

DATA MANAGEMENT FOR DYNAMIC MULTIMEDIA ANALYTICS AND RETRIEVAL

Inauguraldissertation

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

Ralph Marc Philipp Gasser

Basel, 2022

Zusammenfassung

Abstract

Acknowledgements

Good luck.

This work was partly supported by the Swiss National Science Foundation, which is also thankfully acknowledged.

Contents

| | |
|---|-------------|
| Zusammenfassung | iii |
| Abstract | v |
| Acknowledgements | vii |
| List of Figures | xiii |
| List of Tables | xv |
| List of Acronyms | xvii |
| List of Symbols | xix |
| | |
| I Introduction | 1 |
| 1 Introduction | 3 |
| 1.1 Working with Multimedia Data | 4 |
| 1.1.1 Multimedia Retrieval | 4 |
| 1.1.2 Multimedia Analytics | 5 |
| 1.1.3 Dynamic Data Management | 6 |
| 1.2 Research Gap and Objective | 7 |
| 1.2.1 Research Questions | 9 |
| 1.3 Contribution | 10 |
| 2 Applications and Use Cases | 13 |
| 2.1 Use case 1: Multimedia Retrieval System | 13 |
| 2.2 Use case 2: Analysis of Social Media Streams | 13 |
| 2.3 Use case 3: Magnetic Resonance Fingerprinting (MRF) | 13 |
| | |
| II Foundations | 15 |
| | |
| 3 On The Design of a Database Management System | 17 |
| 3.1 Data Management and Data Models | 18 |
| 3.2 The Relational Data Model | 19 |

| | | |
|----------|--|-----------|
| 3.2.1 | Keys and Normal Forms | 22 |
| 3.2.2 | Relational Algebra | 24 |
| 3.2.2.1 | Simple Set Operations | 24 |
| 3.2.2.2 | Cartesian Product | 25 |
| 3.2.2.3 | Rename | 26 |
| 3.2.2.4 | Selection | 26 |
| 3.2.2.5 | Projection | 26 |
| 3.2.2.6 | Natural Join | 26 |
| 3.2.2.7 | Expressing Queries | 27 |
| 3.2.3 | Extensions | 28 |
| 3.2.3.1 | Relations vs. Bag vs. Sequences | 28 |
| 3.2.3.2 | Extended Projection | 28 |
| 3.2.3.3 | Ranked Relational Algebra | 28 |
| 3.2.3.4 | Similarity Search and the Relational Model | 28 |
| 3.3 | Queries: From Expression to Execution | 30 |
| 3.3.1 | Structured Query Language (SQL) | 30 |
| 3.3.2 | Query Planning | 30 |
| 3.3.3 | Query Execution | 30 |
| 3.4 | Storage, Indexes and Caching | 30 |
| 3.5 | Architectual Considerations | 30 |
| 4 | On Multimedia Analysis and Retrieval | 31 |
| 4.1 | Multimedia Data and Multimedia Collections | 32 |
| 4.1.1 | A Data Model for (Multi-)Media Data | 34 |
| 4.1.2 | Descriptive Metadata | 36 |
| 4.1.3 | Features | 37 |
| 4.2 | Multimedia Retrieval | 39 |
| 4.2.1 | Similarity Search and the Vector Space Model | 40 |
| 4.2.1.1 | Nearest Neighbour Search (NNS) | 42 |
| 4.2.1.2 | Range Search | 42 |
| 4.2.1.3 | Reverse Nearest Neighbour Search (RNNS) | 43 |
| 4.2.2 | NNS and High-Dimensional Index Structures | 44 |
| 4.2.2.1 | Vector Approximation Files (VAF) | 45 |
| 4.2.2.2 | Locality Sensitive Hashing (LSH) | 46 |
| 4.2.2.3 | Product Quantisation (PQ) | 47 |
| 4.2.3 | Beyond Similarity Search | 49 |
| 4.3 | Online Multimedia Analysis | 49 |
| 4.4 | Multimedia Analytics | 49 |

| | | |
|---|--|-----------|
| 4.5 | Systems | 49 |
| III Dynamic Multimedia Data Management | | 51 |
| 5 | Modelling a Database for Dynamic Multimedia Data | 53 |
| 5.1 | Generalized Proximity Based Operations | 54 |
| 5.1.1 | Revisiting Distance Computation | 55 |
| 5.1.2 | Extending the Relational Model | 59 |
| 5.1.2.1 | Algebraic Properties of π_E, τ_O, λ_k | 62 |
| 5.1.3 | DFCs and Query Planning | 65 |
| 5.1.3.1 | Functions and Optimised Implementations | 66 |
| 5.1.3.2 | DFCs and High-dimensional Index Structures | 67 |
| 5.2 | Cost Model for Retrieval Accuracy | 68 |
| 5.3 | Adaptive Index Management | 68 |
| 5.4 | Architecture Model | 69 |
| 6 | Cottontail DB | 71 |
| 6.1 | Data Model and Nomenclature | 71 |
| 6.2 | Architecture | 73 |
| 6.2.1 | Query Implementation and Execution | 74 |
| 6.2.1.1 | Execution Model | 75 |
| 6.2.2 | Query Planner | 76 |
| 6.2.2.1 | Basic Optimization Algorithm | 77 |
| 6.2.2.2 | Plan Caching | 78 |
| 6.2.2.3 | Logical Optimization | 79 |
| 6.2.2.4 | Physical Optimization | 79 |
| 6.2.3 | Query Parsing and Binding | 81 |
| 6.2.3.1 | Binding Context | 82 |
| 6.2.4 | Functions and Function Generators | 83 |
| 6.2.5 | Storage | 84 |
| 6.2.5.1 | Storage engine: MapDB | 85 |
| 6.2.5.2 | Storage engine: Xodus | 85 |
| 6.2.5.3 | Storage engine: HARE | 86 |
| 6.2.6 | Transactional Guarantees | 87 |

| | |
|--|---------------|
| IV Discussion | 89 |
| 7 Evaluation | 91 |
| 7.1 Interactive Multimedia Retrieval | 91 |
| 7.2 Adaptive Index Management | 91 |
| 7.3 Cost Model | 91 |
| 8 Related Work | 93 |
| 9 Conclusion & Future Work | 95 |
| Appendix | 97 |
| Bibliography | 99 |
| Curriculum Vitae | 111 |
| Declaration on Scientific Integrity | 117 |

List of Figures

| | | |
|-----|--|----|
| 1.1 | Exploration-search axis of multimedia analytics [ZW14]. | 6 |
| 3.1 | Different levels of abstraction for data modelling. | 18 |
| 4.1 | An extended ERM of the multimedia data model used in the <i>vitriivr</i> project. The model is centered around the notion of media objects and media segments which are described by metadata and features. . | 35 |
| 4.2 | An extended ERM of the multimedia data model used for the purpose of this Thesis. It has been derived from <i>vitriivr</i> 's data model and foregoes the explicit segment entity. | 36 |
| 4.3 | Flow of information in similarity search at ingest- and query time. . . | 40 |
| 4.4 | Graphical illustration of the different similarity search algorithms in $\mathcal{F} \subset \mathbb{R}^2$ using the Euclidean distance (L^2). Matches are highlighted in red. | 44 |
| 4.5 | Simplified illustration of basic VAF indexing scheme in for $\mathcal{F} \subset \mathbb{R}^2$. A vector is assigned to a cell which gives rise to a signature that is stored in a linear list. | 46 |
| 4.6 | A simplified illustration of the basic LSH indexing scheme. A vector is hashed to a bucket by a locality preserving hash function h . The vector is then stored in the assigned bucket in a hash table. | 47 |
| 4.7 | A simplified illustration of the basic PQ indexing scheme. A vector is divided into smaller subvectors and each subvector is assigned to the nearest centroid in its respective codebook. The centroid ID per subspace gives rise to the signature, which can be stored in an inverted file index. | 48 |
| 5.1 | Two examples of distance functions that fall into the broader category of a DFC. On the left, the point-distance between q and p_i and on the right, the distance between a hyperplane $q^T x + b = 0$ and points p_i | 58 |
| 5.2 | Function hierarchy of all $f \in \mathbb{F}$. As we go down the hierarchy, the DBMS gains knowledge about a function's structure. | 65 |
| 5.3 | Adaptive index structures overview. | 68 |

-
- 6.1 Architecture diagram of *Cottontail DB*'s main components. The directed arrows indicate the path a query takes within the system. The dashed, double arrows indicate interactions between components. . . 73
 - 6.2 Operator pipeline for a proximity based query that scans the entity *paintings*, calculates the L2 distance between a *feature* column and a query, orders by the obtained distance and limits the number of outputs (kNN). Pipeline breaking operators are indicated by the red dot. 75
 - 6.3 Operator pipeline that performs kNN on the entity *paintings*. The *scan* and distance *function* steps are partitioned and unioned before order, limit and projection is applied, which allows for parallel execution. 76
 - 6.4 Illustration of deferral of column access. In the first pass, access to columns *id* and *lastname* is deferred until after the filter operation, because the filter only requires *firstname*. In the second pass, access to *lastname* is eliminated completely. 80
 - 6.5 Illustration of decomposing conjunctive predicates. In this example, the filter on the column *firstname* may be pushed down to an index during physical optimization, if such an index is available. 80

List of Tables

| | | |
|-----|---|----|
| 1.1 | List of research questions (RQ) resulting from challenging assumptions(AS) one, two and three. | 9 |
| 3.1 | The relational operators proposed by Codd et al [Cod70; GUW09]. . . | 25 |
| 4.1 | List of distance functions often used in similarity search. | 41 |
| 4.2 | List of high-dimensional index structures employed in similarity search and retrieval. | 45 |
| 5.1 | The relational operators and their properties w.r.t to the proposed extensions. | 63 |
| 6.1 | Data types supported by <i>Cottontail DB</i> . Types in the numeric, vector and complex class allow for domain specific arithmetics. | 72 |
| 6.2 | Main types of operators implemented by <i>Cottontail DB</i> alongside with their arity, their correspondence to relational operators and whether or not they require materialization. | 74 |
| 6.3 | Primitive operations offered by the storage engines's Tx abstractions to interact with DBOs including the argument and return types (arg \rightarrow ret). | 84 |

List of Acronyms

| | |
|------|---|
| 1NF | First Normal Form |
| 2NF | Second Normal Form |
| 3NF | Third Normal Form |
| ACID | Atomicity, Consistency, Isolation, Durability |
| BCNF | Boyce–Codd Normal Form |
| CWA | Closed-world assumption |
| DBMS | Database Management System |
| DBO | Database Object |
| DDL | Data Definition Language |
| DFC | Distance Function Class |
| DML | Data Manipulation Language |
| DQL | Data Query Language |
| ERM | Entity Relationship Model |
| FIFO | First In First Out |
| FK | Foreign Key |
| FNS | Farthest Neighbour Search |
| HOG | Histogram of Oriented Gradients |
| JDBC | Java Database Connectivity |
| JVM | Java Virtual Machine |
| kFN | k-Farthest Neighbour Search |
| kNN | k-Nearest Neighbour Search |
| LSC | Lifelog Search Challenge |
| LSH | Locality Sensitive Hashing |

| | |
|------|--|
| MFCC | Mel-frequency Cepstral Coefficient |
| MIPS | Maximum Inner Product Search |
| MRF | Magnetic Resonance Fingerprinting |
| MVCC | Multi-Version Concurrency Control |
| | |
| NNS | Nearest Neighbour Search |
| | |
| ODBC | Open Database Connectivity |
| | |
| PCP | Pitch Class Profiles |
| PK | Primary Key |
| PQ | Product Quantisation |
| | |
| RNNS | Reverse Nearest Neighbour Search |
| | |
| S2PL | Strict 2-Phase Locking |
| SH | Spectral Hashing |
| SIFT | Scale Invariant Feature Transformation |
| SIMD | Single Instruction Multiple Data |
| SQL | Structured Query Language |
| SURF | Speeded-Up Robust Features |
| | |
| VAF | Vector Approximation Files |

List of Symbols

Basics

| | |
|--------------|-------------------------------------|
| \mathbb{N} | Set of natural numbers. |
| \mathbb{R} | Set of real numbers. |
| \wedge | The logical conjunction. |
| \vee | The logical, inclusive disjunction. |
| \neg | The logical negation. |

Relational Algebra

| | |
|--------------------------|---|
| \mathcal{R} | A relation. Sometimes with subscript, e.g., $\mathcal{R}_{\text{name}}$ to specify name or index. |
| \mathcal{R}^O | A ranked, relation that exhibits a partial ordering of elements induced by O . Sometimes used with subscript, e.g., $\mathcal{R}^O_{\text{name}}$ to specify name or index. |
| \mathcal{D} | A data domain of a relation. Sometimes with subscript, e.g., $\mathcal{D}_{\text{name}}$, to specify name or index. |
| \mathcal{A} | An attribute of a relation, i.e., a tuple (name, \mathcal{D}). Sometimes with subscript to specify name or index. |
| \mathcal{A}^* | An attribute that acts as a primary key. |
| $\overline{\mathcal{A}}$ | An attribute that acts as a foreign key. |
| t | A tuple $t \in \mathcal{R}$ of attribute values $a_i \in \mathcal{D}_i$. Sometimes with subscript to specify the index of the tuple in \mathcal{R} . |
| \mathbb{D} | The set system of all data domains \mathcal{D} supported by a DBMS. |
| SCH | The schema of a relation \mathcal{R} , i.e., all attributes $\text{SCH}(\mathcal{R}) = (\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_N)$ that make up \mathcal{R} . |
| σ | The unary selection operator used to filter a relation \mathcal{R} . Usually printed with a subscript to describe the boolean predicate, e.g. σ_p . |
| π | The unary projection operator used to restrict the attributes \mathcal{A} of a relation \mathcal{R} . Usually used with subscript to list the desired attributes, e.g. $\pi_{\mathcal{A}_1, \mathcal{A}_2}$. |

| | |
|-----------|--|
| τ | The binary sort operator used to construct a ranked relation $\mathcal{R}^{\mathcal{O}}$. |
| λ | The unary k-selection operator restricts \mathcal{R} to the first k tuples. |

Multimedia Retrieval

| | |
|----------------|---|
| $\hat{\delta}$ | A Distance Function Class (DFC) $\hat{\delta}: \mathcal{D}_q \times \mathcal{D}_q \times \mathcal{D}_1 \cdot \times \mathcal{D}_n \rightarrow \mathbb{N}$ |
| \mathcal{F} | The feature domain, which is a set of features $\mathcal{F} = \{f_1, f_2, \dots, f_M\}$. |
| ϕ | A transformation function that maps from a media collection domain \mathcal{C} to a feature domain \mathcal{F} . |
| \mathcal{C} | The media collection domain, which is a set of media objects $\mathcal{C} = \{o_1, o_2, \dots, o_N\}$. |

PART I

Introduction

1

Introduction

The term *multimedia* describes the combination of different forms of digital media – also called *modalities* – into a single, sensory experience that carries a higher level semantic. Those modalities include but are not limited to images and videos (visual), music, sound effects and speech (aural) or textual information. However, more exotic media types such as 3D models or signals produced by sensors can also be seen as modalities, even though experience by a human consumer may depend on pre-processing by specialized hard- and software.

Nowadays, people encounter digital media and multimedia on a daily basis when watching videos on Netflix or YouTube, when listening to music on Spotify or when browsing a private image collection on their laptop. (Multi-)media content makes up a large part of today's Internet and constitutes a major driving force behind its growth, as both volume and variety increases at an ever increasing pace. An important contributing factor are social media platforms, where users act both as consumers and producers of digital content. Current estimates suggest, that there are roughly 4.66 billion active Internet users worldwide, of which 4.2 billion can be considered active social media users¹. Facebook alone contributed to 144 thousand uploaded images per minute in 2020. And many more of these platforms, such as Instagram or Twitter, serve millions of users with mixed, self-made content involving text, images, videos or a combination thereof. A similar study found, that by 2025 we will produce a yearly amount of 175 Zettabytes (i.e, 10^{21} bytes) worth of data².

Looking at these numbers, the need for efficient and effective tools for *managing, manipulating, searching, exploring* and *analysing* multimedia data corpora becomes very apparent, which has given rise to different areas of research.

¹ Source: Statista.com, "Social media usage worldwide", January 2021

² Source: Statista.com, "Big Data", January 2021

1.1 Working with Multimedia Data

On a very high level, multimedia data collections consist of individual multimedia items, such as video, image or audio files. Each item, in turn, comprises of *content*, *annotations* and *metadata*. Unlike traditional data collections that contain only text and numbers, the content of the multimedia item itself is unstructured on a data level, which is why *feature representations* that reflect a media item's content in some way and that can be handled by data processing systems are required [ZW14]. Traditionally, such feature representations have often been numerical vectors $f_i \in \mathbb{R}^d$. However, in theory, any mathematical object that can be processed by a computer can act as a feature.

Multimedia analysis, which has its roots in *computer vision* and *pattern recognition* and started in the early 1960s and deal with the automated, computer-aided analysis of visual information found in images and later videos, i.e., the extraction and processing of feature representations. In the early days of computer vision, a lot of effort went into the engineering of feature representations that captured certain aspects of a media item's content, such as the colour distribution, texture or relevant keypoints [Low99; BTVG06] in an image. Once such features have been obtained, they can be used to perform various tasks such as classification, clustering or statistical analysis. With the advent of deep learning, the extraction of features could largely be automated through neural network architectures such as the *Convolutional Neural Network (CNN)* and sometimes even be integrated with the downstream analysis [GBC16].

Obviously, such analysis is not restricted to the visual domain and can be applied to other types of media such as speech, music, video or 3D models with specific applications, such as, speech recognition, audio fingerprinting in music, movement detection in videos or classification of 3D models, all of which fall into the broader category of multimedia analysis.

1.1.1 Multimedia Retrieval

Traditionally, multimedia retrieval or content-based retrieval could be seen as a special niche within the multimedia analysis domain. It constitutes a dedicated field of research that deals with searching and finding items of interest within a large (multi-)media collection. Even though this may sound like the main function of a database, it is a very different task for multimedia than it is for structured data [BdB⁺07]. On the one hand, given the structure of a relational database and languages like SQL, a user can specify exactly what ele-

ments from the database should be selected using predicates that either match or don't match the items in a collection. For example, when considering a product database that contains price information for individual items, it is trivial to formulate a query that selects all items above a specific price threshold.

Retrieving multimedia data, on the other hand, comes with indirections due to the unstructured nature of the content, the feature representations used as a proxy for it and the *semantic gap* associated with these representations. A very popular model to work with the feature representations involves calculation of (dis-)similarity scores from the features and sorting and ranking of items based on this score. This is commonly referred to as the *vector space model* of multimedia retrieval and similarity search. Over the years, many different combinations of features and ranking models have been proposed to facilitate content-based retrieval of different media types, such as, images, audio or video as have been different types of query formulation, such as *Query-by-Sketch*, *Query-by-Humming* or *Query-by-Sculpting* [CWW⁺10; GLC⁺95; BGS⁺20].

Furthermore, when looking at concrete system implementations that facilitate interactive multimedia retrieval for an end-user today, the lines between multimedia retrieval and multimedia analytics quickly start to blur. This is because, in addition to the extraction of appropriate features and the conception of effective ranking algorithms, multimedia retrieval systems today also concern themselves with aspects such as query (re-)formulation and refinement, results presentation and efficient exploration [LKM⁺19]. In addition, multimedia retrieval systems do not simply operate on features representations anymore but combine *similarity search* on features and *Boolean retrieval* on annotations and metadata [RGL⁺20]. Therefore, one could argue that multimedia retrieval systems perform a very specific type of multimedia analytics task, which is that of finding unknown items that satisfy a specific information need. This makes all the arguments made about data processing and data management requirements for multimedia analytics applicable to multimedia retrieval as well.

Add
sources:
Survey
for each
modality

1.1.2 Multimedia Analytics

Multimedia analytics aims at generating new knowledge and insights from multimedia data by combining techniques from multimedia analysis and visual analytics. While multimedia analysis deals with the different media types and how meaningful representations and models can be extracted from them, visual analytics deals with the user's interaction with the data and the models themselves [CTW⁺10; KKE⁺10]. Simply put, multimedia analytics can be seen as a

back and forth between multimedia (data) analysis and visual analytics, whereas analysis is used to generate models as well as visualisations from data which are then examined and refined by the user and their input. This is an iterative process that generates new knowledge and may in and by itself lead to new information being attached to the multimedia items in the collection.

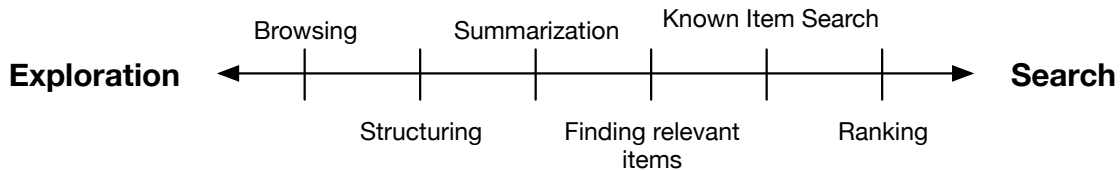


Figure 1.1 Exploration-search axis of multimedia analytics [ZW14].

For analytics on a multimedia collection, Zahalka et al. [ZW14] propose the formal model of an *exploration-search axis*, which is depicted in Figure 1.1. The model is used to characterize the different types of tasks carried out by the user. The axis specifies two ends of a spectrum, with *exploration* marking one end – in case the user knows nothing about the data collection – and *search* marking the other end – in case the user knows exactly which specific items of a collection they’re interested in. During multimedia analytics, a user’s activities oscillate between the two ends of the spectrum until the desired knowledge has been generated. Unsurprisingly, all of the depicted activities come with distinct requirements on data transformation and processing.

As data collections become large enough for the relevant units of information – i.e., feature representations, annotations and metadata – to no longer fit into main memory, multimedia analytics and the associated data processing quickly becomes an issue of scalable data management [JWZ⁺16]. This data management aspect becomes very challenging when considering the volume and variety of the multimedia data, the velocity at which new data is generated and the inherently unstructured nature of the media data itself.

1.1.3 Dynamic Data Management

Finetune!

It is important to emphasize, that a media item can comprise of all of the aforementioned components and that a multimedia collection may contain items of different types. Furthermore, when looking at a media item’s lifecycle, all of the aforementioned aspects are not static; annotations, metadata and features may either be generated upon the item’s creation (e.g., for technical metadata), as a

result of data-processing and analysis or by manually adding the information at some stage. Hence, any data management system must be able to cope with changes to that information. Those requirements are formalized in Section 4.1.

1.2 Research Gap and Objective

It has been pointed out by Jonson et al. [JWZ⁺16] (p. 296) that “Multimedia analytics state of the art [...] has up to now [...] not explicitly considered the issue of data management, despite aiming for large-scale analytics.”. Despite recent advances and the development of concrete architecture models for multimedia database management [GS16; Gia18] and multimedia retrieval systems that refactor data management into distinct components [Ros18], that statement, to some extent, still holds true today. While [Gia18] makes important contributions towards a unified data-, query- and execution model required for effective search and exploration in multimedia collections, scalability aspects and the need for near real-time query performance, especially in the face of dynamic data, are not systematically considered. On the contrary, the proposed models – despite being seminal for data management in certain multimedia retrieval applications – postulate assumptions, that have considerable impact on the practical applicability of data management systems implementing them.

The starting point for the research described in this thesis is therefore the current state-of-the-art for data management in multimedia retrieval and analytics as briefly touched upon in the previous sections. Starting from and inspired by the models and solutions proposed in [GS16; Gia18] and motivated by the “Ten Research Questions for Scalable Multimedia Analytics” [JWZ⁺16], this thesis challenges three basic assumptions currently employed and operated upon in multimedia data management and explores the ramifications of doing so, with the higher level goal of bridging certain gaps between research conducted in multimedia retrieval, analysis and analytics on the one hand, and classical data management and databases on the other. These assumptions are namely:

Assumption 1: Staticity of data collections Most multimedia retrieval systems today make a distinction between an *offline* phase during which media items are analysed, features are generated and derived data is ingested into a data management system, and an *online* phase, during which queries of the data management system take place. Usually, no changes to the data collection are being made during the online phase. This model is proposed by both [Gia18]

and [Ros18] and to the best of our knowledge, most existing multimedia retrieval and analytics systems implement this either explicitly or implicitly. This simplification allows for time consuming processes related to feature extraction and indexing to take place separated from any concurrent query activities and eases requirements on transaction isolation.

Assumption 2: Nearest neighbor search The vector space model used in multimedia retrieval relies on a notion of similarity search that is usually expressed as finding the k nearest neighboring feature vectors $\vec{v}_{i \in [1, k]} \in C$ to a query vector $\vec{q} \in \mathbb{R}^d$ in a collection $C \subset \mathbb{R}^d$ given a certain distance function. Very often, metrics such as the Euclidean or the Manhattan distance are employed for this comparison. While this model is very concise, computationally efficient and rather simple, it merely allows for the ranking of potential results and, given that the underlying model and the query is precise enough, finding the relevant or desired item(s).

Assumption 3: User defines execution Database management systems usually evaluate and select the execution plan for an incoming query during a step that is referred to as *query planning*. The underlying assumption here is that the database system has all the information required to determine the most effective execution path in terms of cost parameters such as required I/O, CPU and memory usage. In multimedia retrieval, this is not the case since, for example, index selection relies on a lot of different aspects that, to some extent, can be parametrized by the client issuing a query or that may be subject to change. Therefore, the index used for executing a query is often selected explicitly by the user issuing the query.

It is worth noting, that Assumption 1 and 3 both go against well-established design principles usually found in modern database systems [Pet19]. While it may be convenient from a perspective of system design, to assume a data collection to be static, such a mode of operation is utterly limiting when considering data that is subject to change, as is the case, for example, when doing analytics or when having an application with CRUD support. A similar argument can be made for manual index selection. Such an assumption may be simplifying the process of query planning but assumes, that a user is always a technical expert. Furthermore, it limits the amount of optimization that can be applied by the data management system especially in the face of non-static data collections, where indexes are changing, or changing query workloads.

As for Assumption 2, one can state that the described model is only able to accommodate the search-end of the *exploration-search axis*, assuming that features are, in fact, real valued vectors. It quickly becomes unusable for tasks such as browsing, structuring and summarization, delegating the required data processing to upper-tier system components. Referring to [JWZ⁺16], it would however be desirable to offer such primitives at the level of the data management system.

Transition from assumptions via overarching research goal to research question.

1.2.1 Research Questions

Challenging the aforementioned assumptions raises very specific questions that fundamentally impact the design of a *multimedia data management system*. These questions are briefly summarized in Table 1.1.

Table 1.1 List of research questions (RQ) resulting from challenging assumptions(AS) one, two and three.

| RQ | Question | Assumption | Domain |
|----|--|------------|--------|
| 1 | Which commonly used, secondary index structures for NNS (e.g., VA [WSB98], LSH [Indyk:1998Approximate], PQ [jegou:2011pq] based indexes) can cope with changes to data and to what extent? | AS 1 | |
| 2 | Can we estimate and quantify deterioration of retrieval quality of index structures from RQ1 as changes are being made to the underlying data collections? | AS 1 | |
| 3 | How can we handle index structures from RQ1 for which to expect deterioration during query planning and execution? | AS 1 | |
| 4 | Can we devise a model that (temporarily) compensates deterioration of retrieval quality of index structures? | AS 1 | |
| 5 | How can user knowledge about the the retrieval task at hand be factored into query planning without forcing the user the make explicit choices about how a query should be executed? | AS 1 & 3 | |
| 6 | How would a cost model that factors in desired retrieval accuracy look like and can it be applied during query planning? | AS 3 | |
| 7 | Assuming the cost model in RQ6 exists, at what levels of the system can it be applied (globally, per query, context)? | AS 3 | |
| 8 | Is there a measurable impact (e.g., on query execution time vs. accuracy) of having such a cost model? | AS 3 | |
| 9 | Can we generalize the model for similarity search (i.e., the vector space model) and what is the consequence of doing so? | AS 2 | |
| 10 | Do the existing applications and use-cases justify a generalization? | AS 2 | |

Associated research questions to domains (retrieval, analytics, dynamic).

RQ1 to RQ5 address the issue of index structures for NNS, which are mostly unable to cope with data that is subject to change, since their correctness deteriorates as data is modified. The focus of these questions are whether deterioration can be quantified and how it can be handled by a system. We argue, that both is necessary for practical application in dynamic data management.

RQ6 to RQ8 explore the possibility of a cost model, that takes accuracy of the produced results into account. Since most techniques for fast NNS rely on approximation, inaccuracy is an inherent factor for such operations. Assuming such a model exists, it can be used by a user or system administrator to make explicit choices between either accuracy or execution performance. In addition, such a cost model can be put to use when deciding what indexes to use in face of deteriorated retrieval quality due to changing data.

And finally, RQ9 and RQ10 address the issue of a more generalized model for similarity search and the impact of such a model on all the different system components. Most importantly, however, they explore and justify the need for such a model, which is not self-evident, based on concrete use-cases and applications.

1.3 Contribution

In this thesis, we try to address the research gap identified and described in Section 1.2 and thereby try to bridge the disparity between the fields of databases and retrieval systems. The contribution of this thesis can be summarized as follows:

- We examine the impact of challenging the *data staticity assumption* on index structures commonly used for nearest neighbor search. Most importantly, we describe a model to *quantify* the effect of changing data for commonly used structures and to *expose* that information to the data management system.
- We describe an *adaptive index management* model by which a data management system can compensate errors introduced at an index level due to changes to the underlying data. The main design goal for that mechanism is, that it can be employed regardless of what type of index is used underneath.
- We propose a *cost-model that factors-in accuracy* of generated results in addition to common performance metrics, such as IO-, CPU-, or memory-usage

and, based on that model, derive mechanisms for the user to express their preference for either accuracy or speed at different levels of the system.

- We postulate a more *generalized model for similarity search* and explore implications of such a model on aspects, such as, query planning.
- We introduce a working implementation that implements the aforementioned models, in the form of *Cottontail DB* [GRH⁺20].
- We present an evaluation of the impact of the model changes on real world datasets to provide a basis upon which their applicability can be assessed.

The described contributions are presented in four parts: The first part, to which this introduction belongs, introduces the problem and provides motivating use-cases and applications (Chapter 2) as well as an overview of relevant research done in the fields of multimedia retrieval, multimedia analytics and data management (Chapter 8).

The second part, gives a brief summary of the theoretical foundation in multimedia analysis & retrieval (Chapter 4) and databases (Chapter 3) required to understand the remainder of this thesis. The function of these chapters is that of a refresher for readers not familiar with either domain.

The third part introduces the theoretical models that make up the aforementioned contributions (Chapter 5) and introduces our reference implementation Cottontail DB (Chapter 6).

The fourth and final part presents the evaluation using Cottontail DB as an implementation (Chapter 7) and discusses the conclusions and potential future work (Chapter 9)

2

Applications and Use Cases

2.1 Use case 1: Multimedia Retrieval System

vitivr, vitivr VR with focus on search and exploration and data mangement & query implications (e.g., for SOMs, staged querying etc.). It remains to be seen how changes to data can be motivated here.

2.2 Use case 2: Analysis of Social Media Streams

Online analysis in Pythia, Delphi. Mainly as a motivating use case for why data may be subject to change.

Demo paper!

2.3 Use case 3: Magnetic Resonance Fingerprinting (MRF)

MRF as a concrete example why the classical NNS is too limited for certain use cases and an extension should be considered.

Paper!

PART II

Foundations

3

Under third normal form, a non-key field must provide a fact about the key, use the whole key, and nothing but the key.

— William Kent

On The Design of a Database Management System

Database Management System (DBMS), or simply “databases”, power everything from small and simple websites to large data warehouses that serve millions of users in parallel. Database systems play a crucial role in banking, e-commerce, science, entertainment and practically every aspect of our socio-economic lives. The first commercial Database Management System (DBMS) were introduced in the 1960s [GUW09] and they have evolved ever since to adapt to a wide range of requirements. Even though many different flavours of DBMS have emerged over the years, at their core, they still serve the same, fundamental purpose:

Management DBMS enable their users to organize data corpora that can range from a few megabytes to hundreds of terabytes according to a defined data model. For example, data can be structured into documents, tables, trees or graphs that in turn can be organized into collections or schemata.

Definition DBMS provide users with the ability to alter the organization of their data within the constraints of the data model using a Data Definition Language (DDL).

Manipulation DBMS provide users with the ability to modify the data within the constraints of the data model using a Data Manipulation Language (DML). Modifications may include adding, removing or changing entries.

Querying DBMS provide users with the ability to query the data using a Data Query Language (DQL). Such queries can be used to answer specific “questions” about the data.

Guarantees DBMS usually provide guarantees, such as assuring durability upon failure or providing access control for concurrent read and write operations. A well-known set of guarantees offered by many modern DBMS is known by its acronym ACID – which stands for **A**tomicity, **C**onsistency, **I**solation and **D**urability [HR83].

Since the contributions of this Thesis rely on decades of database research, we use this chapter to provide a brief overview over the relevant fundamentals and related work. Most of the fundamental aspects are inspired and guided by [GUW09] and [Pet19].

3.1 Data Management and Data Models

A data model is a formal framework that describes any type of data or information. It usually involves a formal description of the data’s *structure*, the *operations* that can be performed and the *constraints* that should be imposed on the data [GUW09]. The purpose of any data model is to formalize how data governed by it can be accessed, modified and queried and any given DBMS usually adopts a specific type of data model.

In the context of database systems and data management, it has become common practice to distinguish between different levels of abstraction for data models, as can be seen in Figure 3.1. At the top, there is the *conceptual data model*, which is often closely tied to some real-world scenario. For example, in an online shop, one can think in terms of customers that order things, products that are being sold and orders that have been placed. In the Entity Relationship Model (ERM) [Che76] – a popular framework used to describe conceptual data models – those “real things” would be modeled as *entities* that come with dedicated

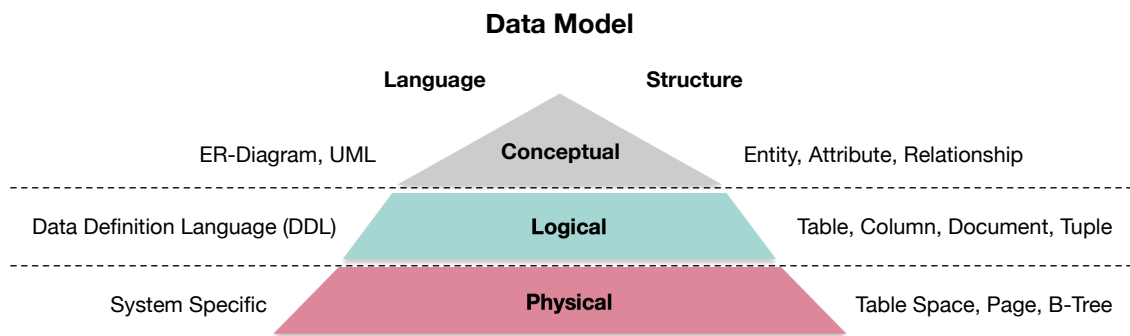


Figure 3.1 Different levels of abstraction for data modelling.

attributes that describe them and can have certain *relationships* among them as required by the concrete application.

The *logical data model* is more closely tied to the type of database being used. Different data models have been conceived over the years, including, but not limited to models centered around documents, key-value pairs, tables or graph structures, each bringing the own advantages and disadvantages. At this level, one can use DDL to describe and modify the data organization and DQL or DML to query and modify the data. As an example, one could store each entity described before in a dedicated table wherein each attribute occupies a different column – this is known as the relational model. A very important database language for the relational model would be the Structured Query Language (SQL), which became an international standard in 1987 under *ISO 9075* and has evolved ever since [Cha12].

At the very bottom, we usually find the *physical data model*, which is specific to the database implementation and describes low-level aspects such data access in terms of *read* and *write* operations to data structures such as *table spaces*, *data pages* or *trees*. An important argument in favour of separating logical from physical data model, even though they are both somewhat specific to a DBMS implementation, is that the end-users should not concern themselves with how exactly data is organized and accessed at the lowest level but should instead describe their intent in terms of the information they are trying to access. Mapping this user-intent to low-level data access operations is then a task of the DBMS.

3.2 The Relational Data Model

In June 1970, E. F. Codd published his pivotal research article *Relational Model of Data for Large Shared Data Banks* [Cod70] in which he describes a logical data model, which he himself refers to as “relational”. This model has become the fundament on which many modern database management systems have been built in the past decades, with early examples dating back to the 1970s [ABC⁺76] and prominent, contemporary examples including systems such as *Maria DB*¹, *PostgreSQL*², *Oracle* or *MS SQL*.

The relational model is structured around *relations*, which are a mathematical construct but can be visualized as two-dimensional tables. Such a table consists of columns – which are called *attributes* – and rows – which are called *tuples*

¹ Open Source, see <https://mariadb.org/>

² Open Source, see <https://www.postgresql.org/>

and hold *attribute values*. Semantically, a relation can be seen as a knowledge-base about some fact – such as, the paintings held by a museum – under a Closed-world assumption (CWA). That is, the relation contains all the information available about the fact and can thus be used to derive conclusive answers or results given a stated question or query.

In order to formalize the structure of and the operations that can be executed on the data represented by a relation, one can use Definition 3.1.

Definition 3.1 Relation according to [Cod70]

Let \mathcal{D}_j be sets we call *data domains* with $j \in [1, N]$ and $N \in \mathbb{N}_{>0}$. A *relation* \mathcal{R} constructed over these data domains is a set of M *tuples* $t_i = (a_{i,1}, a_{i,2}, \dots, a_{i,N})$ with $i \in [1, M]$ and $M \in \mathbb{N}_{>0}$ such that the first attribute value of any tuple comes from \mathcal{D}_1 , the second from \mathcal{D}_2 and so forth. That is, values $a_{i,1} \in \mathcal{D}_1, a_{i,2} \in \mathcal{D}_2, \dots, a_{i,N} \in \mathcal{D}_N$. Ergo, a relation can be seen as a subset of the cartesian product of all data domains over which it was constructed, i.e., $\mathcal{R} \subset \mathcal{D}_1 \times \mathcal{D}_2 \times \dots \times \mathcal{D}_N$.

The number of data domains N is referred to as its *degree* and we call such a relation *N-ary*. The number of tuples M is called the *cardinality* of \mathcal{R} with $M = |\mathcal{R}|$. It must be noted, that the relational model does not dictate what the data domains \mathcal{D} are. However, in a relational DBMS, they usually correspond to the data types supported by the database and the programming environment it was written in, for example, integer and floating point numbers or text. In this Thesis, we therefore sometimes use *data domain* synonymous for *data type*³.

In addition to Definition 3.1, we will use the identities 3.2 to 3.5 throughout this Thesis when working with relations.

Definition 3.2 Attributes of a Relation

Let \mathcal{R} be a N -ary relation over the data domains $\mathcal{D}_i, i \in [1, N]$ with corresponding names or indexes l_i . We call the combination of a data domain with a human readable label *attribute*, that is, $\mathcal{A}_i = (\mathcal{D}_i, l_i)$. To simplify notation, we sometimes use the label of an attribute in the subscript instead of the index.

³ Strictly speaking, the data domains of a given relation \mathcal{R} are merely subsets of the sets that represent the respective data type which, in turn, are subsets of even more basic sets such as \mathbb{N} or \mathbb{R} for Int and Float respectively. The relationship between types and data domains is subject of a more categorical approach to data models and nicely laid out in [Spi09]

Definition 3.3 Schema of a Relation

Let \mathcal{R} be a N -ary relation over the attributes $\mathcal{A}_i, i \in [1, N]$. We call the tuple of all attributes over which \mathcal{R} was constructed the *heading* or *schema* $\text{SCH}(\mathcal{R})$ of \mathcal{R} .

$$\text{SCH}(\mathcal{R}) = (\mathcal{A}: \text{if } \mathcal{A} \text{ is an attribute of } \mathcal{R})$$

Definition 3.4 Accessing Attribute Values

Let \mathcal{R} be a N -ary relation over attributes $\mathcal{A}_i, i \in [1, N]$ and let further $t \in \mathcal{R}$. To address the attribute value $a_i \in t$ that belongs to attribute $\mathcal{A}_i \in \text{SCH}(\mathcal{R})$, we use the *attribute value accessor* $t[\mathcal{A}_i]$.

$$a_i = t[\mathcal{A}_i] = \{a_j \in t: i = j\}$$

Definition 3.5 Supported Data Domains

For a given DBMS, we call \mathbb{D} the set system of data domains or data types supported by the system.

$$\mathbb{D} = \{\mathcal{D}: \text{if } \mathcal{D} \text{ is supported by DBMS}\}$$

In its original form, the relational model assumes the following properties to be true for a relation \mathcal{R} and its attributes [Cod70]:

Ordering of Tuples Tuples $t \in \mathcal{R}$ are inherently unordered and two relations are considered equal if they contain the same tuples, regardless of order.

Ordering of Attributes Attribute values a_i always occur in the same order within the tuple $t \in \mathcal{R}$, which corresponds to the order of the attributes $\mathcal{A}_i \in \text{SCH}(\mathcal{R})$. This order can evolve over time but remains constant in a momentary snapshot of \mathcal{R} . It follows from the definition that $|t| = |\text{SCH}(\mathcal{R})| \forall t \in \mathcal{R}$

Duplicates Since relations are sets, they do not allow for duplicates, i.e., every tuple $t \in \mathcal{R}$ must be unique in terms of their attribute values.

Given the idea of a relation, the sum of all data managed by a DBMS can logically be regarded as a collection of different relations $\mathcal{R}_k, k \in \mathbb{N}_{\geq 0}$ of assorted degrees N_k over data domains $\mathcal{D} \in \mathbb{D}_{\text{dbms}}$ (i.e., a collection of tables). The schema of a database can then be seen as the set system of all $\text{SCH}(\mathcal{R}_k)$. As [Cod70] points out, relations are subject to change over time. These changes can take place on

the level of any relation's structure, i.e., $SCH(\mathcal{R}_k)$, the relations \mathcal{R}_k itself, e.g., by tuples being added to (insert) or removed from (delete) a relation or the level of a tuple $t \in \mathcal{R}_k$, e.g., by altering one or multiple attribute values.

Example 3.1 features an example relation paintings visualized as a table. Each entry in the table represents a painting and the related attribute values.

Example 3.1 Table Representation of a Relation $\mathcal{R}_{\text{paintings}}$

The following table lists the content of a ternary relation ($N = 3$) $\mathcal{R}_{\text{paintings}}$. The attributes $\mathcal{A}_{\text{title}}$, $\mathcal{A}_{\text{artist}}$, $\mathcal{A}_{\text{painted}}$ correspond to the table's columns. The individual tuples t_i are "valid" combinations of painting title, artist and year of conception under CWA and constitute the rows.

| $\mathcal{R}_{\text{paintings}}$ | $\mathcal{A}_{\text{title}}$ | $\mathcal{A}_{\text{artist}}$ | $\mathcal{A}_{\text{painted}}$ |
|----------------------------------|------------------------------|-------------------------------|--------------------------------|
| t_1 | Mona Lisa | Leonardo da Vinci | 1506 |
| t_2 | The Starry Night | Vincent van Gogh | 1889 |
| t_3 | Las Meninas | Diego Velázquez | 1665 |

3.2.1 Keys and Normal Forms

The notion of a relation provides us with the basic tools for data modeling. In addition to Definition 3.1, Codd proposed a range of constraints to guarantee proper data definition using the relational model and the following constructs [Cod70]:

A Primary Key (PK) \mathcal{P} of relation \mathcal{R} is a subset of attributes $\mathcal{P} \subset SCH(\mathcal{R})$ that uniquely identify a tuple $t \in \mathcal{R}$. That is, the attributes that are not part of the PK and the associated values are functionally determined by \mathcal{P} . Using Example 3.2, the painting and all its attributes, i.e., the artist and the year of its creation, are functionally determined by the name of the painting, which is assumed to uniquely identify the entry.

A Foreign Key (FK) \mathcal{F} of relation \mathcal{R} is a subset of attributes $\mathcal{F} \subset SCH(\mathcal{R})$ that are not member of a PK, i.e., $\mathcal{F} \cap \mathcal{P} = \emptyset$, but reference the PK of \mathcal{R} or some other relation \mathcal{R}^* . FK can be used to model relationships between relations. Using Example 3.2, the artist that created a painting is referenced through a FK $\overline{\mathcal{A}}_{\text{artist}}$ in $\mathcal{R}_{\text{painting}}$ that references the PK $\mathcal{A}^*_{\text{artist}}$ in $\mathcal{R}_{\text{artist}}$.

An example of relations with primary and foreign key attributes is given in Example 3.2. PKs and FKs are indicated with star and overline respectively.

Note, that we assume here that people (artists) and paintings are uniquely identified by their first and lastname and their title respectively, which is obviously a simplification. In real-world applications, often artificial PKs are being generated to avoid unintended collisions.

Example 3.2 Relations with Primary and Foreign Keys

The following tables lists the schema and extent of $\mathcal{R}_{\text{paintings}}$ and $\mathcal{R}_{\text{artists}}$.

| $\mathcal{R}_{\text{painting}}$ | $\mathcal{A}^*_{\text{title}}$ | $\overline{\mathcal{A}}_{\text{artist}}$ | $\mathcal{A}_{\text{painted}}$ |
|---------------------------------|--------------------------------|--|--------------------------------|
| t_1 | Mona Lisa | Leonardo da Vinci | 1506 |
| t_2 | The Starry Night | Vincent van Gogh | 1889 |
| t_3 | Las Meninas | Diego Velázquez | 1665 |

| $\mathcal{R}_{\text{artist}}$ | $\mathcal{A}^*_{\text{artist}}$ | $\mathcal{A}_{\text{birth}}$ | $\mathcal{A}_{\text{death}}$ |
|-------------------------------|---------------------------------|------------------------------|------------------------------|
| t_1 | Leonardo da Vinci | 1452 | 1519 |
| t_2 | Vincent van Gogh | 1853 | 1890 |
| t_3 | Diego Velázquez | 1599 | 1660 |

Given the notion of primary and foreign keys, Codd proposed a series of *normal forms* that are an indication of the quality of a data model based on the functional dependency of the attributes on a PK. The basic idea is to avoid redundancy by normalization, i.e., by putting data that belongs together into dedicated relations and modeling relationships between them using foreign keys. The first three normal forms are as follows:

First Normal Form (1NF) requires, that no attribute in a relation has other relations as values, i.e., attributes are atomic and instead, relationships can be established by means of FKs. Given $\mathcal{R}_{\text{painting}}$ in Example 3.2, this means that $\mathcal{R}_{\text{artist}}$ cannot be stored as an attribute of $\mathcal{R}_{\text{painting}}$ (which would be possible according to Definition 3.1, since data domains can hold any type of element) and instead requires a dedicated relation.

Second Normal Form (2NF) requires, that every non-prime attribute is functionally determined by the whole primary key and not any subset thereof. Given $\mathcal{R}_{\text{artist}}$ in Example 3.2 and assuming that $\mathcal{A}^*_{\text{artist}}$ was split into two attributes $\mathcal{A}^*_{\text{firstname}}$ and $\mathcal{A}^*_{\text{lastname}}$ (composite key), this means that all of the non-prime attributes must be determined by $\mathcal{A}^*_{\text{firstname}}$ and $\mathcal{A}^*_{\text{lastname}}$ jointly, i.e., the full name, and not just either $\mathcal{A}^*_{\text{firstname}}$ or $\mathcal{A}^*_{\text{lastname}}$.

Third Normal Form (3NF) requires, that every non-prime attribute is functionally determined solely by the primary key and that they do not depend on any other attribute. Given $\mathcal{R}_{\text{artist}}$ in Example 3.2, this means that all of the non-prime attributes must be determined by $\mathcal{A}_{\text{artist}}^*$ alone.

All the normal forms build onto one another, i.e., for a data model to be considered 3NF is also required to satisfy 2NF and 1NF⁴. Additional normal forms up to 6NF and the Boyce–Codd Normal Form (BCNF), a slightly stronger version of 3NF, have been defined. For the sake of brevity, we will omit those since they are not relevant for the discussion ahead.

3.2.2 Relational Algebra

Having introduced the aspect of data representation and constraints, we can now move to that of operations that can be performed on the data upon querying. For that purpose, [Cod70] proposed the idea of a *relational algebra*, which follows a simple yet powerful idea: All query operations performed on relations are expressed by a set of *relational operators* that take one or multiple relations as input and output a new relation as expressed by Equation (3.1).

$$\text{OP} : \mathcal{R}_1, \dots, \mathcal{R}_n \rightarrow \mathcal{R}_O \quad (3.1)$$

Those relational operators can then be composed to express a query of arbitrary complexity as indicated by Equation (3.2).

$$\text{QUERY} = \text{OP}_1 \circ \text{OP}_2, \dots, \circ \text{OP}_m \quad (3.2)$$

In addition to this idea, Codd proposed a minimal set of relational operators listed in Table 3.1 and explained in the following sections. For the sake of completeness, it must be mentioned that notation and operators may slightly differ depending on the literature. We mainly use [GUW09] as our reference.

3.2.2.1 Simple Set Operations

Since relations are in essence sets of tuples, all basic operations known from set theory can be applied, namely *union*, *intersection* and *difference*, with the only constraint that the two input relations $\mathcal{R}_L, \mathcal{R}_R$ must be *union compatible* and thus exhibit the same attributes, i.e., $\text{SCH}(\mathcal{R}_L) = \text{SCH}(\mathcal{R}_R)$.

⁴ This has led to the mnemonic “*The key, the whole key, and nothing but the key, So help me Codd.*” in reference to a similarly structured oath often used in courts of law.

Table 3.1 The relational operators proposed by Codd et al [Cod70; GUW09].

| Name | Symbol | Arity | Example |
|-------------------|--|-------|---|
| Union | \cup | 2 | Set union of two input relations. |
| Intersection | \cap | 2 | Set intersection of two input relations. |
| Difference | \setminus | 2 | Set difference of two input relations. |
| Cartesian Product | \times | 2 | Pairs each tuple from one the left with every tuple of the right input relation and concatenates them. |
| Rename | $\rho_{\mathcal{A}_A \rightarrow \mathcal{A}_B}$ | 1 | Renames attribute \mathcal{A}_A in input relation to \mathcal{A}_B . |
| Selection | $\sigma_{\mathcal{P}}$ | 1 | Removes tuples from the input relation that don't match predicate \mathcal{P} . |
| Projection | π_{χ} | 1 | Removes attributes from the input relation that are not included in χ . |
| Natural join | $\bowtie_{\mathcal{P}}$ | 2 | Pairs each tuple from the left with every tuple of the right input relation if their shared attributes match and concatenates them. |

The set union $\mathcal{R}_L \cup \mathcal{R}_R$ generates a new relation of all tuples contained in either \mathcal{R}_L OR \mathcal{R}_R , as expressed by Equation (3.3)

$$\mathcal{R}_L \cup \mathcal{R}_R = \{t: t \in \mathcal{R}_L \vee t \in \mathcal{R}_R\} \quad (3.3)$$

The intersection $\mathcal{R}_L \cap \mathcal{R}_R$ generates a new relation of all tuples contained in \mathcal{R}_L AND \mathcal{R}_R , as expressed by Equation (3.4).

$$\mathcal{R}_L \cap \mathcal{R}_R = \{t: t \in \mathcal{R}_L \wedge t \in \mathcal{R}_R\} \quad (3.4)$$

The difference $\mathcal{R}_L \setminus \mathcal{R}_R$ generates a new relation of all tuples contained in \mathcal{R}_L AND NOT in \mathcal{R}_R , as expressed by Equation (3.5).

$$\mathcal{R}_L \setminus \mathcal{R}_R = \{t: t \in \mathcal{R}_L \wedge t \notin \mathcal{R}_R\} \quad (3.5)$$

These basic set operations simply combine two input relations without changing its structure, i.e., $SCH(R_I) = SCH(R_O)$. Due to relations being sets, duplicates resulting from a union operation are *implicitly* eliminated.

3.2.2.2 Cartesian Product

The binary, *cartesian product* or *cross product* $\mathcal{R}_L \times \mathcal{R}_R$ of two input relations $\mathcal{R}_L, \mathcal{R}_R$ concatenates every tuple $t_L \in \mathcal{R}_L$ with every tuple $t_R \in \mathcal{R}_R$ to form a new output tuple, as expressed by Equation (3.6).

$$\mathcal{R}_L \times \mathcal{R}_R = \{(t_L, t_R) : t_L \in \mathcal{R}_L \wedge t_R \in \mathcal{R}_R\} \quad (3.6)$$

The result is a relation that contains all the attributes of \mathcal{R}_L and \mathcal{R}_R , i.e., $SCH(\mathcal{R}_L \times \mathcal{R}_R) = SCH(\mathcal{R}_L) \cup SCH(\mathcal{R}_R)$ and every possible permutation of tuples from the input relations.

3.2.2.3 Rename

The unary *rename* operator $\rho_{\mathcal{A}_A \rightarrow \mathcal{A}_B}(\mathcal{R})$ renames the attribute \mathcal{A}_A to \mathcal{A}_B without changing any attribute values. This can be useful to eliminate collisions before applying the cartesian product or to enforce a natural join on differently named attributes.

3.2.2.4 Selection

The unary, generalized *selection* operator $\sigma_\phi(\mathcal{R})$ applied on an input relation \mathcal{R} creates an output relation that contains a subset of tuples $t \in \mathcal{R}$ such that only tuples that match the predicate ϕ are retained as expressed by Equation (3.7).

$$\sigma_\phi(\mathcal{R}) = \{t \in \mathcal{R} : \phi(t)\} \subset \mathcal{R} \quad (3.7)$$

The predicate ϕ can be any conditional statement consisting of individual atoms that involve attributes of \mathcal{R} or any constant value and comparison operators $=, \neq, >, <, \geq, \leq$. Individual atoms can also be combined by logical operators \wedge, \vee or \neg . Examples could be $\mathcal{A}_1 \geq 2$ (to express that an attribute should be greater to a constant) or $\mathcal{A}_2 = \mathcal{A}_3$ (to express that an attribut must be equal to another attribute) or $\mathcal{A}_1 \geq 2 \wedge \mathcal{A}_2 = \mathcal{A}_3$ to combine the two with $A_1, A_2, A_3 \in SCH(\mathcal{R})$.

3.2.2.5 Projection

The unary *projection* operator $\pi_\chi(\mathcal{R})$ with $\chi \subset SCH(\mathcal{R})$ applied on an input relation \mathcal{R} creates an output relation that only contains the attributes listed in χ , i.e., $SCH(\pi_\chi(\mathcal{R})) = \chi$, as expressed by Equation (3.8).

$$\pi_\chi(\mathcal{R}) = \{t[\chi] : t \in \mathcal{R}\} \text{ with } t[\chi] = \{t[\mathcal{A}] : \mathcal{A} \in \chi\} \quad (3.8)$$

All the tuples in \mathcal{R} are retained, however, resulting duplicates are removed.

3.2.2.6 Natural Join

The binary, *natural join* operator $\mathcal{R}_L \bowtie \mathcal{R}_R$ on two input relations $\mathcal{R}_L, \mathcal{R}_R$ concatenates every tuple $t_L \in \mathcal{R}_L$ with every tuple $t_R \in \mathcal{R}_R$ to form a new output

tuple, if the attribute values of t_L and t_R are the same for the shared attributes $\xi = \{\mathcal{A} : \mathcal{A} \in \text{SCH}(\mathcal{R}_L) \wedge \mathcal{A} \in \text{SCH}(\mathcal{R}_R)\}$ as expressed by Equation (3.9)

$$\mathcal{R}_L \bowtie \mathcal{R}_R = \{(t_L, t_R) : t_L \in \mathcal{R}_L \wedge t_R \in \mathcal{R}_R \wedge t_L[\xi] = t_R[\xi]\} \quad (3.9)$$

The result is a relation that contains all the attributes of \mathcal{R}_L and \mathcal{R}_R , i.e., $\text{SCH}(\mathcal{R}_L \times \mathcal{R}_R) = \text{SCH}(\mathcal{R}_L) \cup \text{SCH}(\mathcal{R}_R)$. If two relations coincide in all their attributes, the natural join becomes a cartesian product. Furthermore, the natural join can be expressed as a concatenation of the cartesian product with a selection. For example, given a relation \mathcal{R}_L with $\text{SCH}(\mathcal{R}_L) = (\mathcal{A}_A, \mathcal{A}_B, \mathcal{A}_C)$ and \mathcal{R}_R with $\text{SCH}(\mathcal{R}_R) = (\mathcal{A}_B, \mathcal{A}_D, \mathcal{A}_F)$, then $\mathcal{R}_L \bowtie \mathcal{R}_R \equiv \sigma_{\mathcal{A}_{L.B}=\mathcal{A}_{R.B}}(\mathcal{R}_L \times \mathcal{R}_R)$.

3.2.2.7 Expressing Queries

The following Example 3.3 illustrates how relational operators can be combined to form complex queries. Since expressing queries in such a way is quite inconvenient, in practice, queries are usually expressed through an intermediate query language which is then translated to the relational operators. A famous example of such a language in the domain of relational databases is SQL.

Example 3.3 Searching for Paintings Using Relational Algebra

The following tables lists the schema and extent of $\mathcal{R}_{\text{paintings}}$ and $\mathcal{R}_{\text{artists}}$.

| $\mathcal{R}_{\text{painting}}$ | $\mathcal{A}^*_{\text{title}}$ | $\overline{\mathcal{A}}_{\text{artist}}$ | $\mathcal{A}_{\text{painted}}$ |
|---------------------------------|--------------------------------|--|--------------------------------|
| t_1 | Mona Lisa | Leonardo da Vinci | 1506 |
| t_2 | The Starry Night | Vincent van Gogh | 1889 |
| t_3 | Las Meninas | Diego Velázquez | 1665 |

| $\mathcal{R}_{\text{artist}}$ | $\mathcal{A}^*_{\text{artist}}$ | $\mathcal{A}_{\text{birth}}$ | $\mathcal{A}_{\text{death}}$ |
|-------------------------------|---------------------------------|------------------------------|------------------------------|
| t_1 | Leonardo da Vinci | 1452 | 1519 |
| t_2 | Vincent van Gogh | 1853 | 1890 |
| t_3 | Diego Velázquez | 1599 | 1660 |

Using relational algebra, the query “return the names of all paintings that were painted by an artist who died after 1800” can be expressed by joining $\mathcal{R}_{\text{paintings}}$ and $\mathcal{R}_{\text{artists}}$, followed by a selection and projection:

$$\mathcal{R}_{\text{result}} = \pi_{\mathcal{A}_{\text{title}}}(\sigma_{\mathcal{A}_{\text{death}} \geq 1800}(\mathcal{R}_{\text{paintings}} \bowtie \mathcal{R}_{\text{artists}}))$$

Execution of this query leads to the following relation $\mathcal{R}_{\text{result}}$.

| | |
|-------------------------------|--------------------------------|
| $\mathcal{R}_{\text{result}}$ | $\mathcal{A}^*_{\text{title}}$ |
| t_2 | The Starry Night |

3.2.3 Extensions

While the relational model and the relational algebra forms the foundation of many modern DBMS, the model as originally proposed by Codd has often turned out to be too limited to accomodate certain functionality as required, e.g., by the SQL standard [Cha12; GUW09]. For example, many applications require storage of duplicate data and therefore, the notion of a relation – which is a set of tuples and thus does not allow for duplicates – is inadequate. Over the years, this has led to a growing list of proposals for extensions, some of which have seen adoption while other’s have remained theoretical in nature.

3.2.3.1 Relations vs. Bag vs. Sequences

The motivation for considering other mathematical structures than sets as an algebraic foundation for relations are twofold: Firstly, relations are unable to provide functionality that may be desirable in a DBMS, such as duplicate entries or ordering. Secondly, some of the mathematical convenience of the relational model, e.g., prohibiting duplicates, may not be too efficient to actually implement [GUW09].

It is therefore not surprising, that SQL formally operates on *bags* rather than sets, which allow for duplicate entries [GUW09; Cha12]

It is therefore not surpri

3.2.3.2 Extended Projection

3.2.3.3 Ranked Relational Algebra

3.2.3.4 Similarity Search and the Relational Model

Using the relationship between (diss-)similarity and distance, it has been shown by Giangreco et al., that for a database system to be able to support similarity search given the relational model for databases as described in Section 3.2, one can extend the set system of allowed data domains \mathbb{D} by $\mathcal{D} \subset \mathbb{R}^{dim}, dim \in \mathbb{N}_{>1}$ and postulate the existence of a relational similarity operator $\tau_{\delta(\cdot, \cdot), a, q}(R)$ that “performs a similarity query under a distance $\delta(\cdot, \cdot)$ applied on an attribute a of relation R and compared to a query vector q .” ([Gia18], p. 138). Such an operation intro-

duces an implicit attribute in the underlying relation \mathcal{R} , which in turn induces an ascending ordering of the tuples. Using this operation, one can go on to define two concrete implementations, namely $\tau_{\delta(\cdot,\cdot),a,q}^{kNN}(\mathcal{R})$, and $\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\mathcal{R})$, which limit the number of retrieved results by their cardinality k or a maximum cut-off distance ϵ respectively.

While postulating a new, relational operation $\tau_{\delta(\cdot,\cdot),a,q}^{kNN}(\mathcal{R})$ or $\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\mathcal{R})$ as proposed by [Gia18] has a certain elegance to it, it comes with limitations that become apparent upon dissection of its structure. In its postulated form, τ addresses several functions at once:

1. It specifies the distance function that should be evaluated.
2. It generates an implicit distance attribute on the underlying relation \mathcal{R} .
3. It imposes an ascending ordering and limits the cardinality of \mathcal{R} based on a predicate or a specified limit.

While being very specific and thus straightforward to implement, the amalgamation of all this functionality into a single operation is very specifically tailored to the use-case of similarity search and only of limited value when considering more general proximity-based query operations. If one would, for example, want to obtain the k farthest neighbours rather than the k nearest neighbours, as necessary when doing MIPS or obtaining negative examples, we would have to either change the distance function or extend the definition of τ .

Another important issue with the definition of τ in its current form is that despite the two operations $\tau_{\delta(\cdot,\cdot),a,q}^{kNN}(\mathcal{R})$ and $\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\mathcal{R})$ serving a very similar purpose, they behave very differently with respect to other operations. Generally, $\tau_{\delta(\cdot,\cdot),a,q}^{kNN}(\mathcal{R})$ does not commute with any selection σ due to the inherent limiting of the cardinality to a constant value, hence:

$$\sigma(\tau_{\delta(\cdot,\cdot),a,q}^{kNN}(\mathcal{R})) \neq \tau_{\delta(\cdot,\cdot),a,q}^{kNN}(\sigma(\mathcal{R})) \quad (3.10)$$

The left-hand side of Equation (3.10) filters the results of a kNN-search on \mathcal{R} , thus returning $n \leq k$ results, wherein $n = k$ only if σ matches all tuples. The right-hand side of Equation (3.10) performs a kNN-search on a pre-filtered relation \mathcal{R} , also returning $n \leq k$ entries. However, n will only be smaller than k if σ selects fewer than k tuples.

The same isn't true for $\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\mathcal{R})$, due to the limitation of the cardinality being facilitated by an *implicit* selection $\sigma_{\delta \leq \epsilon}$.

$$\sigma(\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\mathcal{R})) = \tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\sigma(\mathcal{R})) \quad (3.11)$$

Hence, σ and $\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\mathcal{R})$ commute and yield equivalent results as long as σ does not rely on the distance attribute introduced by $\tau_{\delta(\cdot,\cdot),a,q}^{\epsilon NN}(\sigma(\mathcal{R}))$

3.3 Queries: From Expression to Execution

3.3.1 Structured Query Language (SQL)

3.3.2 Query Planning

3.3.3 Query Execution

3.4 Storage, Indexes and Caching

3.5 Architectual Considerations

4

All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason.

— Immanuel Kant

On Multimedia Analysis and Retrieval

Multimedia data is ubiquitous and used in different forms in every area of our daily lives, be it private or professional. Statista.com estimates, that the amount of data created, captured, copied, and consumed in 2025 will exceed 181 iB from an estimated 2 iB in 2010¹. To a large extent, that data is multimedia data, which is being uploaded and shared on social media and the Internet.

A key development in this regard was the broad adoption of the smartphone in the early 2000s, which enabled almost every person on this planet to not only consume but also produce multimedia content in the form of texts, images, videos and audio snippets and to share those on platforms such as Twitter, Instagram, Facebook or TicToc. Fueled by this technological progress, the past few decades have seen a staggering development in both *volume* and *variety* of multimedia data found in the wild and in private data collections.

This extreme growth as well as the increasing *velocity* at which data is being produced, poses an enormous challenge for information processing, analysis and retrieval systems that deal with multimedia. Therefore, over the past few decades, many researchers have tackled the questions surrounding the efficient and effective analysis and retrieval of multimedia data.

Since the contributions of this Thesis rely on a large research corpus of multimedia research, we use this chapter to provide an overview over the relevant fundamentals and related work. Two major sources are the books *Multimedia Retrieval* by H. Blanken et al. [BdB⁺07] and *Similarity Search – The Metric Space Approach* by P. Zezula et al. [ZAD⁺06], which provide excellent introductions.

¹ Source: Statista.com, “Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025”, February 2022

4.1 Multimedia Data and Multimedia Collections

The term *multimedia* refers to a combination of one or many different *media types*, such as but not limited to, aural or visual information. We distinguish between these types, based on the representation we use when working and interacting with them. Sound, for example, is formed by airpressure waves that are registered by our ears. In contrast, visual information, involves electromagnetic waves captured by our eyes. In both cases, the brain plays an important role in interpreting the underlying processes and forming the human perception of the phenomenon. Similarly, we rely on different techniques when converting these media types from their analogue to their digital representations.

Traditionally, we often think of media as information that somehow can be perceived directly by our human senses. However, we consider (multi-)media data in a wider sense, which may also include less apparent and sometimes even digitally native examples such as motion capture data or data streams stemming from sensors or medical devices.

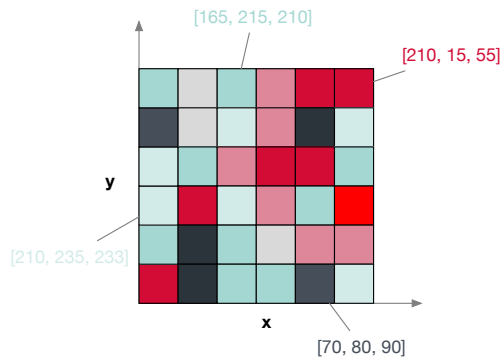
Even though the various media types may exhibit very different characteristics in their original representation, we find some commonalities that are crucial for formalising the problem of processing and analysing such data in information processing systems in general and analytics and retrieval systems in particular (see Examples 4.1 and 4.2).

Relationship with Time At a high level, we can roughly classify all the different media types based on their relationship with time. *Static* media types (e.g., an image) do not exhibit a temporal development, whereas the information in *dynamic* media types (e.g., an audio signal) depends on the time point that is being examined [BdB⁺07].

Unstructured Data The digital representation of any media type is typically highly unstructured, thus, there is no natural data model that can be leveraged for analysis. For example, there is generally no apparent connection between the raw image data and the motives it depicts. Therefore, information processing systems rely on *derivative representations* of the original data, gained through pre-processing and data analysis.

Semantic Gap The derivative representations of the original data usually describe specific aspects thereof and conversion to/from such representations is therefore often accompanied by a loss of information. This is a problem often referred to as the *semantic gap* [BdB⁺07; Ros18].

Example 4.1 Digital Representation of Visual Information



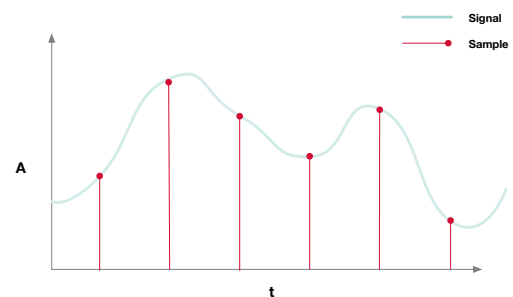
The visual information in a flat image is stored as a two dimensional array of *pixels*. The information in every pixel is typically formed by a sensor, in an array of sensors, that captures the electromagnetic signal. Each pixel holds colour values, usually one per colour channel. For example, with the RGB colour model, every pixel holds three values, one for the red, green and blue channel. The number of

pixels per dimension determines the *resolution* of the image. Typically, we use a fixed number of bits per colour – the *colour depth* – which determines the number colours that can be distinguished.

If we take, for example, a coloured image of 1000×1000 pixel, we must encode 3×10^6 individual colour values. Using 8 bit per colour, we end up storing 24×10^6 bit, which amounts to 3 MB worth of uncompressed image data. Images coming from modern cameras, exhibit resolutions much higher than that.

Example 4.2 Digital Representation of Aural Information

The airpressure waves that form sound can be recorded and translated to an electrical signal by microphones. When digitizing this information, the temporal development of this signal amplitude is *sampled* at a fixed rate. Each sample point consist of a value that quantizes the amplitude's energy at a given point in time



to a number on a given range. An audio stream is then a sequence of these values. The quality of the process is determined by the *sample rate*, i.e., the number of samples per time unit, and the *bit depth*, which determines quantization of the amplitude power.

If we take a 1 s audio snippet, sampled at 44 kHz, we end up storing 44 000 sample points. Using a bit depth of 16 bit per sample, this amounts to 704 000 bit or 88 kB of uncompressed audio data. In modern audio systems, we typically record multiple audio channels independently, leading to even more data.

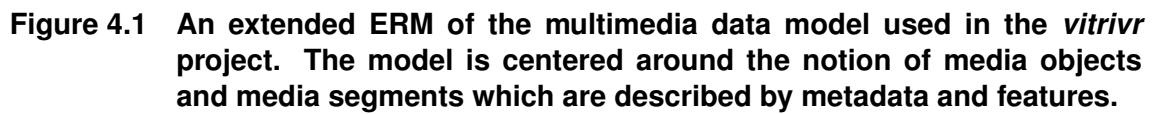
4.1.1 A Data Model for (Multi-)Media Data

In Chapter 3, we have introduced the notion of a data model and its importance for working with any type of data. Despite being agnostic to the concrete, conceptual data model in principle, the work presented in this Thesis still relies on a formal understanding of media data that enables organisation and management thereof. Due to its tight integration into the *vitriov* project [RGT⁺16; GRS19a; HSS⁺20], there is a particular model that we would like to introduce. Under this data model – which is illustrated in Figure 4.1 – a (multi-)media collection consists of multiple *media objects*. The media object describes the data at the level of individual documents – e.g., a video or image file – and comprises of basic attributes such its identifier, its media type or its path. The path forms the link to the raw media data in the filesystem, which we assume to be the most suited form of storage for this type of information.

To account for the temporal development found in dynamic media types, *vitriov*'s data model has a notion of a *media segment*, which represents a clearly defined, temporal slice of the object and stands in a one-to-many relationship with it. However, the model does not make any assumption as to how segments are formed. For static media types, such as images, there is a trivial one-to-one relationship between an object and a segment. For more complex media types, such as audio or video, *media segmentation* is a dedicated area of research [KC01] and can be approached in many different ways, e.g., at a fixed rate, based on changes in the content [Foo00; TYB16] or using self-adaptive, deep neural networks [SML19]. Obviously, different strategies for segmentation yield different degrees of summarization of information over time. Additionally, the proposed distinction also allows for very application specific media types and segmentation strategies, such as *image sequences* – introduced for LSC 2020 [HPG⁺20] – which group related images per day for the purpose of lifelog analysis.

To model the different derivative representations that are used to describe a media object and its segments, the model foresees *features*, which again stand in a one-to-many relationship to the segments. That is, every segment can have an arbitrary number of such features that describe different aspects of the segment's content. In practice, different types of features are obtained and stored in dedicated entities. Additionally, *descriptive metadata* allows for a description of the media both at the document and segment level by means of simple key-value pairs containing descriptions or labels.

While proven to be useful, the proposed model is only one possible approach to model media data and media collections and in certain aspects.



While the difference between the two models may seem marginal, we argue, that the latter offers several advantages over the former: Firstly, it eliminates a level of indirection introduced by the media segment and thus simplifies handling of static media types, which are simply described by different types of metadata that lack a start and end marker. Secondly, it does not assume segmentation to be statically defined and instead makes this a property of the metadata itself. This is reasonable, because the optimal segmentation strategy may depend on a particular feature, especially when multiple media types are involved, e.g., aural and visual information in a video. And finally, the model could be extended to also support spatial aspects as basis for segmentation, e.g., in images, which would enable description of spatio-temporal aspects of any media type. However, such ideas are beyond the scope of this Thesis.

Nevertheless, we will use the proposed model from Figure 4.2 throughout this Thesis and whenever we refer to terms like collection, media object, feature or descriptive metadata, we refer to the concepts introduced in this model.

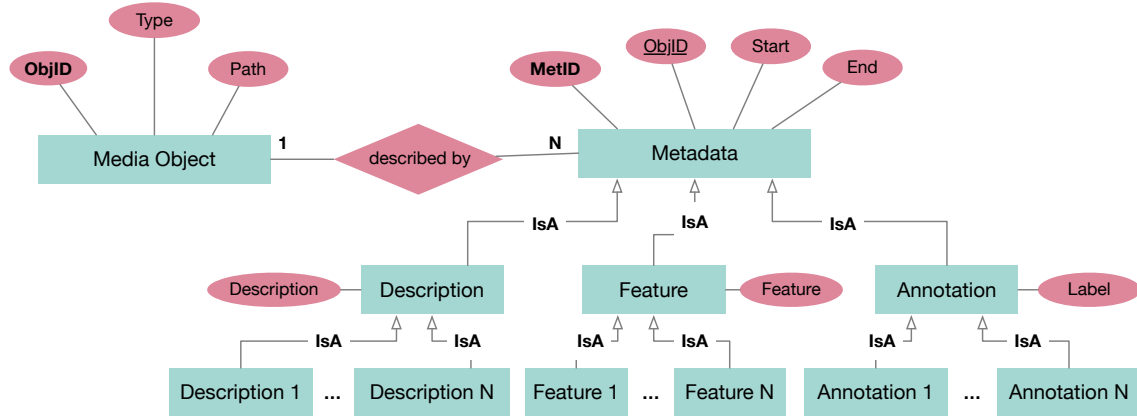


Figure 4.2 An extended ERM of the multimedia data model used for the purpose of this Thesis. It has been derived from *vitrivr*'s data model and foregoes the explicit segment entity.

4.1.2 Descriptive Metadata

Descriptive metadata in the sense of the proposed data model comprises of textual descriptions (e.g., title, summary), technical information (e.g., duration, location, frame rate) and annotations (e.g., category labels). Such information has always played an important role in multimedia retrieval and analysis, because it provides (indirect) access to the media's content and can be leveraged in a structured way, e.g., in database queries or for data organisation.

Over the years, many different metadata standards have emerged. For example, ID3² tags allow for organisation of large music libraries and classification of songs based on information about artists or albums. Similarly, the Dublin Core Metadata Element Set (DCMES or just Dublin Core)³ comprises of 15 basic properties that can be used to describe any type of media, e.g., videos or images. And last but not least, EXIF⁴ can be used to describe images and also includes technical metadata, such as the camera model, the exposure time or the f-number. These well established standards are complemented by a plethora of domain specific, standardised and non-standardised metadata frameworks.

However, while useful and important for the problem of analysis and retrieval, this type of metadata comes with important disadvantages. Firstly, and notwithstanding recent developments in machine learning, e.g., in image captioning [HSS⁺19], such information is traditionally assigned manually in a laborious and time-consuming annotation process. This process is barely able to keep up with the ever increasing velocity at which new content is created. Sec-

² See <https://id3.org/>

³ See <https://www.dublincore.org/>

⁴ See <https://www.loc.gov/preservation/digital/formats/fdd/fdd000146.shtml/>

ondly, descriptions and labels – especially if not standardised – are often subjective due to language, expertise and personal experience and may therefore differ depending on the person assigning them. This is closely related to the problem of the perceptive- and interpretative gap [Ros18] and highly relevant for any search process. And finally, it is often challenging to describe the content of media in a textual manner, especially if temporal development is involved.

Regardless of these challenges, however, experience shows that annotations – be they assigned manually or automatically – play an important role, especially when it comes to information retrieval. A contributing factor here is that of query formulation, which is very low-effort and familiar for textual input and becomes more challenging for other types of queries .

We use this often. But is there some study on this?

Source

4.1.3 Features

Simply put, a *feature* is a mathematical object that describes “derived characteristics” [BdB⁺07] of a media object or a part thereof. Therefore, they are sometimes also referred to as *descriptors*. A bit more formally, given a media collection $C = \{o_1, o_2, \dots, o_N\}$ of media objects, a *feature* $f \in \mathcal{F}$ describes a temporal interval $[\text{start}, \text{end}]$ with $\text{start}, \text{end} \in \mathbb{N}_0$ of a media object $o \in C$ under the feature transformation ϕ , as indicated by Equation (4.1).

$$\phi: C \times \mathbb{N}_0 \times \mathbb{N}_0 \longrightarrow \mathcal{F} \quad (4.1)$$

[BdB⁺07] distinguishes between low-level and high-level features, which is a qualitative assesment of how far removed the derived characteristic of a particular feature is from a concept that has a semantic meaning to a human user. For example, a histogram of colours would be considered a low-level feature, whereas a feature capturing salient keypoints or even objects in an image would be classified as high-level. In practice, high-level features are often derived from low-level features, and again these conversions are subject to the semantic gap.

The process of generating features from media objects is called *feature extraction* [BdB⁺07] and there is an entire corpus of research that deals with the engineering of features that describe the relevant aspects of given media type’s content dating back to the early 60s with first attempts in computer vision and image understanding. Various surveys, such as [MB03; DZJ⁺12; SJ19], cover the topic of feature extraction and we simply name and describe a few selected examples to provide an (unrepresentative) overview.

For images, one can roughly distinguish between low-level colour, texture and shape features [SJ19], as well as higher-level local keypoint-based features.

An example of a colour feature could be a simple colour histogram or a statistical moment that captures colour distribution, e.g., the average colour in a defined area of the image. Prominent examples for keypoint detection features include algorithms such as Scale Invariant Feature Transformation (SIFT) [Low99], Speeded-Up Robust Features (SURF) [BTVG06] or Histogram of Oriented Gradients (HOG) [DT05]. All these features are employed in many different applications ranging from retrieval and similarity search to the detection of higher-level concepts, such as, objects or faces [DBS⁺11; Far16].

Similarly, there is a wide range of features applied in audio analysis and retrieval. Mel-frequency Cepstral Coefficient (MFCC) – a representation of a short-term power spectrum on the perceptual *mel scale* – play an important role in speaker recognition, sound and music classification as well as audio segmentation [KS04] and were shown to outperform audio descriptors outlined in the MPEG-7 standard [QL01], which is a collection of standardised feature descriptors for images, audio and video. Other types of features include Pitch Class Profiles (PCP) for music retrieval and classification [Lee06; DBS19] or features based on rhythm, timbre or speed.

So far, we have mostly considered features that were engineered through the careful application of signal processing and/or statistical analysis of the raw media data. However, advances in machine learning and deep neural network architectures have enabled the feature extraction process to be automated to some extent, leading to a shift from *engineered* to *learned* features [HE10; GAR⁺16]. There is also research that deals with the comparison of the two paradigms in multimedia retrieval and analysis [BGS⁺17] as well as other domains, such as chemical structure analysis [GLSJ⁺21]. Preliminary results show, that while learned features often outperform the engineered ones, best results are attained by a combination of the two.

Given the cornucopia of features and feature extraction techniques to choose from, there are two important aspects that ought to be considered with respect to data modeling and data management: First, and despite the vast range of feature extraction techniques, the resulting features often exhibit a comparatively simple mathematical structure. We will show in Section 4.2 that in multimedia retrieval, one often deals with vectors in a high-dimensional, real-valued vector space [ZAD⁺06]. Second, when considering a complex enough use-case, no single feature can satisfy all the different types of applications because, as we have pointed out, a specific feature usually focuses on a specific aspect of the content. Consequently, a combination of features often yields the best results [DKN08].

Both these aspect have important implications for the the requirements on a data management and processing system for multimedia data, especially, if such a system should be able support different use-cases that are not necessarily known in advance. We would also like to point out, that both data models introduced in Section 4.1.1 are flexible enough to handle these two aspects.

4.2 Multimedia Retrieval

Multimedia retrieval deals with algorithms, methods and systems that allow for content-based search and obtaining items of interest from large (multi-)media collections based on user-defined queries. In addition to text-based search, such queries may also involve new modes of query formulation such as *Query-by-Example*, *Query-by-Sketch* [CWW⁺10], *Query-by-Humming* [GLC⁺95] or *Query-by-Sculpting* [BGS⁺20]. Many different research domains for the different types of media have emerged over the years, including but not limited to image retrieval [DA13], audio retrieval [Lu01], video retrieval [HXL⁺11], 3D model retrieval [YLZ07] and various, specialised subdomains [MK18].

Formally, the multimedia retrieval problem can be charactersied finding all media objects o in a media collection C that may be relevant to a given *information need* expressed by a user and transformed into a query q . This is achieved by obtaining some form of *(dis-)similarity* $s_{q,o}$ between q and o and ranking results based on it. This is therefore referred to as *similarity search* and as opposed to Boolean search, the result is non-binary in the sense that an object o may match a query to a certain degree, which is indicated by $s_{q,r}$.

Since usually, a direct comparison of a media object to a query is not feasible in practice – for reasons described in Section 4.1 – features are used as a proxy instead. This is a manifestation of the idea that derivative representations are considered in lieu of the actual object itself. Consequently, a multimedia retrieval system operates upon features $f \in \mathcal{F}$ derived from the original media objects by means of a feature transformation, i.e., $f = \phi(o), o \in C$. This transformation is applied twice: Once when indexing a media object in the collection – often a step that is thought as taking place offline – and once when deriving the same feature from the query input provided by the user. Subsequently, the system generates the (dis-)similarity measure for q and f , which is sometimes converted to a *similarity score* by a correspondence function⁵. The basic flow of information that results from this approach is illustrated in Figure 4.3.

⁵ In some cases, the similarity score can also be obtained directly from the features.

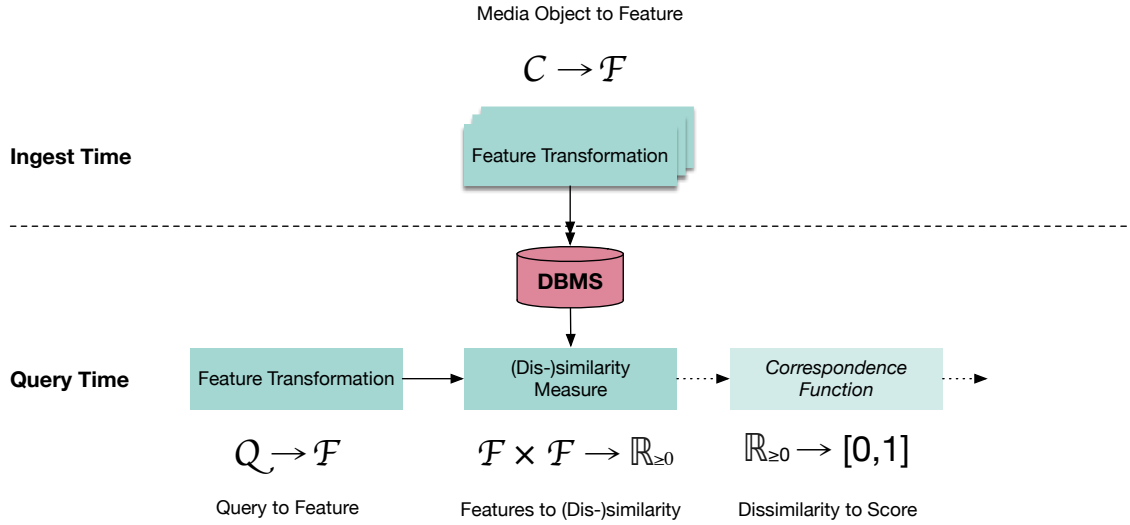


Figure 4.3 Flow of information in similarity search at ingest- and query time.

The main advantage of a similarity score is its normalisation to $[0, 1]$, which makes it more convenient for combining scores when multiple features are considered jointly using, e.g., *score-based late fusion* [DM10; Ros18].

4.2.1 Similarity Search and the Vector Space Model

The *vector space model* of similarity search assumes that the domain of the feature vectors \mathcal{F} is a subset of a mathematical vector space – that is, a set whose elements may be added and multiplied by a scalar. In practice, we often consider that vector space to be real-valued and high-dimensional, i.e., $\mathcal{F} \subset \mathbb{R}^d$ wherein d is the dimension and an inherent property of the feature and the underlying transformation ϕ . Consequently, every feature $f \in \mathbb{R}^d$ is simply a point in that high-dimensional vector space.

Under this premise, we consider the *distance function* $\delta(f, q)$ that calculates the distance between a feature and a query $f, q \in \mathcal{F}$, as defined by Equation (4.2), to be a proxy for their (dis-)similarity. Typically, the farther two vectors lie apart under the distance measure, the more dissimilar they are. Prominent examples of distance functions used in similarity search are listed in Table 4.1.

$$\delta: \mathcal{F} \times \mathcal{F} \rightarrow \mathbb{R} \quad (4.2)$$

At a high level, one can distinguish between *continuous* and *discrete* distance functions [ZAD⁺06]. Continuous functions typically exhibit a very large, potentially infinite co-domain whereas discrete functions map to a limited, often

pre-defined set of values. For example, the *Euclidean distance* is a continuous distance function, whereas the *Levenshtein distance* [Lev65] is discrete.

Furthermore, we can distinguish between metric distance functions that induce a topology on the underlying vector space – often denoted as (\mathcal{F}, δ) – as opposed to functions that do not. A metric distance function satisfies the *non-negativity*, *symmetry*, *identity of indiscernables* and *triangle inequality* constraints and (\mathcal{F}, δ) is called a *metric space*.

$$\delta(f, q) \geq 0 \quad (\text{non-negativity})$$

$$\delta(f, q) = \delta(q, f) \quad (\text{symmetry})$$

$$\delta(f, q) = 0 \iff f = q \quad (\text{identity of indiscernables})$$

$$\delta(f, q) \leq \delta(f, g) + \delta(g, q) \quad (\text{triangle inequality})$$

$$\forall f, g, q \in \mathcal{F} \wedge (\mathcal{F}, \delta)$$

The constraints imposed on a metric give rise to mathematical properties, that can be exploited for efficient execution of similarity search and specialised indexing techniques [ZAD⁺06]. However, despite their mathematical convenience, some applications involve distances that do not necessarily exhibit all the aforementioned properties of a metric, e.g., quasi-metrics (symmetry), semi-metrics (triangle inequality), meta-metrics (identity) [ZAD⁺06].

The choice of (dis-)similarity measure mainly depends on the application. All the Minkowski distances, (i.e., L^1 , L^2 , L^p) can be used for any type of real-valued vector and in practice, it often makes little difference which one is employed [Ros18]. Of all the Minkowski distances, the Euclidean distance (L^2) is the most common. The Chi-squared distance is well suited, if the feature vectors represent

| Name | Symbol | $\delta(f, q)$ | Domain | Co-Domain | Metric |
|-------------|----------|--|----------------|--------------|--------------|
| Manhattan | L^1 | $\sum_{i=1}^d f_i - q_i $ | \mathbb{R}^d | \mathbb{R} | metric |
| Euclidean | L^2 | $\sqrt{\sum_{i=1}^d (f_i - q_i)^2}$ | \mathbb{R}^d | \mathbb{R} | metric |
| Minkowski | L^p | $\sqrt[p]{\sum_{i=1}^d (f_i - q_i)^p}$ | \mathbb{R}^d | \mathbb{R} | metric |
| Chi-Square | χ^2 | $\sum_{i=1}^d \frac{(f_i - q_i)^2}{f_i + q_i}$ | \mathbb{R}^d | \mathbb{R} | metric |
| Cosine | - | $1 - \frac{\sum_{i=1}^d f_i q_i}{\sqrt{\sum_{i=1}^d f_i^2} \sqrt{\sum_{i=1}^d q_i^2}}$ | \mathbb{R}^d | $[-1, 1]$ | metric |
| Jaccard | - | $1 - \frac{A \cap B}{A \cup B}$ | A | $[0, 1]$ | metric |
| Levenshtein | - | - | strings | \mathbb{N} | metric |
| Hamming | - | - | strings | \mathbb{N} | quasi-metric |

Table 4.1 List of distance functions often used in similarity search.

histograms [PW10]. The Hamming- and Levenshtein distances belong to the class of *Edit distances* and can be used to compare strings.

Regardless of what (dis-)similarity measure is selected, the similarity search problem always looks the same: Given a query object $q \in \mathcal{F}$, the search algorithm should return all features $f \in \mathcal{F}$ that lie within a given proximity or distance to the query object q and that satisfy the selection condition. Depending on this selection condition, we distinguish between different types of similarity search problems [ZAD⁺06]. Those are illustrated in Figure 4.4 and Example 4.3

4.2.1.1 Nearest Neighbour Search (NNS)

NNS is probably one of the most prominent types of queries used in multimedia retrieval and certainly the one employed most often in the context of the *vitriov* [RGT⁺16; GRS19a] stack. Given the query object q and an arbitrary limit $k \in \mathbb{N}$ the task is to find the k features $f \in \mathcal{F}$ that are closest to q as illustrated in Figure 4.4(a). This is also referred to as k-Nearest Neighbour Search (kNN). Formally, the resultset R is given by Equation (4.3).

$$\{R \subset \mathcal{F} : \forall f_r \in R, f \in \mathcal{F} \setminus R, \delta(q, f_r) \leq \delta(q, f) \wedge |R| = k\} \quad (4.3)$$

Algorithmically, the distance between all $f \in \mathcal{F}$ and q must be evaluated exhaustively and the top k results must be retained by ranking and ordering the results, e.g., in a *bound priority queue*. This is sometimes referred to as *brute-force* approach to NNS. For $\mathcal{F} \subset \mathbb{R}^d$, its runtime complexity is $\mathcal{O}(dN)$ and thus depends linearly on the cardinality $N = |\mathcal{F}|$ and the dimension d .

A variant of kNN is k-Farthest Neighbour Search (kFN) wherein instead of the nearest, the farthest neighbours are obtained. This can be useful, for example, to obtain negative examples to train recommender systems [PSS⁺15]. However, the underlying algorithm remains the same.

4.2.1.2 Range Search

The range query or ϵ NN is useful for finding objects that lie within a given range, typically expressed by a value $\epsilon \in \mathbb{R}$. Given the query object q , the task is to find all features $f \in \mathcal{F}$ that lie within ϵ of q as illustrated in Figure 4.4(b). Formally, the resultset R is given by Equation (4.4).

$$\{R \subset \mathcal{F} : \forall f_r \in R, \delta(q, f_r) \leq \epsilon\} \quad (4.4)$$

Algorithmically, this search strategy is similar to NNS in that all features must be evaluated exhaustively and in that its complexity for $\mathcal{F} \subset \mathbb{R}^d$ is $O(dN)$. However, in contrast to NNS, there is no need for intermediate sorting, since the selection is based on a simple, numerical comparison to the constant ϵ .

Variants of ϵ NN include instances, where f must be larger than ϵ or where f must fall into an interval $[\epsilon_{lower}, \epsilon_{upper}]$. However, these variants are equivalent to the base-case since only the selection predicate changes.

4.2.1.3 Reverse Nearest Neighbour Search (RNNS)

In RNNS we try to find all $f \in \mathcal{F}$ that given a limit $k \in \mathbb{N}$ consider the query $q \in \mathcal{F}$ to be part of their kNN resultset as illustrated in Figure 4.4(c). Formally, the resultset R is given by Equation (4.5)

$$\{R \subset \mathcal{F} : \forall f_r \in R, q \in \text{kNN}(f_r) \wedge f \in \mathcal{F} \setminus R : q \in \text{kNN}(f)\} \quad (4.5)$$

Algorithmically, this type of search is more complex than kNN or ϵ NN, because basically, a NNS problem must be solved for every element $f \in \mathcal{F}$. Therefore, the runtime complexity is $O(dN^2)$ and scales with the square of $N = |\mathcal{F}|$.

Example 4.3 Comparing different types of similarity search algorithms

For this example, we consider $\mathcal{F} \subset \mathbb{R}^2$ and $f \in \mathcal{F}$ to be points of interest (e.g., museums, hotels, parks) on a two-dimensional city map. Furthermore, we consider $q \in \mathcal{F}$ to be another position on the same map, e.g., given by our current position or a selected point of interest. The different types of similarity queries can now be thought of as follows:

Nearest Neighbour Search is the equivalent of finding the k points of interest that are closest to q (e.g., the three museums closest to my location).

Range Search is the equivalent of finding all points of interest that are within ϵ distance from q (e.g., all museums within 3 km from my location).

Reverse Nearest Neighbour Search is the equivalent of finding all points of interest that consider q to be among their closest points of interest (e.g., finding all hotels that have a specific museum in the set of their three closest sites that must be seen when visiting the city).

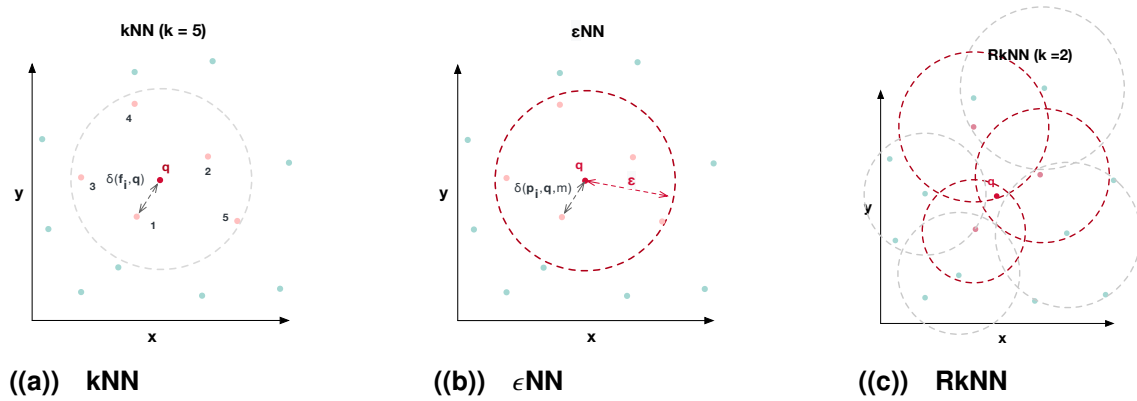


Figure 4.4 Graphical illustration of the different similarity search algorithms in $\mathcal{F} \subset \mathbb{R}^2$ using the Euclidean distance (L^2). Matches are highlighted in red.

4.2.2 NNS and High-Dimensional Index Structures

As we have argued in Section 4.2, even the simple NNS problem in \mathbb{R}^d exhibits a non-negligible runtime complexity of $O(dN)$ for a linear, brute-force scan – thus being proportional to the cardinality of $N = |\mathcal{F}|$ and the dimension d . Experience from database research shows that linear scans are too slow in terms of time required for query execution if N becomes large enough, especially when interactive query workloads are involved. This necessitates the use of index structures that allow for sub-linear data access.

For search in high-dimensional vector spaces, the dependency on the d has a huge impact on execution performance of any NNS algorithm, due to what is known as the *curse of dimensionality* [IM98; ZAD⁺06]. The impact of this “curse” is twofold: First, the execution time of a brute-force algorithm is directly proportional to the size of the feature vector. With a tendency towards more complex and thus higher-dimensional vectors, there is a direct, negative effect on runtime. Second, and more importantly, research has shown that the task of indexing high-dimensional vectors for more efficient access is not trivial and that traditional approaches, e.g., based on partitioning or classical tree structures, degenerate as dimensionality increases [IM98; WSB98] with the consequence, that they only yield marginal or no improvement over executing a linear scan. In fact, it was formally proven by [SR06] in their *Theory of Nearest Neighbor Indexability* that the performance of almost all known index structures at that time deteriorate to that of a linear scan if d becomes large enough.

Nevertheless, the issue of index structures that allow for more efficient and thus sub-linear NNS has been taken on by various researchers over the past few decades with results of high practical relevance. At a high-level, indexing algo-

| Name | Guarantees | Structure | Method |
|---------------------------------------|----------------------------|-----------------|--------------------|
| KDB-Tree [Rob81] | exact | tree | space partitioning |
| R^* -Tree [BKS ⁺ 90] | exact | tree | data partitioning |
| VAF [WSB98] | exact | list | quantization |
| VA ⁺ [FTA ⁺ 00] | exact | list | quantization |
| LSH [IM98; WZs ⁺ 18] | approximate, probabilistic | hash table i.a. | hashing |
| PQ [JDS10] | approximate | inverted file | quantization |
| NV-tree [LÁJ ⁺ 09] | approximate | tree | quantization |
| MRNG [LÁJ ⁺ 09] | approximate | graph | quantization |

Table 4.2 List of high-dimensional index structures employed in similarity search and retrieval.

gorithms can be classified along three dimensions: First, based on whether they provide any guarantees w.r.t. to the accuracy of the results, wherein “accuracy” is often measured in terms of precision and recall of a result produced by an index as compared to the the result produced by a linear scan [EZP21]. Second, based on their *structure* [SR06], e.g., *list*, *tree*, *graph*, *hash table* and *inverted file*. And finally, by the method that is employed, e.g., *quantization*, *hashing* or *partitioning*. An overview of different, prominent algorithms is provided in Table 4.2.

Early examples of high-dimensional indexes used to rely on space or data partitioning methods. Both have in common, that they assign features in the vector space to buckets or data files based on some partitioning of either the vector space or the data. Space partitioning methods include but the KD-Trees [Ben75], KDB-Trees [Rob81] or Quadrees [FB74]. The data partitioning methods involve R-Trees [Gut84], R^* -Trees [BKS⁺90] or M-Trees [CPZ97]. All of these methods were shown to perform well as long as dimensionality remains small (i.e., 4 to 12 dimensions) but they tend to deteriorate as d increases.

In the next few sections, we will highlight a few more recent examples of high-dimensional indexing methods to provide a non-representative overview.

4.2.2.1 Vector Approximation Files (VAF)

VA-files were first proposed by Weber et al. [WSB98] to address the problem of the dimensionality curse and were later refined by [FTA⁺00]. The basic idea of a VAF is the division of the data space into 2^b cells, as illustrated in Figure 4.5, where b denotes a fixed number of bits. Typically, one choses a specific number of bits per dimension b_d and uniformly partitions the vector space along each dimension, i.e., $b = b_d d$. A feature vector’s position in the grid is concatenated into a compact *signature* of size b and the VA-file is a list of those signatures.

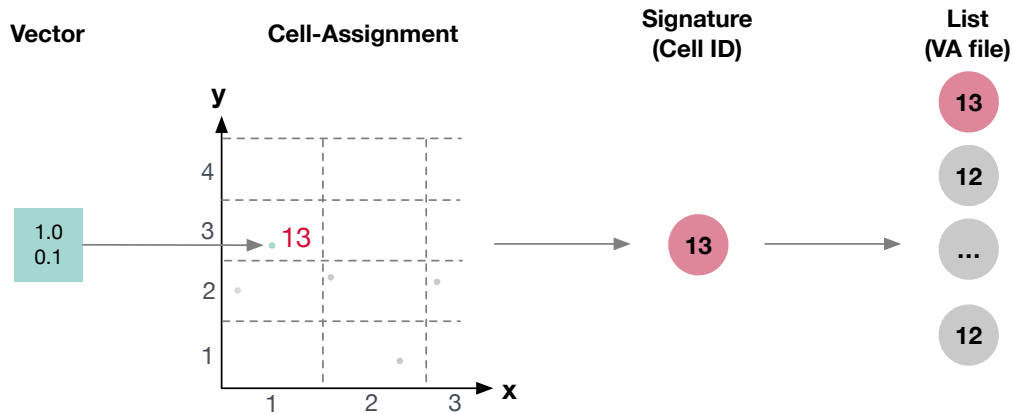


Figure 4.5 Simplified illustration of basic VAF indexing scheme in for $\mathcal{F} \subset \mathbb{R}^2$. A vector is assigned to a cell which gives rise to a signature that is stored in a linear list.

NNS is then realised as a linear scan of the file. Speed-up over a linear scan of the original vectors is achieved in two ways: Firstly, the signatures are more compact than the original representation, which reduces IO cost through compression. Secondly, the signatures can be used to obtain approximate bounds that are computationally less expensive than the actual distance calculation. These bounds can be used to filter out a majority of the points in the file, further reducing IO and CPU costs. Experimental results presented in [WSB98] demonstrate that 90% of all features can be skipped thus a distance must only be obtained for the remaining 10%. However, [EZP21] determine theoretically that for VAF degeneration is to be expected nevertheless for high dimensions, depending on how the data is distributed.

The VA+-files proposed by [FTA⁺00] introduce a few optimisations to the original algorithm. They suggest a prior decorrelation of the individual dimensions by applying a *Karhunen-Loève Transform* and a dynamic assignment of bit per dimension based on data distribution as opposed to a uniform distribution proposed in the original paper.

4.2.2.2 Locality Sensitive Hashing (LSH)

LSH refers to a family of algorithms [EZP21; WZs⁺18] that all share the same, basic idea of *locality preserving hashing* as first proposed by [IM98]. The idea is centered around a hash function $h : \mathcal{F} \rightarrow B$ that maps any two points $f, q \in \mathcal{F}$ to a bucket $b \in B$ such that Equations 4.6 and 4.7 hold, wherein (\mathcal{F}, δ) must

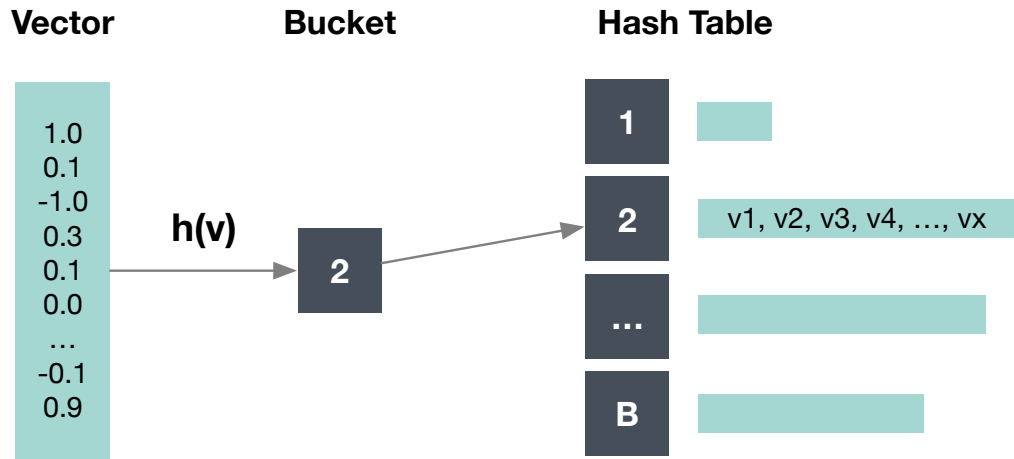


Figure 4.6 A simplified illustration of the basic LSH indexing scheme. A vector is hashed to a bucket by a locality preserving hash function h . The vector is then stored in the assigned bucket in a hash table.

constitute a metric space, $P_1, P_2 \in [0, 1], P_1 > P_2$ denote probabilities, $R > 0$ denotes an arbitrary threshold and $c > 1$ denotes an approximation factor.

$$\delta(p, q) \leq R \Rightarrow h(p) = h(q) \text{ with a probability of at least } P_1 \quad (4.6)$$

$$\delta(p, q) \geq cR \Rightarrow h(p) = h(q) \text{ with a probability of at most } P_2 \quad (4.7)$$

In simple terms, a locality preserving hash function has a high probability for collision (i.e., mapping to the same bucket) if two features $f, q \in \mathcal{F}$ lie close to one another, i.e., $\delta(q, f) \leq R$, but exhibits a low probability for collision if $\delta(q, f) \geq cR$. Given such a hash function, a feature $f \in \mathcal{F}$ can be hashed to a bucket and stored in a hash table with other vectors that lie close to it. This is illustrated in Figure 4.6.

Speedup for NNS can be achieved by limiting distance calculation to the vectors in the bucket a query q falls into. This is referred to as non-exhaustive search. However, due to the probabilistic nature of the hash function, this type of search strategy may exhibit errors due to vectors that were hashed into the “wrong” bucket. More refined strategies can be employed including hash-code ranking as well as single- and multi-table lookup [WZs⁺18] to further minimize that error. Nevertheless, LSH remains an approximate search strategy. A survey of different LSH algorithms can be found in [WZs⁺18].

4.2.2.3 Product Quantisation (PQ)

The idea for PQ was proposed by [JDS10] and describes a technique for quantizing vectors. The main idea involves decomposition of the original vector space

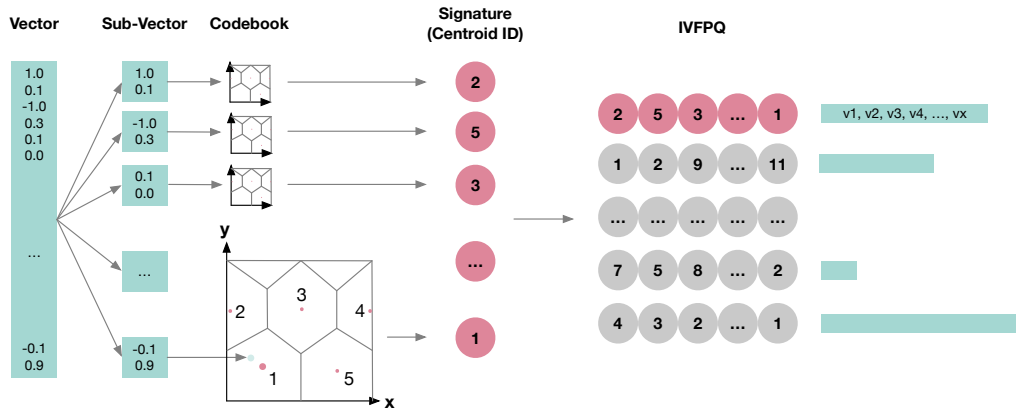


Figure 4.7 A simplified illustration of the basic PQ indexing scheme. A vector is divided into smaller subvectors and each subvector is assigned to the nearest centroid in its respective codebook. The centroid ID per sub-space gives rise to the signature, which can be stored in an inverted file index.

\mathbb{R}^d into smaller subspaces \mathbb{R}^s with $s < d$ and separate quantization of each subspace. The quantization of a subspace is achieved by learning a codebook for every subspace through k-means clustering on the data. Upon indexing, all subvectors of each feature vector $f \in \mathcal{F}$ are assigned to the nearest centroid in the codebook for the respective subspace. This gives rise to a compact signature that can be stored in an inverted file index, as illustrated in Figure 4.7.

Speedup of NNS is achieved in two ways: First, the resulting signature is a highly compressed representation of the original vector, which reduces IO by a large margin as is demonstrated in Example 4.4. Secondly, the signature can be used directly to approximate the distance between a query q and a feature f using a pre-calculated lookup-table, which reduces CPU costs considerably.

[JDS10] also elaborates on several optimisations that can be employed for very large data collection, e.g., by encoding the *residuals* instead of the actual feature vectors or by using multiple levels of coarser quantizers to introduce an additional partitioning of the data.

Example 4.4 Compression of PQ-Signature

Let us assume $\mathcal{F} \subset \mathbb{R}^{256}$, i.e., a 256-dimensional vector space ($d = 256$). Assuming a float data type, we require 4 byte per vector component, i.e., 1024 byte to store a single vector.

Using PQ, this space could now be divided into 16 subspaces with a dimensionality of 16 each, i.e., $s = 16 < d$. If now we assume, that create a code-

book consisting of 128 clusters for each subspace, then every subvector will be assigned to one of 128 centroids, which will result in 16 numbers (one per subspace) between 0 and 127. Each of these numbers can be stored in a single byte, that is, we only need 16 byte per vector to store an index entry. This is a factor of 64.

4.2.3 Beyond Similarity Search

Retrieval and analytics techniques that go beyond simple similarity search (e.g. SOM, summarization, clustering)

4.3 Online Multimedia Analysis

Introducing an online analysis pipeline (e.g., Pythia / Delphi).

4.4 Multimedia Analytics

Describe how the combination of analysis

4.5 Systems

PART III

Dynamic Multimedia Data Management

5

Modelling a Database for Dynamic Multimedia Data

As we have described in Chapter 2, multimedia retrieval and analytics workloads pose specific requirements on any data management system. These requirements can be summarised as follows:

Support for non-atomic data Multimedia workloads process features, which are usually real-valued, high-dimensional vectors but can theoretically be any mathematical object (e.g., complex numbers, matrices etc.). A DBMS with multimedia support must support these objects as first-class citizens.

Support for distance computation The notion of distance or proximity between features plays a crucial role in all types of multimedia workloads, be it for simple nearest neighbor search or clustering. While certain workloads rely on raw speed for simple computations and data structures that allow for this type of lookup, other workloads may require the ability to compose more complex computations and execute them efficiently without the use of an index. Both aspects must be covered by a DBMS with multimedia support.

Dynamically changing data Data today is subject to constant change, and a DBMS for multimedia data must be able to cope with and propagate these changes to all the relevant data structures that are required to satisfy the aforementioned requirements efficiently.

Accuracy vs. Time Multimedia queries often trade retrieval accuracy against speed or vice-versa, e.g., when using a high-dimensional index for lookup. The DBMS must allow the user to express their preference, which is often highly use-case dependent, at different levels of the system.

To address these requirements, we propose and formalize a set of functionality, which will be specified in the next sections. First, we define a notion of *generalized, proximity based queries* and their relationship to the high-dimensional index structures currently in existence. Second, we introduce a general-purpose mechanism to enable high-dimensional indexes to cope with changing data regardless of how they work internally. And finally, we describe a *cost-model* that takes the notion of accuracy of results into account. All these contributions can be seen as extensions to the functionality provided by a classical DBMS.

5.1 Generalized Proximity Based Operations

Starting with the metric space model for similarity search [ZAD⁺06] presented in Chapter 4, we propose to extend the notion of distance-based similarity search to that of a more general *proximity based query* following Definition 5.1.

Definition 5.1 Proximity Based Query

Let \mathcal{R} be a relation. Any database query that relies on the evaluation of a distance function $\delta : \mathcal{D}_q \times \mathcal{D}_q \rightarrow \mathbb{R}_{\geq 0}$ that calculates the distance between an attribute $\mathcal{A}_p \in \text{SCH}(\mathcal{R})$ and some query $q \in \mathcal{D}_q$, is called a *proximity based query*. We call q the *query* and \mathcal{A}_p the *probing argument*, which both share the same *query data domain* \mathcal{D}_q .

It is important to note, that Definition 5.1 simply requires a notion of proximity between some attribute and a query to be obtained by evaluating some distance function. The definition does not make any assumption as to what data domains \mathcal{D}_q query and probing attribute belong to nor how the distance is being used within the query, once it has been obtained.

It is obvious, that similarity search falls into the broader category of a proximity based query, wherein the distance is used to rank a relation and subsequently limit its cardinality. In addition to this common application, the definition of proximity based queries also includes operations such as evaluating a distance function followed by some filtering or aggregation based on the computed value. That is, we are not limited to mere NNS, since we consider obtaining the distance as one step, which can be freely combined with all other operations supported by the underlying algebra.

5.1.1 Revisiting Distance Computation

Since the choice of the distance function δ is a crucial part of any proximity based query, it is worth revisiting its definition. Again, starting with the metric space model [ZAD⁺06], we identify the following (implicit) constraints with respect to the distance function:

1. The domain (i.e., the input) of the distance function δ is assumed to be $\mathbb{R}^{dim} \times \mathbb{R}^{dim}$, that is, the distance function is assumed to be a binary function and arguments are restricted to real-valued vectors.
2. The codomain (i.e., the output) of the distance function δ is assumed to be $\mathbb{R}_{\geq 0}$, hence, the generated distance value is a positive, real number.
3. The pair (\mathcal{D}_q, δ) usually constitutes a metric, thus satisfying the non-negativity, identity of indiscernibles, symmetry and subadditivity properties.

Upon closer examination and if we consider the use cases presented in Chapter 2, one must come to challenge some of these constraints. If, for example, we turn to Example 5.1, we realise that it is not reasonable to impose a general restriction of the domain of the distance function to \mathbb{R}^{dim} with $dim \in \mathbb{N}^+$

Example 5.1 Maximum Inner Product Search for MRF

In MRF (see Section 2.3), we try to obtain the signal vector $a_{i \in \mathbb{N}} \in \mathcal{D}_q \subset \mathbb{C}^{dim}$ so that it maximizes the inner product to a signal (query) vector $q \in \mathbb{C}^{dim}$. In this case, the distance function δ has the form $\delta: \mathbb{C}^{dim} \times \mathbb{C}^{dim} \rightarrow \mathbb{R}_{\geq 0}$. Hence, the domain of δ is a complex vector space.

Obviously, this limitation of the definition of δ can be easily remediated simply by extending the set of supported data domains \mathbb{D} by \mathbb{C}^{dim} similarly to how we did for \mathbb{R}^{dim} as proposed by [Gia18]. Nevertheless, we have to acknowledge, that the text-book definition of a distance function is obviously too limited for some real-world applications, and that the query and probing arguments of a distance function could be any type of value supported by the DBMS. Another, similar example could be the *Levenshtein distance* [Lev65] – often used in computer linguistics – which is a distance metric on string values.

If now in addition, we consider Example 5.2, we see that restricting oneself to binary functions with $\mathbb{R}_{\geq 0}$ is also too limiting when facing certain practical scenarios.

Example 5.2 Distance Between a Vector and a Hyperplane

To find positive and negative examples $a_i \in \mathcal{D}_q \subset \mathbb{R}^{dim}$ given a linear classifier, e.g., provided by a SVM, we evaluate the distances between the attributes and a (query-)hyperplane described as $\mathbf{q}^T \mathbf{x} - b = 0$ with $\mathbf{q}, \mathbf{x} \in \mathbb{R}^{dim}$ and $b \in \mathbb{R}$. The distance function is then given by:

$$\delta(\mathbf{a}_i, \mathbf{q}, b) = \frac{\|\mathbf{q}^T \mathbf{a}_i + b\|}{\|\mathbf{q}\|} \quad (5.1)$$

δ now has the form $\delta: \mathbb{R}^{dim} \times \mathbb{R}^{dim} \times \mathbb{R} \rightarrow \mathbb{R}$. Hence, δ is no longer a binary but a ternary function with arguments \mathbf{a}_i , \mathbf{q} and b . Furthermore, the distance may take negative values depending on whether a result is considered a positive or negative example.

Again, we are confronted with an example that violates the text-book definition of a distance function. While the idealized idea that distance functions used in proximity based queries must always constitute a metric on some real-valued vector space may be very common [ZAD⁺06], we see in practice that this assumption is often violated [BS19] and that many functions used to calculate proximity between objects, are not actual metrics. Even though, this can be a disadvantage when considering high-dimensional index structures that exploit the geometric properties of metric spaces, it is obvious that such functions exist and must thus be considered in any generalized database application supporting proximity based queries.

In contrast, there is actually good reason to assume the codomain of a distance function δ to lie in \mathbb{R} . On the one hand, it is obviously convenient both for the underlying mathematics as well as from an implementation prespective. More importantly, however, real numbers – unlike, for example, complex numbers or vectors – come with a natural, total ordering, which is required for a lot of downstream operations such as sorting or clustering.

To summarize, we now find ourselves in the position where, on the one hand, we require a proper definition of what is considered to be a valid distance function so as to be able to efficiently plan and execute queries that use them, while on the other hand keeping that definition open enough to accomodate the many different, applications.

To address the aforementioned limitations while still accomodating classical, metric distances, we propose the extension of a the distance function to the more general notion of a Distance Function Class (DFC) following Definition 5.2.

More ex-
amples

Definition 5.2 Distance Function Class

A DFC $\hat{\delta}: \mathcal{D}_q \times \mathcal{D}_q \times \mathcal{D}_1 \dots \times \mathcal{D}_N \rightarrow \mathbb{R}$ is an N -ary but at least binary function $\hat{\delta}(p, q, s_1, \dots, s_n)$ that outputs a distance $d \in \mathbb{R}$ between a probing argument $p \in \mathcal{D}_q$ and a query argument $q \in \mathcal{D}_q$ using a defined number of *support arguments* $s_{k \in \mathbb{N}}$ from any of the data domains in \mathbb{D} .

A DFC (and any function for that matter) is uniquely defined by its *signature*, a $(N + 1)$ -tuple specifying the name of the DFC and the data domains \mathcal{D}_i of the arguments it can accept as defined in Equation (5.2). As a convention for notation, the probing argument p of a DFC $\hat{\delta}(p, q, s_1, \dots, s_n)$ will always come first, followed by the query argument q , again followed by its support arguments s_k in no particular order.

$$\text{SIG}(\hat{\delta}) = (\text{NAME}, \mathcal{D}_p, \mathcal{D}_q, \mathcal{D}_1, \dots, \mathcal{D}_{N-2}) \quad (5.2)$$

In the context a concrete execution plan for a proximity based query, we observe that most of the time, the query argument effectively remains constant, while the support arguments may or may not be subject to change. For example, for simple NNS, there is a single, constant query vector that is compared against all the vectors in the DBMS, i.e., the probing argument. Hence, irrespective of the arity of the DFC, its (partial-) *implementation* δ used during execution, comes with an arity of at most $N - 1$. This is denoted by Equation (5.3).

$$\delta = \text{IMP}(\hat{\delta}) \ni \text{SIG}(\delta) \subset \text{SIG}(\hat{\delta}) \quad (5.3)$$

With this idea of a DFC and its implementation in mind, we can start to reason about their properties in the broader context of database operations in general and proximity based queries in particular:

DFC and Proximity Based Queries In extension to Definition 5.1, we realise that any query execution plan that involves the evaluation of a (partial) implementation of a DFC, falls into the category of a proximity based query.

Purity of Domain Since the purpose of a DFC is to quantify proximity between a probing argument p and a query argument q , we can assume that $p, q \in \mathcal{D}_q$.

Scalarity of Codomain A DFC and its implementations always produce a single, scalar output $d \in \mathbb{R}$ to quantify the distance.

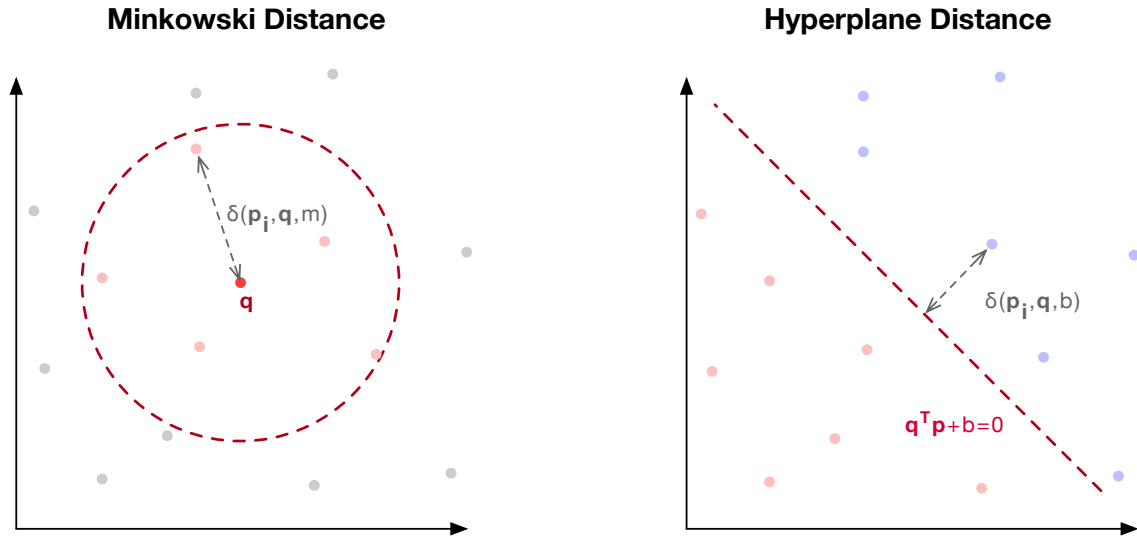


Figure 5.1 Two examples of distance functions that fall into the broader category of a DFC. On the left, the point-distance between q and p_i and on the right, the distance between a hyperplane $q^T x + b = 0$ and points p_i .

From the perspective of the DBMS, DFCs often represent a family of higher-order functions, which parametrize lower-order functions using the query q and support arguments s_k . Implementing a DFC in essence means fixing values for q and s_k in the context of a query, where this is possible given its general structure. This implementation requirement enshrines that for a proximity based query, altering q and (sometimes) even s_k constitutes a different query.

An illustrative and widely used example of a DFC would be the family of *Minkowski Distances* $\hat{\delta}_M: \mathbb{R}^{dim} \times \mathbb{R}^{dim} \times \mathbb{R} \rightarrow \mathbb{R}_{\geq 0}$ with its two, partial implementations, the Manhattan (L1) and the Euclidean (L2) distance:

$$\hat{\delta}_M(\mathbf{a}, \mathbf{q}, p) = \left(\sum_{i=1}^n |q_i - a_i|^p \right)^{\frac{1}{p}} \quad (5.4)$$

$$\hat{\delta}_{L1}(\mathbf{a}, \mathbf{q}) = \text{IMP}_{p=1}(\hat{\delta}_M) = \sum_{i=1}^n |q_i - a_i| \quad (5.5)$$

$$\hat{\delta}_{L2}(\mathbf{a}, \mathbf{q}) = \text{IMP}_{p=2}(\hat{\delta}_M) = \sqrt{\sum_{i=1}^n |q_i - a_i|^2} \quad (5.6)$$

However, using the idea of a DFC, we can express many different types of much more complex distance functions in a simple yet powerful framework. Two examples are illustrated in Figure 5.1 and it is easy to see that both the functions used in Example 5.1 (complex domain) and Example 5.2 (ternary function with scalar support argument) fall into the category of a DFC.

As an aside, we must address the role of dimensionality dim in the case of data domains that are subsets of vector spaces, i.e., $\mathcal{D}_q \subset \mathbb{R}^{dim}$ or $\mathcal{D}_q \subset \mathbb{C}^{dim}$. One could argue, that the dimensionality of such a vector space can also be seen as a parameter of the DFC. Irrespective of the merits such an argument might have, we consider the dimensionality of the vector space to be a strictly structural property of the underlying data domain \mathcal{D}_q and thus tightly coupled to the data type. This means, that the data types and by extension the dimensionality of all of the attributes involved are well-defined and remain constant for a query.

5.1.2 Extending the Relational Model

Using Definition 5.2, one can now start to integrate DFCs into the relational data model. In line with [Gia18], we first assume \mathbb{D} – the set system of data domains supported by the database – to be extended by whatever data domain \mathcal{D}_q is required, such as but not limited to \mathbb{R}^{dim} or \mathbb{C}^{dim} .

Extended Projection and DFCs We use the idea of an extended projection $\pi_{\mathcal{E}}$ – as proposed by [GHQ95; GUW09] and introduced in Section 3.2 – wherein \mathcal{E} denotes a list of projection expressions e that involve attributes $\mathcal{A}_i \in \mathcal{R}$. That is, in addition to the simple projection onto attributes \mathcal{A}_i , an extended projection may also include the evaluation of literals or functions as expressed in Definition 5.3.

Definition 5.3 Extended Projection $\pi_{\mathcal{E}}$

Let \mathcal{R} be a N-ary relation over attributes $\mathcal{A} \in \text{SCH}(\mathcal{R})$ with tuples $t \in \mathcal{R}$. We call $\pi_{\mathcal{E}}$ the *extended projection* with K projection elements $e_i \in \mathcal{E}$, i.e.,

$$\mathcal{E} = \{e_1, e_2, \dots, e_K\}, \text{ with } e_i \in \text{SCH}(\mathcal{R}) \vee e_i \in \mathcal{D}_i \vee e_i \in \mathbb{F}$$

where \mathbb{F} is a set of all functions $f_i: \mathcal{D}_1 \times \mathcal{D}_2 \dots \mathcal{D}_M \rightarrow \mathcal{D}_{out}$ known to the DBMS and $\mathcal{D}_i \in \mathbb{D}$. The extended projection is then defined as follows:

$$\pi_{\mathcal{E}}(\mathcal{R}) = \{t[\mathcal{E}] : t \in \mathcal{R}\} \text{ with } t[\mathcal{E}] = \bigcup_{e_i \in \mathcal{E}} \begin{cases} t[e_i] & \text{if } e_i \in \text{SCH}(\mathcal{R}) \\ e_i & \text{if } e_i \in \mathcal{D}_i \\ e_i(t[\mathcal{E}']) & \text{if } e_i \in \mathbb{F} \end{cases}$$

It is to be noted, that the domain of every function $f_i \in \mathbb{F}$ is again, an implicit, extended projection $t[\mathcal{E}']$, where \mathcal{E}' contains all projection elements required as function arguments. Hence, function invocations can be nested arbitrarily.

If now we assume any DFC supported by the DBMS to be member of \mathbb{F} , the evaluation of a DFC as part of a query is given by Definition 5.4.

Definition 5.4 Distance Function Class in Extended Projection $\pi_{\mathcal{E}}$

Let $\hat{\delta}: \mathcal{D}_q \times \mathcal{D}_q \times \mathcal{D}_1 \dots \times \mathcal{D}_{N-2} \rightarrow \mathbb{R}$ be a N-ary DFC. The *extended projection* $\pi_{\hat{\delta}(a_1, a_2, \dots, a_N)}(\mathcal{R})$ with $a_i \in \text{SCH}(\mathcal{R}) \vee a_i \in \mathcal{D}_i \vee a_i \in \mathbb{F}$ denotes the evaluation of $\hat{\delta}$ using any number of argument. Applying $\pi_{\hat{\delta}(\cdot)}$ on a M-ary relation \mathcal{R} introduces a new distance attribute \mathcal{A}_{δ} , i.e., $\text{SCH}(\pi_{\hat{\delta}(\cdot), \mathcal{A}_1, \dots, \mathcal{A}_M}(\mathcal{R})) = \text{SCH}(\mathcal{R}) \cup \{\mathcal{A}_{\delta}\}$

Obviously, the evaluation of multiple DFCs in a single, extended projection or the combination of simple attribute projections with the evaluation of DFCs are also allowed. In fact, all of the following expressions are valid examples of the extended projection on relation \mathcal{R} with $\text{SCH}(\mathcal{R}) = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4\}$ and $\delta_i = \text{IMP}(\hat{\delta}_i)$: (i) Evaluation of multiple DFCs with same or different probing arguments (e.g., $\pi_{\delta_1(\mathcal{A}_1), \delta_2(\mathcal{A}_1)}(\mathcal{R})$), (ii) evaluation of same DFC using same or different probing arguments (e.g., $\pi_{\delta_1(\mathcal{A}_1), \delta_1(\mathcal{A}_2)}(\mathcal{R})$), (iii) combination of DFCs with any attribute projection and/or algebraic expression evaluation (e.g., $\pi_{\delta(\mathcal{A}_1), \mathcal{A}_2, \mathcal{A}_4}(\mathcal{R})$).

With the proposed, extended projection, the obtained distance value becomes an explicit attribute in the relation $\pi_{\hat{\delta}(\cdot)}(\mathcal{R}) = \mathcal{R}'$, which can then be used by downstream, relational operators and/or be returned by the query to any calling system. This makes sense, for example in NNS, because the distance is usually converted to a similarity score by some correspondence function.

Ranked Relations To support sorting by the obtained distance (or any other attribute for that matter), which is an essential part of many types of proximity based queries, we use a variant of the *ranked relation* as proposed by [LCI⁺05] and described in Section 3.2.3. A ranked relation $\mathcal{R}^{\mathcal{O}}$ exhibits an ordering of tuples $t \in \mathcal{R}^{\mathcal{O}}$, which is induced by \mathcal{O} . As opposed to [LCI⁺05] and more in line with [GUW09], we assume $\mathcal{O} \subset \text{SCH}(\mathcal{R})$ to be a sequence of *attributes* by which the relation should be sorted,¹ as is specified in Definition 5.5.

¹ We argue that formally, the evaluation of a *scoring function*, as proposed by [LCI⁺05], can be expressed within the scope of the extended projection and does not need to be part of the order operation.

Definition 5.5 **Ranked relation \mathcal{R}^O**

Let \mathcal{R} be a N-ary relation over attributes $\mathcal{A} \in \text{SCH}(\mathcal{R})$ with M tuples $t_m \in \mathcal{R}$. We call \mathcal{R}^O a *ranked relation* with ranking attributes $O \subset \text{SCH}(\mathcal{R})$. O induces a non-strict, total ordering using the binary, lexicographic ranking relation \leq , such that:

$$t_i \leq t_k = \bigvee_{a \in O} t_i \leq_a t_k, \text{ with}$$

$$t_i \leq_a t_k \leftrightarrow t_i[a] \leq t_k[a]$$

The position of a tuple $t_m \in \mathcal{R}^O$ w.r.t. \leq is signified by the ranking function

$$\text{RANK} : \mathcal{R} \rightarrow [1, M]$$

which assigns a rank $m \in [1, M]$ depending on the tuple's position within \mathcal{R}^O , i.e., the first tuple t_1 gets rank 1, second tuple t_2 rank 2 etc.

It is to be noted, that the ranking O is an inherent property of a ranked relation \mathcal{R}^O as is, for example, its schema $\text{SCH}(\mathcal{R})$. That property is retained or induced as operators are being applied. Furthermore, if a relation does not exhibit a specific ordering, then O is empty and $\mathcal{R}^O = \mathcal{R}^\emptyset = \mathcal{R}$. Importantly, though, we expect an unordered relation to still exhibit some internal ordering of tuples, which we call the *natural ordering* of \mathcal{R} and which is not related to any observable property. Consequently, RANK is also defined for such relations, even though the returned values may appear arbitrary.

To explicitly order a relation, we introduce the *relational order operator* τ_O , which simply imposes order O on relation \mathcal{R} , overriding any previous ordering, as denoted by Equation (5.7).

$$\tau_O(\mathcal{R}) = \mathcal{R}^O \tag{5.7}$$

For example, using the relation \mathcal{R} with $\text{SCH}(\mathcal{R}) = \{\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4\}$, the application of $\tau_{\mathcal{A}_1, \mathcal{A}_2}$ on relation \mathcal{R} sorts the tuples $t_i \in \mathcal{R}$ based on \mathcal{A}_1 and breaks ties sorting by \mathcal{A}_2 . Tuples that are equal in both attributes appear in natural, i.e., arbitrary order. To support ascending (\uparrow) and descending (\downarrow) sort direction, we can use the duality property of the individual ranking relations in an actual implementation.

k-selection To be able to execute a specific type of proximity based query – namely, kNN or kFN – we require one last extension to the relational algebra in the form of the *k-selection* operator $\lambda(\mathcal{R})$, which simply limits the cardinality of a relation \mathcal{R} to the first k tuples. It is trivial to see that for a ranked relation \mathcal{R}^O , $\lambda_k(\mathcal{R}^O)$ limits the tuples to the top k results w.r.t. to the ordering induced by O .

$$\lambda_k(\mathcal{R}^O) = \{t \in \mathcal{R}^O : \text{RANK}(t) \leq k\} \quad (5.8)$$

We can now use Example 5.3 to demonstrate the flexibility of the proposed algebra and the different types of queries that can be expressed through it.

Example 5.3 Relation listing paintings with feature vectors

\mathcal{R}_p lists paintings with their title \mathcal{A}_t , the year of their creation \mathcal{A}_y and some arbitrary feature vector \mathcal{A}_f . Note that $\mathcal{D}_f \subset \mathbb{R}^3$.

| \mathcal{R}_p | \mathcal{A}_t | \mathcal{A}_y | \mathcal{A}_f |
|-----------------|------------------|-----------------|------------------|
| t_1 | Mona Lisa | 1506 | [0.0, 0.2, -1.3] |
| t_2 | The Starry Night | 1889 | [1.0, 0.9, 2.6] |
| ... | ... | ... | ... |
| t_N | Las Meninas | 1665 | [-0.5, 3.0, 0.8] |

Using DFCs and the proposed, relational algebra extensions, we can now express:

| Name | Result | Algebraic Form |
|-------------------------|--|--|
| NNS / kNN | $\mathcal{R}^{\mathcal{A}_d \uparrow}$ | $\lambda_k(\tau_{\mathcal{A}_d \uparrow}(\pi_{\mathcal{A}_y, \delta(\mathcal{A}_f)}(\mathcal{R}_p)))$ |
| FNS / kFN | $\mathcal{R}^{\mathcal{A}_d \downarrow}$ | $\lambda_k(\tau_{\mathcal{A}_d \downarrow}(\pi_{\mathcal{A}_y, \delta(\mathcal{A}_f)}(\mathcal{R}_p)))$ |
| ϵ NN [Gia18] | $\mathcal{R}^{\mathcal{A}_d \uparrow}$ | $\tau_{\mathcal{A}_d \uparrow}(\sigma_{\mathcal{A}_d \leq \epsilon}(\pi_{\mathcal{A}_y, \delta(\mathcal{A}_f)}(\mathcal{R}_p)))$ |
| NNS w. selection | $\mathcal{R}^{\mathcal{A}_d \uparrow}$ | $\tau_{\mathcal{A}_d \uparrow}(\pi_{\mathcal{A}_{year}, \delta(\mathcal{A}_f)}(\sigma_{\mathcal{A}_y=1889}(\mathcal{R}_p)))$ |
| Multi order | $\mathcal{R}^{\mathcal{A}_{d1} \uparrow, \mathcal{A}_{d2} \uparrow}$ | $\tau_{(\mathcal{A}_{d1} \uparrow, \mathcal{A}_{d2} \uparrow)}(\pi_{\delta_1(\mathcal{A}_f), \delta_2(\mathcal{A}_f)}(\mathcal{R}_p))$ |
| Aggregation by distance | \mathcal{R} | |

5.1.2.1 Algebraic Properties of $\pi_{\mathcal{E}}, \tau_O, \lambda_k$

Since a query planner of a DBMS must be able to manipulate the operators that were introduced before, there is a series of algebraic properties that must be specified. For unary operators, we must consider *commutativity*, i.e., whether changing the order of application leads to the same result or not. Two unary operators OP_1, OP_2 commute if, and only if, $\text{OP}_1(\text{OP}_2(\mathcal{R})) = \text{OP}_2(\text{OP}_1(\mathcal{R}))$.

Table 5.1 The relational operators and their properties w.r.t to the proposed extensions.

| Name | Symbol | Arity | Order | Commutativity | Distributivity |
|---------------------|---|-------|-------------|------------------------------------|--|
| Union | $\mathcal{R}_L^{O_1} \cup \mathcal{R}_R^{O_2}$ | 2 | \emptyset | ✓ | - |
| Intersection | $\mathcal{R}_L^{O_1} \cap \mathcal{R}_R^{O_2}$ | 2 | O_1 | ✗ | - |
| Difference | $\mathcal{R}_L^{O_1} \setminus \mathcal{R}_R^{O_2}$ | 2 | O_1 | ✗ | - |
| Cartesian Product | $\mathcal{R}_L^{O_1} \times \mathcal{R}_R^{O_2}$ | 2 | \emptyset | ✗ | - |
| Natural Join | $\mathcal{R}_L^{O_1} \bowtie_{\mathcal{P}} \mathcal{R}_R^{O_2}$ | 2 | O_1 | ✗ | - |
| Rename | $\rho_{\mathcal{A}_A \rightarrow \mathcal{A}_B}(\mathcal{R}^O)$ | 1 | O | $\pi, \sigma, \rho, \lambda, \tau$ | $\cup, \cap, \setminus, \times, \bowtie$ |
| Selection | $\sigma_{\mathcal{P}}(\mathcal{R}^O)$ | 1 | O | π, σ, ρ, τ | $\cup, \cap, \setminus, \times, \bowtie$ |
| Extended Projection | $\pi_{\mathcal{E}}(\mathcal{R}^O)$ | 1 | O | $\pi, \sigma, \rho, \lambda, \tau$ | \cup |
| Limit | $\lambda(\mathcal{R}^O)$ | 1 | O | π, ρ | ✗ |
| Order | $\tau_O(\mathcal{R}^{O'})$ | 1 | O | π, σ, ρ | \cap, \setminus, \bowtie |

For binary operators, we must consider the *commutativity of operands*, i.e., whether $OP(\mathcal{R}_L, \mathcal{R}_R) = OP(\mathcal{R}_R, \mathcal{R}_L)$. Furthermore, w.r.t. to unary operation, there is *distributivity* to consider, i.e., whether $OP_1(OP_2(\mathcal{R}_L, \mathcal{R}_R)) = OP_2(OP_1(\mathcal{R}_L), OP_1(\mathcal{R}_R))$.

In addition, to support ranked relations, we must require all existing, relational operators to be either *order retaining* or *order inducing*. A summary of these properties can be found in Table 5.1.

It follows directly from its definition, that τ_O is an order inducing operator. Furthermore, σ, π, ρ and λ are order retaining, since they do not alter the input relation w.r.t ordering.

It is a bit more complicated for the binary operators, which do not behave consistently: On the one hand, the set difference (\setminus), the set intersection (\cap) and the natural join ($\bowtie_{\mathcal{P}}$) retain the order of the left operand \mathcal{R}_L , since the operators retain or remove elements from \mathcal{R}_L based on their presence in \mathcal{R}_R , without taking ordering into account. As a consequence, the operands no longer commute, since it is always the left operand that defines the output ordering.

The set union (\cup) and the cross product (\times), on the other hand, combine two relations \mathcal{R}_L and \mathcal{R}_R . The resulting relation does not have an order that can be expressed in the proposed framework and must thus be considered unordered, i.e., the operators are order inducing in that they induce a new, natural ordering.

When turning to the individual operators, the extended projection $\pi_{\mathcal{E}}$ inherits all its properties from the normal projection π if we require $f \in \mathbb{F}$ to be stateless and side-effect free, i.e., it must not influence the state of the DBMS in any other way than producing the desired output and it must not maintain an internal state itself. Irrespective of $\pi_{\mathcal{E}}$, this is a reasonable assumption also from a implementation perspective.

Double-check properties in table.

The order operator τ_O is idempotent and commutes with other unary operators if they are neither *rank sensitive* nor *rank inducing*, namely, π , σ and ρ . Importantly, τ_O does not commute with itself.

$$\tau_{O'}(\tau_O(\mathcal{R})) \neq \tau_O(\tau_{O'}(\mathcal{R}))$$

In addition, distributivity with all the order retaining binary operations is given, e.g., when taking set intersection as an example.

$$\tau_O(\mathcal{R}_L^{O_L} \cap \mathcal{R}_R^{O_R}) = \tau_O(\mathcal{R}_L^{O_L}) \cap \tau_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \cap \mathcal{R}_R)^O$$

However, the order operator does not distribute with the order inducing, binary operators \cup and \times , for example, taking set union as an example.

$$\tau_O(\mathcal{R}_L^{O_L} \cup \mathcal{R}_R^{O_R}) = (\mathcal{R}_L^{O_L} \cup \mathcal{R}_R^{O_R})^O \neq \tau_O(\mathcal{R}_L^{O_L}) \cup \tau_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \cup \mathcal{R}_R)^O$$

Finally, the k-selection operator λ is also idempotent but does neither commute with the selection operator σ nor the order operator τ .

$$\lambda(\tau(\mathcal{R})) \neq \tau(\lambda(\mathcal{R}))$$

$$\lambda(\sigma(\mathcal{R})) \neq \sigma(\lambda(\mathcal{R}))$$

Lack of commutativity of λ with τ follows directly from its definition: λ is sensitive to the ranking of \mathcal{R}^O and τ influences that ranking. Consequently, executing λ before or after imposing a certain order will yield different results. For commutativity with σ , one can turn to Proof 5.1.

Proof 5.1 Non-commutativity of σ and λ

Let \mathcal{R} be a relation with M rows and $\text{SCH}(\mathcal{R}) = \{\mathcal{A}_R\}$ and let further be $t[\mathcal{A}_R] = \text{RANK}(t) \forall t \in \mathcal{R}$. If σ and λ_k were to commute, then

$$\sigma_{\mathcal{A}_R > M-2}(\lambda_{M-1}(\mathcal{R})) = \lambda_{M-1}(\sigma_{\mathcal{A}_R > M-2}(\mathcal{R}))$$

However, we can proof by contradiction that this does not hold since:

$$\sigma_{\mathcal{A}_R > M-2}(\lambda_{M-1}(\mathcal{R})) = \{t_M\} \neq \lambda_{M-1}(\sigma_{\mathcal{A}_R > M-2}(\mathcal{R})) = \{t_{M-1}, t_M\}$$

That is, whenever λ_k removes tuples that match the predicate of σ , the order of execution influences the result and the two operators do not commute.

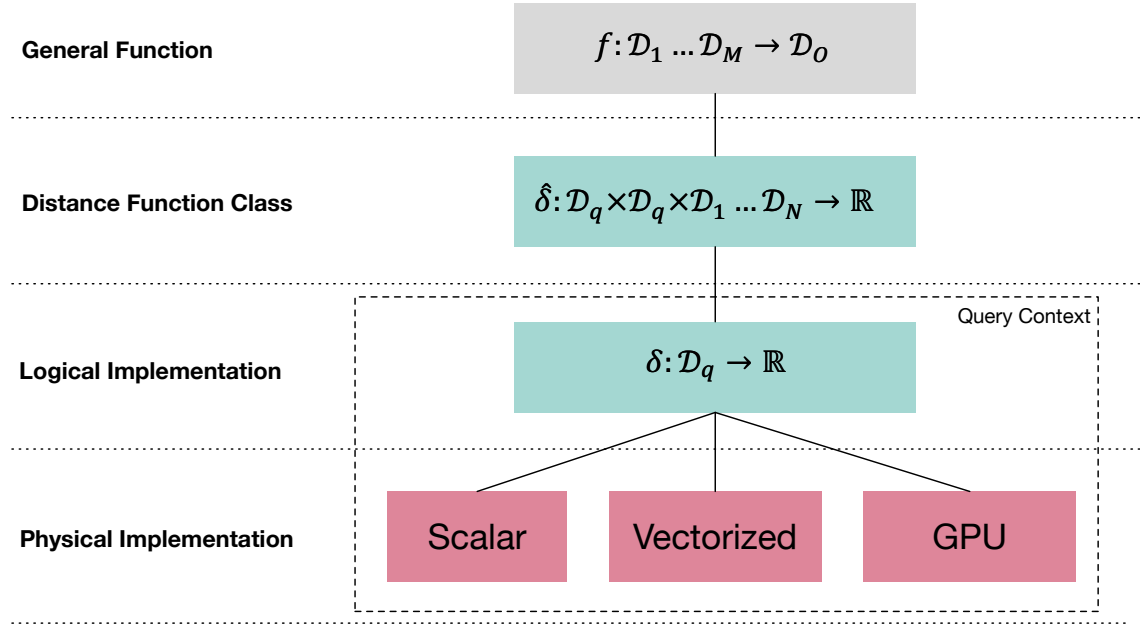


Figure 5.2 Function hierarchy of all $f \in \mathbb{F}$. As we go down the hierarchy, the DBMS gains knowledge about a function's structure.

5.1.3 DFCs and Query Planning

In addition to the algebraic properties of the extended relational algebra, we can also leverage the requirements imposed on the structure of a function $f \in \mathbb{F}$ to enable a DBMS to plan and optimize the execution of a proximity based query by altering the execution plan in terms of DFC implementations that should be employed. To do so, we must first assume that a DBMS maintains the collection of functions \mathbb{F} . Since every function can be identified by its signature SIG , \mathbb{F} can be used for obtaining the correct function implementation given the specification of the query. We refer to this data structure as *function registry*, which can also be used to store additional properties and metadata.

In addition, we propose the notion of a *function hierarchy* depicted in Figure 5.2. As we move down that hierarchy, more information about the structure of a DFC and the concrete implementation employed in a query plan becomes available to the DBMS. That information can be leveraged when deciding about the best execution path for a query, wherein every level offers certain properties that can be leveraged:

General Function Every function $f \in \mathbb{F}$ is known with information about its name and domain (i.e., its signature $\text{SIG}(f)$). This enables the DBMS to provide optimised implementations or to provide replacements for composite operations.

Distance Function Class Since DFCs constitute a specific type of function, we can embedd equivalences between a DFC and different types of high-dimensional index structures directly within the function registry. Those equivalences allow a query planner to select one over the other, for a particular plan.

Logical Implementation As an implementation of a query manifests, the structure of a DFC may change through implementation $\delta = \text{IMP}(\hat{\delta})$. For example, certain attributes may turn-out to remain constant, which can be leveraged by the planner to optimise execution.

Physical Implementation As the physical execution plan emerges, a plan can make decisions as to what type of implementation for a DFC should be selected. For example, it may select a vectorised version of a DFC, which leverages Single Instruction Multiple Data (SIMD) instructions of the CPU, over a scalar implementation that calculates the distance in a tight loop.

5.1.3.1 Functions and Optimised Implementations

Optimising implementations of DFCs and general function invocations can be achieved in multiple ways. First, it is reasonable to provide and select implementations that are specific for the data type of the function arguments, so as to avoid type conversion in a tight loop over a large amount of data. From the perspective of execution performance, it is desirable to make this selection during planning rather than at query execution time.

Another aspect that can be optimised is the use of vectorized implementations for computations that involve vectors or matrices, i.e., exchanging a scalar implementation of a DFC by an implementation that uses SIMD instructions. However, it is to expected that one challenge lies in finding the threshold after which usage of such an optimised version pays off. If we take real-valued vectors as an example, it is likely that vectorised execution will not benefit low-dimensional vectors (e.g., $\text{dim} = 3$) while it will greatly benefit high-dimensional vectors (e.g., $\text{dim} = 2048$). Finding this threshold poses challenge in and by itself, which could be subject for future research.

Finally, one can also try to match constellations of nested functions and replace them with optimised version. An example could be the expression $a * b + c$, i.e., $\text{add}(\text{mul}(a, b), c)$, which can be replaced by a single *fused multiply-add* function $\text{fma}(a, b, c)$ to avoid loss of preission. In a manner of speaking, trading-in a high-dimensional index for the execution of a DFC also falls into this category.

5.1.3.2 DFCs and High-dimensional Index Structures

One identity that is particularly relevant for the execution of proximity based queries is the relationship between the execution of a DFC and a high-dimensional index. While there are many different types of HD-indexes, we distinguish between three broader categories proposed by [LÁJ⁺18] and introduced and discussed in Chapter 4, namely *quantization based* (e.g., PQ or VAF), *hash based* (e.g., LSH or SH) and *tree based* (e.g., NV-tree [LJA11]) indexes. We postulate the relationship between a high-dimensional index and a DFC given in Definition 5.6.

Definition 5.6 DFCs and high-dimensional index structures (exact case).

Let $\hat{\delta}: \mathcal{D}_q \times \mathcal{D}_q \times \mathcal{D}_1 \dots \times \mathcal{D}_n \rightarrow \mathbb{R}$ be a DFC and $\pi_{\mathcal{E}}$ be the extended projection involving execution of that DFC, i.e., $\hat{\delta} \in \mathcal{E}$. A high-dimensional index INDEX is a pre-computed database object such that $\pi_{\hat{\delta}(\cdot)}(\mathcal{R}) = \text{INDEX}(\mathcal{R})$ and therefore, the following equation holds:

$$\pi_{\mathcal{E}}(\mathcal{R}) = \pi_{\mathcal{E} \setminus \hat{\delta}}(\mathcal{R}) \bowtie \text{INDEX}(\mathcal{R})$$

We call such an index *exact* and we can use it as a replacement for the DFC.

In addition to Definition 5.6, the following restrictions apply (in addition to implementation specific restrictions for a specific index):

- Indexes are always derived from a materialized relation \mathcal{R} and a feature attribute $\mathcal{A}_f \in \text{SCH}(\mathcal{R})$. Therefore, an index can only replace a DFC if they operate on an unaltered version of \mathcal{A}_f . Changes to \mathcal{A}_f , e.g., through application of a nested function, will render the index unusable unless the same alteration was applied when creating the index.
- Indexes are usually trained for a specific distance function (e.g., Euclidean) or class of distance functions (e.g., Minkowski). Therefore, an index can only replace a DFC if that DFC coincides with the original function (-class).

With Definition 5.6 and the aforementioned, we have defined the basic relationship and rule that can be applied by a DBMS, when deciding whether or not to use a particular high-dimensional index structure. We will provide more insights into the nature of *approximate* index structures in Section 5.2.

5.2 Cost Model for Retrieval Accuracy

Describe cost model for execution plans with following properties:

- Cost as a function of atomic costs: $f(a_{cpu}, a_{io}, a_{memory}, a_{accuracy}) \rightarrow C$
- Means to estimate results accuracy and associated considerations from execution path (e.g., when using index) based on properties of the index
- Means to specify importance of accurate results (e.g., global, per-query, context-based i.e. when doing 1NN search) in comparison to other factors
- Systems perspective 1: How can such a cost model be applied during query planning and optimization?

5.3 Adaptive Index Management

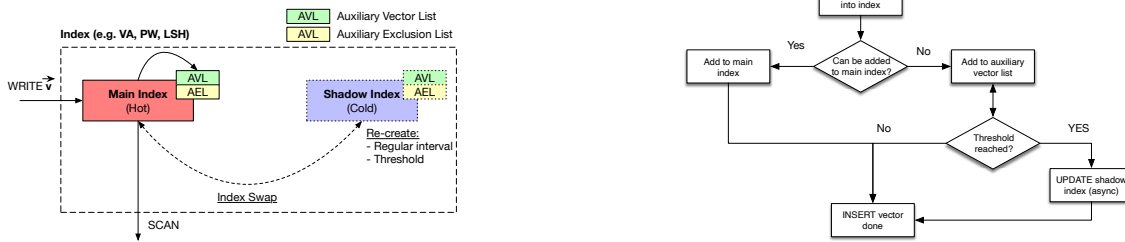


Figure 5.3 Adaptive index structures overview.

Describe model for index management in the face of changing data (adaptive index management):

- Reason about properties of secondary indexes for NNS (e.q., PQ, VA, LSH) with regards to data change
- Derivation of error bounds possible (e.g., usable for planning)?! Use in query planning?
- Systems perspective 1: How to cope with “dirty” indexes? Proposal: hot vs. cold index, auxiliary data structure, offline optimization, see Figure 5.3
- Systems perspective 2: On-demand index based on query workload?

5.4 Architecture Model

Putting everything together into a unified systems model (base on previous work + aforementioned aspects).

6

Cottontail DB

Cottontail DB [GRH⁺20] is our reference implementation for the query, cost and index-models described in Chapter 5. Starting out as a drop-in replacement for ADAMpro [GS16], it has grown to be a full fledged DBMS with support for both classical boolean retrieval as well as proximity based search used in multimedia retrieval and analytics applications. Both aspects are combined in a unified query and execution model and many explicit and implicit decisions, which are described in this chapter, went into its conception and design.

Cottontail DB is written in Kotlin¹ and runs on the Java Virtual Machine (JVM). *Cottontail DB*'s source code can be downloaded from GitHub². It is freely available under a permissive open source license and can be used as standalone application, Docker container or in an embedded mode.

6.1 Data Model and Nomenclature

Cottontail DB uses a data model very similar to that of other relational DBMS': All data is organized into *entities* – which correspond to tables or relations. Every entity consists of individual, strongly typed *columns*. The different data types currently supported are listed in Table 6.1. Columns can hold values of the declared type or NULL to indicate the absence of information (if the column has been declared as being nullable). To support the features used in multimedia retrieval applications, vector types are first-class members the type system.

An entity can host multiple *records*, which correspond to tuples in the relation. Since *Cottontail DB* is a column store, records are only logical constructs in memory and do not reflect the physical data representation on disk. Conse-

¹ See: <https://kotlinlang.org/>

² See: <https://github.com/vitrivr/cottontaildb>

Table 6.1 Data types supported by *Cottontail DB*. Types in the numeric, vector and complex class allow for domain specific arithmetics.

| Name | Scalar | Numeric | Vector | Complex | JDBC / SQL |
|------------------|--------|---------|--------|---------|-----------------------|
| String | ✓ | ✗ | ✗ | ✗ | VARCHAR |
| Date | ✓ | ✗ | ✗ | ✗ | TIMESTAMP |
| Boolean | ✓ | ✓ | ✗ | ✗ | BOOLEAN |
| Byte | ✓ | ✓ | ✗ | ✗ | TINYINT |
| Short | ✓ | ✓ | ✗ | ✗ | SMALLINT |
| Int | ✓ | ✓ | ✗ | ✗ | INT |
| Long | ✓ | ✓ | ✗ | ✗ | BIGINT |
| Float | ✓ | ✓ | ✗ | ✗ | REAL |
| Double | ✓ | ✓ | ✗ | ✗ | DOUBLE |
| Complex 32 | ✓ | ✗ | ✗ | ✓ | - |
| Complex 64 | ✓ | ✗ | ✗ | ✓ | - |
| Boolean Vector | ✗ | ✗ | ✓ | ✗ | ARRAY[BOOLEAN] (1, d) |
| Integer Vector | ✗ | ✗ | ✓ | ✗ | ARRAY[INT] (1, d) |
| Long Vector | ✗ | ✗ | ✓ | ✗ | ARRAY[BIGINT] (1, d) |
| Float Vector | ✗ | ✗ | ✓ | ✗ | ARRAY[REAL] (1, d) |
| Double Vector | ✗ | ✗ | ✓ | ✗ | ARRAY[DOUBLE] (1, d) |
| Complex32 Vector | ✗ | ✗ | ✓ | ✗ | - |
| Complex32 Vector | ✗ | ✗ | ✓ | ✗ | - |

quently, records are assembled on-the-fly as queries are being executed. Similarly to columns, every record is also strongly typed, wherein a record is a tuple type of its strongly typed elements. Internally, every record is uniquely identified by a *tuple ID* (TID), which is a long value that can be used to address the record within an entity or an (in-memory) *recordset*. This TID is not exposed to the outside because it remains at the discretion of the storage and execution engine to generate, assign, change and (re-)use them as data gets (re-)organized.

Furthermore, *Cottontail DB* allows for multiple entities to be grouped into *schemata*, which currently serves an organisational purpose only. Every entity can also host one or multiple secondary *indexes* that index a single or multiple *columns* for more efficient data access. In addition to indexes for Boolean and Fulltext search, *Cottontail DB* also hosts different types of high-dimensional index structures used for proximity based queries, for example, PQ [JDS10], VAF [WSB98] and LSH-based [Indyk:1998lsh] indexes.

To address database objects, *Cottontail DB* uses a hierarchical namespace, i.e., every schema, entity, column and index must be uniquely named by a *fully qualified name*. All the information about the database objects is tracked in an internal *catalogue*, which is backed by the main storage engine.

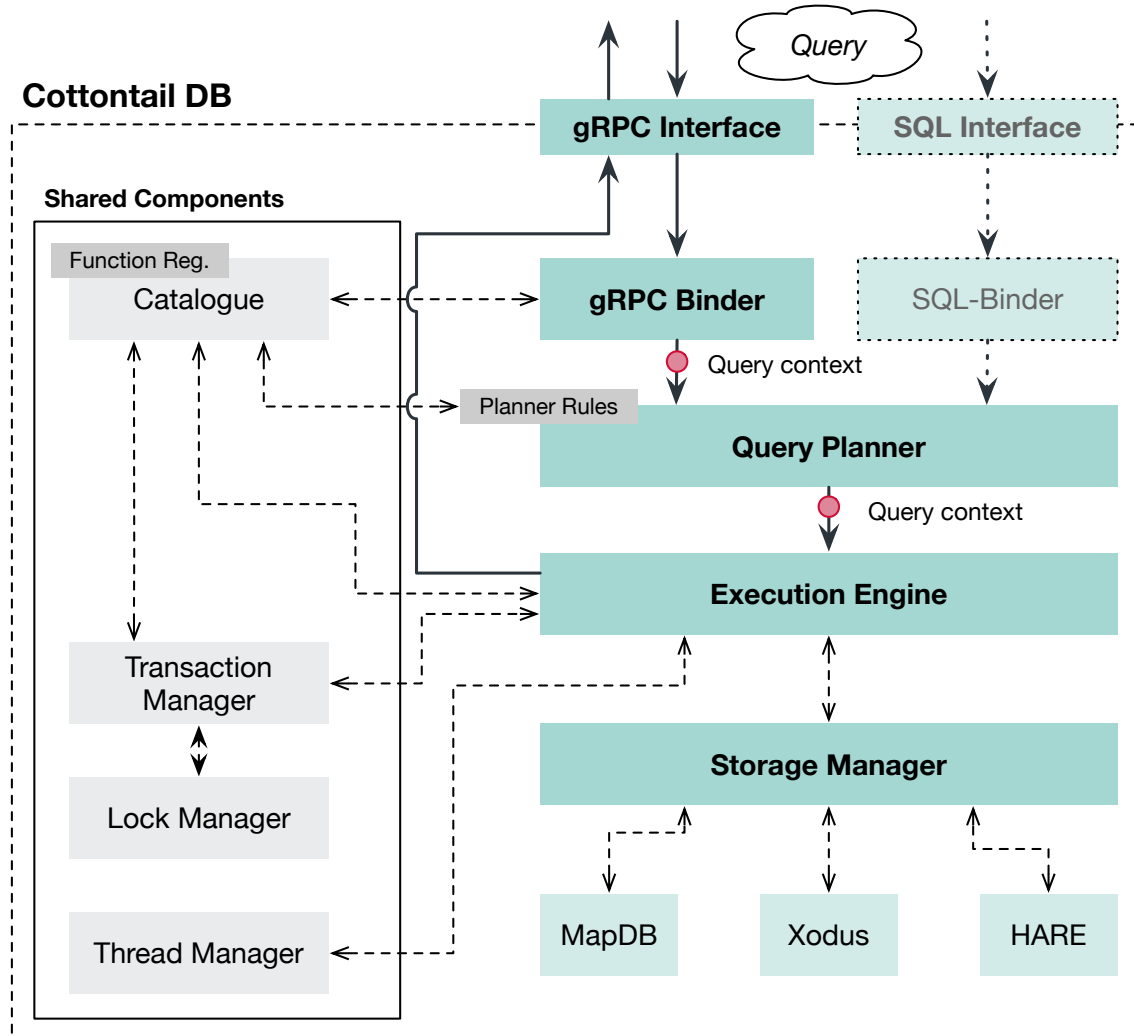


Figure 6.1 Architecture diagram of *Cottontail DB*'s main components. The directed arrows indicate the path a query takes within the system. The dashed, double arrows indicate interactions between components.

6.2 Architecture

The main components of *Cottontail DB* are depicted in Figure 6.1. We use the path a query takes within the system, as indicated by the directed arrows, to illustrate the components involved in its implementation.

At a high level, every query must undergo *parsing*, *binding*, *planning* and *execution* in that order, even though, planning may be skipped for certain types of queries (e.g. DDL statements). Starting with the binding step, all the information required for query processing is accumulated in a *query context* object, which is passed between the steps. In the following sections, we will describe the aforementioned steps in reverse order, since important concepts can be introduced more naturally that way.

Table 6.2 Main types of operators implemented by *Cottontail DB* alongside with their arity, their correspondence to relational operators and whether or not they require materialization.

| Type | Arity | Rel. Op. | Mat. | Description |
|----------|-------|---|------|---|
| Scan | 0 | $\pi_{\mathcal{E}}$ $\sigma_{\mathcal{P}}$ $\rho_{\mathcal{A}_A \rightarrow \mathcal{A}_B}$ | | Act as data sources. Usually scan an <i>entity</i> or an <i>index</i> and thus interact directly with the storage manager to do so. |
| Fetch | 1 | $\pi_{\mathcal{E}}$ | | Extend every incoming record by fetching values for one or multiple columns and appending them to the record. Usually interact directly with the storage manager to perform random reads. |
| Function | 1 | $\pi_{\mathcal{E}}$ | | Extend every incoming record by evaluating a specific function and appending the result as a new column to the record. |
| Filter | 1 | $\sigma_{\mathcal{P}}$ | | Filter incoming records by evaluating a given predicate. Records that don't match the predicate are not handed down the pipeline. |
| Sort | 1 | $\tau_{\mathcal{O}}$ | ✓ | Sort the incoming records based on the specified columns in the specified direction. |
| Limit | 1 | λ_k | | Skip and drop incoming records according to specification and thus limit the number of out-bound records to a specified number |
| Project | 1 | π_{χ} $\rho_{\mathcal{A}_A \rightarrow \mathcal{A}_B}$ | | A terminal operator that actively removes and/or renames columns in the incoming records. This is sometimes necessary because columns that are fetched for processing may not be desired in the final result. |

6.2.1 Query Implementation and Execution

Cottontail DB implements an *iterator model* for query execution. This means that an implemented query is a pipeline of operators, wherein each operator processes single records it receives from its upstream (input) operator(s) and passes single records to the next operator in the pipeline. Most operators have a direct correspondence to the relational operators (Chapter 3) and extensions (Chapter 5) introduced thus far. However, not all operators behave exactly as required by the model due to implementation details.

Conceptually, *Cottontail DB* distinguishes between *Nullary*, *Unary*, *Binary* and *N-Ary* operators, which differ in the number of inputs they can accept (0, 1, 2 or N). Furthermore, some operators require materialization of intermediate result-sets and therefore act as pipeline breakers since they must collect all inbound records before they can start handing them down (e.g., sort operator). The most important types of operators implemented by *Cottontail DB* along with their correspondence to the relational operators are listed in Table 6.2.

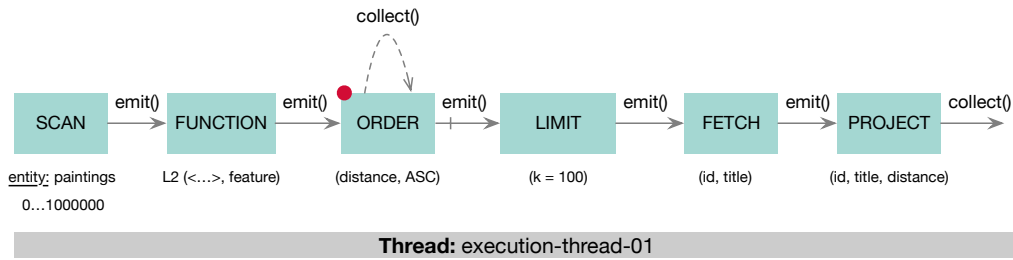


Figure 6.2 Operator pipeline for a proximity based query that scans the entity *paintings*, calculates the L2 distance between a *feature* column and a query, orders by the obtained distance and limits the number of outputs (kNN). Pipeline breaking operators are indicated by the red dot.

Every operator is implemented in Kotlin and pre-compiled to byte code. Ergo, there is no on-the-fly code generation at runtime. The operator implementations are derived from the plan that is selected in the planning stage.

6.2.1.1 Execution Model

Operators in *Cottontail DB* are implemented as *coroutines*, i.e., every operator is a suspendable function that yields to the next operator in the pipeline upon emission of a value. Each operator collects a single record from its input operator(s), performs processing and forwards the resulting record by calling `emit()`, thus yielding to the next operator in the pipeline. Source operators usually iterate some data collection through a cursor and thus only `emit()` values, whereas sink operators `collect()` values without emitting anything (practically, the collected values are usually sent to the client that issued the query). Pipeline breaking operators are required to `collect()` all inbound values before they can start to `emit()`, i.e., they materialize a recordset. The suspension and continuation of operator calls is orchestrated by Kotlin’s coroutines “Flow” framework³. A very simple example of a naive operator pipeline is provided in Figure 6.2.

Inter-query parallelism is achieved, by collecting different operator pipelines on different threads. The assignment of operator pipeline to thread is delegated to a *dispatcher*, which has a defined thread pool at its disposal and which is part of Kotlin’s coroutines framework. Transaction isolation is guaranteed at all time by the *transaction manager*.

In addition, this execution model also allows for intra-query parallelism, which is used for queries that incur high CPU cost. This is realised in a *data parallel* fashion by partitioning on the input. The decision whether or not to allow for parallelism is made upon implementation of a query – i.e., the last step

³ See: <https://kotlinlang.org/docs/coroutines-overview.html>

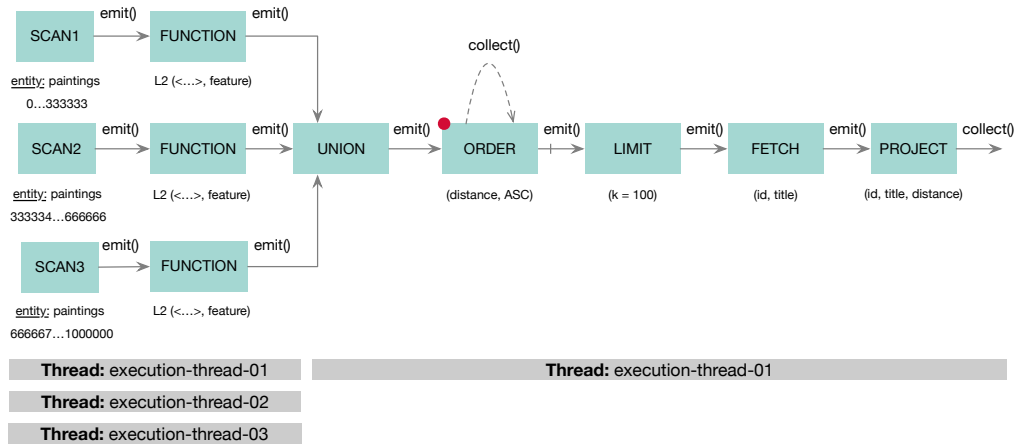


Figure 6.3 Operator pipeline that performs kNN on the entity *paintings*. The *scan* and distance *function* steps are partitioned and unioned before order, limit and projection is applied, which allows for parallel execution.

after the final plan has been selected – since it requires structural changes to the operator pipeline. The degree of parallelism depends on the expected CPU cost of the plan and the available CPUs on the machine *Cottontail DB* is running on.

To prepare for intra-query parallelism, part of the operator pipeline is partitioned wherein every partition operates on a fraction of the input data. Since not all steps in a pipeline can be parallelised effectively, usually only parts of the pipeline are partitioned. Whether partitioning is possible at a given point in the pipeline depends on whether the operator itself as well as all upstream operators allow for partitioning, which is queried upon implementation. An additional concurrency aware *union* operator is introduced after such an operator, which synchronizes and merges all the incoming strands using a FIFO scheme. Since union operators do not retain ordering, sort operations must take place afterwards. An example for this is given in Figure 6.3.

With an operator pipeline prepared in this fashion, it again remains up to the dispatcher to allocate parts of the pipeline to concrete threads. *Cottontail DB* only declares how allocation should take place by choosing the appropriate type of dispatcher. Scheduling then depends on available hardware and system load. Consequently, the number of partitions is merely an upper bound on parallelism without guarantee, that every strand will be processed in its own thread.

6.2.2 Query Planner

The goal of *query planning* is to find a cost-optimal execution plan for a query given any *policy* valid for the current query context and *Cottontail DB*'s cost model. The input into the query planner is a logical representation of the oper-

ators that should be executed as part of the operator pipeline, which we call the *canonical operator node tree*. It results directly from the *query binding* stage.

Query planning in *Cottontail DB* then takes place in three steps: First, and optionally, the canonical operator node tree is compared to the *plan cache*. If the cache contains a hit for that tree, the already optimized plan in the cache can be re-used and all the subsequent steps can be skipped. Second, if there is no hit in the cache or the cache is bypassed, the canonical operator node tree undergoes *optimization*. The optimization step generates different versions of the input plan by applying optimization rules that act on and manipulate the tree to generate new, equivalent plans. Finally, once optimization has concluded, the plan that minimizes the expected cost is selected and *implemented*, that is, converted to a concrete sequence of executable operators.

6.2.2.1 Basic Optimization Algorithm

The query planner used by *Cottontail DB* is rather naive and can be categorized as a rule-based query planner. It is loosely inspired by the Cascades Framework that was built as part of the Volcano project [GM93]. The planner generates new plans by enumerating and applying a set of rules to the input plan(s) in a recursive fashion. These rules are transformations that recognize patterns in the structure of the operator node tree and try to re-arrange and/or replace nodes in the tree to generate a more optimal but equivalent version of the plan.

The optimization follows the same, basic algorithm: The available list of rules is enumerated and every rule is applied to every node starting from the base of the tree. The application of rules takes place recursively, i.e., every node propagates a rule up the tree to its inputs. At every node, the rule first checks if it can be applied to the current node. This check usually incurs little cost and simply considers the type of node and other, available properties. If the check is successful, the actual application of the rule follows. This application may or may not yield a new version of the query plan, depending on more complex and more costly factors, such as the availability of an index, which requires a catalogue lookup. If a new plan results from the application of a rule, that new query plan is stored in a *memoization table* for future reference.

Every planning stage involves a defined number of passes, and the sum of all plans generated by one pass act as the input for the next. All the intermediate plans are therefore run through the same algorithm and all the rules are applied again. Consequently, the search space grows exponentially with the complexity of the input plan, the number of rules to apply and the number of passes. To

address this issue, *Cottontail DB*'s query planner employs certain heuristics to keep the search space manageable:

- Very large, complex plans (e.g., stemming from sub-selects or JOINS) are broken into groups and every group is optimized and treated as an isolated plan. The assumption being that the combination of smaller, near-optimal plans must be close to optimal as well.
- The planning of every group is done in two stages – a *logical* and *physical* stage – with distinct and disjoint sets of rules. The results of the logical stage act as an input for the physical stage.
- Logical and physical query plans are uniquely identified by a hash. Memoization in combination with the aforementioned hash is used to track trees that have already been optimized and skip them.
- Physical execution plans generated in the second stage are actively pruned by the planner between the different passes based on the cost estimate.

Between the logical and the physical planning, there is a step that maps every logical operator node to its naive, physical counterpart. The number of passes per stage and active pruning of plans during the physical phase are used to steer the high-level behaviour of the query planner. During the logical phase, the goal is to generate as many, equivalent, logical query representations as possible to have a corpus of plans to optimize on (expansion phase). In contrast, during the physical phase, the planner tries to limit the number of intermediate plans to prevent the search space from exploding.

6.2.2.2 Plan Caching

The plan cache is an optimization mechanism that amortizes the cost of the potentially expensive query planning if a certain type of query is encountered multiple times. At its core, the cache maps the hash code of every incoming canonical operator tree to the resulting, optimized physical plan. If the same query is encountered again, that plan can be looked-up and re-used directly, hence, avoiding another round of planning.

Currently, there are certain limitations to the plan cache mechanism in *Cottontail DB*, which is why it is disabled by default. Firstly, the actual execution performance of a plan is not being evaluated against the cost model, i.e., the level of optimality of an execution plan is not assessed. And secondly, there is no mechanism that invalidates cached entries as changes to the data occur.

6.2.2.3 Logical Optimization

The logical optimization step acts on structural properties of the operator node tree and aims at expanding the list of query plans in a way that increases the likelihood of finding the optimal plan during the physical phase. It basically leverages the algebraic properties of the operators involved. The logical optimization stage in *Cottontail DB* currently involves the following type of rules:

Defer Fetch On Scan Rewrite Rule This rule tries to defer access to columns upon scanning the data. It makes sure, that only columns are accessed during a table scan, that are required by the first operator following the scan that requires the presence of specific columns. All remaining columns are pushed down in the tree into a dedicated *fetch* operation following that node.

Defer Fetch On Fetch Rewrite Rule This rule tries to defer access to columns upon fetching data. Similarly to the *Defer Fetch on Scan Rewrite Rule*, it pushes the fetching of columns not required by any of the downstream operators that express a specific requirement, into a dedicated fetch operation following such an operator.

Conjunction Rewrite Rule This rule refactors a filter operator evaluating a conjunctive predicate into two consecutive *filter* operators, one for each side of the conjunction. There are two versions of this rule, one that applies the left side first and one that applies the right side first.

The first two rules make sure, that columns are accessed only if they are needed and as close to the site of processing as possible. Furthermore, and since these rules are applied multiple times, columns that are not accessed at all will eventually be pushed out and eliminated from the tree completely. This is illustrated in the example given in Figure 6.4. Since *Cottontail DB* is a column store, deferral of column access can drastically reduce I/O, especially in the presence of filtering predicates or if a column is not needed.

The conjunction rewrite rule decomposes filtering predicates that contain conjunctions. This is illustrated in the example given in Figure 6.5. The application of this rule can be seen as a preparation for the physical optimization phase, where specific predicates may be pushed down to a secondary index.

6.2.2.4 Physical Optimization

The physical optimization step tries different implementations of operators to arrive at a more cost-effective plan. While structural changes to the operator

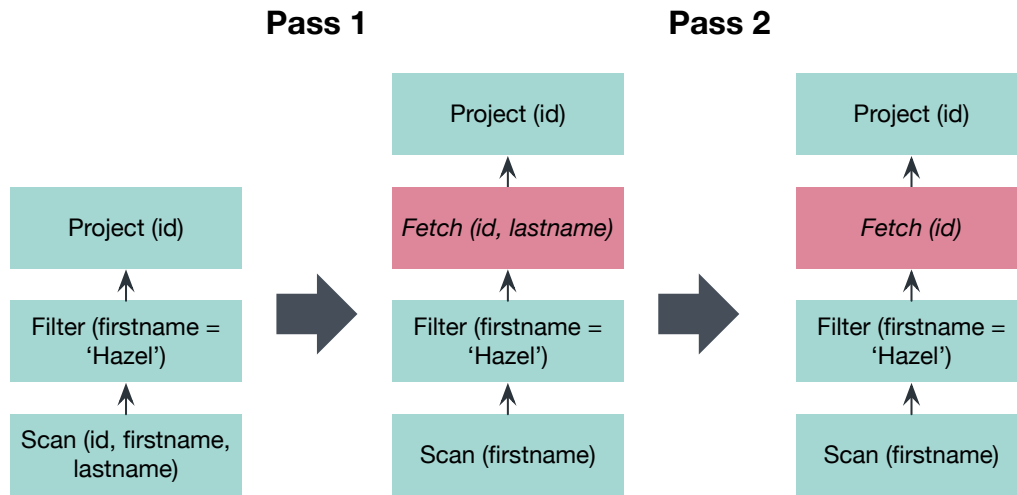


Figure 6.4 Illustration of deferral of column access. In the first pass, access to columns *id* and *lastname* is deferred until after the filter operation, because the filter only requires *firstname*. In the second pass, access to *lastname* is eliminated completely.

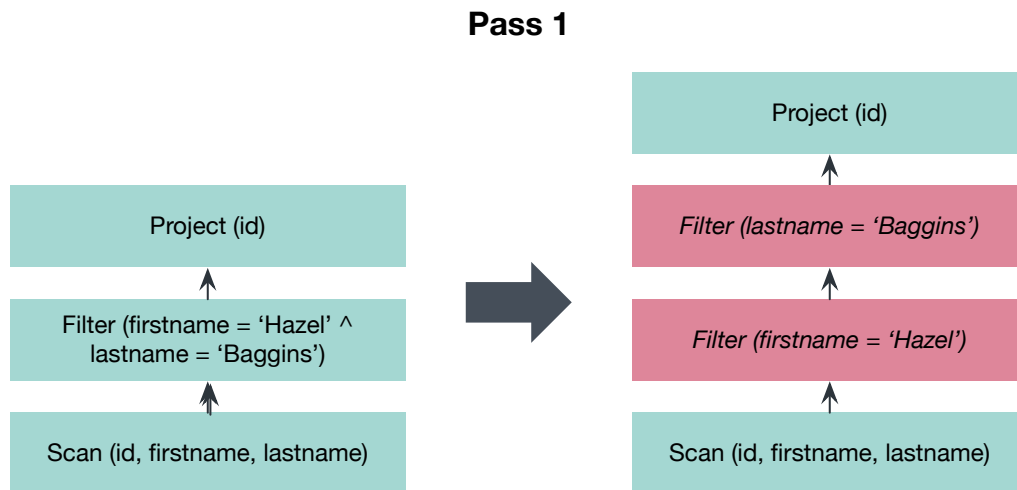


Figure 6.5 Illustration of decomposing conjunctive predicates. In this example, the filter on the column *firstname* may be pushed down to an index during physical optimization, if such an index is available.

tree may be possible, the focus lies on implementation aspects. At a high level, we distinguish between the following three types of rules:

Pushdown Rules These rules try to push down *filter* predicates or *function* invocations to the *scan* operation, usually by leveraging a secondary index or existing statistics (e.g., for counting). For example, an entity scan followed by a filter can be replaced by a more efficient scan of a B^+ -tree index followed by a *fetch* operation or proximity based operations can be delegated to an available, high-dimensional index using Definition 5.6.

Summarization Rules These rules try to merge operations to arrive at a more efficient execution. For example, a *sort* operation followed by a *limit* operation can be replaced by a single, *limiting heap sort* operation, which applies sorting and limiting in a single step. Combining these two operations and the use of the heap-sort algorithm allows for more effective sorting in terms of CPU and memory cost, as compared to naively sorting all records first and then limiting to the top k records thereafter.

Implementation Rules These rules simply replace different implementations of a operation one-to-one, e.g., hash-join vs. nested-loop join for a theoretical JOIN operation.

After each pass in the physical optimization step, the list of resulting plans is pruned to the n best performing plans before handing them to the next pass. By this, we limit the size of the search space that is being explored. Once all passes have concluded, the best plan in terms of cost is selected and implemented. We directly apply the cost model introduced and described in Chapter 5.

6.2.3 Query Parsing and Binding

As every DBMS, *Cottontail DB* uses a query language that allows for interaction with other systems. These interactions include data definition, data management, transaction management and the formulation of queries. In its current version, *Cottontail DB*'s query language is based on *gRPC*⁴, hence, all queries are expressed as *Protocol Buffer*⁵ messages. Internally, the *gRPC* endpoint is a collection of four services for handling DDL, DML, DQL and transaction management messages. The parsing of incoming messages, the mapping of messages to the correct service and endpoint and low-level handling of communication with clients is delegated to the *gRPC* Kotlin library.

The reason for the choice of *gRPC* is threefold: Firstly, *Cottontail DB*'s predecessor – ADAMpro [GS16] – already used and introduced *gRPC* into the *vitivr* ecosystem. Consequently, building on this work greatly simplified the task of integrating *Cottontail DB*. Secondly, *gRPC* – being a platform independent remote procedure call framework – introduces out-of-the-box support for a large range of programming environments⁶. And finally, the *gRPC* Kotlin library

⁴ See: <https://grpc.io/>

⁵ See: <https://developers.google.com/protocol-buffers/>

⁶ Furthermore, there are specialized client libraries for Java, Kotlin and Python

greatly simplified the task of structuring query endpoints, parsing incoming query messages and handling the query responses.

Irrespective of the use of gRPC, *Cottontail DB* is compatible with the SQL standard insofar as it supports a specific feature. This was demonstrated by the successful integration of *Cottontail DB* as a storage engine for *Polypheny-DB*. Consequently, it is possible to add additional endpoints that can accept, for example, SQL queries. However, for complete SQL compliance, one would have to add language features that are currently not supported by *Cottontail DB* (e.g., JOINS). Furthermore, handling the technical details of parsing the incoming query and communication with the client must be handled as well, which would lead to the implementation of many different standards such as JDBC or ODBC.

Irrespective of its source, any incoming query message undergoes a *binding* step in *Cottontail DB*. Query binding includes three aspects:

1. Names that reference database objects such as entities, columns and indexes are resolved using *Cottontail DB*'s catalogue and replaced by concrete pointers to the respective object.
2. Literal values, function calls and column references, e.g., in predicates, are extracted and replaced by *bindings* and attached to a *binding context*.
3. The parsed message is converted into a *canonical operator node tree* that logically represents the query.

The result of the parsing and binding step is the canonical operator node tree, which acts as a logical and unoptimized query representation. Every node in the tree represents an operator that acts on a *record* or *tuple* and that must be executed in order to generate the result specified by the query. The conversion of a message to a tree is performed in a naive manner and only syntactic checks and optimization are performed at this stage (e.g., type checks, removal of trivial predicates such as " $1 = 1$ ").

6.2.3.1 Binding Context

The binding context is a data structure that is member of the query context and a very important piece of functionality for query planning and execution. At their core, bindings are proxies for values used in a query. *Cottontail DB* distinguishes between *literal*, *column* and *function* bindings. At any point during query execution, the binding context holds the current values for all the bindings associated with it, so that these can be accessed as the query is being evaluated.

The layer of indirection introduced by bindings enables many of *Cottontail DB*'s functionalities, e.g., in the evaluation of complex expressions for predicates or functions. Most importantly, however, it allows for re-use of query plans, as bindings can be simply rebound to another binding context. Plan re-use then simply comes down to copying a plan with all its bindings and assigning it a new binding context afterwards.

With regards to intra-query parallelism, it is important to note that binding contexts are not thread-safe, which is why they are copied upon partitioning of a query plan. Consequently, every branch of an operator tree acts on its own child of the same, root binding context.

6.2.4 Functions and Function Generators

In compliance with the model proposed in Chapter 5, *Cottontail DB* implements a *function registry*, which is part of the catalogue and can be used to obtain function implementations during query binding and planning. The function registry comprises of three parts: The registry itself, *function generators* and *functions*.

The registry is a singleton object that is kept in memory for an instance of *Cottontail DB*. It provides the interface to register function generators and resolve function implementations given a *closed signature*. Function resolution then is facilitated by the function generators, which are again singleton objects per type of function – e.g., Euclidean distance or scalar multiplication – and act as factories for concrete function implementations. A function generator is always associated with an *open signature*, which comprises of a function's name and potentially *open* arguments, which only specify the class (e.g., number, vector) but not the concrete type (e.g., float number, double vector) of arguments. It is then up to the generator to return an implementation based on a closed signature, if such an implementation exists.

Given this facility, function resolution takes place in two steps: First, a closed signature is converted to an open signature to lookup the responsible function generator in a hash map. That generator then produces the concrete implementation, based on the closed signature. This two-step process allows for easy extension, since adding new functions comes down to adding a generator and the type-specific implementations. Furthermore, the mechanism could be adapted to generate code at runtime, which *Cottontail DB* doesn't support.

Equivalences between DFCs and high-dimensional indexes, as required by Definition 5.6, are currently implemented at the index-level, i.e., an index is aware of the type of DFC it can replace.

Table 6.3 Primitive operations offered by the storage engines's Tx abstractions to interact with DBOs including the argument and return types (arg \rightarrow ret).

| Type | Schema | Entity | Index | Column |
|--------------|--------|--------------------------|-------------------|--------------------------------|
| create() | ✓ | ✓ | ✓ | ✗ |
| drop() | ✓ | ✓ | ✓ | ✗ |
| read() | ✗ | TID \rightarrow Record | ✗ | TID \rightarrow Value |
| update() | ✗ | Record | ✗ | TID, Value \rightarrow Value |
| insert() | ✗ | Record \rightarrow TID | ✗ | Value \rightarrow TID |
| delete() | ✗ | TID \rightarrow Record | ✗ | TID \rightarrow Value |
| scan() | ✗ | Cursor < Record > | Cursor < Record > | Cursor < Value > |
| filter() | ✗ | ✗ | Cursor < Record > | ✗ |
| count() | ✗ | ✗ | ✓ | ✓ |
| maxTupleId() | ✗ | ✗ | ✓ | ✓ |
| commit() | ✓ | ✓ | ✓ | ✓ |
| rollback() | ✓ | ✓ | ✓ | ✓ |

6.2.5 Storage

Over time, *Cottontail DB* has seen multiple different layers for low-level data organisation and storage. Each of these layers addressed certain requirements that were valid at the time of conception. To accomodate these different engines, the *storage manager* offers a unified interface to interact with underlying *storage engines*, regardless of the concrete implementation.

The storage manager offers abstractions for Database Object (DBO), namely, Schema, Entity, Index and Column, in the form of interfaces. A instance of a Tx can be obtained through the respective DBO to initiate transactional interaction (i.e., Schema.Tx, Entity.Tx, Index.Tx and Column.Tx). For that, each Tx implementation provides low-level primitives, e.g., Column.Tx offers read(), update(), insert() and delete() for individual Values. A complete list of primitives can be found in Table 6.3.

Even though the storage manager abstraction layer could be used to combine different storage engines within a single instance, this is currently not implemented. This is mainly because replacement of a storage engine was often triggered by a very specific requirement the other engines could not fulfil. On an implementation side, however, *Cottontail DB* also relies on low-level aspects of the engines to ensure atomicity and (sometimes) transaction isolation. It would take some additional engineering to maintain these guarantees accross different storage engines.

6.2.5.1 Storage engine: MapDB

The *MapDB* storage engine is based on version 3.0.8 of a library with the same name⁷ and was the first storage engine for *Cottontail DB*. The main requirement at that time was speed and ease of integration.

At its core, *MapDB* offers persistent versions of the Map interface in Java backed by segmented hash trees. It can thus be regarded as a simple key-value store. *MapDB* uses memory mapped files, which allow for fast I/O, and a generic and extendible serialization engine to read and write arbitrary types from and to disk. Atomicity and durability of transactions is guaranteed by write-ahead logging and in-memory latches mediate concurrent access to the underlying tree structure.

Cottontail DB integrates with *MapDB* by storing every column in a dedicated page-file that maps TID to record value. To do so, it bypasses the official high-level APIs offered by *MapDB* and instead uses internal, lower-level storage mechanism to write records to pages directly. This has proven to be significantly faster than using the higher level, abstractions, especially for linear scans. However, the catalogue and all information about schemata, entities etc. are stored in persistent maps as are some of the data structures required for secondary indexes.

While easy to use in practice and reasonably fast by default, *MapDB* has several disadvantages. Firstly, the use of memory-mapped files, while tempting, is highly discouraged for DBMS as shown by [CLP22]. Secondly, the implementation of memory mapped files provided by the JVM is far from optimal and plagued by issues with resource retention on certain platforms. And finally, *MapDB* does not offer any MVCC, which is why isolation is provided by S2PL.

6.2.5.2 Storage engine: Xodus

The *Xodus* storage engine is based on version 1.3.238 of the open source, transactional and schema less, embedded database for Java and Kotlin with the same name, developed and maintained by JetBrains⁸. As opposed to *MapDB*, *Xodus* does not rely on the use of memory mapped files. Atomicity and durability in *Xodus* is guaranteed through write-ahead logging. A property that is very desirable in the context of dynamic index management described in Chapter 5, is *Xodus*' support for non-blocking reads through the use of MVCC and snapshot isolation, which was one of the reasons to experiment with this engine.

Xodus offers three different levels of abstraction over the same core IO facility:

⁷ See: <https://mapdb.org/>

⁸ See [https://github.com/JetBrains/xodus\(\)](https://github.com/JetBrains/xodus())

Environments encapsulate multiple, named key-value stores that can hold arbitrary keys and values. They allow for transactional reading and modification of data across stores. The interface for stores provides simple primitives such as `read()`, `put()` and `delete()` based on the key as well as `scan()` operations using `Cursors`.

Entity Stores use environments to store more complex entities with attributes and links to other entities. This higher level abstraction allows for more advanced data modelling as well as the formulation of complex queries.

Virtual File Systems use the environments to provide a transactional file system within *Xodus*.

Cottontail DB integrates directly with *Xodus*'s environment API, by having dedicated store for each column within a single environment. In a store, the TID is mapped to the respective value. Furthermore, *Cottontail DB* maintains dedicated stores for all the necessary catalogue information as well as data structures required for indexes.

While *Xodus* offers many advantages for Boolean retrieval and transactional workloads, it is slightly slower than *MapDB* for linear scans of columns containing large values in a read-only setting. This is due to the tree traversal that is involved, which incurs more overhead than scanning *MapDB*'s page files. However, we were able to compensate for this drawback by the application *Snappy*⁹ compression for high-dimensional vectors, leading to even higher throughput for full table scans.

6.2.5.3 Storage engine: HARE

HARE is an experimental storage engine for *Cottontail DB* that so far has not made it to a final version and has thus never been used in a productive setting. The idea of *HARE* was to build the storage engine end-to-end to have more fine-grained control over data access patterns.

HARE sports all components required for persistent data management, including a dedicated *disk manager* and a *buffer pool manager*. Atomicity and durability are provided by maintaining an undo and redo logs on a page level, which is reasonably straightforward to implement, but too slow for write-heavy workloads.

⁹ See <https://github.com/google/snappy/>

Structurally, *HARE* is built around page files, which hold the values for an entry. In interfaces with *Cottontail DB*'s type system through a serialization subsystem not unlike that of *MapDB*. Internal data organisation in *HARE* comes in two forms: Fixed-length columns essentially organise data in a persistent array, which enables addressing of values based on the *TID* by simple arithmetics. This allows for very fast random reads as well as extraordinary scan performance, but comes with many of challenges for transactional workloads, for which solutions have only emerged in highly experimental form.

Variable-length records use a skip-list structure with *index-* and *slotted data pages* for lookup, which is slower for random reads but still reasonably fast for full table scans, since index pages can simply be skipped.

In summary, *HARE* can be seen as a proof-of-concept of building a dedicated storage engine for *Cottontail DB*, which certainly offers a very interesting foundation for future research but falls short of existing engines, in its current form.

Question: Should HARE be mentioned at all?

6.2.6 Transactional Guarantees

The *transaction manager* provides transactional semantics and guarantees in *Cottontail DB*. A transaction formally starts with the creation of a *transaction context* and ends with either a call to `commit()`, `rollback()` or `kill()`, which is a rollback for deadlocked transactions. The transaction context is always attached to the query context. However, unlike the query context, the transaction context may live beyond the scope of a single query. This depends on whether a transaction was *explicitly* started through the respective endpoints, or whether it is an *implicit* transaction, spawned by the issuing of a single query. Transactions of the first type must be explicitly terminated by the caller by calling `commit()` or `rollback()` whereas transactions of the second type end with the execution or abortion of the query that created them.

The transaction context provides access to the DBO's TX instances used for query execution. Since all these accesses are routed through the transaction context, it can take necessary steps to ensure transaction isolation. Irrespective of the guarantees provided by the storage engine, isolation can always be guaranteed through *Cottontail DB*'s *lock manager*, which provides DBO-level locking through the S2PL algorithm. While this is not the most efficient solution, it is applicable even for storage engines that do not provide any means of transaction isolation. In case they do, though, the lock manager can be bypassed.

To guarantee atomicity and durability of transactions, *Cottontail DB* relies on the capabilities of the underlying storage engine. The transaction manager simply assures, that calls to `commit()` or `rollback()` are propagated accordingly. As was mentioned, both properties are assured by some form of write-ahead logging for all storage engines, which is state-of-the-art for current DBMS.

There are only few guarantees w.r.t. to consistency, since *Cottontail DB* currently does not support any advanced constraints on columns. What *Cottontail DB* enforces, though, is that non-null columns are not nulled and that changes to indexes are propagated to the underlying index structures as described in Chapter 5. Failure to fulfill this, will raise errors, which inevitably lead to a rollback of a transaction.

PART IV

Discussion

7

Evaluation

7.1 Interactive Multimedia Retrieval

vitivr's participation to VBS, LSC etc, DRES.

7.2 Adaptive Index Management

Brute force vs. plain index vs. index with auxiliary data structure

7.3 Cost Model

Benchmark effect of cost model in different settings (e.g. based on use cases from chapter 2)

8

Related Work

Publications related to this thesis, i.e., because they may be similar. No foundations!

9

Conclusion & Future Work

Appendix

Bibliography

- [ABC⁺76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976. ISSN: 0362-5915. DOI: 10.1145/320455.320457.
- [BTVG06] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded Up Robust Features. In *European Conference on Computer Vision*, pages 404–417, 2006.
- [BKS⁺90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD '90*, pages 322–331, New York, NY, USA. Association for Computing Machinery, 1990. ISBN: 0-89791-365-5. DOI: 10.1145/93597.98741.
- [Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975. ISSN: 0001-0782. DOI: 10.1145/361002.361007.
- [BS19] David Bernhauer and Tomáš Skopal. Non-metric similarity search using genetic TriGen. In Giuseppe Amato, Claudio Gennaro, Vincent Oria, and Miloš Radovanović, editors, *Similarity Search and Applications*, pages 86–93, Cham. Springer International Publishing, 2019. ISBN: 978-3-030-32047-8.
- [BdB⁺07] Henk M Blanken, Arjen P de Vries, Henk Ernst Blok, and Ling Feng. *Multimedia Retrieval*. Springer, 2007.
- [BGS⁺20] Samuel Börlin, Ralph Gasser, Florian Spiess, and Heiko Schuldt. 3D Model Retrieval Using Constructive Solid Geometry in Virtual Reality. In *Proceedings of the 2020 IEEE International Conference on Artificial Intelligence and Virtual Reality, AIVR 2021*, pages 373–374, 2020.

- [BGS⁺17] Mateusz Budnik, Efrain-Leonardo Gutierrez-Gomez, Bahjat Safadi, Denis Pellerin, and Georges Quénot. Learned Features Versus Engineered Features for Multimedia Indexing. *Multimedia Tools and Applications*, 76(9):11941–11958, 2017.
- [CWW⁺10] Yang Cao, Hai Wang, Changhu Wang, Zhiwei Li, Liqing Zhang, and Lei Zhang. MindFinder: interactive sketch-based image search on millions of images. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, pages 1605–1608, New York, NY, USA. Association for Computing Machinery, 2010. ISBN: 978-1-60558-933-6. DOI: 10.1145/1873951.1874299.
- [Cha12] Donald D. Chamberlin. Early History of SQL. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012. DOI: 10.1109/MAHC.2012.61.
- [Che76] Peter Pin-Shan Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976. ISSN: 0362-5915. DOI: 10.1145/320434.320440.
- [CTW⁺10] Nancy A. Chinchor, James J. Thomas, Pak Chung Wong, Michael G. Christel, and William Ribarsky. Multimedia Analysis + Visual Analytics = Multimedia Analytics. *IEEE Computer Graphics and Applications*, 30(5):52–60, 2010. DOI: 10.1109/MCG.2010.92.
- [CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 1997 International Conference on Very Large Data Bases (VLDB)*, volume 97, pages 426–435, 1997.
- [Cod70] E. F. Codd. Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.
- [CLP22] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are You Sure You Want to Use MMAP in Your Database Management System? In *Conference on Innovative Data Systems Research (CIDR)*, Chaminade, CA, 2022.
- [DT05] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, 886–893 vol. 1, 2005. DOI: 10.1109/CVPR.2005.177.

- [DBS19] Emir Demirel, Baris Bozkurt, and Xavier Serra. Automatic Chord-Scale Recognition using Harmonic Pitch Class Profiles. In *Proceedings of the 16th Sound & Music Computing Conference*, Málaga, Spain. Sound & Music Computing Conference, 2019.
- [DBS⁺11] Oscar Déniz, Gloria Bueno, Jesús Salido, and Fernando De la Torre. Face Recognition using Histograms of Oriented Gradients. *Pattern Recognition Letters*, 32(12):1598–1603, 2011.
- [DM10] Adrien Depeursinge and Henning Müller. Fusion Techniques for Combining Textual and Visual Information Retrieval. In *ImageCLEF: Experimental Evaluation in Visual Information Retrieval*, pages 95–114. January 2010. ISBN: 978-3-642-15180-4. DOI: 10.1007/978-3-642-15181-1_6.
- [DKN08] Thomas Deselaers, Daniel Keysers, and Hermann Ney. Features for Image Retrieval: An Experimental Comparison. *Information Retrieval*, 11(2):77–107, 2008. DOI: 10.1007/s10791-007-9039-3.
- [DA13] T Dharani and I Laurence Aroquiaraj. A Survey on Content Based Image Retrieval. In *Proceedings of the 2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pages 485–490. IEEE, 2013.
- [DZJ⁺12] Shifei Ding, Hong Zhu, Weikuan Jia, and Chunyang Su. A Survey on Feature Extraction for Pattern Recognition. *Artificial Intelligence Review*, 37(3):169–180, March 2012. ISSN: 1573-7462. DOI: 10.1007/s10462-011-9225-y.
- [EZP21] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. High-Dimensional Similarity Search for Scalable Data Science. In *Proceeding of the 37th IEEE International Conference on Data Engineering (ICDE)*, pages 2369–2372, 2021. DOI: 10.1109/ICDE51399.2021.00268.
- [Far16] Javeria Farooq. Object Detection and Identification using SURF and BoW Model. In *Proceedings of the 2016 International Conference on Computing, Electronic and Electrical Engineering (ICE Cube)*, pages 318–323. IEEE, 2016.
- [FTA⁺00] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. Vector Approximation Based Indexing for Non-Uniform High Dimensional Data Sets. In *Proceedings of the Oth*

- International Conference on Information and Knowledge Management*, pages 202–209, 2000.
- [FB74] Raphael A Finkel and Jon Louis Bentley. Quad-Trees a Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.
- [Foo00] Jonathan Foote. Automatic Audio Segmentation Using a Measure of Audio Novelty. In *2000 Ieee International Conference on Multimedia and Expo. Icme2000. Proceedings. Latest Advances in the Fast Changing World of Multimedia (Cat. No. 00th8532)*, volume 1, pages 452–455. IEEE, 2000.
- [GLSJ⁺21] Liliana C Gallegos, Guilian Luchini, Peter C St. John, Seonah Kim, and Robert S Paton. Importance of Engineered and Learned Molecular Representations in Predicting Organic Reactivity, Selectivity, and Chemical Properties. *Accounts of Chemical Research*, 54(4):827–836, 2021.
- [GUW09] Hector Garcia-Molina, Jeffrey D. Ullmann, and Jennifer Widom. *DATABASE SYSTEMS The Complete Book, 2nd Edition*. second edition, 2009. ISBN: 1.
- [GRH⁺20] Ralph Gasser, Luca Rossetto, Silvan Heller, and Heiko Schuldt. Cottontail DB: An Open Source Database System for Multimedia Retrieval and Analysis. In *Proceedings of the 28th ACM International Conference on Multimedia*, ACM MM 2020, pages 4465–4468, 2020.
- [GRS19a] Ralph Gasser, Luca Rossetto, and Heiko Schuldt. Multimodal Multimedia Retrieval with vitivr. In *Proceedings of the 2019 International Conference on Multimedia Retrieval (ICMR 2019)*, ICMR 2019, pages 391–394, New York, NY, USA. Association for Computing Machinery, 2019. ISBN: 978-1-4503-6765-3. DOI: 10.1145/3323873.3326921.
- [GRS19b] Ralph Gasser, Luca Rossetto, and Heiko Schuldt. Towards an all-purpose content-based multimedia information retrieval system. *CoRR*, abs/1902.03878, 2019.
- [GLC⁺95] Asif Ghias, Jonathan Logan, David Chamberlin, and Brian C Smith. Query by Humming: Musical Information Retrieval in an Audio Database. In *Proceedings of the Third ACM International Conference on Multimedia*, pages 231–236, 1995.

- [Gia18] Ivan Giangreco. *Database Support for Large-scale Multimedia Retrieval*. PhD thesis, University of Basel, Switzerland, August 2018.
- [GS16] Ivan Giangreco and Heiko Schuldt. ADAMpro: Database support for big multimedia retrieval. *Datenbank-Spektrum*, 16(1):17–26, 2016. DOI: 10.1007/s13222-015-0209-y.
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [GAR⁺16] Albert Gordo, Jon Almazán, Jerome Revaud, and Diane Larlus. Deep Image Retrieval: Learning Global Representations for Image Search. In *Proceedings of the 2016 European Conference on Computer Vision*, pages 241–257. Springer, 2016.
- [GM93] Goetz Graefe and William J. MacKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218. IEEE, 1993.
- [GHQ95] Ashish Kumar Gupta, Venky Harinarayan, and Dallan Quass. Generalized Projections: A Powerful Approach to Aggregation. In 1995.
- [Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84*, pages 47–57, New York, NY, USA. Association for Computing Machinery, 1984. ISBN: 0-89791-128-8. DOI: 10.1145/602259.602266.
- [HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.
- [HE10] Philippe Hamel and Douglas Eck. Learning Features from Music Audio with Deep Belief Networks. In J. Stephen Downie and Remco C. Veltkamp, editors, *Proceedings of the 11th International Society for Music Information Retrieval (ISMIR 2010)*, pages 339–344, Utrecht, Netherlands. International Society for Music Information Retrieval, 2010.
- [HGI⁺21] Silvan Heller, Ralph Gasser, Cristina Illi, Maurizio Pasquinelli, Loris Sauter, Florian Spiess, and Heiko Schuldt. Towards Explainable Interactive Multi-Modal Video Retrieval with vitivr. In *Proceedings of the 27th International Conference on Multimedia Modeling*,

- MMM 2021, pages 435–440, Cham. Springer International Publishing, 2021. ISBN: 978-3-030-67835-7. DOI: 10 . 1007 / 978 - 3 - 030 - 67835-7_41.
- [HPG⁺20] Silvan Heller, Mahnaz Amiri Parian, Ralph Gasser, Loris Sauter, and Heiko Schuldt. Interactive Lifelog Retrieval with vitrivr. In *Proceedings of the Third ACM Workshop on Lifelog Search Challenge, LSC@ICMR 2020*, pages 1–6, Dublin, Ireland. ACM, 2020. DOI: 10.1145/3379172.3391715.
- [HSS⁺20] Silvan Heller, Loris Sauter, Heiko Schuldt, and Luca Rossetto. Multi-stage Queries and Temporal Scoring in vitrivr. In *2020 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, pages 1–5. IEEE, 2020.
- [HSS⁺19] MD Zakir Hossain, Ferdous Sohel, Mohd Fairuz Shiratuddin, and Hamid Laga. A Comprehensive Survey of Deep Learning for Image Captioning. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.
- [HXL⁺11] Weiming Hu, Nianhua Xie, Li Li, Xianglin Zeng, and Stephen Maybank. A Survey on Visual Content-based Video Indexing and Retrieval. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(6):797–819, 2011.
- [IM98] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.
- [JDS10] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 33(1):117–128, 2010.
- [JWZ⁺16] Björn Þór Jónsson, Marcel Worring, Jan Zahálka, Stevan Rudinac, and Laurent Amsaleg. Ten Research Questions for Scalable Multimedia Analytics. In *International Conference on Multimedia Modeling*, pages 290–302, 2016.
- [KKE⁺10] Daniel Keim, Jörn Kohlhammer, Geoffrey Ellis, and Florian Mansmann. Mastering the information age: Solving problems with visual analytics, 2010.

- [KS04] HyounG-Gook Kim and Thomas Sikora. Comparison of MPEG-7 Audio Spectrum Projection Features and MFCC Applied to Speaker Recognition, Sound Classification and Audio Segmentation. In *Proceedings of the 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)*, pages 925–928, Montreal, Canada. IEEE, 2004. DOI: 10.1109/ICASSP.2004.1327263.
- [KC01] Irena Koprinska and Sergio Carrato. Temporal Video Segmentation: A survey. *Signal processing: Image communication*, 16(5):477–500, 2001.
- [Lee06] Kyogu Lee. Automatic Chord Recognition from Audio using Enhanced Pitch Class Profile. In *Proceedings of the 2006 International Computer Music Conference (ICMC 2006)*, New Orleans, USA. Michigan Publishing, 2006.
- [LÁJ⁺09] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):869–883, 2009. DOI: 10.1109/TPAMI.2008.130.
- [LÁJ⁺18] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. Transactional support for visual instance search. In Stéphane Marchand-Maillet, Yasin N. Silva, and Edgar Chávez, editors, *Similarity Search and Applications*, pages 73–86, Cham. Springer International Publishing, 2018. ISBN: 978-3-030-02224-2.
- [LJA11] Herwig Lejsek, Björn Þór Jónsson, and Laurent Amsaleg. NV-Tree: Nearest neighbors at the billion scale. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval, ICMR '11*, New York, NY, USA. Association for Computing Machinery, 2011. ISBN: 978-1-4503-0336-1. DOI: 10.1145/1991996.1992050.
- [Lev65] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10:707–710, 1965.
- [LCI⁺05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. RankSQL: Query Algebra and Optimization for Relational

- Top-k Queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 131–142, New York, NY, USA. Association for Computing Machinery, 2005. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066173.
- [LKM⁺19] Jakub Lokoč, Gregor Kovalčík, Bernd Münzer, Klaus Schöffmann, Werner Bailer, Ralph Gasser, Stefanos Vrochidis, Phuong Anh Nguyen, Sitapa Rujikietgumjorn, and Kai Uwe Barthel. Interactive Search or Sequential Browsing? A Detailed Analysis of the Video Browser Showdown 2018. *ACM Transactions on Multimedia Computing, Communications, and Applications*. TOMM, 15(1):1–18, 2019.
- [Low99] David G Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the 7th IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999.
- [Lu01] Goujun Lu. Indexing and Retrieval of Audio: A sSurvey. *Multimedia Tools and Applications*, 15(3):269–290, 2001.
- [MB03] Martin F. McKinney and Jeroen Breebaart. Features for Audio and Music Classification. In *Proceedings of the 4th International Conference on Music Information Retrieval (ISMIR 2003)*, Baltimore, USA, 2003.
- [MK18] Y. V. Srinivasa Murthy and Shashidhar G. Koolagudi. Content-based Music Information Retrieval (CB-MIR) and its Applications Toward the Music Industry: A Review. *Acm Computing Surveys*, 51(3), June 2018. ISSN: 0360-0300. DOI: 10.1145/3177849.
- [PSS⁺15] Rasmus Pagh, Francesco Silvestri, Johan Sivertsen, and Matthew Skala. Approximate Furthest Neighbor in High Dimensions. In *International Conference on Similarity Search and Applications*, pages 3–14. Springer, 2015.
- [PW10] Ofir Pele and Michael Werman. The Quadratic-Chi Histogram Distance Family. In *European Conference on Computer Vision*, pages 749–762. Springer, 2010.
- [Pet19] Alex Petrov. *Database Internals*. O'Reilly Media, Inc., 2019. ISBN: 978-1-4920-4034-7.
- [QL01] S. Quackenbush and A. Lindsay. Overview of MPEG-7 Audio. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(6):725–729, 2001. DOI: 10.1109/76.927430.

- [Rob81] John T. Robinson. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 10–18, New York, NY, USA. Association for Computing Machinery, 1981. ISBN: 0-89791-040-0. DOI: 10.1145/582318.582321.
- [Ros18] Luca Rossetto. *Multi-Modal Video Retrieval*. PhD thesis, University of Basel, Switzerland, September 2018.
- [RAPG⁺19] Luca Rossetto, Mahnaz Amiri Parian, Ralph Gasser, Ivan Giangreco, Silvan Heller, and Heiko Schuldt. Deep learning-based concept detection in vitivr. In *Proceedings of the 25th International Conference on Multimedia Modelling*, MMM 2019, pages 616–621, Cham. Springer International Publishing, 2019. ISBN: 978-3-030-05716-9.
- [RGH⁺21] Luca Rossetto, Ralph Gasser, Silvan Heller, Mahnaz Parian-Scherb, Loris Sauter, Florian Spiess, Heiko Schuldt, Ladislav Peska, Tomas Soucek, Miroslav Kratochvil, Frantisek Mejzlik, Patrik Vesely, and Jakub Lokoc. On the User-Centric Comparative Remote Evaluation of Interactive Video Search Systems. *IEEE Multimedia*:1–1, 2021. DOI: 10.1109/MMUL.2021.3066779.
- [RGL⁺20] Luca Rossetto, Ralph Gasser, Jakub Lokoc, Werner Bailer, Klaus Schoeffmann, Bernd Muenzer, Tomas Soucek, Phuong Anh Nguyen, Paolo Bolettieri, Andreas Leibetseder, et al. Interactive Video Retrieval in the Age of Deep Learning - Detailed Evaluation of VBS 2019. *IEEE Transactions on Multimedia*. TOMM, 2020.
- [RGS⁺21] Luca Rossetto, Ralph Gasser, Loris Sauter, Abraham Bernstein, and Heiko Schuldt. A System for Interactive Multimedia Retrieval Evaluations. In *Proceedings of the 27th International Conference on Multimedia Modeling*, MMM 2021, pages 385–390, Cham. Springer International Publishing, 2021. ISBN: 978-3-030-67835-7.
- [RGG⁺18] Luca Rossetto, Ivan Giangreco, Ralph Gasser, and Heiko Schuldt. Competitive Video Retrieval with vitivr. In *Proceedings of the 24th International Conference on Multimedia Modelling*, MMM 2018, pages 403–406, Cham. Springer International Publishing, 2018. ISBN: 978-3-319-73600-6.

- [RGT⁺16] Luca Rossetto, Ivan Giangreco, Claudiu Tanase, and Heiko Schuldt. Vitivr: A Flexible Retrieval Stack Supporting Multiple Query Modes for Searching in Multimedia Collections. In *Proceedings of the 24th ACM International Conference on Multimedia*, pages 1183–1186, 2016.
- [SJ19] Ayodeji Olalekan Salau and Shruti Jain. Feature Extraction: A Survey of the Types, Techniques, Applications. In *Proceedings of the 2019 International Conference on Signal Processing and Communication (ICSC)*, pages 158–164, 2019. DOI: 10.1109/ICSC45622.2019.8938371.
- [SR06] Uri Shaft and Raghu Ramakrishnan. Theory of Nearest Neighbors Indexability. *ACM Transactions on Database Systems*, 31(3):814–838, September 2006. ISSN: 0362-5915. DOI: 10.1145/1166074.1166077.
- [SML19] Tomáš Souček, Jaroslav Moravec, and Jakub Lokoč. TransNet: A Deep Network for Fast Detection of Common Shot Transitions. *arXiv preprint arXiv:1906.03363*, 2019.
- [Spi09] David I Spivak. Simplicial Databases. *arXiv preprint arXiv:0904.2012*, 2009.
- [TYB16] Yi-Hsuan Tsai, Ming-Hsuan Yang, and Michael J Black. Video Segmentation via Object Flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3899–3908, 2016.
- [WZs⁺18] Jingdong Wang, Ting Zhang, jingkuan song, Nicu Sebe, and Heng Tao Shen. A Survey on Learning to Hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):769–790, 2018. DOI: 10.1109/TPAMI.2017.2699960.
- [WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*, volume 98, pages 194–205, New York City, NY, USA. Morgan Kaufmann, 1998.
- [YLZ07] Yubin Yang, Hui Lin, and Yao Zhang. Content-based 3-D Model Retrieval: A Survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(6):1081–1098, 2007.

- [ZW14] Jan Zahálka and Marcel Worring. Towards interactive, intelligent, and integrated multimedia analytics. In *2014 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 3–12, 2014. doi: 10.1109/VAST.2014.7042476.
- [ZAD⁺06] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search The Metric Space Approach*. Springer Science + Business Media, Inc., 2006. ISBN: 978-0-387-29146-8.

Curriculum Vitae

| | |
|---------------|---|
| Name | Ralph Marc Philipp Gasser Brunnenweg 10, 4632 Trimbach |
| Date of Birth | 28.03.1987 |
| Birthplace | Riehen BS, Switzerland |
| Citizenship | Switzerland |

Education

| | |
|-----------------------|---|
| since Oct. 2017 | Ph.D. in Computer Science under the supervision of Prof. Dr. Heiko Schuldt, Databases and Information Systems Teseach Group, University of Basel, Switzerland |
| Sep. 2014 – Jul. 2017 | M.Sc. in Computer Science, University of Basel, Switzerland |
| Sep. 2007 – Jul. 2014 | B.Sc. in Nanoscience, University of Basel, Switzerland |
| Sep. 2002 – Dec. 2006 | Matura, Gymnasium Liestal, Liestal, Switzerland |

Employment

| | |
|-----------------------|---|
| since Oct. 2017 | Research & Teaching Assistant, Databases and Information Systems Research Group, University of Basel, Switzerland |
| since Dec. 2010 | IT Consultant & Partner, pontius software GmbH, Bubendorf, Switzerland |
| Jun. 2008 - Dec. 2011 | Application Manager “imdas pro”, Bildungs-, Kultur- und Sportdirektion des Kantons Basel-Landschaft, Liestal, Switzerland |

Publications

2021

- Luca Rossetto, Ralph Gasser, Loris Sauter, et al. A System for Interactive Multimedia Retrieval Evaluations. In *Proceedings of the 27th International Conference on Multimedia Modeling*, MMM 2021, pages 385–390, Cham. Springer International Publishing, 2021. ISBN: 978-3-030-67835-7
- Silvan Heller, Ralph Gasser, Cristina Illi, et al. Towards Explainable Interactive Multi-Modal Video Retrieval with vitrivr. In *Proceedings of the 27th International Conference on Multimedia Modeling*, MMM 2021, pages 435–440, Cham. Springer International Publishing, 2021. ISBN: 978-3-030-67835-7. DOI: 10.1007/978-3-030-67835-7_41
- Luca Rossetto, Ralph Gasser, Silvan Heller, et al. On the User-Centric Comparative Remote Evaluation of Interactive Video Search Systems. *IEEE Multimedia*:1–1, 2021. DOI: 10.1109/MMUL.2021.3066779

2020

- Ralph Gasser, Luca Rossetto, Silvan Heller, et al. Cottontail DB: An Open Source Database System for Multimedia Retrieval and Analysis. In *Proceedings of the 28th ACM International Conference on Multimedia*, ACM MM 2020, pages 4465–4468, 2020 (Best Open Source System Award)
- Samuel Börlin, Ralph Gasser, Florian Spiess, et al. 3D Model Retrieval Using Constructive Solid Geometry in Virtual Reality. In *Proceedings of the 2020 IEEE International Conference on Artificial Intelligence and Virtual Reality*, AIVR 2021, pages 373–374, 2020
- Luca Rossetto, Ralph Gasser, Jakub Lokoc, et al. Interactive Video Retrieval in the Age of Deep Learning - Detailed Evaluation of VBS 2019. *IEEE Transactions on Multimedia*. TOMM, 2020

2019

- Ralph Gasser, Luca Rossetto, and Heiko Schuldt. Multimodal Multimedia Retrieval with vitrivr. In *Proceedings of the 2019 International Conference on Multimedia Retrieval (ICMR 2019)*, ICMR 2019, pages 391–394, New York, NY, USA. Association for Computing Machinery, 2019. ISBN: 978-1-4503-6765-3. DOI: 10.1145/3323873.3326921 (Best Demo Award)
- Jakub Lokoč, Gregor Kovalčík, Bernd Münzer, et al. Interactive Search or Sequential Browsing? A Detailed Analysis of the Video Browser Show-

down 2018. *ACM Transactions on Multimedia Computing, Communications, and Applications*. TOMM, 15(1):1–18, 2019

- Ralph Gasser, Luca Rossetto, and Heiko Schuldt. Towards an all-purpose content-based multimedia information retrieval system. *CoRR*, abs/1902.03878, 2019
- Luca Rossetto, Mahnaz Amiri Parian, Ralph Gasser, et al. Deep learning-based concept detection in vitivr. In *Proceedings of the 25th International Conference on Multimedia Modelling*, MMM 2019, pages 616–621, Cham. Springer International Publishing, 2019. ISBN: 978-3-030-05716-9

2018

- Luca Rossetto, Ivan Giangreco, Ralph Gasser, et al. Competitive Video Retrieval with vitivr. In *Proceedings of the 24th International Conference on Multimedia Modelling*, MMM 2018, pages 403–406, Cham. Springer International Publishing, 2018. ISBN: 978-3-319-73600-6

Advised Theses

- Renato Farrugio. *Scene Text Recognition in Images and Video with Cineast*. Bachelor’s Thesis, University of Basel
- Florian Mohler. *Online Image Analysis to Identify Politicians in Twitter Images*. Bachelor’s Thesis, University of Basel
- Gabriel Zihlmann. *Magnetic Resonance Fingerprinting Reconstruction using Methods from Multimedia Retrieval*. Master’s Thesis, University of Basel
- Yan Wang. *Detecting Visual Motives Using Online Clustering of Similar Images*. Bachelor’s Thesis, University of Basel
- Samuel Börlin. *3D Model Retrieval using Constructive Solid Geometry in Virtual Reality*. Master’s Thesis, University of Basel
- Manuel Hürbin. *Retrieval Optimization in Magnetic Resonance Fingerprinting*. Bachelor’s Thesis, University of Basel
- Loris Sauter. *Fighting Misinformation: Image forgery detection in social media streams*. Master’s Thesis, University of Basel

Honors and Awards

Best VBS System Award for the winning system of the Video Browser Show-down (VBS) and paper titled *Towards Explainable Interactive Multi-Modal Video Retrieval with vitrivor* at the *27th International Conference on Multimedia Modeling (MMM 2021)*

Best Demo Award for the best demo paper titled *A System for Interactive Multimedia Retrieval Evaluations* at the *27th International Conference on Multimedia Modelling (MMM 2021)*

Best Open Source Award for the best open source system paper titled *Cottontail DB: An Open Source Database System for Multimedia Retrieval and Analysis* at the *28th ACM International Conference on Multimedia (ACM MM 2020)*

Best Demo Award for the best demo paper titled *Multimodal Multimedia Retrieval with vitrivor* at the *ACM International Conference on Multimedia Retrieval (ICMR 2019)*

Best VBS System Award for the winning system of the Video Browser Show-down (VBS) and paper titled *Deep Learning-based Concept Detection in vitrivor* at the *25th International Conference on Multimedia Modeling (MMM 2019)*

Fritz Kutter Award 2017 for the Master's Thesis titled *Towards an All-Purpose, Content-based Multimedia Retrieval System*. The Fritz Kutter trust managed by the ETH Zürich honors the best thesis (diploma, master, doctoral) at a Swiss University every year.

Other Relevant Experience

- Technical Program Committee Member (PC Member) of the *International Conference on Multimedia Modeling (MMM 2022)*
- Technical Program Committee Member (PC Member) of the *International Conference on Multimedia Retrieval (ICMR 2021)*, *Workshop – Lifelog Search Challenge*
- Technical Program Committee Member (PC Member) of *ViRaL 2021*

Ask Luca what exactly this

-
- Technical Program Committee Member (PC Member) of the *ACM International Conference on Multimedia (ACM MM 2020)*

Declaration on Scientific Integrity

includes Declaration on Plagiarism and Fraud

Author

Ralph Marc Philipp Gasser

Matriculation Number

2007-050-131

Title of Work

Data Management for Dynamic Multimedia Analytics and Retrieval

PhD Subject

Computer Sciences

Declaration

I hereby declare that this doctoral dissertation "*Data Management for Dynamic Multimedia Analytics and Retrieval*" has been completed only with the assistance mentioned herein and that it has not been submitted for award to any other university nor to any other faculty at the University of Basel.

Basel, DD.MM.YYYY

Signature

Hand-in separately