *solution*
1: *solution* ← null; *i* ← *N*
2: **while** *i* > 0 **do**
3:   *B* ← randomInitialState(*D*); *finished* ← *false*
4:   **while** !*finished* **do**
5:     *neighbors* ← applyAllPossibleTransformations(*B*)
6:     *B'* ← stateWithSmallestLoss(*neighbors*)
7:     **if** l(*B'*) < l(*B*) **then**
8:       *B* ← *B'*
9:     **else**
10:      *finished* ← *true*
11:   **if** l(*B*) < l(*solution*) **then**
12:     *solution* ← *B*
13:     *i* ← *N*
14:   --*i*

Loss function. Guiding the local search algorithm requires the definition of a loss function, taking into account the end-user preferences. Here, this is a function to be minimized. Hence, the end-user can assign weights to each cost function according to their importance in the use case. Then, for a given design *C*, we define the loss as the normalized weighted sum of each cost function $l(C) = \sum_{k=1}^{n} w_k \frac{CF_k(C) - CF_k^{min}}{CF_k^{max} - CF_k^{min}}$. The expression considers the weight w_k of each cost function, which is used on the transformed loss function for *C*. This is a normalized value that considers the *utopia* (i.e., the expected minimal) and the maximal design costs, yielding values between zero and one, depending on the accuracy of both CF_i^{min} and CF_i^{max}, which rely on estimations.

Shotgun hill-climbing. Algorithm 6 depicts an overview of shotgun hill-climbing. The method generates random designs and systematically improves them by applying at each step all available transformations keeping the one that yields the minimum loss value. Note, however, that this approach is highly susceptible to fall in local minimums. To overcome this issue, we repeat the process a certain number of iterations and keep the one with the minimum loss, overall. Once such a solution is not improved for a certain number of iterations denoted by *N* (i.e., we do not find any new state with a smaller loss), it is highly probable that we have found the optimal design.

Thus, DocDesign 2.0 will provide a pareto-optimal design depending on the importance given to the each of the loss functions and the number of non-improving iterations given by the user. For instance, retaking the

running example, if the user had specified to optimize on the storage space, the solution would be individual collections with references with minimal or no nested documents (Listing 5.1). Or, if the query performance is the only important factor, it is likely to generate collections that can answer the queries without joins increasing the redundancy (Listing 5.2).

Listing 5.2: Optimized for queries

```
"USER-BIDS": {
  "U_ID": int(4),
  "U_F_NAME": varchar(20),
  "REGION": {
    "R_ID": int(4),
    "R_NAME": varchar(10)
  },
  "BIDS": [{
    "B_ID": int(4),
    "B_PRICE": int(6),
    "U_ID": int(4)
  }]
}
"USER-COMMENTS": {
  "U_ID": int(4),
  "U_F_NAME": varchar(20),
  "R_ID": int(4),
  "REGION": {
    "R_ID": int(4),
    "R_NAME": varchar(10)
  },
  "COMMENTS": [{
    "C_ID": int(4),
    "C_TITLE": varchar(20),
    "U_ID": int(4)
  }]
}
"PRODUCT-COMMENT": {
  "P_ID": int(4),
  "P_TITLE": varchar(10),
  "COMMENTS": [{
    "C_ID": int(4),
    "C_TITLE": varchar(20),
    "U_ID": int(4)
  }],
  "B_ID": int(4),
  "U_ID": int(4)
}
"BID-USER": {
  "B_ID": int(4),
  "B_PRICE": int(6),
  "U_ID": int(4),
  "U_F_NAME": varchar(20),
}
```

Listing 5.1: Optimized for storage

```
"REGION": {
  "R_ID": int(4),
  "R_NAME": varchar(10)
}
"USER": {
  "U_ID": int(4),
  "U_F_NAME": varchar(20),
  "R_ID": int(4),
}
"PRODUCT": {
  "P_ID": int(4),
  "P_TITLE": varchar(10),
  "C_ID": int(4),
  "B_ID": int(4),
  "U_ID": int(4)
}
"COMMENT": {
  "C_ID": int(4),
  "C_TITLE": varchar(20),
  "U_ID": int(4),
  "P_ID": int(4)
}
"BID": {
  "B_ID": int(4),
  "B_PRICE": int(6),
  "U_ID": int(4)
}
```

4 Canonical Model

In this section, we present the canonical data model we use to represent database designs for document stores. We extend our previous work intro-

99

duced in Chapter 3, where we presented a hypergraph-based canonical data model for polyglot systems. Here, we extend it with additional artifacts and class methods specific to document stores. To that end, we distinguish three levels of detail from most abstract to most specific (immutable, storage agnostic, and document store-specific constructs). Each subsection corresponds to the three abstraction levels. From now on, we will use the conventions listed in Table 5.1.

Table 5.1: Variables used in the paper

C	The catalog (the design)	E_N	Node (either atom or edge)
A	Atom	$\mathbb{A}$	Set of all atoms in C
A_C	Class atom	$O(h)$	Root atom of a *struct* h
$E_R^{x,y}$	Directed relationship between two atoms x and y	$\mathbb{E}_R$	Set of all relationships in C
E_H	Hyperedge	E_H^+	Transitive closure of E_H
E_{Struct}	*Struct*	$\mathbb{E}_{Struct}$	Set of all the E_{Struct}s in C
E_{Set}	*Set*	$\mathbb{E}_{Set}$	Set of all the E_{Set}s in C
E_{Doc}^{Doc}	Hyperedge representing a document of a document store	E_{Top}^{Doc}	E_{Doc}^{Doc} representing top level document of a collection of a document store
E_{List}^{Doc}	Hyperedge representing a list in a document store	E_{Col}^{Doc}	E_{List}^{Doc} representing a top level list (a.k.a collection)
$E_R^{A_C,A_C}$	Relationship between two class atoms	E	Generic superclass for edges

4.1 Immutable Graph

Figure 5.4 depicts the main constructs of our canonical data model with the three levels of abstraction highlighted. The ER diagram provided by the user (e.g., Comments and Products) is considered immutable. This immutable information is a simple graph consisting of *Atoms* (A) and *Relationships* (E_R). An *Atom* is either an attribute A_A or a class representative A_C (e.g., comment text and comment ID respectively). This immutable graph can only represent binary relationships of an ER diagram. However, the representation of ternary relationships is also possible reification of the relationship and transforming into binary relationships.

Over the immutable metadata, we define the different design artifacts depending on the model being used (e.g., documents in our case). Therefore, we define design operations for those artifacts at two specialization levels,
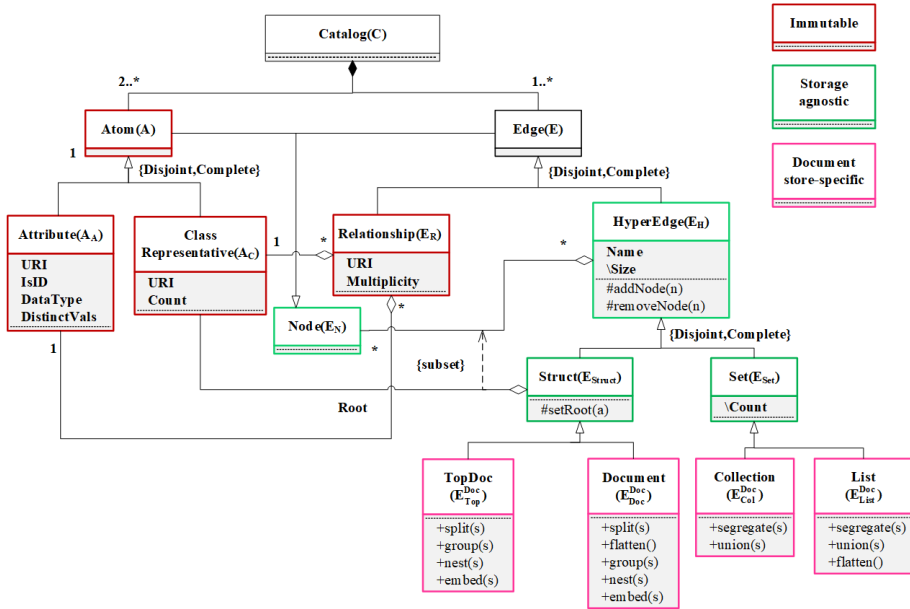
Fig. 5.4: Class diagram of the canonical model

Table 5.2: Hypergraph methods

Method	⟨⟨**preconditions**⟩⟩	⟨⟨**postconditions**⟩⟩
$Hyperedge(C, nodes :$ $Set\ of\ E_N)$		• $nodes \subseteq self$ • $self \in C$
$\backsim Hyperedge()$		• $self \notin self.C$ • $self.children@pre \subseteq$ $self.parent$
$E_H.addNode(n : E_N)$	• $self \notin n^+$	• $n \in self$
$E_H.removeNode(n : E_N)$	• $n \in self$	• $n \notin self$

each containing different class methods on the canonical model, getting more concrete as the specialization progresses. Operations at an abstraction level have access and use the operations at more generic levels above.

4.2 Storage-Agnostic Constructs

The main design construct of our model is a *Hyperedge* (E_H). By definition, a hyperedge is an edge that connects more than one vertex in a graph. We use the concept of generalized hypergraph where a hyperedge can also contain other hyperedges. *Hyperedges* (E_H) in our canonical model consists of a set of *Nodes* (E_N), which can be either *Atoms* (A), *Relationships* (E_R) or other *Hyperedges* (E_H).

In Table 5.2, we identify methods involving *Hyperedges*. These hypergraph operations are independent of the design constructs and the concrete data store, and we show them directly through pre- and post- conditions. Firstly, creating a *Hyperedge* defines a set of nodes in the catalog. On destroying it, all nodes inside are absorbed by its parent to ensure the validity of the catalog as per Definition 12. On adding a new node E_N to a *Hyperedge* E_H, it must happen that the *Hyperedge* E_H does not transitively contain itself to avoid cyclic references.

Now, we introduce two specialized E_Hs that are generic to any database system. E_{Struct} represents the structure of the stored elements in the database (i.e., a row in an RDBMS or a document in a document store) with a specific A_C identified as the *root* (primary attribute of the E_{Struct}). E_{Set} represents the collections of the database (table in an RDBMS, list or collection in a document store), which can contain different kinds of structures (in the case of document stores). We use the hypergraph operations to manipulate specialized hyperedges E_{Struct} and E_{Set}, defined at Defs. 10 and 11.

Definition 10

A *Struct* is a subclass of *Hyperedge* with a prominent node inside (called *root*, noted $O(self)$, which is an A_C), so that:

a) All the *Atoms* in a *Struct* must have a single path of relationships to its *root* which is also part of the *Struct*.
$$\forall a \in (self \cap \mathbb{A}) - O(self) : \exists! \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,a}\} \subseteq self$$

b) All the *roots* of the nested *Structs* inside a parent *Struct* must have a single path of relationships from the *root* of the parent, and this path must be inside the parent.
$$\forall s \in (self \cap \mathbb{E}_{Struct}) : O(self) = O(s) \vee \exists! \{E_R^{O(self),x_1}, \ldots., E_R^{x_n,O(s)}\} \subseteq self$$

c) All the *Sets* inside a parent *Struct* must contain a set of relationships that connect any *Atom* of the parent to the *root* of the child *Struct* or a child

Atom of the *Set*.

$\forall s \in (self \cap \mathbb{E}_{Set}), \forall t \in (s \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : \exists! \{E_R^{y,x_1}, \ldots, E_R^{x_n,z}\} \subseteq s \wedge y \in (self \cap \mathbb{A}) \wedge (t \in \mathbb{A} ? z = t : z = O(t))$

d) To make sure that there are no dangling relationships inside the *Struct*, all of the relationships inside it must be involved in a path that connects either the child *Atoms* or the child *Structs* to its *root*.

$\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : (\exists y \in (self \cap \mathbb{A}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,y}\} \subseteq self) \vee (\exists y \in (self \cap \mathbb{E}_{Struct}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,O(y)}\} \subseteq self)$

Definition 11

A *Set* is a subclass of *Hyperedge*, so that:

a) *Sets* cannot directly exist within another *Set* (i.e., they must be contained in an intermediate *Struct*).

$\forall h \in (self \cap \mathbb{E}_H) : h \in \mathbb{E}_{Struct}$

b) Together with invariant 1c, it guarantees that all the *Structs* inside a *Set* are connected to the parent *Struct* of the *Set* by a set of relationships in the *Set* itself. Finally, all the relationships inside the *Set* must be involved in the path that connects its child *structs* or *Atoms* to the parent *Struct* to avoid dangling relationships.

$\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : \exists A_C^{x_1} \in self.parent, \exists y \in (self \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : E_R^{a,b} \in \{E_R^{x_1,x_2}, \ldots, E_R^{x_n,z}\} \subseteq self \wedge (y \in \mathbb{A} ? z = y : z = O(y))$

Table 5.4 shows the methods of *Set* and *Struct* constructors. Even if for the sake of simplicity, they are not explicit there, we consider all properties in Defs. 10 and 11 are invariants for these methods and consequently guaranteed also in those at document store-specific level. Here, *super* referrs to the constructor of the super class.

Table 5.4: Struct and set methods

Method	Activity
$Struct(C, r : A_C, At : Set\ of\ A, Re : Set\ of\ E_R, Hy : Set\ of\ E_H, p : E_H)$	$super(C, \{r\} \cup At \cup Re \cup Hy)$ $self.setRoot(r)$ $p.addNode(self)$
$Set(C, Re : Set\ of\ E_R, Hy : Set\ of\ E_{Struct}, At : Set\ of\ A, p : E_{Struct})$	$super(C, Re \cup Hy \cup At)$ $p.addNode(self)$

4.3 Document Store-Specific Constructs

Document store-specific constructs are specializations of E_{Struct}s and E_{Set}s specific to document stores. We specifically identify the document structure at the top level as E_{Top}^{Doc} and the collection as E_{Col}^{Doc}. All other documents and nested lists are identified as E_{Doc}^{Doc} and E_{List}^{Doc} respectively. We now use the *Struct* and *Set* constructors to define the operators considering document store-specific constraints. The constraints and mappings we consider correspond to the following grammar:

$$C \implies E_{Col}^{Doc} +,$$

$$E_{Col}^{Doc} \implies E_{Top}^{Doc} +$$

$$E_{Top}^{Doc} \implies A_C(A \mid E_{List}^{Doc} \mid E_{Doc}^{Doc})*$$

$$E_{List}^{Doc} \implies E_R {}^+ (E_{Doc}^{Doc} \mid A)^+$$

$$E_{Doc}^{Doc} \implies A_C(A \mid E_{List}^{Doc} \mid E_{Doc}^{Doc})*$$

We define the constructors of the data store-specific structures considering these production rules, as shown in Table 5.6.

Table 5.6: Document store-specific constructor methods

Method	Activity
$Document(C, r : A_C, Re : Set\ of\ E_R, At : Set\ of\ A,$ $Do : Set\ of\ E_{Doc}^{Doc}, Li : Set\ of\ E_{List}^{Doc}, p : (E_{List}^{Doc} \cup E_{Struct}^{Doc}))$	$super(C, r, At, Re, Do \quad \cup$ $Li, p)$
$TopDoc(C, r : A_C, Re : Set\ of\ E_R, At : Set\ of\ A,$ $Do : Set\ of\ E_{Doc}^{Doc}, Li : Set\ of\ E_{List}^{Doc}, p : E_{Col}^{Doc})$	$super(C, r, At, Re, Do \quad \cup$ $Li, p)$
$Collection(C, Do : Set\ of\ E_{Top}^{Doc})$	$super(C, \varnothing, Do, \varnothing, \varnothing)$
$List(C, Re : Set\ of\ E_R, Do : Set\ of\ E_{Doc}^{Doc}, At : Set\ of\ A, p :$ $E_{Struct}^{Doc})$	$super(C, Re, Do, At, p)$

Finally, we define a valid design using these constructs, which guarantees that we do not lose any information provided in the input ER diagram.

Definition 12
A design D is a set of collection *Hyperedges* and is *valid* if it contains all the *Atoms* and *Relationships* in the closure of at least one of its collection *Hyperedges*. **Formally**: $\forall x \in (\mathbb{A} \cup \mathbb{E}_R) : \exists E_{Col}^{Doc} \in D \wedge x \in E_{Col}^{Doc+}$.

Generating arbitrary constructs cannot guarantee a valid design as per Def. 12. Thus, when using these document store-specific constructors, the validity must be explicitly enforced.

5 Design Processes Over the Canonical Model

Now that we formally defined our canonical model to represent document store data designs, we can use it in our shotgun hill-climbing approach introduced with Algorithm 6 to find the near-optimal design. To achieve this, we need to create a initial state with a random design (line 3) and apply transformations to generate neighboring designs (line 5). Thus, we introduce two design processes over the canonical model: random design generation and design transformation each corresponding to a subsection.

5.1 Random Design Generation

The key concept used in the random design generator is generating connected components (i.e., subgraphs) of the immutable graph until all the *Atoms* and *Relationships* are in one of these components. This ensures that none of the input ER diagram information is lost, adhering to a valid design. Each connected component represents then a collection in the document store schema. Algorithm 7 is responsible for generating a random design together with the aid of Algorithm 8 to make the design structure decisions. The main requirement behind these algorithms is to make the relationship storage choices randomly. For the simplicity of the algorithms, we omit the flattened representation in the random generation process. Thus, a relationship can be referred, nested, or skipped (in the case of chained relationships). In our running example, the *region* collection can have *bids* embedded or referred without storing the *user* information. However, the relationship between the *users* and *bids* must be stored in another collection (i.e., *user* collection referring/embedding *bids*) to ensure no information is lost from the original ER diagram adhering to the validity of a design.

Definition 13
Each connected component (*Comp*) is represented as a tree of *Cnode*s, each representing a relationship and its storage choice, except for the root (where the *from* and the relationship are empty) and the leaves (where children are empty). Thus, each *Cnode* of the tree contains five elements: 1. the *from* A_C, 2. the *to* A_C, 3. a relationship E_R that connects the parent and the child, 4. the representation (i.e., nest, refer, or skip) of the E_R connecting them, or an indicator of being the topmost element of the component identified as the *ROOT*, and 5. the set of child *Cnode*s. **Formally**: $Comp = Cnode\langle from, to, rel, (NEST \mid REF \mid SKIP \mid ROOT), \{Cnode\}\rangle$ s.t : $from, to \in \mathbb{A}_C \wedge rel = E_R^{from,to}$

Algorithm 7 keeps track of unused A_Cs and E_Rs that connect two A_Cs (lines 1 and 2) and maintains a list of E_Rs to be explored and a list of connected

---

Algorithm 7 Main Algorithm

Input: *graph G* containing *Atoms* and *Relationships*
1: $allA_Cs \leftarrow G.getA_Cs()$ ▷ *all A_Cs in G*
2: $allE_Rs \leftarrow G.getE_R^{A_C,A_C}s()$ ▷ *all E_Rs that connect two A_Cs in G*
3: $E_Rs \leftarrow newList\langle E_R \rangle()$ ▷ *E_Rs to be explored*
4: $comps \leftarrow List\langle Comp \rangle()$ ▷ *list of connected components*
5: **repeat**
6:   **if** $E_Rs \neq \varnothing$ **then** ▷ *connected E_R to an explored one*
7:     $next \leftarrow E_Rs.remove(RandInt(E_Rs.size))$
8:     $allE_Rs.remove(next)$
9:   **else** ▷ *pick a new random unexplored E_R*
10:     $next \leftarrow allE_Rs.remove(RandInt(allE_Rs.size))$
11:   $[root, comps, allE_Rs, allA_Cs] \leftarrow choose(next, comps, allE_Rs, allA_Cs)$
12:   $E_Rs.addAll(G.getUnusedE_R^{A_C,A_C}(next.get(root)))$ ▷ *add connected E_Rs to explore*
13: **until** $allE_Rs = \varnothing \wedge E_Rs = \varnothing$
14: **for each** $atom \in allA_Cs$ **do** ▷ *make remaining A_Cs into new Comps*
15:   $col \leftarrow newCnode(null, atom, null, ROOT)$
16:   $comps.put(col)$
17: **for each** $tree \in comps$ **do** ▷ *transform Comps into E_Hs*
18:   $buildHyperedge(tree, G, allA_Cs)$

---

components (lines 3 and 4). The generation process is initialized by randomly picking one of the available relationships. This can be from the list of relationships to explore (lines 6–8), if any, or from all unexplored relationships (lines 9–10). This chosen E_R will create a new connected component or extend an existing one depending on the current components using Algorithm 8, which also returns the *root* of this connected component (e.g., assume that it is the U_ID). Then, in line 13, we take all the unused E_Rs that connect other A_Cs to the picked *root* (e.g., $E_R^{U\_ID,R\_ID}$, $E_R^{U\_ID,C\_ID}$) expanding the connected edges to be explored in. We continue this procedure until all the E_Rs have been used for the connected components. Then, we generate new connected components for all the remaining A_Cs that are not used in any of the existing connected components (lines 15–18). Finally, in lines 19–21, we build the E_{Col}s corresponding to the connected components in *G* by transforming the *Comp*s into corresponding E_Hs. Due to space limitations, we introduce this transformation in Appendix E as the procedure is purely technical.

Algorithm 8 is responsible for determining the direction and the representing of a particular E_R chosen by Algorithm 7. The inputs consist of a chosen E_R, the list of currently connected components, and the list of all unused E_Rs sent by Algorithm 7. We determine the direction of the relationship randomly in line 2, which determines the from A_C of the new *Cnode*. Next, we go through the list of currently connected components (line 5), doing a post-order traversal (line 6) to determine if one of the currently connected components can be extended with the new *Cnode* as a child. This is possible only if there is a *Cnode* with *to* as the *from* of the new *Cnode* and *from* is not the *to* of the

---

Algorithm 8 Choose Algorithm

Input: E_R *rel*, List⟨*Comp*⟩ *comps*, List *allE$_R$s*, List *allA$_C$s*
 1: *pSkip* ← 0.25
 2: *opChoice* ← *flip(LEFT | RIGHT | BOTH)*
 3: **if** *opChoice* = *LEFT* **then** ▷ need the choice otherwise we will always grow components if its connected
 4:   *canExtend* ← *false*
 5:   **for each** *tree* ∈ *comps* **do**
 6:     **for each** *node* ∈ *tree.postOrderTraversal()* **do**
 7:       **if** *node.to* = *rel*(0) ∧ *node.from* ≠ *rel*(1) ∧ (∄ *n* ∈ *node.children* s.t *n.rel* = *rel*) **then** ▷ find a node with rel(0) as "to" which is not connected by rel(1) and has no children or none of the children has used rel
 8:         *node.addChild(rel(0), rel(1), rel, flip(REF | NEST))*   ▷ add new child to the tree
 9:         *allA$_C$s.remove(rel(0))*
10:         **if** *randomDouble()* < *pSkip* **then**   ▷ skip with probability
11:          *node.type* = *SKIP*
12:          *allE$_R$s.add(node.rel)*   ▷ add the relationship back to the pool
13:         *Connectionfound* ← *true*
14:         **break**   ▷ only add the node to the tree and stop the iteration within the tree
15:     **if** *Connectionfound* **then**   ▷ stop looking in more trees if the node is already added
16:       **break**
17:   **if** !*Connectionfound* **then**   ▷ new component
18:     *root* ← *newCnode(null, rel(0), null, ROOT)*
19:     *root.addChild(newCnode(rel(0), rel(1), rel, flip(REF | NEST)))*
20:     *comps.put(root)*
21: **else if** *opChoice* = *RIGHT* **then**
22:   same as above swap 0 and 1
23: **else if** *opChoice* = *BOTH* **then**
24:   *do opChoice LEFT and RIGHT (nest on both ends or refer on both ends)*
25: *opChoice* = *BOTH ? flip(LEFT | RIGHT) : opChoice*
26: **return** *opChoice, comps, allE$_R$s, allA$_C$s*

---

new *Cnode* and the existing *Cnode* does not use the selected E_R in any of the children. Once we find the location, we update that connected component with the new *Cnode* with a random choice of reference or nesting (line 8).

In the case of a chain of relationships between two A_C within a connected component, it is possible to skip some of them in the final document representation. For example, we can store the list of *bids* of a particular *region* without the *user*'s details even though the *user* is related to the *bid*. The skip choice enables such design decisions. We introduce a probability to skip a relationship in line 1 and change the *Cnode* type to *SKIP* and add the E_R back to the pool of unused E_Rs (to ensure that particular relationship information is not lost) with that probability in lines 10–13. We add the new *Cnode* to only one connected component and to a single particular branch (lines 14–21). If no component can be updated, we make a new connected component with the new *Cnode* as the *ROOT* as shown in lines 22–31. Finally, we return the parent side of the E_R that we used back to Algorithm 7, in the case of both

sides, we randomly return one of the A_Cs (lines 31–33).

The above choices are carried out until all the entities and relationships belong to at least one of the connected components. Finally, each of the components is represented as a document store collection. These initial designs do not contain heterogeneous collections or lists, yet, since we initially ignore the choice of flattening and only use the nested option for structuring concerning the options in Fig. 5.1. This decision reduces the complexity of the random generation and the number of starting schemas. However, we introduce this through design transformations to ensure that we do not lose certain designs in the process.

Let us consider the running example of `products` and `comments` from RUBiS and also include `users` to have a complex scenario to generate a random design. Let's assume we picked $E_R^{P\_ID,C\_ID}$ as the first E_R in Algorithm 7 line 5. Next, in Algorithm 8 we got *LEFT* as the random *opChoice* in line 2. Since there are no existing *Comp*s we move to line 21 and create a new *Comp* to the *comps* list with $Cnode\langle P\_ID, null, null, ROOT\rangle$ as the root and a single child $Cnode\langle P\_ID, C\_ID, E_R^{P\_ID,C\_D}, NEST\rangle$ if we got *NEST* option. Now, coming back to line 7 in Algorithm 7, we have $E_R^{P\_ID,U\_D}$ in E_Rs as the only unused $E_R^{A_C,A_C}$ connected to *P_ID*. Here, at line 13 we pick this E_R and go back to Algorithm 8. Let's assume that we got *RIGHT* as the *opChoice*. We can't extend the previous *Comp* that we made as it doesn't satisfy the extensible criteria. Thus, we create a new *Comp* with $Cnode\langle U\_ID, null, null, ROOT\rangle$ as the root and $Cnode\langle U\_ID, P\_ID, E_R^{P\_ID,U\_D}, REF\rangle$ as its child.

Finally, similarly, if we assume the last remaining E_R between *U_ID* and *C_ID* got *BOTH* and *REF*, both the *Comp*s of the *product* and the *user* will be extended with $Cnode\langle C\_ID, U\_ID, E_R^{U\_ID,C\_D}, REF\rangle$ and $Cnode\langle U\_ID, C\_ID, E_R^{U\_ID,C\_D}, REF\rangle$ respectively. Now that we have exhausted all E_Rs, we build the E_Hs that represent the corresponding design (algorithm in E). In this case, the design is `products` embedding `comments` in one collection with `comments` having a reference to the `users` and a second collection of `users` with a reference to both `comments` and `products`.

5.2 Design transformations

In order to generate neighboring designs to a given **valid** design, we introduce now seven public methods specific for document stores at the corresponding specific design constructs. A detailed formalization of these transformation rules is available in Appendix F.

1. **Union** : Merges two sibling E_{Set}^{Doc} into one.
2. **Segregate**: Separates a E_{Struct}^{Doc} from inside a heterogeneous E_{Set}^{Doc} into a new independent E_{Set}^{Doc}.

3. **Embed**: Embeds E_{Struct}^{Doc} into another sibling E_{Struct}^{Doc} that have a path of E_Rs.
4. **Split** Separates a E_{Struct}^{Doc} into two under a given partition of its elements.
5. **Nest** Creates a new E_{Struct}^{Doc} within an existing E_{Struct}^{Doc}, given a subset of its elements.
6. **Group** Creates a new heterogeneous E_{Set}^{Doc} containing two E_{Struct}^{Doc}s.
7. **Flatten** Removes an E_H^{Doc} and let its parent absorb the content.
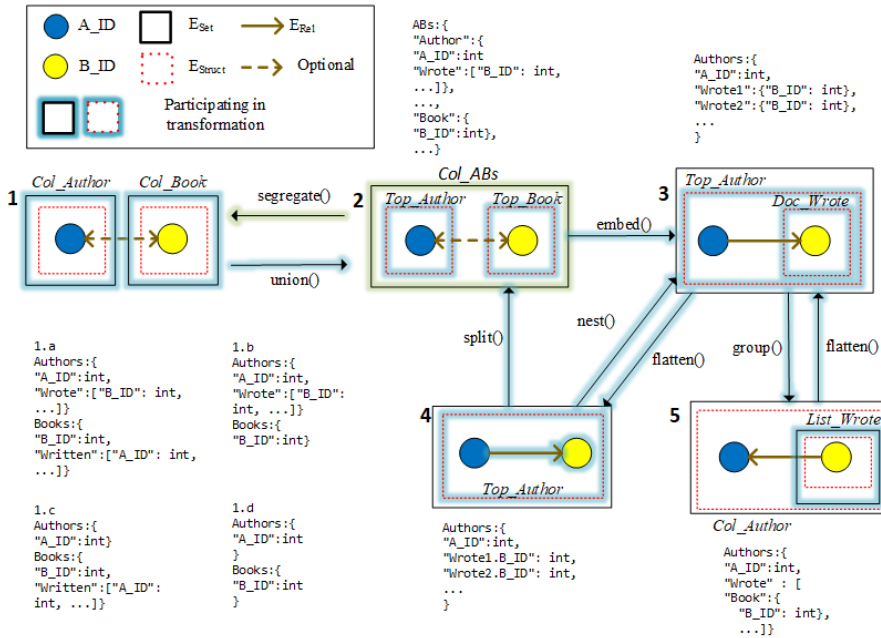
Fig. 5.5: Sketch of schema transformations in document stores using transformation rules

Let us retake the running example storing *products* and *comments* to illustrate the transformations. Figure 5.5 shows eight different designs (Design 1.a-1.d, 2–5) that can be conceived and sketches the transitions between them using the transformations in Table F.1 (some symmetries are not shown for the sake of simplicity). Since, only A_Cs are relevant for the transformations, namely *P_ID* and *C_ID*, only these are shown to keep the figure clean. Nevertheless, we assume that the attributes of any A_C are always attached to it (e.g., *P_NAME* will always be in the same hyperedge as *P_ID*). Glowing lines indicate the hyperedges that participate in the transformations that follow each design. Additionally, the hyperedge where the relationship belongs to is assumed to be that of the tail of the arrow (i.e., in Design 3, E_{Rel} belongs to the E_{Top}^{Doc} containing *P_ID*; in Design 5, it is the E_{Set}^{Doc} of *C_ID*; and in Design 4, that of the A_C). Notice that in Designs 1 and 2, there is a double-head dashed

arrow, which means that, for *segregation* and *union*, its existence is optional and also can be at either one or other side. The optionality of the E_R in `Design 1` implies that the reference between the collections can reside on either side, which gives rise to four alternative schemas, namely references on both collections (`1.a`), one collection (`1.b` and `1.c`), or none (`1.d`). Even though `1.d` is an invalid design in this particular scenario as we break the invariant in Def. 12, we use it for illustration as it is possible to have disconnected collections in other cases.

Let's assume that we start with `Design 1.b` and follow the transformations as illustrated in Fig. 5.5. According to Table F.1, in order to unite two E_{Set}^{Doc}s, they need to share the same parent. Thus, if we call $E_{Col\_Prod}^{Doc}.union(E_{Col\_Com}^{Doc})$, firstly, the $E_{Col\_Com}^{Doc}$ is added to the source $E_{Col\_Prod}^{Doc}$), followed by the removal of the absorbed collection from the catalog. Finally, $E_{Col\_Com}^{Doc}$ is disposed, leaving its children in the new parent $E_{Col\_Prod}^{Doc}$, as represented in `Design 2`, where `comments` and `products` are in the same collection (notice that in this case only `products` references `comments`).

To *embed* a E_{Doc}^{Doc} into another, they must have the same parent, and their roots must be the same, or there must be a path between them. We exemplify this by $E_{Top\_Prod}^{Doc}.embed(E_{Top\_Com}^{Doc})$, which moves $E_{Top\_Com}^{Doc}$ inside $E_{Top\_Prod}^{Doc}$ as in `Design 3`. The result is that for each *product*, there will be several `comments` in the form of a flattened list, and each document will carry the name of the relationship suffixed by a counter. Implementation-wise, we rename the embedded E_{Doc}^{Doc} with the name of the relationship followed by a counter, as shown in the JSON.

In order to *flatten* an E_H, its parent must be a E_{Doc}^{Doc} (or E_{Top}^{Doc}) to ensure a correct design (i.e., *sets* directly inside *sets* shouldn't be allowed without a *struct* hyperedge in between). If so, the E_H is simply disposed of, letting its children to be absorbed by its parent. By applying $E_{Doc\_Com}^{Doc}.Flatten()$ to `Design 3`, we obtain `Design 4`, where the `comments` are directly embedded inside each `product` without an enclosing `comments` document. However, the prefix `gotCom` followed by the counter still needs to be included in the JSON to distinguish different instances.

The *group* transformation creates a E_{Set}^{Doc} around a child E_{Doc}^{Doc}(s) within another E_{Struct}^{Doc}. Both the child E_{Doc}^{Doc} and the defining path of E_Rs to it must already be inside the original E_{Struct}^{Doc}. By $E_{Top\_Prod}^{Doc}.group(E_R^{A\_ID,B\_ID}, E_{Struct\_gotCom}^{Doc})$ in `Design 3`, we obtain `Design 5` which embeds the `comments` inside each `product` document. Afterwards, the child E_{Doc}^{Doc} and the E_Rs that are no longer used in any path to the remaining children, are removed from the original E_{Struct}^{Doc}. The transformation results in a new *List* inside the JSON containing the data in the child E_{Doc}^{Doc} which is linked to the container by the path of E_Rs. To revert this transformation, we can call *flatten* on the created E_{List}^{Doc} moving us back to `Design 3`.

The *nest* transformation creates a new embedded E_{Doc}^{Doc} inside another E_{Doc}^{Doc} (or E_{Top}^{Doc}). It is necessary to use the same parameters to create any document, but they must all be contained within the original E_{Doc}^{Doc}. After creating the embedded E_{Doc}^{Doc}, all its nodes are removed from the original E_{Doc}^{Doc}, except the E_Rs needed to keep it connected. Thus, *nest* does not allow keep redundant As or E_Hs in the original E_{Doc}^{Doc}; if redundancy is required, a *split* transformation needs to be done beforehand. By calling *nest* in `Design 4` to nest the `comment`, we can obtain back `Design 3`.

The *split* transformation allows creating a sibling independent E_{Doc}^{Doc} with all or part of the content of another one, inside its parent (this can result in E_{Top}^{Doc} instead of E_{Doc}^{Doc} if the parent is actually a collection). Some or all of the nodes in the new E_{Doc}^{Doc} can be removed from the original. Thus, the parameters of the transformation (which must all be inside the original E_{Doc}^{Doc}) are both the contents of the new E_{Doc}^{Doc} and which elements out of these are removed from the original. As a result, both E_{Struct}^{Doc}s either share the same *root*, or there will be a path between the *root* of the original E_{Doc}^{Doc} and that of the new one. Notice that parameters must be so that both resulting E_{Doc}^{Doc} satisfy the invariants, but there is still freedom to determine whether this path is at the end contained in the original, new, or both E_{Struct}^{Doc}s. In our example, by splitting the *gotCom* from `product` in `Design 4`, we can obtain back `Design 2` (the dashed arrow depending on the path the parameters determine).

Finally, the *segregate* transformation divides an E_{List}^{Doc} (or E_{Col}^{Doc}) containing multiple E_{Doc}^{Doc}s or As into two. The only condition is that the segregated nodes must be already contained in the original E_{List}^{Doc}. After the transformation, the E_Rs that are no longer used by any of the children inside the original E_{List}^{Doc} are removed from it together with the segregated E_{List}^{Doc}. As a result, the corresponding JSON will contain two independent *list*s (or collections if we are talking about E_{Col}^{Doc}) with E_{Doc}^{Doc} or A, whose contents will depend on the path to E_{Doc}^{Doc} or A from the parent of the original E_{Set}^{Doc}. By calling $E_{Col\_ABs}^{Doc}.segregate(E_{Top\_Com}^{Doc})$ on `Design 2`, we can obtain `Design 1.a/b/c`.

Not having any normal forms or design algorithms to use as a baseline for comparison (like in RDBMS), we validate our document store design transformations against existing rule-based patterns. Luckily, MongoDB ones are publicly available.[4] Hence, we showcase our canonical hypergraph representation with MongoDB patterns and analyze how to implement them using a sequence of transformations in Appendix G.

---

[4]https://www.mongodb.com/blog/post/building-with-patterns-a-summary

6   Experiments

We implemented our approach in a system called DocDesign 2.0(demo in Appendix B), using HypergraphDB[5] to store the canonical model, AIMA3e[6] for optimization using Java together with query cost estimator using Gekko[7] written in Python. In this section, we present its experimental evaluation, which is twofold. First, we analyze the quality of the designs generated (Sec. 6.1); and second, we evaluate the scalability of DocDesign 2.0 when the complexity of the entities and their relationships increase (Sec. 6.2).

6.1   Quality of the Design

To evaluate the quality of the generated designs, we use DocDesign 2.0 on the running example of the RUBiS benchmark [25] (see Fig. 5.3). We prioritized query performance (0.7) followed by the storage space (0.2), depth of documents (0.05) and heterogeneity (0.05) together with the number of non-improving iterations (N in Algorithm,6) of 10. The generated design was then compared against the ones presented by the DBSR framework [95]. We used a higher weight for the query performance for our design to be comparable with DBSR while trying to improve storage size and this configuration will be the typical ones that one would use where query performance is the most important aspect. Moreover, we have evaluated DocDesign 2.0's capability to generate alternate designs depending on the weights provided by the user in our previous work as shown in Appendix B.

We extended the DBSR evaluation benchmark to include the design suggested by DocDesign 2.0. All the queries were executed using MongoDB Java driver 3.8.2. We used a single instance of MongoDB Community Edition version 4.2 running on Intel Xeon E5520, 24 GB of RAM with Debian 4.9 as the experimental setup. First, we generated data consisting of 1 million users, 10 million items, 5 million bids, 10 million comments, 3 million bids, and 4 regions. Then, the same data was stored in the alternative designs suggested by DBSR and DocDesign 2.0. Next, 1 million random queries were executed, consisting of 11 different queries with their respective probabilities. Finally, we measured the throughput of each of the alternate designs.

Table 5.8, shows the schemas of the designs generated by each of the systems. Designs generated by DBSR are based on joining the collections. Thus, the results can be controlled through the number of collections in the final design. In this scenario, we present the solutions of both 3 and 5 collections for comparison. It is clear that the designs generated by DBSR contain multiple redundancies, especially on the `product`. On the contrary,

---

[5]`http://www.hypergraphdb.org`
[6]`https://github.com/aimacode/aima-java`
[7]`https://gekko.readthedocs.io/en/latest`

Table 5.8: Final designs generated by DocDesign 2.0 and DBSR

DocDesign 2.0	DBSR (3 col)	DBSR (5 col)
```"USER": { "U_ID": int(4), "U_F_NAME": varchar(20), "REGION": { "R_ID": int(4), "R_NAME": varchar(10) } } "PRODUCT": { "P_ID": int(4), "P_TITLE": varchar(10), "BIDS": [{ "B_ID": int(4), "B_PRICE": int, "U_ID": int(4) }], "COMMENTS": [{ "C_ID": int(4), "C_TITLE": varchar(20), "U_ID": int(4) }], "U_ID": int(4) }```	```"BID-PRODUCT":{ "B_ID": int(4), "B_PRICE": int, "U_ID": int(4), "PRODUCT": { "P_ID": int(4), "P_TITLE": varchar(10) }} "PRODUCT-SELLER-REGION": { "P_ID": int(4), "P_TITLE": varchar(10), "USER": { "U_ID": int(4), "U_F_NAME": varchar(20), "REGION": { "R_ID": int(4), "R_NAME": varchar(10) }}} "PRODUCT-COMMENTS": { "P_ID": int(4), "P_TITLE": varchar(10), "COMMENTS": [{ "C_ID": int(4), "C_TITLE": varchar(20), "U_ID": int(4) }]}```	```"PRODUCT-SELLER": { "P_ID": int(4), "P_TITLE": varchar(10), "USER": { "U_ID": int(4), "U_F_NAME": varchar(20) } } "PRODUCT-BIDS": { "P_ID": int(4), "P_TITLE": varchar(10), "BIDS": [{ "B_ID": int(4), "B_PRICE": int, "U_ID": int(4) }] } + DBSR (3 col)```

DocDesign 2.0 leans more towards having references and only embedding the region within the user. From an end-user perspective, the design from DocDesign 2.0 is much cleaner and has less maintenance compared to the ones of DBSR. Moreover, doing any updates will be pretty expensive in DBSR designs as it will involve updating multiple documents in different collections. The documents used in DBSR contain rather small documents and are unrealistic for a real-world scenario. Because of this, we conduct the same experiment with increased document sizes by converting the integer identifiers into MongoDB UUID fields (24 bytes instead of only 4) and increasing the description attribute size.

Table 5.9: Performance comparison of the original dataset

	Runtime (ms)					
	Min	**Q1**	**Mean**	**Media**	**Q3**	**Max**
DocDesign 2.0	70	512	733	604	726	80639
DBSR (3 col)	228	470	760	575	1074	80063
DBSR (5 col)	262	523	787	585	2067	76259
DBSR (5 col agg.)	253	542	1304	607	2471	75583

Table 5.9 depicts the summary of the throughput values obtained for the 1 million random queries. The five-collection design of DBSR was also evaluated using the MongoDB aggregation framework. However, this has the worst performance out of all the designs. Since DBSR can answer most of the queries

Table 5.10: Performance comparison of the realistic dataset

	Runtime (ms)					
	Min	**Q1**	**Mean**	**Media**	**Q3**	**Max**
DocDesign 2.0	650	1121	1842	1268	2003	88869
DBSR (3 col)	519	1292	2629	1782	4091	115290
DBSR (5 col)	639	1558	2683	1866	7317	97611
DBSR (5 col agg.)	619	1555	4147	1814	9587	108083

with a single collection, the minimum runtime is lower as it is the time taken to retrieve the smallest cached document. The min runtime per query is highest on DocDesign 2.0 in both original and the large document experiment due to the smallest document being larger than the ones of DBSR (38 bytes *user* in DocDesign 2.0 vs 26-byte *bid-product* in DBSR).

However, DBSR loses the advantage when looking at the other statistics. Especially, the design with five collections falls quite behind; this could be mainly because the collections are competing on the available memory and a higher proportion of documents need to be fetched from the disk rather than the memory. In the original experiment, DocDesign 2.0 has a slight advantage in the mean and a higher one at Q3. However, once we increase the document size to be more realistic (Table 5.10), DBSR always falls behind the performance of DocDesign 2.0. Overall DocDesign 2.0 has better average performance and less skewed results by looking at the inter-quartile range, especially with larger document sizes. The maximum value of almost all the designs is quite similar because these queries involve fetching documents from the disk in the event of a cache miss. Both DBSR and DocDesign 2.0 did not generate heterogeneous designs as the optimal ones. In the case of DBSR, this is never considered, and in DocDesign 2.0, since we are optimizing for query performance, the designs with heterogeneous collections become non-optimal. DocDesign 2.0 designs are less complex with a maximum depth of one level of nesting, while DBSR has two levels of nesting in the case of product-seller-region collection. When it comes to the total storage space, DBSR required 7GB in the three collections and 12GB in the case of five collections (due to the high redundancy in the generated designs of DBSR) as opposed to DocDesign 2.0 that required only 6.5GB. Thus, it is clear that DocDesign 2.0 is capable of generating document store designs with better performance and superior space optimization.

6.2 Scalability of the Approach

We tested the scalability of our approach in DocDesign 2.0 as it is an essential factor in larger use cases. In order to achieve this, we measured how many

iterations it would take to get the near-optimal solution when the number of entities and relationships grow. For that purpose, we needed random ER diagrams with a varying number of entities as well as differentiate the topologies for each number of entities. This eliminates any opportunity of the topology affecting the final outcome of the experiment. We generated a synthetic ER diagram since it is impossible to find real-world ER diagrams that satisfy this requirement. Thus, we used gMark [14] (a graph instance and query workload generator) to create random ER topologies. We used a pre-defined number of entities and Gaussian distribution ($\mu = 0.31$ and $\sigma = 1$) of the relationships to generate the gMark graph and transformed it into our immutable graph.
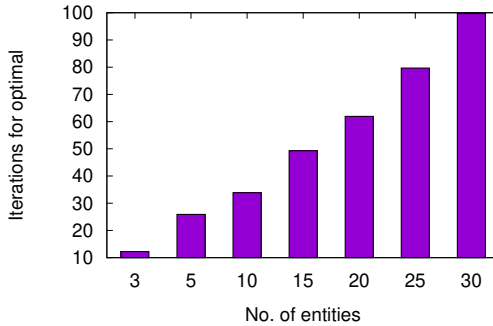
Fig. 5.6: Scalability of DocDesign 2.0 with number of entities

Next, we generated as many random queries as the number of entities with equal probabilities. Finally, we used these values as input to DocDesign 2.0 to find the optimal design. We measured the number of iterations until there is no improvement for the next 100, assuming that this would give us the closest to the optimal solution. For each number of entities (experiment), we generated 10 random topologies, and for each topology ran DocDesign 2.0 10 times to obtain the average. As shown in Fig. 5.6, the number of shots required increases linearly as the number of entities grows. We appreciate that it requires around 100 (exactly 99.68) iterations to completely stabilize the design with 30 entities. However, in reality, one can obtain a near-optimal solution with much fewer iterations.

Figure 5.7 shows the evolution of the loss function that DocDesign 2.0 makes as the iterations (shots) progress in the experiment with 30 entities. The average and the standard deviation are of the 100 instances (10 topologies 10 runs) mentioned before. The initial shots make significant improvements fast, but as the shots progress, the improvement is minimal. Thus, we can safely assume that it is possible to obtain a near-optimal solution for this problem in around 15 iterations (i.e. N=15 in Algorithm 6).
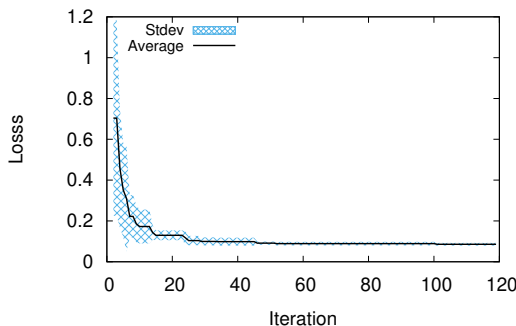
Fig. 5.7: Improvement over the number of shots

In summary:
- The design generated by using weights that represent typical requirement of optimizing queries with consideration on storage space outperforms the design generated by DBSR.
- DocDesign 2.0's design was not only performant, but also require less storage space.
- DocDesign 2.0's multicriteria-based approach provides flexibility for the end users to optimize according to their requirements.
- The non-improving iterations (N) determines the optimality of the design. Through a synthetic workload with 30 entities with varying topology, we concluded that $N = 15$ would already generate a near-optimal solution.

6.3 Threats to Validity

The first threat to validity is the possible bias on evaluating the quality of the design. The authors perceived having large collections with redundant data as a negative property of a design in comparison to DBSR. Nevertheless, we assume that this is the perspective of a traditional relational database designer in most of the cases.

Secondly, another threat to validity is on the testing where we increased the size of the dataset based on the fact that the design proposed by DocDesign 2.0 was not out performing on all the aspects (Q1 and Max). However, in all the other aspects DocDesign 2.0 outperformed DBSR.

Another threat to validity is that DocDesign 2.0 only provides a pareto-optimal solution and we do not know what is the best solution to a given problem. This is only possible through testing all the possible implementations through an exhaustive search. This is a typical property of optimization problems. Thus,, we used DBSR as a baseline to compare our solution instead.

Finally, an external threat to validity is the use of synthetic data to test the scalability of the approach instead of real world data. We decided to use synthetic data because we have full control over the scale factors of the input ER diagrams. Finding real world data with such specific requirements is non-viable.

Chapter 6

Conclusions and Future Directions

1 Conclusions

In this doctoral dissertation, we have presented our approach for automated data design on document stores. The main goal of this thesis is to provide a novel, multicriteria-based approach in the context of database design, particularly for document stores. To this end, we introduced the problem of database design as a complex problem that grows exponentially with the number of relationships in an entity-relationship diagram. Moreover, apart from the apparent query performance, the designs affect multiple aspects such as storage size, depth of the stored documents, and heterogeneity among the documents stored in a collection/nested lists. Consequently, multicriteria optimisation seems to be the ideal candidate, proven its successful track record on similar problems in diverse domains. Nevertheless, a fully automated data design optimisation for document stores poses several challenges. The lack of precise design methodology, formal representation of NoSQL systems and their designs, and a proper cost model in query processing (compared to RDBMS). Focusing on these challenges, we proposed a novel approach to automated database design for document stores. In what follows, we first summarise the contributions presented in Chapters 2–5 and finally conclude the thesis.

Chapter 2 explored the impact of using JSON as a semi-structured storage alternative to traditional RDBMS. The chapter began with identifying the motive behind choosing document stores as a storage alternative. Document stores are motivated by the lack of flexibility in the rigid structure as well as the impedance mismatch problem of RDBMSs. In addition, the semi-structured

nature of document stores makes them ideal candidates for rapid prototyping approaches. Nevertheless, the effect on alternative design suggestions provided by document stores has not been systematically evaluated. Thus, we identified six data representational differences under three categories between document stores and RDBMSs. Next, we empirically quantified the impact of these design choices. Finally, we experimentally evaluated the effect of these different designs in terms of storage requirements and query performance.

Chapter 3 introduced a hypergraph-based canonical model, one of the crucial components for achieving the overall goal of automated database design. This canonical model can represent not only document store designs but also RDBMS and other NoSQL storage systems. The model consisted of three levels: *Immutable* level contained the information on stored entities and their relationships represented in a simple graph, *Storage agnostic* level identified common design constructs to any data store as hyperedges built on top of the immutable graph. Finally, *Document store-specific* constructs introduced specialised hyperedges representing design constructs unique to document stores. Apart from this, we introduced grammars specific to different data stores that formalised their storage constructs through constraints. Using these, we presented algorithms for the generation of data store-specific queries automatically from the ones issued on the immutable graph. We also introduced several algorithms to calculate essential attributes of a data store, such as storage size and complexity of the stored documents.

Chapter 4 introduced a cost model for random access queries for document stores under a constant workload. Document stores rely on primitive approaches in determining query plans, such as parallel execution of all possible query plans (with heuristics to cut down the exploration) and choosing the winning one and caching it to be used on similar subsequent queries or forcing the end-user to indicate the query plan. However, to compare alternate designs, we needed a formal cost model to estimate the query cost. Thus, we introduced a disk access-based cost model similar to the ones of RDBMSs, focusing on memory usage as document stores rely heavily on caching to optimise the query performance. First, we introduced a cost model consisting of a generic component together with a document store implementation-specific components. Each of these specific components was a series of formulas for different techniques on memory mapping, memory associativity, and cache eviction policies. Then, the relevant component could be picked depending on the document store implementation and applied together with the generic cost component to obtain the final memory usage estimation. Once we had the memory estimation for each of the collections and the indexes, we could estimate the relative query cost for any random access query under the constant workload. This is the first cost model for document stores to the best of our knowledge.

Finally, in Chapter 5, we introduced our final automated data design solu-

tion for document stores. We used the canonical model to represent the search space of alternate designs. Our first prototype DocDesign 1.0 (Appendix A), allowed the end-users to manually input alternate designs together with a query workload and compare them in terms of storage space obtained through the canonical model and the relative query performance calculated through the cost model in Chapter 4. Then, we introduced a formalised palette of transformations to transform a design into its neighbouring designs without loss of information while adhering to the design constraints of a document store (making sure the new design is a valid one). Next, using these transformations with the shotgun hill-climbing algorithm, we could produce the Pareto-optimal design for a given use case (ER diagram and workload) and weights for the optimisation criteria. We added two additional optimisation parameters, the depth of first level documents and heterogeneity of documents in a collection/nested list. The final prototype, DocDesign 2.0 (Appendix B), could generate optimal designs better than the compared document store data design solutions purely based on query optimisation in terms of performance, storage space, and complexity of the design.

2 Future Directions

The proposed automated schema design process in this thesis opens the door to many interesting future directions to extend our current work.

The canonical model algorithms were mainly applied to document stores to achieve our overall automated database design. Nevertheless, it would be beneficial to extend the capabilities to other data stores. Most of the introduced algorithms can be easily applied to other data stores such as RDBMS since they are more restrictive on the design compared to document stores. Extending the design transformation rules will make it possible to develop the automated schema design for other data stores with ease. We also see the potential of eliminating the limitations of the canonical model, such as representing specialization.

We foresee the extension of the proposed cost model for document stores with additional document store implementations. The current cost model only supports random access queries under the assumption that all queries are using an index. Thus, modeling the cache behaviour on full collection scans will broaden the usability of the cost model as it is likely to have full scans for certain queries that don not use indexes. Moreover, it would be interesting to model non-B-tree-based indexes such as geospatial and text indexes, allowing the end-users to predict the performance of any design on a document store.

We obtained optimal designs by using shotgun hill-climbing as the optimization algorithms. However, it will be interesting to analyze the possibility of using more exploratory approaches such as genetic algorithms that are robust to local minima. Regarding the actual implementation of the prototype, we encountered several performance drawbacks while using HypergraphDB in representing the canonical model. Thus, by using an alternative approach to represent the canonical model, we will be able to improve the performance of the optimization process. Moreover, we see the perspective of introducing other optimization criteria into the process and investigating fine-tuning the existing cost functions to suit the end-user needs better.

Finally, by extending both the canonical model and the cost model to other data stores, we can expand our database design problem to include the choice of the data storage model. Thus, given entity-relationships and a query workload, answering the question, what will be the best data store combination to use and how will the data be stored in that particular polyglot system.

Appendices

Appendix A

DocDesign: Cost-Based Database Design for Document Stores

1 Introduction

Before the last couple of decades, traditional Relational Database Management Systems (RDBMSs) have been the go-to solution for data storage, given their maturity and popularity. However, with the big data era, NoSQL systems were introduced as alternate storage solutions, giving rise to novel data storage paradigms [24, 74]. More than 200 NoSQL systems are currently available, catering to specific niches of modern data storage[1]. Among these systems, document stores provide extended features (e.g., complex queries) because of the flexible, semi-structured data storage model. The JSON storage is now widely used in data analytics due to the fast serialization and ease of interchange between programs. This semi-structured nature of document stores allows them to handle the problem of data variety efficiently but introduces new challenges in database design.

Table A.1: Design Alternatives of the use case

Description	Representation	Description	Representation	Description	Representation

1. *Books* as the top level collection and their *Authors* are embedded	`Books:{`
	`"B_ID":int,`
	`"B_NAME": varchar,`
	`"Authors" : [{`
	`  "A_ID" : int,`
	`  "A_NAME": varchar,`
	`}]}`
		2. *Authors* as the top level collection and their *Books* are embedded	`Authors:{`
			`"A_ID":int,`
			`"A_NAME": varchar,`
			`"Books" : [{`
			`  "B_ID": int,`
			`  "B_NAME": varchar,`
			`}]}`
				3. Both *Authors* and *Books* as top level and redundantly embedding their counterparts	`Authors:{`
					`"A_ID":int,`
					`"A_NAME": varchar,`
					`"Books" : [{`
					`  "B_ID": int,`
					`  "B_NAME": varchar,`
					`}]} ,`
					`Books:{`
					`"B_ID":int,`
					`"B_NAME": varchar,`
					`"Authors" : [{`
					`  "A_ID" : int,`
					`  "A_NAME": varchar,`
					`}]}`

4. *Books* have only references to their *Authors*	`Books:{`
	`"B_ID":int,`
	`"B_NAME": varchar,`
	`"Authors" :`
	`  ["A_ID": int]`
	`},`
	`Authors:{`
	`"A_ID":int,`
	`"A_NAME": varchar`
	`}`
		5. *Authors* have only references to their *Books*	`Authors:{`
			`"A_ID":int,`
			`"A_NAME": varchar,`
			`"Books" :`
			`  ["B_ID": int]`
			`},`
			`Books:{`
			`"B_ID":int,`
			`"B_NAME": varchar,`
			`}`
				6. Both *Authors* and *Books* have redundant references to their counterparts	`Authors:{`
					`"A_ID":int,`
					`"A_NAME": varchar,`
					`"Books" :`
					`  ["B_ID": int]`
					`},`
					`Books:{`
					`"B_ID":int,`
					`"B_NAME": varchar,`
					`"Authors" :`
					`  ["A_ID": int]`
					`}`

7. Reification of the relationship in a bridge collection	`Authors:{`
	`"A_ID":int,`
	`"A_NAME": varchar`
	`},`
	`Books:{`
	`"B_ID":int,`
	`"B_NAME": varchar`
	`},`
	`Author_Book:{`
	`"A_ID":int,`
	`"B_ID": int`
	`}`				

RDBMS concepts are based on relational algebra. Thus, normalization governs the database design, and reaching 3NF or BCNF guarantees an optimal design for most use cases. However, there are no such methodologies available for document stores because they encourage de-normalization and

---

[1]http://nosql-database.org

accept data redundancy, making the database design an extremely difficult task. Thus, the database design process of document stores is driven by the queries and carried out in an ad-hoc manner with a trial and error approach. Most of the performance issues of the queries are addressed, in some cases, simply by introducing more powerful hardware. Sadly, this practice leads to sub-optimal use of resources (e.g., computing power, money). Moreover, not everyone can afford to upgrade their hardware, and some issues arising from design mistakes cannot be compensated by powerful hardware. It has been shown [22] that better database designs in NoSQL systems have a significant impact on quality requirements such as performance, cost, and scalability. Thus, having a better database design for a document store is vital in many aspects of its operation. Unfortunately, the decision making for optimal design is time-consuming, given less priority in the development life cycle, and definitely not trivial.

Considering the above challenges, we propose a decision aiding framework in the context of database design for document stores: DocDesign. DocDesign can estimate important parameters that are affected by database design decisions such as storage space, and query performance for a given use case. Thus, it helps the end-user to make informed decisions rather than using a trial and error process. DocDesign also generates executable queries of the workload over the different corresponding designs. This availability of the designs and the queries can fast-track the initial development process and also mitigates the query rewriting costs in design migration scenarios. **_Use cases._**
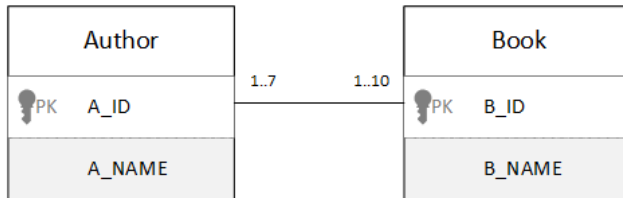
Fig. A.1: Conceptual Schema of the Example use case

We take a simple use case of *Authors* and *Books* as running example (Fig. A.1) and show how DocDesign can be used to aid the task of determining the optimal database design for a document store. We assume that we want to store the information of 2.5 million *Authors* and 4 million *Books*. We choose MongoDB to store this data as it is the most used document store at the time of writing[2]. We also assume that all primary keys are 12 bytes (default size of MongoDB identifier), and *A_NAME* and *B_NAME* are 105 and 155 bytes, respectively. For this use case in an RDBMS, there is only one design option being considered following 3NF with separate tables for *Author* and *Book* and

---

[2]https://db-engines.com/en/ranking/document+store
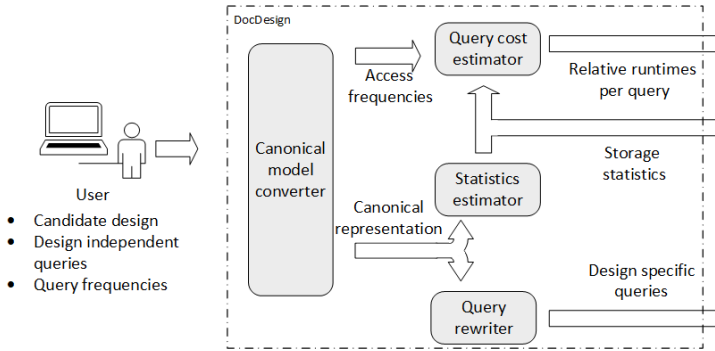
Fig. A.2: Overview of DocDesign

a bridge table *Author_Book*. However, in document stores, there are multiple alternative designs that one can obtain through referencing and embedding. Let us assume that our document store should be able to answer the following queries.

Q1 Find the author name by A_ID
Q2 Find the book name by B_ID
Q3 Find all the books with a given A_ID
Q4 Find all the author names with a given B_ID

In reality, document stores handle multiple queries simultaneously, and each query is executed with different probabilities, conforming a workload. We assume that each of the above queries is executed with the same probability (0.25). The goal is to find the best design to use in MongoDB, attending to the performance of the queries.

2 DocDesign

DocDesign is capable of evaluating several alternative designs for a document store and aid the decision making of the user in choosing the optimal design. It takes the design alternatives and the query workload as inputs. Then, using a hypergraph-based canonical representation introduced in Chapter 3 estimates the storage metadata, such as disk space taken by collections and indexes, repetition of each of the attributes within a collection, and cardinalities. Finally, using these metadata and the workload, DocDesign estimates the relative runtimes for the queries using the cost model in Chapter 4. The user can determine which design suits his/her needs by evaluating these storage space and query performance estimates. The development is further simplified through DocDesign by presenting the actual queries on the specific design for the entire workload.

2.1 Design Alternatives

The fundamental difference with RDBMS is that, by breaking 1NF, document stores can represent relationships between entities either by referencing or nesting. In referencing, the reference of the related entity is kept on the primary entity following 1NF. However, document stores do not implement join algorithms, and these need to be done outside of the datastore engine at the application level. Thus, nesting is encouraged to overcome such limitations [86]; rather than keeping a reference, the entire related entity can be embedded in the primary entity. Through de-normalization, they make redundancy acceptable in certain scenarios. Considering these facts, we can exhaustively identify the seven database designs for our use case of books and authors, shown in Table A.1. Designs #1, #2, and #3 use nesting to represent the many-to-many relationship while Designs #4, #5, #6, and #7 use referencing. Designs #3 and #6 introduce redundancy, replicating the relationship on both sides.

   When producing the designs in Table A.1, it is essential that all the information of the use case in Fig. A.1 is preserved, and no information is lost (e.g., at least one side of the many-to-many relationship should be included in the design). Note that our use case only contains two entities with two attributes each. Vertical partitioning of the entities could introduce additional designs if the entities had additional attributes. Also, adding more entities adds more complexity to the problem with different combinations of referencing and embedding. In such scenarios, the database design decisions are impossible to be analyzed manually. Consequently, we plan to include automatic design generation in future iterations of DocDesign using transformation rules. Nevertheless, for this demo, we manually generate the designs to be considered by DocDesign.

2.2 Canonical Representation

A meta-representation of the designs is required to compare one against another. Chapter 3 proposed a hypergraph-based canonical model able to represent heterogeneous NoSQL systems and design constructs. Thus, a small extension of this model, as shown in Fig. 1.6, also allows us to generate design dependant queries automatically. Indeed, in order to compute the required statistics, we included the data type and the number of distinct values in each of the attribute atoms (A_A), the total number of entities as count in the class atoms (A_C), and the multiplicity of the relationships between the atoms. Moreover, since a hyperedge represents a set or a struct of the design, the storage size of each hyperedge and the count of the elements inside a set as calculated values (highlighted in purple). Atoms and their relationships are immutable and are considered as the storage agnostic representation of infor-

mation. We provide this immutable information in JSON together with the additional attributes mentioned above. In our running example, we will have two class atoms (for A_ID and B_ID), two attribute atoms (for A_NAME and B_NAME), and four relationships ($A\_ID \rightarrow A\_NAME$, $B\_ID \rightarrow B\_NAME$, $A\_ID \rightarrow B\_ID$, and $B\_ID \rightarrow A\_ID$). Once DocDesign has this, the designs can be represented on top of it using hyperedges.

2.3 Query Workload

DocDesign takes storage agnostic selection queries with their access frequency as input (e.g., Q1, Q2, Q3, and Q4, with 0.25). From those, the algorithm introduced in Chapter. 3 generates projection queries over the document store. Moreover, we extended the very same algorithm and included simple selection predicates when representing the query over the immutable information. For example, Q1 can be represented as projecting A_NAME and having a selection on A_ID. DocDesign then transforms this query over the immutable information into an actual query over the design. It will generate "$db.Authors.find(\{A\_ID : \langle id \rangle\}, \{A\_NAME : 1\})$" for Designs #2, #3, #4, #5, #6 and #7 and "$db.Books.find(\{ "Authors.A\_ID" : \langle id \rangle\}, \{"Authors.A\_NAME" : 1\})$" for Design #1. In the current version, due to limitations of the cost model, DocDesign only supports single attribute selection on predicates and expects selections done through indexes. We consider nested-loop join algorithm executed on the client application for joins.

2.4 Estimating the Runtime

We use the cost model for document stores based on disk access, that can estimate the relative query cost for random access queries (i.e., request documents through an index). This cost model is inspired by RDBMS cost models used for query optimization [77]. However, we introduce the memory usage for the document store cost model since most of them encourage having data in memory for better performance. This cost model requires the average document sizes, number of documents, *multiplier* of attribute values inside, and access frequency of each collection.

Among these, access frequencies are given per query, and DocDesign calculates the access frequencies per collection, which obviously depends on the design. We obtain the rest of the parameters through the canonical model. First, we can calculate the storage size of the collections using a modified version of the query generation algorithm. Moreover, for each of the attributes within a collection, both top-level and nested, we calculate the *multiplier* of the attributes with regard to the top-level document, to estimate the size of the index defined on that attribute, as well as the selectivity factor of the corresponding query. For top-level attributes, this is always 1, but when

Table A.2: Storage Metadata of the Designs

Design			#1	#2	#3	#4	#5	#6	#7
Authors	Avg doc size (Bytes)		-	1300	1300	150	244.5	244.5	150
	Storage size (MB)		-	3200	3200	375	611.3	611.3	375
	P.index size (MB)		-	50.1	50.1	50.1	50.1	50.1	50.1
	S.index size (MB)		-	275	275	-	275	275	-
	Multiplier	A_ID	-	1	1	1	1	1	1
		B_ID	-	5.5	5.5	0	5.5	5.5	0
Books	Avg doc size (Bytes)		740	-	740	265.6	200	265.6	200
	Storage size (MB)		3000	-	3000	1100	800	1100	800
	P.index size (MB)		80.2	-	80.2	80.2	80.2	80.2	80.2
	S.index size (MB)		275	-	275	275	-	275	-
	Multiplier	A_ID	3.43	-	3.43	3.43	0	3.43	0
		B_ID	1	-	1	1	1	1	1
Authors Books	Avg doc size (Bytes)		-	-	-	-	-	-	64
	Storage size (MB)		-	-	-	-	-	-	878.3
	P.index size (MB)		-	-	-	-	-	-	275
	A.index size (MB)		-	-	-	-	-	-	275
	B. index size (MB)		-	-	-	-	-	-	275
	Multiplier	A_ID	-	-	-	-	-	-	5.5
		B_ID	-	-	-	-	-	-	3.43
Total storage (MB)			3355.2	3525.1	6880.3	1880.3	1816.6	2391.6	3008.6

indexing nested attributes, the same top-level document can be referred by several indexed values (in Designs #2 and #5, there are 5.5 *Books* per *Author* in Table A.2). Finally, DocDesign estimates the relative runtimes for each of the queries per design using the cost model. The actual runtimes for all the queries and designs on a single instance of MongoDB 4.2 using MongoDB Java driver 3.8.2 on Debian 4.9 with Intel Xeon E5520, 24 GB of RAM running. are shown in Table. A.3. We restricted the cache size of MongoDB to 256MB to make sure the collections do not fit in memory.

DocDesign presents the storage requirements and the query costs to the end-user for each design alternative. Depending on other user requirements, such as the importance of a particular query or restrictions on storage space, the ideal design can be selected. Our running example results in Table. A.3 show that the best design is #3 (same as the prediction). We calculated the Discounted Cumulative Gain (DCG) [62] of each of the query predictions and the overall design rank. We then calculated the min-max normalized gain against the worst and the best rankings with a minimum result of 0.85 for the Q2, but 0.95 for the overall prediction.

3 Demonstration Overview

In the on-site demonstration, we intend to showcase DocDesign using the running example of this chapter. It is essential to understand the complexity

of the database designing problem for document stores. We give a small questionnaire[3] asking the users to select the best design for different scenarios of our use case. After the feedback of the questionnaire, we introduce DocDesign. First, the user will be presented with the pre-defined queries together, will be allowed to provide the frequencies in our use case, and guess the best out of the seven designs. Then, we will present the values that we get from DocDesign (notice that response time of all individual queries changes when modifying the query frequencies, because of different memory allocation due to LRU-like eviction policy in use). Through this, we will showcase how time and effort can be saved by using DocDesign in the database design decision-making process for document stores (demo video in `https://vimeo.com/396513259`) We will also use TPC-C to show a complex use case with DocDesign, using only the read queries of the workload. We will compare the predictions given by DocDesign against the actual runtimes we obtain by running TPC-C in MongoDB . Next, we will compare different designs for TPC-C and see the DocDesign predictions. Finally, we will show the additional features of system such as allowing the users to input the queries over the immutable information and compare the generated queries on the desired design.[4]

4 Conclusion

Table A.3: Query Runtimes of the Designs

Design	Actual runtime (ms)						Prediction					
	Q1	Q2	Q3	Q4	Total	Rank	Q1	Q2	Q3	Q4	Total	Rank
1	25.63	24.78	69.28	25.19	**36.22**	5	1.82	1.58	6.10	1.58	**11.08**	3
2	16.36	28.23	17.38	51.67	**28.41**	2	1.39	1.65	1.39	3.97	**8.41**	2
3	14.15	14.55	15.70	14.44	**14.71**	1	1.34	1.45	1.34	1.45	**5.57**	1
4	10.09	14.82	48.05	44.56	**29.38**	3	1.26	1.61	5.94	5.93	**14.74**	5
5	15.65	11.53	77.92	38.95	**36.01**	4	1.42	1.45	9.40	3.78	**16.06**	6
6	20.19	22.59	66.45	40.35	**37.40**	6	1.62	1.72	5.94	4.01	**13.29**	4
7	12.67	13.56	93.80	93.64	**53.96**	7	1.06	1.28	9.68	8.75	**20.77**	7

Document stores have gained wide popularity, mainly because of addressing the data variety problem through semi-structured data storage. However, database designing for document stores is not trivial and is typically performed using heuristics. A good database design is vital for performance, as poor design decisions can incur substantial costs, both performance and cost-wise, especially in big data systems. Hence, we present DocDesign to aid in the design decision-making process for document stores. DocDesign is

---

[3]https://moditha.typeform.com/to/NRTjEm
[4]DocDesign demo available in `http://www.essi.upc.edu/dtim/tools/DocDesign`

capable of successfully predicting the storage requirements and relative query runtimes for a given design and query workload. Thus, the end-user can make an informed decision on which design to choose rather than resorting to trial and error approaches. We plan to enhance DocDesign by adding automatic design generation and multi-criteria optimization for suggesting the optimal designs.

Appendix B

DocDesign 2.0: Automated Database Design for Document Stores with Multi-criteria Optimization

1   Introduction

The plethora of current NoSQL systems introduces alternative data storage methods to the traditional relational database management systems (RDBMSs) [24]. Among these, document stores have gained popularity due to the semi-structured data storage model. In contrast to the RDBMS normalization, document stores favor embedding, trying to keep the data related to a single instance together instead of spreading it across different tables. This increases the complexity of database design for document stores as opposed to RDBMS, where reaching 3NF or BCNF guarantees an optimal database design in the majority of the use-cases. Database design for document stores is, in general, given low precedence, and mostly carried out in a rule-based ad-hoc manner. For instance, MongoDB, the leading document store, provides a set of design patterns[1] that provide certain guidelines on how to structure documents. However, it has been shown that the choice of design has a major impact on performance, specially in the NOSQL realm [10]. Thus, it is advantageous to have a better design by exploiting any prior knowledge on the requirements rather than a purely random one.

Let us take an example of implementing an online auction system based on the RUBiS benchmark [25] in a document store. Fig. B.1 shows the 5

Fig. B.1: ER diagram of RUBiS Benchmark

entities and 6 relationships composing the RUBiS framework. We can have a normalized solution, similar to that in a RDBMS, an embedded single-document solution, or the solution suggested by a purely workload-based schema recommender, such as DBSR [95], which denormalizes certain entities. To show the complexity of finding the optimal database design in a document store, let us define a running example use case consisting of two single entities from RUBiS, namely Product and Comments, and an equiprobable hypothetical workload defined as follows:

- Given a Comment ID, find its text.
- Given a Product ID, find its name.
- Given a Comment ID, find the Product name.
- Given a Product ID, find all of its Comments.

---

[1] https://www.mongodb.com/blog/post/building-with-patterns-a-summary

1. Introduction
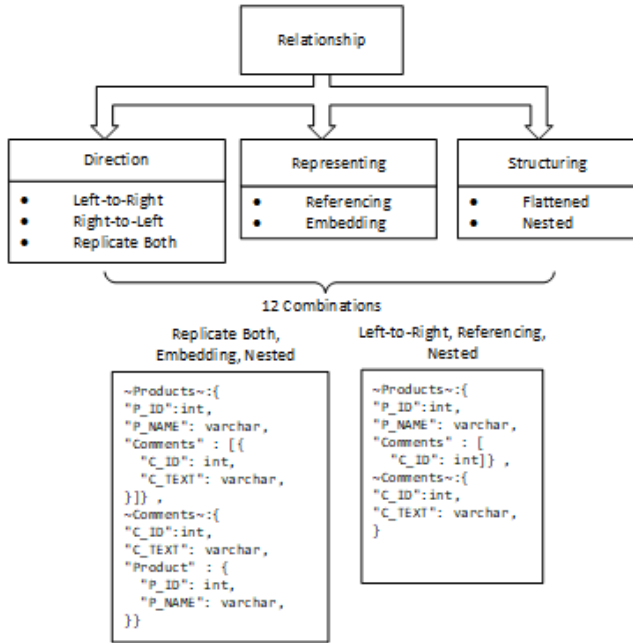
Fig. B.2: Relationship design choices, and two examples
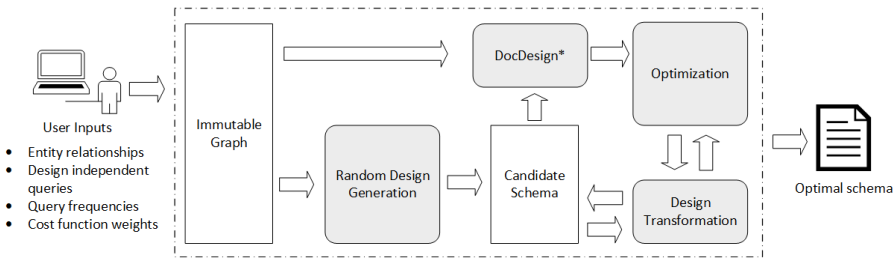
Fig. B.3: Overview of the DocDesign 2.0 Architecture

In this scenario, we have two entities and one relationship. If we assume that all attributes for an entity are kept together within a document, we are left with the decision on where the relationship must be stored in the final design. Thus, database designs can be enumerated based on the alternatives to store the relationship, which depend on three independent choices: direction, representing, and structuring as shown in Fig. B.2, together with two examples. **Direction** determines which entity keeps the information about the relationship. It can be one of the two entities, or both. **Representation** affects how this relationship is stored either by keeping a reference or embedding the object. Finally, **Structuring** determines how we structure the

relationship, either as a nested list or flattened. For example, if keep the references to the comments in the product, they can be stored as a list of references (*comment:...*) or in a flattened manner (*comment_1:..., comment_2:..*). Hence, we end up with 12 possible designs for our running example. Each of these could potentially be the optimal solution for an end-user depending on their preferences. For example, the design where products and comments nest their counterparts redundantly (i.e., both directions are stored by embedding the objects) will benefit query performance, as all queries can be answered with a single random access. However, this is at the expense of storage space due to redundancy. What if we only have a single reference on the product for its comments? Does the reduction of storage space justify the impact on performance? This trade-off between alternatives makes the process of finding the optimal design a complex one.

The number of relationships of the use-case (r) determines the number of candidate designs, which is exponential (12^r), as the storage option of each relationship is independent of others. Note, however, that here we did not consider allowing heterogeneous collections/lists, which is possible in the context of schemaless databases, leading to a complexity increase. For example, collections at the top level could potentially contain different kinds of documents. In our running example, user and region documents could be stored in a single heterogeneous collection mixing both. Precisely, for a design with c top-level collections, the total number of combinations will be $\sum_{i=1}^{c} \left\{ {c \atop i} \right\}$ where $\left\{ {n \atop k} \right\}$ is the Stirling number of the second kind, used to calculate the number of ways to partition n distinct elements into k non-empty subsets [51]. Overall, such exponential growth makes impossible to enumerate and evaluate all candidate designs. Hence, existing solutions, such as DBSR [95], NoSE [88] and Mortadelo [34], mainly rely on the query workload to propose a database design.

Contributions. Considering the above observations it is clear that the problem of storage design for document stores has a large search space. Moreover, each candidate solution potentially performs differently among the considered cost functions. It is, hence, obvious that exhaustively exploring the search space is prohibitively expensive. To overcome this issues, in this paper, we present DocDesign 2.0, a novel solution that addresses the complex problem of database design for document stores. DocDesign 2.0's contributions involve *automatically generating potential designs*, as well as *evaluating the performance of a design on four objectives: storage size, query performance, degree of heterogeneity, and average depth of documents*. Finally, DocDesign 2.0 *presents the end-user with the near-optimal database design specific to his/her preference of the objective* for a given use-case and query workload. Precisely, in this paper, we consider read-only query workloads. DocDesign 2.0 embeds and extends our former solution DocDesign in Appendix A, which aids on evaluating database designs

based on storage size and query performance, requiring however to provide a concrete schema as input. Contrarily, DocDesign 2.0 automatically generates such designs yielding, with a high probability, the near-optimal one with respect to a set of objectives.

Outline. In the rest of the paper we introduce DocDesign 2.0's demonstrable features to resolve the motivational example and other database design for document stores scenarios. We first provide an overview of DocDesign 2.0and its core features. Lastly, we outline our on-site demonstration, involving the motivational scenario as well as other more complex real-world use cases.

2 DocDesign 2.0 in a nutshell

DocDesign 2.0 adopts multi-objective optimization techniques, which have shown to be effective on obtaining near-optimal solutions out of a large search space in the presence of contradicting objectives [84]. In these scenarios, one can only aim to obtain a Pareto solution (a solution that, in the presence of multiple objectives, cannot improve one objective without worsening another).

Search algorithm. Local search algorithms consist of the systematic modification of a given state, by means of action functions, in order to derive an improved state. The intricacy of these algorithms consists of their parametrization, which is at the same time their key performance aspect. Due to the genericity of different use cases DocDesign 2.0 can tackle, we decided to choose *hill-climbing*, a non-parametrized search algorithm which can be seen as a local search, always following the path that yields higher utility values. Nevertheless, the cost functions we use are highly variable and non-monotonic, which can cause hill-climbing to provide different outputs depending on the initial state. To overcome this problem, we adopt a variant named *shotgun hill-climbing*, which consists of a hill-climbing with restarts using random initial states.

An overview of DocDesign 2.0 is shown in Fig. B.3 and we present the modules and components of DocDesign 2.0 in the following subsections.

2.1 User Inputs

There are three inputs the end-user must provide, namely the equivalent to an Entity-Relationship diagram of the domain, query workload, and the weights of the cost functions.

Listing B.1: Input entity relationships

```
{"atoms":[{
    "PRODUCT":{
        "*P_ID":{"count":2500000,"size":4},
        "P_NAME":varchar(155)}},
    {"COMMENT":{
        "*C_ID":{"count":5000000,"size":4},
        "C_TEXT":varchar(105)}}],
"relationships":[
    {"P_ID":{"C_ID":"1~2"}}
]}
```
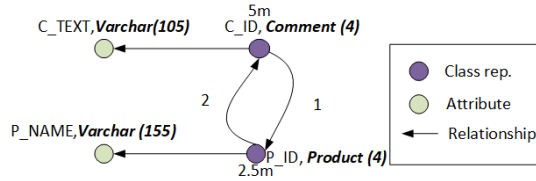

Fig. B.4: Immutable graph

Listing B.2: Query workload

```
[
{"freq":0.25,"q":["C_ID","C_TEXT"]},
{"freq":0.25,"q":["P_ID","P_NAME"]},
{"freq":0.25,"q":["C_ID","P_ID","P_NAME"]},
{"freq":0.25,"q":["P_ID","C_ID","C_TEXT"]}
]
```

Entity-Relationship. Refers to the use case-specific entities, their attributes, and the relationships between them. To accurately measure the different cost functions, DocDesign 2.0 requires the number of instances of each entity, the size of its attributes, and the relationship multiplicities (Listing B.1). This information is considered immutable, and the database design is carried out on top of it. We use a hypergraph-based canonical model internally to represent them (shown in Fig. B.4). Furthermore, the entities are atomic, meaning that attributes related to an entity cannot be split.

Query workload. Consists of a set of queries together with their frequencies to be executed in the use case. These queries are independent of the database design and are represented as subsets of the immutable information (Listing B.2).

Cost function weights. Allows the end-user to include his/her preference in the database design. Currently, DocDesign 2.0 supports tuning four cost functions corresponding to the objectives: query cost, storage size, degree of heterogeneity within collections and sets, and average depth of the documents. The end-user can decide how important each of these costs are and resolve trade-offs between them. For instance, forcing higher importance to query cost and lowering the one of storage size would lead to a schema with higher redundancy and better performance.

2.2 Design Operations

Information about entities, their attributes, and the relationships are considered immutable, and the database design is built from it. Indeed, with regard to our running example, the final design must have information on all the warehouses, districts, and the relationships between them. A hypergraph-based representation enables DocDesign 2.0 to guarantee this property (refer to Chapter 3 and Appendix A for further details). We introduce two methods to fit the shotgun hill climbing approach: generation of a random design, and evolution of a design using valid transformations.

Random design generation. The random schema generator relies on identifying subsets of entities and relationships that will be made into a collection (referred to as connected components) and the structure of the documents inside the collection in a document store database design. Based on the 12 possible designs that a relationship can be stored, we make the following decisions randomly in the schema generation process.

- **Root of the connected component** is chosen at random from the available entities. This choice determines the root document of the document store collection that this component represents. In our running example, this is either picking the warehouse or the district as the root of the collection. Let us assume we picked the warehouse in this case.
- **Choosing the next path to explore** expands the connected component and determines it's structure. Potentially multiple relationships connect an entity to others in a connected component. Thus, for a given entity of a connected component, a random subset of these relationships is picked to further expand, determining the depth and the related documents of the final design. This, together with the root of the document determines the choice of the direction in Fig. B.2 except for replicating both. In the running example, we choose the relationship to the district from the warehouse (already inside the component).
- **Embedding or Reference** determines possible ways to represent the relationship between two entities of a component. If embedding is chosen, the entire document is embedded in the parent and referencing only keeps the reference of the related document on the parent. In the running example, if the embedding option is chosen, the final collection will be warehouses with embedded districts. We also make the decision of replicating both based on a given probability.

The above choices are carried out until all the entities and relationships belong to at least one of the connected components. Finally, each of the components is represented as a document store collection. These initial designs do not contain heterogeneous collections or lists, yet, since we initially ignore the choice of flattening and only use the nested option for structuring with regard to the options in Fig. B.2. This decision reduces the complexity

of the random generation and the number of starting schemas. However, we introduce this through design transformations to ensure that we do not lose certain designs in the process.

Design transformations. Even though it is possible to generate most of the potential designs through the random generator, it is very unlikely to reach an optimal state randomly. Moreover, we omitted the heterogeneous collections/lists and flattened ones in the random process. Thus, we introduce seven design transformation operations and use five of them to generate the neighbors of a particular design. These transformations are inspired by the rule-based design patterns proposed by MongoDB.We have validated them by recreating the MongoDB design patterns as sequences of transformations[2].

- **Union** - merges two collections/lists at the same level and creates a heterogeneous one.
- **Segregate** - separates a homogeneous collection/list out of a heterogeneous one.
- **Embed** - embeds a related document inside another.
- **Flatten** - flattens an embedded document or a list inside it's parent.
- **Group** - creates an embedded list of related documents inside another (opposite of flattening a list).

We also identify two other operations, namely, **Nest** and **Split**. Nest operation creates a nested document inside another and is unnecessary as we already cover it through the random generation. Split is similar to vertical partitioning a document. However, adhering to the atomic entity rule, we decided not to include this operation as it would also expand the search space uncontrollably.

2.3 Optimization

Candidate designs obtained through random generation or transformation need to be evaluated in order to assess their optimality.

Cost functions. We introduce four cost functions to be measured and optimized in DocDesign 2.0: query cost, storage cost, degree of heterogeneity, and average depth of the documents. These are defined as follows:

- **Query cost** (CF_Q), is the sum of the relative query performance values calculated from the schema using the cost model for document stores.
- **Storage size** (CF_S), is the total storage size required by the collections and indexes, calculated using the canonical model.
- **Degree of heterogeneity** (CF_H), is the number of different types of documents in a collection/list. We use the average over all the collections and lists of the schema. Each heterogeneity is given a weight depending on

---

[2]More details at `https://www.essi.upc.edu/~moditha/transformations`
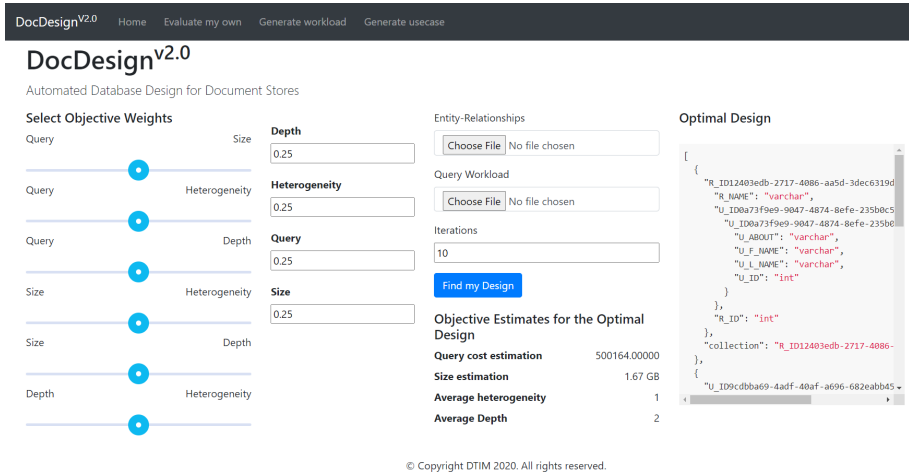
Fig. B.5: DocDesign 2.0 user interface

which level the list/collection lies in the document. The higher the level, the higher the assigned weight, penalizing heterogeneities at higher levels of the document structure.

• **Depth of the documents** (CF_D), is the average depth of the documents of the design.

Utility function. Guiding the local search algorithm requires the definition of a utility function taking into account the end-user's preferences. Here, this is a function to be minimized. Hence, the end-user can assign weights to each of the cost functions according to their importance in the use-case. Then, for a given design C, we define the utility as the normalized weighted sum of each cost function $u(C) = \sum_{i=1}^{n} w_i \frac{CF_i(C) - CF_i^o}{CF_i^{max} - CF_i^o}$. The expression considers the weight w of each cost function, which is used on the transformed utility function for C. This is a normalized value that considers the *utopia* (i.e., the expected minimal) and the maximal design costs, yielding values between zero and one.

3   Demonstration Overview

DocDesign 2.0 has a web interface as shown in Fig. B.5. In the on-site demonstration, we will showcase DocDesign 2.0 using the RUBiS usecase as a real-world example The manual database design process is expensive as RuBiS contains five entities and six relationships, leading to a large solution space. Moreover, we use the 11 queries with their access frequencies as the workload.

First, for the ease of explanation, we will use the paper's running example (i.e., Products and Comments) and the four queries to showcase the ease of using DocDesign 2.0, initially with equal weights and then higher weight to query cost. In the first scenario with equal weights, the optimal schema is products having references to their comments. When optimizing only for the query performance, DocDesign 2.0 suggests redundantly nesting comments inside the product and product inside the comment. This approach reduces the actual runtime almost by half at the expense of double the storage space. This establishes the functionality and the efficiency of DocDesign 2.0.

Then, we will import the full RUBiS E/R to DocDesign 2.0 together with the queries and showing the ability of DocDesign 2.0 to solve more complex usecases. The results presented by DocDesign 2.0 have a higher throughput once implemented compared to the best solution suggested by DBSR [95]. Moreover, the suggestion by DocDesign 2.0 has far less redundancy compared to the ones by DBSR. The participants are also allowed to interact with the DocDesign 2.0 demonstration with the ability to choose between different queries and objective function weights as well as generate their own. The resulting updates made to the design can be discussed by means of changes introduced (e.g.: giving more importance to query cost will result data redundancy). We also present the actual runtimes (calculated by a benchmarking suite) and storage sizes for the usecases and the designs that we demonstrate. This allows the users to validate the effectiveness of the solutions generated by DocDesign 2.0.

Since the JSON input format is specific to DocDesign 2.0, we also include a functionality to create them through an intuitive UI. Moreover, the users can suggest their own design to compare against the one suggested in terms of the four objective functions. The designs suggested by DocDesign 2.0 rely on pre-defined queries. If the queries are unknown the end users have to rely on the other three cost functions to obtain a "good enough" design. Through this hands-on experience, we are able to show the ability of DocDesign 2.0 to address the complex problem of document store database design improving the quality and productivity as opposed to a manual design process.[3]

---

[3]Demo video available at `https://vimeo.com/505248323`

Appendix C

Calculating Internal B-tree Blocks

We calculated the probability of a leaf block of data B-tree in memory being requested $P_d^{req}(C)$ as $1 - (1 - SF(C))^{R_d(C)}$, in Eq. 4.10. . Hence, we continue the calculation of the internal nodes of the data B-tree as follows.

If a document is in the cache, the data block containing the document and the internal block containing the reference entry to that data block must be in the cache. On the contrary, if an internal block is not in the cache, none of the data blocks pointed by the reference entries in it can be in the cache. Therefore, for an internal block not to be in the cache, all of the reference entries of the block should reference blocks, not in the cache. Thus, since the probability of a single reference entry referring to a leaf block not in the cache is $1 - P_d^{req}(C)$, and there are $R_{int}(C)$ reference entries in a single internal block, the probability of an internal block in memory being requested can be defined as follows.

$$P_{int}^{req}(C) = 1 - (1 - P_d^{req}(C))^{R_{int}(C)} \qquad (C.1)$$

Moreover, we estimate the number of internal blocks pointing to the leaves, $Inter(C)$ as follows.

$$Inter(C) = \left\lceil \frac{\lceil \frac{|C|}{R_d(C)} \rceil}{R_{int}(C)} \right\rceil \qquad (C.2)$$

Finally, we can state the cached internal blocks $M_{int}(C)$ as follows.

$$M_{int}(C) = Inter(C) * P_{int}^{req}(C) \qquad (C.3)$$

Appendix D

Cost Calculation Examples for MongoDB

We present the application of our cost model in MongoDB with two examples. First, a single collection accessed through primary index with complete set of equations and calculations. Second, we present a real world usecase with only the initial calculation of the inputs because the complexity and the number of equations increase in such scenario.

1 Single Collection with Primary Index

Let us take an scenario with Test 1 (13 million documents) and average document size of 40 bytes. First, we calculate the average number of documents and index entries in a block, together with the total number of data and index blocks as follows (by applying Eqs.1 and 2).

$$|C| = 13 * 10^6 \quad Bsize_d = 32Kb \quad Bsize_i = 32Kb$$

$$Size_d(C) = 40b \quad Size_{i\_id}(C) = 22b \quad F = 0.7$$

$$R_d(C) = 0.7 \cdot \left\lfloor \frac{32768}{40} \right\rfloor = 573$$

$$R_{\_id}(C) = 0.7 \cdot \left\lfloor \frac{32768}{22} \right\rfloor = 1042$$

$$B_d(C) = \left\lceil \frac{13 * 10^6}{573} \right\rceil = 22676 \quad K = 10Mb \quad M = 256Mb$$

$$B_{\_id}(C) = \left\lceil \frac{13 * 10^6 * 1}{1042} \right\rceil = 12473 \quad u = 0.80$$

1. Single Collection with Primary Index

Since we have only the primary index, $Rep_{\_id} = 1$, $P(C, \_id) = 0.5$, and $P(C) = 0.5$. Now, applying Eqs. 4.7–4.13 together with Eqs. 4.4 and 4.19, we come up with the following set of equations.

$$Req_{\_id}(C) = |Q| \cdot 0.5$$

$$E_{\_id}(C) = 13 * 10^6 * \left(1 - \left(\frac{13 * 10^6 - 1}{13 * 10^6}\right)^{Req_{\_id}(C)}\right)$$

$$SF_{\_id}(C) = \frac{E_{\_id}(C)}{13 * 10^6} = \left(1 - \left(\frac{13 * 10^6 - 1}{13 * 10^6}\right)^{Req_{\_id}(C)}\right)$$

$$SF(C) = SF_{\_id}(C) = \left(1 - \left(\frac{13 * 10^6 - 1}{13 * 10^6}\right)^{Req_{\_id}(C)}\right)$$

$$P_d^{req}(C) = 1 - (1 - SF(C))^{573}$$

$$P_{\_id}^{req}(C) = 1 - (1 - SF_{\_id}(C))^{1042}$$

$$M_d^{sat}(C) = 22676 * P_d^{req}(C) \quad M_{\_id}^{sat}(C) = 12473 * P_{\_id}^{req}(C)$$

$$M_d^{sat}(C) * 32768 + M_{\_id}^{sat}(C) * 32768 = ((0.8 * 256) - 10) * 1024^2$$

By solving the above set of equations, we obtain the values for $|Q| = 4242.85$, $M_d^{sat}(C) = 3038.98$, and $M_{\_id}^{sat}(C) = 2847.82$. Using these values at the memory saturation point, we can come up with the following set of equations by applying Eqs. 4.21–4.27 together with Eqs. 4.4 and 4.19.

$$P_d^{in}(C) = P_d^{out}(C) \quad P_{\_id}^{in}(C) = P_{\_id}^{out}(C)$$

$$Shots_d^{in}(C) = 3038.98 \cdot \frac{M_d(C)}{22676}$$

$$Shots_{\_id}^{in}(C) = 2847.82 \cdot \frac{M_{\_id}(C)}{24962}$$

$$E_d(C) = M_d(C) - Shots_d^{in}(C)$$

$$E_{\_id}(C) = M_{\_id}(C) - Shots_{\_id}^{in}(C)$$

$$W_d(C) = \frac{M_d(C) \cdot 32768}{((0.8 * 256) - 10) * 1024^2}$$

$$W_{\_id}(C) = \frac{M_{\_id}(C) \cdot 32768}{((0.8 * 256) - 10) * 1024^2}$$

$$P_d^{out}(C) = \frac{W_d(C) \cdot \frac{E_d(C)}{M_d(C)}}{W_d(C) \cdot \frac{E_d(C)}{M_d(C)} + W_{\_id}(C) \cdot \frac{E_{\_id}(C)}{M_{\_id}(C)}}$$

$$P_{\_id}^{out}(C) = \frac{W_{\_id}(C) \cdot \frac{E_{\_id}(C)}{M_{\_id}(C)}}{W_d(C) \cdot \frac{E_d(C)}{M_d(C)} + W_{\_id}(C) \cdot \frac{E_{\_id}(C)}{M_{\_id}(C)}}$$

$$P_d^{in}(C) = \frac{3038.98 \cdot (1 - 0.5)}{3038.98 \cdot (1 - 0.5) + 2847.82 \cdot (1 - 0.5)} = 0.52$$

$$P_{\_id}^{in}(C) = \frac{2847.82 \cdot (1 - 0.5)}{3038.98 \cdot (1 - 0.5) + 2847.82 \cdot (1 - 0.5)} = 0.48$$

$$M_d(C) * 32768 + M_{\_id}(C) \cdot 32768 = ((.8 * 256) - 10) * 1024^2$$

By solving the above equations we obtain $M_d(C) = 2847.82$ and $M_{\_id}(C) = 3038.98$. By applying these values on Eqs. 3 and 4 we get a relative cost for a query through _id as follows.

$$P_d(C) = \frac{2847.82}{22676} = 0.12 \quad P_d(C) = \frac{3038.98}{12473} = 0.24$$

$$Cost_{Rand} = 2 - (0.24 + 0.12) = 1.64$$

2 Multiple Collections

Let us take a use-case of storing the data of authors and their books. Let's assume that we chose to have a reference to the authors inside each of the books (as shown in Listing D.1) out of the possible design choices. Moreover, let us also assume that each author has 5 books and each book has 3 authors on average.

Listing D.1: Example schema of a document store

```
"Books":{
    "_id": <int>,
    "B_NAME": <varchar>,
    "Authors" : ["A_ID": <int>]
},
"Authors":{
    "_id":<int>,
    "A_NAME": <varchar>
}
```

In this scenario, we have two collections and three indexes (two primary on each of the collections and one secondary index on *A_ID* in Books). We calculate the number of documents/indexes in a block and the total number of blocks for each of these five B-trees as follows.

$$|\mathsf{Books}| = 4 * 10^6 \quad |\mathsf{Authors}| = 2.5 * 10^6 \quad Bsize_{d/i} = 32Kb$$

$$Size_d(\mathsf{Books}) = 265b \quad Size_d(\mathsf{Authors}) = 150b \quad F = 0.7$$

$$Size_i(\mathsf{Books/Authors}) = 22b \quad R_d(\mathsf{Authors}) = 152$$

$$R_d(\mathsf{Books}) = 89 \quad B_d(\mathsf{Authors}) = 16448$$

$$B_d(\mathsf{Books}) = 44944$$

$$R_{\_id}(\mathsf{Books}) = R_{\mathsf{A\_ID}}(\mathsf{Books}) = R_{\_id}(\mathsf{Authors}) = 1042$$

$$B_{\_id}(\mathsf{Books}) = 3843 \quad B_{\_id}(\mathsf{Authors}) = 2402$$

$$Mult_{\mathsf{A\_ID}}(\mathsf{Books}) = 5 \quad B_{\mathsf{A\_ID}}(\mathsf{Books}) = 19213$$

The following queries are executed with equal probability (0.25) on our documents store.

Q1 Find the author name by _id

Q2 Find the book name by _id

Q3 Find all the book names with a given _id of an author

Q4 Find all the author names with a given _id of a book

Since our cost model depends on the access probability on each of the B-tree structures, we calculate them as shown in Table D.1. In Q3, we have single access to the secondary index on A_ID and five access to the data B-tree. Q4 involves two queries, first, one to retrieve a book through its _id and then on average, there would be 3 A_IDs which need to be retrieved as three independent requests through _id of the Authors collection.

Table D.1: Calculating the access probability of the B-trees

	Index usage			Collection usage	
	Book		Author	Book	Author
	_id	A_ID	_id		
Q1	-	-	0.25	-	0.25
Q2	0.25	-	-	0.25	-
Q3	-	0.25	-	5*0.25	-
Q4	0.25	-	3*0.25	0.25	0.75
Total	0.5	0.25	1	1.75	1
Probability	0.111	0.056	0.222	0.389	0.222

Now, we have the final input for our cost model, together with $Rep_{\mathsf{A\_ID}}(\mathsf{Books})$ by applying Eq. 4.6, as follows:

$$P(\mathsf{Book}) = 0.389 \quad P(\mathsf{Book}, \_id) = 0.111$$
$$P(\mathsf{Book}, \mathsf{A\_ID}) = 0.056$$
$$P(\mathsf{Author}) = 0.222 \quad P(\mathsf{Author}, \_id) = 0.222$$
$$Rep_{\mathsf{A\_ID}}(\mathsf{Books}) = \frac{5 * 2.5 * 10^6}{4 * 10^6} = 3.125$$

Now, we can apply Eqs. 4.7–4.13 together with Eqs. 4.4 and 4.19 on the inputs to obtain the values for memory distribution at the saturation point. Then, using these results on Eqs. 4.21–4.27 together with Eqs 4.4 and 4.19, we obtain the following final memory distribution. These calculations are similar to the example in Appendix D.1, but we omit listing them out due to their extensiveness.

$$M_d(\mathsf{Books}) = 2532 \quad M_{\_id}(\mathsf{Books}) = 638$$
$$M_{\mathsf{A\_ID}}(\mathsf{Books}) = 351$$
$$M_d(\mathsf{Authors}) = 1401 \quad M_{\_id}(\mathsf{Authors}) = 935$$

Finally, we calculate the miss rates and the relative cost of each of the queries as follows.

$$P_d(\mathsf{Books}) = \frac{2532}{44944} = 0.056 \quad P_{\_id}(\mathsf{Books}) = \frac{638}{3843} = 0.166$$
$$P_{\mathsf{A\_ID}}(\mathsf{Books}) = \frac{351}{19213} = 0.018 \quad P_d(\mathsf{Authors}) = \frac{1401}{16448} = 0.08$$
$$P_{\_id}(\mathsf{Authors}) = \frac{935}{2402} = 0.38$$

$$Cost(Q1) = 2 - (0.38 + 0.08) = 1.54$$
$$Cost(Q2) = 2 - (0.166 + 0.056) = 1.778$$
$$Cost(Q3) = 1 - 0.018 + 5 * (1 - (0.056)) = 5.702$$
$$Cost(Q4) = 2 - (0.166 + 0.056) + 3 * (2 - (0.38 + 0.08))$$
$$= 6.398$$

Appendix E

Algorithm to build hyperedges from connected components

Algorithm 9 *BuildHyperedge* Algorithm

Input: Cnode *node*, HyperG *G*, List<A_C> *allAtoms*

1: *list elements* ← *newList*()
2: **for each** *child* ∈ *node.children* **do**
3:     *elements.addAll*(*BuildHyperedge*(*child*, *G*, *allAtoms*))
4: **if** *node.type = SKIP* **then**
5:     *elements.add*(*node.rel*)
6: **else if** *node.type = REF* **then**
7:     *elements.add*(*node.to*)
8:     *elements.add*(*node.rel*)
9: **else if** *node.type = NEST* **then**
10:     *hyp* ← *G.newHyperedge*(*DOCUMENT*, *node.to*, {*attributesrelationships*(*node.to*), *elements*})
11:     *elements.clear*()
12:     *elements.add*(*node.rel*)
13:     *elements.add*(*hyp*)
14:     *set* ← *G.newHyperedge*(*LIST*, *elements*)
15:     *elements.clear*()
16:     *elements.add*(*set*)
17:     *allAtoms.remove*(*node.to*)
18: **else if** *node.type = ROOT* **then**
19:     *hyp* ← *G.newHyperedge*(*TOPDOC*, *node.to*, {*attributesrelationships*(*node.to*), *elements*})
20:     *G.newHyperedge*(*COLLECTION*, *hyp*)
21:     *elements.clear*()
22:     *allAtoms.remove*(*node.to*)
23: **return** *elements*

Algorithm 9 generates corresponding E_H^{Doc}s recursively for a given connected component tree. It takes a *Cnode*, the hypergraph, and a list of unused A_Cs from Algorithm 7. We use a recursive call to build hyperedges to all the child *Cnode*s of a given *Cnode* (lines 2-4) and keep them in a list of elements. Then, if the *Cnode* type is *SKIP*, we only add the E_R of the *Cnode* to the elements (lines 5-6). If it is *REF*, we add both E_R and its originating A_C (to) to the elements (lines 7-9). If it is *NEST*, we make a new E_{Doc}^{Doc} with the originating A_C (to) as the root, the A_As connected to the root and their corresponding E_Rs, and the built up elements (line 11). Once we clear the elements in line 12, we build a E_{List}^{Doc} containing the newly created E_{Doc}^{Doc} and the E_R of the *Cnode* (lines 11-15). Then, we reset the element list to contain only the new E_{List}^{Doc} and take out the originating A_C from the unused A_Cs in lines 16-18. If the *Cnode* is *ROOT*, we build a E_{Top}^{Doc} with the element list and an E_{Col}^{Doc} containing the new E_{Top}^{Doc}, clear the element list and remove the originating A_C from the unused A_Cs (lines 19-24). Finally, the collected element list is returned to be used by the calling function (line 25).

Appendix F

Formalized transformations

Formal definitions of the transformations discussed in Chapter 5.2 are described in Table F.1. They allow to transform any valid document design and at the same time **guarantee the validity** of the resulting design. We use an auxiliary function $findRelPath(x : \mathbb{E}_H, y : \mathbb{E}_H)$ which will find the path of relations from x to $O(y)$.

<div align="center">

Table F.1: Document store-specific transformation methods

</div>

Method	$\langle\langle$**preconditions**$\rangle\rangle$	Activity

$E_{Struct}^{Doc}.embed($ $: E_{Struct}^{Doc})$	• $self.parent = s.parent$
	• $O(self) \neq O(s) \Rightarrow$ $(\exists\{E_R^{x_1,x_2}, \ldots, E_R^{x_n,O(s)}\}$ $\subseteq self \wedge x_1 \in self) \vee$ $(\exists\{E_R^{y_1,y_2}, \ldots, E_R^{y_n,O(self)}\}$ $\subseteq s \wedge y_1 \in s)$
		$keep \leftarrow \varnothing$
		for each $c \in (self.parent.children \cap$ $\mathbb{E}_{Struct}^{Doc}) - s$ **do**
		  $keep \leftarrow$
		$keep \cup FindRelPath(self.parent, c)$
		for each $r \in$
		$FindRelPath(self.parent, s) - keep$ **do**
		  $self.parent.removeNode(r)$
		$self.parent.removeNode(s)$
		$self.addNode(s)$

$E_{Struct}^{Doc}.group($ $Re : Set\ of\ E_R,$ $s : E_{Doc}^{Doc}|A)$	• $Re \subset self$
	• $s \in self$
		if $s \in \mathbb{E}_{Struct}^{Doc}$ **then**
		  $new\ List(self.C, Re, \{s\}, \varnothing, self)$
		else
		  $new\ List(self.C, Re, \varnothing, \{s\}, self)$
		$self.removeNode(s)$
		$used \leftarrow \varnothing$
		for each $c \in self.children \cap (\mathbb{E}_{Doc}^{Doc} \cup$ $\mathbb{A})$ **do**
		  $used \leftarrow$
		$used \cup FindRelPath(self, c)$
		for each $r \in (Re - used)$ **do**
		  $self.removeNode(r)$

$E_{Struct}^{Doc}.split($ $r : A_C$, $Re :$ Set of E_R, $At :$ Set of A, $Do :$ Set of E_{Struct}^{Doc}, $Li :$ Set of E_{Set}^{Doc}, $Rm :$ Set of E_N)	$r \in (self \cap At)$ $Rm \subseteq (Re \cup At \cup Do \cup Li) - O(self) \subset self$ $O(self) \neq r \implies (\exists \{E_R^{x_1,x_2}, \ldots, E_R^{x_n,r}\} \subseteq (self - Rm) \wedge x_1 \in (self - Rm)) \vee (\exists \{E_R^{y_1,y_2}, \ldots, E_R^{y_n,O(self)}\} \subseteq Re \wedge y_1 \in At)$	**if** $self \in \mathbb{E}_{Top}^{Doc}$ **then**
		   new $TopDoc(self.C, r, Re, At, Do, Li, self.parent)$
		 else
		   $Rels \leftarrow FindRelPath(O(self), r)$
		   for each $re \in Rels$ **do**
		     $self.parent.addNode(re)$
		   new $Document(self.C, r, Re, At, Do, Li, self.parent)$
		 for each $n \in Rm$ **do**
		   $self.removeNode(n)$

$E_{Set}^{Doc}.segregate($ $s : E_{Struct}^{Doc} \mid A)$	$s \in self$	**if** $self \in \mathbb{E}_{Col}^{Doc}$ **then**
		   new $Collection(self.C, s)$
		 else
		   $keep \leftarrow \varnothing$
		   $Re \leftarrow FindRelPath(self.parent, s)$
		   for each $c \in (self.children - s)$ **do**
		     $keep \leftarrow keep \cup FindRelPath(self.parent, c)$
		   if $s \in \mathbb{E}_{Doc}^{Doc}$ **then**
		     new $List(self.C, Re, \{s\}, \varnothing, self.parent)$
		   else
		     new $List(self.C, Re, \varnothing, \{s\}, self.parent)$
		   for each each $r \in (Re - keep)$ **do**
		     $self.removeNode(r)$
		 $self.removeNode(s)$

$E_{Struct}^{Doc}.nest($ $r : A_C$, $Re :$ Set of E_R, $At :$ Set of A, $Do :$ Set of E_{Doc}^{Doc}, $Li :$ Set of E_{List}^{Doc})	$r \in self$	new $Document(self.C, r, Re, At, Do, Li, self)$
		 $keep \leftarrow \varnothing$
		 for each $c \in (self.children \cap \mathbb{E}_{Doc}^{Doc}) - Do$ **do**
		   $keep \leftarrow keep \cup FindRelPath(self, c)$
		 for each $c \in (self.children \cap \mathbb{E}_{List}^{Doc}) - Li$ **do**
		   for each $g \in (c.children \cap \mathbb{E}_{Doc}^{Doc})$ **do**
		     $keep \leftarrow keep \cup FindRelPath(self, g)$
		 for each $n \in ((Re \cup At \cup Do \cup Li) - keep)$ **do**
		   $self.removeNode(n)$

$E_{Set}^{Doc}.union(s : E_{Set}^{Doc})$	$self.parent = s.parent$	$self.addNode(s)$ $self.parent.removeNode(s)$ $s.dispose()$
$E_H^{Doc}.flatten()$	$self.parent \in \mathbb{E}_{Struct}^{Doc}$	$self.dispose()$

Appendix G

Validation of operations against MongoDB Design Patterns

In the following, we go through each of the patterns, digest them and use the same examples for illustration.[1] Nevertheless, our design transformations are defined at the logical level, so, some physical patterns (e.g., involving indexing) cannot be fully represented.

Attribute pattern (Fig. G.1) tries to identify a subset of fields that share some common characteristics that are frequently queried together, and add them into an array (e.g., release dates of the movies in different regions). The original document without the array has the field name suffixed by the region name, but since our immutable graph does not contain details about the instances, we simply use a counter instead as shown on `Design 4` in Fig. 5.5. Our set of transformation can implement this pattern by first using *nest* to create a flat embedded document, and then using *group* to make a list out of the documents.

Bucket pattern (Fig. G.2) groups specific data together (e.g., time series of sensor data). This pattern can be easily represented through our transformations by expliciting intervals a priori in the form of different *atom*s and *Relationship*s, since the predicates used for the bucket arrangement (e.g., start and end dates) cannot be generated as they are not part of the immutable graph. Moreover, the immutable graph must contain an additional A_C to identify each of the measurements. With that information, we can firstly use a sequence of *split*, *embed*, and *flatten* operations to change the root of the document (e.g., from measurement to sensor), then similar to the attribute pattern,
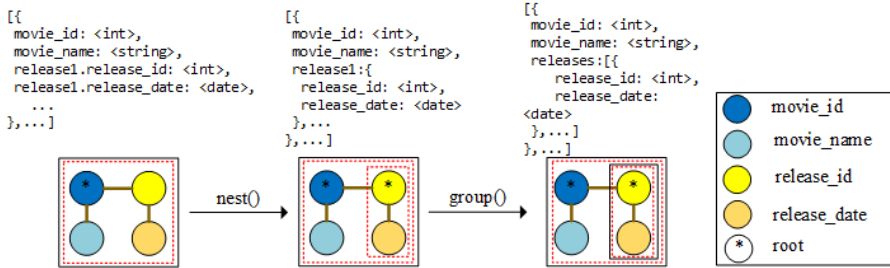

Fig. G.1: Transformations of the **Attribute** pattern

through *nest* and *group* we can create an embedded list with all measurements of the same sensor.
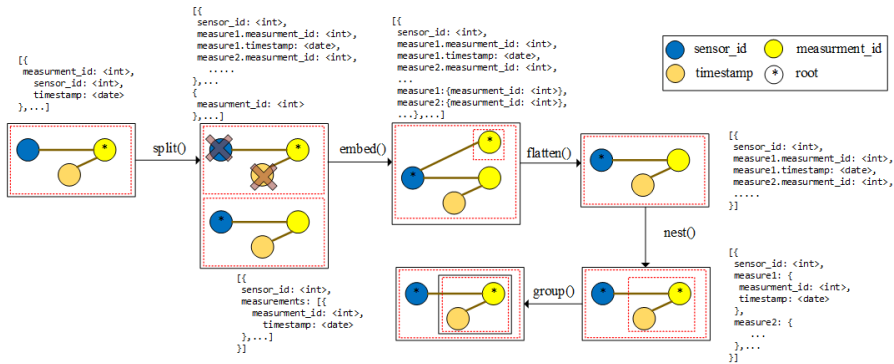

Fig. G.2: Transformations of the **Bucket** pattern

Polymorphic pattern (Fig. G.3) is used to merge collections with documents that share multiple attributes (e.g., Bowling and Tennis Athletes). We can easily deal with such transformation by *union*, *embed*, and *flatten* operations. However, currently, our hypergraph does not support specialization due to the complexity of guaranteeing the validity of the design (i.e., ensuring none of the subclasses/partitions are missed as a result of the transformations). Thus, this pattern can only be partially represented. However, it is possible to fully represent if the immutable graph contained the subclass information.

Extended reference pattern (Fig. G.4) is a mean of avoiding joins by embedding frequently accessed data of two entities (e.g., customer address in an order). First, we use *split* to extract from one document the information that needs to be embedded in the other (e.g., the address from the customer) and *segregate* this into a new collection. Then, we *union* the new collection and the one that requires embedding, to finally, join the information by *embed*, *flatten*, and *nest*.
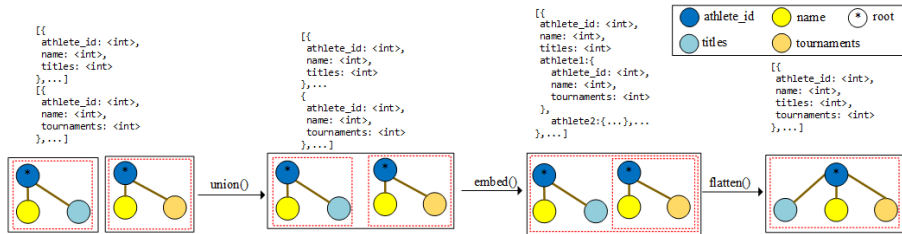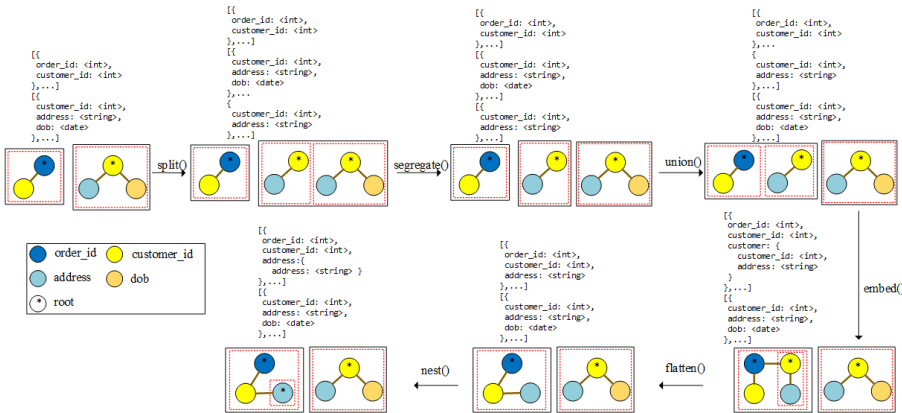
Fig. G.3: Transformations of the **Polymorphic** pattern

Fig. G.4: Transformations of the **Extended reference** pattern

Subset pattern (Fig. G.5) is used to prevent unnecessary growth of documents. A typical example would be to only store the first *n* documents in a list, keeping the rest of the list in a separate collection. As in the bucket pattern, *n* should be encoded somehow in the immutable graph (e.g., representing the two sets of documents in different classes). First, we remove the list with two consecutive *flatten* operations. Next, we *split* the document and *segregate* the part that needs to be moved to a new collection. Finally, we *nest* and *group* to recreate the original list.

The remaining seven MongoDB design patterns (namely **Outlier**, **Approximation**, **Computed**, **Document versioning**, **Preallocated**, **Schema versioning**, and **Tree and graph**) can be represented in our canonical model, provided the immutable graph contains the required information (e.g., schema/document version, the average of an attribute), but besides that, they require changes in the client application logic or the engine configuration rather than in the document design. Thus, they are out of the scope of this work.
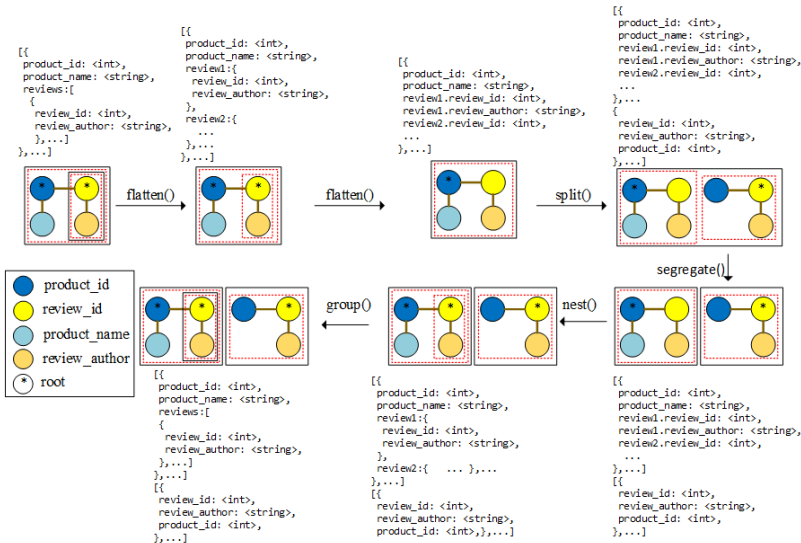

Fig. G.5: Transformations of the **Subset** pattern

References

[1] S. Abiteboul. Querying semi-structured data. In F. N. Afrati and P. G. Kolaitis, editors, *International Conference on Database Theory*, volume 1186, pages 1–18. ICDT, 1997.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD International Conference on Management of Data*, pages 527–538. ACM, 2003.

[3] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] S. H. Aboutorabi, M. Rezapour, M. Moradi, and N. Ghadiri. Performance evaluation of SQL and MongoDB databases for big e-commerce data. In *International Symposium on Computer Science and Software Engineering, CSSE*, pages 1–7, 2015.

[6] V. Abramova and J. Bernardino. Nosql databases: Mongodb vs cassandra. In *International C* Conference on Computer Science & Software Engineering, C3S*, pages 14–22. ACM, 2013.

[7] R. Aghi, S. Mehta, R. Chauhan, S. Chaudhary, and N. Bohra. A comprehensive comparison of SQL and MongoDB databases. *International Journal of Scientific and Research Publications*, 5(2), 2015.

[8] S. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley& Sons, 2003.

[9] W. W. Armstrong. Dependency structures of data base relationships. In J. L. Rosenfeld, editor, *Information ProcessingIFIP Congress*, pages 580–583, 1974.

[10] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, 67, 2020.

[11] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to non-relational database systems: The SOS platform. In *Int. Conf. on Adv. Inf. Sys. Eng., CAiSE*, pages 160–174, 2012.

[12] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to NoSQL systems. *Information Systems*, 43, 2014.

[13] A. Badia and D. Lemire. A call to arms: revisiting database design. *SIGMOD Rec.*, 40(3), 2011.

[14] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869, 2017.

[15] L. D. R. Beal, D. C. Hill, R. A. Martin, and J. D. Hedengren. GEKKO Optimization Suite. *Processes*, 6(8):106–131, 2018.

[16] E. Bertino and P. Foscoli. On Modeling Cost Functions for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, 1997.

[17] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language, 2007.

[18] A. Boicea, F. Radulescu, and L. I. Agapin. MongoDB vs Oracle–database comparison. In *International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*, pages 330–335. IEEE, 2012.

[19] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[20] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. Flexible hybrid stores: Constraint-based rewriting to the rescue. In *IEEE 32nd Int. Conf. on Data Engineering, ICDE*, 2016.

[21] F. Bugiotti, D. Bursztyn, U. C. S. Diego, and I. Ileana. Invisible glue : Scalable self-tuning multi-stores. *CIDR*, 2015.

[22] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. Database design for NoSQL systems. In *International Conference on Conceptual Modeling*. ER, 2014.

[23] C. J. F. Candel, D. S. Ruiz, and J. J. G. Molina. A unified metamodel for NoSQL and relational databases. *CoRR*, abs/2105.06494, 2021.

[24] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[25] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, pages 246–261, 2002.

[26] D. Che, K. Aberer, and M. T. Özsu. Query optimization in XML structured-document databases. *The VLDB Journal*, 15, 2006.

[27] J. Cho, Y. Wang, I. Chen, K. S. Chan, and A. Swami. A survey on modeling and optimizing multi-objective systems. *IEEE Commun. Surv. Tutorials*, 19(3):1867–1901, 2017.

[28] J. Clark, S. DeRose, et al. XML path language (XPath) version 1.0, 1999.

[29] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of ACM*, 13(6):377–387, 1970.

[30] P. Contos and M. Svoboda. JSON schema inference approaches. In *Advances in Conceptual Modeling - ER*, volume 12584, pages 173–183, 2020.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Symposium on Cloud Computing, SoCC*, pages 143–154. ACM, 2010.

[32] A. Dan and D. Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. *SIGMETRICS Performance Evaluation Review*, 18(1):143–152, 1990.

[33] R. Dautov and S. Distefano. Quantifying volume, velocity, and variety to support (big) data-intensive application development. In *IEEE International Conference on Big Data*, pages 2843–2852, 2017.

[34] A. de la Vega, D. García-Saiz, C. Blanco, M. E. Zorrilla, and P. Sánchez. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. *Future Generation Computer Systems*, 105, 2020.

[35] C. de Lima and R. dos Santos Mello. A workload-driven logical design approach for NoSQL document databases. In *International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 73:1–73:10, 2015.

[36] B. J. D'mello, M. Satheesh, and J. Krol. *Web Development with MongoDB and Node, 3rd Ed.* Packt Publishing, 2007.

[37] M. Duan and G. Chen. Assessment of MongoDB's spatial retrieval performance. In S. Hu, editor, *International Conference on Geoinformatics*. IEEE, 2015.

[38] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The bigdawg polystore system. *SIGMOD Record*, 44(2):11–16, 2015.

[39] J. Euzenat and P. Shvaiko. *Ontology Matching, Second Edition*. Springer, 2013.

[40] R. Fagin. Asymptotic Miss Ratios Over Independent References. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.

[41] D. Florescu. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. *Language*, 1999.

[42] M. Fruth, K. Dauberschmidt, and S. Scherzinger. Josch: Managing schemas for NoSQL document stores. In *International Conference on Data Engineering, ICDE*, pages 2693–2696. IEEE, 2021.

[43] E. Gallinucci, M. Golfarelli, and S. Rizzi. Schema profiling of document-oriented databases. *Information Systems*, 75:13–25, 2018.

[44] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[45] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[46] G. Gardarin, J. Gruser, and Z. Tang. A Cost Model for Clustered Object-Oriented Databases. In *International Conference on Very Large Data Bases*, pages 323–334, 1995.

[47] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases*. Addison-Wesley, 1989.

[48] P. Gómez, C. Roncancio, and R. Casallas. Towards quality analysis for document oriented bases. In *International Conference on Conceptual Modeling*, pages 200–216. ER, 2018.

[49] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[50] G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[51] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed*. Addison-Wesley, 1994.

[52] N. Grimsmo. Faster path indexes for search in XML data. In *Conferences in Research and Practice in Information Technology Series*, volume 75, 2008.

[53] F. Guo and Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. *SIGMETRICS Performance Evaluation Review*, 34(1):228–239, 2006.

[54] G. Haughian, R. Osman, and W. J. Knottenbelt. Benchmarking Replication in Cassandra and MongoDB NoSQL Datastores. In S. Hartmann and H. Ma, editors, *International Conference on Database and Expert Systems Applications {DEXA}*, volume 9828. Springer, 2016.

[55] M. Hausenblas and J. Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big data*, 1(2):100–104, 2013.

[56] R. Hecht and S. Jablonski. NoSQL Evaluation: A Use Case Oriented Survey. In *IEEE International Conference on Cloud and Service Computing*, pages 336–341, 2011.

[57] A. Hernández, F. Santiago, E. Calvo, G. Herzig, S. A. Ostapowicz, M. Melli, and J. D. Fernández. Performance Benchmark PostgreSQL/-MongoDB (Tech. R.). 2019.

[58] V. Herrero, A. Abelló, and O. Romero. NoSQL design for analytical workloads: Variability matters. In *International Conference on Conceptual Modeling*, pages 50–64. ER, 2016.

[59] T. Hills. *NoSQL and SQL Data Modeling: Bringing Together Data, Semantics, and Software*. Technics Publications, 2016.

[60] A. A. Imam, S. Basri, R. Ahmad, J. Watada, M. T. Gonzalez-Aparicio, and M. A. Almomani. Data Modeling Guidelines for NoSQL Document-Store Databases. *International Journal of Advanced Computer Science and Applications*, 9(10):544–555, 2018.

[61] Y. E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[62] K. Järvelin and J. Kekäläinen. IR evaluation methods for retrieving highly relevant documents. *SIGIR Forum*, 51(2), 2017.

[63] B. Jiang, P. Nain, and D. Towsley. LRU Cache under Stationary Requests. *SIGMETRICS Performance Evaluation Review*, 45(2):24–26, 2017.

[64] R. Johnson, J. Hoeller, K. Donald, M. Pollack, et al. The spring framework–reference documentation. *Interface*, 21, 2004.

[65] A. Kamsky. Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB. *PVLDB*, 12(12):2254–2262, 2019.

[66] A. Kanade, A. Gopal, and S. Kanade. A study of normalization and embedding in MongoDB. In *Advance Computing Conference (IACC)*, pages 416–421. IEEE, 2014.

[67] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. of the VLDB Endowment*, 9(12), 2016.

[68] M. S. Kester, M. Athanassoulis, and S. Idreos. Access path selection in main-memory optimized data systems: Should I scan or should I probe? In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *International Conference on Management of Data, SIGMOD*, pages 715–730. ACM, 2017.

[69] J. Kim, W. Lee, and K. Lee. The Cost Model for XML Documents in Relational Database Systems. In *IEEE International Conference on Computer Systems and Applications*, pages 185–187, 2001.

[70] W. F. King III. Analysis of Demand Paging Algorithms. In *IFIP Congress (1)*, pages 485–490, 1971.

[71] M. Klettke, U. Störl, and S. Scherzinger. Schema extraction and structural outlier detection for json-based nosql data stores. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 425–444, 2015.

[72] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`. Accessed: 2018-02-16.

[73] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira. The cloudmdsql multistore system. In *International Conference on Management of Data, SIGMOD*, pages 2113–2116. ACM, 2016.

[74] S. Y. Lee, M.-L. Lee, T. W. Ling, and L. A. Kalinichenko. Designing good semi-structured databases and conceptual modeling. In *International Conference on Conceptual Modeling*, page 131–145. ER, 1999.

[75] J. Lewis. *Cost-based Oracle fundamentals*. Apress, 2006.

[76] Y. T. Liao, J. Zhou, C. H. Lu, S. C. Chen, C. H. Hsu, W. Chen, M. F. Jiang, and Y. C. Chung. Data adapter for querying and transformation between SQL and NoSQL database. *Future Generation Computer Systems*, 65:111–121, 2016.

[77] S. Lightstone, T. J. Teorey, and T. P. Nadeau. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann, 2007.

[78] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In S. Ghandeharizadeh, S. Barahmand, M. Balazinska, and M. J. Freedman, editors, *Symposium on Cloud Computing, SoCC*, pages 153–166. ACM, 2015.

[79] Z. H. Liu, B. C. Hammerschmidt, D. Mcmahon, H. J. Chang, Y. Lu, J. Spiegel, A. C. Sosa, S. Suresh, G. Arora, and V. Arora. Native JSON datatype support: Maturing SQL and NoSQL convergence in Oracle database. *VLDB Endowment*, 13(12):3059–3071, 2020.

[80] T. F. Llano-Ríos, M. Khalefa, and A. Badia. Evaluating NoSQL systems for decision support: An experimental approach. In *International Conference on Big Data*, pages 2802–2811. IEEE, 2020.

[81] T. F. Llano-Ríos, M. Khalefa, and A. Badia. Experimental comparison of relational and NoSQL document systems: The case of decision support. In *TPC Technology Conference, TPCTC*, pages 58–74. Springer, 2020.

[82] S. Manegold, P. Boncz, and M. Kersten. Generic database cost models for hierarchical memory systems. Jan. 2002.

[83] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *International Conference on Very Large Data Bases*, pages 191–202, 2002.

[84] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[85] N. Megiddo and D. S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *IEEE Computer*, 37(4):58–65, 2004.

[86] E. Meijer and G. M. Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4), 2011.

[87] J. Michels, K. Hare, K. Kulkarni, C. Zuzarte, Z. H. Liu, B. Hammerschmidt, and F. Zemke. The New and Improved SQL: 2016 Standard. *ACM SIGMOD Record*, 47(2):51–60, 2018.

[88] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. *Transactions on Knowledge and Data Engineering*, 29(10):2275–2289, 2017.

[89] C. Mohan. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, 2013.

[90] T. Nguyen and S. Lee. I/O characteristics of MongoDB and trim-based optimization in flash SSDs. In C. K. Leung, J. Kim, Y. Kim, J. Geller, W. Choi, and Y. Park, editors, *International Conference on Emerging Databases: Technologies, Applications, and Theory, EDB*, pages 139–144. ACM, 2016.

[91] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, NoSQL and newsql databases. *CoRR*, abs/1405.3631, 2014.

[92] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems 4th ed.* Springer Science & Business Media, 2020.

[93] F. Pezoa, J. L. Reutter, F. Suárez, M. Ugarte, and D. Vrgoc. Foundations of JSON Schema. In *International Conference on World Wide Web, WWW*, pages 263–273, 2016.

[94] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Third edition, 2003.

[95] V. Reniers, D. Van Landuyt, A. Rafique, and W. Joosen. A Workload-Driven Document Database Schema Recommender (DBSR). In *Conceptual Modeling*, pages 471–484, 2020.

[96] M. A. Rodriguez. The gremlin graph traversal machine and language. *CoRR*, abs/1508.03843, 2015.

[97] S. Scherzinger and S. Sidortschuck. An Empirical Study on the Design and Evolution of NoSQL Database Schemas. *CoRR*, abs/2003.00054, 2020.

[98] P. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

[99] F. Saltor, M. Castellanos, and M. García-Solaco. Suitability of data models as canonical models for federated databases. *ACM Sigmod Record*, 20(4), 1991.

[100] J. Schindler. I/O Characteristics of NoSQL Databases. *Proceedings of the VLDB Endowment*, 5(12):2020–2021, 2012.

[101] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *International Conference on Very Large Data Bases*, 2002.

[102] R. Sellami, S. Bhiri, and B. Defude. Supporting multi data stores applications in cloud environments. *IEEE Transactions on Services Computing*, 9(1), 2016.

[103] P. Shvaiko and J. Euzenat. Ontology matching: State of the art and future challenges. *IEEE Transactions on knowledge and data engineering*, 25(1):158–176, 2013.

[104] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[105] A. J. Smith. Cache Evaluation and the Impact of Workload Choice. *SIGARCH Computer Architecture News*, 13(3):64–73, 1985.

[106] R. A. S. N. Soransso and M. C. Cavalcanti. Data modeling for analytical queries on document-oriented DBMS. In *ACM Symposium on Applied Comp.*, pages 541–548, 2018.

[107] M. Stonebraker. SQL databases v. NoSQL databases. *Communications of ACM*, 53(4):10–11, 2010.

[108] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson. Enabling query processing across heterogeneous data models: A survey. In *IEEE International Conference on Big Data*, pages 3211–3220, 2017.

[109] C. Truica, E. S. Apostol, J. Darmont, and T. B. Pedersen. The forgotten document-oriented database management systems: An overview and benchmark of native XML DODBMSes in comparison with JSON DODBMSes. *Big Data Res.*, 25:100205, 2021.

[110] C. Truica, F. Radulescu, A. Boicea, and I. Bucur. Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. In *International Conference on Control Systems and Computer Science, CSCS*, 2015.

[111] N. Vafaei, R. A. Ribeiro, and L. M. Camarinha-Matos. Data normalisation techniques in decision making: case study with TOPSIS method. *International Journal of Information and Decision Sciences*, 10(1):19–38, 2018.

[112] T. Vajk, L. Deák, K. Fekete, and G. Mezei. Automatic NoSQL schema development: A case study. In *Artificial Intelligence and Applications*, 2013.

[113] Á. Vathy-Fogarassy and T. Hugyák. Uniform data access platform for SQL and NoSQL database systems. *Information Systems*, 69, 2017.

[114] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache Modeling and Optimization using Miniature Simulations. In *USENIX Annual Technical Conference*, pages 487–498, 2017.

[115] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, et al. The myria big data management and analytics system and cloud services. In *Biennial Conference on Innovative Data Systems Research, CIDR*, 2017.

[116] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, and C. Wang. Schema Management for Document Stores. *PVLDB*, 8(9):922–933, 2015.

[117] Y. Widyani, H. Laksmiwati, and E. D. Bangun. Mapping spatio-temporal disaster data into MongoDB. In *International Conference on Data and Software Engineering*, 2017.

[118] J. Yao. An Efficient Storage Model of Tree-Like Structure in MongoDB. In *IEEE International Conference on Semantics, Knowledge and Grids*, pages 166–169, 2016.