# 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

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| 1NF | First Normal Form |
| 2NF | Second Normal Form |
| 3NF | Third Normal Form |
| ACID | Atomicity, Consistency, Isolation, Durability |
| ANNS | Approximate Nearest Neighbour Search |
| API | Application Programming Interface |
| ARIES | Algorithm for Recovery and Isolation Exploiting Semantics |
| AST | Abstract Syntax Tree |
| AVX | Advanced Vector Extensions |
| BASE | Basically Available, Soft state, Eventually consistent |
| BCNF | Boyce–Codd Normal Form |
| CLI | Call Level Interface |
| CNN | Convolutional Neural Network |
| CP | Cluster Pruning |
| CPU | Central Processing Unit |
| CRUD | Create, Read, Update, Delete |
| CWA | Closed-World Assumption |
| DBMS | Database Management System |
| DBO | Database Object |
| DCG | Discounted Cumulative Gain |
| DDL | Data Definition Language |
| DFC | Distance Function Class |
| DML | Data Manipulation Language |
| DQL | Data Query Language |
| ERM | Entity Relationship Model |
| FAISS | Facebook AI Similarity Search |
| FIFO | First In First Out |

| | |
|---|---|
| FK | Foreign Key |
| FNS | Farthest Neighbour Search |
| FPGA | Field-programmable Gate Array |
| GPU | Graphics Processing Unit |
| HNSW | Hierarchical Navigable Small World |
| HOG | Histogram of Oriented Gradients |
| JDBC | Java Database Connectivity |
| JVM | Java Virtual Machine |
| kFN | k-Farthest Neighbour Search |
| KLT | Karhunen-Loève Transform |
| kNN | k-Nearest Neighbour Search |
| LIRe | Lucene Image Retrieval |
| LRU | Least Recently Used |
| LSC | Lifelog Search Challenge |
| LSH | Locality Sensitive Hashing |
| MAP | Mean Average Precision |
| MFCC | Mel-frequency Cepstral Coefficient |
| MIPS | Maximum Inner Product Search |
| MRF | Magnetic Resonance Fingerprinting |
| MRI | Magnetic Resonance Imaging |
| MVCC | Multi-Version Concurrency Control |
| NGT | Neighborhood Graph and Tree |
| NMR | Nucleic Magnetic Resonance |
| NNS | Nearest Neighbour Search |
| NUMA | Non-Uniform Memory Access |
| ODBC | Open Database Connectivity |
| OLTP | Online Transaction Processing |
| PCP | Pitch Class Profiles |

| | |
|---|---|
| PK | Primary Key |
| POSIX | Portable Operating System Interface |
| PQ | Product Quantisation |
| RAID | Redundant Array of Independent Disks |
| RAM | Random Access Memory |
| RDF | Resource Description Framework |
| RNNS | Reverse Nearest Neighbour Search |
| S2PL | Strict 2-Phase Locking |
| SIFT | Scale Invariant Feature Transformation |
| SIMD | Single Instruction Multiple Data |
| SOM | Self-organizing Map |
| SQL | Structured Query Language |
| SSD | Solid State Drive |
| SSE | Streaming SIMD Extensions |
| SURF | Speeded-Up Robust Features |
| UDF | Used Defined Function |
| VAF | Vector Approximation Files |
| VBS | Video Browser Showdown |

# List of Symbols

### Basics

| | |
|---|---|
| $\mathbb{N}$ | Set of natural numbers. |
| $\mathbb{R}$ | Set of real numbers. |
| $\mathbb{C}$ | Set of all complex numbers. |
| $\wedge$ | Logical conjunction. |
| $\vee$ | Logical, inclusive disjunction. |
| $\neg$ | Logical negation. |

### Relational Algebra

| | |
|---|---|
| $\mathcal{R}$ | A relation. Sometimes with subscript, e.g., $\mathcal{R}_{\texttt{name}}$ 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., $\mathcal{R}^O{}_{\texttt{name}}$ to specify name or index. |
| $\mathcal{D}$ | A data domain of a relation. Sometimes with subscript, e.g., $\mathcal{D}_{\texttt{name}}$, to specify name or index. |
| $\mathcal{A}$ | An attribute of a relation, i.e., a tuple $(\texttt{name}, \mathcal{D})$. 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. |
| $\texttt{SCH}$ | The schema of a relation $\mathcal{R}$, i.e., all attributes $\texttt{SCH}(\mathcal{R}) = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_N)$ that make up $\mathcal{R}$. |
| $\sigma$ | The unary selection operator used to filter a relation $\mathcal{R}$. Usually printed with a subscript to describe the boolean predicate $S$, e.g. $\sigma_S$. |

| | |
|---|---|
| $\pi$ | The unary projection operator used to restrict the attributes $\mathcal{A}$ of a relation $\mathcal{R}$. Usually used with subscript to list the projection attributes $P$, e.g., $\pi_{\mathcal{A}_1,\mathcal{A}_2}$. |
| $\rho$ | The unary rename operator used to rename the attributes $\mathcal{A}$ of a relation $\mathcal{R}$. Usually used with subscript to list the rename operations, e.g., $\rho_{\mathcal{A}_1 \to \mathcal{A}_2}$. |
| $\omega$ | The binary sort operator used to construct a ranked relation $\mathcal{R}^O$. Usually printed with a subscript to list order attributes $O$, e.g., $\omega_{\mathcal{A}_1\uparrow,\mathcal{A}_2\downarrow}$. |
| $\lambda$ | The unary k-selection operator restricts $\mathcal{R}$ to the first $k$ tuples. Usually printed with a subscript to define $k$, e.g., $\lambda_k$. |
| $\gamma$ | The unary group operator aggregates tuples in $\mathcal{R}$. Usually printed with a subscript to list the group attributes $G$, e.g., $\gamma_G$. |

### Multimedia Retrieval

| | |
|---|---|
| $\mathbb{O}$ | The media collection domain $\mathbb{O} = \{o_1, o_2, \ldots, o_N\}$. |
| $\mathbb{F}$ | The feature domain $\mathbb{F} = \{f_1, f_2, \ldots f_M\}$. |
| $\mathfrak{t}$ | Feature transformation function $\mathfrak{t}: \mathbb{O} \to \mathbb{F}$ |
| $\mathfrak{d}$ | Dissimilarity or distance function $\mathfrak{d}: \mathbb{F} \times \mathbb{F} \to \mathbb{R}$. |
| $\mathfrak{s}$ | Similarity function $\mathfrak{s}: \mathbb{F} \times \mathbb{F} \to [0, 1.]$. |
| $\mathfrak{c}$ | A correspondence function $\mathfrak{c}: \mathbb{R} \to [0, 1]$. |

### System Model

| | |
|---|---|
| $\hat{\mathfrak{d}}$ | A Distance Function Class (DFC) $\hat{\mathfrak{d}}: \mathcal{D}_q \times \mathcal{D}_q \times \mathcal{D}_1 \cdot \times \mathcal{D}_n \to \mathbb{N}$ |
| $\mathcal{I}$ | An index on relation $\mathcal{R}$ indexing attributes $I \subset \mathtt{SCH}(\mathcal{R})$. |
| $\mathtt{OP}$ | A (relational) database operation, typically acting on a relation $\mathcal{R}$. |
| $\mathcal{P}$ | A query execution plan, i.e., a concatenation of $N$operators $\mathtt{OP}_1 \circ \mathtt{OP}_2, \ldots \circ \mathtt{OP}_N$. |
| $\mathcal{T}$ | A transaction, which is used to executed multiple queries. Subscript is typically used to indicate a point in time, e.g., $\mathcal{T}_1$. |

| | |
|---|---|
| $\mathcal{S}$ | A consistent snapshot of the database and all objects it contains (e.g., relations and indexes). Subscript is typically used to indicate a point in time, e.g., $\mathcal{S}_1$. |
| $\texttt{OP}_c$ | A change operation $\texttt{OP}_c(\mathcal{R}, \cdot)$ that can either be an insert, update or delete. |
| $C$ | A change on relation $\mathcal{R}$ with $C = (\texttt{OP}_c, \mathcal{R}, t, t')$. |
| `WRITEM` | A write model function that determines if a change $C$ can be proagated to an index $\mathcal{I}$. |
| `FAILM` | A failure model function that adjusts the state of an index $\mathcal{I}$ for a change that occurred $C$. |
| `REBUILD` | A rebuild function that can rebuild the $\mathcal{I}$ based on primary data in $\mathcal{R}$ . |

Part I

# Introduction

*At first I was afraid, I was petrified...*

— Gloria Gaynor, 1978

# 1 Introduction

The term *multimedia* describes the combination of different forms of content – sometimes called *medium* or *modality* – into a single, sensory experience for the purpose of observation and interpretation. Those content formats include but are not limited to images and videos (visual), music, sound effects and speech (aural) or textual information. However, exotic content forms such signals produced by sensors can also be seen as modalities, even though experience by a 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 photo collection on their computer. (Multi-)media content makes up a large part of today's Internet and constitutes a major driving force behind its growth, as both its 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 (so called "prosumers" [RJ10; RDJ12]). 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 users[1]. Facebook alone contributed to 144 thousand uploaded images per minute in 2020. And many more of these platforms, such as Instagram, Twitter or TikTok, serve millions of users with mixed, self-made content involving text, images, videos or a combination thereof. A study found, that by 2025, we are bound to produce an estimated, annual amount of 175 ZB (i.e, $175 \times 10^{21}$ B) worth of data[2] a large part of which can be considered multimedia, which must be *managed*, *manipulated*, *searched*, *explored* and *analysed* in an efficient and effective manner.

[1] Source: Statista.com, "Social media usage worldwide", January 2021
[2] Source: Statista.com, "Big Data", January 2021

## 1.1 Working with Multimedia Data

At 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* and *metdata*. Unlike traditional data collections that contain mainly (semi-)structured information such as text or numbers, the content of the multimedia item itself is unstructured at a data level, which is why *feature representations* or *descriptors* 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 real-valued vectors. However, in theory, any mathematical object that can be processed by a computer can act as such a descriptor.

Multimedia analysis – which has its roots in *computer vision* and *pattern recognition* and started in the early 1960s – deals with the automated, computer-aided analysis of media data, i.e., the extraction and processing of feature representations. In the early days of computer vision, for example, a lot of effort went into the engineering of feature representations that captured certain aspects of an image's content, such as the colour distribution, texture or relevant keypoints [Low99; BTVG06]. Once such features have been obtained, they could 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 deep 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 including speech recognition, audio fingerprinting in music (re-)identification, movement detection in videos or classification of 3D models, all of which fall into the broader category of multimedia analysis.

### 1.1.1 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, wheras 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 a collection.

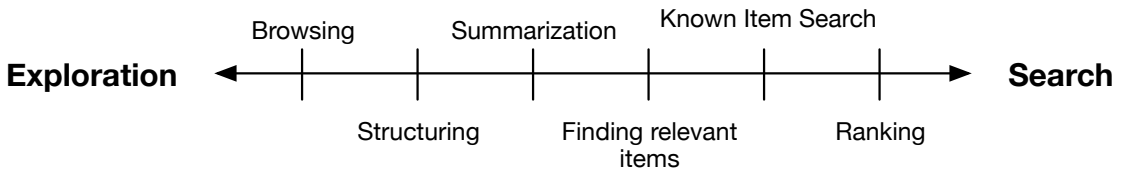



**Figure 1.1 Exploration-search axis of multimedia analytics [ZW14].**

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 a multimedia analytics task, a user's activities often 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.

### 1.1.2 Multimedia Retrieval

Traditionally, multimedia 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 large bodies of (multi-)media data, based on information that may not necessarily be available in a structured manner, such as, motives depicted in an image. Even though search and retrieval may sound like the main function offered by 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, e.g., a relational database and languages like SQL, a user can specify exactly what elements from the database should be selected using predicates that lead to a binary decision whether items in a collection match or not. For example, when considering a product database that contains price information for individual items on sale, 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* [BdB+07; Ros18] associated with them. A very popular model to work with the feature representation in multimedia retrieval 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 [ZAD+06] and it is inherently non-binary in nature, i.e., an item matches a query to some degree and may be among the results even though the score indicates that it is a bad match. 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 [HXL+11; DA13; MK18] as have been different types of query formulation, that complement traditional, text-based input. These models include methods such as *Query-by-Example* [KCH95], *Query-by-Sketch* [SMM99], *Query-by-Humming* [GLC+95] or *Query-by-Sculpting* [BGS+20].

It is worth noting, that when considering concrete, "real-world" multimedia retrieval system implementations today, the lines between traditional multimedia retrieval and multimedia analytics quickly start to blur. This is because traditional multimedia retrieval operates under the assumption that a specific retrieval model produces the correct result in the top ranks (e.g., the top 5) and that an item can thus be retrieved by a single query. In practice, this is often not the case due to limitations of the model itself, imprecissions in the query or simply because an information need may be very general rather than specific. Therefore, finding an item of interest often requires multiple queries and other types of interaction with the data collection. This is sometimes refered to interactive multimedia retrieval and it usually includes a user in the loop.

Therefore, in addition to the extraction of appropriate features and the conception of effective ranking algorithms, interactive multimedia retrieval systems also concern themselves with aspects such as query (re-)formulation and refinement, results presentation and efficient exploration [LKM+19]. Moreover, such systems do not simply operate on features representations anymore but combine similarity and Boolean search [RGL+20]. Consequently, one could argue that interactive multimedia retrieval systems perform a 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 interactive multimedia retrieval as well.

### 1.1.3 Multimedia Data Management

As (multi-)media collections become large enough for the relevant units of information to no longer fit into main memory (RAM), multimedia retrieval and analytics quickly becomes an issue of scalable data management [JWZ+16; PYC+18]. Therefore, it is no longer just a question of effective feature engineering and ranking but also of strategies to store and index the original data as well as derived descriptors in a way that allows for efficient and effective processing in the face of interactive users, changing query workloads and changing data [SWS+00]. This data management aspect brings about many challenges that do, however, bear some similarity with problems that have already been addressed in database research.

However, as of today "Multimedia analytics state of the art [...] has up to now [...] not explicitly considered the issue of data management, despite aiming for large-scale analytics." [JWZ+16] (p. 296), a statement that can also be made about multimedia retrieval. In contrast, the database research community has identified support for unstructured- and multimedia data as well as the integration of information retrieval and database functionality as important research directions in both the *Lowell* and the *Claremont* reports on the state of database research in 2005 and 2008 respectively [AAB+05; AAB+08]. Despite these initiatives, however, the two fields have remained largely disjoint areas of reasearch until today, with only a few, isolated contributions.

## 1.2 Research Gap and Objective

Despite some theoretical work towards multimedia data management [MS96; AN97], the conception of concrete multimedia database systems [GS16; YLF+20; WYG+21] as well as multimedia retrieval systems that refactor data management into distinct components [CHS+95; RGT+16; GRS19a], there is still a gaping disconnect between multimeda retrieval, analytics and database research. While [Gia18] proposes 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 changing data, are not systematically considered. Contrarily, the proposed models – despite being seminal for data management in multimedia retrieval applications – postulate assumptions, that have considerable impact on the practical applicability of 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: Nearest Neighbour Search** The metric space model employed in multimedia retrieval [ZAD+06] relies on a notion of similarity search that is usually expressed as finding the $k$ nearest neighboring feature vectors $f_{i\in[1,k]} \in \mathbb{F}$ to a query vector $q \in \mathbb{R}^d$ in a collection $\mathbb{F} \subset \mathbb{R}^d$ given a certain distance function. A common strategy to accomodate this type of search is to add this operation as a new query primitive in addition to whatever an existing data model might already bring [GdR+09; GS16; YLF+20] without considering the interplay between this primitive and other database operations systematically.

**Assumption 2: Staticity of Data Collections** Multimedia retrieval systems today often 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 take place. Usually, no changes to the data collection are allowed during the online phase. This model is, for example, advertised by [FSN+95; KCG+03; Gia18; Ros18] and to the best of our knowledge, most existing multimedia retrieval and analytics systems implement this either explicitly or implicitly with a few notable exceptions such as the system presented in [WYG+21].

**Assumption 3: User Driven Query Execution** Database systems usually evaluate and select the execution plan for an incoming query during a step that is refered to as *query planning*. The underlying assumption is that the database 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 contrast, this is often not the case for multimedia retrieval systems and instead if falls to the user to make decisions as to how a query should be executed, e.g., in terms of indexes that should be employed.

On the issue of Assumption 1, one can say that while the described model may be very concise and simple to implement, it merely allows for the ranking of results and is therefore only able to accommodate the search-end of the *exploration-search axis*, assuming that features are, in fact, real-valued vectors. However, the model does not generalise to other operations that involve distance calculation and becomes too limited for tasks such as browsing, structuring and summarization, thus delegating the required data processing to upper-tier system components. Refering to [JWZ+16], it would however be desirable to offer such primitives at a database level. Furthermore, the metric space model with its Nearest Neighbour Search (NNS) paradigm may be too limited for certain use-cases and a broader range of operations therefore be preferable.

It is also worth noting, that Assumption 2 and 3 both go against well established design principles usually found it modern database systems [Pet19; Ams14]. While it may be convenient from a perspective of system design, to assume a data collection to be static – because we don't have to deal with transaction isolation – this is always never the case and such a mode of operation is utterly limiting when considering data that is subject to change, e.g., when performing online analytics or when working with applications that offer CRUD support. A similar argument can be made for the manual selection of query execution paths. Even though it may be simplifying the process of query planning, such a design decision (falsely) assumes, that a user is always a technical expert and that all the potential query workloads are known in advance. As a consequence, it limits the optimisation options available to the system.

On a more general point, we argue that one cannot assume that all the query workloads a multimedia database management system must handle can be anticipated at build time, especially if a session involves a user in the loop. This was emphasised by Smeulders at al. in 2000 already, who stated that "The interacting user brings about many new challenges for the response time of the system. Content-based image retrieval is only scalable to large data sets when the database is able to anticipate what interactive queries will be made. A frequent assumption is that the image set, the features, and the similarity function are known in advance. In a truly interactive session, the assumptions are no longer valid. A change from static to dynamic indexing is required." [SWS+00] (p. 1369). This statement is still of high, practical relevance and summarises what we have expressed in the three assumptions. It can thus be regarded as yet another argument to consider multimedia data management as an important and interesting research challenge.

## 1.3 Contribution and Outline

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

- We propose a *generalised model for similarity search* as an extension to the relational data model and explore implications of such a model on aspects, such as, optimisation of query execution time and query planning.
- We examine the impact of challenging the *data staticity assumption* on index structures commonly used for similarity search and describe an *high-dimensional index management* model by which a multimedia database system can cope with and reason about changing index structures.
- We propose a *cost-model that factors-in accuracy* of generated results in addition to common performance metrics, 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 introduce the multimedia database *Cottontail DB* [GRH+20] that implements the aforementioned models and present an evaluation thereof.

This thesis is presented in four parts, each of which consists of several chapters that incrementally build on one another.

**Part I** provides an introcution and problem statement (Chapter 1) and describes motivating use cases and the requirements they bring (Chapter 2).

**Part II** gives a brief summary of the theoretical foundation and state of the art in database (Chapter 3) and multimedia analysis & retrieval (Chapter 4) research and tries to combine the two aspects on a systems level in a brief survey of multimedia database systems (Chapter 5). In addition to establishing a common foundation and framework for the remainder of this thesis, we also use this part to survey related work.

**Part III** formalises the theoretical models that make up the aforementioned contributions (Chapter 6) and introduces and describes our reference implementation Cottontail DB (Chapter 7).

**Part IV** presents and discusses the evaluation of aforementioned contributions in Cottontail DB (Chapter 8) and concludes (Chapter 9).

*If your only tool is a hammer then every problem looks like a nail.*

— Abraham Maslow, 1966

# 2

# Applications and Use Cases

A fascinating aspect of computer science and information technology is, that it has applications in every conceivable domain. It is this combination of purely technological aspects – the computational part – and the domain knowledge, that make the field challenging as well as interesting. Digital transformation – the act of improving businesses and processing using digital technology [Via19] – has been a driving force behind economic activity over the past decades, and while many non-technological factors define digital transformation, optimisation through the application of dispruptive technologies remains the main incentive behind it.

Consequently, we do not want to detach the work presented in this thesis from its concrete applications and instead aim to motivate it based on real-world scenarios. In the sections that follow, we will therefore be introducing three use cases, for which we believe that a general-purpose multimedia database system would fulfill an important function. From these use cases, we then go on to derive a series of requirements for such a multimedia database.

## 2.1 Use case 1: Interactive Multimedia Retrieval

Multimedia retrieval in a broader sense describes the act of finding items of interest in large multimedia collections, wherein "multimedia" can refer to any type of modality, such as text, images, videos or audio and any combination thereof. Consequently, a multimedia retrieval system must be able to have a user express their *information need*, which is translated to a *query* that the system can use to produce the item(s) of interest as a *result*. The process is visualized in Figure 2.1 and a formal problem definition will be provided in Chapter 4.

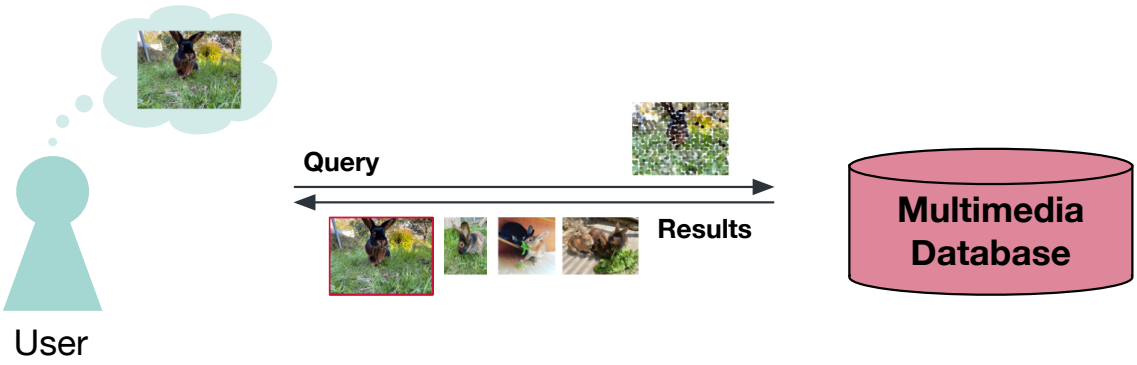



**Figure 2.1 The idealised multimedia retrieval problem: A user expresses an information need as a query and the multimedia retrieval system returns matching items from a collection as a result.**

There are many potential applications of the multimedia retrieval problem, and to name a few examples, one could consider libraries of media items produced by radio or TV stations [WOK+98], collections of cultural heritage data as maintained by museums, archives and archaeology departments [Tsa07] or medical image databases containing X-rays or MRIs [MMB+04] that must somehow be made searchable.

What makes multimedia retrieval a challenging problem is the inherently unstructured nature of the data, by which we mean that the raw, (digital) information may not directly reflect the content of the item as a human consumer might perceive or describe it. Let us take, for example, the image of a rabbit the user in Figure 2.1 is looking for[1]. If our task were to describe that image so that we can retrieve it from a large database, we might use terms like "rabbit", "bunny" or "grass". We might assume, judging from the image, that the scene is taking place in a "garden" and maybe we notice that the rabbit is currently "munching" on a "leaf" or "strain of grass". We might even know the rabbit's breed or name, if we happened to be its owner.

All of this information is formed based on a series of steps ranging from perception, over interpretation to cognition as illustrated in Figure 2.2 and each of these steps is accompanied by a loss or distortion of information [JLX21; Ros18], which makes this process highly subjective, giving rise to a series of "gaps" that influence the outcome. Most importantly, however, all of the aforementioned descriptions are not explicitly contained in the raw image data, which is merely an array of colour values that constitutes the image and does neither come prelabeled nor pre-described.

[1] On purpose, we do not consider the aspect of reconstructing the image from memory, which makes the problem even more challenging.

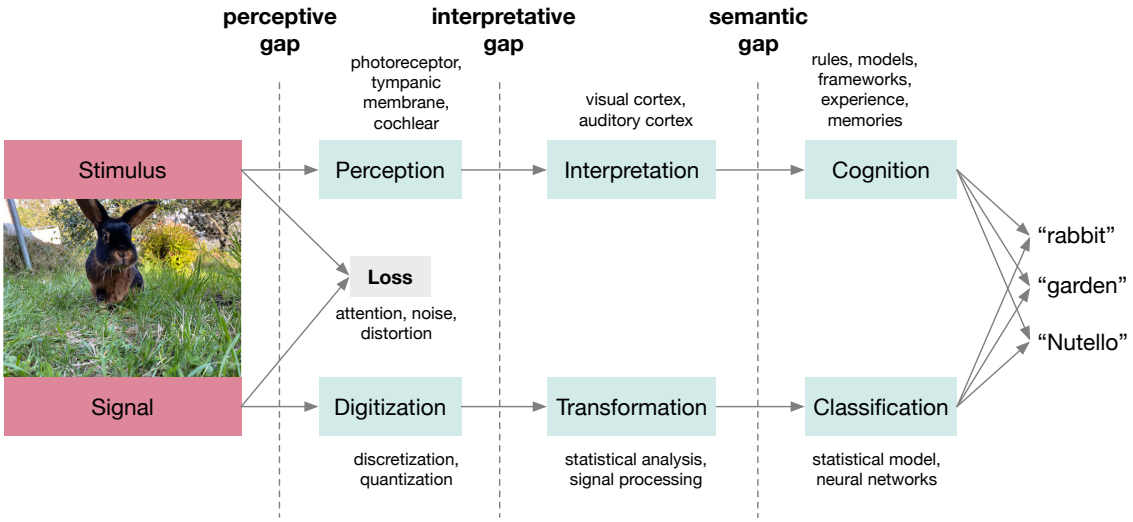



**Figure 2.2 A simplified model of knowledge formation from an analogue input in humans (adapted from [JLX21]) and computers.**

This impedance mismatch between the content being stored and the information required to find it, is the main challenge that unstructured data brings with it, as opposed to structured data, which consists of retrievable units of information by its very nature. Bridging the gaps between the information available and the information one might use to actually search for an item of interest is therefore one of the obstacles to overcome in multimedia retrieval. Over the years, different solutions have emerged:

**Manual Labeling** The obvious solution is to manually embed the aforementioned labels and descriptions into the multimedia item so that we can use this information to find it at a later point in time. While as of today, this approach is still widely used, it comes with two important disadvantages. Obviously, it remains a subjective task, because as we have argued, the process of forming these labels may yield different results depending on who is assigning them (due to the perceptive, interpretative and semantic gap). Most importantly, though, the task of manual label assignment is laborious and unable to scale with the large volumes and the ever increasing velocity at which multimedia data is produced. This is related to the general challenge of working with Big Data [KUG14].

**Automatic Labeling** Recent advances in machine learning have provided us with the ability to automatically label [RDG+16] and describe [RKH+21] different types of multimedia through classification. So instead of annotating the content ourselves, we can leave this to pre-trained models that have, at a high level, a similar mode of operation as a human operator would have, as

is illustrated in Figure 2.2. While this solves the problem of scalability, it still remains a subjective task since the process of model training is also prone to the aforementioned gaps found in the data being trained, which can lead to biases [BK17]. Furthermore, there may be discrepancies between the (mental) model used for classification and the one employed during query formulation, which are typically caried out at different times by different people.

**Content-Based Search** The last approach and the one that "classical" multimedia retrieval has concerned itself with for decades is that of *content-based search*. Instead of using the higher-level concepts in the form of textual labels, content-based search works with intermediate representations called *descriptors* that are derived from the data directly using signal processing or statistical analysis. These descriptors – which are often real-valued vectors [ZAD+06] – can then be used to establish a notion of *similarity* between items in a collection and a query. Instead of a list of exact matches, this type of query returns a ranked list of items that match the query well enough. This is called similarity search [BdB+07] and in this context, the query may refer to more than simply textual input. Given the example in Figure 2.1, one could use another image of a rabbit (Query-by-Example, [KCH95]) or try to create a sketch (Query-by-Sketch, [SMM99]). One could even come-up with more sophisticated types of queries, such as, searching for poses held by people in an image or video [HAG+22] or using visual-text co-embeddings [RKH+21; SGH+22] to map textual descriptions to images and vice-versa.

Research has shown that while individual methods falling into any of the three categories may yield acceptable or even exceptional results, there is clear indication that ultimately, a combination of the three is the most powerful solution, especially, since the exact type of information need cannot always be anticipated [RGL+20]. Furthermore, and partly because different techniques may perform better or worse depending on the information need, we often find that instead of the simple "issue a query, get a result" scheme illustrated in Figure 2.1, actual multimedia retrieval applications are more interactive and require a back and forth between system and user [LBB+22] that involves query (re-)formulation potentially using different descriptors, exploration, examination, query refinement and relevance feedback [LKM+19; GSJ+19] as illustrated in Figure 2.3. These types of interactive multimedia retrieval systems are explicitly tested at evaluation campaigns such as the Video Browser Showdown (VBS) [Sch19] or the Lifelog Search Challenge (LSC) [GJS+21].

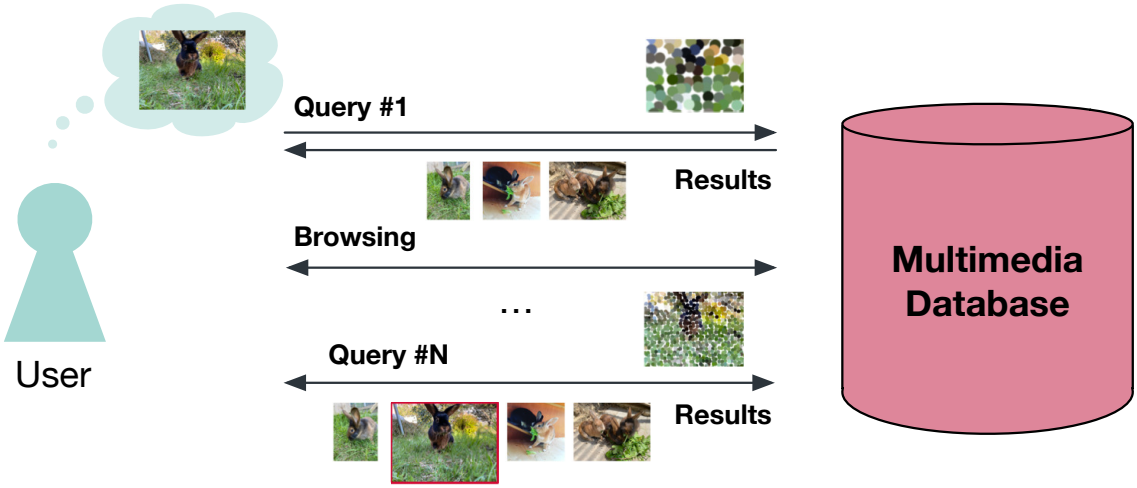



**Figure 2.3 The interactive multimedia retrieval problem: A continuous back and forth between system and user involving different types of workload as the user tries to find the item of interest through query (re-)formulation, browsing and examination.**

Both aspects have important implications for the aforementioned data management system that must support (interactive) multimedia retrieval:

Firstly, such a system must consider different data- and query models when dealing with information from the aforementioned categories as well as a wide range of query workloads. While searching for labels and textual descriptions at scale can be achieved by the well-established *Boolean retrieval* model, we require the completely different *vector space retrieval* model when dealing with descriptors derived from multimedia data and, ideally, we should have the ability to combine the two [HSS+20]. This is a problem that has been acknowledged by other authors before [JWZ+16] and that has been partially addressed, for example, by [Gia18; GS16; WYG+21].

Secondly, any database management system supporting interactive multimedia retrieval must be able to satisfy the different types of query workloads in a reasonable amount of time, so that the user in the loop must not wait for results too long. One can even state that retrieval speed is more important than perfect accuracy, especially in timeboxed settings such as VBS or LSC, because (i) the retrieval process itself is inherently inaccurate due to the aforementioned gaps, and (ii) there is a user in the loop who examines the results and makes the final decisions.

A final aspect to consider is the staticity of the data itself. In multimedia retrieval, we often implicitly assume that data collections remain static while they are being queried but it must be emphasized that for most use cases, this is very likely not to be the case.

## 2.2 Use case 2: Analysis of Social Media Streams

Social media platforms such as Twitter, Facebook, Instagram and TikTok have gained enormous traction over that past few decades. 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 users[2]. Obviously, this is an enormous market that can be used to place advertisement, reach customers or potential electors and to observe consumer reaction to products and services.

Consequently, social media constitutes an important communcation channel for companies and politicians [BZ18]. The field of social media analytics aims "to derive actionable information from social media" [ZCL+10] (p. 15) for economic, political or scientific purposes. Probably one of the most prominent research challenges involves the identification and countering of "fake news" [LBB+18], by which we mean information that comes disguised as authentic but contains fabricated, misleading or even wrong information. The term became highly polarised in the 2016 U.S. elections [QFB+19] – when orchestrated campaigns on social media were targeted at voters and may have tipped the scales – and it has remained a topic high-up on the agenda ever since [FCC+20].

While the problem of fake news can arguably not be solved by mere technological means, the issue has brought about many different research questions in the realm of social media analysis and analytics with potential applications in different domains. One focus lies on the direct detection of misinformation on different channels by different means such as content, propagation patterns [ZZ20] or sources. Others try to tackle the detection of bots [DVF+16; Cre20], user-accounts that are not backed by real humans and that are often used to create or spread misinformation. And then there is the topic of sentiment analysis [YCL+19], which can yield important clues about how (fake-) news stories are received by their consumers.

While many of the aforementioned problems mainly deal with textual information, multimedia obviously plays an important role as well. Instagram and TikTok, for example, focus solely on the sharing of images and videos respectively. A very recent and preliminary analysis suggests [Ciu22], that the sharing of images and videos plays an important role in the international covering of Russia's war on Ukraine. Potential problems in need of solution could, for example, be the automatic detection of image forgeries [Far09] or analysis of the propagation patterns of images [ZCB+18].

[2] Source: Statista.com, "Social media usage worldwide", January 2021

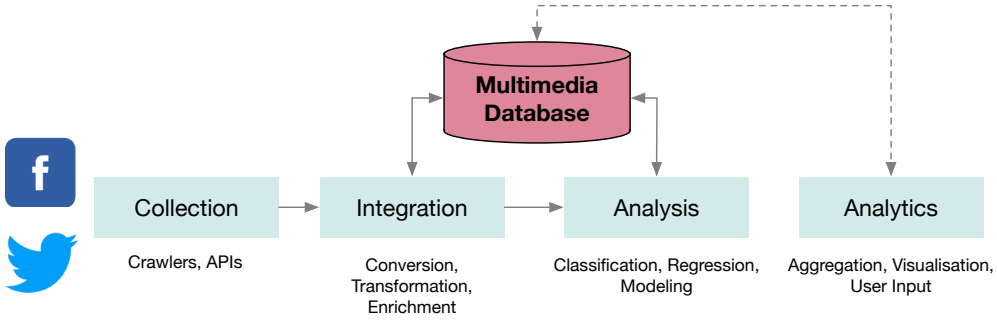



**Figure 2.4 The architecture of a fictious social media analytics platform.**

If we examine what a fictious and generalised pipleine for social media analysis and analytics would look like, we might arrive at something as illustrated in Figure 2.4 [ZCW+15; CSW+19; YPM+19; BPS+20]. The many different channels in existence, require a broad range of tools that deal with data collection using crawlers or APIs provided by the respective source. Since all these potential sources are likely to use very different types of data models, some sort of integration into a common framework is very likely. This step may also be accompanied by data enrichtment using external services or data existing in the system's own collection. This is then followed by what we call analysis, which may include a wide range of machine learning and data analysis techniques that may or may not rely on existing data. A common theme for this step is that, similarly to multimedia retrieval, it probably involves generation and storage of formal descriptors of the multimedia content. Therefore, the entire pipeline is very likely to be backed by some type of persistent data store, which contains labels, descriptors and metadata and which allows queries by both the analysis and the analytics components built on top of it. From the described architecture we can, again, derive certain needs with respect to that underlying database:

Firstly, for this use case, the multimedia database must be able to deal with non-static data, since the various sources potentially generate a steady flow of new information which is likely to be contributing to future inquiries.

Secondly, and similarly to the multimedia retrieval problem, the pipline is likely to deal with structured, semi-structured and unstructured information that must be stored and processed jointly and the multimedia database must be able to cope with this variety.

And finally, with social media being a very classical field for applying multimedia analytics techniques [PYC+18; JWZ+16], the different tools built upon the multimedia database are likely to rely on a wide range of query workloads, which the database should be able to accomodate.

## 2.3 Use case 3: Signal Analysis in MRF

Multimedia retrieval for medical applications has been an important area of research for many decades now [MU17; MMB+04] and many of the arguments that we have made in Section 2.1 can be repeated for this specialised sub-domain.

However, in this section, we focus on a very specific application, in which we do not consider media in the classical sense but in a broader sense of raw signals stemming from a Magnetic Resonance Imaging (MRI) device. One can also refer to this as "science data" [SBZ+13]. MRI is a non-invasive, non-ionizing imaging technique that has gained huge importance in various fields of modern medicine. MRI is enabled by Nucleic Magnetic Resonance (NMR), which was first described by Purcell and Bloch in 1946 [Blo46; PTP46]. The process behind it is roughly illustrated in Figure 2.5.

In simple terms, an MRI-machine operates by applying a very strong, external, magnetic field - referred to as $B_0$ – which forces nuclei with a non-zero spin, e.g., a $^1$H nucleus (proton), to align to and precede around it. That static field is disrupted by applying a second, radio-frequency pulse that interferes with the nucleus' alignment. As this pulse is switched of, the protons start to re-align with $B_0$ over time in what is called relaxation. This process is characterized by the relaxation times $T_1$ (spin-lattice relaxation) and $T_2$ (spin-spin relaxation), which are specific to a type of tissue, and it is accompanied by a measurable, oscillating magnetic field – the free induction decay – that induces a current in the detector coil. This signal recorded can be used to reconstruct magnetic properties of the material. To optimise the signal to noise ratio, the procedure is repeated multiple times. Furthermore, spatial resolution is atained by repetition while scanning the body or a specific part thereof. Both aspects contribute to a direct trade-off between acquisition time and image quality.

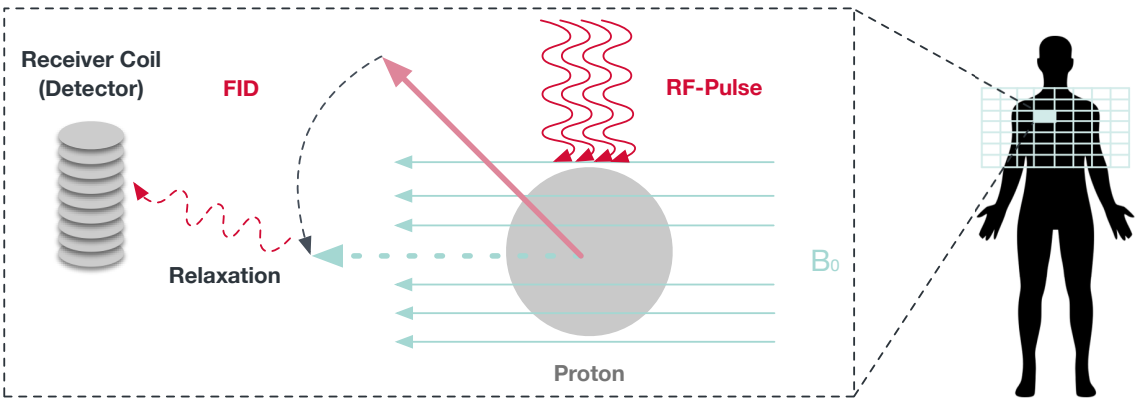



**Figure 2.5 Simplified illustration of the physical processes enabling MRI.**

MRI has the ability to generate fine-grained contrasts in different types of soft tissue. However, it comes with two critical disadvantages: Firstly, there is an inherently long acquisition time that can range anything from several minutes to over an hour per scan, especially when one tries to achieve quantification of magnetic properties of soft tissue. This is an issue because the patient is required to lie still for that time, which they are often unwilling or unable to do. Secondly, and due to this lack of fast, quantitative tools, MRI images have become mostly qualitative in nature. Anatomical areas are often referred to as either "hyperintense" or "hypointense" with respect to their immediate surrounding, but the difference between such areas or even the absolute values themselves are not measured directly.

Magnetic Resonance Fingerprinting (MRF) is a technique [MGS+13] that offers a solution to this problem and allows for a quantitative mapping of multiple properties simultaneously while lowering acquisition time. The method is described in [BMCFM+19] and illustrated in Figure 2.6. In summary, MRF is based on the assumption that different types of tissue produce a distinct signal evolution – called *fingerprint* – when exposed a specific acquisition scheme. This means that during data acquisition, measurements take place under varying (pseudo-random), external magnetic field and RF-pulse configurations and signals are recorded for the different configurations resulting in the fingerprint. That fingerprint can then be matched against a database (called dictionary) of pre-calculated, discretised fingerprints that have been simulated using *Bloch*'s theorem of magnetic resonance [Blo46]. Through that matching step, values for $B_0$, $T_1$ and $T_2$ (and other parameters) can be obtained and subsequently used to reconstruct an image. Similarly to MRI, spatial resolution in MRF is attained by scanning and the described process is therefore repeated several times.

Most MRF setups implement the pattern matching step as an exhaustive search that retrieves the simulated entry from the dictionary that maximises the inner product with the obtained signal [BMCFM+19]. This is called Maximum Inner Product Search (MIPS) and it involves evaluating the inner product between a complex signal vector and all the vectors in the database. This is a problem related to the similarity search problem being solved in multimedia retrieval, with the difference that the vectors are complex and not real-valued, that the inner product is maximised instead of minimised and that there is interest in a single result per query only. This step constitutes a bottleneck of the entire process, since many queries must be performed in sequence. Several attempts at accelerating it have been made over the years [MPM+14; CSM+15; CZR18].

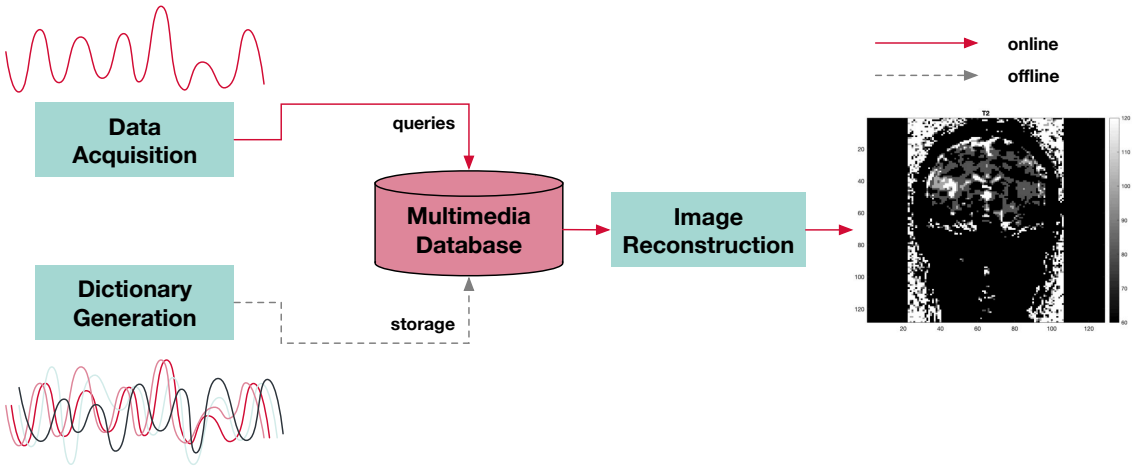



**Figure 2.6 Simplified block diagram of a MRF system (adapted from [BMCFM+19], reconstruction of brain scan kindly provided by Manuel Hürbin).**

Once again, we assume that this pattern matching and image reconstruction step should be supported by a dedicated multimedia database, which is a reasonable assumption once dictionaries become so large that they no longer fit into main memory. For a last time, we can try to describe the requirements imposed on such a database:

Firstly, and similarly to the multimedia retrieval problem, one requires a dedicated data- and query model to store complex vectors and perform MIPS. This model differs from the classical, metric-space model applied in multimedia retrieval because it does not restrict itself to real-valued vectors. However, considering complex vectors has been discussed in previous work already, e.g., in the context of similarity search on time series [RM97]. At the same time, the database must also be able to store and access very classical data that fits into the Boolean retrieval model such as the individual parameters $B_0$, $T_1$ or $T_2$.

Secondly, and also similarly to interactive multimedia retrieval, query time is an important aspect and should be minimised as much as possible, especially in a clinical setup. However, in contrast to multimedia retrieval, this optimisation cannot be bought by trading retrieval quality, because the MRF problem does not have a human in the loop who might correct an error in an individual query. Instead, the process relies on the lookup of the correct signal vector for every single query. Preliminary work on applying multimedia retrieval techniques to speeding up MRF pattern matching has shown that sacrificing accuracy leads to sub-optimal image reconstruction results [Hür20; Zih21] and thus high-dimensional indexing methods known from multimedia retrieval may not be directly applicable.

## 2.4 Deriving Requirements

In Sections 2.1, 2.2 and 2.3 we have discussed three applications that, (i) deal with large amounts of unstructured (multimedia) data that must be stored, managed and queried, (ii) use some mathematical descriptors of that unstructured (multimedia) data to enable analysis, comparison and queries, (iii) and that combine these descriptors with data such as labels, descriptions and other information, that must be accessed and sometimes queried as well.

We acknowledge, that while the details may vary, the basic needs outlined in the aforementioned use cases are overlaping to such an extent that they would benefit from a dedicated multimedia database management system. In fact, the argument for such a system has been made by different authors before us [AN97; SWS+00; ZW14; JWZ+16; Kha21] and steps towards such a system have been taken [Gia18; WYG+21]. However, all concrete approaches somehow narrow the scope in a way that restricts their generalisability.

In this section, we assume that a general-purpose multimedia database is desirable and based on the argument thus far we formalise the requirements imposed on such a system. We do not claim that we address all of the listed requirements in this Thesis. We merely attempt to raise and formalise them, similarly to and based on [AN97; JWZ+16; Kha21].

### 2.4.1 Traditional Database Capabilities

Regardless of application, any multimedia database is very likely to rely on traditional DBMS capabilities at some point. Be it when data changes or is accessed concurrently and guarantees regarding consistency must be provided or when executing Boolean queries on some aspect of the available metadata. This has been pointed out by [AN97; Kha21] already and we can therefore reiterate, that a multimedia database must bring the same, basic abilities that any traditional DBMS would w.r.t. to data modeling, data management, execution of Boolean queries and the enforcement of guarantees such as ACID or BASE.

**Requirement 2.1 A Multimedia Database is a DBMS**

At its core, any general-purpose multimedia database is a classical Database Management System and should provide the same functionality in terms of data modeling, data management, query execution and guarantees with regards to integrity- and concurrency control, persistence and recovery.

### 2.4.2 Multimedia Query Support

Multimedia queries are fundamentally different in that, as opposed to Boolean queries, they do not produce exact results but instead rely on scores and ranking to quantify similarity between a query and an object in a database. Obviously, a multimedia database system must therefore have the ability to support such queries at all levels from query formulation (i.e., query language) to query execution [AN97]. Furthermore, the combination of similarity search with Boolean queries should be possible and well-defined in terms of outcome.

It is worth noting, that by "Boolean query" we merely refer to the ability to produce exact matches based on Boolean predicates without making an assumption about the underlying data model. If a multimedia database system choses to implement the graph model then traditional graph traversal queries fall under the category "Boolean query" as well.

**Requirement 2.2 Multimedia Query Support**

A multimedia database must be able to support multimedia queries based on similarity and scores in addition to Boolean queries and it must allow the combination of the two search paradigms in a unified framework.

### 2.4.3 Multimedia Data is Big Data

At its core, dealing with multimedia data equates to dealing with *Big Data* in that a system deals with large *volumes* of information, that may change at a high *velocity* and exhibit a large *variety*, ranging from highly structured metadata attributes to completely unstructured, raw data[3]. A multimedia database can, of course, favour one of the three aspects over the others (e.g., assume that velocity at which data changes is zero) but in general, all three aspects must be considered jointly, which results in interesting engineering and research problems, e.g., for high-dimensional indexing [HJB19] or distribution of data and computation.

**Requirement 2.3 Multimedia Data is Big Data**

A multimedia database must be able to deal with large volumes of data that exhibit a broad variety and may be subject to change at a high velocity within a single, well-defined framework.

[3] This is known as the "3 V's of Big Data". Over time, more of these V's have been described in literature, for example in [KUG14].

### 2.4.4 Modelling Retrieval Quality

Classical databases produce results that match or don't match a given query, i.e., there is no notion of uncertainty or imprecission in the results. In contrast, the models used in multimedia retrieval are inherently inaccurate to some extent, in that the "best matching" entries are obtained, meaning that even bad results may be produced in the absence of better options. This imprecission or inaccuracy is influenced by many factors, not the least of which are the design and choice of the multimedia descriptors themselves.

While the inherent (in-)accuracy of any given descriptor is impossible to predict and thus out of scope for the database that holds it, the notion of accuracy still opens a relevant avenue, since some of that accuracy can be deliberately sacrificed in order to gain speed, which is often done in high-dimensional indexing. Since some use cases are very robust to such decisions while others are not, this aspect must be made explicit at a system level.

**Requirement 2.4 Model for Retrieval Quality**

A multimedia database must employ an internal model of quality of results. That model must (i) enable a user to specify the desired quality of the result (ii) allow the database system to derive the expected quality for a query execution path (iii) take both aspects into account when planning query execution.

### 2.4.5 Complex Data Types as First-class Citizen

Since multimedia analysis and analytics workloads rely on mathematical objects such as vectors, complex numbers or matrices, we argue, that the data model of a multimedia database must be extended to consider those types of data as *first-class citizens* in addition to well-established data types such as numbers or strings. This means, that the structure as well as the mathematical properties of those types must be considered for storage, query planning and execution.

This is not a new idea, per-se, and to some extent self-evident since new use cases obiously require additions to existing data models. Therefore, unsurprisingly, the idea of extensible data models and type systems was already discussed as early as 1988 [LKD+88] and it was also argued by [Gia18] that to support multimedia retrieval workloads in a relational database, one must extend the relational model to support those vectors as basic data types. However, it has been pointed out by Michael Stonebraker that the mere ability to model the data, e.g., in the scope of the relational model, is not enough to provide

satisfactory performance [SBZ+13]. He states that "In summary, RDBMSs seem inappropriate for the vast majority of science applications because they have the wrong data model, the wrong query language, lack important features, and don't provide scalability in an open source code base." [SBZ+13] (p. 56). Similarly, [WLK+10] demonstrated that the mere ability to extend a type system with user-defined types through – what they call – loose-coupling, is not enough to attain satisfactory performance for queries involving spatial data. The same observation was made when adding information retrieval support [WLL+15] and we add, that this argument can be extended to the handling of multimedia data and associated data types, which are potentially even more complex.

**Requirement 2.5 Complex Data Types as First-class Citizen**

A multimedia database must treat composite data types (e.g., vectors, complex numbers, matrices, tensors) as first-class citizens of the data, storage and execution model in addition to primitive types such as numbers or strings.

### 2.4.6 Functions as First-class Citizen

All applications considered so far involve the execution of functions that operate on complex data atoms, e.g., to obtain the similarity between a query and items in the database. For a multimedia database to be able to operate effectively, those functions must be considered first-class citizens as well, in that the multimedia database must have the ability to reason about them and make decisions regarding their execution. Hence, they should not be regarded as mere black-boxes, as, for example, a UDF. We argue, that similarly to user-defined types, there is a trade-off between loose- and tight integration of such functions into the DBMS.

This has also been layed out in the 2020 *Seattle Report on Database Research*, where the combination of "relational algebra and linear algebra in a richer query paradigm" [AAA+20] (p. 52) is listed as a possible research focus.

**Requirement 2.6 Functions as First-class Citizen**

A multimedia database must treat mathematical functions as first-class citizens of the execution model and must have the ability to reason about those functions' properties during query planning and execution.

## 2.5 Mapping Requirements to Contribution

In reference to the contributions listed in Section 1.3, we can make the following mapping between those contributions and the requirements listed in Section 2.4:

**Generalised Model for Similarity Search** addresses Requirement 2.2 on a theoretical level in that it combines the different query paradigms in a consistent yet versatile framework. The proposed model and especially its implementation also address Requirement 2.5 and Requirement 2.6 to some extent, with one focus lieing on the optimisation of function execution within a query as well as the connection between index structures and the execution of such functions. However, there are stil many aspects to be explored.

**Adaptive Index Management** provides an answer to the velocity aspect of Requirement 2.3, because managing indexes in the face of changing data obviously is a consequence of changing data. As we have argued, these changes can take place fast and they take place concurrently to other query operations, which the multimedia database must handle elastically. The proposed solution is a general-purpose approach and can obviously be complemented with more specialised optimisations at an index level.

**Cost Model for Quality** is a direct consequence of Requirement 2.3 because, as we will demonstrate, responding to changes in the data can lead to the deterioration of indexing quality. Hence, the multimedia database is forced to decide between one over the other. Since some use cases may tolerate such deterioration while other's do not, the contribution also directly addresses Requirement 2.4.

Part II

# Foundations

# Preface to Part II

Since the contributions of this Thesis rely on decades of research in database systems as well as multimedia retrieval and analysis, we will use this part to provide an overview over the relevant fundamentals. Furthermore, we use it to survey related work and to establish the common terminology, which we think is necessary to understand the contributions of this Thesis.

**Chapter 3** discusses the theory surrounding Database Management System both at a data model and systems level. Most of the fundamentals are guided by the books *DATABASE SYSTEMS The Complete Book* [GUW09] and *Database Internals* [Pet19]. Furthermore, some information was adapted from a video lecture kindly provided by Andy Pavlo – Associate Professor at Carnegie Melon University. That lecture is freely available on YouTube[4].

**Chapter 4** shifts the focus to the analysis and retrieval of multimedia data – including but not limited to images, audio and video. In addition to the basic techniques, we also discuss concrete system implementations. The two major sources used in this chapter are the books *Multimedia Retrieval* [BdB+07] and *Similarity Search – The Metric Space Approach* [ZAD+06].

**Chapter 5** tries to combine the two previous chapters and briefly surveys what efforts have been made to converge the two domains, which are, to this day, still largely disjoint areas of research. This is where the transition to the next part is staged.

All the aforementioned text books and resources provide an excellent overview of their respective fields and this is as a good time as any to thank the authors for their invaluable work. While we do indicate if specific ideas were taken directly from those sources, some information that we considered to be "general knowledge", is not always cited explicitly.

[4] https://www.youtube.com/c/CMUDatabaseGroup/

*The key, the whole key and nothing but the key, so help be Codd!*

— Mnemonic

# 3

# On The Design of a Database Management System

Database Management Systems (DBMSs), or simply "databases", power everything from small and simple websites to large data warehouses that serve millions of users in parallel. These systems play a crucial role in banking, e-commerce, science, entertainment and practically every aspect of our socio-economic lives, because more and more we rely on the processing and the insights generated from the analysis of data [Dha13] and DBMS form the backbone of many (if not most) information and data processing systems.

The first commercial DBMS were introduced in the 1960s [GUW09] and they have evolved ever since to adapt to a wide range of requirements leading to a long list of open source and commercial systems such as PostgreSQL, Microsoft SQL or MongoDB. Even though many different flavours of DBMS have emerged over the years, at their core, they still serve the same purpose:

**Management** DBMS enable their users to organise data corpora that can range from a few megabytes to houndreds of terrabytes according to a data model. For example, data can be structured into documents [HR16], graphs [AG08] or tables [Cod70] 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 and to generate the aforementioned insights.

**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]. Another one is known as BASE [Pri08].

## 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 (or multiple models, for that matter).

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 object, fact or process. For example, in an online shop, one can think in terms of customers that place orders, products that are being sold and invoices that must be sent out. 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 *attributes* that describe them and that have *relationships* among them as required by the concrete application.

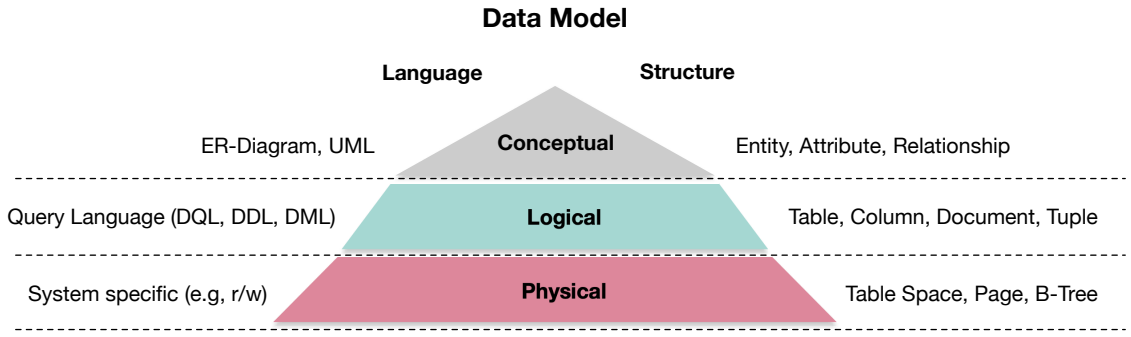



**Figure 3.1 Different levels of abstraction for data modelling.**

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 [HR16], graphs [AG08], key-value pairs and tables [Cod70], each bringing their own (often domain-specific) 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 itself. 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 data model [Cod70] and an important query language for the relational model is 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 user-intent to low-level data access operations then becomes 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] where 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, with examples dating back to the 1970s [ABC+76] and promintent, contemporary examples including systems such as Maria DB, PostgreSQL or Oracle.

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* 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) [Rei81]. That is, the relation contains all the information available about the fact and can thus be used do 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* $t_i = (a_{i,1}, a_{i,2}...a_{i,N})$ 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 $a_{i,1} \in \mathcal{D}_1, a_{i,2} \in \mathcal{D}_2...a_{i,N} \in \mathcal{D}_N$. Ergo, a relation can be seen as a subset of the cartesian product of all data domains over which it was constructed, i.e., $\mathcal{R} \subset \mathcal{D}_1 \times \mathcal{D}_2... \times \mathcal{D}_N$.

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 (apart from the constraint that its values must be atomic). 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* [1].

In addition to Definition 3.1, we will use the identities 3.2 to 3.5.

**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.

**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 list of all attributes over which $\mathcal{R}$ was constructed the *heading* or *schema* $\texttt{SCH}(\mathcal{R})$ of $\mathcal{R}$.

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

[1] 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 layed out in [Spi09]

**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 $\mathcal{A}_i \in \texttt{SCH}(\mathcal{R})$, 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 $\mathcal{A}_i \in \texttt{SCH}(\mathcal{R})$. This order can evolve over time but remains constant in a momentary snapshot of $\mathcal{R}$. It follows from the definition that $|t| = |\texttt{SCH}(\mathcal{R})| \forall t \in \mathcal{R}$

**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 $\mathcal{D} \in \mathbb{D}_{\texttt{dbms}}$ (i.e., a collection of tables). The schema of a database can then be seen as the set system of all $\texttt{SCH}(\mathcal{R}_k)$. 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., $\texttt{SCH}(\mathcal{R}_k)$, 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 `painting` 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 $\mathcal{R}_{\texttt{painting}}$**

The following table lists the schema and extent of a ternary relation ($N = 3$) $\mathcal{R}_{\texttt{painting}}$. The attributes $\mathcal{A}_{\texttt{title}}, \mathcal{A}_{\texttt{artist}}, \mathcal{A}_{\texttt{painted}}$ 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.

| $\mathcal{R}_{\texttt{painting}}$ | $\mathcal{A}_{\texttt{title}}$ | $\mathcal{A}_{\texttt{artist}}$ | $\mathcal{A}_{\texttt{painted}}$ |
|---|---|---|---|
| $t_1$ | 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 $\mathcal{P} \subset \texttt{SCH}(\mathcal{R})$ 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 $\mathcal{F} \subset \texttt{SCH}(\mathcal{R})$ 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 $R^*$. FK can be used to model relationships between relations. Using Example 3.2, the artist that created a painting is referenced through a FK $\overline{\mathcal{A}}_{\texttt{artist}}$ in $\mathcal{R}_{\texttt{painting}}$ that references the PK $\mathcal{A}^*{}_{\texttt{artist}}$ in $\mathcal{R}_{\texttt{artist}}$.

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 reald-world applications, often artificial PKs are being generated to avoid unintended collisions. Furthermore, it is typically the task of the DBMS to guarantee entity and referential integrity as part of keeping the data consistent.

**Example 3.2 Relations with Primary and Foreign Keys**

The following tables lists the schema and extent of $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$.

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

| $\mathcal{R}_{\texttt{artist}}$ | $\mathcal{A}^*_{\texttt{artist}}$ | $\mathcal{A}_{\texttt{birth}}$ | $\mathcal{A}_{\texttt{death}}$ |
|---|---|---|---|
| $t_1$ | 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 $\mathcal{R}_{\texttt{painting}}$ in Example 3.2, this means that $\mathcal{R}_{\texttt{artist}}$ cannot be stored as an attribute of $\mathcal{R}_{\texttt{painting}}$ (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 $\mathcal{R}_{\texttt{artist}}$ in Example 3.2 and assuming that $\mathcal{A}^*_{\texttt{artist}}$ was split into two attributes $\mathcal{A}^*_{\texttt{firstname}}$ and $\mathcal{A}^*_{\texttt{lastname}}$ (composite key), this means that all of the non-prime attributes must be determined by $\mathcal{A}^*_{\texttt{firstname}}$ and $\mathcal{A}^*_{\texttt{lastname}}$ jointly, i.e., the full name, and not just either $\mathcal{A}^*_{\texttt{firstname}}$ or $\mathcal{A}^*_{\texttt{lastname}}$.

**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 $\mathcal{R}_{\texttt{artist}}$ in Example 3.2, this means that all of the non-prime attributes must be determined by $\mathcal{A}^*_{\texttt{artist}}$ 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 1NF[2]. 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 ommit 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).

$$\texttt{OP} : \mathcal{R}_1, \ldots, \mathcal{R}_n \rightarrow \mathcal{R}_O \tag{3.1}$$

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

$$\texttt{QUERY} = \texttt{OP}_1 \circ \texttt{OP}_2, \ldots, \circ \texttt{OP}_\texttt{m} \tag{3.2}$$

In addition to this idea, Codd proposed a minimal set of relational operators listed in Table 3.1 and explained in the following sections. We must note that the notation and operators may differ slightly depending on the source. We mainly use [GUW09] as our reference, with a few adjustments of our own for the sake of internal consistency.

#### 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., $\texttt{SCH}(\mathcal{R}_L) = \texttt{SCH}(\mathcal{R}_R)$.

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). Due to relations being sets, duplicates resulting from a union operation are *implictly* eliminated.

[2] 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 | $\rho_{\mathcal{A}_\mathrm{A} \to \mathcal{A}_\mathrm{B}}$ | 1 | Renames attribute $\mathcal{A}_\mathrm{A}$ in input relation to $\mathcal{A}_\mathrm{B}$. |
| Selection | $\sigma_S$ | 1 | Removes tuples from the input relation that don't match predicate $S$. |
| Projection | $\pi_P$ | 1 | Removes attributes from the input relation that are not included in $P$. |
| Natural join | $\bowtie$ | 2 | Pairs each tuple from the left with every tuple of the right input relation if their shared attributes match and concatenates them. |

$$\mathcal{R}_L \cup \mathcal{R}_R = \{t : t \in \mathcal{R}_L \vee t \in \mathcal{R}_R\} \tag{3.3}$$

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).

$$\mathcal{R}_L \cap \mathcal{R}_R = \{t : t \in \mathcal{R}_L \wedge t \in \mathcal{R}_R\} \tag{3.4}$$

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).

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

These basic set operations simply combine two input relations without changing its structure, i.e., $\mathtt{SCH}(R_L) = \mathtt{SCH}(R_R) = \mathtt{SCH}(R_O)$.

#### 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).

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

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

#### 3.2.2.3 Rename

The unary *rename* operator $\rho_{\mathcal{A}_\mathtt{A} \to \mathcal{A}_\mathtt{B}}(\mathcal{R})$ renames the attribute $\mathcal{A}_\mathtt{A}$ to $\mathcal{A}_\mathtt{B}$ without changing any value. This can be useful to elliminate collisions before applying the cartesian product or to enable a natural join on differently named attributes.

#### 3.2.2.4 Selection

The unary, generalized *selection* operator $\sigma_S(\mathcal{R})$ 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 $S$ are retained as expressed by Equation (3.7).

$$\sigma_S(\mathcal{R}) = \{t \in \mathcal{R} \colon S(t)\} \subset \mathcal{R} \tag{3.7}$$

The predicate $S$ 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 $\mathcal{A}_1 \geq 2$, $\mathcal{A}_2 = \mathcal{A}_3$ or $\mathcal{A}_1 \geq 2 \wedge \mathcal{A}_2 = \mathcal{A}_3$ to express that an attribute should be greater than a constant, equal to another attribute or combine the two with $A_1, A_2, A_3 \in \mathtt{SCH}(\mathcal{R})$.

#### 3.2.2.5 Projection

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

$$\pi_P(\mathcal{R}) = \{t\,[P] : t \in \mathcal{R}\} \text{ with } t\,[P] = \{t[\mathcal{A}] \colon \mathcal{A} \in P\} \tag{3.8}$$

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 attribues $\xi = \{\mathcal{A} : \mathcal{A} \in \mathtt{SCH}(\mathcal{R}_L) \wedge \mathcal{A} \in \mathtt{SCH}(\mathcal{R}_R)\}$ as expressed by Equation (3.9)

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

The result is a relation that contains all the attributes of $\mathcal{R}_L$ and $\mathcal{R}_R$, i.e., $\mathtt{SCH}(\mathcal{R}_L \bowtie \mathcal{R}_R) = \mathtt{SCH}(\mathcal{R}_L) \cup \mathtt{SCH}(\mathcal{R}_R)$. Shared attributes are only retained once.

#### 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 formulated in a human-readable query language, which is then translated to the relational operators. A famous example of such a language is SQL [Cha12] for relational databases.

**Example 3.3 Searching for Paintings Using Relational Algebra**

The following tables lists the schema and extent of $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$.

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

| $\mathcal{R}_{\texttt{artist}}$ | $\mathcal{A}^*_{\texttt{artist}}$ | $\mathcal{A}_{\texttt{birth}}$ | $\mathcal{A}_{\texttt{death}}$ |
|---|---|---|---|
| $t_1$ | 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 $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$, followed by a selection and projection:

$$\mathcal{R}_{\texttt{result}} = \pi_{\mathcal{A}_{\texttt{title}}}(\sigma_{\mathcal{A}_{\texttt{death}}>1800}(\mathcal{R}_{\texttt{painting}} \bowtie \mathcal{R}_{\texttt{artist}}))$$

This query produces the relation $\mathcal{R}_{\texttt{result}}$ that contains $t_2$ of $\mathcal{R}_{\texttt{painting}}$.

### 3.2.3 Extensions

While the relational model and its 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 ANSI SQL standard [Lib03; Xopb]. 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. Another example could be the support for sorting or aggregation (`ORDER BY` and `GROUP BY` in SQL), which also isn't covered by the original algebra.

Over the years, this has led to a growing list of proposals for extensions, some of which have seen adoption while a majority has remained theoretical in nature. In the following sections, we will introduce a few examples. While at a first glance, these adaptions seem to be an elegant way of extending the expressiveness of the relational data model, there are important implications: Most importantly, some operators may not behave in the way specified in Section 3.2.2 or we may require additional or different operators to accomodate all the functionality needed. This adds complexity to the DBMS, especially when considering aspects such as query planning.

#### 3.2.3.1 Relations vs. Bags vs. Sequences

The motivation for considering other mathematical structures than sets as an algebraic foundation for a data model are twofold: Firstly, relations are unable to provide functionality that may be desirable in a DBMS, such as duplicate entries or explicit ordering of tuples. Secondly, some of the mathematical convenience of the relational model, e.g., prohibiting duplicates, may be inefficient to actually implement [GUW09]. It is therefore not surprising that ANSI SQL [Xopb] formally operates on *bags* rather than sets, which allow for duplicate entries [GUW09; Cha12]. If one would want to express explicit ordering of tuples, we could even have to move to sequences, in which every tuple occupies a specific position.

We will not elaborate on all the consequences such a transition may have and refer to [GUW09], which discusses this issue in great detail. However, for illustrative purposes, we still provide the Example 3.4 inspired by [GUW09]. It demonstrates that even minor changes to the properties of the purely relational data model must be taken into account when implementing systems, e.g., during algebraic query optimisation.

**Example 3.4 Searching for Paintings Using SQL**

The following tables list the schema and extent of $\mathcal{R}_{\mathtt{p1}}$ and $\mathcal{R}_{\mathtt{p2}}$, which are union compatible and have one tuple in common ($t_1$).

| $\mathcal{R}_{\mathtt{p1}}$ | $\mathcal{A}^*{}_{\mathtt{title}}$ | $\overline{\mathcal{A}}_{\mathtt{artist}}$ | $\mathcal{A}_{\mathtt{painted}}$ |
|---|---|---|---|
| $t_1$ | Mona Lisa | Leonardo da Vinci | 1506 |
| $t_2$ | The Starry Night | Vincent van Gogh | 1889 |
| $t_3$ | Las Meninas | Diego Velázquez | 1665 |

| $\mathcal{R}_{\text{p2}}$ | $\mathcal{A}^*_{\texttt{title}}$ | $\overline{\mathcal{A}}_{\texttt{artist}}$ | $\mathcal{A}_{\texttt{painted}}$ |
|---|---|---|---|
| $t_1$ | Mona Lisa | Leonardo da Vinci | 1506 |
| $t_2$ | The Birth of Venus | Sandro Botticelli | 1485 |
| $t_3$ | The Night Watch | Rembrandt Harmenszoon van Rijn | 1642 |

If now consider the union operation, i.e., $\mathcal{R}_p = \mathcal{R}_{\text{p1}} \cup \mathcal{R}_{\text{p2}}$, we will see, that for ordinary relations, $|\mathcal{R}_p| = 5$, whereas for bags, $|\mathcal{R}_p| = 6$, since since the common tuple will appear twice.
Now let us further consider the set difference and the distributive law with union, i.e., $(\mathcal{R}_{\text{p1}} \cup \mathcal{R}_{\text{p2}}) \setminus \mathcal{R}_{\text{p1}} = (\mathcal{R}_{\text{p1}} \setminus \mathcal{R}_{\text{p1}}) \cup (\mathcal{R}_{\text{p2}} \setminus \mathcal{R}_{\text{p1}})$. We can easily see, that this identity holds for sets but not for bags, since $|(\mathcal{R}_{\text{p1}} \setminus \mathcal{R}_{\text{p1}}) \cup (\mathcal{R}_{\text{p2}} \setminus \mathcal{R}_{\text{p1}})| = 2$ wheras $|(\mathcal{R}_{\text{p1}} \cup \mathcal{R}_{\text{p2}}) \setminus \mathcal{R}_{\text{p1}}| = 3$, since one of the duplicates will be retained.

#### 3.2.3.2 Extended Projection

The extended project as described by [GUW09] is an addition to the projection operator $\pi_P$ specified in the relational data model and necessitated by the ANSI SQL standard [Xopb]. It involves a more general definition of $P$, that can now contain any of the following elements for a relation $\mathcal{R}$ [3]: (i) any attribute $\mathcal{A} \in \texttt{SCH}(\mathcal{R})$, (ii) any literal value $\mathcal{L} \in \mathcal{D}_{\mathcal{L}}$, with $\mathcal{D}_{\mathcal{L}} \in \mathbb{D}$ or, (iii) any N-ary function expression $f : \mathcal{G}_1 \times, \ldots, \times \mathcal{G}_A \rightarrow \mathcal{A}_{\text{out}}$ with $\mathcal{G}_j$ being either an attribute $\mathcal{A}_i \in \texttt{SCH}(\mathcal{R})$ or a literal $\mathcal{L}_j \in \mathcal{D}_{\mathcal{L}}$.

Therefore, in addition to projecting on existing attributes, the extended projection can be used to project onto literal values and arbitrary function expressions, thus appending new attributes to a relation. This is a very powerful extension, because it enables the generation of new values based on existing ones. A lot of DBMS also allow users to define their own Used Defined Function (UDF).

#### 3.2.3.3 Aggregation

Aggregation is the act of summarizing certain attribute values based on the membership of a tuple in a specific category or group, which is established based on other attribute values. An example could be a customer who would like to see the maximum price of all products per brand. The ANSI SQL standard [Xopb] defines the `GROUP BY` operator for this purpose.

[3] We deliberately deviate from [GUW09] in that we don't limit ourselves to algebraic expressions and special functions.

[GUW09] formalises the grouping operator $\gamma_G$, which can be used to aggregate on attributes in $\mathcal{R}$. $G$ is a list of elements, which can either be: (i) any attribute $\mathcal{A} \in \texttt{SCH}(\mathcal{R})$, the attribute(s) that the operator aggregates by or, (ii) any aggregation or set function $\texttt{AGR}\colon \mathcal{A} \rightarrow \mathcal{A}_{\text{out}}$

The output relation $\gamma_G(\mathcal{R})$ is then constructed as follows: First, the tuples in the input relation $\mathcal{R}$ are grouped into $N$ groups $\mathcal{G}_1, \ldots, \mathcal{G}_N \subset \mathcal{R}$ based on the values in the specified grouping attribute(s). Subsequently, the aggregation function is executed for all tuples in each group, giving rise to an output relation that contains $N$ tuples – one per group – each bearing the grouping attributes as well as the ouput of the set functions. The ANSI SQL standard [Xopb] defines a list of supported set functions including but not limited to functions such as `COUNT`, `MIN`, `MAX`, `SUM` and `MEAN`.

#### 3.2.3.4 Sorting and Ranking

Sorting is the act of arranging items in a specific order based on their attribute values. Consider, for example, a customer who wants to look at prices of products from lowest to highest. The ANSI SQL standard [Xopb] defines the `ORDER BY` operator for this purpose.

However, formally, the act of sorting is difficult in the context of "pure" relational algebra because neither sets nor bags exhibit any ordering. [GUW09] proposes the $\omega_T$ operator to make this operation explicit. Given a relation $\mathcal{R}$, $T$ is simply a list of attributes to sort on, i.e., $T \subset \texttt{SCH}(\mathcal{R})$ and the result $\omega_T(\mathcal{R})$ is a sorted list of values. The conversion between set, bag and list takes place implicitly through application of the operator and the consequences for the algebra are not addressed.

More formal approaches to sorting were proposed, e.g., by [RDR+98] who introduced four additional operators and a *sequence algebra* that can deal with ordered queries. Similarly, [LCI+05] proposes a more systematik approach to *ranking*, which is sorting based on an external function. They introduce the idea of a *rank relation*, which carries explicit properties w.r.t to the ranking applied.

#### 3.2.3.5 Recursive Data Access

Recursion is an important concept in in computer programming, e.g., to test for reachability in graph-structured data or to model trees. Yet, neither the relational model nor the original ANSI SQL [Xopb] standard provided support for recursion [Lib03]. It was not until ANSI SQL-99, that support for recursive queries in the form of Common Table Expressions (CTEs) was added [PBB+10].

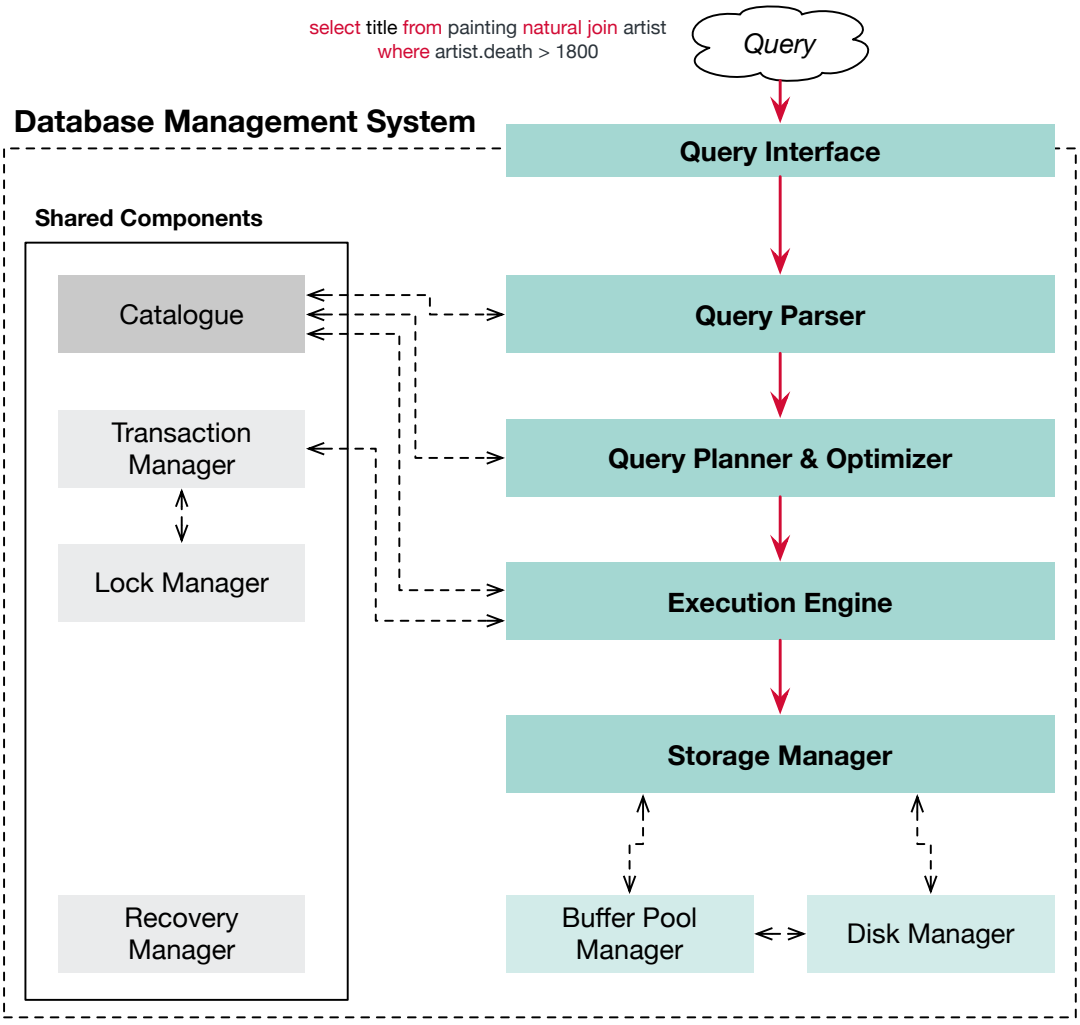



**Figure 3.2 The general architecture of a DBMS and its individual components.**

## 3.3 Database Management System (DBMS)

Irrespective of the concrete system and data model, DBMS generally exhibit a common internal structure [Pet19; HSH07] even though, specifics may vary widely between implementions. This overall structure is depicted in Figure 3.2 and in this section we use the path a query[4] takes within the system, as indicated by the directed, red arrows, to illustrate the components involved.

### 3.3.1 Query Interface

The query interface is the DBMS's channel to the outside world and enables its use by other systems. The query interface takes care of client connections, accepts and forwards queries for processing and returns results to the caller.

[4] Henceforth, the term "query" refers to any DDL, DML or DQL expression.

Queries accepted by the query interface are usually expressed in a specific query language – which are often designed for human users and therefore human-readable. We have already mentioned SQL [Xopb; Cha12] as the most prominent example of such a query language, which can be used to express queries using a declarative, textual syntax (see Example 3.5). In addition to SQL, there also exist other, domain-specific languages such as SPARQL [PAG09] to query Resource Description Framework (RDF) data, *Cypher* to query labeled property graphs [FGG+18] found in graph databases such as *Neo4j* [5] or the document-oriented query language used by *MongoDB* [6].

**Example 3.5 Searching for Paintings Using SQL**

Let $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$ be the two relations from Example 3.3, the query "return the names of all paintings that were painted by an artist who died after 1800" can be expressed in SQL as follows:

```
select title from painting natural join artist
where artist.death > 1800
```

The `select` clause expresses the projection $\pi$ to attribute `title`, the `natural join` clause expresses the natural join $\bowtie$ between `paintings` and `artist` and the `where` clause expresses the selection $\sigma$ to tuples whose `artist.death` attribute is greater than 1800.

Early attempts at a client-facing standardisation of the query interface for SQL-based systems led to the inception of the Call Level Interface (CLI) standard [Xopa], which in its early version enabled embedding of SQL commands into C or Cobol programms. The CLI standard was later complemented by Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC), which provide database connectivity to a wide range of SQL and NoSQL systems. In addition, most of the SQL (e.g., MySQL, PostgreSQL, Microsoft SQL, Oracle) and NoSQL (e.g., MongoDB, Neo4j, Redis) provide database specific and therefore un-standardised connectivity thorugh a plethora of client libraries that are maintained for different platforms and programming environments by either companies and / or open source contributors.

[5] See https://neo4j.com/, Accessed June 2022
[6] See https://www.mongodb.com/, Accessed June 2022

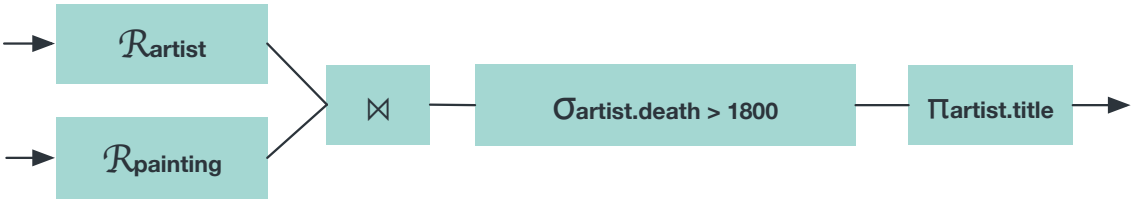



**Figure 3.3 The query from Example 3.5 represented as a tree of relational operators also called a logical query execution plan. By convention, the tree is executed and read from left to right (bottom to top) and information flows from the leafs to the root.**

### 3.3.2 Query Parser

The query parser's main task is to transform a query provided in a human-readable query language to a logical representation that can be processed by a DBMS. This often involves a conversion of a syntax to the operators specified by the data model, e.g., a tree of the relational operators that represent an SQL query as illustrated in Figure 3.3, for example. The result is what is generally called a *logical query (execution) plan*, since it outlines all the operations necessary to generate the desired results without, however, specifying concrete access methods or algorithms.

Practically, query parsing involves several steps [Gra93; GUW09]: First, the syntax of the textual query must be analysed and converted to a data structure that can be processed. Very often, Abstract Syntax Tree (AST)'s or *parse trees* based on formal grammars of the query language, and libraries such as *ANTLR*[7] or JavaCC[8] are employed in this step. Subsequently, the leaf-nodes in the AST must be mapped to Database Object (DBO)'s contained in the DBMS in a step called preprocessig. This involves lookups in the *catalogue* – a data structure that lists a DBMS' units of data organisation. For a relational database, for example, the catalogue would be used to lookup named tables and columns (e.g., tables `artist` and `painting` from Example 3.5). And finally, the nodes in the AST must be converted to an internal representation that matches the data model, e.g., a tree of relational operators. This conversion step may also involve basic sanity checks for type compatibility or the existence of requested DBOs. Certain optimisations can also be applied, e.g., simplifying trivial identities such as $1 = 1$ (which is always true) or $0 > 1$ (which is always false). If we take the query from Example 3.5, the logical query plan would look as illustrated in Figure 3.3.

[7] See https://www.antlr.org/, Accessed July 2022
[8] See https://javacc.github.io/, Accessed July 2022

### 3.3.3 Query Planner & Optimizer

A query planner tries to transform the *logical* query plan produced by the parser to a *physical* plan in a way that allows for efficient and effective query execution in terms of minimal execution time or resource usage [JK84; GUW09]. The distinction between a logical and physical representation of a query is very common in DBMS and illustrated in Example 3.6. While the former determines *what* should be produced terms of some logical algebra, the latter defines the *how* in terms of operators and concrete access methods. Every logical relation, attribute and operation has one (or multiple) physical counterparts. This is the context, in which query optimisation can take place.

**Example 3.6 Logical Query Plan vs. Physical Quer Plan**

Let $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$ be the two relations from Example 3.3, the query "return the names of all paintings that were painted by an artist who died after 1800" can be expressed in relational algebra as follows:

$$\mathcal{R}_{\texttt{result}} = \pi_{\mathcal{A}_{\texttt{title}}}(\sigma_{\mathcal{A}_{\texttt{death}}>1800}(\mathcal{R}_{\texttt{painting}} \bowtie \mathcal{R}_{\texttt{artist}}))$$

The naive, unoptimised phyiscal implementation can be obtained by mapping every relational algebra operator to a physical operator.

| Logical | Physical | Description |
|---|---|---|
| $\mathcal{R}_{\texttt{painting}}$ | $\texttt{SCAN}_{\texttt{painting}}$ | Scans every tuple in $\mathcal{R}_{\texttt{painting}}$ |
| $\mathcal{R}_{\texttt{artist}}$ | $\texttt{SCAN}_{\texttt{artist}}$ | Scans every tuple in $\mathcal{R}_{\texttt{artist}}$ |
| $\bowtie$ | $\texttt{HASHJOIN}_{\texttt{artist}}$ | Performs a hash-join on the two input relations based on the common attribute. |
| $\sigma_{\mathcal{A}_{\texttt{death}}>1800}$ | $\texttt{FILTER}_{\texttt{death}>1800}$ | Filters every tuple based on comparison. |
| $\pi_{\mathcal{A}_{\texttt{title}}}$ | $\texttt{PROJECT}_{\texttt{title}}$ | Projects onto attribute `title`. |

Despite being a central task of every DBMS and a huge body of research, there are only very few, comprehensive surveys and overviews of established techniques [JK84; Gra93; Cha98]. This has been pointed as early as 1984 and one of the reasons identified is that query optimisation is achived "by integrating a large number of techniques and strategies, ranging from logical transformations of queries to the optimization of access paths and the storage of data on the file system level" [JK84]. Consequently, many of the techniques are very specific to a concrete DBMS implementation and the underlying data and execution model.

**Example 3.7 Implementation of a `JOIN` between two relations.**

Let $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$ be the two relations from Example 3.5 with $N = |\mathcal{R}_{\texttt{artist}}|$, $M = |\mathcal{R}_{\texttt{painting}}|$ and $N < M$. The `JOIN` between the two relations, i.e., $\mathcal{R}_{\texttt{artist}} \bowtie \mathcal{R}_{\texttt{painting}}$ can be implemented by the following algorithms [Gra93]:

**Nested-Loop Join** The nested loop join iterates over all elements in the smaller relation, i.e., $\mathcal{R}_{\texttt{artist}}$ and tries to find matching entries in the larger relation, i.e., $\mathcal{R}_{\texttt{painting}}$ for every element. If $\mathcal{R}_{\texttt{painting}}$ exhibited an index on the join column, that index could be used for lookup. Otherwise, a full scan of $\mathcal{R}_{\texttt{painting}}$ must be performed for every element in $\mathcal{R}_{\texttt{artist}}$. Therefore, computational complexity ranges between $O(N \log M)$ and $O(NM)$.

**Hash Join** The hash join algorithm builds-up a hash-table for one side of the `JOIN` to speed-up the lookup of join keys, hence, ommitting the inner loop of the nested-loop join. If resource consumption of building and accessing the hash-table is ignored, the computational complexity becomes $O(N + M)$. However, typically building the hash-table is accompanied by a non-negligible cost, especially, if it does not fit into main memory.

**Sort-Merge Join** The sort-merge join can only be used if $\mathcal{R}_{\texttt{artist}}$ and $\mathcal{R}_{\texttt{painting}}$ were sorted by the join key (i.e., the name of the artist) and if the comparison operator checks for equality between the two keys. Given this pre-condition, the join can be realised in a single loop that collects the entries for distinct values of the join keys, i.e., computational complexity is $O(N + M)$, if we assume that the relations come pre-sorted.

Over the years, two high-level strategies at optimising a query have emerged: bottom-up and top-down [JK84]. Early research focused on bottom-up approaches wherein individual operators and special edge-cases were studied and optimised. This turned out to be important, foundational work since at the lowest level, query optimisation can obviously be achieved by employing efficient algorithms. Example 3.7 tries to demonstrate this using possible implementations of a `JOIN` between two relations.

However, as queries became more complex, bottom-up optimisation started to reach its limits, especially in terms of generalisability. If we turn to Example 3.7, we will realise that the optimal path of execution does not only depend on the choice of `JOIN` algorithm but also on the join order, the presence or absence of indexes that allow for more efficient access and the estimated size of

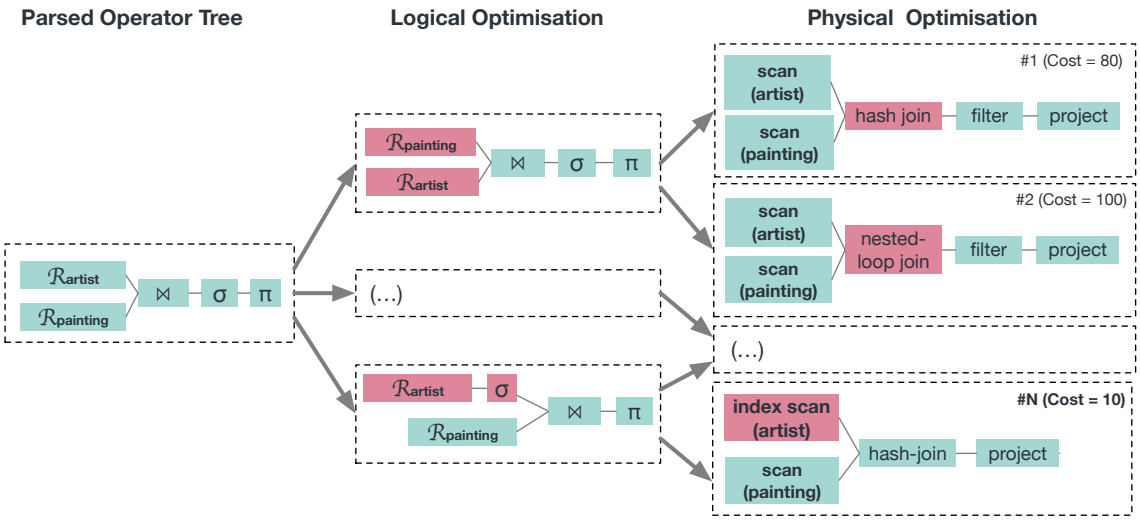



**Figure 3.4 Cost-based query planning in a DBMS based on Example 3.5. A parsed query is transformed to a series of equivalent, logical and physical plans with known costs. At the end, the most cost-effective plan is selected. Red boxes indicate adjustments to the operator tree.**

intermediate data structures and thus the size of the relations themselves. If now, in addition, we consider queries that involve multiple `JOIN`s, we must acknowledge that these metrics may differ for every combination of relation, i.e., there may not be a single, optimal choice of algorithm.

As a consequence, attention has shifted towards top-down approaches that focused on holistic optimisation of the entire query plan. Early top-down approaches tried to focus on heuristic methods [JK84]. However, the System-R paper [SAC+79] published in 1979 established the idea of *cost-based query optimisation* as a gold standard for DBMS. A cost-based query planner enumerates physical plans that employ different strategies and selects the one that minimises the estimated cost. Those costs may involve computation (time used on a CPU), storage access (data accessed on disk) or transmission (data transfered between nodes) required by a query [JK84; GUW09]. Which of those metrics are used is highly dependent on the system: In the not too distant past before the invention of the Solid State Drive (SSD), access to secondary storage was considered to be the defining cost [GUW09]. However, for distributed databases, the transfer of data between nodes has become another factor to consider [BJZ13].

Cost-based query planning relies on an internal representation of the query, e.g., the logical plan illustrated in Figure 3.3, which is transformed into eqiuvalent alternatives over multiple iterations as illustrated in Figure 3.4. As queries become more complex (e.g., multiple joins) the solution space grows exponentially, i.e., the problem is NP-hard and must be solved heuristically.

Often, the query optimisation process is divided into three phases as described for the *Volcano Optimizer Generator* [JK84; GM93] and the *Cascades Framework* [Gra95] (which still serve as a theoretical foundation for systems today [SAR+14; BCH+18]):

1. Transformation of the logical input plan to either simplify it or to replace parts that can be covered by a specific edge-cases. This results in different, alternative logical query plans.
2. Maping the transformed, logical query plans into sequences of elementary operations with known costs by selecting specific algorithms or access patterns. This results in a (potentially larger) set of physical query plans.
3. Estimate the total cost for the physical query plans and select the best one.

Each of the aformentioned steps involve sub-steps that apply algebraic equivalence rules [GUW09] (e.g., commutativity of operations), heuristics [GUW09; Gra93; Swa89; BGJ10; TSF+12] (e.g., predicate pushdown) or special rules to handle certain (edge-)cases [JK84; Gra93] (e.g., replace table by index scan).

Indexes deserve a special mention in this context, since they are of huge, practical importance. An index is an auxiliary data structure that organises access to the data in a relation based on one or multiple attributes to allow for more efficient data access when searching for an indexed attribute – $O(\log N)$ instead of $O(N)$, with $N = |\mathcal{R}|$ for $B^+$-trees. Example 3.8 illustrates this for a simple filter operation. We will address indexes structurally in Section 3.3.5.

**Example 3.8 Index Scan vs. Table Scan**

Let us consider the logical and physical plan of Example 3.6. And let us further assume that there exists a $B^+$-tree index on $\mathcal{A}_{\texttt{death}}$ in $\mathcal{R}_{\texttt{artist}}$. The following query plan could be an optimisation of the original plan:

| Logical | Physical | Description |
|---|---|---|
| $\mathcal{R}_{\texttt{painting}}$ | $\texttt{SCAN}_{\texttt{painting}}$ | Scans every tuple in $\mathcal{R}_{\texttt{painting}}$ |
| $\sigma_{\mathcal{A}_{\texttt{death}}>1800}(\mathcal{R}_{\texttt{artist}})$ | $\texttt{INDEX}^{\texttt{death}>1800}_{\texttt{artist}}$ | Scans every tuple in the index that matches the predicate. |
| $\bowtie$ | $\texttt{HASHJOIN}_{\texttt{artist}}$ | Performs a hash-join on the two input relations based on the common attribute. |
| $\pi_{\mathcal{A}_{\texttt{title}}}$ | $\texttt{PROJECT}_{\texttt{title}}$ | Projects onto attribute `title`. |

A critical factor for any cost-based query optimizer is the model used for cost and cardinality estimation [JK84; YHM15]. Most existing systems rely on statistical modeling based on the data stored in the database [GTK01; Ioa03] as well as model for cost incurred by accessing system resources [MBK02]. Simple statistics involve information about the number of entries in a relation or statistical moments of columns but may also rely on more complex analysis of data distribution such as histograms. This pre-hoc modeling was later complemented by taking the atained execution speed into account, which requires post-hoc analysis of an execution plan with respect to the cost model and establishes a continuous control loop between planner and execution engine [ML86]. Recent work in the field also explores the application of machine learning at different levels of the cost-estimation and planning process [WCZ+13; Vu19; AAA+20].

### 3.3.4 Execution Engine

Once a query execution plan has been selected, the execution of that plan falls to the DBMS's execution engine. At a high level, every operator in the plan is typically mapped to an actual operator in a query execution pipeline that performs the necessary operations, e.g., executes a scan of a table or a specific `JOIN` algorithm. This is visualised in Figure 3.5. In the resulting *operator tree*, every operator functions independently, reads input, processes it and produces output that is then handed to the next operator in the tree (or to the client).

While this model is simple enough, there are a few subtle differences in how this can be implemented by a system. First, there is a distinction as to how data is being handed from operator to operator:

**Iterator Model** This is the model described in [GM93; Gra93]. In this model, every operator reads a single tuple, processes it and hands it to the next operator in the pipeline. Therefore, tuples are iterated one by one and handed from the leafs to the root of the operator tree.

**Materialization Model** In this model, every operator reads all tuples from the input, performs its work and then forwards the entire output to the next operator. This requires potentially large buffers to store intermediate results.

**Batch Model** In this model, every operator reads a specific number of tuples from it inputs (a batch), processes these tuples and then forwards the batched results to the next operator. Therefore, data is processed batch by batch.

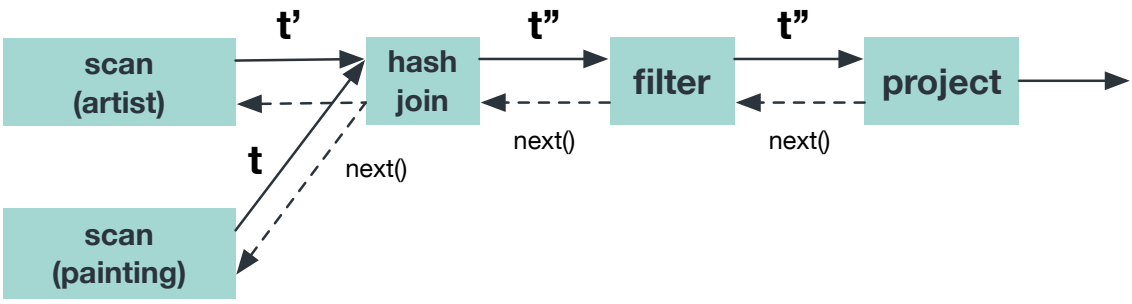



**Figure 3.5 Illustration of the iterator model for query listed in Example 3.5. Every operator calls the upstream operator with `next()` to request the next tuple for processing.**

It must be mentioned, that even in the iterator model, some operators still require intermediate materialisation. For example, a sort operation must be able to "see" all tuples in order to establish a total ordering. Consequently, intermediate caching is required in these cases. In addition, we can distinguish between which operator initiates data exchange:

**Pull-based** Operators higher-up in the operator tree request the next item from the previous operator. This is the common model for disk-based systems and leads to the slowest operator in the pipeline determining execution speed.

**Push-based** Operators lower in the operator tree send their results to the next operator, which has to buffer if it is not ready to process. While potentially more efficient, this requires a buffering infrastructure.

The pull-based iterator model is very easy to implement, since basically, every operator just needs to provide a `next()` method that returns the next tuple (in addition to `open()` and `close()` to signal start and end). Every operator then simply calls `next()` on its input to get the next tuple. The model is illustrated in Figure 3.5. It is probably the most common processing model found in disk-based DBMS, even though, it has downsides such a poor instruction locality and overhead due to the large number of method calls [NL14]. The materialization model can be beneficial for small datasets and OLTP queries. However, it is not a good fit for analytical workloads on large datasets, especially, if main memory is limited. The batch model is something in between the previous two and can be beneficial when using vectorised processing and SIMD instructions such as those provided by the AVX or SSE extensions, since multiple tuples can be processed in batches. It is thus often employed in analytical databases such as MonetDB [IGN+12].

#### 3.3.4.1 Query Parallelism

Since nowadays most computing platforms offer multiple CPUs or CPU cores, parallel execution of queries can provide a considerable speed-up in terms of execution time. [Gra93] distinguishes between three types of parallelism a DBMS can support:

**Inter-query parallelism** refers to the parallel execution of different, independent queries (potentially from different users). Roughly, there are two approaches to achieve inter-query parallelism, namely, by mapping different queries to different system processes (e.g., PostgreSQL) or threads within the same process (e.g., MySQL), with the advantage of the latter being that all queries share the same memory address space.

**Inter-operator parallelism** refers to the parallel execution of different operators within a query, which are then synchronised by a dedicated operator that merges the results. For example, one could parallelise the execution of the respective order operations on two relations prior to a sort-merge join. This is a form of *intra-query parallelism*.

**Intra-operator parallelism** refers to the splitting-up of the work performed by a single operator. This typically involves the ability to parallelise based on the data, e.g., by partitioning on it and letting different operator instances taking care of parts of the workload. This is also a form of *intra-query parallelism*.

All three aspect must be handled by the query execution engine (and other component involved) and give rise to a large number of implementation details that must be considered when building a DBMS. For example, the challenges that arise from parallel query execution and the resulting, concurrent data access – especially in the presence of parallel reads and writes – are covered by the *isolation* part of ACID and typically mitigated by a *transaction manager* and *concurrency control protocols*.

#### 3.3.4.2 Optimising Execution Speed

Optimising query execution at different levels of the system has always been an important area of research. At a hardware level, we have already mentioned the use of specialised CPU instructions such as AVX or SSE [IGN+12; PR20; PR19; SXJ21]. Recent trends take this a step further by leveraging the GPU or specialised hardware, such as FPGAs [AAA+14; AAA+20; PLH+21].

Another pupular approach used to minimise the overhead incurred by the iterator model is compilation [KVH21; FT21], e.g., implemented by HyPer [NL14]. This technique tries to pre-compile parts of the execution plan directly into machine code, i.e., merging multiple operators into one, which increases instruction locality and allows for caching of pre-compiled elements. The method can be combined with vectorised execution [SZB11; RBM22].

### 3.3.5 Storage Manager

At the lowest levels of any DBMS there is the storage manager, which orchestrates access to data on primary and/or secondary storage devices. Many different approaches at storage have been implemented over the years and we will simply focus on some basic concepts and refer to [Pet19] for more information on the topic. What most DBMS have in common is that they implement a two-level memory architecture wherin data resides on disk (which is slower but durable) and in Random Access Memory (RAM) (which is faster but volatile). The two aspects are handled by the two main components: the *disk manager* and the *buffer pool manager* [Pet19]. Recent years have also seen a trend towards main memory databases that keep data primarily in RAM [GS92; FKL+17]

#### 3.3.5.1 Data Organisation and Disk Manager

DBMS are typically required to accomodate multiple design goals at once. They are supposed to store data, such that, (i) space occupied by the data, (ii) time required to access (i.e., read) records and (iii) time required to update (i.e., write) records are minimised. In order to achieve all these goals, a DBMS often combines different types of data structures, which results in a distinction between two main types of files[9]: The larger *data files* (or primary files) and smaller *index files*. While the data files contain the data itself, index files contain metadata that allows for faster access. Most database systems access both types of files in terms of fixed-size database pages that are read and/or written in their entirety[10]. It is the main task of the *disk manager* to provide these low-level primitives. The organisational structure of the (meta-)data is then superimposed onto the page structure. There are many implementation details to be considered, such as dealing with variable-length data, dealing with data that is larger than a page, checksumming and versioning and different techniques are being applied to address these issues.

[9] These are not necessarily separate files in the file system.
[10] The size of a page is often a fixed multiple of the block-size implemented by the file system.

In a relational database, the data files can be regarded as the physical manifestation of a relation. There are many different ways to organise such data files and we refer to [Pet19] for details. Roughly, we can distinguish between index-organised, hash-organised and heap-organised tables, which arrange the tuples in different data structures (i.e., trees, hash tables or heaps), with advantages and disadvantages inherent to the respective choice. Recent work has given rise to novel ways of data organisation, e.g., in systems like MonetDB where data is organised in fixed size arrays called binary association tables [BKM08]

In order to be able to locate and relate individual records accross different pages, files and data structures, there is usually a notion of an internal *tuple identifier* `TID`, which uniquely identifies every record (see Definition 3.6). Conceptually, the tuple identifier connects the logical notion of a tuple to a physical world in which such a tuple must be stored, located and accessed as a record.

**Definition 3.6 Tuple Identifier of a Relation $\mathcal{R}$**

Let $\mathcal{R}$ be a relation. We call `TID` the relation's *tuple identifier* if it uniquely identifies every tuple in $\mathcal{R}$, i.e., has a unique value per tuple.

`TID` can be considered an implicit, internal primary key of $\mathcal{R}$ that is often not visible to the outside and may even change its value as entries are being inserted, updated and deleted. However, some systems actually make the tuple identifier accessible, e.g., Oracle with its `ROWID` and in some database implementations, an explicit primary key may act as `TID`, which, however, does not affect its definition and purpose.

Index files enable faster data access, especially in the presence of predicates on specific attributes. [Pet19] distinguishes between primary indexes and secondary indexes. A primary index typically indexes the primary key or tuple identifier and is the main mean of data access. However, depending on how the data file is organised, no primary index is required (e.g., index-organised tables). Queries that do not involve the evaluation of the primary key, typically result in a *linear scan* of the primary index and/or the data files, which scales poorly to a large number of entries. This is addressed by auxiliary, secondary indexes, which are used to optimise query execution performance for certain queries by indexing selected attributes. A very popular index structure is the $B^+$-tree [BM70], which reduces runtime complexity of certain search operations from $O(N)$ (linear scan) to $O(\log N)$ with $N = |\mathcal{R}|$. Logically, an index can be regarded as described in Definition 3.7.

**Definition 3.7 Index Structure for Relation $\mathcal{R}$**

Let $\mathcal{R}$ be a relation with tuple identifier TID and $I = \{\mathcal{A}_1, \ldots, \mathcal{A}_K\} \subset \texttt{SCH}(\mathcal{R})$. We call $\mathcal{I}_I^{\mathcal{R}}$ a (dense) index of relation $\mathcal{R}$ with respect to $I$ if $\mathcal{I}_I^{\mathcal{R}}$ contains at least TID and if $|\mathcal{I}_I^{\mathcal{R}}| = |\mathcal{R}|$ and $\exists(t_1, t_2) : t_1[\texttt{TID}] = t_2[\texttt{TID}] \forall t_1 \in \mathcal{R}, t_2 \in \mathcal{I}_I^{\mathcal{R}}$.

In simple terms, an index $\mathcal{I}_I^{\mathcal{R}}$ can logically be seen as a copy of $\mathcal{R}$ with respect to the attribute TID, i.e., every value of TID in $\mathcal{R}$ is somehow contained in $\mathcal{I}_I^{\mathcal{R}}$. For example, an index could cluster all TID based on the attributes in $I$. As a consequence, we can always use $\mathcal{I}_I^{\mathcal{R}}$ instead of $\mathcal{R}$ and we can always reconstruct the attibutes $A = \{\mathcal{A} \colon \mathcal{A} \in \texttt{SCH}(\mathcal{R}) \wedge \mathcal{A} \notin \texttt{SCH}(\mathcal{I}_I^{\mathcal{R}})\}$ that are contained in $\mathcal{R}$ but not contained in $\mathcal{I}_I^{\mathcal{R}}$ by considering the equi-join[11]:

$$\mathcal{R} \equiv \mathcal{I}_I^{\mathcal{R}} \bowtie_{\texttt{TID}} \pi_A(\mathcal{R}) \tag{3.10}$$

However, many of the properties that go beyond Definition 3.7, e.g., which additional attributes are contained in $\mathcal{I}_I^{\mathcal{R}}$ or how data in the index is organsied, depend on the type of index and the concrete implementation.

#### 3.3.5.2 Buffer Pool Manager

The main objective of the buffer pool manager is to keep frequently accessed pages in memory for faster lookup. That is, if upper-levels of the DBMS require access to a given page, they request it from the buffer pool manager, which can either serve it from memory or read it from disk through the disk manager. Meanwhile, the buffer pool manager can keep track of page access and can evict or retain pages depending on usage. In addition to caching, buffer pool managers also implement different strategies for page replacement and eviction (e.g., FIFO or LRU) and pre-fetching of pages depending on workloads.

Some DBMS, such as MonetDB [BKM08] or LMDB [Hen19], forego the implementation of a buffer pool manager and instead rely on the POSIX `mmap` system call [Sto81], which allows the mapping of files on disk into the address space of an application and leaves caching of pages to the operating system. While this simplifies the implementation because loading and evicition of pages must not be handled explicitly, research suggest that the use of `mmap` in a DBMS not optimal [CLP22], and that knowledge about the database workload (e.g., random access vs. scan) allows for much more finegrained control than `mmap` for optimisations.

[11] This is merely a logical construct not related to how data access is implemented.

### 3.3.6 Common Components

The following components are also part of a DBMS and are briefly described for the sake of completeness:

**Catalogue** The catalogue is used to persistently store and lookup information used by DBMS to organise the data it contains, i.e., information about units of data organisation such as tables, columns, collections, indexes as well as metadata such as statistics, transactions or locks.

**Transaction** One or multiple queries processed by a DBMS are often wrapped in transactions, i.e., a series of operations that should be executed together and either be committed or rolled back at the end. Often, the DBMS provides guarantees with respect to the execution of transactions such as ACID [HR83] or BASE [Pri08], e.g., assuring an all-or-nothing semantic (atomicity) for operations being executed within a transaction (the A ind ACID).

**Transaction Managers** In order to provide the aforementioned guarantees for a transaction, the query workloads must be orchestrated across different components (e.g., disk manager, buffer pool manager, execution engine, lock manager, recovery manager). This is the main task of the transaction manager in addition to keeping track of a transaction's lifecycle.

**Lock Manager** One approach to provide transaction isolation (the I in ACID), i.e., proper protection of the data in case different transactions access it concurrently, is the implementation Strict 2-Phase Locking (S2PL). This is typically provided by a lock manager, which keeps shared and exclusive locks on pages that are accessed and detects lock cycles.

**Recovery Manager** Typically, databases are required to provide some form of durability with respect to the execution of a transaction. Therefore, once a transaction commits, changes are expected to be permanent (the D in ACID). This is guaranteed by different techniques such as write-ahead logging which allow for data reconciliation in case of a system failure (e.g., power outage). This is the main task of the recovery manager and a famous algorithm is known as ARIES [MHL+92].

Obviously, some databases feature additional system components depending on concrete applications, e.g., a distribution manager for DBMS that provide distribution of data and processing accross multiple nodes.

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

— Immanuel Kant, 1781

# 4

# On Multimedia Analysis and Retrieval

Multimedia refers to the combination of different content forms for the purpose of transport and perception. Over the centuries, countless content forms have been conceived by mankind, natural language being one of the earliest examples. With the invention of the modern computer, more and more of these content forms were transfered to the digital world. Today, multimedia data is ubiquitous and used in different forms in every area of our socioeconomic lives. Statista.com estimates, that the amount of data created, captured, copied, and consumed in 2025 will exceed 181 ZB from an estimated 2 ZB in 2010[1]. To a large extent, that data is multimedia data that is being created, stored, analysed and shared for various purposes.

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 turning them effectively into prosumers [RJ10; RDJ12]. Fueled by this technological progress, the past decades have seen a staggering development in both *volume* and *variety* of multimedia data found on the Internet and in commercial and 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. Therefore, researchers have tried to tackle the questions surrounding the efficient and effective management, analysis and retrieval of multimedia data at large scales for many decades now, with first attempts reaching back to the early 70s.

[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 *content formats* or *media types*, such as but not limited to, observable aural or visual information. We distinguish between these content formats based on the representation we use when working and interacting with them. Raw sound, for example, is formed by airpressure waves that are registered by our ears. In contrast, visual information involves electormagnetic waves captured by our eyes. In both cases, the brain interprets the underlying processes thus forming the human perception of the phenomenon and the meaning it carries.

Traditionally, we often think of media as information that can somehow be observed directly by our senses. However, since a medium is merely a mean to store and transport information, it may also include less apparent and sometimes even digitally native examples such as data streams stemming from sensors, which require specialised hard- and software to allow for observation.

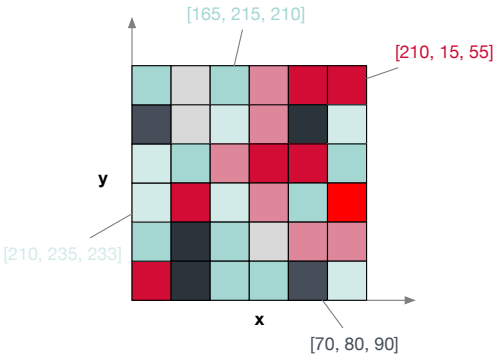
Even though the various types of media may exhibit very different characteristics in their original representation, we find some commonalities that are crucial for formalising the problem of processing and analysing their digital (data) representation (see Examples 4.1 and 4.2) in information processing systems – which is our main interest for the purpose of this Thesis.

**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 in that there is no connection between the raw data and the higher-level meaning it carries. For example, there is no way to deduce the motives visible in an image by just examining the raw image data. 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**

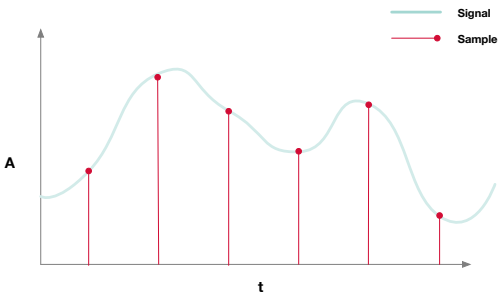



The visual information in a flat image is stored as a two dimensional array of *pixels*. The information in every pixel is typically formed by a sensor, in an array of sensors, that captures the electormagnetic signal. Each pixel holds colour values, usually one per colour channel. For example, with 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 $1000 \times 1000$ pixel, we must encode $3 \times 10^6$ individual colour values. Using 8 bit per colour, we end up storing $24 \times 10^6$ 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 quantises 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 quantisation 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 Collections

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 *vitrivr* 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 (typically a large number of) *media object*s. 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. That raw data must always be available for processing and presentation.

To account for the temporal development found in dynamic media types, *vitrivr*'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. In addition, the proposed distinction also allows for application specific media types and segmentation strategies, such as image sequences – which were introduced for LSC 2020 [HAPG+20] and group 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 with 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 that can contain technical metadata, descriptions, labels or other type of structured information.

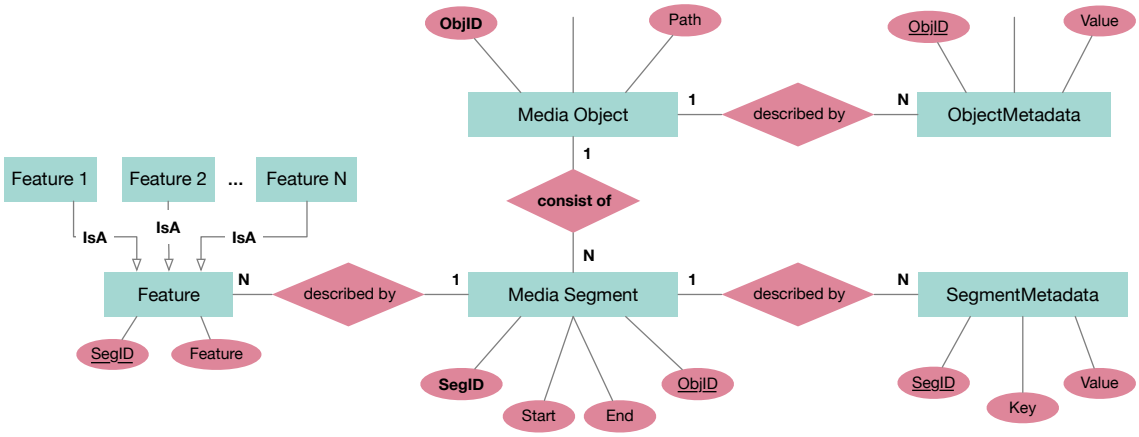



**Figure 4.1 An extended ERM of the multimedia data model used in the *vitrivr* project. The model is centered around the notion of media objects and media segments which are described by metadata and features.**

For the purpose of this Thesis, we consider a slightly altered version of the aforementioned data model, which is more in line with [BdB+07] and is illustrated in Figure 4.2. This model also considers media objects to be an abbstraction of individual files. However, this model foregoes the indirection introduced by the media segment and treats *features*, *annotations* and *descriptions* as different types of a more general *metadata* type that describes the media object directly. To model the temporal aspect in dynamic media types, the metadata itself exhibits information about the part of the object that is being described, denoted by an optional *start* and an *end* marker. In practice, this could be a frame number or a timestamp.

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 information as a 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.

We will use the proposed, conceptual data 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.

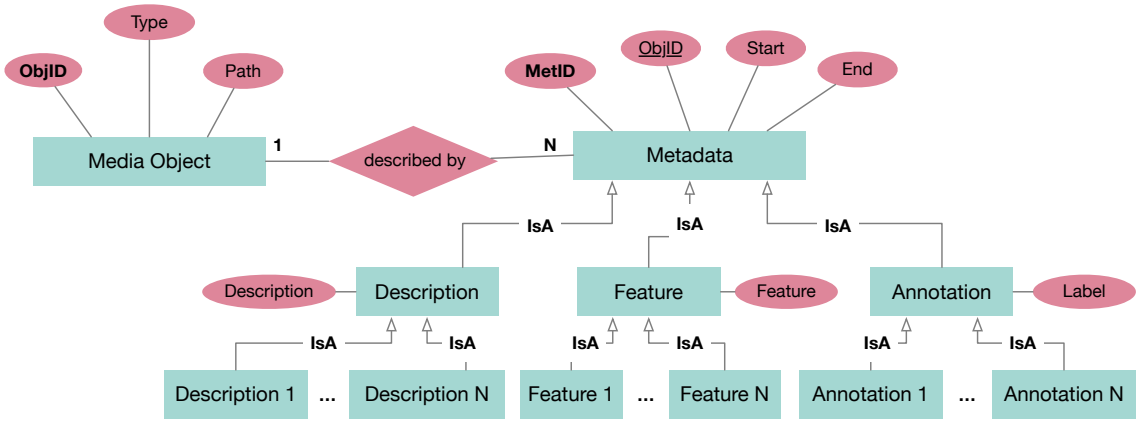



**Figure 4.2 An extended ERM of the multimedia data model used for the purpose of this Thesis. It has been derived from *vitrivr*'s data model and foregoes the explicit segment entity.**

### 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, ID3[2] tags allow for organisation of large music libraries and classification of songs based on information about artists or albums. Similarily, 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, EXIF[4] 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.

### 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 formal definition is given in Definition 4.1.

**Definition 4.1 Definition of a Feature in Multimedia Analysis**

Let $\mathbb{O} = \{o_1, o_2, \ldots, o_N\}$ be a media collection of media objects $o$ (i.e., files). A *feature* $f_{i,\texttt{start},\texttt{end}} \in \mathbb{F}$ describes a temporal interval $[\texttt{start}, \texttt{end}]$ with $\texttt{start}, \texttt{end} \in \mathbb{N}_0, \texttt{start} \leq \texttt{end}$ of the media object $o_i \in C$ under the feature transformation $\mathfrak{t}$, with

$$\mathfrak{t} : C \times \mathbb{N}_0 \times \mathbb{N}_0 \longrightarrow \mathcal{F} \tag{4.1}$$

We call $\mathbb{O}$ the media object domain and $\mathbb{F}$ the feature domain. The structure of $\mathbb{F}$ is fully determined by $\mathfrak{t}$.

Features play a central role in many different domains, ranging from mere (multi-)media analysis to retrieval. [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 the colours found in an image would be considered a low-level feature, whereas a feature capturing salient keypoints or even objects in the same 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 a given media type's content, with first attempts in computer vision and image understanding dating back to the early 60s. 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. Obviously, every type of media has their own analysis techniques that can be used to generate features.

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].

Similarily, 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 neutwork architectures have enabled the feature extraction process to be automated to some extend, 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 chose 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 multimeda 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 as 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 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 [SWS+00]. 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* [KCH95], *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 content-based image retrieval [DA13], audio retrieval [Lu01], video retrieval [HXL+11], 3D model retrieval [YLZ07] and various subdomains [MK18].

Formally, the multimedia retrieval problem can be characterised as finding all media objects $o \in \mathbb{O}$ that may be relevant to a given *information need* expressed by a user as query $q$. This is achieved by obtaining some form of *(dis-)similarity*[5] between $q$ and $o$ and ranking results based on it, which is why the method is also referred to as *similarity search*. In contrast to Boolean search, the results of similarity search are non-binary in the sense that an object $o$ may match a query $q$ only to a certain degree but still be considered part of the resultset, as opposed to a strict match / no-match semantic.

[5] Depending on the context, the similarity of an object to a query may be directly or inversely proportional to the obtained score, which is what makes the difference.

Since a direct comparison of a media object to a query is often not feasible – 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, leading to the formal Definition 4.2 of similarity search.

**Definition 4.2 Definition of the General Similarity Search Problem**

Let $\mathbb{O} = \{o_1, o_2, \ldots, o_N\}$ be a media collection of media objects $o$ and let further $\mathbb{F}$ be a feature domain of $\mathbb{O}$ under the feature transformation $\mathfrak{t}$. Furthermore, we assume a query $q \in \mathbb{F}$ and the existence of an inverse $\mathfrak{t}'$ with $\mathfrak{t}'(f) = o$ [a]. Similarity search then is the optimisation problem of finding the object $o \in \mathbb{O}$ that given a query $q$ and a similarity function $\mathfrak{s}\colon \mathbb{F} \times \mathbb{F} \to [0,1]$ maximises the following expression.

$$o_r = \mathfrak{t}'(\operatorname*{argmax}_{f \in \mathbb{F}} \mathfrak{s}(f, q)) \tag{4.2}$$

We call the resulting item $o_r$ most similar w.r.t. to $q$, which is indicated by the similarity score $\mathfrak{s}(f, q)$. In practice, we may be interested in the top $k$ results rather than just a single item.

[a] While this may seem like a wild assumption at first, this is a given since we explicitly store the relationship between $o$ and $f$, according to the data model in Section 4.1.

In simple terms, a multimedia retrieval system operates upon features $f \in \mathbb{F}$ derived from the original media objects by means of a feature transformation. 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, leading to $q$. Subsequently, the system generates the similarity measure $\mathfrak{s}(q, f)$ which quantifies the similarity between $q$ and $f$ from low (0.0) to almost identical (1.0) and selects the top $k$ entries that maximise this score. The flow of information that results from this approach is illustrated in Figure 4.3.

Oftentimes,s the function $\mathfrak{s}$ may not express similarity but rather dissimilarity between objects, in which case we can simply consider the dual problem. Furthermore, the score may not always be normalized to $[0,1]$, which makes it difficult to combine scores of multiple features, e.g., in *score-based late fusion* [DM10; Ros18]. However, a normalised similarity score can always be derived by applying an additional correspondence function $\mathfrak{c}\colon \mathbb{R}_{\geq 0} \to [0,1]$. Both aspects are of high practical relevance but they do not alter the problem definition.

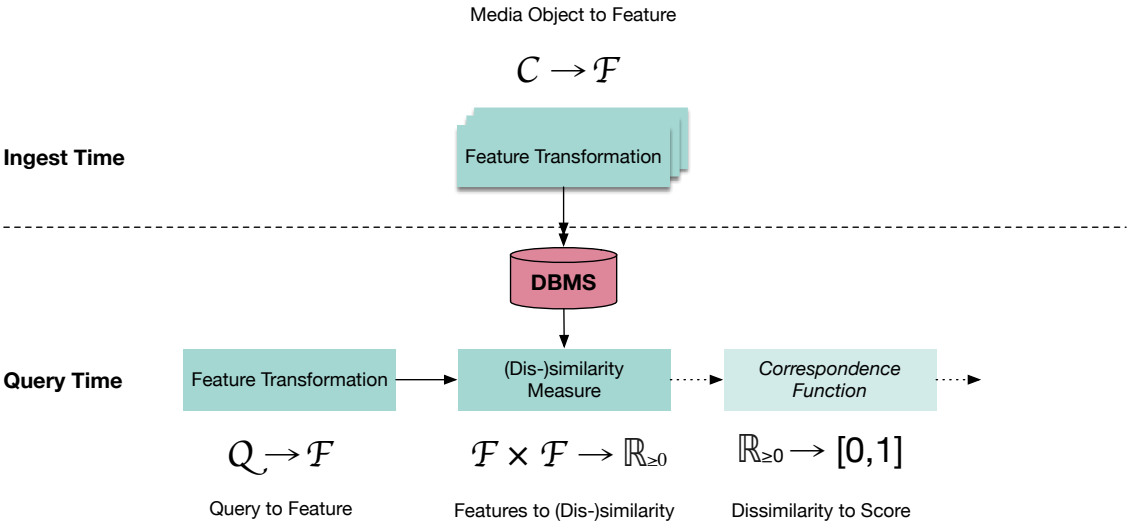



**Figure 4.3 Flow of information in similarity search at ingest- and query time.**

### 4.2.1 Similarity Search and the Vector Space Model

The *vector space model* of similarity search [SWY75] assumes that the domain of the feature vectors $\mathbb{F}$ 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., $\mathbb{F} \subset \mathbb{R}^d$ wherein $d$ is the dimension and an inherent property of the feature and the underlying transformation $\mathfrak{t}$. Consequently, every feature $f \in \mathbb{R}^d$ is simply a point in that high-dimensional vector space.

Under this premise, we consider the dissimilarity function $\mathfrak{d}(f, q)$ to be a function that calculates the distance between a feature and a query $f, q \in \mathbb{F}$, as defined by Equation (4.3). Typically, the farther two vectors lie apart under the distance measure, the more dissimilar they are, which is why we also call this proximity based search. A list of important distance functions used in similarity search is given in Table 4.1.

$$\mathfrak{d} \colon \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{R} \tag{4.3}$$

In broad terms, one can distinguish between *continuous* and *discrete* distance functions [ZAD+06]. Continuous functions typically exhibit a very large, potentially infinite co-domain wheras discrete functions map to a limited, often pre-defined set of values. Furthermore, some distance functions induce a topology on the underlying vector space – often denoted as *metric space* $(\mathbb{F}, \mathfrak{d})$. We call these *metric distances* and they satisfy the *non-negativity*, *symmetry*, *identity of indiscernables* and *triangle inequality*.

$$\mathfrak{d}(f,q) \geq 0 \qquad \text{(non-negativity)}$$
$$\mathfrak{d}(f,q) = \mathfrak{d}(q,f) \qquad \text{(symmetry)}$$
$$\mathfrak{d}(f,q) = 0 \iff f = q \qquad \text{(identity of indiscernables)}$$
$$\mathfrak{d}(f,q) \leq \mathfrak{d}(f,g) + \mathfrak{d}(g,q) \qquad \text{(triange inequality)}$$
$$\forall f,g,q \in \mathbb{F} \wedge (\mathbb{F},\mathfrak{d})$$

The constraints imposed on a metric space 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 exhibit all the aforementioned properties of a metric, e.g., quasi-metrics (lack symmetry), semi-metrics (lack triangle inequality), meta-metrics (lack identity) [ZAD+06] or even necessitate the use of non-metric functions [SB11].

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

In the follwing sections, we will explore different variants of the similarity search problem, which are all variants of Definition 4.2 and illustrated in Figure 4.4 and Example 4.3

| Name | Symbol | $\mathfrak{d}(\mathbf{f},\mathbf{q})$ | Domain | Co-Domain | Metric |
|---|---|---|---|---|---|
| Manhattan | $L^1$ | $\sum_{i=1}^{d} \lvert f_i - q_i \rvert$ | $\mathbb{R}^d$ | $\mathbb{R}$ | 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 if $p \in \mathbb{N}_{\geq 1}$ |
| Chi-Square | $\chi^2$ | $\sum_{i=1}^{d} \frac{(f_i - q_i)^2}{f_i + q_i}$ | $\mathbb{R}^d$ | $\mathbb{R}$ | metric |
| Kullback-Leibler | - | $\sum_{i=1}^{d} q_i \log(\frac{q_i}{f_i})$ | $\mathbb{R}^d$ | $\mathbb{R}$ | non-metric |
| Cosine | - | $1 - \frac{\sum_{i=1}^{d} f_i q_i}{\sqrt{\sum_{i=1}^{d} f_i^2} \sqrt{\sum_{i=1}^{d} q_i^2}}$ | $\mathbb{R}^d$ | $[-1,1]$ | semi-metric |
| Jacccard | - | $1 - \frac{A \cap B}{A \cup B}$ | $\mathbb{F}$ | $[0,1]$ | metric |
| Levenshtein | - | - | $\mathbb{S}$ | $\mathbb{N}$ | metric |
| Hamming | - | - | $\mathbb{S}$ | $\mathbb{N}$ | quasi-metric |
| LCS [Hir77] | - | - | $\mathbb{S}$ | $\mathbb{N}$ | non-metric |

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

#### 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 *vitrivr* [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 $f \in \mathbb{F}$ 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 (4.4).

$$\{R \subset \mathbb{F} \colon \forall f_r \in R, f \in \mathbb{F} \setminus R, \mathfrak{d}(q, f_r) \leq \mathfrak{d}(q, f) \wedge |R| = k\} \tag{4.4}$$

Algorithmically, the distance between all $f \in \mathbb{F}$ 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 $\mathbb{F} \subset \mathbb{R}^d$, its runtime complexity is $O(dN)$ and thus depends linearly on the cardinaltiy $N = |\mathbb{F}|$ 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

Range search or $\epsilon$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 $f \in \mathbb{F}$ that lie within $\epsilon$ of $q$ as illustrated in Figure 4.4(b). Formally, the resultset $R$ is given by Equation (4.5).

$$\{R \subset \mathbb{F} \colon \forall f_r \in R, \mathfrak{d}(q, f_r) \leq \epsilon\} \tag{4.5}$$

Algorithmically, this search strategy is similar to NNS in that all features must be evaluated exhaustively and in that its complexity for $\mathbb{F} \subset \mathbb{R}^d$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 $\epsilon$.

Variants of $\epsilon$NN include instances, where $f$ must be larger than $\epsilon$ or where $f$ must fall into an interval $[\epsilon_l, \epsilon_u]$. 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 [KM00] we try to find all $f \in \mathbb{F}$ that given a limit $k \in \mathbb{N}$ consider the query $q \in \mathbb{F}$ to be part of their kNN resultset as illustrated in Figure 4.4(c). Formally, the resultset $R$ is given by Equation (4.6)

$$\{R \subset \mathbb{F} \colon \forall f_r \in R, q \in \mathtt{kNN}(f_r) \wedge f \in \mathbb{F} \setminus R \colon q \in \mathtt{kNN}(f)\} \tag{4.6}$$

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

**Example 4.3 Comparing different types of similarity search algorithms**

For this example, we consider $f \in \mathbb{F} \subset \mathbb{R}^2$ to be points of interest (e.g., museums, hotels, parks) on a two-dimensonal city map. Furthermore, we consider $q \in \mathbb{F}$ 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:

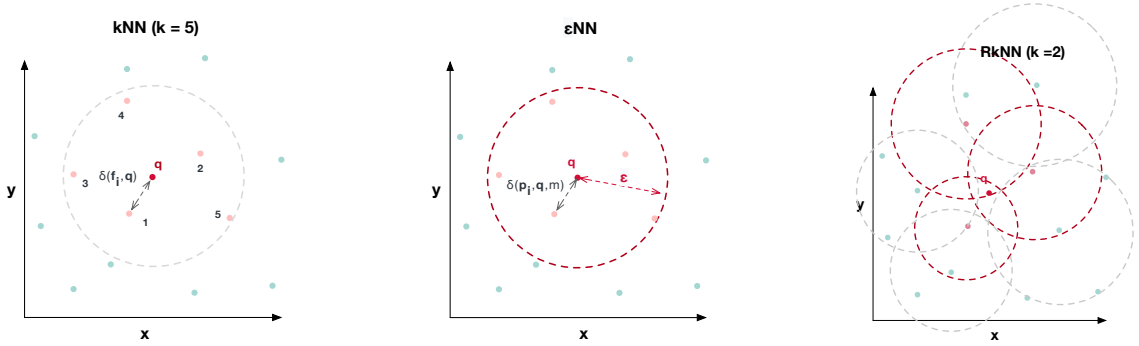
**Nearest Neighbour Search** is the equivalent of finding the $k$ points of interest that are closest to $q$ (e.g., the three museums closest to my location).

**Range Search** is the equivalent of finding all points of interest that are within $\epsilon$ 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).

### 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 $N = |\mathbb{F}|$ 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.




((a)) kNN ((b)) $\epsilon$NN ((c)) RkNN

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

In addition, for search in high-dimensional vector spaces, the dependency on $d$ has a huge impact on the 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 NNS was taken on by various researchers and is still an active area of research with results of high practical relevance. At a high-level, indexing algorithms can be classified along three dimensions: First, based on whether they provide any guarantees w.r.t. to the quality of the results, wherein quality is often measured in terms of precission and recall of a result produced by an index as compared to the result produced by a linear scan [EZP21]. Since a majority of established techniques sacrifice quality for speed [SJH+05], this is also known as Approximate Nearest Neighbour Search (ANNS). Second, based on their *organisation structure* [SR06], e.g., graph and inverted index [SZ03] or multi-index [BL14]. And finally, by the method that is employed, e.g., *quantisation*, *hashing*, *partitioning* or *clustering*. An overview of different algorithms is provided in Table 4.2.

| Name | Guarantees | Structure | Method |
|---|---|---|---|
| KDB-Tree [Rob81] | exact | tree | space partitioning |
| $R^*$-Tree [BKS+90] | exact | tree | data partitioning |
| VAF & VA+ [WSB98; FTA+00] | exact | list | scalar quantisation |
| LSH [IM98; WZs+18] | approximate, probabilistic | inverted index | hashing |
| PQ [JDS10] | approximate | list / inverted index | vector quantisation |
| CP & eCP [CPR+07; GJA10] | approximate | inverted index | clustering |
| NV-tree [LÁJ+09] | approximate | tree | scalar quantisation |
| NGT [Iwa16] | approximate | graph / tree | proximity graph |
| MRNG [LÁJ+09] | approximate | graph | proximity graph |
| HNSW [MY18] | approximate | graph | proximity graph |
| SPANN [CZW+21] | approximate | graph | proximity graph |

**Table 4.2 Non-exhaustive list of high-dimensional index structures employed in similarity search and retrieval.**

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 [Ben75; Rob81; FB74] or the data [Gut84; BKS+90; 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. These method were followed by approximate methods, [IM98; JDS10].

Recent research focuses on the use of graphs [SOV+21], for example, proximity graphs [ZWN22] or Hierarchical Navigable Small World (HNSW) graphs [MY18; CZW+21], to facilitate fast ANNS. Another focus lies on the ability to execute NNS indexing and search on a GPU [JDJ19; ZTL20] or specialised hardware [LCM+22]. Another research focus goes in the direction of dynamic indexes, i.e., indexes that allow for online processing of changes to the data [ÓÞJA+11; ZWN22] and disk-based indexing [JSDS+19]. In the next few sections, we will highlight a few, selected examples of high-dimensional index structures.

#### 4.2.2.1 Vector Approximation Files (VAF)

VA-files were proposed by Weber et al. [WSB98] to address the problem of the dimensionality curse. 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's position in the resulting lattice is concatenated into a compact *signature* of size $b$ and the VA-file constitutes a list of those signatures.

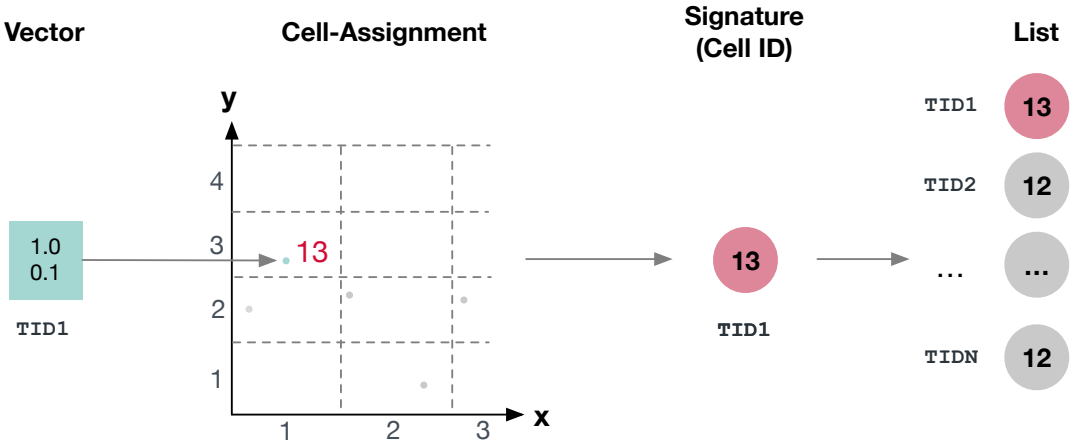



**Figure 4.5 Simplified illustration of basic VAF indexing scheme in for $\mathbb{F} \subset \mathbb{R}^2$. A vector is assigned to a cell in an overlay grid, which gives rise to a signature that is stored in a linear list with the vector or its tuple identifier.**

NNS is then realised as a linear scan of the VA-file. Speed-up over a brute-force scan of the original vectors is achieved in two ways: Firstly, the signatures are more compact than the original representation ($b$ bits per dimension), which reduces IO cost through quantisation. 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 more than 90% of all features can be skipped and thus a distance must only be obtained for the remaining 10%. However, VA-files remain linear in nature and [EZP21] determine theoretically that for VAF, a degeneration is still to be expected for high dimensions. However, a huge advantage of the VAF index is, that it does neither sacrifice precision nor recall to obtain speed-up.

The VA+-files described by [FTA+00] are an extension to the VAF-file and introduce a few optimisations to the original algorithm, which implicitly assumes independence of the individual dimensions within a feature (which is often not a given). They suggest a prior decorellation of the individual dimensions by applying a Karhunen-Loève Transform (KLT). Furthermore, they propose a non-uniform assignment of bits per dimension, based on data distribution, as opposed to a uniform distribution employed in the original paper. This leads to an independent quantiser per transformed dimension. Results suggest that these adjustments lead to a better estimation of lower- and upper-bounds, which improves the VA-file's filtering capabilities and thus further reduces IO cost.

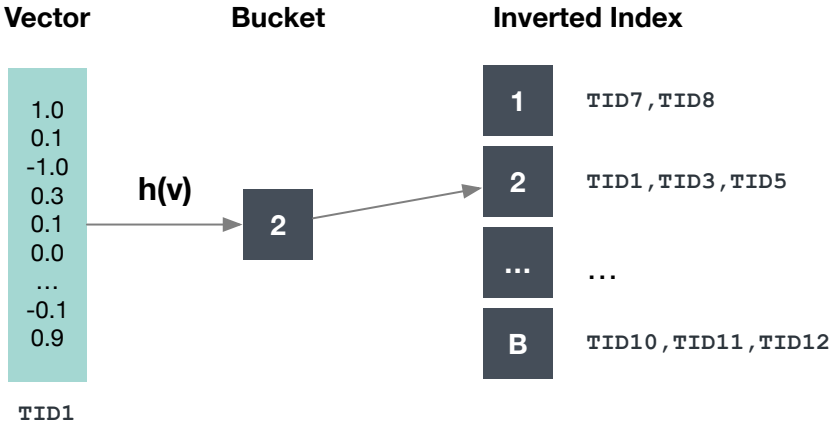



**Figure 4.6** **A simplified illustration of the basic LSH indexing scheme. A vector is hashed to a bucket by a locality preserving hash function $h$ and stored in a hash table.**

#### 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 $h : \mathbb{F} \rightarrow B$ that maps any two points $f, q \in \mathbb{F}$ to a bucket $b \in B$ such that Equations 4.7 and 4.8 hold, wherein $(\mathbb{F}, \mathfrak{d})$ must 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.

$$\mathfrak{d}(p, q) \leq R \Rightarrow h(f) = h(q) \text{ with a probability of at least } P_1 \tag{4.7}$$

$$\mathfrak{d}(p, q) \geq cR \Rightarrow h(f) = h(q) \text{ with a probability of at most } P_2 \tag{4.8}$$

In simple terms, a locality preserving hash function has a high probability for collision (i.e., mapping to the same bucket) if two features $f, q \in \mathbb{F}$ lie close to one another, i.e., $\mathfrak{d}(q, f) \leq R$, but exhibits a low probability for collision if $\mathfrak{d}(q, f) \geq cR$. A feature $f \in \mathbb{F}$ can then be hashed to a bucket and stored in a hash table with other vectors that lie close to it as 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 probabilistc 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].

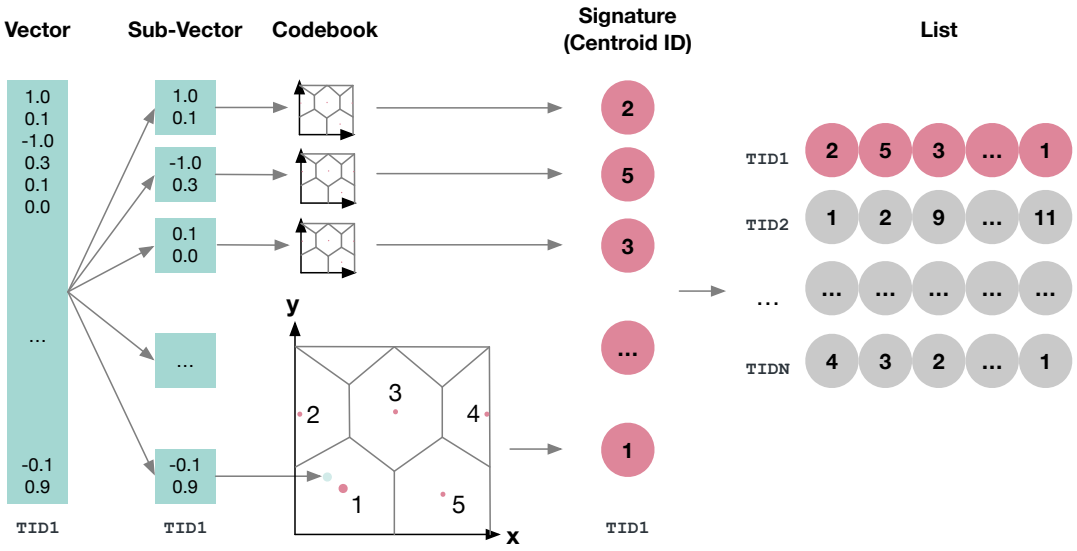



**Figure 4.7 A simplified illustration of the basic PQ indexing scheme. A vector is divided into smaller subvectors and each subvector is assigned to the nearest centroid in the codebook of the respective subspace, which produces the signature.**

#### 4.2.2.3 Product Quantisation (PQ)

The idea for PQ was proposed by [JDS10] and it describes a technique for quantising high-dimensional vectors. The main idea involves decomposition of the original vector space $\mathbb{R}^d$ into smaller subspaces $\mathbb{R}^s$ with $s < d$ and separate quantisation of each subspace. The quantisation is achieved by learning a codebook for every subspace through k-means clustering on a representative sample of the data. Upon indexing, all subvectors of each feature vector $f \in \mathbb{F}$ are assigned to the nearest centroid in the codebook for the respective subspace. Concatenation of the centroid indexes gives rise to a compact signature that can be stored in an 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 to approximate the distance between a query $q$ and a feature $f$ using a precalculated lookup table and inexpensive operations, which significantly reduces CPU costs. However, this speedup is bought by a distortion of the approximated distance, which may lead to errors if features lie very close to one another and makes PQ a suboptimal fit if exact distance values are required. [JDS10] also discusses several optimisations, e.g., encoding the difference between feature vector and centroid – the residual – and to use an additional, coarse quantiser to assign vectors to entries in an inverted file index (IVFPQ).

**Example 4.4 Size of PQ-Signature vs. original feature**

Let us assume $\mathbb{F} \subset \mathbb{R}^{256}$, i.e., a 256-dimensional vector space ($d = 256$). Assuming a `float` data type, we require 4 B per vector component, i.e., 1024 B 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 we create a codebook 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 B per vector to store an index entry.

### 4.2.3 Beyond Similarity Search

As we have argued in Section 2.1, the query workloads in real-world multimedia retrieval systems are often far more diverse than what we have described so far. In addition to one-shot queries, users find themselves searching, refining, browsing and exploring datasets with the aim to find a particular item [LKM+19; RGL+20]. This is exemplified by interactive retrieval competitions such as VBS [Sch19; LBS+18] or LSC [GJS+21], where participants are required to find items of interest in large, standardised data collections [BRS+19; RSB21] within a limited amount of time.

The various participants to the aforementioned campaigns have demonstrated interesting techniques that complement and replace mere similarity search. While the 2022 iteration was won by a system combining search using the CLIP model [RKH+21] and sophisticated browsing [HSJ+22], the team behind [KVM+20] has won the 2020 iteration of VBS with a system that relies on Self-organizing Map (SOM) [Koh90] to generate a clustered visualisation of the high-dimensonal data collection initialised by search input. Similarily, the system that won in 2016 used [BHM16] hierarchical graphs and visually sorted image maps to facilitate exploration without providing any classical search functionality. Generally, it was found that efficient browsing is as important for effective retrieval as an effective similarity search model [LKM+19]. Another comparison of systems at the LSC 2018 has shown, that a combination of functionality that complements similarity search is very common, including functions such as Boolean search on biometric data, facet filters and visual clustering [GSJ+19]. Hence, as argued before, multimedia retrieval goes way beyond mere similarity search but often relies on it to some extent.

*Information is the oil of the 21st century, and analytics is the combustion engine.*

— Peter Sondergaard, 2011

# 5

# Multimedia Data Management

The insight that special requirements must be addressed to accomodate multimedia data in traditional DBMS can be traced back to the 1990s [MS96; AN97] with many theoretical contributions but very little in terms of concrete (open-source) implementations. However, from a database research perspective, the topic received renewed attention with the publication of the *Lowell Database Research Self-Asessment* in 2005 [AAB+05] and the *Claremont Report on Database Research* in 2008 [AAB+08], both of which identified support for unstructured- and multimedia data as well as the integration of information retrieval and database functionality as important research directions to consider.

In contrast, multimedia retrieval has traditionally relied on highly optimised data structures for storage and retrieval that were typically tightly coupled with the retrieval system itself in a monolithic architecture, offering little to no support for advanced data management functionalty. It was not until the *Ten Research Questions for Scalable Multimedia Analytics* [JWZ+16] that the issue of scalable data management for multimedia analytics and a convergence with database research was raised in the face of growing data collections [BRS+19; RSB21]. Interestingly, many of the points mad by Jónsson et al. are somehow related to aspects listed in the 2014 *The Beckman Report on Database Research* [AAA+14], namely the focus on "end-to-end processing systems" and "data understanding", demonstrating, that the two fields do have much in common.

However, while the amalgamation of text retrieval with classical database functionality became a topic of interest, multimedia retrieval remained a field of research largely separated from that of database research with the exception of a few, isolated contributions that tried to converge specific aspects such as scoring and ranking [LCI+05; ZHC+06], extensions to relational algebra based on fuzzy logic [MT99] or the conception of a multimedia query language [BBZ12].

## 5.1 Early Multimedia Retrieval Systems

As an alternative to a monolithic architecture, some of the early multimedia retrieval systems relied on and potentially combined existing database systems and built the special requirements for multimedia retrieval on top in either a single software-component (2-tier architecture) or some type of middleware that could then be used by other systems (3-tier architecture). Examples that use the 2-tier approach include systems such as the video retrieval system OVID [OT93], the image retrieval system MARS [RHM97] or systems such as QBIC [FSN+95] or MUVIS [KCG+03]. Unfortunately, neither of these systems are publicly available and therefore could not be examined in detail.

A prominent example is IBM's QBIC system [FSN+95], which allowed for (visual) similarity search on image and video data. QBIC's system architecture indicates the existence of a dedicated database layer used for storing features. However, there is little detail as to what type of database was used and what tasks were executed at a database level. Judging from the architecture and description, however, it seems that the queries were rather processed by the "matching engine" and the database itself was only used for storage. What is noteworthy is, that QBIC strictly distinguished between "database population" and "database query" as two different phases that took place independently of one another. A separation rather atypical for tradtional database applications.

MUVIS [KCG+03] is another 2-tier multimedia retrieval system, which, in contrast to the aforementioned systems, combined multiple databases for storing image, video and audio information separately without, however, providing an explicit reason for doing so. Again, though, there is little detail as to what tasks fell to these database layers and which tasks were carried out by the browsing layer built on top of it. Interestingly, we can again observe the distinction between "indexing" and "retrieval" as two separate and independent phases in the system's lifecycle.

The *Garlic Approach* [CHS+95] proposed by IBM is an example of a 3-tier architecture. The Garlic middleware combines and integrates different, dedicated types of datastores to support heterogenous data under an extensible and modular query and runtime system. This system can then be used by other systems via standardised APIs. It was later used to integrate QBIC [CHN+95], which allowed for the combined search in visual and non-visual data. In a manner of speaking, Garlic can be seen as an early attempt at a polystore approach for multimedia data, like, for example, *Polypheny DB* [VSS18].

The 2-tier architecture involving classical databases for information and multimedia retrieval is still very common today, with a tendency towards the use of NoSQL systems [MSL+14; dSP+17]. Furthermore, there has been some work on retrofitting established DBMS for multimedia retrieval, either by using existing extension endpoints (e.g., user-defined data types or Used Defined Function (UDF)), by writing extensions or altering core system components [GdR+09; WLK+10; GAKS14; FC13; WLL+15; YLF+20].

## 5.2 Libraries for Multimedia Retrieval

Instead of tightly integrating multimedia retrieval techniques with systems that use them, the past two decades bore wittness to a trend towards open source libraries that could be (re-)used. The standardised ANNS benchmarking campaign known as *ANN-Benchmarks* [ABF17] [1], for example, lists and evaluates 25 individual libraries, such as, *Spotify Annoy* [2], ScaNN [GSL+20] by Google Research, *Hnswlib* [MY18], Facebook's FAISS [JDJ19] or Yahoo's NGT [Iwa16].

A noteworthy, early attempt at providing such a library for content-based image retrieval was Lucene Image Retrieval (LIRe) [LC08] – an open source Java library based on the Apache Lucene engine, which is typically used for fulltext search. LIRe was later used and extended to create *LIvRE* [dLG16], a similar project but for video retrieval. While both LIRe and LIvRE were an important step in the right direction, they unfortunately tightly integrated feature extraction and storage and were thus of very limited use when trying to use different types of features. Nonetheless, these project demonstrated Apache Lucene's capability of executing NNS. Consequently, Lucene based projects such as ElasticSearch [3] or OpenSearch [4] still provide NNS and ANNS through plugins and extensions such as Elastiknn [5].

An important step towards an engine for scalable NNS was the open source library Facebook AI Similarity Search (FAISS) [JDJ19]. FAISS brings together different methods for ANNS – for example, PQ [JDS10], HNSW [MY18] and LSH [IM98] based index structures – and provides SIMD and GPU support both for indexing and querying. While FAISS is not a multimedia database system per-se, it provides a formidable foundation for building one.

[1] See http://ann-benchmarks.com/, Accessed July 2022
[2] See https://github.com/spotify/annoy, Accessed July 2022
[3] See https://www.elastic.co/, Accessed July 2022
[4] See https://opensearch.org/, Accessed July 2022
[5] See https://github.com/erikbern/ann-benchmarks/, Accessed July 2022

However, even though all these projects are important and useful, they typically require integration into a system and do not provide an "out-of-the-box" solution that can be used via standardised interfaces such as SQL.

## 5.3 Vector Database Engines

The past few years have seen the conception of novel vector databases as either dedicated systems or extensions to existing ones. We refrain from presenting all these systems in detail and instead briefly name and describe a few, with the exception of one system that receives a more detailed discussion. Most of these systems have in common that they have been built as cloud-native applications with distribution in mind and that they are open source. For every system, we reference the respective GitHub repository as well as so the current version and the time of the last update to illustrate the activity in this particular domain.

**Weaviate (1.14.1, July 2022)** [6] is a vector search engine that allows the storage of vectors and attributes. Both the vector and attributes can be queried using the *GraphQL* query language and it relies on HNSW [MY18] for ANNS. Weaviate claims full CRUD support, i.e., changes to entries are possible, which however limits the types of index structures used (currently, a custom HNSW index is employed). The recent, major update to version 1.14.0 has added support for the Euclidean distance and the dot product.

**Vald (1.5.5, July 2022)** [7] is a vector search engine that relies on NGT for ANNS. It allows for similarity search as well as lookup's by identifier using a gRPC based query API. A unique selling point is the *asynchronous auto indexing* feature, which refreshes outdated indexes in the background in a non-blocking fashion, which allows for changes to the data.

**Qdrant (0.8.5, July 2022)** [8] is a similarity search engine that can be used to store high-dimensional vectors with additional attributes – called payload. It can be characterised as document database. Qdrant supports similarity search queries as well as filtering for payload attributes via either a gRPC or RESTful interface. Both payload attributes as well as vectors can be indexed and HNSW [MY18] is used for the latter. A query planner decides about query execution in case multiple index structures are employed.

[6] See https://github.com/semi-technologies/weaviate/, Accessed July 2022
[7] See https://github.com/vdaas/vald/, Accessed July 2022
[8] See https://github.com/qdrant/qdrant/, Accessed July 2022

### 5.3.1 Milvus

Milvus (2.0.2, April 2022) [WYG+21] is vector database system with its most recent 2.0 release dating back to January 2022. Milvus considers itself a "vector database built for scalable similarity search" and it relies on many of the libraries mentioned in Section 5.2. According to information on the official blog [9], it was built as a cloud-native application with high scalability and availability in mind. It offers support for functionalty that goes beyond mere NNS, such as, Boolean filtering as well as hybrid queries involving both similarity search and Boolean predicates on scalar data types as well as index support for both types of queries. In addition, Milvus is also capable of processing online changes to data (i.e., inserts and deletes).

Data in Milvus is organised into named collections, which can consist of multiple attributes and are distributed among multiple partitions. The logical data model and type system is similar to that of Cottontail DB, however, in addition to scalar types (e.g., `int`, `float`, `double` or `string`) it only allows for `float` or binary vectors. Milvus offers client libraries for Go, Java (and Kotlin), Python and Node, which offers a user-friendly facade over the gRPC-based query interface and allows for data definition, data management and querying.

Milvus provides fast NNS through a component named *Knowwhere*, which provides a standardised operations interface that abstracts away the underlying state-of-the-art libraries and techniques such as FAISS [JDJ19], Hnswlib [MY18] and Spotify's Annoy [10] and which offers suport for SIMD and GPU execution as well as various index structures. Dynamic data support for indexes is enabled by an eventual consistency model, wherein data is being (re-)indexed on a partition by partition basis, i.e., if changes occur, only the affected partitions must be rebuilt. This mechanism is combined with a MVCC-based snapshot isolation, which also allows for querying the state of the database at a specific point in time. Deletes are filtered on the fly during query time based on *tombstone* markers in the database, a concept known from Apache Cassandra.

It is important to mention that while data held by Milvus resides on disk, a collection must be loaded into a main memory in its entirety in order to be abel to query it. Therefore, for Milvus to be able to execute NNS or Boolean fitlering, all the data (or at least the indexes), must fit into the combined main memory of the available query nodes.

[9] See https://milvus.io/blog/, Accessed July 2022
[10] See https://github.com/spotify/annoy/, accessed in June 2022

## 5.4 Databases for Data Science

With the start of the NoSQL movement some researchers began to address the shortcomings of the relational data model in the context of scientific applications and proposed different architectures, such as array databases. Early examples include SimDB [SAA+10] or SciDB [SBP+11; SBZ+13]. The topic of science oriented databases has has been listed as highly-relevant in the 2020 database research self-assessment report, due to the continued trend towards data driven applications, data science and data analytics [AAA+20]. In addition to challenges in the domain of data governance, one focus that was identified is the convergence between linear algebra and relational algebra in an efficient and effective framework [LGG+18], an effort that could also greatly benefit multimedia retrieval and analytics applications [BDO95]. Another area of interest is the use of specialised hardware such as FPGAs or GPUs, to accelerate query evaluation.

## 5.5 vitrivr, ADAM and ADAM$_{pro}$

To the best of our knowledge, the open source *vitrivr* video retireval [RGT+16] and later multimedia retrieval [GRS19a] system was one of the first systems to advertise a 3-tier architecture that included a specialised multimedia database in addition to a dedicated feature extraction and fusion engine and a user interface. The first generation database layer came in the form of ADAM [GAKS14], which can be seen as an early attempt at bridging the gap between databases and multimedia retrieval. At its core, ADAM was an extension to PostgreSQL – called *dbADAM* – that added support for NNS and VAF-based indexes [WSB98]. In that, it is comparable to projects like *pgvector* [11], *pgANN* [12] and PASE [YLF+20], which however use more recent index structures such as PQ [JDS10] and HNSW [MY18]. The *dbADAM* component was complemented by *wsADAM*, which could orchestrate query execution over multiple PostgreSQL instances using the MapReduce paradigm [DG08]. ADAM's successor ADAM$_{pro}$ [GS16] tried to address these limitations with a multi-model (polyglot) database system approach that allowed for query execution on different storage engines and distribution of data and computation using *Apache Spark*, a large-scale data analytics platform. ADAM$_{pro}$ still relied on PostgreSQL for storage of classical, relational data but used *Apache Parquet* to store and access feature vectors.

[11] See https://github.com/pgvector/pgvector/, Accessed July 2022
[12] See https://github.com/netrasys/pgANN/, Accessed July 2022

The work on $\text{ADAM}_{\text{pro}}$ was complemented by a theoretical model [Gia18]: Giangreco et al. demonstrated that for a database system to be able to support similarity search given the relational model 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 $\tau_{\delta(\cdot,\cdot),a,q}(\mathcal{R})$ that can be used to express similarity search. According to the definition, such an operation introduces an implicit attribute in the underlying relation $\mathcal{R}$, which in turn induces an ascending ordering of the tuples. Two variants of $\tau$ were proposed, namely $\tau^{kNN}_{\delta(\cdot,\cdot),a,q}$ (NNS), and $\tau^{\epsilon NN}_{\delta(\cdot,\cdot),a,q}$ (range search), which select tuples by their cardinality $k$ or a maximum cut-off distance $\epsilon$ respectively.

While postulating two new, relational operators is well within tradition of how relational algebra has been extended over the years (see Section 3.2.3), it comes with certain limitations. In its proposed form, $\tau$ addresses several functions at once: It (i) specifies the distance function that should be evaluated, (ii) generates an implicit distance attribute on the underlying relation $\mathcal{R}$, (iii) imposes an ascending ordering and limits the cardinality of $\mathcal{R}$ based on a predicate or a specified limit.

While being straightforward to implement, the amalgamation of all this functionality into a single operation is very specifically tailored to the use case of NNS and range search and only of limited value when considering more general distance-based query operations. If one were to, for example, obtain the $k$ farthest neighbours rather than the $k$ nearest neighbours, as necessary when doing MIPS or obtaining negative examples, they would have to either change the distance function, define new operators or extend the definition of $\tau$.

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

$$\sigma(\tau^{kNN}_{\delta(\cdot,\cdot),a,q}(\mathcal{R})) \neq \tau^{kNN}_{\delta(\cdot,\cdot),a,q}(\sigma(\mathcal{R})) \tag{5.1}$$

The left-hand side of Equation (5.1) filters the results of a kNN-search on $\mathcal{R}$, thus returning $n \leq k$ results, wherein $n = k$ only if $\sigma$ matches all tuples. The right-hand side of Equation (5.1) 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 $\sigma$ selects fewer than $k$ tuples.

Part III

# Dynamic Multimedia Data Management

*The science of operations, as derived from mathematics more especially, is a science of itself, and has its own abstract truth and value.*

— Ada Lovelace, 1843

# 6

# Modelling a Database for Dynamic Multimedia Data

We have argued in Chapter 2 that multimedia retrieval, analysis and analytics workloads (and workloads that exhibit similiar characteristics) pose specific requirements on a database system. In this chapter, we focus on four aspects and derive a formal model to accomodate them on a systems level:

**Support for Complex Data Types** Multimedia workloads process features that are often real-valued, high-dimensional vectors but can be any type of mathematical object (e.g., complex numbers). A multimedia DBMS must support these objects as first-class citizens (see Requirement 2.5).

**Support for Distance Computation** The notion of proximity between features plays a crucial role in all types of multimedia workloads, which involves the evaluation of functions. A multimedia DBMS must be able to execute these funcitons efficiently, using the available data structures and execution methods (see Requirement 2.6).

**Dynamic Data** Data is usually subject to constant change and a multimedia DBMS must be able to propagate these changes to all the relevant data structures while maintaining an internal consistency model (see Requirement 2.3 and Requirement 2.1).

**Quality vs. Time** Multimedia queries often trade retrieval quality against speed or vice-versa, e.g., when using a high-dimensional indexes. A multimedia DBMS must make these decisions explicit and allow the user to express their perference at different levels of the system (see Requirement 2.4).

To address these requirements, we propose and formalise a set of functionality, which will be specified in the next sections. First, we propose a notion of *generalised, proximity based queries* as an extension to the relational model. Second, we describe a *cost-model* that takes the notion of quality of results into account. And finally, we introduce the formal requirements to enable high-dimensional indexes to cope with dynamic data and derive a model for maintaining such indexes. All these contributions can be seen as extensions to the functionality provided by a traditional DBMS. To establish a frame of reference, we assume the relational model and ACID properties for the model DBMS. However, the introduced concepts should generalise to other data- and consistency models.

## 6.1 Generalised Proximity Based Operations

Starting with the vector 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 6.1.

**Definition 6.1 Proximity Based Query**

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

Definition 6.1 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 $\mathcal{D}_f$ query and probing attribute belong to nor how the distance is being used within the query.

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 (e.g., range search) or aggregation based on the computed value (e.g., clustering). 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.

### 6.1.1 Revisiting Distance Computation

Since the choice of the distance function $\mathfrak{d}$ is a crucial part of any proximity based query, it is worth revisiting its definition. Again, starting with the metric space modell [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 $\mathfrak{d}$ is assumed to be $\mathbb{R}^{\mathtt{dim}} \times \mathbb{R}^{\mathtt{dim}}$, 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 $\mathfrak{d}$ is assumed to be $\mathbb{R}_{\geq 0}$, hence, the generated distance value is a positive, real number.

3. $(\mathcal{D}_f, \mathfrak{d})$ 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 6.1, we realise that it is not reasonable to impose a general restriction of the domain of the distance function to $\mathbb{R}^{\mathtt{dim}}$ with $\mathtt{dim} \in \mathbb{N}^+$

**Example 6.1 Maximum Inner Product Search for MRF**

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

Obviously, this limitation of the definition of $\mathfrak{d}$ can be easily remediated simply by extending the set of supported data domains $\mathbb{D}$ by $\mathbb{C}^{\mathtt{dim}}$ similarily to how it was done for $\mathbb{R}^{\mathtt{dim}}$ 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 6.2, we see that restricting oneself to binary functions with $\mathbb{R}_{\geq 0}$ is also too limiting when facing certain practical scenarios.

**Example 6.2 Distance Between a Vector and a Hyperplane**

To find positive and negative examples $a_i \in \mathcal{D}_f \subset \mathbb{R}^{\mathtt{dim}}$ 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 $\mathbf{q}, \mathbf{x} \in \mathbb{R}^{\mathtt{dim}}$ and $b \in \mathbb{R}$. The distance function is then given by:

$$\mathfrak{d}(\mathbf{a}_i, \mathbf{q}, b) = \frac{\left\|\mathbf{q}^T\mathbf{a_i} + b\right\|}{\left\|\mathbf{q}\right\|} \tag{6.1}$$

$\mathfrak{d}$ now has the form $\mathfrak{d}\colon \mathbb{R}^{\mathtt{dim}} \times \mathbb{R}^{\mathtt{dim}} \times \mathbb{R} \to \mathbb{R}$. Hence, $\mathfrak{d}$ is no longer a binary but a ternay 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 idealised 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 [SB11; 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 generalised multimedia database application supporting proximity based queries.

In contrast, there is actually good reason to assume the codomain of a distance function $\mathfrak{d}$ to lie in $\mathbb{R}$. On the one hand, it is 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 downstream operations such as sorting, ranking or clustering.

To summarize, we have established that, 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 we must keep 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 6.2.

**Definition 6.2 Distance Function Class**

A DFC $\hat{\mathfrak{d}}\colon \mathcal{D}_f \times \mathcal{D}_f \times \mathcal{D}_1 \ldots \times \mathcal{D}_N \to \mathbb{R}$ is an N-ary but at least binary function $\hat{\mathfrak{d}}(p, q, s_1, \ldots, s_n)$ that outputs a distance $d \in \mathbb{R}$ between a probing argument $p \in \mathcal{D}_f$ and a query argument $q \in \mathcal{D}_f$ using a defined number of *support arguments* $s_{k\in\mathbb{N}}$ from any of the data domains in $\mathbb{D}$.
As a convention for notation, the probing argument $p$ of a DFC $\hat{\mathfrak{d}}(p, q, s_1, ..., s_n)$ will always come first, followed by the query argument $q$, again followed by its support arguments $s_k$ in no particular order.

**Example 6.3 Examples of Distance Function Classes**

A merely illustrative yet widely used example of a DFC would be the family of *Minkowski Distances* $\hat{\mathfrak{d}}_M\colon \mathbb{R}^{\mathtt{dim}} \times \mathbb{R}^{\mathtt{dim}} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$. In this context, support argument $m$ is used to convert the higher-order function to a classical distance function, such as, the Manhattan ($L_1$) or Euclidean ($L_2$) distance:

$$\hat{\mathfrak{d}}_M(\mathbf{p}, \mathbf{q}, m) = \left( \sum_{i=1}^{n} \mid p_i - q_i \mid^m \right)^{\frac{1}{m}} \tag{6.2}$$

$$\mathfrak{d}_{L1}(\mathbf{p}, \mathbf{q}) = \hat{\mathfrak{d}}_M(\mathbf{p}, \mathbf{q}, 1) = \sum_{i=1}^{n} \mid p_i - q_i \mid \tag{6.3}$$

$$\mathfrak{d}_{L2}(\mathbf{p}, \mathbf{q}) = \hat{\mathfrak{d}}_M(\mathbf{p}, \mathbf{q}, 2) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{6.4}$$

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. For example, the following distance function is used to obtain a dissimilarity between poses using the joint locations of an 18-keypoint skeleton (more details are presented in [HAG+22]):

$$\hat{\delta}(\mathbf{p}, \mathbf{q}, \mathbf{w}_q, \mathbf{w}_f, s) = \sum_{i=1}^{n} \mid (p_i - q_i) \mid \min(w_{q,i}, w_{f,i}) + 2\pi(s - \sum_{i=1}^{n} \min(w_{q,i}, w_{f,i})) \tag{6.5}$$

The query and support arguments $\mathbf{p}$ and $\mathbf{q}$ are the actual, normalised key-points and are accompanied by two weight-vectors $\mathbf{w}_q$ and $\mathbf{w}_f$, that encode the presence or absence of a joint in a scene. To normalise for missing joints, a normalisation factor based on the weight-sum $s$ is employed.

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 (6.6).

$$\texttt{SIG}(\hat{\mathfrak{d}}) = (\texttt{NAME}, \mathcal{D}_f, \mathcal{D}_f, \mathcal{D}_1, ...\mathcal{D}_{N-2}) \tag{6.6}$$

In the context of a concrete execution plan for a proximity based query, we observe the query argument effectively remains constant, while the probing argument changes as tuples are being evaluated. 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, the function used during query execution comes with an arity of at most $N-1$. We call this the *implementation* of $\hat{\mathfrak{d}}$ as denoted by Equation (6.7).

$$\mathfrak{d} = \texttt{IMP}(\hat{\mathfrak{d}}) \ni \texttt{SIG}(\mathfrak{d}) \subset \texttt{SIG}(\hat{\mathfrak{d}}) \tag{6.7}$$

With this idea of a DFC and its (partial) implementations 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 6.1, 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.

**Variable and Constant Arguments** In the context of a proximity based query, every attribute of a DFC can either be regarded as *constant* or *variable*. That is, as elements in the database are iterated during query execution, they either remain the same or assume new values. This can be used for optimisation.

**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 $p, q \in \mathcal{D}_f$. Furthermore, we require that the query argument is always constant, wheras the probing argument is always variable.

**Scalarity of Codomain** A DFC and its implementations always produce a single, scalar output $d \in \mathbb{R}$ to quantify the distance. Depending on whether a function generates a distance or a similarity score, a DBMS could use special, annotated types (e.g., `DISTANCE` or `SCORE`) to allow for specialised downstream operations.

As an aside, we must address the role of dimensionality $\texttt{dim}$ in the case of data domains that are subsets of vector spaces, i.e., $\mathcal{D}_f \subset \mathbb{R}^{\texttt{dim}}$ or $\mathcal{D}_f \subset \mathbb{C}^{\texttt{dim}}$. 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 $\mathcal{D}_f$ and thus a constant in the context of a query.

### 6.1.2 Extending the Relational Model

Using Definition 6.2, 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 $\mathcal{D}_f$ is required, such as but not limited to $\mathbb{R}^{\texttt{dim}}$ or $\mathbb{C}^{\texttt{dim}}$.

#### 6.1.2.1 Extended Projection and DFCs

We use the idea of an extended projection $\pi_P$ – as proposed by [GHQ95; GUW09] and introduced in Section 3.2.3 – wherein $P$ denotes a list of projection expressions $p$ 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 6.3.

**Definition 6.3 Extended Projection $\pi_P$**

Let $\mathcal{R}$ be a N-ary relation over attributes $\mathcal{A} \in \texttt{SCH}(\mathcal{R})$ with tuples $t \in \mathcal{R}$. We call $\pi_P$ the *extended projection* with $K$ projection elements $p_i \in P$, i.e., $P = \{p_1, p_2, \ldots p_K\}$, with $p_i \in \texttt{SCH}(\mathcal{R}) \vee p_i \in \mathcal{D}_{\mathcal{L}} \vee p_i \in \mathcal{F}$, where $\mathcal{F}$ denotes a set of all functions $f_i \colon \mathcal{D}_1 \times \mathcal{D}_2 \ldots \mathcal{D}_M \rightarrow \mathcal{D}_{\texttt{out}}$ known to the DBMS and $\mathcal{D}_{\mathcal{L}} \in \mathbb{D}$ denotes a domain of literal values. The extended projection is then defined as follows:

$$\pi_P(\mathcal{R}) = \{t\,[P] : t \in \mathcal{R}\} \text{ with } t\,[P] = \bigcup_{p_i \in P} \begin{cases} t\,[p_i] & \text{if } p_i \in \texttt{SCH}(\mathcal{R}) \\ p_i & \text{if } p_i \in \mathcal{D}_{\mathcal{L}} \\ p_i(t\left[P'\right]) & \text{if } p_i \in \mathcal{F} \end{cases}$$

Note, that the domain of every function $f_i \in \mathcal{F}$ is again, an implicit, extended projection $t\left[P'\right]$, where $P'$ 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 $\mathcal{F}$, the evaluation of a DFC as part of a query is given by Definition 6.4 as illustrated in Example 6.4.

**Definition 6.4 Distance Function Class in Extended Projection $\pi_P$**

Let $\hat{\mathfrak{d}}\colon \mathcal{D}_f \times \mathcal{D}_f \times \mathcal{D}_1 \ldots \times \mathcal{D}_{N-2} \rightarrow \mathbb{R}$ be a N-ary DFC. The *extended projection* $\pi_{\hat{\mathfrak{d}}(p_1,p_2,\ldots p_N)}(\mathcal{R})$ with $p_i \in \mathtt{SCH}(\mathcal{R}) \vee p_i \in \mathcal{D}_{\mathcal{L}} \vee p_i \in \mathcal{F}$ denotes the evaluation of $\hat{\mathfrak{d}}$ using $N$ arguments. Applying $\pi_{\hat{\mathfrak{d}}(\cdot)}$ on a M-ary relation $\mathcal{R}$ introduces a new distance attribute $\mathcal{A}_d$, i.e., $\mathtt{SCH}(\pi_{\hat{\mathfrak{d}}(\cdot),\mathcal{A}_1,\ldots,\mathcal{A}_M}(\mathcal{R})) = \mathtt{SCH}(\mathcal{R}) \cup \{\mathcal{A}_d\}$

**Example 6.4 Obtaining a Distance using Extended Projection**

The following table lists the schema and extent of $\mathcal{R}_{\mathtt{painting}}$, which now contains a column of feature vectors $\mathcal{A}_{\mathtt{feature}}$ in addition to the scalar attributes, i.e., $\mathcal{D}_{\mathtt{feature}} \subset \mathbb{R}^3$.

| $\mathcal{R}_{\mathtt{painting}}$ | $\mathcal{A}^*_{\mathtt{title}}$ | $\overline{\mathcal{A}}_{\mathtt{artist}}$ | $\mathcal{A}_{\mathtt{painted}}$ | $\mathcal{A}_{\mathtt{feature}}$ |
|---|---|---|---|---|
| $t_1$ | Mona Lisa | Leonardo da Vinci | 1506 | $[2.0, 7.0, -8.0]$ |
| $t_2$ | The Starry Night | Vincent van Gogh | 1889 | $[1.0, 0.0, 3.0]$ |
| $t_3$ | Las Meninas | Diego Velázquez | 1665 | $[-1.0, 3.0, 9.0]$ |

The extended projection $\pi_P(\mathcal{R})$ with $P = (\mathcal{A}_{\mathtt{title}}, \mathfrak{d}([0,0,0], \mathcal{A}_{\mathtt{feature}}))$, projects onto the attribute $\mathcal{A}_{\mathtt{title}}$ and the distance $\mathfrak{d} \in \mathcal{F}$ between query argument $q = [0,0,0] \subset \mathbb{R}^3$ and probing argument $\mathcal{A}_{\mathtt{feature}}$. This is the relational algebra equivalent to the following SQL query and leads to $\mathcal{R}_{\mathtt{result}}$ (assuming $\mathfrak{d}$ to be the the Manhattan distance).

```
select title, d([0,0,0],feature) from paintings
```

| $\mathcal{R}_{\mathtt{result}}$ | $\mathcal{A}^*_{\mathtt{title}}$ | $\mathcal{A}_{\mathtt{distance}}$ |
|---|---|---|
| $t_1$ | Mona Lisa | 17.0 |
| $t_2$ | The Starry Night | 4.0 |
| $t_3$ | Las Meninas | 13.0 |

From Definition 6.4, we can follow that the evaluation of multiple DFCs in an extended projection (e.g., to support aggregate NNS [PTM+05]) or the combination of attribute projections with the evaluation of DFCs are also allowed.

With the proposed, extended projection, the obtained distance value becomes an explicit attribute $\mathcal{A}_d$ in the relation $\mathcal{R}' = \pi_{\hat{\delta}(\cdot)}(\mathcal{R})$, 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, where the distance is often converted to a similarity score by some additional correspondence function.

#### 6.1.2.2 Ranked Relations

To support sorting by the obtained distance (or any other attribute for that matter), we use a variant of the ranked relation as proposed by [LCI+05] and described in Section 3.2.3. A ranked relation $\mathcal{R}^O$ exhibits an ordering of tuples $t \in \mathcal{R}^O$ induced by $O$. In contrast to [LCI+05] and more in line with [GUW09], we assume $O$ to be a sequence of attributes by which the relation should be sorted[1], as specified in Definitions 6.5 & 6.6 and shown in Example 6.5 .

**Definition 6.5 Ranked Relation $\mathcal{R}^O$**

Let $\mathcal{R}$ be a N-ary relation over attributes $\mathcal{A} \in \mathtt{SCH}(\mathcal{R})$ with $M$ tuples $t_m \in \mathcal{R}$. We call $\mathcal{R}^O$ a *ranked relation* with $K$ ranking attributes $O = (o_1, o_2, \ldots, o_K)$, such that $o_a = (\mathcal{A}_a, \preceq_a)$ with $\mathcal{A}_a \in \mathtt{SCH}(\mathcal{R})$ and $\preceq_a \in \{\preceq_\uparrow, \preceq_\downarrow\}$. $O$ induces a non-strict, total ordering of tuples $t_m$ using the binary, lexicographic ranking relation $\preceq$:

$$t_i \preceq t_k = \bigvee_{o \in O} t_i \preceq_o t_k, \text{with} \begin{cases} t_i \preceq_o t_k \leftrightarrow t_i[\mathcal{A}_o] \leq t_k[\mathcal{A}_o] & \text{if } \preceq_o = \preceq_\uparrow \\ t_i \preceq_o t_k \leftrightarrow t_i[\mathcal{A}_o] \geq t_k[\mathcal{A}_o] & \text{if } \preceq_o = \preceq_\downarrow \end{cases}$$

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 $\mathtt{SCH}(\mathcal{R})$. It is retained or induced as operators are being applied to a relation. Furthermore, if a relation does not exhibit a specific ordering, then $O$ is empty and $\mathcal{R}^O = \mathcal{R}^\emptyset = \mathcal{R}$.

**Definition 6.6 Order Operation $\omega_O$**

Let $\mathcal{R}$ be a N-ary relation over attributes $\mathcal{A} \in \mathtt{SCH}(\mathcal{R})$ and let $O$ be list of ranking attributes with $o[0] \in \mathtt{SCH}(\mathcal{R}) \forall o \in O$. The *relational order operator* imposes order $O$ on relation $\mathcal{R}$, overriding any previous ordering:

$$\omega_O(\mathcal{R}) = \mathcal{R}^O$$

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

**Example 6.5 Applying Order using the Order Operation $\omega_O$**

The following table lists the schema, extent and rank of ranked relation $\mathcal{R}^{\emptyset}_{\texttt{painting}}$, which now contains a attribute $\mathcal{A}_{\texttt{distance}}$ (e.g., obtained by executing a proximity based query) and does not exhibit an explicit ordering.

| $\mathcal{R}^{\emptyset}_{\texttt{painting}}$ | $\mathcal{A}^{*}{}_{\texttt{title}}$ | $\overline{\mathcal{A}}_{\texttt{artist}}$ | $\mathcal{A}_{\texttt{painted}}$ | $\mathcal{A}_{\texttt{distance}}$ |
|---|---|---|---|---|
| $t_1$ | Mona Lisa | Leonardo da Vinci | 1506 | 17.0 |
| $t_2$ | The Starry Night | Vincent van Gogh | 1889 | 4.0 |
| $t_3$ | Las Meninas | Diego Velázquez | 1665 | 13.0 |
| $t_4$ | Mars and Venus | Sandro Botticelli | 1483 | 17.0 |

The order operation $\omega_O(\mathcal{R}_{\texttt{painting}})$ with $O = ((\mathcal{A}_{\texttt{distance}}, \preceq_{\downarrow}), (\mathcal{A}_{\texttt{title}}, \preceq_{\uparrow}))$, which orders by $\mathcal{A}_{\texttt{distance}}$ in descending and $\mathcal{A}_{\texttt{title}}$ in ascending order. This is the relational algebra equivalent to the following SQL query and leads to $\mathcal{R}_{\texttt{result}}$.

```
select * from paintings order by score desc,
title asc
```

| $\mathcal{R}^{O}_{\texttt{painting}}$ | $\mathcal{A}^{*}{}_{\texttt{title}}$ | $\overline{\mathcal{A}}_{\texttt{artist}}$ | $\mathcal{A}_{\texttt{painted}}$ | $\mathcal{A}_{\texttt{distance}}$ |
|---|---|---|---|---|
| $t_1$ | Mars and Venus | Sandro Botticelli | 1483 | 17.0 |
| $t_2$ | Mona Lisa | Leonardo da Vinci | 1506 | 17.0 |
| $t_3$ | Las Meninas | Diego Velázquez | 1665 | 13.0 |
| $t_4$ | The Starry Night | Vincent van Gogh | 1889 | 4.0 |

On a practical note, it must be mentioned, that until a query is evaluated, the ordering induced by $O$ must not necessarily be materialised. To determine the position of a tuple within a relation upon materialisation, we assume the existence of a ranking function.

$$\texttt{RANK}_O : \mathcal{R} \rightarrow [1, M] \text{ with } M = |\mathcal{R}| \tag{6.8}$$

The ranking function assigns a position $m \in [1, M]$ to a tuple based on $O$, i.e., the first tuple $t_1$ gets position 1, second tuple $t_2$ position 2 etc. Unordered relations also exhibit an internal ordering of tuples – the *natural ordering* of $\mathcal{R}$ – which is not necessarily related to any observable property. Consequently, RANK is also defined for such relations, even though positions may appear arbitrary.

#### 6.1.2.3 s,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 *s,k-selection* operator $\lambda_{s,k}(\mathcal{R})$, which skips $s$ tuples and limits the cardinality to $k$ tuples according to Definition 6.7 and as demonstrated in Example 6.6.

**Definition 6.7 s,k-selection Operation $\lambda_{s,k}$**

Let $\mathcal{R}$ be a N-ary relation over attributes $\mathcal{A} \in \mathtt{SCH}(\mathcal{R})$ and let $s, k \in \mathbb{N}$ and $s < k$. The s,k-selection operation $\lambda_{s,k}$ selects tuples based on ranking as follows:

$$\lambda_{s,k}(\mathcal{R}^O) = \{t \in \mathcal{R}^O : s \leq \mathtt{RANK}_O(t) \leq k\}$$

The s,k-selection operation is inherently sensitive to ranking since it selects tuples based on it. Furthermore and since oftentimes, $s = 0$ we define a short-cut $\lambda_k = \lambda_{0,k}$ to express a simple selection to $k$ entries without any skipping.

**Example 6.6 Selection Based on Ranking using $\lambda_{s,k}$**

The following table lists schema, extent and ranking of ranked relation $\mathcal{R}^{\emptyset}_{\mathtt{painting}}$, which does not exhibit an explicit ordering.

| $\mathcal{R}^{\emptyset}_{\mathtt{painting}}$ | $\mathcal{A}^*_{\mathtt{title}}$ | $\overline{\mathcal{A}}_{\mathtt{artist}}$ | $\mathcal{A}_{\mathtt{painted}}$ |
|---|---|---|---|
| $t_1$ | Mars and Venus | Sandro Botticelli | 1483 |
| $t_2$ | Mona Lisa | Leonardo da Vinci | 1506 |
| $t_3$ | The Starry Night | Vincent van Gogh | 1889 |
| $t_4$ | Las Meninas | Diego Velázquez | 1665 |

$\lambda_{0,3}(\mathcal{R}_{\mathtt{painting}})$ limits the result to the first two entries. It is the relational algebra equivalent to the following SQL query and produces $\mathcal{R}_{\mathtt{result}}$.

```
select * from paintings limit 3
```

| $\mathcal{R}^{\emptyset}_{\mathtt{result}}$ | $\mathcal{A}^*_{\mathtt{title}}$ | $\overline{\mathcal{A}}_{\mathtt{artist}}$ | $\mathcal{A}_{\mathtt{painted}}$ |
|---|---|---|---|
| $t_1$ | Mars and Venus | Sandro Botticelli | 1483 |
| $t_2$ | Mona Lisa | Leonardo da Vinci | 1506 |
| $t_3$ | The Starry Night | Vincent van Gogh | 1889 |

#### 6.1.2.4 Combining $\pi_P$, $\omega_O$, $\lambda_k$

The proposed operations can now be freely combined to form proximity based queries. In Example 6.7 we demonstrate this for kNN and range search. A list of additional types of queries is provided in Table 6.1.

**Example 6.7 kNN and $\epsilon$NN using Extended Relational Algebra**

The following table lists the schema, extent and rank of $\mathcal{R}^{\emptyset}_{\texttt{painting}}$, with their title $\mathcal{A}_{\texttt{(t)itle}}$ and a feature $\mathcal{A}_{\texttt{(f)eature}}$. Let further $P = (\mathcal{A}_{\texttt{title}}, \mathfrak{d}([0,0,0], \mathcal{A}_{\texttt{feature}}))$ be a list of projections, $\mathfrak{d} \in \mathcal{F}$ be a distance between query argument $q = [0,0,0] \in \mathcal{D}_f \subset \mathbb{R}^3$ and probing argument $\mathcal{A}_{\texttt{feature}}$.

| $\mathcal{R}^{\emptyset}_{\texttt{(p)ainting}}$ | $\mathcal{A}^{*}_{\texttt{(t)itle}}$ | $\overline{\mathcal{A}}_{\texttt{(a)rtist}}$ | $\mathcal{A}_{\texttt{(p)ainted}}$ | $\mathcal{A}_{\texttt{(f)eature}}$ |
|---|---|---|---|---|
| $t_1$ | Mona Lisa | Leonardo da Vinci | 1506 | [2.0, 7.0, −8.0] |
| $t_2$ | The Starry Night | Vincent van Gogh | 1889 | [1.0., 0.0, 3.0] |
| $t_3$ | Las Meninas | Diego Velázquez | 1665 | [−1.0, 3.0, 9.0] |

kNN-search can be expressed as $\lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_P(\mathcal{R})))$ with $k = 2$. This is the relational algebra equivalent to the following SQL query and leads to $\mathcal{R}_{\texttt{result}}$ (assuming $\mathfrak{d}$ to be the Manhattan distance).

```
      select title, d(feature, [0.0,0.0,0.0]) as dst
from paintings order by dst limit 2
```

| $\mathcal{R}_{\texttt{result}}$ | $\mathcal{A}^{*}_{\texttt{title}}$ | $\mathcal{A}_{\texttt{distance}}$ |
|---|---|---|
| $t_1$ | Mona Lisa | 17.0 |
| $t_2$ | Las Meninas | 13.0 |

Range search (or $\epsilon$NN) can be expressed as $\sigma_{\mathcal{A}_d > \epsilon}(\pi_P(\mathcal{R}))$ with and $\epsilon = 15.0$. This is the relational algebra equivalent to the following SQL query and leads to $\mathcal{R}_{\texttt{result}}$ (assuming $\mathfrak{d}$ to be the Manhattan distance).

```
      select title, d(feature, [0.0,0.0,0.0]) as dst
from paintings where dst > 15.0 order by dst
```

| $\mathcal{R}_{\texttt{result}}$ | $\mathcal{A}^{*}_{\texttt{title}}$ | $\mathcal{A}_{\texttt{distance}}$ |
|---|---|---|
| $t_1$ | Mona Lisa | 17.0 |

**Table 6.1 Proximity based queries supported by the extended relational algebra.**

| Name | Result | Algebraic Form |
|---|---|---|
| NNS / kNN | $\mathcal{R}^{\mathcal{A}_d\uparrow}$ | $\lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_{\mathcal{A}_y,\mathfrak{d}(\mathcal{A}_p)}(\mathcal{R}_p)))$ |
| FNS / kFN | $\mathcal{R}^{\mathcal{A}_d\downarrow}$ | $\lambda_k(\omega_{\mathcal{A}_d\downarrow}(\pi_{\mathcal{A}_y,\mathfrak{d}(\mathcal{A}_p)}(\mathcal{R}_p)))$ |
| $\epsilon$NN / Range Search | $\mathcal{R}^{\mathcal{A}_d\uparrow}$ | $\omega_{\mathcal{A}_d\uparrow}(\sigma_{\mathcal{A}_d\leq\epsilon}(\pi_{\mathcal{A}_y,\mathfrak{d}(\mathcal{A}_p)}(\mathcal{R}_p)))$ |
| NNS w. Selection | $\mathcal{R}^{\mathcal{A}_d\uparrow}$ | $\lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_{\mathcal{A}_p,\mathfrak{d}(\mathcal{A}_p)}(\sigma_{\mathcal{A}_y=1889}(\mathcal{R}_p))))$ |
| Group NNS & Aggregate NNS [PST+04; PTM+05] | $\mathcal{R}^{\mathcal{A}_d\uparrow}$ | $\lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_{f(\mathfrak{d}_1(\mathcal{A}_p),\ldots,\mathfrak{d}_n(\mathcal{A}_p))}(\mathcal{R}_p)))$ |
| Batch NNS [SHS+08] | $\mathcal{R}^{\emptyset}$ | $\lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_{\mathfrak{d}_1(\mathcal{A}_p)}(\mathcal{R}_p))) \cup \lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_{\mathfrak{d}_2(\mathcal{A}_p)}(\mathcal{R}_p)))$ |
| Multi Order | $\mathcal{R}^{\mathcal{A}_{d1}\uparrow,\mathcal{A}_{d2}\uparrow}$ | $\omega_{\mathcal{A}_{d1}\uparrow,\mathcal{A}_{d2}\uparrow}(\pi_{\mathfrak{d}_1(\mathcal{A}_p),\mathfrak{d}_2(\mathcal{A}_p)}(\mathcal{R}_p))$ |
| Aggregation by Distance | $\mathcal{R}^{\emptyset}$ | $\gamma_{\texttt{MEAN}}(\pi_{\mathfrak{d}(\mathcal{A}_p)}(\mathcal{R}_p))$ |

#### 6.1.2.5 Algebraic Properties of $\pi_P$, $\omega_O$, $\lambda_k$

Since a query optimiser of a multimedia DBMS must be able to manipulate the operators that were introduced before, there is a series of algebraic properties that must be taken into account. For unary operators, we must consider *commutativity*, i.e., whether changing the order of application leads to the same result. For binary operators, we must consider the *commutativity of operands* and *distributivity* with respect to other operations. The identities are provided in Equation (6.9).

$$\begin{aligned}
\texttt{OP}_1(\texttt{OP}_2(\mathcal{R})) &= \texttt{OP}_2(\texttt{OP}_1(\mathcal{R})) && \text{(commutativity of operators)}\\
\texttt{OP}_1(\texttt{OP}_2(\mathcal{R}_L,\mathcal{R}_R)) &= \texttt{OP}_2(\texttt{OP}_1(\mathcal{R}_L),\texttt{OP}_1(\mathcal{R}_R)) && \text{(distributivity of operators)}\\
\texttt{OP}(\mathcal{R}_L,\mathcal{R}_R) &= \texttt{OP}(\mathcal{R}_R,\mathcal{R}_L) && \text{(commutativity of operands)}\\
\texttt{OP}(\texttt{OP}(\mathcal{R})) &= \texttt{OP}(\mathcal{R}) && \text{(idempotence of operators)}
\end{aligned} \tag{6.9}$$

In addition, to account for ranked relations, we must distinguish between relational operators that are either *order retaining* (order of the input remains the same), *order inducing* (order of the input is changed). Furthermore, we call operators that do not commute with order inducing operators *order sensitive*.

It follows directly from its definition, that $\omega_O$ is an order inducing operator. Furthermore, it commutes with other unary operators if they are neither *order sensitive* nor *order inducing*, namely, $\pi$, $\sigma$ and $\rho$. Importantly, $\omega_O$ does not commute with its own kind unless the operators are identical (idempotence)!

Furthermore, $\sigma$, $\pi$, $\rho$ and $\lambda$ 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 (\), the set

**Table 6.2 Relational operators and their algebraic properties.**

| Name | Symbol | Order | Com. | Com. Op. | Dist. |
|---|---|---|---|---|---|
| Union | $\mathcal{R}_L^{O_1} \cup \mathcal{R}_R^{O_2}$ | induces $\emptyset$ | - | ✓ | - |
| Cartesian Product | $\mathcal{R}_L^{O_1} \times \mathcal{R}_R^{O_2}$ | induces $\emptyset$ | - | ✗ | - |
| Intersection | $\mathcal{R}_L^{O_1} \cap \mathcal{R}_R^{O_2}$ | induces $O_1$ | - | ✗ | - |
| Difference | $\mathcal{R}_L^{O_1} \setminus \mathcal{R}_R^{O_2}$ | induces $O_1$ | - | ✗ | - |
| Natural Join | $\mathcal{R}_L^{O_1} \bowtie_P \mathcal{R}_R^{O_2}$ | induces $O_1$ | - | ✗ | - |
| Rename | $\rho_{\mathcal{A}_A \to \mathcal{A}_B}(\mathcal{R}^O)$ | retains $O$ | $\pi$, $\sigma$, $\rho$, $\lambda$, $\omega$ | - | $\cup$, $\cap$, $\setminus$, $\times$, $\bowtie$ |
| Selection | $\sigma_S(\mathcal{R}^O)$ | retains $O$ | $\pi$, $\sigma$, $\rho$, $\omega$ | - | $\cup$, $\cap$, $\setminus$, $\times$, $\bowtie$ |
| Extended Projection | $\pi_P(\mathcal{R}^O)$ | retains $O$ | $\pi$, $\sigma$, $\rho$, $\lambda$, $\omega$ | - | $\cup$ |
| Limit | $\lambda_{s,k}(\mathcal{R}^O)$ | retains $O$ | $\pi$, $\rho$ | - | ✗ |
| Order | $\omega_{O_2}(\mathcal{R}^{O_1})$ | induces $O_2$ | $\pi$, $\sigma$, $\rho$ | - | $\cap$, $\setminus$, $\bowtie$ |

intersection ($\cap$) and the natural join ($\bowtie$) retain the order of the left operand $\mathcal{R}_L$, since the operators select elements from $\mathcal{R}_L$ based on their presence in $\mathcal{R}_R$. As a consequence, the operands do not 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 explicit order that can be expressed within the proposed framework and must thus be considered unordered, i.e., the two operators are order inducing in that they establish a new natural ordering. In addition, distributivity with all the order retaining binary operations is given as expressed by Equation (6.10).

$$\omega_O(\mathcal{R}_L^{O_L} \cap \mathcal{R}_R^{O_R}) = \omega_O(\mathcal{R}_L^{O_L}) \cap \omega_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \cap \mathcal{R}_R)^{O_L} \tag{6.10}$$

$$\omega_O(\mathcal{R}_L^{O_L} \setminus \mathcal{R}_R^{O_R}) = \omega_O(\mathcal{R}_L^{O_L}) \setminus \omega_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \setminus \mathcal{R}_R)^{O_L} \tag{6.11}$$

$$\omega_O(\mathcal{R}_L^{O_L} \bowtie_P \mathcal{R}_R^{O_R}) = \omega_O(\mathcal{R}_L^{O_L}) \bowtie_P \omega_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \bowtie_P \mathcal{R}_R)^{O_L} \tag{6.12}$$

However, the order operator does not distribute with the operators $\cup$ and $\times$.

$$\omega_O(\mathcal{R}_L^{O_L} \cup \mathcal{R}_R^{O_R}) = (\mathcal{R}_L^{O_L} \cup \mathcal{R}_R^{O_R})^O \overset{!}{\neq} \omega_O(\mathcal{R}_L^{O_L}) \cup \omega_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \cup \mathcal{R}_R)^\emptyset$$

$$\omega_O(\mathcal{R}_L^{O_L} \times \mathcal{R}_R^{O_R}) = (\mathcal{R}_L^{O_L} \times \mathcal{R}_R^{O_R})^O \overset{!}{\neq} \omega_O(\mathcal{R}_L^{O_L}) \times \omega_O(\mathcal{R}_R^{O_R}) = (\mathcal{R}_L \times \mathcal{R}_R)^\emptyset$$

The extended projection $\pi_P$ inherits all its properties from the normal projection $\pi$ if we require $f \in \mathcal{F}$ 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 $\pi_P$, this is a reasonable assumption. However, it is important to note that operators may

rely on attributes introduced by a projection $\pi_P$ and therefore commutativity is limited to cases, where no such interdependence exists.

Finally, the s,k-selection operator $\lambda$ is also idempotent if $s = 0$ and it commutes with most $\rho$ and $\pi$. However, it does neither commute with the selection operator $\sigma$ nor the order operator $\omega$.

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

Lack of commutativity of $\lambda$ with $\omega$ follows directly from its definition: $\lambda$ is sensitive to the ranking of $\mathcal{R}^O$ and $\omega$ influences that ranking. Consequently, executing $\lambda$ before or after imposing a certain order will yield different results. For commutativity with $\sigma$, one can turn to Proof 6.1. A summary of all properties can be found in Table 6.2.

**Proof 6.1 Non-commutativity of $\sigma$ and $\lambda$**

Let $\mathcal{R}$ be a unary relation with $M = |\mathcal{R}|$ and $\texttt{SCH}(\mathcal{R}) = \{\mathcal{A}_R\}$. Let further be $t\,[\mathcal{A}_R] = \texttt{RANK}(t) \forall t \in \mathcal{R}$, i.e., the attribute holds the rank of the tuple according to the natural ordering as a fixed value. If $\sigma$ and $\lambda_k$ were to commute, then following identity would hold for any $x < M \in \mathbb{N}_{>0}$:

$$\sigma_{\mathcal{A}_R > x}(\lambda_x(\mathcal{R})) = \lambda_x(\sigma_{\mathcal{A}_R > x}(\mathcal{R}))$$

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

$$\sigma_{\mathcal{A}_R > x}(\lambda_x(\mathcal{R})) = \emptyset \overset{!}{\neq} \lambda_x(\sigma_{\mathcal{A}_R > x}(\mathcal{R})) = \{t_{x+1}, \ldots, t_{\min(x+x,M)}\}$$

That is, whenever $\lambda_k$ removes tuples that would match the predicate of $\sigma$, the order of execution influences the result and the two operators do not commute. An example for $M = 6$ and $x = 2$ is given below,

| $\mathcal{R}$ | $\mathcal{A}^*_{\texttt{R}}$ | $\lambda_x(\mathcal{R})$ | $\sigma_{\mathcal{A}_R > x}(\mathcal{R})$ | $\sigma_{\mathcal{A}_R > x}(\lambda_x(\mathcal{R}))$ | $\lambda_x(\sigma_{\mathcal{A}_R > x}(\mathcal{R}))$ |
|---|---|---|---|---|---|
| $t_1$ | 1 | ✓ | ✗ | ✗ | ✗ |
| $t_2$ | 2 | ✓ | ✗ | ✗ | ✗ |
| $t_3$ | 3 | ✗ | ✓ | ✗ | ✓ |
| $t_4$ | 4 | ✗ | ✓ | ✗ | ✓ |
| $t_5$ | 5 | ✗ | ✓ | ✗ | ✗ |
| $t_6$ | 6 | ✗ | ✓ | ✗ | ✗ |

### 6.1.3 DFCs and Query Optimisation

In addition to the algebraic properties of the extended relational algebra, we can also leverage the requirements imposed on the structure of a function $f \in \mathcal{F}$ to enable a DBMS to plan and optimise 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 make a few assumptions as to how queries are being executed:

1. We assume that a DBMS maintains the collection of functions $\mathcal{F}$. Since every function can be identified by its signature $\texttt{SIG}(f)$, $\mathcal{F}$ 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.

2. We assume that the execution of a (proximity based) query takes place in a *query context*, which remains constant as the query is being evaluated, with the exception of the tuples that are being loaded and pushed into the operator tree.

3. We require all $f \in \mathcal{F}$ to be stateless and side-effect free with respect to that query context.

Assuming these properties, we can distinguish between functions that remain *constant* within the scope of the query context, which can be converted to a literal value, and such that do not. This very basic property allows for early evaluation and caching of function values. Since the extended projection also allows for the nesting of functions, such an optimisation can be relevant.

In addition, we propose the notion of a *function hierarchy* depicted in Figure 6.1. As we move down that hierarchy, more information about the structure of a function employed in a query plan becomes available. That information can be leveraged to decide about the best execution path:

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

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

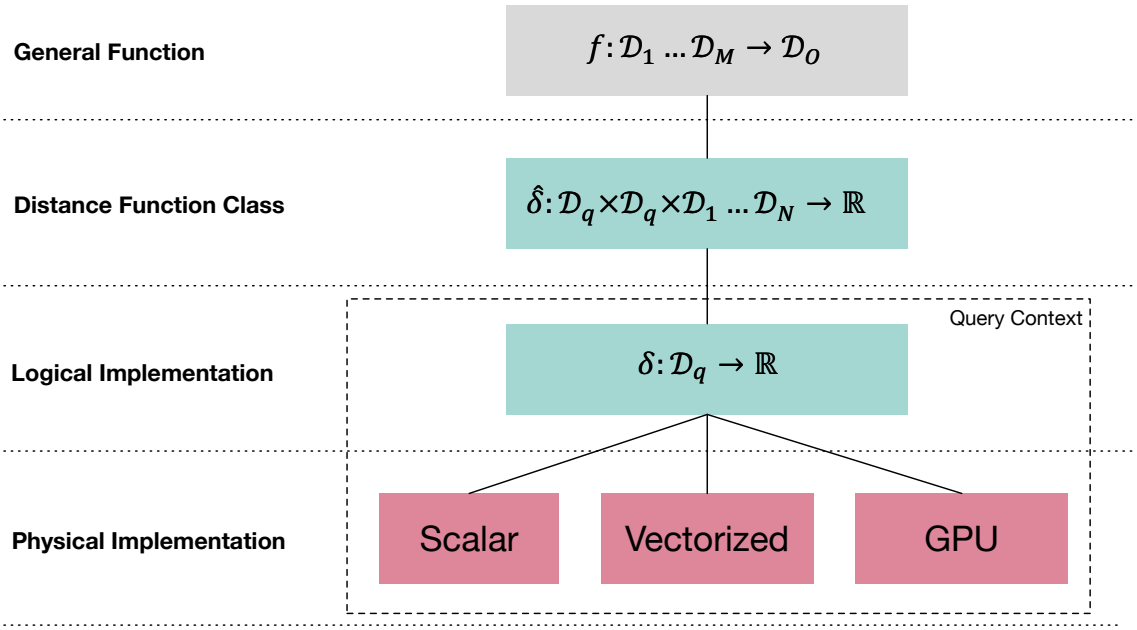



**Figure 6.1 Function hierarchy of all $f \in \mathcal{F}$. As we go down the hierarchy, the DBMS gains knowledge about a function's structure.**

**Logical Implementation** As an implementation of a query manifests, the structure of a DFC may change through implementation $\mathfrak{d} = \texttt{IMP}(\hat{\mathfrak{d}})$. 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, which leverages SIMD instructions of the CPU, over a scalar implementation that calculates the distance in a tight loop.

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. This is especially true for DFCs that operate on complex datatypes, such as vectors. This goes into the direction of Requirement 2.5.

Another aspect that can be optimised is the use of vectorised implementations for computations that involve vectors or matrices, i.e., exchanging a scalar implementation of a DFC by an implementation that uses SIMD instructions. This can be beneficial both for a multimedia DBMS that employs the iterator model and even more so for a batched processing model. However, for the it-

erator model it is to expected that one challenge lies in finding the threshold at 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., dim = 3) while it will greatly benefit high-dimensional vectors (e.g., dim = 2048). Finding this threshold poses challenge in and by itself, which could be subject for future research.

Finally, one can try to match constellations of functions and replace them with an optimised version. A simple example could be the expression $a * b + c$, i.e., $\texttt{add}(\texttt{mul}(\texttt{a},\texttt{b}),\texttt{c})$, which can be replaced by a single *fused multiply-add* function $\texttt{fma}(\texttt{a},\texttt{b},\texttt{c})$ to avoid loss of precission. Replacing the evaluation of a DFC with a lookup in a high-dimensional index also falls into this category, however, this transformation depends on more than just the function itself.

### 6.1.4 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 the use of a high-dimensional index. As we have discussed in Section 4.2.2, there exist many different types of indexing techniques that can be used to speed-up NNS in different ways. We propose to formalise the use of high-dimensional indexes[2] by defininining equivalence classes between an index and the relational algebra expressions it replaces. For this, we rely on the index definitions presented in Section 3.3.5. In addition to Definition 3.7, the following restrictions may apply in addition to implementation specific restrictions for a specific high-dimensional index: (i) High-dimensional indexes are always derived from a feature attribute $\mathcal{A}_f \in \texttt{SCH}(\mathcal{R})$. 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 $\texttt{fun}(\mathcal{A}_f)$, will render the index unusable unless the same alteration was applied when creating the index. (ii) High-dimensional indexes are sometimes trained for a specific distance function (e.g., Euclidean) or class of distance functions (e.g., Minkowski). Therefore, an index can only replace an expression if the DFC used in that expression coincides with the original function (-class). Using these constraints, we can now formalise the replacement of a relational expression by an $\mathcal{I}_I^{\mathcal{R}}$ which, for high-dimensional indexes and depending on what expression is matched, falls into one of three classes given by Definitions 6.9, 6.10 or 6.11.

[2] By "index" we refer to a concrete implementation rather than a family since all techniques introduced in Section 4.2.2 can be employed in different ways.

**Definition 6.8 Exact Index Replacement**

Let $\mathcal{R}$ be a relation, $\mathcal{P}$ a relational (sub-)expression (i.e., an execution plan) involving $\mathcal{R}$ and $\mathcal{I}_I^{\mathcal{R}}$ an index on $\mathcal{R}$. We call the replacement of $\mathcal{P}$ with $\mathcal{I}_I^{\mathcal{R}}$ an *exact index replacement* if $\mathcal{P} \equiv \mathcal{I}_I^{\mathcal{R}}$.

**Definition 6.9 High-Dimensional Index Replacement of Class 1**

Let $\mathcal{R}$ be a relation, $\mathcal{A}_f \in \mathtt{SCH}(\mathcal{R})$ an attribute of $\mathcal{R}$ , $\hat{\mathfrak{d}}$ a DFC, $q \in \mathcal{D}_f$ a query and $\pi_P$ the extended projection involving execution of that DFC using the query, i.e., $\mathfrak{d} = \hat{\mathfrak{d}}(\mathcal{A}_f, q, \ldots) \in P$. We call a replacement with a high-dimensional index $\mathcal{I}^{\mathcal{R}}_{\mathcal{A}_f,\mathfrak{d}} \equiv \pi_{\hat{\mathfrak{d}}}(\mathcal{R})$ *class 1*, if:

$$\pi_P(\mathcal{R}) = \pi_{\hat{\mathfrak{d}}}(\mathcal{R}) \bowtie_{\mathtt{TID}} \pi_{P\setminus\hat{\mathfrak{d}}}(\mathcal{R}) = \mathcal{I}^{\mathcal{R}}_{\mathcal{A}_f,\hat{\mathfrak{d}}} \bowtie_{\mathtt{TID}} \pi_{P\setminus\hat{\mathfrak{d}}}(\mathcal{R})$$

Class 1 index replacements constitute the simplest and most versatile case. The index acts as a drop-in replacement for the distance calculation. One high-dimensional index-structures that can provide this functionality it the simplest variant of the PQ index, namely, the exhaustive SDC and ADC algorithms described in [JDS10] .

**Definition 6.10 High-Dimensional Index Replacement of Class 2**

Let $\mathcal{R}$ be a relation, $\mathcal{A}_f \in \mathtt{SCH}(\mathcal{R})$ an attribute of $\mathcal{R}$ , $\hat{\mathfrak{d}}$ a DFC, $q \in \mathcal{D}_f$ a query and $\pi_P$ the extended projection involving execution of that DFC using the query, i.e., $\mathfrak{d} = \hat{\mathfrak{d}}(\mathcal{A}_f, q, \ldots) \in P$ and $\omega_{\mathcal{A}_{d\updownarrow}}$ be an order operation involving the derived distance attribute $\mathcal{A}_d$. We call a replacement with a high-dimensional index $\mathcal{I}^{\mathcal{R}}_{\mathcal{A}_f,\hat{\mathfrak{d}}} \equiv \omega_{\mathcal{A}_{d\updownarrow}}(\pi_{\hat{\mathfrak{d}}}(\mathcal{R}))$ *class 2* if:

$$\omega_{\mathcal{A}_{d\updownarrow}}(\pi_P(\mathcal{R})) = \omega_{\mathcal{A}_{d\updownarrow}}(\pi_{\hat{\mathfrak{d}}}(\mathcal{R})) \bowtie_{\mathtt{TID}} \pi_{P\setminus\hat{\mathfrak{d}}}(\mathcal{R}) = \mathcal{I}^{\mathcal{R}}_{\mathcal{A}_f,\hat{\mathfrak{d}}} \bowtie_{\mathtt{TID}} \pi_{P\setminus\hat{\mathfrak{d}}}(\mathcal{R})$$

Class 2 index replacements return a ranked relation that comes pre-sorted according to the derived distance in addition to obtaining the actual distance value. This can be convenient for certain applications but it is also a rare trait and we do not have a concrete example of an index that provides this property. However, the caching of (sub-)queries could lead to such a replacement.

**Definition 6.11 High-Dimensional Index Replacement of Class 3**

Let $\mathcal{R}$ be a relation, $\mathcal{A}_f \in \mathtt{SCH}(\mathcal{R})$ an attribute of $\mathcal{R}$ , $\hat{\mathfrak{d}}$ a DFC, $q \in \mathcal{D}_f$ a query and $\pi_P$ the extended projection involving execution of that DFC using the query, i.e., $\mathfrak{d} = \hat{\mathfrak{d}}(\mathcal{A}_f, q, \ldots) \in P$, $\omega_{\mathcal{A}_{d\updownarrow}}$ an order operation involving the derived distance attribute $\mathcal{A}_d$, $\lambda_k$ a $k$-selection and $\sigma_S$ a selection involving a comparison of the distance attribute $\mathcal{A}_d$. We call a replacement with a high-dimensional index $\mathcal{I}^{\mathcal{R}}_{\mathcal{A}_f,\hat{\mathfrak{d}}} \equiv \mathtt{OP}(\omega_{\mathcal{A}_{d\updownarrow}}(\pi_{\mathfrak{d}}(\mathcal{R})))$ with $\mathtt{OP} \in \{\lambda_k, \sigma_S\}$ *class 3* if:

$$\mathtt{OP}(\omega_{\mathcal{A}_{d\updownarrow}}(\pi_P(\mathcal{R}))) = \mathtt{OP}(\omega_{\mathcal{A}_{d\updownarrow}}(\pi_{\hat{\mathfrak{d}}}(\mathcal{R}))) \bowtie_{\mathtt{TID}} \pi_{P\setminus\hat{\mathfrak{d}}}(\mathcal{R}) = \mathcal{I}^{\mathcal{R}}_{\mathcal{A}_f,\hat{\mathfrak{d}}} \bowtie_{\mathtt{TID}} \pi_{P\setminus\hat{\mathfrak{d}}}(\mathcal{R})$$

Class 3 replacements constitute an optimisation of a specific edge-case, which is that of kNN, kFN and or $\epsilon$NN search. It is probably the most common since it allows for most optimisation but is also the least versatile type of index replacement, since it merely supports one specific type of search. An example of such an index replacement is provided in Example 6.8

**Example 6.8 Index Replacement for kNN Search**

The following table lists the schema, extent and rank of $\mathcal{R}^{\emptyset}_{\mathtt{painting}}$, with their title $\mathcal{A}_{\mathtt{(t)itle}}$, the year of their creation $\mathcal{A}_{\mathtt{(p)ainted}}$ and some arbitrary feature vector $\mathcal{A}_{\mathtt{(f)eature}}$. Let further $\hat{\mathfrak{d}}$ be the Manhattan distance.

| $\mathcal{R}^{\emptyset}_{\mathtt{(p)ainting}}$ | $\mathcal{A}^{*}{}_{\mathtt{(t)itle}}$ | $\overline{\mathcal{A}}_{\mathtt{(a)rtist}}$ | $\mathcal{A}_{\mathtt{(p)ainted}}$ | $\mathcal{A}_{\mathtt{(f)eature}}$ |
|---|---|---|---|---|
| $t_1$ | Mona Lisa | Leonardo da Vinci | 1506 | $[2.0, 7.0, -8.0]$ |
| $t_2$ | The Starry Night | Vincent van Gogh | 1889 | $[1.0., 0.0, 3.0]$ |
| $t_3$ | Las Meninas | Diego Velázquez | 1665 | $[-1.0, 3.0, 9.0]$ |
| ... | ... | ... | ... | ... |
| $t_N$ | Mars and Venus | Sandro Botticelli | 1483 | $[-3.0, 1.0, 0.0]$ |

The result of the kNN-search can be produced by a class 3 index replacement with $\mathcal{I}^{\mathcal{R}_p}_{\mathcal{A}_f,\hat{\mathfrak{d}}}$.

$$\mathcal{R}^{\mathcal{A}_d\uparrow}_{\mathtt{result}} = \lambda_k(\omega_{\mathcal{A}_d\uparrow}(\pi_{\mathcal{A}_t,\mathfrak{d}(\mathcal{A}_f)}(\mathcal{R}_p))) = \mathcal{I}^{\mathcal{R}_\mathtt{p}}_{\mathcal{A}_f,\hat{\mathfrak{d}}} \bowtie_{\mathtt{TID}} \pi_{\mathcal{A}_t}(\mathcal{R}_p)$$

An index structure could be employed here is the VAF index [WSB98].

With the aforementioned definitions and restrictions, we have defined the basic relationship and rules that can be applied by a multimedia DBMS, when deciding whether or not to use a high-dimensional index structure. However, the definitions so far consider the replacement an exact transformation, i.e., the result of the original expression and the index is supposed to be identical. In practice, this is often not the case, which leads to Definition 6.12.

**Definition 6.12 Approximate Index Replacement**

Let $\mathcal{R}$ be a relation, $\mathcal{P}$ a relational (sub-)expression involving $\mathcal{R}$ and $\mathcal{I}_I^{\mathcal{R}}$ an index on $\mathcal{R}$. We call the replacement of $\mathcal{P}$ with $\mathcal{I}_I^{\mathcal{R}}$ an *approximate index replacement* if $\mathcal{P} \approx_b \mathcal{I}_I^{\mathcal{R}}$ with:

$$\mathcal{P} \approx_b \mathcal{I}_I^{\mathcal{R}} \leftrightarrow Q(\mathcal{P}, \mathcal{I}_I^{\mathcal{R}}) \geq b, b \in [0, 1]$$

$Q$ is a *quality estimator* that derives a score between zero and one by comparing the results of $\mathcal{P}$ and $\mathcal{I}$.

In a manner of speaking, $\mathcal{P}$ acts a groundtruth and is compared to the approximate results produced by $\mathcal{I}_I^{\mathcal{R}}$. The choice of quality metric $Q$ is implementation specific. One could, for example, pick recall if one is primarily interested in set-based overlap rather than the ranking of individual items, which is enough for unranked query plans:

$$Q_1(\mathcal{P}, \mathcal{I}) = \frac{|\mathcal{P} \cap \mathcal{I}|}{|\mathcal{P}|}$$

An alternative could be the use of a normalised Discounted Cumulative Gain (DCG) [JK02], if the ranking of items should be taken into account as well:

$$Q_2(\mathcal{P}, \mathcal{I}) = \frac{\texttt{DCG}(\mathcal{P}, \mathcal{I})}{\texttt{iDCG}(\mathcal{P})}$$

Regardless of the choice of metric, the challenge lies in predicting its value for a query prior to its execution. Since the result of $Q$ can only be produced in hindsight once a query has been executed, we need techniques to anticipate its value for an incoming query so that the DBMS can use this downstream to make decision about query execution.

## 6.2 Cost Model for Multimedia Databases

As we have explained in Chapter 3, most DBMS rely on a cost-model for planning and selecting the execution and access path of a user-specified query. In fact, ever since the System R paper [SAC+79], cost-based query planning and optimisation has been considered a gold standard for database systems [MCS88]. Most traditional DBMS exhibit cost-models that mainly rely on the cost incurred by accessing (reading / writing) database pages from or to disk [MCS88; GUW09; Pet19]. Based on what has been discussed thus far, we argue that this model must be adapted to be useful for multimedia databases so that it takes the following aspects into account when chosing a query execution plan:

**Disk Access (IO)** Persistent storage and the access to information residing on disk is the factor that contributes the most to long query execution times and is thus a factor to minimise [SAC+79]. This is also true for proximity based queries, where part of the optimisation lies in reducing access to vectors through compression and/or early filtering, e.g., [WSB98; CPR+07].

**CPU** Due to the computational complexity of proximity based operations, especially when the evaluation of functions on high-dimensional vectors are involved, the processing time on CPU is a factor that can no longer be ignored and must thus be taken into account as well. In fact, several index structures achieve speed-up by reducing the computational complexity of the distance calculation [JDS10] or by avoiding it alltogether [WSB98].

**Memory** Some algorithms can benefit greatly from pure in-memory processing. While memory used to be a scarce resource, some environments allow for complete in-memory processing even for very large datasets. This is exploited by systems like Milvus [WYG+21], that load entire data collections into memory.

**Quality** The quality of a result incurred, e.g., by the choice of a certain high-dimensional index structure [IM98; JDS10], is a price that can be paid to attain speed-up. This is common practice in approximate NNS and should be made transparent to the system and user for reasons explained in Requirement 2.4, especially since this trade is not always viable (see Section 2.3).

These four factors can be used to characterise query workloads in a purely local setup consisting of a single node. For distributed databases, the cost incurred by data and message exchange must obviously be considered as well.

While this is not in scope for the work presented in this Thesis, the proposed model can be easily extended to accomodate those (and other) types of costs. The aforementioned factors lead us to Definition 6.13 and Definition 6.14 for the cost incurred by an operator OP and query execution plan $\mathcal{P}$.

**Definition 6.13 Cost of a Relational Operator OP**

Let OP be a (relational) operator. We call the 4-tuple $C_{\texttt{OP}} = (c_{\texttt{IO}}, c_{\texttt{CPU}}, c_{\texttt{MEM}}, c_{\texttt{Q}})$ the *cost* of OP, which captures the atomic costs in terms of disk access (IO), CPU usage (CPU), memory usage (MEM) and quality (Q).

**Definition 6.14 Cost of a Query Exection Plan $\mathcal{P}$**

Let $\mathcal{P} = \texttt{OP}_1 \circ \texttt{OP}_2, \ldots \circ \texttt{OP}_N$ be a query execution plan consisting of $N$ operators. The plan's total cost $C_{\mathcal{P}}$ is given by the component-wise sum of the individual cost components:

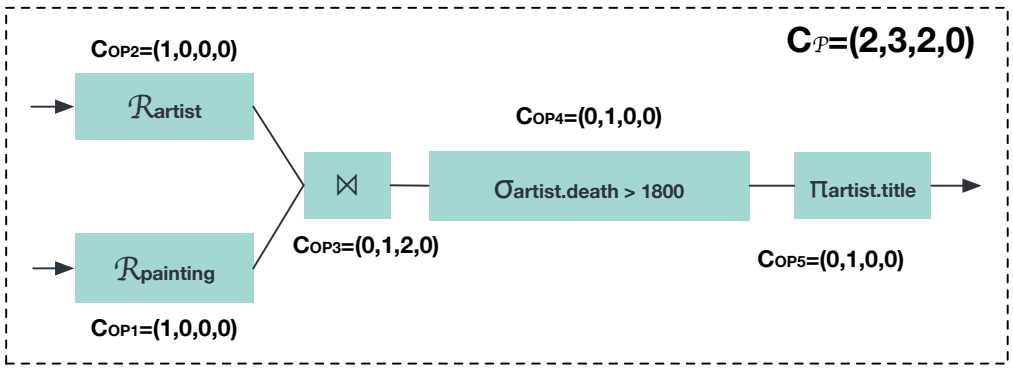
$$C_{\mathcal{P}} = \sum_{i=1}^{N} C_{\texttt{OP}_i} = (\sum_{i=1}^{N} c_{\texttt{IO},i}, \sum_{i=1}^{N} c_{\texttt{CPU},i}, \sum_{i=1}^{N} c_{\texttt{MEM},i}, \sum_{i=1}^{N} c_{\texttt{Q},i})$$

**Example 6.9 Cost of a Query Execution Plan**

Let $\mathcal{R}_{\texttt{painting}}$ and $\mathcal{R}_{\texttt{artist}}$ be relations from Example 3.3, the query "return the names of all paintings that were painted by an artist who died after 1800" results in the following, unoptimised execution plan:

$$\mathcal{P} = \pi_{\mathcal{A}_{\texttt{title}}}(\sigma_{\mathcal{A}_{\texttt{death}}>1800}(\mathcal{R}_{\texttt{artist}} \bowtie \mathcal{R}_{\texttt{painting}}))$$

This query gives rise to following operator tree with individual and total costs (the cost factors are purely illustrational and do not relate to any real values):



The role of $C_{\mathcal{P}}$ is twofold: First, its absolute values can be used by the DBMS to determine hard constraints imposed for query execution. For example, if there is a fixed limit on memory that can be used by a query, an execution plan may be discarded simply based on exceeding that memory limit regardless of other factors. Similarily, a hard constraint on quality could be imposed. Second, it can be used to compare execution plans relatively to one another. Example 6.9 illustrates cost calculation for a simple execution plan.

The cost factors per operator can be determined in many ways and the model does not assume any specific approach. However for $c_{\mathtt{Q}}$, we will propose an approach in Section 6.2.2. We do require, though, that the method for determining the factors must be internally consistent and should not change so as to keep costs comparable. For illustrative purposes, we outline the following, rough metrics to estimate $c_{\mathtt{IO}}$, $c_{\mathtt{CPU}}$, $c_{\mathtt{MEM}}$: IO cost can be estimated by the number of bytes that must be read / written by an operator. This is equivalent to the estimated number of tuples read ($T_r$) or written ($T_w$) times the estimated size of a tuple $s$ in bytes times the atomic IO cost $a_r$ or $a_w$ for reading or writing a byte. One can also introduce a correction factor $c$ to distinguish between different types of accesses (e.g., sequential vs. random).

$$c_{\mathtt{IO}} = (a_{\mathtt{r}} T_r + a_{\mathtt{w}} T_w) sc$$

CPU cost can be estimated by the number memory accesses and/or the number of arithmetic operations that are executed by an operator. Here we assign an atomic CPU cost $a_{\mathtt{CPU}}$ to a type of operation and the total cost can then be expressed as that atomic cost times the number of tuples $T$ being processed ($T_p$).

$$c_{\mathtt{CPU}} = a_{\mathtt{CPU}} T_p$$

Memory cost can be estimated by the size of the in-memory data structures maintained by an operator (e.g., for sorting or hash-joins), i.e., the number of tuples materialised in memory $T_m$ times the size of a tuple s.

$$c_{\mathtt{MEM}} = sT_m$$

Since the different cost factor stem from completely different domains and are therefore not directly comparable and because we only compare plans relative to one another, we require a cost normalisation step, which we base on available plan candidates $\mathcal{P} \in \mathtt{PLANS}$ and which is outlined in Definition 6.15.

**Definition 6.15 Cost Normalisation**

Let $\texttt{PLANS} = \mathcal{P}_1, \ldots, \mathcal{P}_N$ be $N$ equivalent query execution plans with associated costs $\texttt{COSTS} = C_{\mathcal{P}_1}, \ldots, C_{\mathcal{P}_N}$. We consider the maximum cost $C_{\texttt{max}}$ for those plans to be the component-wise maximum of all $C_{\mathcal{P}} \in \texttt{COSTS}$. Using $C_{\texttt{max}}$, the normalised cost $C_{\mathcal{P},i,\texttt{norm}}$ can be expressed as:

$$C_{\mathcal{P},i,\texttt{norm}} = \left(\frac{c_{\texttt{IO},i}}{c_{\texttt{IO,max}}}, \frac{c_{\texttt{CPU},i}}{c_{\texttt{CPU,max}}}, \frac{c_{\texttt{MEM},i}}{c_{\texttt{MEM,max}}}, \frac{c_{\texttt{Q},i}}{c_{\texttt{Q,max}}}\right)$$

It follows from the definition, that each component of a normalised cost falls into an interval between 0 and 1 due to the normalisation, i.e., $c_{*,\texttt{norm}} \in [0, 1]$.

It is important to note, that the normalised cost can only be used for relative comparison within an existing set of plans. It generally does not provide any information about the original cost value and the underlying properties they may represent. Using this normalised costs, we can now derive *cost score* according to Definition 6.16.

**Definition 6.16 Simple Cost Score**

Let $\mathcal{P}$ be a query execution plan with associated, normalised cost $C_{\mathcal{P}_{\texttt{norm}}}$. We consider the *cost score* to be the sum of the atomic costs.

$$\mathcal{S} = c_{\texttt{IO,norm}} + c_{\texttt{CPU,norm}} + c_{\texttt{MEM,norm}} + c_{\texttt{Q,norm}}$$

The basic cost score $\mathcal{S}$ falls into an interval between 0 and $N$ wherein $N$ denotes the number of atomic costs, i.e., $\mathcal{S} \in$ [0,N] for $N = |C_{\mathcal{P}_{\texttt{norm}}}|$.

The cost score $\mathcal{S}$ can be used to directly compare query plans in a given context and, e.g., sort them in ascending order during plan enumeration in order to select the one plan that minimises the costs. However, it assumes that all the components have the same importance, which in practice is often not the case.

### 6.2.1 Cost Policies

The definitions employed so far treat every component of a cost $C_{\mathcal{P}_{\texttt{norm}}}$ equally with regards to the resulting cost score $\mathcal{S}$. This is rarely desirable since the different components influence the outcome to a different extent (e.g., in terms of query execution time or result quality). This can be expressed by employing a *cost policy* as outlined in Definition 6.17.

**Definition 6.17 Cost Policy and Weighted Score**

Let $\mathcal{P}$ be a query execution plan with associated, normalised cost $C_{\mathcal{P}_{\text{norm}}}$. We call the 4-tuple $\mathcal{W}$ a cost-policy.

$$\mathcal{W} = (w_{\text{IO}}, w_{\text{CPU}}, w_{\text{MEM}}, w_{\text{Q}}), \text{ with } w_{\text{IO}}, w_{\text{CPU}}, w_{\text{MEM}}, w_{\text{Q}} \in [0, 1]$$

The weighted cost score $\mathcal{S}_{\mathcal{W}}$ with respect to $\mathcal{W}$ can be expressed as the linear combination of $C_{\mathcal{P}_{\text{norm}}}$ and $\mathcal{W}$:

$$\mathcal{S}_{\mathcal{W}} = C_{\mathcal{P}_{\text{norm}}} \cdot \mathcal{W} = w_{\text{IO}} c_{\text{IO,norm}} + w_{\text{CPU}} c_{\text{CPU,norm}} + w_{\text{MEM}} c_{\text{MEM,norm}} + w_{\text{Q}} c_{\text{Q,norm}}$$

The cost policy provides us with a blunt but simple instrument to assign importance to individual cost factors. For example, one can prioritise execution speed by assigning more weight to the CPU and IO components or one can favour the quality of the results by having a high $w_{\text{Q}}$. In an actual multimedia DBMS implementation, $\mathcal{W}$ can be influenced at different levels, which supersede one another, namely (i) as a system-wide configuration, (ii) for a user-session (connection-level) or transaction, or (iii) for an individual query. Consequently, different users should have the ability to express different preferences in terms of cost policy.

### 6.2.2 Estimating Cost of Approximation

In order to complete the cost model, we must be able to generate an estimate of how the quality of a result is influenced by chosing a particular execution plan over another. We propose a simple method that is directly related to the approximate index replacement problem given in Definition 6.12. It involves a post-hoc comparison of query results to estimate the quality measure at different levels using the quality function $Q$.

**Definition 6.18 Empirical Estimation of Quality Cost**

Let $\mathcal{P}$ be a naive, unoptimised query plan and let $\mathcal{P}^*$ be an optimised version thereof. Let further $\lambda_k$ be the $s,k$-selection and $Q$ be a quality estimator that derives a score by comparing the results produced by $\mathcal{P}$ and $\mathcal{P}^*$. The cost estimate for quality $c_{Q,k}$ at a given level $k$ is then is given as:

$$c_{Q,k,\mathcal{P}^*} = 1 - q_k = 1 - Q(\lambda_k(\mathcal{P}), \lambda_k(\mathcal{P}^*))$$

For a proximity based query, the optimised version of the plan $\mathcal{P}^*$ does involve the evaluation of an approximate index while the unoptimised version $\mathcal{P}$ does not. However, any type of query plan that involves an approximate query method can be compared that way (e.g., stochastic scanning proposed by [Gia18]). In practice, the obtained numbers for $c_{Q,k}$ can be stored in a lookup-table and updated on a regular basis (e.g., when querying the index anyway). In case no observation is available, one can provide a *cost prior* with the index implementation, which is an index-dependent estimate of the quality cost incurred. Furthermore, the DBMS could also pre-calculate certain costs factors based on the known classes of high-dimensional index replacements described Section 6.1.3 and queries that are likely to be executed. Whenever a query is being planned, the DBMS can lookup the estimate that best fits the query to perform the cost calculation.

As indicated by the subscripts $k$ and $\mathcal{P}^*$, we estimate $c_{Q,k,\mathcal{P}^*}$ separately for multiple recall levels $k$ and for different families of plans. We propose a logarithmic progression of the ranking levels, e.g., $k = \{1, 2, 4, 8, \ldots\}$ for two reasons: Firstly, the likelihood of an extreme (outlier) value is higher for lower than for higher levels for purely stochastical reasons. This is also visualised in Figure 6.2. Secondly, items with a lower rank are often considered more relevant than items that exhibit a high rank [JK02], i.e., we require a more accurate estimate for the lower levels. We also expect the impact of approximation to be different depending on the type of query that is being executed. For example, [JDS10] report that the distortion of the distance estimate for PQ is not very relevant for NNS but ought to be considered, if one is interested in the actual distance values.

Due to its simplicity, the model obviously only provides a rough estimate. Furthermore, there are some important implicit assumptions we operate upon: Firstly, we assume that the quality of an execution plan depends on the type of plan and not on the query parameters provided by the user (e.g., the query vectors used for proximity based search). The underlying assumption is, that the query parameters are being drawn from the same, (albeit unknown) probability distribution that has produced the indexed data. We believe this to be a reasonable assumption, because as we have argued in Chapter 4, these vectors are usually produced by some well-defined feature transformation. The second assumption is, that the quality of the results at lower, constrained ranking levels provides a valid estimator for quality at higher levels, i.e., that the agreement is continued indefinitely. Both assumptions require some well-behavednes' of the index structure, which may not necessarily be a given.

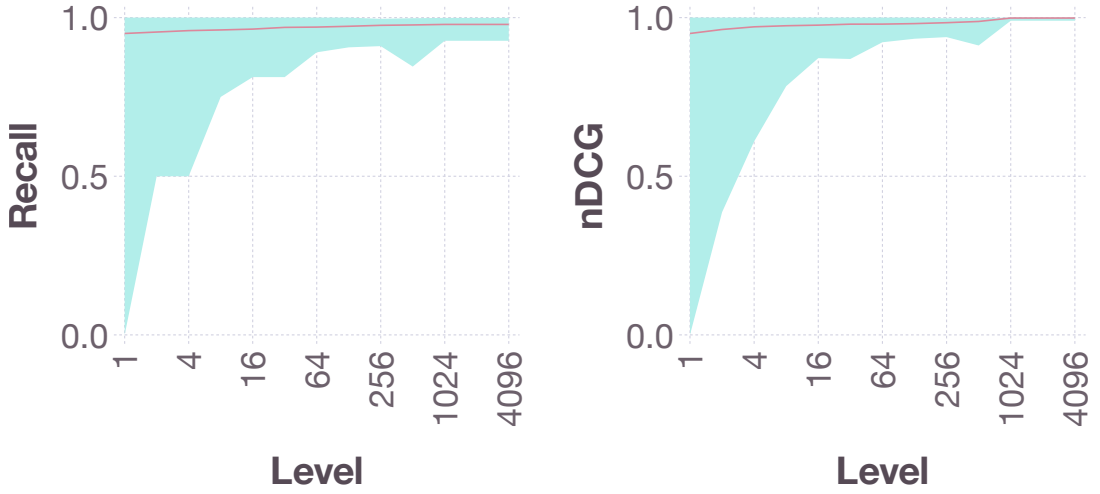



**Figure 6.2 Quality of results produced by a PQ index in a NNS query in terms of recall and normalised DCG at different levels observed and averaged over a 1000 queries using different query vectors. The mean value remains more or less constant while minimum and maximum exhibit a large spread for smaller levels.**

## 6.3 High-Dimensional Index Maintenance

As collections grow beyond the capacity for main memory architectures, the on-disk storage of indexes becomes an important aspect to consider [LÁJ+09; JSDS+19; CZW+21]. Yet, while the challenge of maintaining durable index structures and handling incremental as well as concurrent updates to them is a well studied problem for, e.g., the $B^+$-tree index [GUW09; Pet19], the on-disk storage and maintenance of high-dimensional indexes is not [Ams14], with the exception of a few, isolated examples [ÓÞJA+11; HJB19; LWW+20]. In addition to practical considerations, e.g., how data structures designed for use in main memory can be optimised for disk access to guarantee durability, there is the fundamental problem of an index's ability to cope with incremental changes to the data in a consistent manner, especially when concurrent read and write access is involved. What makes the issue challenging is that, as we have demonstrated in Section 4.2.2, many different types of algorithms exist for high-dimensional indexing, all of which bring their own characteristics. Since the problem of dynamic, high-dimensional indexing on secondary storage has not been widely considered in literature and is in fact often ignored by the inventors of new indexing methods, [3] there are no established best practices as to how to handle the issue at a systems level [ÓÞJA+11; Ams14].

[3] It is either assumed that data is static or that re-indexing the entire collection is an option.

In this section, we propose a model to accomodate and maintain different types of durable high-dimensional index structures in a consistent manner. The model is supposed to be applicable to any type of index and considers it to be an auxiliary data structure, that provides certain (potentially imperfect) operations that enable us to make incremental changes to it. Furthermore, indexes in this model exhibit an explicit state that allows the DBMS to make decisions as to if and how the index can and should be utilised. Failure of an index due to accumulating changes is an explicit scenario that must be handled by the system. The proposed model consists of a series of components, based on which it can operate. Those are: (i) A *storage model*, (ii) a *write model* and, (iii) a *failure- and associated cost model* for every type of index, (iv) a system-wide notion of transaction isolation and (v) the ability to (re-)build an index from scratch. The model can be applied for any type of index and we will see, that a traditional index structure can be considered a specific corner-case of the model.

### 6.3.1 Storage Model

The basic assumption and prerequisite is that all the high-dimensional index structures offered by the DBMS always act as secondary indexes. Consequently, the primary data (i.e., the original vectors) are always stored without loss of information and in a manner that allows for seamless propagation of changes, e.g., as index- or heap-organised tables in the case of a relational database. [4] At this point, we re-iterate the formal definition of a secondary index $\mathcal{I}_I^{\mathcal{R}}$ with respect to the relation $\mathcal{R}$ it indexes from Definition 3.7 and the related Equation (3.10) that enables us to reconstruct attributes that are missing in the index.

The fallback strategy that the DBMS therefore can always resort to, in case an index becomes unavailable, is that of a sequential scan of the primary data, which results in a higher latency for reasons discussed in Section 4.2, and should therefore be avoided. Furthermore, the primary data can be used to reconstruct the index or parts thereof in larger batches – in a (re-)indexing process. Since, to the best of our knowledge, all relevant publications on high-dimensional indexes outline the steps required to construct the index from a data collection, we can safely assume that this is always possible. Moreover, the primary data can be used to correct missing or outdated information on the fly at query time, e.g., using random lookups.

[4] This is an approach (implicitly) taken by methods such as DiskANN [JSDS+19] or SPANN [CZW+21], however, for the simple reason that in-memory storage is not feasable for very large collections.

In addition to this basic requirement, one must consider how the in-memory representation of a particular index can be efficiently mapped to disk. Refering to the list of indexes presented in Table 4.2, a naive mapping is often straightforward. Many indexes are organised as lists, trees or inverted (multi-)indexes [SZ03; BL14] and can therefore be organised using the persistent version of the respective data structure. At a high level, if an index is always scanned in its entirety, an array or list may be the easiest and most efficient form of storage. One could, for example, use a binary association table-like structure as used by MonetDB [IGN+12] to map `TID` to signature. If an index can benefit from filtering, e.g., to restrict a scan to a subset of the index in an inverted index [CPR+07; GJA10; JDS10], then a ($B^+$-)tree structure might be more beneficial. In practice, there are many caveats to sort out, such as, balancing the size of clusters in inverted indexes [HJB19] or deciding which part of the data to keep in memory [LÁJ+09; JSDS+19; CZW+21]. We consider these to be important but index-specific aspects, which are, however, out of scope for this thesis.

### 6.3.2 Write Model of an Index

An index's write model characterises its ability to incorporate incremental changes into the data structures outlined by its storage model. To lay the groundwork, we formalise the notion of a change to a relation in the form of three operations listed in Definition 6.19.

**Definition 6.19 Change Operations to Relation $\mathcal{R}$**

Let $\mathcal{R}$ be a relation and $t \in \mathcal{R}$, $t' \in \mathcal{R}'$ tuples with $\texttt{SCH}(\mathcal{R}) = \texttt{SCH}(\mathcal{R}')$. We call $\texttt{OP}_c(\mathcal{R}, \cdot) \in [\texttt{INSERT}(\mathcal{R}, \cdot), \texttt{DELETE}(\mathcal{R}, \cdot), \texttt{UPDATE}(\mathcal{R}, \cdot)]$ a change operation on $\mathcal{R}$ and the quadruple $C = (\texttt{OP}_c, \mathcal{R}, t, t')$ a *change*.

$$\begin{aligned}
\texttt{INSERT}(\mathcal{R}, t') &= \mathcal{R} \cup \{t'\} \\
\texttt{DELETE}(\mathcal{R}, t) &= \mathcal{R} \setminus \{t\} \\
\texttt{UPDATE}(\mathcal{R}, t, t') &= ((\mathcal{R} \setminus \{t\}) \cup \{t'\})
\end{aligned}$$

That is, $\texttt{OP}_c(\mathcal{R}, \cdot)$ can be used to add, remove or replace tuples in $\mathcal{R}$. These operations correspond to the basic CRUD operations available to a DBMS. It is worth noting that formally, we can decompose any `UPDATE` operation into a consequtive `INSERT` and `DELETE`.

The write model $\texttt{WRITEM}_{\mathcal{I}}$ is now a function as given by Definition 6.20. It decides for every change $C$ if and how it can be propagated to index $\mathcal{I}$ and applies it, if possible. In case a change $C$ can be applied successfully, the write model returns `true`. If the change cannot be propagated, the write model returns `false`. In both cases, the change is forwarded to the index's failure model alongside with the outcome $W$ of the write model.

**Definition 6.20 Write Model for Index $\mathcal{I}$**

Let $\mathcal{R}$ be a relation and $\mathcal{I}$ an index on attributes $I \subset \texttt{SCH}(\mathcal{R})$. The write model $\texttt{WRITEM}_{\mathcal{I}}$ is a function that determines, if a change $C = (\texttt{OP}_c, \mathcal{R}, t, t')$ can be applied to the index given additional parameters $P = \{p_1, p_2, \ldots p_n\}$:

$$\texttt{WRITEM}_{\mathcal{I}}(C, P) = \begin{cases} \texttt{true}, & \text{if } \texttt{OP}_c(\mathcal{R}, t, t') \text{ can be applied to } \mathcal{I} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

The application of $\texttt{WRITEM}_{\mathcal{I}}$ always takes place within the context of the transaction $\mathcal{T}_c$ that has issued the change and it therefore inherits all guarantees provided by it. Most importantly, we assume that all effects of $\texttt{WRITEM}_{\mathcal{I}}$ are made visible to other transactions upon `COMMIT` and are negated by a `ROLLBACK`.

### 6.3.3 Failure Model and State of an Index

The failure model $\texttt{FAILM}_{\mathcal{I}}$ is anothers function as given by Definition 6.21 that determines the new state of the index given change $C$, the result of the write model $W$ and optional parameters $P = \{p_1, p_2, \ldots p_n\}$ (e.g., up-to-date index statistics).

**Definition 6.21 Failure Model for Index $\mathcal{I}$**

Let $\mathcal{R}$ be a relation and $\mathcal{I}$ an index on attributes $I \subset \texttt{SCH}(\mathcal{R})$. The failure model $\texttt{FAILM}_{\mathcal{I}}$ is a function that determines the index's new state $S \in \{\texttt{CLEAN}, \texttt{DIRTY}, \texttt{STALE}\}$ given a change $C = (\texttt{OP}_c, \mathcal{R}, t, t')$, the outcome of the write model $W$ and parameters $P = \{p_1, p_2, \ldots p_n\}$:

$$\texttt{FAILM}_{\mathcal{I}}(C, W, P) = \begin{cases} \texttt{CLEAN}, & \text{if change can be applied without imparing the index} \\ \texttt{DIRTY}, & \text{if change may impair the index} \\ \texttt{STALE}, & \text{if change leads to index failure} \end{cases}$$

Based on the outcome of the failure model, the index's state and cost characteristics can be updated. The state determines, if and how the index can be used for query execution and thereby enforces consistent results within the quality parameters provided by the DBMS. The cost characteristic allows for more fine grained control, in case, certain cost factors are directly influenced by a change (or an accumulation thereof), for example, if changes lead to a deterioration in quality. We distiguish between three states as illustrated in Figure 6.3:

**Clean** indexes are ready for use by the DBMS and able to produce results within the speed and quality parameters they naturally provide.

**Dirty** indexes are ready for use by the DBMS. However, the flag signals that an index's ability to produce results within speed and quality parameters is impaired. This flag is usually accompanied by adjustments to the cost characteristic of the index that account for this.

**Stale** indexes have stopped functioning within their defined parameters and are therefore removed from consideration for query planning. Reason for this state is often an inconsistency, e.g., due to a missing record.

Freshly (re-)built indexes start in the `CLEAN` state. If the write model fails for a change $C$, i.e., $\texttt{WRITEM}(C, P) = \texttt{false}$, the state is updated to `STALE` and the index becomes unavailable and is marked for rebuilding. However, even if the write model succeeds, the change may still trigger a transition to either the `DIRTY` or the `STALE` state, if the changes that have accumulated in the meanwhile are expected to lead to index deterioration that goes beyond a certain threshold. Internal statistics can be consulted and updated to, e.g., adjust the index's cost model.

Again, we require that the application of $\texttt{WRITEM}_{\mathcal{I}}$ – including all updates to state and cost model – take place within the context of the transaction $\mathcal{T}_c$ that has issued the change. Consequently, it also inherits all guarantees and is subject to its `COMMIT` and `ROLLBACK` semantics.

### 6.3.4 Concurrent Index Rebuilding

Once an index has been marked as `STALE` it can no longer be used by the DBMS until it is rebuilt by the $\texttt{REBUILD}(\mathcal{R}, \mathcal{I}, P)$ function, where $P = \{p_,p_2, \ldots p_n\}$ again denotes a list of additional parameters. In essence, the rebuild function simply re-constructs the index $\mathcal{I}$ based on the primary data held in $\mathcal{R}$.

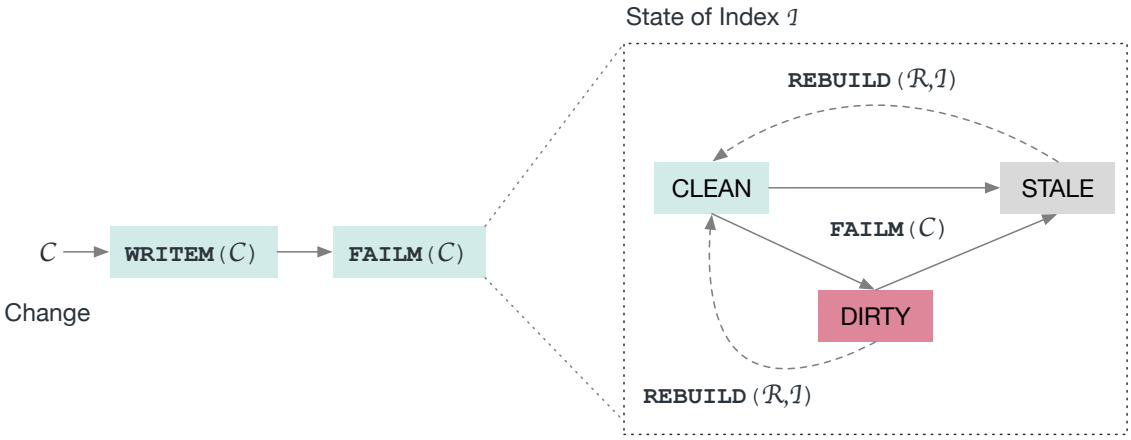



**Figure 6.3 Illustration of a change $C$ to the data that is process by an index $\mathcal{I}$'s write- and failure model and leads to potential state changes. An index knows the three states `CLEAN`, `DIRTY` and `STALE`.**

Using this model, we prevent a faulty index from generating potentially erroneous results, e.g., if entries go missing because they could not be added. Hence, we can ensure consistency even as potentially unprocessable changes accumulate. However, the failure of an index means that the DBMS must resort to a sequential scan, which may negatively impact query performance. This can only be remediated by an index rebuild, which is often a long-running process that may take several minutes to hours for large datasets to complete. A time during which, new changes may arrive.

This is why we propose an algorithm that minimises conflicts with other, concurrent database operations. As a pre-requisite for this algorithm to work, we assume that every transaction $\mathcal{T}_t$ operates on a consistent snapshot $\mathcal{S}_t$ of the database state at the time $t$ when the transaction started. This is a concurrency control mechanism known as *snapshot isolation* [BBG+95], which is very common in modern DBMS. In this section, we will refer to snapshots of relations and indexes with subscripts, e.g., $\mathcal{R}_1 \in \mathcal{S}_1$ is a relation and $\mathcal{I}_1 \in \mathcal{S}_1$ is an index for $t = 1$. The rebuilding of a (high-dimensional) index can now be separated in two steps: The `BUILD` and the `REPLACE` phase, which start at timepoints $s$ and $r$ respectively, with $s < r$. The basic assumption behind separating these two steps is, that building up the index structure is the time consuming part, while merging the index to the persistent data structures (i.e., writing it to disk) is faster, since it does not involve any access to the primary data nor any complex computations. Hence, the aim of our approach is to minimise exclusive access to the primary data structures maintained by the DBMS. The entire algorithm is also illustrated in Figure 6.4.

During the BUILD phase, the primary data in $\mathcal{R}_s$ is scanned in a read-only transaction $\mathcal{T}_s$ and the obtained data is used to build up the data structures required by the index. The constructed data structure exists outside of a proper database snapshot and is therefore not part of any visible database state other than $\mathcal{T}_s$ [5]. We therefore refer to this as a *shadow index* $\mathcal{I}'$. Since transaction $\mathcal{T}_s$ is read-only, concurrent reads and writes to the database can take place and all transaction $\mathcal{T}_{s \geq t < r}$ will see the state of the original $\mathcal{I}_{s<t<r}$ at the given point in time, regardless of whether that index is still operational (CLEAN), in process of deterioration (DIRTY) or has failed in the meanwhile (STALE).

The index build-up is followed by the REPLACE phase, which takes place in a second transaction $\mathcal{T}_r$ that follows directly after $\mathcal{T}_s$ and writes the shadow index created during the BUILD to the DBMS snapshot $\mathcal{S}_r$ and then commits that snapshot. $\mathcal{T}_r$ is a write transaction and may require exclusive access on the data structures involved. It replaces the $\mathcal{I}_{r-1}$ with an updated version $\mathcal{I}_r$ that can take over. All transactions $\mathcal{T}_{t \geq r}$ will see the state of the newly built index $\mathcal{I}_{t \geq m}$.

Separating these two steps obviously ignores one important aspect, which is that of change operations that take place between timepoints $s$ and $r$. Since the SCAN phase operates on snapshot $\mathcal{R}_s$, those are not taken into account when rebuilding the index. We solve this, by introducing an additional *side-channel* between $\mathcal{T}_s$ and concurrent transactions $\mathcal{T}_{s<t<m}$ that LEAK relevant information. Every concurrent transaction that commits successfully, reports all relevant changes to the entity that orchestrates the REBUILD operation, which will be responsible for merging these changes with the shadow index $\mathcal{I}'$ using its write- and failure model. In practice, this can be realised as some kind of event-bus allowing for a *publish-subscribe* like communication model. The entity that orchestrates the rebuild can then subscribe to changes of the relation it is scanning.

#### 6.3.4.1 Failure Cases and Limitations

The proposed algorithm has several failure cases that must be handled and limitations that must be addressed. Obviously, REBUILD($\mathcal{R}, \mathcal{I}, P$) fails, if the leaking process fails (i.e., information is lost) or if during the BUILD phase, the shadow index becomes unusable, i.e., its failure model signals a problem with some of the leaked information. In both cases, the shadow index must be discarded and rebuilding of the index must be and resumed from scratch in a new, up-to-date transaction. Therefore, for this algorithm to work, the index's write model should bring a certain degree of robustness to accumulating changes .

[5] We do not make any assumptions as to whether the structure is built in-memory or on disk.

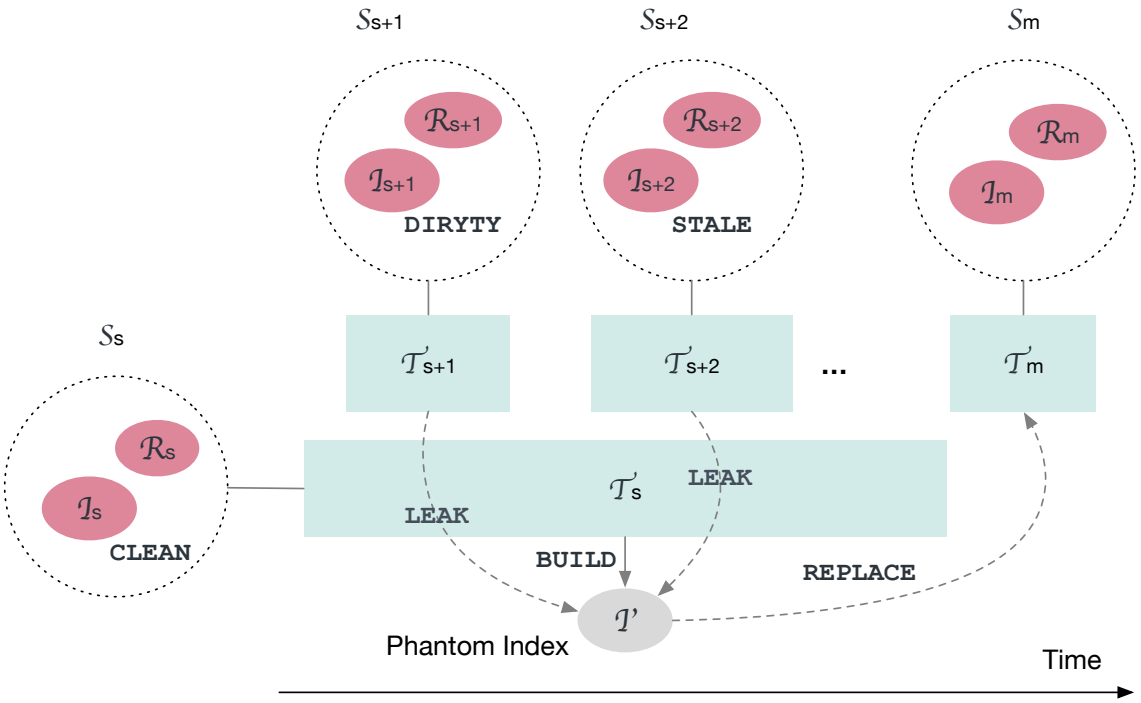



**Figure 6.4 Illustration of the asynchronous index rebuilding process, which takes place in two separate phases `BUILD` and `SCAN`. Information from transactions that take place concurrently to the `BUILD` process are `LEAK`ed through a side-channel upon commit of the respective transaction.**

Since the likelihood of accumulating enough changes for the rebuild to fail increases with runtime, the algorithm can also be combined with other strategies to reduce time spent in the critical `BUILD` phase, e.g., by having multiple, smaller partitions of the primary data in $\mathcal{R}$ being (re-)built independently. [6] This is a strategy, e.g., employed by Milvus [WYG+21] and formally, this leads to an index $\mathcal{I}$ that is composed of multiple sub-indexes, each with a shared write- and failure model but their own state. The state of the index is then determined by the state of its sub-indexes .

Last but not least, it is important to note that the proposed solution becomes a poweful tool if index failure can be (approximately) anticipated. Presuming this is possible, the index rebuild can be started ahead of time while the original index $\mathcal{I}$ is still functional. Due to snapshot isolation, concurrent read transactions can continue to use the original index until the new version becomes available. In addition to heuristics (e.g, after $x$ change operations the index is being rebuilt) one can also leverage the cost model proposed in Section 6.2 to determine if index quality is deteriorating.

[6] Such a model can also make sense if we consider partitioning of data during parallel execution and/or distribution.

### 6.3.5 Case Studies

We illustrate the idea of the write- and failure model using two index structures VAF [WSB98] and PQ [JDS10]. These case studies serve as mere examples and the concepts themselves can be applied to other indexes as well.

#### 6.3.5.1 Vector Approximation Files (VAF)

The VAF (see Section 4.2.2.1) algorithm assigns a vector $f$ to a cell in a high-dimensional lattice that is determined by a partitioning of the vector space. That lattice – which acts as a scalar quantiser that must be stored with the index metadata – is derived from the component-wise minimum and maximum $f_l$ and $f_u$ of all feature vectors $f \in \mathcal{D}_f$. From that we can determine the write model for a relation $\mathcal{R}$ with the attribute $\mathcal{A}_f$ of data domain $\mathcal{D}_f \subset \mathbb{R}^d$ with $d \in \mathbb{N}_{>0}$. The `INSERT` is illustrated in Equation (6.13) and Figure 6.5.

$$\texttt{WRITEM}_{\texttt{VAF}}((\texttt{INSERT}, \mathcal{R}, t'), f_l, f_u) = \begin{cases} \texttt{true} & \text{if } f_{l,i} \geq t'[\mathcal{A}_f]_i \geq f_{u,i} \forall i \in [1, d] \\ \texttt{false} & \text{otherwise} \end{cases} \tag{6.13}$$

When a new tuple $t'$ is inserted, the write model for the VAF index must check if the vector $t'[\mathcal{A}_f]$ falls within bounds of $f_l$ and $f_u$. If it does, then the signature can be calculated directly and appended to the list – the write model returns `true`. Otherwise, the vector's position in the grid is undefined and the write model should return `false`. However, to harden the index for those types of failures, one can simply assign the closest cell (i.e., the first or the last one) instead, as illustrated in Figure 6.5. Alternatively, such an entry could also be explicitly marked using a special number. Both measures ultimately have a similar effect: When the index is scanned and either an out-of-bounds marker is encountered OR the bounds are not sufficient for filtering out the vector, the original entry must be fetched from the primary data and the exact distance must be obtained. Over time, this will lead to a deterioration of query speed, since more vectors must be read from disk.

If an existing tuple $t \in \mathcal{R}$ is removed, then this change can be applied in any case. The signature is simply looked-up and removed from the list. However, as deletes accumulate, the component-wise minimum and maximum of the $\mathcal{D}_f$ may shift, which potentially leads to an overly large and therefore sparsely populated lattice.

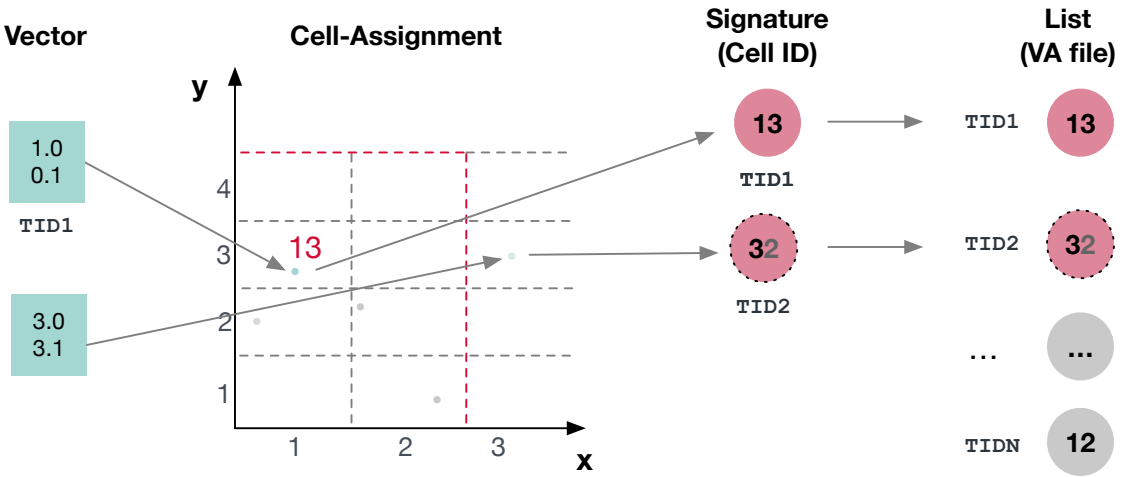



**Figure 6.5 Illustration of the VAF index's write model (insert). If a vector lies within the bounds spanned by the minimum and maximum, the signature can be appended. If not, the closest cell is assigned, leading to a poor estimate of bounds druing scanning.**

$$\texttt{WRITEM}_{\texttt{VAF}}(\texttt{DELETE}, \mathcal{R}, t) = \texttt{true} \tag{6.14}$$

Formally, the ability to apply an update can be directly expressed as a combination of $\texttt{WRITEM}((\texttt{INSERT}, \mathcal{R}, t'), \cdot)$ and $\texttt{WRITEM}((\texttt{DELETE}, \mathcal{R}, t), \cdot)$, as outlined in Definition 6.19. In practice, the two steps would be merged into a dedicated operation to optimise performance.

The failure model must now determine, how to handle the aforementioned changes. Both the sparsenes of the overlay grid as well as the accumulation of out-of-bound entries potentially lead to a deterioration of the index's ability to produce results in a reasonable amount of time. Hence, as these changes accumulate, the need for an index rebuild becomes more pressing. As a very simple option, the DBMS could determine a fixed threshold after which an index will go `STALE` or after which re-indexing will be triggered. While simple to realise, it is probably difficult to find a good estimate for that threshold.

Alternatively, one can use the VAF index's filtering capability as a measure. According to [WSB98], a typical VA-file should be able filter-out between 90 and 99% of the candidates without accessing the original vectors. That number can be used as a default value. Every time an index is scanned when executing a query, that value can be updated and with it, the index's cost model. Eventually, the index will be removed from consideration automatically as the cost of scanning it exceeds the cost of a full table scan.

#### 6.3.5.2 Product Quantisation (PQ)

PQ (see Section 4.2.2.3) involves quantisation of a vector based on the assignment to a list of centroids in a codebook. The resulting signature is then either stored in list (PQ) or an inverted (multi-)index (IVFPQ). For both PQ and IVFPQ, the quantiser used to construct the original index can be stored and re-used to process changes to the data. Consequently, an incoming vector is simply quantised and appended to the respective data structure and the write model for a (IVF-)PQ should always return `true`, as indicated by Equation (6.15).

$$\texttt{WRITEM}_{\texttt{PQ}}(\texttt{INSERT}, \mathcal{R}, t') = \texttt{true} \tag{6.15}$$

Similarily, for deletes, the entry associated with the deleted `TID` can be removed from the list. One can arrange `TID` and associated signature in a tree structure to accelerate this lookup. In the case of an IVFPQ index, the procedure incurs some additional overhead, since for a successful delete, the original vector must be read and its position in the inverted index must be determined by applying the coarse quantiser. Subsequently, that slot must be scanned until the entry with the deleted `TID` can be found. In any case, the write model should return true as indicated by Equation (6.15).

$$\texttt{WRITEM}_{\texttt{PQ}}(\texttt{DELETE}, \mathcal{R}, t) = \texttt{true} \tag{6.16}$$

Again, formally, the ability to apply an update can be directly expressed as a combination of $\texttt{WRITEM}(\texttt{INSERT}, \mathcal{R}, t')$ and $\texttt{WRITEM}(\texttt{DELETE}, \mathcal{R}, t)$, as outlined in Definition 6.19.

The failure model of a PQ must potentially take multiple aspects into account. Firstly, a quantiser learned for a given size of the collection may become non-representative of the collection as it grows, especially if the growth exceeds orders of magnitude [LLN+20]. Secondly, a growing number of potentially very similar vectors may lead to a degeneration of the indexes ability to distinguish between them (part of the reason, why [JDS10] proposed the quantisation of residuals). And lastly, for IVFPQ, the different clusters constructed by coarse quantisation may become imbalanced over time, similarly to CP [CPR+07]. In all these cases, re-indexing with potentially different parameters may be required[7].

[7] It is to be noted, that finding the optimal choice of parameters given a collection is still a poorly understood problem [GHK+14].

*All the world will be your enemy, Prince with a Thousand Enemies, and whenever they catch you, they will kill you. But first they must catch you, digger, listener, runner, prince with the swift warning. Be cunning and full of tricks and your people shall never be destroyed.*

— Richard Adams, 1972

# 7

# Cottontail DB

Cottontail DB [GRH+20] is our reference implementation for the query, cost and index maintenance models described in Chapter 6. Starting out as a drop-in replacement for ADAMpro [GS16], it has grown to be a full fledged DBMS with support for classical Boolean, fulltex (powered by Apache Lucene [1]) as well as proximity based search used in multimedia retrieval and analytics applications in the *vitrivr* [RGT+16; GRS19a] context. All three 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 Kotlin[2] and runs on the Java Virtual Machine (JVM). Cottontail DB's source code can be downloaded from GitHub[3]. It is freely available under a permissive open source license and can be used as standalone application, Docker container or in an embedded mode.

## 7.1 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 7.1. 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.

[1] See https://lucene.apache.org/
[2] See: https://kotlinlang.org/
[3] See: https://github.com/vitrivr/cottontaildb/

**Table 7.1 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` |
| ByteString | ✓ | ✗ | ✗ | ✗ | `BLOB` |
| 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 | ✗ | ✗ | ✓ | ✓ | - |

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. Consequently, records are assembled on-the-fly as queries are being executed. Similarily 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 identifier* (`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-)organised.

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 [IM98] 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 – including all column and index statistics – is tracked in an internal *catalogue*, which is backed by the main storage engine.

## 7.2 Architecture

The main components of Cottontail DB are depicted in Figure 7.1. 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) or upon request. 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.

### 7.2.1 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 6) introduced thus far. However, not all operators behave exactly as required by the model due to implementation details. For example, the `FETCH` operation corresponds to a variant of the projection $\pi$ but instead of projecting onto a subset of the incoming attributes, it extends them, by looking up values in the database, i.e. logically (logically, `FETCH` corresponds to the operation $\texttt{OP}(\mathcal{R}) \bowtie_{\texttt{TID}} \pi_{\mathcal{A}_+}(\mathcal{R})$ for the additional attribute $\mathcal{A}_+ \in \texttt{SCH}(\mathcal{R})$, wherein the join acts on the tuple identifier). Such an operation can be beneficial since Cottontail DB allows for access to individual columns and thus it can reduce the cost of IO to defer access to certain columns.

Conceptually, Cottontail DB distinguishes between *Nullary*, *Unary*, *Binary* and *N-Ary* operators, which differ in the number of inputs they can accept (0, 1,

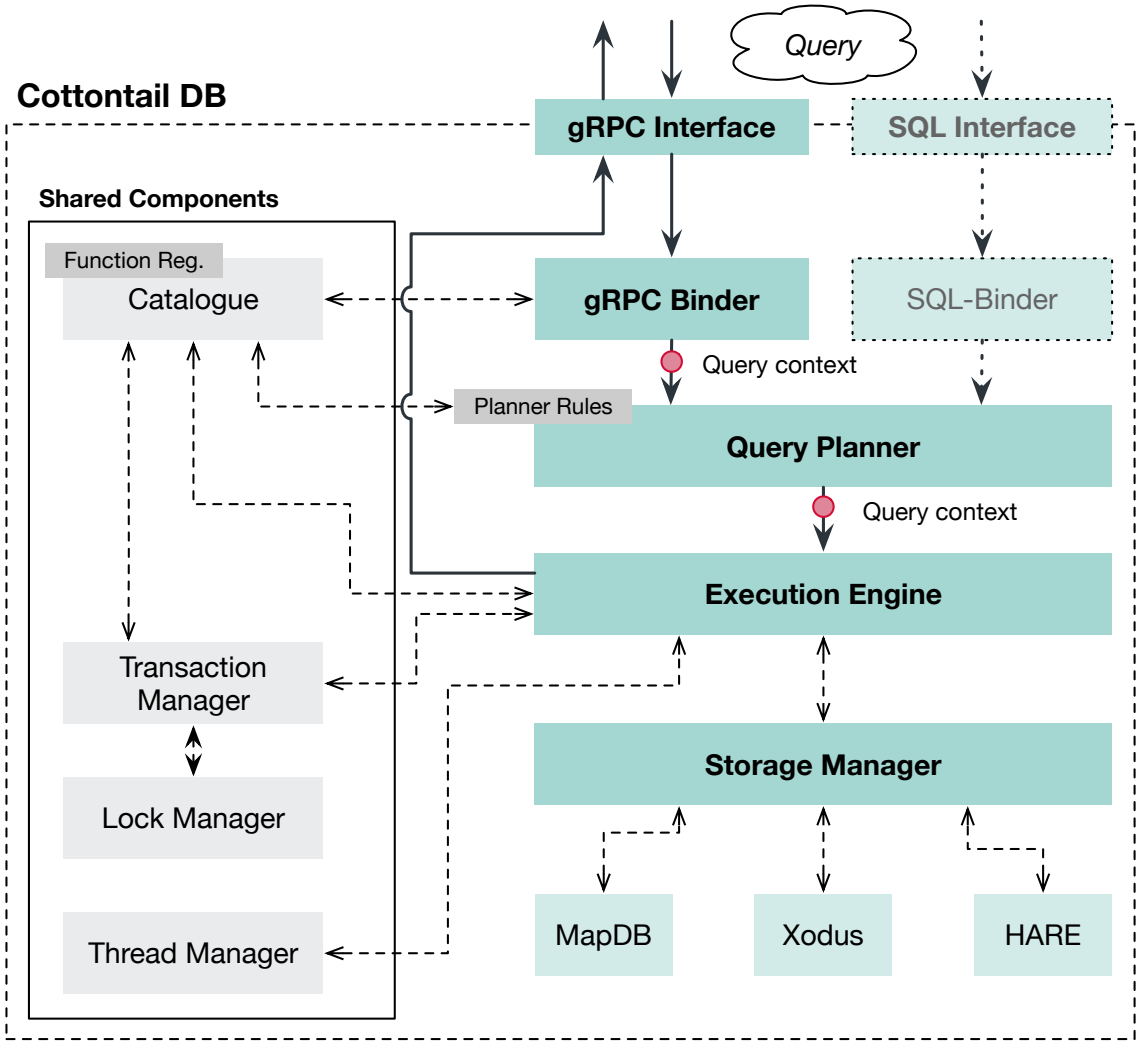



**Figure 7.1 Architecture diagram of Cottontail DB's main components. The directed arrows indicate the path a query takes within the system (dashed for a potential SQL interface). The dashed, double arrows indicate interactions between components.**

2 or N). Furthermore, some operators require materialization of intermediate resultsets 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 7.2. Every operator is implemented in Kotlin and pre-compiled to byte code.

#### 7.2.1.1 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 opera-

**Table 7.2 Main types of physical operators implemented by Cottontail DB alongside with their arity, their correspondence to relational operators and whether or not they require materialization.**

| Type | Arty | Rel. Op. | Mat. | Description |
|---|---|---|---|---|
| SCAN | 0 | $\pi_P(\mathcal{R})$<br>$\rho_{\mathcal{A}_A \to \mathcal{A}_B}(\pi_P(\mathcal{R}))$ | | Act as a data source. Scans an *entity* and interacts directly with the storage manager. |
| INDEX | 0 | $\sigma_S(\mathcal{R})$<br>$\pi_{\mathfrak{d}}(\mathcal{R})$<br>$\omega_{\mathcal{A}_{d\updownarrow}}(\pi_{\mathfrak{d}}(\mathcal{R}))$<br>$\lambda_k(\omega_{\mathcal{A}_{d\updownarrow}}(\pi_{\mathfrak{d}}(\mathcal{R})))$ | | Acts as data source. Scans an *index* and interacts directly with the storage manager. Can be any type of operator sequence outlined in Section 6.1.4 |
| FETCH | 1 | $\bowtie_{\text{TID}}(\cdot, \pi_{\mathcal{A}_+}(\mathcal{R}))$ | | Extends every incoming record by fetching values for one or multiple columns and appending them to the record. This is a specialised projection that extends the columns present in the outgoing relation. |
| FUNCTION | 1 | $\bowtie_{\text{TID}}(\cdot, \pi_{f(\cdot)}(\mathcal{R}))$ | | Extends every incoming record by evaluating a specific function and appending the result as a new column to the record. This is a specialised projection that extends the columns present in the outgoing relation. |
| FILTER | 1 | $\sigma_S(\cdot)$ | | Filter incoming records by evaluating a given predicate. Records that don't match the predicate are not handed down the pipeline. |
| SORT | 1 | $\omega_O(\cdot)$ | ✓ | Sort the incoming records based on the specified columns in the specified direction. |
| LIMIT | 1 | $\lambda_k(\cdot)$ | | Skip and drop incoming records according to specification and thus limit the number of outbound records to a specified number |
| PROJECT | 1 | $\pi_P(\cdot)$<br>$\rho_{\mathcal{A}_A \to \mathcal{A}_B}$ | | 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. |
| MERGE | N | - | | An operator that merges tuples from incoming strands of query execution in an arbitrary order (union semantics). Used for intra-query parallelism. |

tor(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 emiting 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 *Flows*, which are part of its coroutines framework. An example of a naive operator pipeline is provided in Figure 7.2.

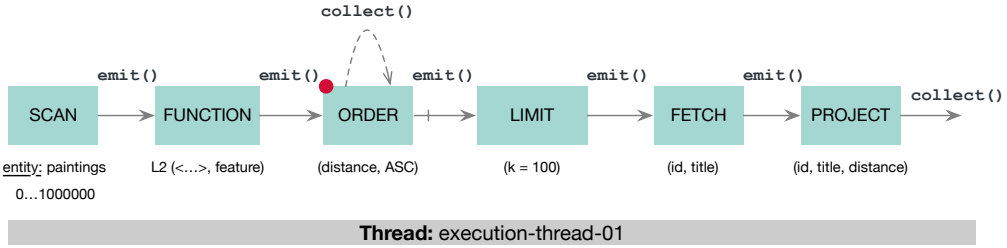



**Figure 7.2 Operator pipeline for a proximity based query that scans the entity *paintings*, calulates the L2 distance between a *feature* column and a query, orders by the obtained distance and limits the number of outputs (kNN). Pipeline breaking operators are indicated by the red dot.**

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 is also part of Kotlin's coroutines framework. Transaction isolation is guaranteed at all time by the *transaction manager* and is achieved through *snapshot isolation*. In addition, this execution model also allows for intra-query parallelism, which is used for queries that incur a high CPU cost. Intra-query parallelism 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 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 host machine.

To prepare for intra-query parallelism, part of the operator pipeline is partitioned wherein every partition operates on a fraction of the input data. 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 of the plan. An additional concurrency aware `MERGE` operator is introduced after such a partitioning point, which synchronizes and merges all the incoming strands using a FIFO scheme. Since `MERGE` operators do not retain ordering, the sort operations must take place afterwards. An example for this is given in Figure 7.3. In practic, the three steps are combined into a single operator.

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 chosing 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.

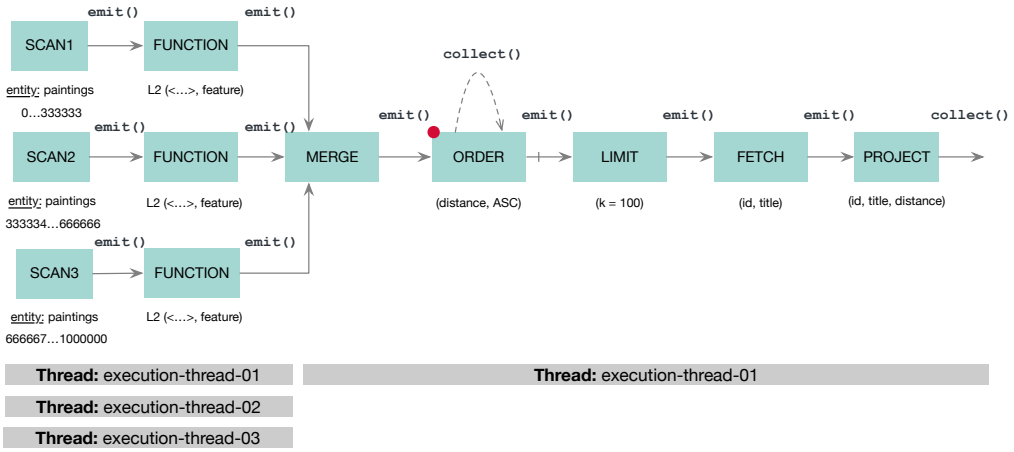



**Figure 7.3 Operator pipeline that performs kNN on the entity *paintings*. The *scan* and distance *function* steps are partitioned and merged before order, limit and projection is applied, which allows for parallel execution.**

### 7.2.2 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 outlined in Section 6.2. The input into the query planner is a logical representation of the operators that should be executed as part of the operator pipeline, which we call the *canonical operator 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 optimised 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 tree undergoes *optimisation*. The optimisation step generates different versions of the input plan by applying rules that act on and manipulate the tree. Finally, the plan that minimises the expected cost is selected and implemented.

#### 7.2.2.1 Optimisation Algorithm

The query planner used by Cottontail DB can be categorized as a rule-based query planner. It is losely 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 recognise patterns in the structure of the operator tree and try to re-arange and/or replace nodes to generate a more optimal but equivalent version.

The optimisation 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) are broken into groups and every group is optimised 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 optimised 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.

Once a collection of potential plans has been generated for each group, the query planer enumerates the (sub-)plans per group, determines the expected cost and selects the (sub-)plan that minimises it. It then reconstructs the full plan from the selected (sub-)plans, which is then implemented and executed. Furthermore, the mapping of the input plan to the resulting output plan is written to the plan cache for future reference.

#### 7.2.2.2 Plan Caching

The plan cache is a mechanism that amortises 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 unique digest of every incoming canonical operator tree to the resulting, optimised 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. Secondly, there is no mechanism that invalidates cached entries as changes to the data occur, e.g., due to indexes becoming unavailable as a result of its write- and failure model (see Section 6.3). And finally, the optimal plan does not only depend on the plan itself but also on the applied cost policy, which can currently not be reflected by the cache (see Section 6.2).

#### 7.2.2.3 Logical Optimisation

The logical optimisation 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, e.g., the ability to decompose a `FILTER` operator evaluating a conjunction into two consecutive `FILTER` operators – one for each side. This is illustrated in the example given in Figure 7.4. The applicaton of this rule can be seen as a preparation for the physical optimisation phase, where specific predicates may be satisifed by a secondary index.

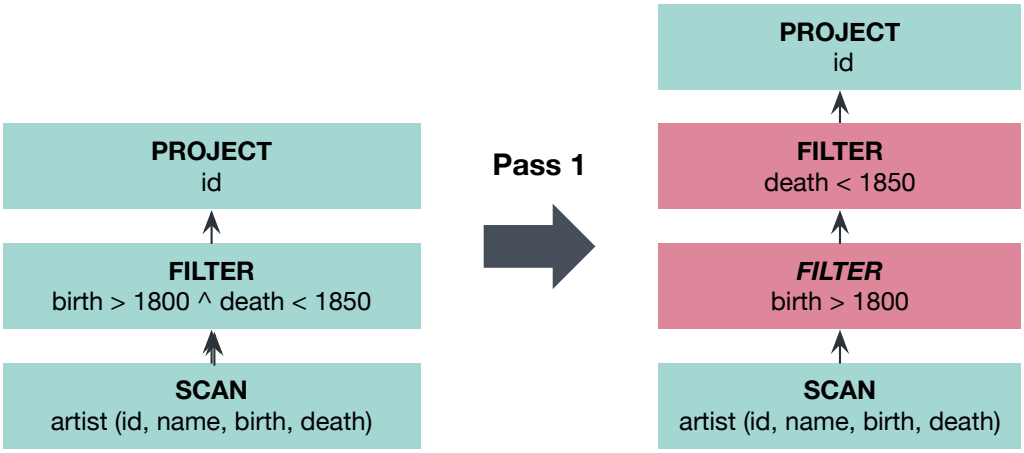



**Figure 7.4 Decomposition of conjunctive predicates. The filter on the column `death` may be pushed down at a later stage during physical optimisation, if an index were to be available.**

#### 7.2.2.4 Physical Optimisation

The physical optimisation step tries different implementations of operators to arrive at a more cost-effective plan. While structural changes to the operator tree may be possible, the focus lies on implementation aspects. At a high level, we distinguish between the following types of rules:

**Index Replacement Rules** try to replace `FILTER` predicates or `FUNCTION` invocations with an `INDEX` operation. 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 the classes of index replacements listed in Definitions 6.9, 6.10 and 6.11.

**Deferral Rules** try to defer the evaluation of functions and/or the fetching of columns to later stages of the query plan, which can be beneficial if a `LIMIT` or a `FILTER` leads to a reduction of the output cardinality. An example is given in Figure 7.5. This can significantly reduce IO and/or CPU costs. These rules also make sure, that only columns that are actually accessed are read from disk (i.e., they defer access until a column is removed from the plan).

**Implementation Rules** simply replace different implementations of a operation one-to-one, e.g., hash-join vs. nested-loop join for a theoretical JOIN operation or a scalar version of a `FUNCTION` invocation by a vectorised version that uses SIMD instructions.

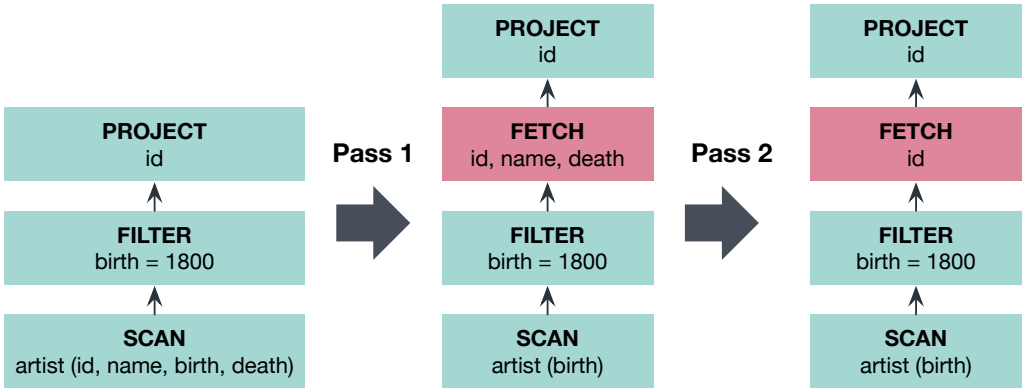



**Figure 7.5 Illustration of deferring `FETCH` operations. Over multiple passes, the `FETCH` of the columns `id`, `name` and `death` is pushed up until after the `FILTER` has been applied. Columns `name` and `death` are eventually eliminated because they are not required.**

**Summarization 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 allows for more effective sorting, as compared to naively sorting all records first and then limiting to the top $k$ entries thereafter.

After each pass in the physical optimisation 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 6.

### 7.2.3 Query Parsing and Binding

Cottontail DB uses a *gRPC*[4] based query language that allows for interaction with other systems. Hence, all queries are expressed as *Protocol Buffer*[5] messages. These interactions include data definition, data management, transaction management and the formulation of queries. 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.

[4] See: https://grpc.io/
[5] See: https://developers.google.com/protocol-buffers/

The reason for the choice of gRPC is threefold: Firstly, Cottontail DB's predecessor – ADAMpro [GS16] – already used and introduced gRPC into the *vitrivr* 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 environments.[6] And finally, the gRPC Kotlin library greatly simplified the task of structuring query endpoints, parsing incoming query messages and handling the query responses.

Even though Cottontail DB uses gRPC, it can be seen as compatible with the SQL standard insofar as it supports a specific feature. This was demonstrated by be the successful integration of Cottontail DB as a storage engine for *Polypheny-DB* [VSS18; VHS+20]. Consequently, it would be 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 unoptimised 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 optimisation are performed at this stage (e.g., type checks, removal of trivial predicates such as ”1 = 1“).

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

#### 7.2.3.1 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 copy of the same, root binding context.

### 7.2.4 Functions and Function Generators

In compliance with the model proposed in Section 6.1, 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 `FunctionRegistry` is a singleton object that is kept in memory for an instance of Cottontail DB. It provides the interface to register `FunctionGenerators` and resolve `Functions` given a `ClosedSignature`, which specifies the name of the requested `Function` and the types of arguments it can take. Such a request is then forwarded to a `FunctionGenerator` with matching `OpenSignature`, which are again objects that exist per type of `Function` – e.g., a Euclidean distance or a scalar multiplication – and act as factories for concrete `Function` implementations. The `OpenSignature` comprises of a `Function`'s name and *open* arguments that specify the class (e.g., number, vector) but not the concrete type (e.g., `float` number, `double` vector). It is up to the `FunctionGenerator` to generate and return a concrete `Function` implementation based on a `ClosedSignature`, if such an implementation exists. This generation process can be a simple lookup of a pre-existing piece of logic or a more complex operation (e.g., for code generation at runtime).

Given this facility, function resolution takes place in two steps: First, a `ClosedSignature` of the requested function is converted to an `OpenSignature` to lookup the responsible `FunctionGenerator` in a map. That generator then produces the concrete implementation, based on the `ClosedSignature`. 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 compile- or runtime, e.g., to automatically generate optimised, type specific versions of the same logic, which Cottontail DB currently does not support.

The equivalence between the execution of DFCs and high-dimensional indexes are implemented as dedicated planner rules that support the proposed classes of index replacements listed in Definitions 6.9, 6.10 and 6.11. Furthermore, some `Functions` implement the `VectorisableFunction` interface. This is a signal to Cottontail DB, that a vectorised version of the `Function` exists and can be obtained through a call to the `vectorised()` method. However, this functionality relies on the still incubating Java Vector API, and is highly experimental.

### 7.2.5 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 main abstractions is that of a Database Object (DBO), namely, `Schema`, `Entity`, `Index` and `Column`, in the form of interfaces. An 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 7.3.

Even though the storage manager abstraction layer could be used to combine different storage engines within a single instance, this is currently not done as proposed by [Gia18]. This is mainly because replacement of a storage engine was often triggered by a very specific requirement the other engines could not fullfil. 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.

**Table 7.3 Primitive operations offered by the storage engines's `Tx` abstractions to interact with DBOs including the argument and return types (`arg` → `ret`).**

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

#### 7.2.5.1 Storage Engine: MapDB

The *MapDB* storage engine is based on version 3.0.8 of a library with the same name[7] 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 IO, 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. Only the catalogue and all metadata 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 implemen-

[7] See https://mapdb.org/, Accessed May 2022

tation 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.

#### 7.2.5.2 Storage Engine: Xodus

The *Xodus* storage engine is based on version 2.0.1 of the open source, transactional and schema less, embedded database for Java and Kotlin with the same name, developed and maintained by JetBrains.[8] 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 high-dimensional index maintenance described in Section 6.3, 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 levels of abstraction over the same core IO facility:

**Environments** encapsulate multiple, named key-value stores that can hold arbitrary keys and values. They allow for transactional reading and modification of data accross 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 envoronments 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

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

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.

#### 7.2.5.3 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.

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 ans is inspired by the binary association tables introduced by Monet DB [IGN+12]. However, this form of storage comes with 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 for 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.

### 7.2.6 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

[9] See https://github.com/google/snappy/

context may live beyond the scope of a single query. This depends on whether a transaction was explicitly started through the respecitve 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 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 Section 6.3. Failure to fullfil this leads to a rollback of a transaction.

Part IV

# Discussion

*Wer misst, misst Mist.*

— German Saying

# 8

# Evaluation

In this chapter we present the evaluation of Cottontail DB [GRH+20], which implements the previously described concepts and has become an integral part of the *vitrivr* [RGT+16; GRS19b; GRS19a] stack. This quantiative evaluation is structured around three experiments that highlight three different aspects:

**Multimedia Analytics Workloads** This experiment tries to simulate multimedia analytics workloads on million-scale datasets, for which Cottontail DB was primarily designed. We demonstrate Cottontail DB's versatility and discuss and compare different execution strategies. We use the Vimeo Creative Commons Collection 1 & 2 (V3C) [BRS+19; RSB21], which currently serves as the reference dataset for both VBS [Sch19] and TRECVID AVS [CHW+21]. The dataset consists of 17 235 videos, which amounts to 2300 h of content. We use a series of features derived by *vitrivr*'s feature extraction engine Cineast.

**Large-Scale Similarity Search** In this benchmark, we test Cottontail DB's ability to scale out to very large datasets as employed in challenges such as the (Big-) ANN-Benchmark [ABF17; SWA+22]. While not the primary focus of Cottontail DB's design, this is still an interesting task to obtain a baseline for future efforts. We use this opportunity to compare Cottontail DB to Milvus (see Section 5.3.1) in a series of large-scale NNS queries. All queries are run against shards of the Yandex Deep1B dataset [BL16], which is a collection of descriptors derived from a GoogLeNet [SLJ+15] neural network.

**High-Dimensional Index Maintenance** In this experiment we examine how Cottontail DB copes with incremental changes to the dataset that take place concurrently to query activities. We demonstrate the index's ability to implement such changes as well as the deterioration of indexing quality over time. Finally, we test the proposed mechanisms that maintain consistent indexes.

## 8.1 Evaluation Setup

The entire evaluation is centered around database queries that are sent to the system under testing. We generate measurements from these queries and the results they return. Typically, we average the obtained values over 10 repetitions in order to compensate for anomalies and we execute a single warm-up query to give the system's the ability to initialise necessary caches (exceptions are mentioned explicitly).

Rather than working with randomly generated data, we decided to use existing data corpora that reflect the structure of features as generated by different models. For that, we use on the Vimeo Creative Commons Collection (V3C) [BRS+19; RSB21] and the Yandex Deep1B dataset [BL16]. For V3C, we rely on features generated by our own extraction engine Cineast [RGT+16] and from the Deep1B dataset we have prepared different shards that contain the first 5 million, 10 million, 100 million and all 1 billion pre-generated feature vectors. All data is imported as a dedicated entity (Cottontail DB) or collection (Milvus) respectively, prior to executing the actual workloads. A list of all data sets and the resulting entities is provided in Table 8.1 and we will refer to the collections and entities by their name throughout this chapter. Necessary index structures are prepared beforehand if not stated otherwise. Every entity exhibits the columns `id` and `feature`. The `id` column serves as a primary key and is either a `string` (all V3C-based datasets) or a `long` value (all Deep1B-based datasets). The Deep1B shards also exhibit an additional `category` column, which holds a randomly assigned `int` value that is used for Boolean filtering. The state of all collections is listed in Appendix A.

Both Cottontail DB and Milvus are deployed on the same physical server (but they do not run at the same time). This server exhibits two NUMA nodes with an Intel Xeon CPU E5-2630 v4 (20 cores@2.2 GHz) and 192 GB RAM each. Therefore, the machine provides 40 compute cores and a total of 384 GB of RAM. The CPU supports the AVX2 instruction set extension, which allows for vectorised execution. The data directories from which Milvus and Cottontail DB access their data resides on three 500 GB SSD's that have been combined into a single, logical disk using RAID0 (striping). The server runs Ubuntu 20.04 and the OpenJDK Java version 17.0.3. All benchmarks are orchestrated by and sent from a separate node on the same network to minimise the impact on query execution. The evaluation scripts we used are available online[1].

[1] See https://github.com/ppanopticon/cottontaildb-evaluation/, Accessed July 2022

| Entity | Source | d | N | Description |
|---|---|---|---|---|
| cineast_segment | V3C1&2 | - | 2512715 | Segment metadata for V3C2. Does not contain any vectors. |
| averagecolor | V3C1&2 | 3 | 2512715 | Average colours derived from video segments. |
| visualtextcoembedding | V3C1&2 | 25 | 2506273 | Video to text co-embedding derived from vide segments. See [SGH+21a]. |
| hogmf25k512 | V3C1&2 | 512 | 2500943 | HOG [BTVG06] describtors derived from video segments. |
| inceptionresnetv2 | V3C1&2 | 1536 | 2508358 | Vector derived from last fully connected layer of a pre-trained InceptionResNetV2 applied on video segments. |
| conceptmasksade20k | V3C1&2 | 2048 | 2469844 | Image embedding for query by semantic sketch. See [RGS19]. |
| yandex_deep5M | Deep1B | 96 | 5000000 | See [BL16] for more details. |
| yandex_deep10M | Deep1B | 96 | 10000000 | See [BL16] for more details. |
| yandex_deep100M | Deep1B | 96 | 100000000 | See [BL16] for more details. |
| yandex_deep1B | Deep1B | 96 | 1000000000 | See [BL16] for more details. |

**Table 8.1 The data collections used for this evaluation.**

The binary version of Cottontail DB is started with a minimum and maximum heap size of 64 GB and 256 GB respectively. We have setup Cottontail DB's cost model so that it parallelises workloads aggressively whereas, by default, it tries to balance resource use per query and expected speed-up gained. The set of configuration parameters including explanation can also be found in Appendix A. For Milvus we use the standalone version and we followed the official tutorial for setting it up [2]. All benchmarks were conducted with the latest version 2.0.2 without further adjustments to the default configuration. In this default state, Milvus is allowed to use all available resources.

### 8.1.1 Metrics

We assume that for every query, we receive a list of results $R$ from the respective database. That list contains $K$ items $r_i \in R, i \in [1, K]$ that match the query. The returned items $r_i$ are ranked either explicitly based on score or distance (for NNS, FNS or range search) or just arbitrarily, i.e., $r_1$ comes at position one, $r_2$ at position two and so forth. The ranking is solely dependent on the database and query. Every item $r_i$ is simply a tuple containing the primary key of the retrieved database entry, which we always use to test for equality.

[2] See https://milvus.io/docs/v2.0.x/, Accessed July 2022

For every query, we are interested in assessing the efficiency and effectiveness of its execution. Efficiency can be easily gauged in terms of query execution time or latency, which is the elapsed real time in seconds between issuing the query and having received all results. This metric is simple enough to obtain and reason about and does not require further elaboration. The effectiveness of the query, i.e., the quality of the results it produces, is a bit more complicated. At a high-level, the results obtained by a query are typically compared to a resultset known to be correct – the *groundtruth*. In our case, we do not have an absolute groundtruth and we therefore rely on the results produced by sequential, non-parallelised scan of the data. [3] Given a list of results $R$ produced by the database and the groundtruth $G$, there are different ways to obtain a quality metric as, e.g., described in [WMZ10]. Those include but are not limited to precision and recall (sometimes at fixed levels), average precision, Mean Average Precision (MAP), Spearman's rank correlation and Discounted Cumulative Gain (DCG) [JK02], to name a few. Some of those metrics are merely set-based (e.g., recall and precision) while others take the position of the individual items, i.e., the ranking, into account (e.g., DCG or Spearman's rank correlation).

We limit ourselves to obtaining the recall as well as the normalised DCG [JK02] of the result $R$ compared to the groundtruth $G$ at a given level $k \leq K$ with $K = |G|$. [4] Using these two metrics, we can assess both the completeness as well as the ranking quality of the results produced by different execution strategies.

Recall at a fixed level $k$, for which a definition is provided in Equation (8.1), is purely set-based and simply checks for the existence of an element from the groundtruth $G_k$ in the result $R_k$, without taking the exact positioning of the items into account. It provides a good estimate as to whether an access method is able to produce all the relevant items.

$$\texttt{REC}_k(R_k, G_k) = \frac{|R_k \cap G_k|}{k} \tag{8.1}$$

If $R_k$ contains items that are not contained in the groundtruth (false positives) or $G_k$ contains documents that are missing in $R_k$ (false negative), this directly results in a drop in $\texttt{REC}_k$. Therefore, if all of the items $r_i \in R_k$ were to be contained in $G_k$, i.e., $R_k \cap G_k = G_k$, the recall would become 1.0 and a result could be considered a perfect match. In contrast, if none of the items in $r_i \in R_k$ were contained in $G_k$, i.e., $R_k \cap G_k = \emptyset$ then recall drops to 0.0.

[3] Having verified the correctness of Cottontail DB's search algorithms on multiple occasions over the years, we believe this to be a reasonable choice.

[4] We use $k$ as subscript to indicate, that a set has been limited to the first $k$ items.

However, recall does not provide us with any information about the ranking of the individual items. In an extreme case, two lists containing the same items but in a reversed order would produce the same recall value of 1.0. Often, though, we find that items with a low rank are more important than items with a very high rank. This is true both when considering a human user browsing a list from top to bottom, typically paying attention only to the top entries, or a use-case in which we are only interested in the top item(s) (see, for example, Section 2.3). Nevertheless, and despite these limitations, the recall metric and variants thereof are very popular in ANNS evaluations [ABF17; SWA+22].

To compensate for the absence of information about the ranking quality, we turn to a variant of the DCG at a given level $k$, which in its original form was proposed by [JK02]. Our adapted version is specified in Equation (8.2). It builds the sum over all items $r_i \in R_k$. The numerator of the expression specifies the assigned relevance of an item based on its position in the groundtruth $G_k$, which is expressed by the function $\text{rel}(r_i)$ (Equation (8.3)). If $r_i$ has rank 1 in $G_k$, it is considered most relevant and thus receives the relevance $k$. If $r_i$ has rank $k$ in $G_k$, it is not considered very releeant and therefore receives the relevance 1. If $r_i$ is not contained in $G_k$ at all (false positive), then $\text{rank}(g_i)$ returns the lowest possible relevance 0. The denominator is the logarithm of the actual rank of $r_i \in R_k$ and therefore the farther down it appears, the smaller its impact on the overall score. Consequently, items that are highly relevant but appear far down in the list are penalised.

$$\texttt{DCG}_k(R_k, G_k) = \sum_{i=1}^{k} \frac{\text{rel}(r_i) + 1}{\log_2(i+1)} \tag{8.2}$$

$$\text{rel}(r_i) = \begin{cases} k - \text{rank}_{G_k}(r_i), & \text{if } r_i \in G_k \\ 0, & \text{if } r_i \notin G_k \end{cases} \tag{8.3}$$

The reasoning behind using this DCG variant is that items with a low rank in $G_k$ are assumed to be more important than items with a very high rank. The numerator accounts for this by assigning high relevance to items that appear early on, which quantifies the gain of inspecting such an item. That gain is discounted by an item's actual rank in $R_k$, i.e., the further down in the list an item appears the larger the discount. If an item in the result $R_k$ does not appear in the groundtruth (false positive), then there is no gain at all. What is not directly captured by the DCG, however, are entries that appear in $G_k$ but that are missing in $R_k$ (false negatives).

To make the DCG values comparable across queries (potentially with different values of $k$), we normalise it by the ideal $\texttt{iDCG}_k$ (Equation (8.4)) to obtain the normalised $\texttt{nDCG}_k$ (Equation (8.5)). The $\texttt{iDCG}_k$ simply quantifies the maximum DCG that could be obtained if the ranking of $R_k$ was perfect with respect to the groundtruth $G_k$. Therefore, the normalised $\texttt{nDCG}_k$ always assumes values between 0.0 (poor) and 1.0 (perfect). All the metrics described here are also illustrated in Example 8.1.

$$\texttt{iIDCG}_k(G_k) = \sum_{i=0}^{K} \frac{k+1-i}{\log_2(i+1)} \tag{8.4}$$

$$\texttt{nDCG}_k(R_,G_k) = \frac{\texttt{DCG}_k(R_k, G_k)}{\texttt{iIDCG}_k(G_k)} \tag{8.5}$$

**Example 8.1 Result $R$, Groundtruth $G$ and obtained metrics.**

We consider the following result $R$ and the associated groundtruth $GT$ ($k = 7$), which contain the same items but in exact reverse order.

| **Rank** | **GT** | $k+1-i$ | **R** | $R_k \cap G_k$ | $\text{rel}(r_i)$ | $\log_2(i+1)$ |
|---|---|---|---|---|---|---|
| 1 | 87 | 7 | 597 | ✓ | 2 | 1 |
| 2 | 123 | 6 | 331 | ✓ | 3 | 1.58 |
| 3 | 542 | 5 | 3213 | ✓ | 4 | 2 |
| 4 | 3213 | 4 | 542 | ✓ | 5 | 2.32 |
| 5 | 313 | 3 | 123 | ✓ | 6 | 2.58 |
| 6 | 597 | 2 | 87 | ✓ | 7 | 2.81 |
| 7 | 757 | 1 | 888 | ✗ | 0 | 3 |

The values for $\texttt{REC}_7$, $\texttt{DCG}_7$, $\texttt{IDCG}_7$ and $\texttt{NDCG}_7$ according to Equations 8.1, 8.2 and 8.5 are given as follows.

$$\texttt{REC}_7 = \frac{6}{7} = 0.86$$

$$\texttt{DCG}_7 = \frac{2}{1} + \frac{3}{1.58} + \frac{4}{2} + \frac{5}{2.32} + \frac{6}{2.58} + \frac{7}{2.81} + \frac{0}{3} = 12.87$$

$$\texttt{IDCG}_7 = \frac{7}{1} + \frac{6}{1.58} + \frac{5}{2} + \frac{4}{2.32} + \frac{3}{2.58} + \frac{2}{2.81} + \frac{1}{3} = 17.23$$

$$\texttt{NDCG}_7 = \frac{12.87}{17.23} = 0.75$$

## 8.2 Execution of Multimedia Analytics Workloads

This series of experiments is about demonstrating Cottontail DB's ability to execute different types of multimedia retrieval and analytics workloads. For this, we have prepared a test protocol consisting of queries that we execute and that leverage the proposed, relational algebra extensions and the generalised, proximity based operations described in Section 6.1. In addition to the basic performance measurements, we also try to highlight different aspects of query execution and how they influence the results.

The test protocol starts by obtaining a random vector from the test collection, which will then act as a query vector $q$. We then go on to obtain the mean distance $m$ of all vectors in the collection from that vector $q$ to then perform a range search to find all entries that are between $m$ and $\frac{m}{2}$ away from $q$, which results in result $r_1$. Subsequently, we perform a simple NNS query using $q$ as a query vector, which results in $r_2$. Finally, we obtain the segment entries that are associated with the returned entries in $r_1$ and $r_2$. In summary, we execute two ordinary Boolean qureries and three proximity based queries of different types. The queries are expressed as Pseudo-SQL in Listing 8.1.

```
/* Q1a: Select a random vector from collection --> q. */
select feature from <collection> skip <random> limit 1

/* Q1b: Select mean distance from the query vector --> m*/
select mean(euclidean(feature, <q>)) from <collection>

/* Q1c: Perform a range search query --> r1. */
select id, euclidean(feature, <q>) as dst from <collection>
where dst BETWEEN (m/2.0, m) order by dst asc limit 1000

/* Q1d: Perform a nearest neighbour search query --> r2. */
select id, euclidean(feature, <q>) as dst from <collection>
order by dst asc limit 1000

/* Q1e: Obtain segment entries associated with the IDs. */
select * from from cineast_segment where segmentid in <r1 + r2>
```

Listing 8.1: Pseudo-SQL of the queries executed for the analytics workload.

We run all the queries against the V3C-based collections listed in Table 8.1, which is basically an instance of the dataset that has been prepared for *vitrivr*'s [RGT+16; GRS19a] participation to VBS 2022 [HAG+22]. vitrivr's data model has been explained in Section 4.1.1.

For the benchmark, we prepared the following indexes on the respective entities in Cottontail DB: The PQ index variant for exhaustive search (8 subspaces, 2048 centroids) and a VAF index (35 marks per vector component). In addition to the obtained metrics, we also let Cottontail DB report the execution plan it selected for a given query using its `EXPLAIN` query endpoint. These execution plans are illustrated for the purpose of discussion.

### 8.2.1 Experiment 1: Influence of Parallelism & Use of Indexes

For the first experiment, we nudge Cottontail DB into using specific, high-dimensional index structures in order to be able to compare them directly. Furthermore, we fix the level parallelism to a given value between 2 and 16. Both can be done by providing *hints* with the query, which are respected by the query planner as far as possible given the solution space. The results of these measurements are presented in Figure 8.1. Unsurprisingly, the total execution time for the entire workload goes up as the dimensionality increases. There is one anomaly for *inceptionresnetv2* ($d$ = 1536) and *conceptmasksade20k* ($d$ = 2048), where a sequential scan seems to be faster for the latter for all three types of proximity based queries (i.e., Q1b, Q1c and Q1d). While somewhat counter-intuitive, this can be explained by the structure of the data itself: The vectors contained in *conceptmasksade20k* are very spare and can therefore be compressed very efficiently by Cottontail DB's rather naive compression scheme. In a quick follow-up experiment, we observed a compression to 550 B on average, which is a reduction of roughly 95%. In contrast, *inceptionresnetv2* contains very dense vectors and compression is not possible at all. Therefore, the full 6144 B plus an additional 5 B overhead introduced by the compression must be read from disk. While this is a glitch in the implementation, its impact on query performance demonstrates the importance of IO and efficient on-disk representations.

We can also see from Figure 8.1, that all the workloads, with the exception of the Fetch (Q1a) and the Select (Q1e) query, clearly benefit from increasing the level of parallelism both for full entity scans as well as index scans, which confirms the need for an efficient parallelisation and distribution model [Gia18]. The two only exceptions are rather simple queries that cannot be parallelised effectively because they are either inherently sequential (Q1a) or because the partitioning model of Cottontail DB does not allow for parallelisation (Q1e).[5] The overall execution time of the workload is clearly determined by the proximity based queries.

[5] Q1e leverages a $B^+$-tree index, for which a scan cannot be parallelised in its current version.

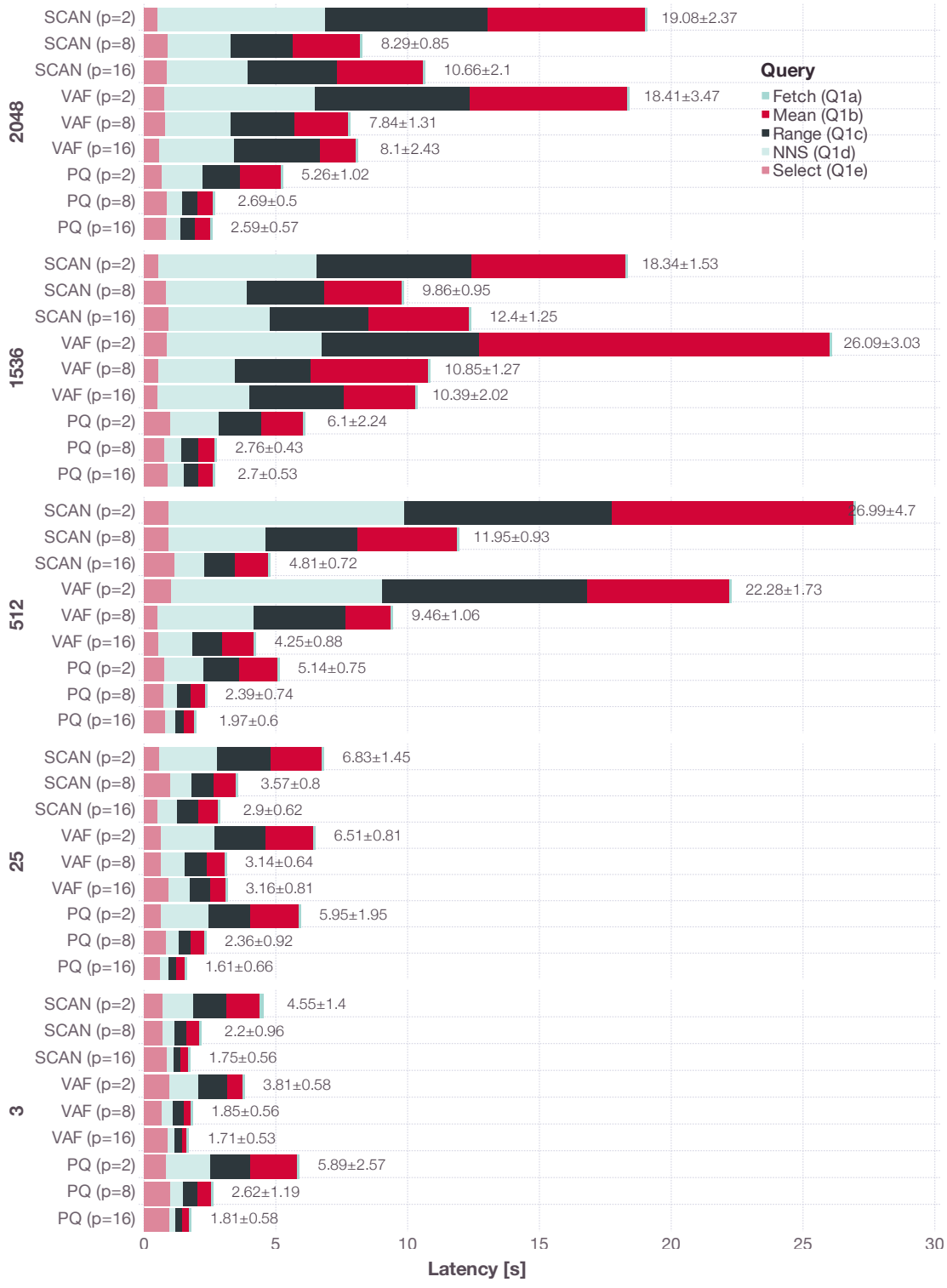
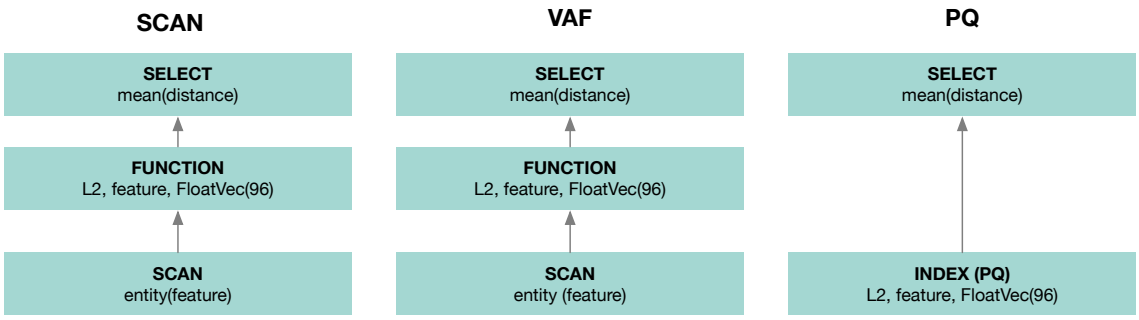
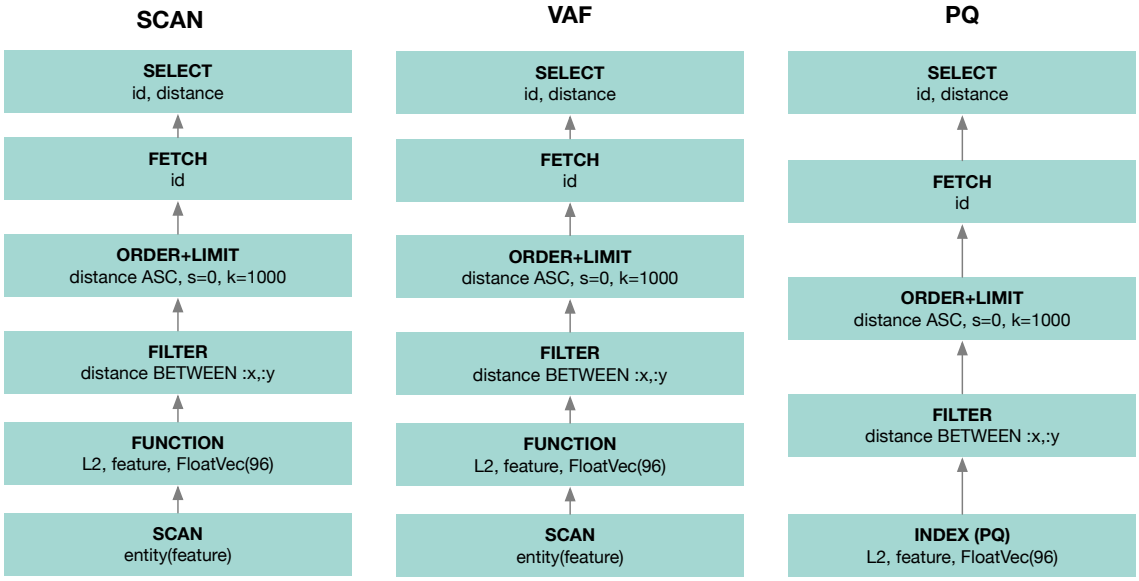


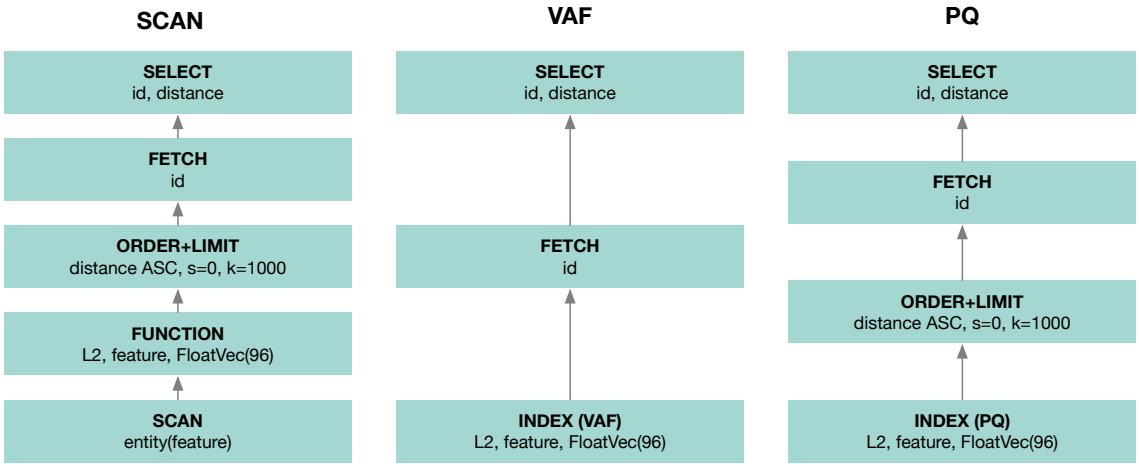

**Figure 8.1** **Latency in seconds (x-axis) on different entities using different access methods and levels of parallelisation (y-axis) for the analytics workloads during the first experiment. The individual queries Q1a-Q1e are highlighted in different colours. The use of the VAF and PQ index was enforced using query hints.**




((a)) Execution plans for mean query (Q1b)




((b)) Execution plans for range query (Q1c)




((c)) Execution plans for NNS query (Q1d)

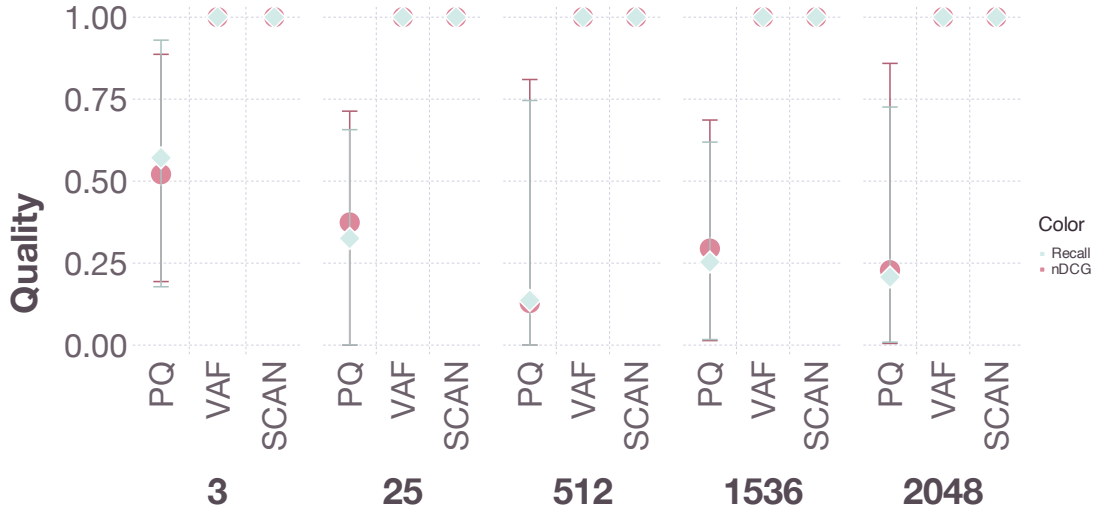
**Figure 8.2 Execution plans produced by Cottontail DB for the mean (Q1b), range (Q1c) and NNS (Q1d) query workloads during the first experiment. The use of the VAF and PQ index was enforced using query hints.**

An aspect that we would like to draw the attention to is the influence of indexes: The Mean (Q1b), Range (Q1c) and NNS (Q1d) queries all seem to benefit from the PQ index, which, according to Figure 8.1, provides the lowest latency across all entities and for every degree of parallelisation. This is confirmed when cross-examining the query plans produced by Cottontail DB, which are listed in Figure 8.2. The plans for the Q1b (8.2(a)), Q1c (8.2(b)) and Q1d (8.2(c)) indeed use the PQ, if the respective hint is present. This is possible because PQ can be used to perform a class 1 index replacement (see Definition 6.9).

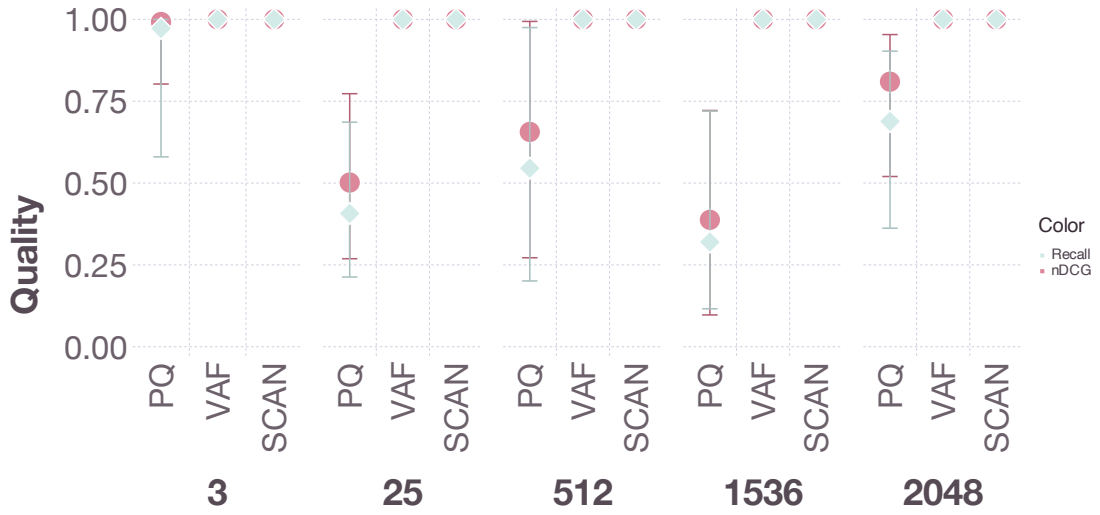
However, the observed speed-up is traded for a non-negligible, negative impact on quality in terms of recall and nDCG, as can be seen in Figure 8.3. Slightly higher values for nDCG provide us with an indication that mismatches occur further down in the ranking and may therefore be less relevant in practice. What is striking, though, is the very large spread for both recall and nDCG across all collections, which indicates that the performance of an index is highly dependent on the concrete query vector and thus hard or even impossible to predict reliably. Furthermore, we can observe that the negative impact on quality seems to be larger for range (Q1c, 8.3(a)) than for NNS (Q1d, 8.3(b)) queries for both recall and nDCG. This can be explained by the distortion of the approximate distance as a side-effect of PQ, which has been reported by [JDS10] and which is obviously very relevant for range queries. We see this as a confirmation of our proposition, that quality metrics and statistics for the purpose of query planning should be obtained for a type of execution plan rather than globally per index (see Section 6.2.2). Additionally, we can observe that the same index configuration for PQ yields very different results for the different collections. Unfortunately, there does not seem to be an apparent correlation between the dimensionality and the quality of PQ index results, which implies that the quality depends on more than just the size of the vectors.

In contrast, the VAF index can only provide speed-up for NNS queries (Q1d), since its use is limited to class 3 replacements (see Definition 6.11) and the current implementation does not support range queries [6]. Again, this is confirmed by the query plans shown in Figure 8.2, where Cottontail DB resorts to a sequential scan in cases where the VAF cannot be employed. The speed-up gained by using VAF seems rather small and it is directly related to the efficiency of the filtering. Over the different runs, we observed VAF filter efficiency values of between 80% and 99%. Again, there seems to be no apparent connection between dimensionality and the effectiveness of the filtering step.

[6] The latter is a limitation of our implementation and could be added, according to [WSB98]




((a)) **Quality for range query (Q1c)**




((b)) **Quality for NNS query (Q1d)**

**Figure 8.3 Quality of results (x-axis) in terms of recall (mint) and DCG (red) on different entities using different access methods (y-axis) for the range (Q1c) and NNS (Q1d) query workloads during the first experiment. The use of the VAF and PQ index was enforced using query hints.**

We must assume, that the quality of all the tested index structures can be fine-tuned by chosing appropriate values for the hyperparameters. Since these values seem to depend on a multitude of factors (e.g., collection size, distribution of the data, dimensionality), this may not be a straightforward undertaking. However, learning an optimal set of hyperparameters for a given collection and index could be an interesting research direction for the future, very similar to learned index structures proposed by [KBC+18].

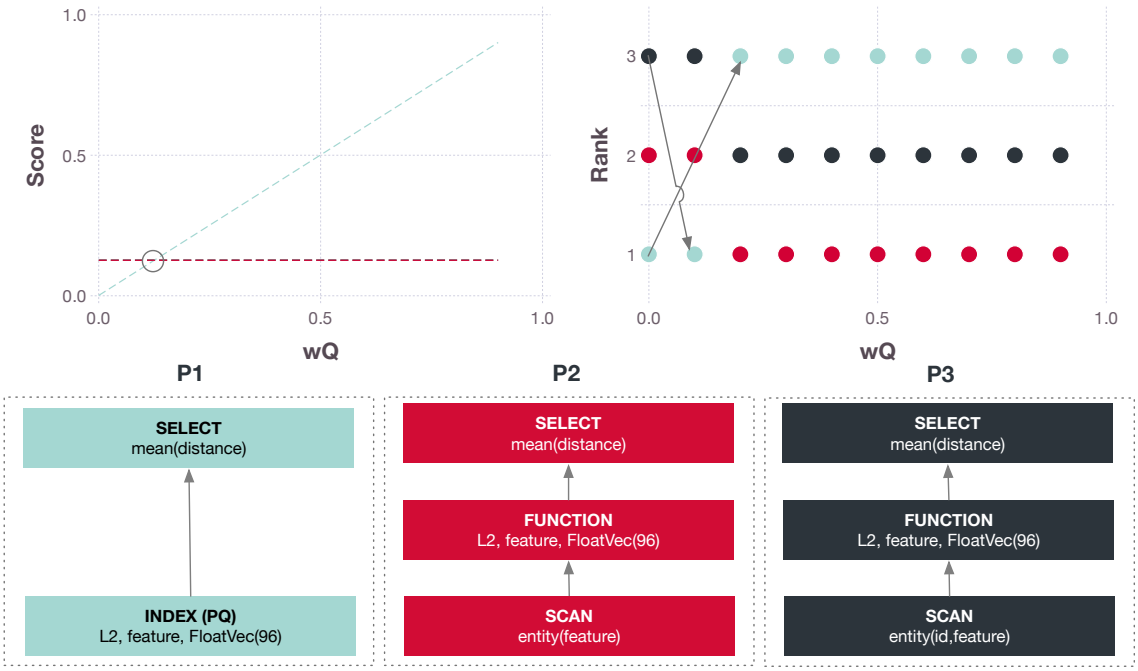



**Figure 8.4 Score and rank of different query plan options for the mean query (Q1b). The prefered option for $w_Q = 0.0$ involves the scan of a PQ index. As $w_Q$ goes up, the cost of impaired quality increases and a full entity scan ($P_2$) takes over, since none of the available indexes can satisfy the query. The scores for $P_2$ and $P_3$ do not depend on the quality parameter and incur almost the same cost, with $P_2$ being the less expensive variant.**

### 8.2.2 Experiment 2: Influence of Cost Policy

So far, we have forced Cottontail DB into using one index over another by providing query hints that make an explicit choice. For this second experiment, we delegate the decision to Cottontail DB and observe how changing parameters in the cost policy influence plan selection. The idea behind the cost policies was explained in Section 6.2. Basically, they assign weight to the parameters of the cost model – the cost of IO, CPU, memory use and reduction in quality – and can be used to steer the planner's behaviour depending on the use-case. For the experiment, we fixed the values of $w_{\texttt{CPU}} = 0.3$, $w_{\texttt{IO}} = 0.6$ and $w_{\texttt{MEM}} = 0.1$ respectively (Cottontail DB's default values) via the respective query hint. We then used the `EXPLAIN` endpoint to generate and print the execution plans as we increased the weight of the quality parameter $w_Q$ from 0.1 to 0.9 in steps of 0.1. The results are depicted separately for the mean (Q1b, 8.4), range (Q1c, 8.5) and NNS (Q1d, 8.6) queries, for which we plot the scores and the ranking assigned to the individual plans by Cottontail DB's query planner. Furthermore, we visualise all query execution plans that appear in the top three ranks at some point.

For Q1b (Figure 8.4) we see that at least three different options exists and that by default, i.e., for $w_{\mathtt{Q}} = 0.0$, the option involving the PQ index ($P_1$) is favoured. This is consistent with the findings from our first benchmark, where we demonstrated that PQ is the fastest of the available options. As long as the quality weight parameter is low ($w_{\mathtt{Q}} \in [0.1, 0.2]$), $P_1$ remains the plan with the lowest score and thus the selected candidate. As $w_Q$ increases, the score of $P_1$ increases as well and it moves up in the ranking and plan $P_2$, which involves a full table scan, takes over the lead. The two alternatives $P_2$ and $P_3$ are almost equivalent with the only difference that $P_3$ includes the `id` column in the initial scan, which is not required to calculate the mean and can therefore be optimised out (the application the deferral rules described in Section 7.2.2). Consequently, $P_3$ is slightly more expensive and $P_2$ is selected instead.

A very similary behaviour can be observed for Q1c (Figure 8.5). Over the course of the measurements, six options appear in the top 3 ranks. Again, we start with $w_{\mathtt{Q}} = 0.0$, where $P_1$ is the fastest option and again, it involves a scan of the PQ index. $P_2$ and $P_3$ are different variants involving the same index scan but using different intermediate operations and therefore differ in the fetching of required columns and sort algorithms (application of the deferral and implementations rules described in Section 7.2.2). As $w_Q$ increases, $P_1$, $P_2$ and $P_3$ disappear from the top 3 ranks, since all three plans use the same index and therefore result in impaired quality. $P4$ – which is the query plan involving the evaluation of a full entity scan – takes over and keeps the lowest rank. Again, $P5$ and $P6$ are less optimal variants of $P4$ and are therefore not considered.

For both Q1b and Q1c using PQ or a full entity scan were the only available options due to the constraints discussed in Section 6.1.4. For Q1d (Figure 8.6) we see a third option appear, which is the use of a VAF index ($P3$). As $w_{\mathtt{Q}}$ exceeds 0.1, the scan of the VAF index becomes the prefered candidate and $P_1$ and $P_2$ become the second and third options. As $w_{\mathtt{Q}}$ moves beyond 0.2, $P_1$ and $P_2$ disappear from the top 3 and the entity scan becomes the prefered alternative to the VAF index scan. However, since the VAF index does not incur a quality cost, it remains the favoured option.

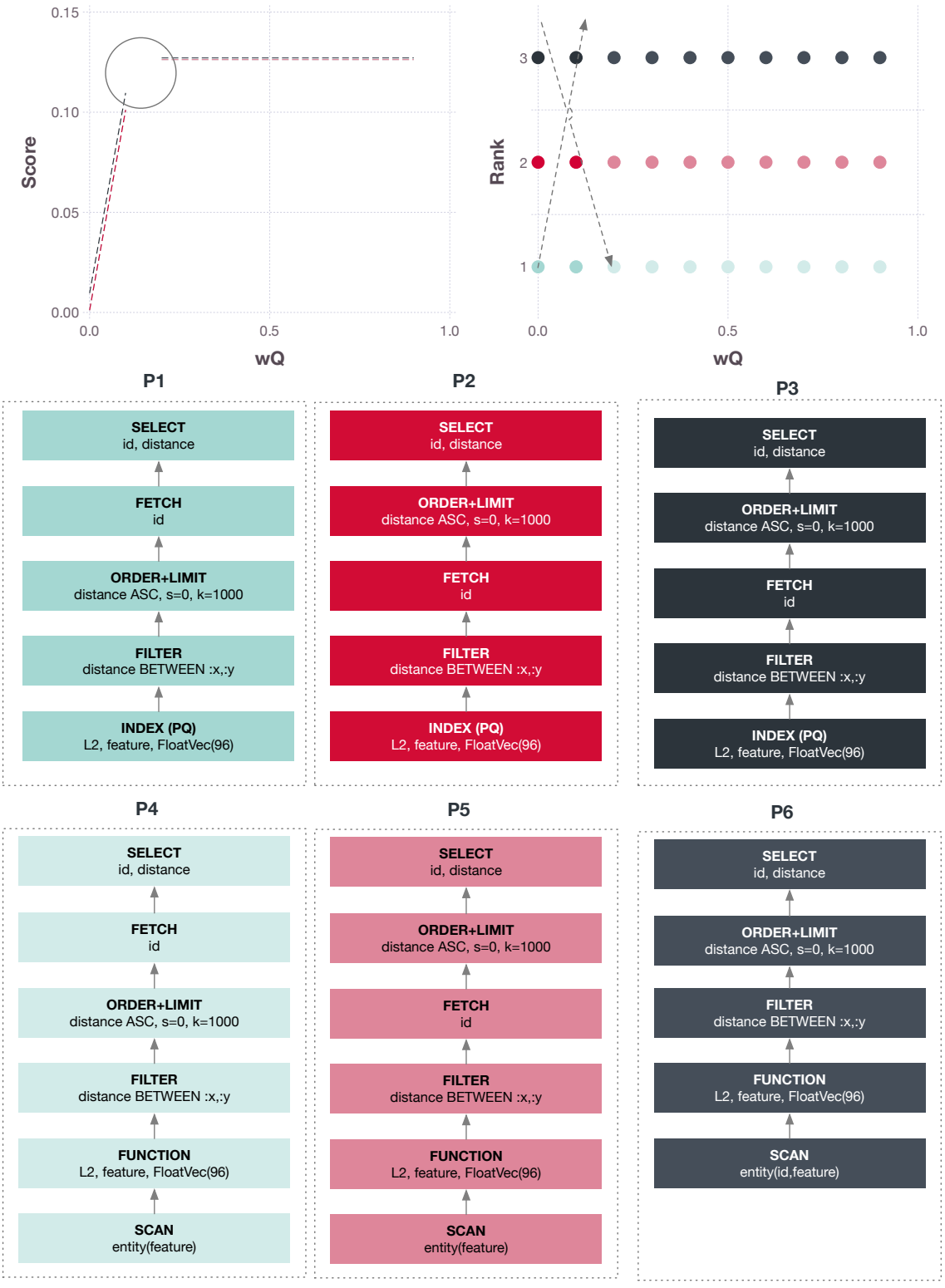



**Figure 8.5** **Score and rank of different query plan options for the range query (Q1c). The prefered option for $w_Q = 0.0$ involves the scan of a PQ index. As $w_Q$ goes up, the cost of impaired quality increases and a full entity scan takes over, since none of the available indexes can satisfy the query. The scores for $P_4$ through $P_6$ do not depend on the quality parameter and incur almost the same cost, with $P_4$ being the least expensive.**

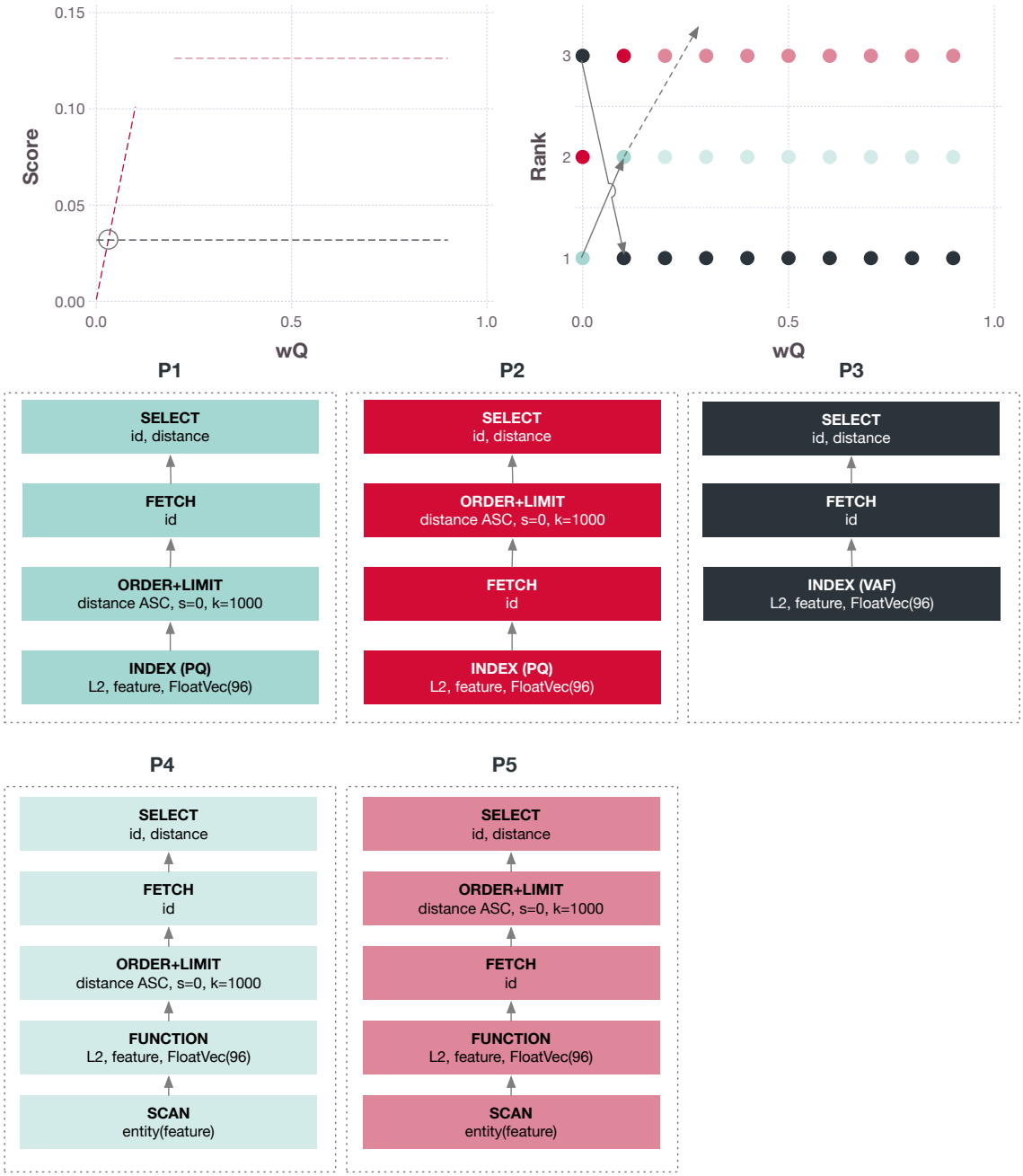



**Figure 8.6** **Score and rank of different query plans options for the NNS query (Q1d). The prefered option for $w_Q = 0.0$ involves the scan of a PQ index. As $w_Q$ goes up, the cost of impaired quality increases and a VAF index scan ($P_3$) takes over, which can also satisfy NNS queries. Since VAF does not incur a quality cost, it remains the favoured option.**

### 8.2.3 Benchmark 3: Influence of Optimisation

We have demonstrated in the first experiment, that high-dimensional index selection has a considerable impact on query execution performance. However, as described in Chapter 7, in addition to selecting indexes, the query planner also performs other optimisations of the plan before executing it. For this experiment, we bypassed plan optimisation completely and observed the execution time for an unoptimised plan – which is a naive implementation of the canoncial operator tree. All experiments were performed with parallelisation switched off, i.e., single-threaded. The results are depicted in Figure 8.7.

We can see from this graph, that optimisation seems to have a larger effect on latency for proximity based queries (Q1b to Q1d) involving high-dimensional vectors than it does for low dimensional ones. However, optimisation also seems to introduce a larger variance in latency, making it less predictable. Both effects can be attributed to the column-access deferral optimisation rule, which pushes access to columns that are not required for query execution up in the tree until after a $(s, k$-)selection has been executed. The idea is to reduce or even avoid access to data that is not needed and therefore the number of IO operations. This optimisation seems to be highly relevant for large vectors but it diminishes as dimensionality decreases. We suspect that the page cache, which – given the huge amount of memory available – can keep many of the requested pages in memory, is responsible for this behaviour. When larger entities are being scanned, the cache becomes more saturated and the likelihood of a cache miss becomes higher resulting in a higher, overall latency. This effect is of course amplified if more than one column is being scanned, as in the unoptimised case.

As a second observation, we see that optimisation consistently benefits the Select (Q1e) query, where we load segment entries that match the preceding query results. The explanation here is as simple as unexciting: The select query (Q1e) benefits from a $B^+$-tree index scan in the optimised case, while the unoptimised case uses a full entity scan. Last but not least, we see that the impact of optimisation is slight larger for range (Q1c) and NNS (Q1d) queries than it is for the mean query. This can be attributed to a second optimisation, which combines the sorting and limiting used for Q1c and Q1d into a single step that uses a specialised heap data structure. This sort algorithm can be executed more efficiently and uses less memory than naive sorting. We expect the impact to be even larger for setups that do not have the same amount of memory and must therefore resort to on-disk sorting.

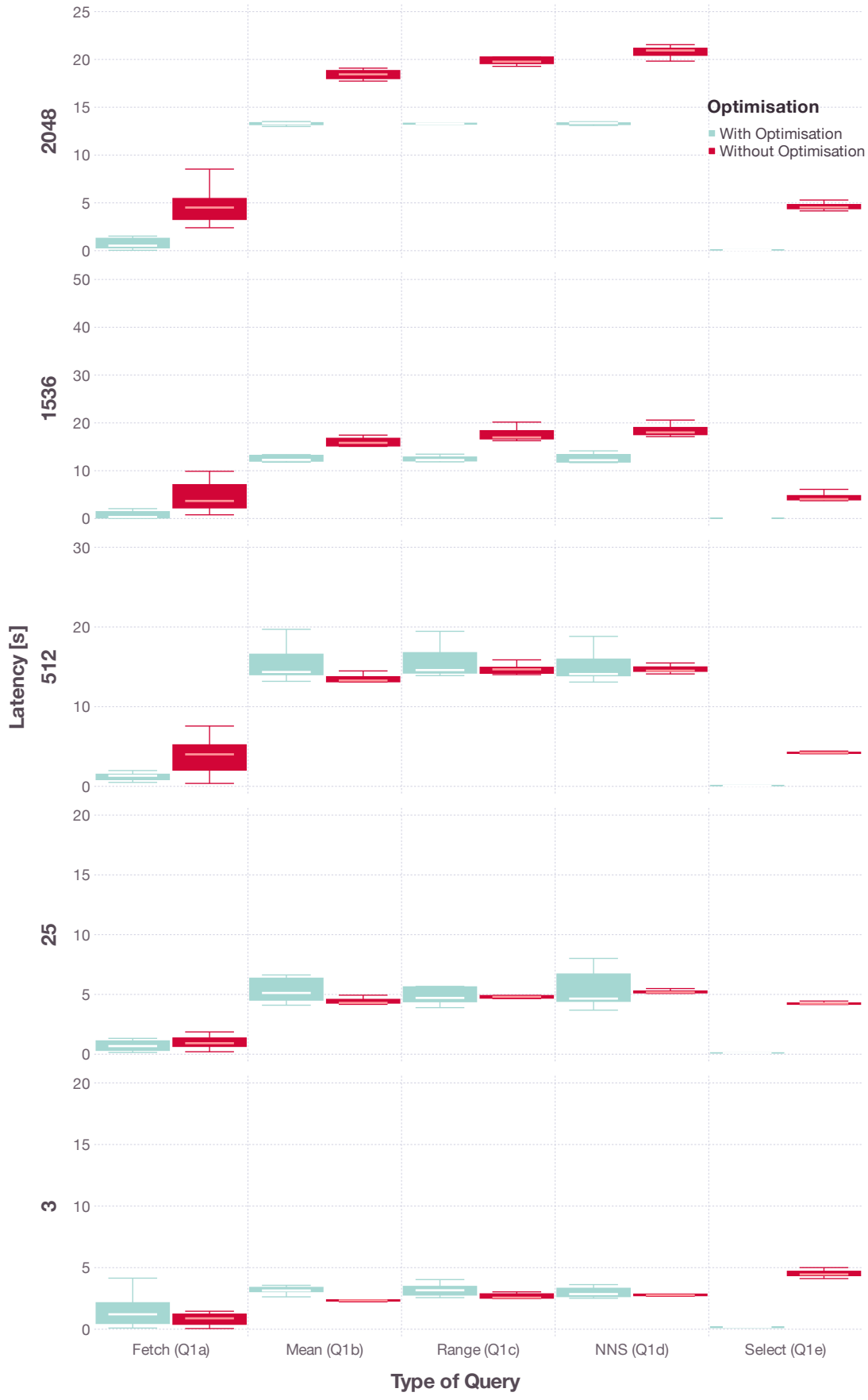



**Figure 8.7** **Latency in seconds on different collections (y-axis) for the query workloads Q1a through Q1e (x-axis) with (mint) and without (red) query optimisation.**

## 8.3 Large-Scale Similarity Search

This experiment is a direct comparison of Milvus and Cottontail DB and is about mere performance. We execute all queries on prepared shards of the Deep1B [BL16] dataset ($d = 96$) with $k = 1000$ using different execution strategies and compare the obtained metrics. The shards contain 5 million, 10 million, 100 million and 1 billion 96-dimensional `float` vectors and are stored in dedicated collections (Milvus) or entities (Cottontail DB). The three types of queries are: (i) A simple NNS query that only returns the primary key and the distance (Q2a), (ii) a NNS query that additionaly returns the query vector (Q2b), (iii) a NNS query with a Boolean filter (Q2c, hybrid query in Milvus terminology).. The Pseudo-SQL is provided in Listing 8.2. We use the query vectors provided with the Deep1B dataset and establish a groundtruth by executing a brute-force search.

```
/* Q2a: Simple NNS without feature vectors. */
select id, euclidean(feature, <query>) as dst from <collection>
order by dst limit 1000

/* Q2b: NSS that returns feature vectors. */
select id, feature, euclidean(feature, <query>) as dst from <
collection> order by dst limit 1000

/* Q2c: Hybrid query without feature vector. */
select id, euclidean(feature, <query>) as dst from <collection>
where category = <category> order by dst limit 1000
```

Listing 8.2: Pseudo-SQL of the queries executed for this measurement.

### 8.3.1 Cottontail DB

For this benchmark, we prepared a selection of indexes on the respective entities in Cottontail DB: Two variants of a PQ index, once organised as a list for exhaustive search (8 subspaces, 256 centroids) and once organised as an inverted list for approximate search (8 subspaces, 256 centroids and 256 coarse centroids) and a VAF index. Furthermore, we created a $B^+$-Tree index on the column `categories`, which we expect to be beneficial for Boolean search. We again use query hints to nudge Cottontail DB into using certain high-dimensional index structures so as to be able to compare the different query execution strategies. Accross all workloads, we compare the performance of four different types: Sequential scan and the scan of the VAF and the (IVF-)PQ indexes.

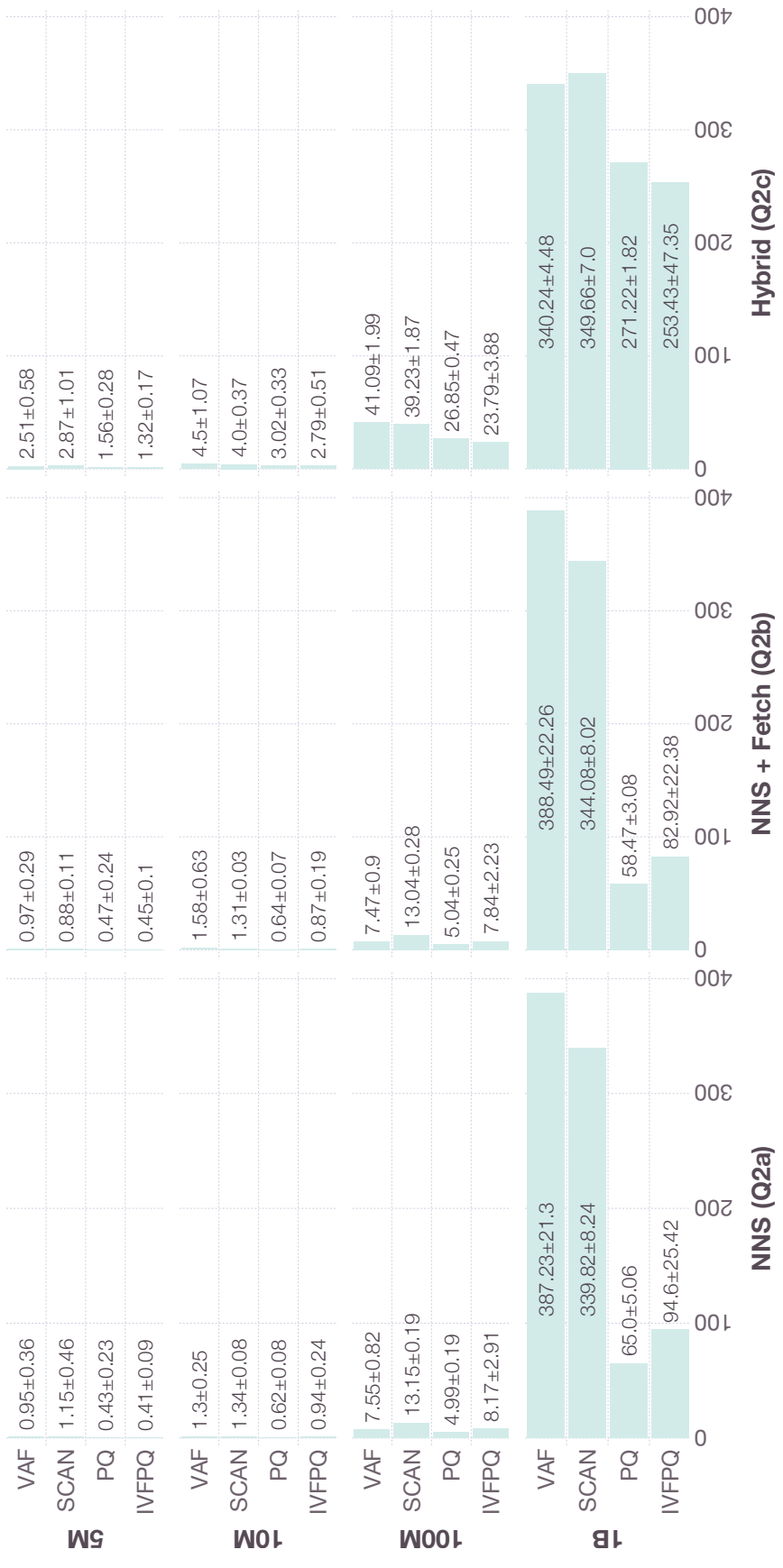
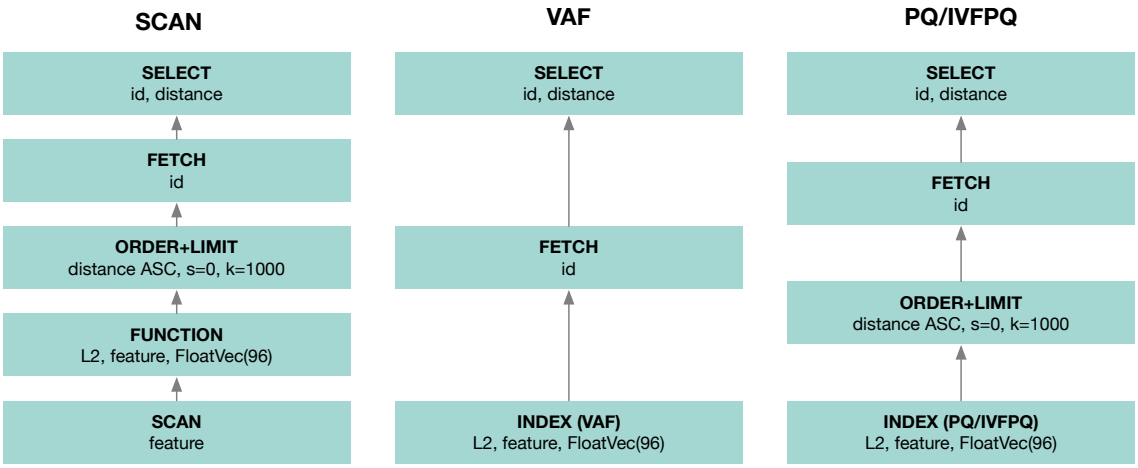



**Figure 8.8 Cottontail DB's latency in seconds for different workloads (x-axis) on different shards of the Deep 1B data set using different execution strategies (y-axis).**

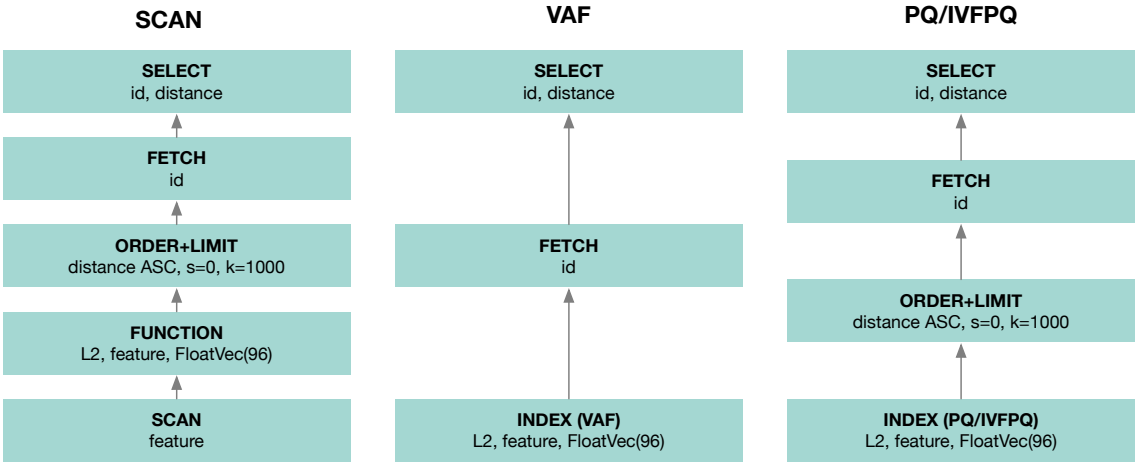
In our first experiment, we executed the simple NNS workload (Q2a). The results are depicted in Figure 8.8. Brute-force search took between $0.95 \pm 0.36$ s (5 million) and $339.82 \pm 8.24$ s (1 billion). We can observe, that the VAF index brings only minor advantages for the smaller datasets but reduces execution time by almost 4 s for the 100 million entries dataset. This is an artifact of the query execution engine, which uses less than the assigned 32 workers for the smaller collections because of the derived CPU costs. Unfortunately, VAF seems to perform poorly for the 1 billion entries dataset, where it is outperformed by a entity scan by more than 40 s. We believe this to be due to the many random accesses necessary for entries that could not be filtered out. If one considers a filter efficiency of approximately 90% as advertised by [WSB98], one must still fetch 100 million entries through random access, which seems more expensive than simply scanning the entire collection.

We can also cleary see, that using the PQ and IVFPQ index reduces the execution time significantly, but at the cost of impaired quality, which is below 0.5 on average for both recall and nDCG, as can be seen in Figure B.1 (see Appendix B). Similarily to the analytics workload, we observe quite a large variance between individual queries. We expect, however, that the overall performance can be optimised by tuning the hyperparameters used upon index construction, specifically, the number of coarse and fine centroids. It is also worth noting, that the IVFPQ index does currently not support intra-query parallelism due to Cottontail DB's partitioning model, i.e., the execution times we see for the IVFPQ index ($94.6 \pm 25.42$ s for 1 billion entries) are always single-threaded. The execution plans are depicted in Figure 8.9(a) and are fairly unsurprising: The use of VAF and PQ constitutes a class 3 resp. class 1 index replacement according. The fetching of the `id` columns is pushed down to after the sort and limit operations, since this significantly reduces the amount of IO (only 1000 fetches instead of scanning millions of entries).

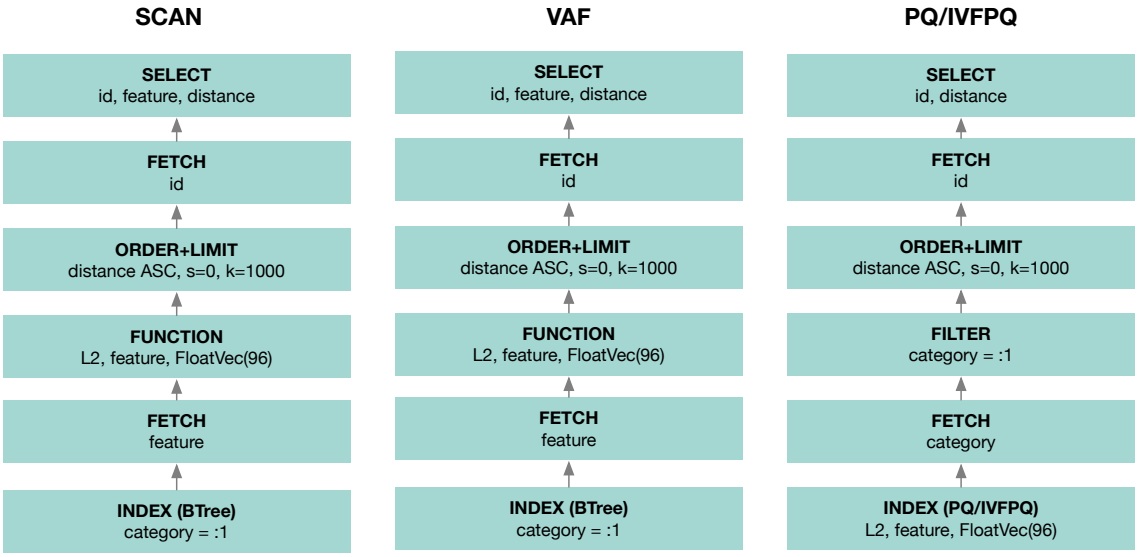
Our second experiment involved the combined NNS and fetching of the resulting vectors. One can see in Figure 8.9(b) that in terms of execution plan, accessing this additional column does not make a difference. Since both the plan involving the entity scan as well as the VAF index scan produce the `feature` column early on, there is not need to fetch it later. Only for the PQ index – which uses a distance approximation that is obtained without accessing the `feature` column – an additional fetch is required. Consequently, the execution times are very similar to the simple NNS as one can see in Figure 8.8 (NNS + Fetch). The numbers for a sequential scan range between $0.68 \pm 0.06$ s (5 million) and




**((a)) Simple NNS.**




**((b)) NNS and fetching of vectors.**




**((c)) Hybrid query with Boolean filter.**

**Figure 8.9 Execution plans produced by Cottontail DB for different query workloads prior to intra-query parallelisation. The use of the VAF and PQ index was enforced using query hints.**

$344.08 \pm 8.02$ s (1 billion). The impact of the additional fetching of 1000 feature columns is negligible for the PQ index, which can be attributed to the fact that the fetch is pushed until after the $s, k$-selection by the optimiser. We expect the impact of this fetching step to become more important for larger values of $k$ but it should remain negligible since typically, $k << N$. The quality metric shown in the appendix (Figure B.2), are pretty much comparable to those of the first experiment, exhibiting the same, low numbers and high variance.

For the final experiment, we executed hybrid queries, i.e., NNS was restricted to a subset of the data that matches a Boolean predicate. The predicate involves a simple equality check that should effectively limit the exhaustive NNS to roughly $\frac{1}{10}$ of the collection. The values for a sequential scan range between $1.47 \pm 0.15$ s (5 million) and $349.66 \pm 7.0$ s (1 billion) with very similar numbers for the VAF and PQ indexes. To make sense of these values, we again turn to the query plans illustrated in Figure 8.8 (Hybrid). The sequential scan was executed with the help of a $B^+$-tree index, which speeds-up the Boolean filtering. Unfortunately, the current implementation of the VAF index cannot natively accomodate Boolean predicates, since it constitutes a class 3 index replacement according to Definition 6.11 and therefore, a special implementation of the index would be required[7]. Consequently, a $B^+$-tree scan was executed instead. In contrast, the PQ index can be combined with Boolean predicates, since it constitutes a class 1 index replacement according to Definition 6.9. However, one can see in Figure 8.8 (NNS + Fetch) that the advantage of using the PQ index is negligible. This can be attributed to the fact that a fetch operation for the `category` column must be executed for every entry before executing the filtering. The IO overhead is much larger than the speed-up gained through the PQ index. This is confirmed by the numbers we see for the IVFPQ index, which restricts the scan to a small subset of the collection. Similarly to the IVFPQ index, the $B^+$-tree index does currently not allow for parallel evaluation due to Cottontail DB's partitioning model. This is severly limiting for the hybrid queries and something that should be addressed in the future. The nDCG and recall values are again shown in Appendix B (Figure B.3).

In summary, all modes of operation fail to deliver interactive latency for very large collections, which currently is the price that must be paid for being bound to disk. It is to be noted, however, that Cottontail DB does not employ any caching other than a page cache. We presume that such advanced caching of, e.g., signatures could result in a considerable speed-up.

[7] It is, however, unclear if such an implementation would be beneficial in practice.

### 8.3.2 Milvus

For executing queries in Milvus, the mode of operation is a bit different than it is for Cottontail DB, since Milvus requires a data collection to be available in main memory, in order to be able to query it. Therefore, every query consists of two steps: Loading the collection and then executing the query once the collection is ready. We have obtained the ellapsed time for loading and query execution separately, so that we can reason about the individual components. The way queries can be specified is outlined in Listing 8.3. We show the Python instead of the Java syntax, because it is less verbose and more readable, assuming, that the functionality of the two client libraries is identical.

```
from pymilvus import Collection

# Load an existing collection.
collection = Collection("images")
collection.load()

# Perform search.
query = [[0.0, 0.0]]
params = {"metric_type": "L2", "params": {"nprobe": 10}}
results = collection.search(
    data=query, anns_field="feature", param=params, limit=10
)
```

Listing 8.3: Example of a similarity search query in a collection “images” using a 2-dimensional query vector and the Euclidean distance. Before executing the query, the data collection must be loaded.

Accross all workloads, we compared the performance of two different types of indexes: The `FLAT` index is the equivalent of a sequential scan and it is the only index in Milvus that guarantees a recall of 1.0. The `IVF_SQ8` index uses an inverted file of clusters and limits search to a subset of these clusters based on the parameters provided by the user and an initial distance calculation between the query and the cluster centers. The approach is comparable to cluster pruning [CPR+07] or the two stage quantisation process described in [JDS10], where each vector is mapped to an inverted list using a coarse quantiser. In addition to limiting the search space, the `IVF_SQ8` index also compresses the 4 B (`float`) into a 1 B (`int8`) presentation, which significantly reduces memory and CPU usage by up to 70%. We built the `IVF_SQ8` index beforehand using 4096 clusters and instructed Milvus to consider 1024 and 2048 (query parameter `nprobe`) clusters respectively, when searching the collection.

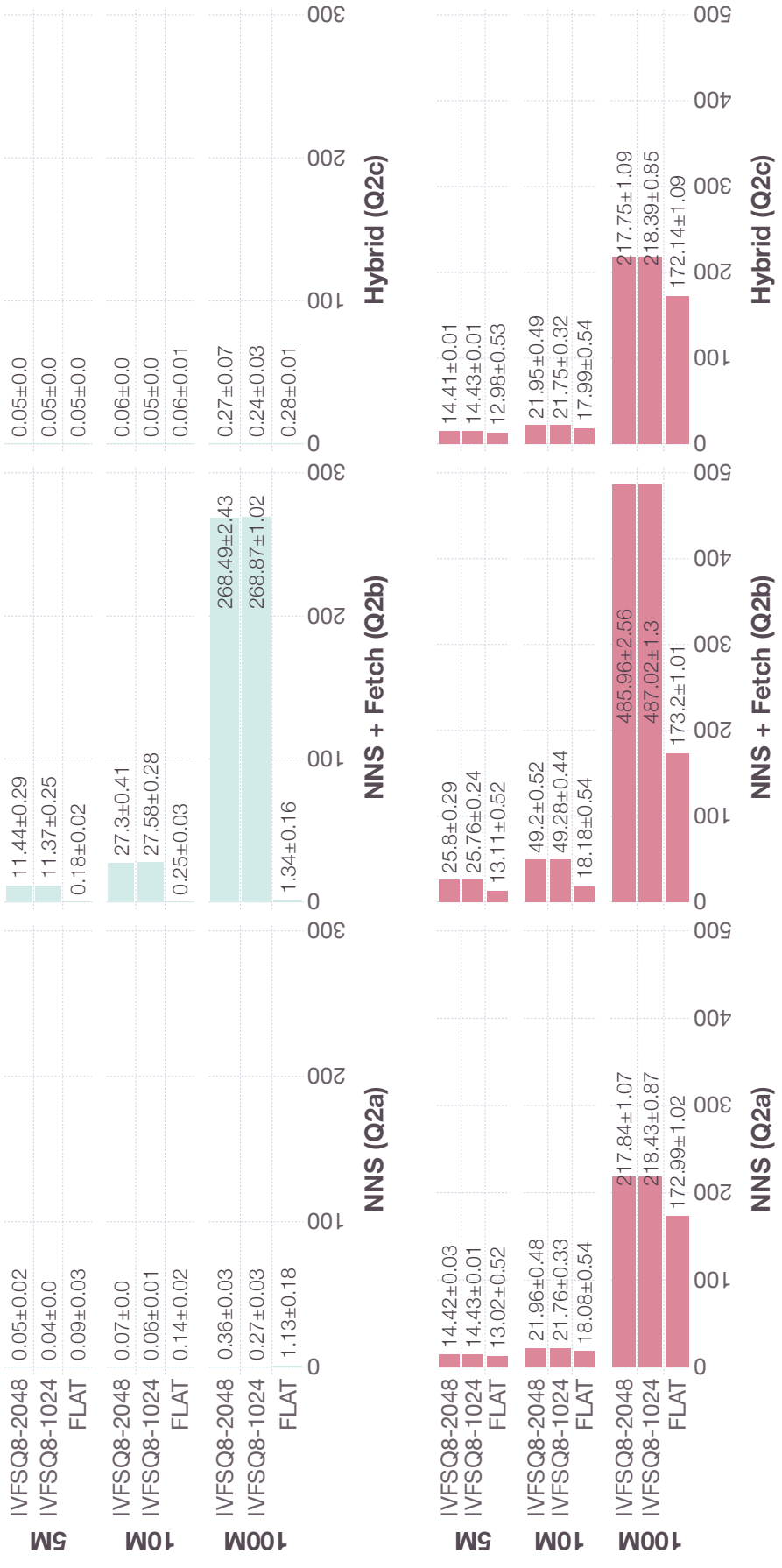



**Figure 8.10 Milvus's latency in seconds for different workloads (x-axis) on different shards of the Deep 1B data set using different access methods (y-axis), with (red) and without (mint) time used for loading the data from disk.**

In our first experiment we executed the simple NNS workload (Q2a). The results are depicted in Figure Figure 8.10. We can summarise that even for brute-force search (`FLAT`), query execution speed is very impressive once a collection has been loaded into main memory. For our test collection, it took between $0.09 \pm 0.03$ s (5 million) and $1.3 \pm 0.18$ s (100 million) to execute a query, all the while retaining a perfect recall and DCG of 1.0. However, if one considers time required to load a collection from disk to be part of the query time, these numbers rise to between $13.02 \pm 0.52$ s and $172.99 \pm 1.02$ s. If one loads a collection once to then execute many queries thereafter, the performance provided by Milvus is very desirable as the cost of loading the collection can be amortised over time. However, if a large number of collections must be queried without the ability to predict which one, and not all collections can be pre-loaded, query exection time is expected to deteriorate. Furthermore, the requirement to load the collection prior to querying it turned out to be an insurmountable roadblock for the shard that contained 1 billion entries. Milvus was unable to load the collection as it ran out of available memory. We can confirm, that Milvus indeed used-up the entire 376 GB of ree RAM on the node.

In an attempt to aleviate the memory pressure, we also considered the `IVF_SQ8` index, which seems a logical choice due to its data compression characteristics. Unfortunately, using this index instead did not resolve the problem of collection loading, since apparently, Milvus always loads the indexes and the original data. Nevertheless, there are two interesting insights from the mesurements: (i) `IVF_SQ8` brings considerable speed-up for raw query execution time, especially for larger collections (roughly 1 s faster on 100 million shard), (ii) however, it exacerbates the performance impact of collection loading, since the indexes must apparently be loaded in addition to the raw data.

The second experiment involved the combined NNS and fetching of the resulting vectors. Unfortunately, Milvus currently does not support the returning of vectors in a NNS, which is why this is a two-step process. First, the NNS is executed to subsequently lookup the vectors for the resulting primary key's in a second query. We measure the time for both steps. The results are depicted in Figure 8.10 (Q2b). The additional fetching step, while cumbersome, does not seem to have a negative impact in cases where the `FLAT` index is used. However, use of the `IVF_SQ8` index seems to lead to significant deterioration of the the query performance for the fetching step, adding between 10 s (5 million) and more than 250 s (100 million) to total query execution time. We do not have an explanation and consider this to be abnormal behaviour.

Last but not least, we did execute hybrid queries, wherein NNS was restricted to a subset of the data that was filtered by a simple predicate. In Milvus, this is a query primitive that is part of the similarity search API. The results are visualised in Figure 8.10 (Q2c) and do not hold any surprises. In-memory query execution speed is again outstanding whereas time required to load a collection is significant. Interestingly, the `IVF_SQ8` did not seem to have a negative impact on query execution speed, which confirms our suspicion that the deterioration observed in the second experiment must be considered a bug.

We refrain from visualising nDCG and recall values but can report that both always remained at 1.0 for `FLAT` and around 0.99 for all queries that used the `IVF_SQ8` index.

#### 8.3.2.1 Qualitative Assesment

While very convincing in terms of sheer execution speed, the current version of Milvus also has some limitations that we list here for future reference:

- Milvus currently only supports the Euclidean and Inner Product distance for single-precision floating point embeddings. There is no way to use other metrics or data types.
- Milvus does not support proximity-based search strategies other than NNS. However, according to the official GitHub issue tracker, [8] range search is due for the 2.2.0 release.
- Milvus cannot retrieve the feature vectors as part of a NNS query, i.e., they must be fetched in an additional lookup step based on the primary key. This issue is also due for being addressed in the 2.2.0 release. [9]
- Milvus can only maintain a single index per feature column. This effectively limits the set of available distance functions to one, since most indexes must be trained for a specific distance.

In addition, we have observed that as Milvus uses up all the memory during collection loading, it starts to (mis-)behave eraticly making it difficult to unload the collection again. In some of our runs, Milvus started to reload the same collection after being restarted after a crash, effectively rendering the entire instance unusable for several hours.

[8] See https://github.com/milvus-io/milvus/issues/17599/, Accessed August 2022
[9] See https://github.com/milvus-io/milvus/issues/16538/, Accessed August 2022

## 8.4 High-Dimensional Index Maintenance

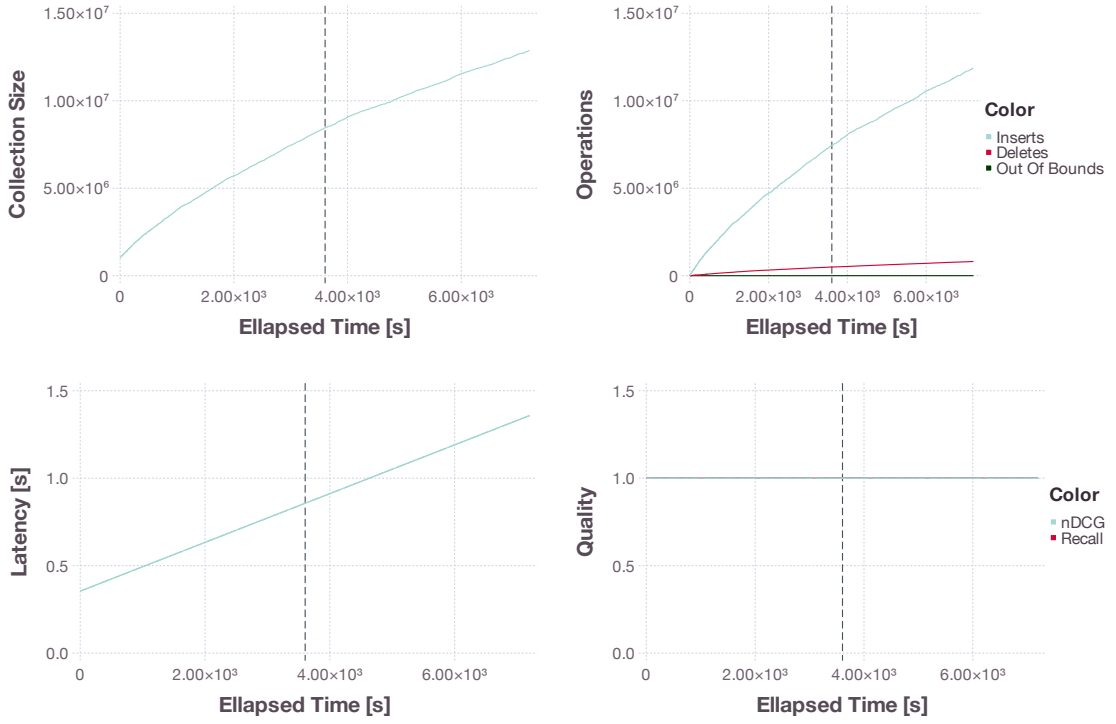
With this evaluation, we demonstrate Cottontail DB's ability to cope with dynamic, high-dimensional data. It is a test of the index maintenance mechanism described in Section 6.3 and the transaction support Cottontail DB provides. The setup of this experiment is slightly different than that of all previous experiments, in that it involves different workloads that run concurrently. The evaluation starts by creating an entity and loading it with 1 million entries. We again use the Yandex Deep 1B ($d = 96$) dataset as a basis here and we start with the first one million entries, i.e., the state of the collection at the beginning is identical for all runs. After data loading has completed, we generate and build an index – either a VAF (35 marks per vector component) or a PQ index (8 subspaces, 4096 centroids). Once the entity and the indexes have been prepared, the actual measurements start. We run two threads is parallel:

The first thread runs a loop that either performs an insert or a delete operation per cycle. For an insert, we randomly draw between 100 and 7500 entries from the Yandex Deep 1B dataset. For the delete operation, we randomly select between 10 and 500 IDs to be deleted. Both the insert and the delete takes place in one transaction that executes a single, batched operation. Some of the inserted vectors are stored in a queue for later use. Between a cycle there is a pause of between 10 ms and 1000 ms.

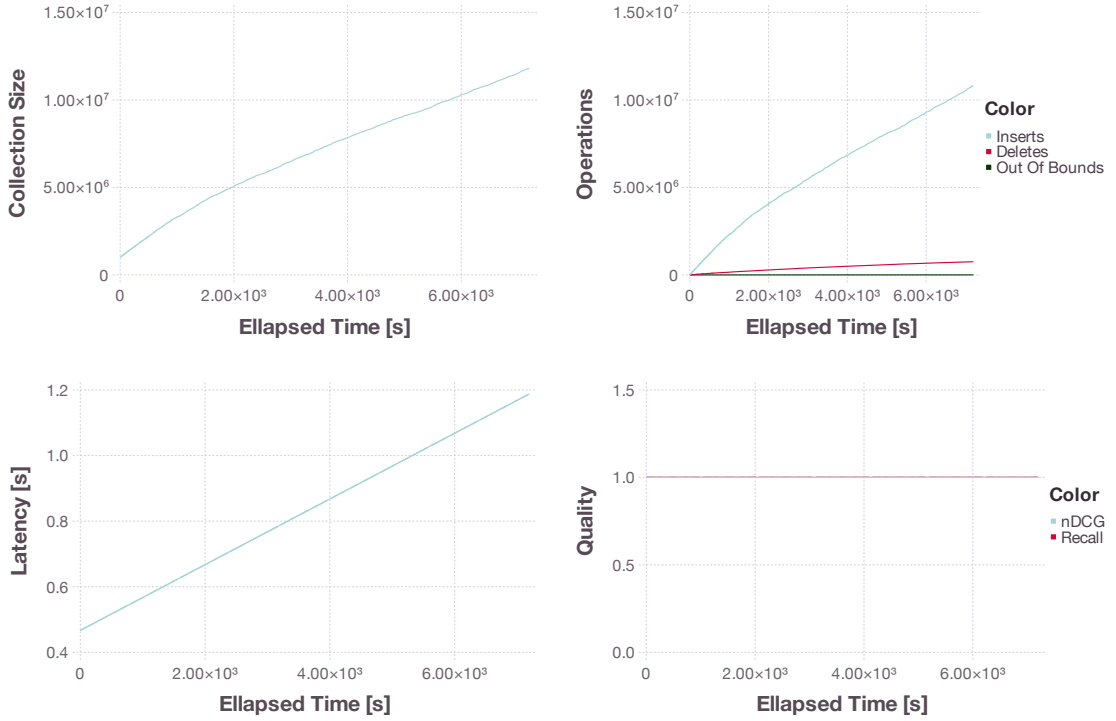
The second thread starts with a 3 s delay after the first and then continuously executes NNS queries using the query vectors that have been inserted by the first thread. One NNS query explicitly uses the previously created index whereas the other query employs a brute-force scan to obtain the groundtruth. As in previous experiments, we use query hints to nudge Cottontail DB into either direction. To make sure that the two queries operate on the same state of the database, they are executed in a single transaction [10]. In addition to the queries themselves, we also obtain the query plan generated by Cottontail DB to make sure that the index is actually used.

We run the entire experiment over the course of 2 h doing two runs. In the first run, we trigger concurrent index rebuilding at half-time, using the asynchronous rebuild mechanism described in Section 6.3. In the second run, we let the workload complete without any index rebuilding. Over the entire time span, we observe how the collection itself, the latency of the NNS queries and the quality of the results develop.

[10] If we were to execute the queries without an explicit transaction, we would expect differences due to the changes that may interleave with the execution of the two queries.

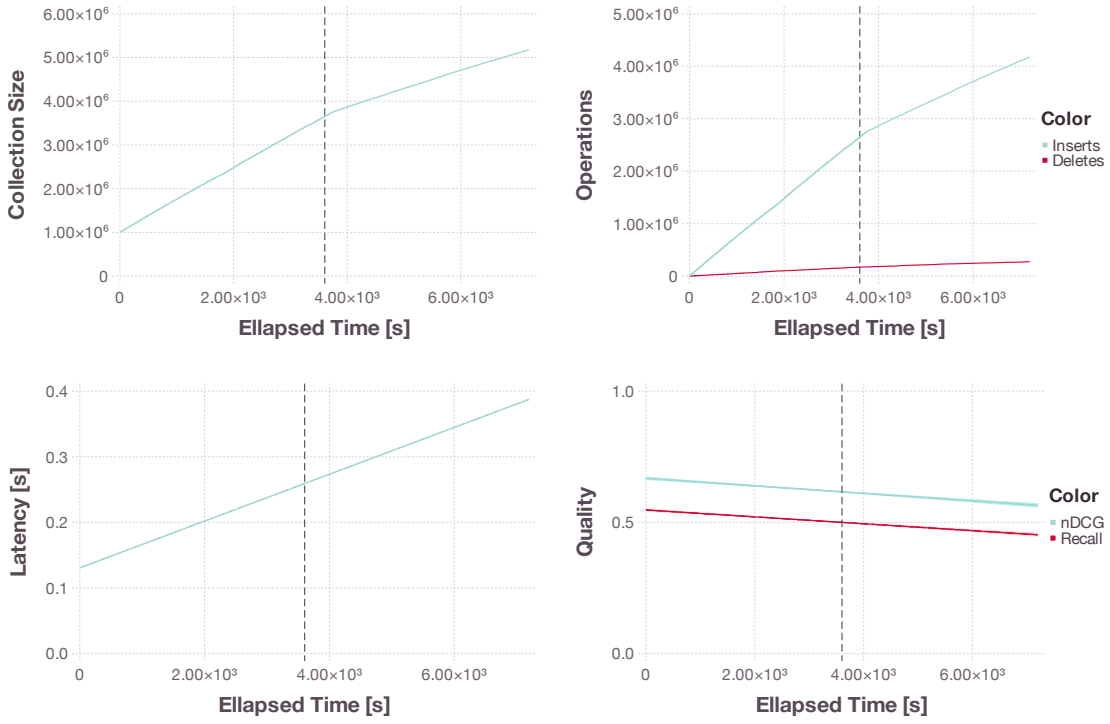



((a)) **Index adaptiveness measurements for VAF index. Start of index rebuilding is indicated by dashed line.**

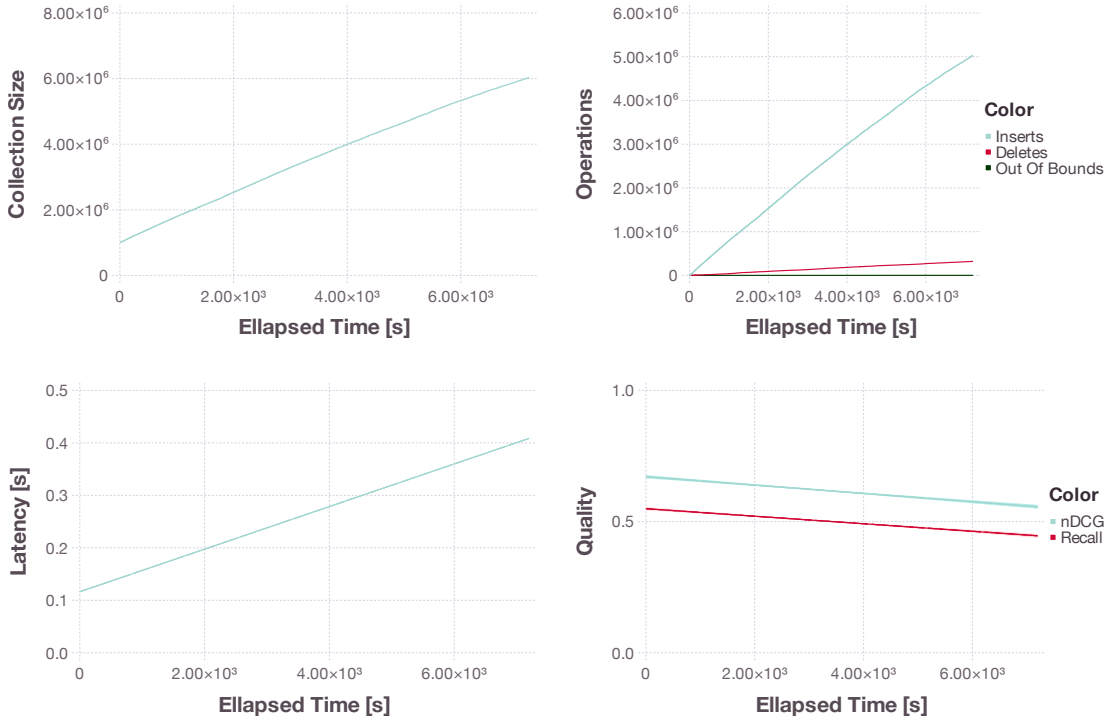



((b)) **Index adaptiveness measurements for VAF index (without rebuild)**

**Figure 8.11 Index adaptiveness measurements for VAF index showing the size of the collection (top-left), number of accumulated insert- and delete operations (top-right), latency for NNS using the index (bottom-left) and quality of NNS w.r.t. to groundtruth (bottom-right) over time (x-axis).**




((a)) **Index adaptiveness measurements for PQ index. Start of index rebuilding is indicated by dashed line.**




((b)) **Index adaptiveness measurements for PQ index (without rebuild)**

**Figure 8.12 Index adaptiveness measurements for PQ index showing the size of the collection (top-left), number of accumulated insert- and delete operations (top-right), latency for NNS using the index (bottom-left) and quality of NNS w.r.t. to groundtruth (bottom-right) over time (x-axis).**

The outcome of the experiment are depicted in Figure 8.11(a), Figure 8.11(b) (VAF) and Figure 8.12(a), Figure 8.12(b) (PQ). From the plots we can make several observations: Within the considered time window of 2 h, approximately 12.5 million inserts could be executed for the VAF index whereas only 4 million entries were inserted for PQ. This leads to approximately 1750 vs. 550 inserts per second. While this is not the maximum throughput possible, since the workloads are characterised by a random but fixed number of inserts per cycle as well as random pauses in between, we can still clearly see that given these identical settings, more than twice as many inserts could be executed for VAF. This can be directly attributed to the computational complexity of deriving the PQ signature, which is more expensive than for VAF. Consequently, VAF might be the better choice for insert- or update heavy workloads.

Latency for executing NNS grows for both the VAF and the PQ index in an approximately linear fashion. This is to be expected, since both indexes perform linear scans of the respective data collection. In this setting, we cannot observe any visible deterioration in terms of latency building up over time other than the linear increase. Also both recall and nDCG remain constant for the VAF index over the course of benchmark despite the concurrent inserts, which is the desired and thus expected behaviour. For PQ, however, there seems to be a clear downward trend as the collection grows in size.

One hour into the second run, we trigger the index rebuild (dashed line in Figures 8.11(a) and 8.12(a)). Also, if we compare quality and latency for the runs with and without an index rebuild, we do not see a noticeable difference in the time span between the rebuild and the end of the experiment. Regardless of whether the index is rebuilt from skratch or changes incrementally, the indexing performance remains the same both in terms of latency and quality. The same can be said for PQ, however, there is a noticable impact on the throughput which goes done once index rebuilding starts, as one can see in Figure 8.12(a).

In order to test our method with data that does not exhibit this kind of well-behavedness, we ran a second series of measurement in which we introduced random noise. This *jitter* is simply a random number between zero and the component-wise mean times a configurable magnitude $M$ that is either added or subtracted from each vector component. The actual vectors are again drawn from the Yandex Deep 1B data collection. Therefore, we introduce an increasing variance to each dimension while maintaining the original distribution.

## 8.5 Summary and Discussion

With the evaluation we have demonstrated the applicability of the concepts introduced in Chapter 6 to practical scenarios using our reference implementation Cottontail DB. Even though Cottontail DB was unable to compete with Milvus in terms of raw processing speed, we consider this evaluation to be a success. Maintaining and accessing all the relevant data on disk incurs an overhead that makes it difficult or even impossible to reach the processing speed of pure in-memory systems like Milvus. This was pointed out my L. Amsaleg, who stated that "When purposely dealing with secondary storage, it is hard if not impossible to beat high-dimensional indexing solutions that have been designed to run in main memory. This is unfortunately true even for collections that might fit in main memory, as enforcing durability etc. generates a substantial overhead." ([Ams14], p. 12). Regardless of these limitations, Cottontail DB still seems to perform pretty well against Milvus when loading data from disk is considered as well. In our experiments, during the time that Milvus required to load a collection, Cottontail DB could query that same collection between 15 (VAF) to 34 (PQ) times, all the while retaining the flexibility to switch between collections and/or execute different types of workloads that Milvus could not (e.g., range search). Our experiments have also demonstrated the limitations of pure in-memory processing in cases where collections simply do not fit into main memory, regardless of how optimised the execution engine may be.

### 8.5.1 Generalised Proximity Based Queries

The notion of generalised, proximity based operations as an extension to the relation algebra described in Section 6.1 provide us with the ability to construct and execute a wide range of query workloads in a much more flexible way then purpose built systems such as Milvus [WYG+21] can provide. Furthermore, the proposed algebra enables the DBMS to decompose and optimise queries based on the needs expressed by a system user and the parameters of the system itself.

In the past decades, many information retrieval models have been proposed [Uma83; SFW83; WZW85]. Some of these models are set-based [SFW83; Uma83], and thus similar to the relational model [Cod70], while others are based entirely on the vector space model that considers documents to be high-dimensional vectors [WZW85]. Our approach uses the relational model as a foundation and integrates the notion of complex data types (e.g., vectors), functions that operate on these types as well as scoring and ranking as explicit extensions in way that en-

ables its use beyond mere multimedia retrieval. In that it follows an established tradition of extending the expressive power of SQL and the relational model [Lib03], that has been practiced for many years [LCI+05; ZHC+06; BOV07], as we have argued in Section 3.2.3. Conceptually, we see the proposed extensions as a first, minimally invasive step towards the realisation of the *Extended Boolean Retrieval Model* as proposed in [SFW83]. In such a model, distances and scores can be employed to perform advanced operations such as fuzzy-set operations or similarity joins [Uma83; Zad96; BMS+01]. From our perspective and with an eye on analytics workloads, we would also like to see query primitives such a clustering or RNNS [KM00] in that list of supported operations.

The convergence of information retrieval and databases as well as the addition of multimedia support have been a reocurring subject at many of the database self-assessment meetings since the 1980s [AAB+08]. Unsurprisingly, many attempts at convergence have been taken over the years at a conceptual [MS96; AN97; WOK+98], logical [ZHC+06; BOV07] and physical [SAA+10; GAKS14; WLL+15; GS16; YLF+20] level. We follow that trend and the model we propose provides a sound, theoretical foundation rooted in relational algebra and has been demonstrated to be useful in practical scenarios.

An important, mental step that enables such simple yet expressive models is that of "separation of concerns", or what [Gia18; Fer14] refer to as "pragmatics", which is in fact realised throughout the *vitrivr* stack [RGT+16; GRS19a]. While similarity search and retrieval may be the end goal, from a database perspective, we merely execute simple primitives such as the evaluation of distance functions, sorting and selection. By carefully ignoring the noise introduced by higher-level concepts such as documents, features, scores and distances – which may not be useful for data processing – the database system can focus on its core purpose which is that of providing reliable and fast data management and access and leave the interpretation and advanced data modelling to upper-tier systems. This is a common approach in database research and with our model, we further move in such a direction, similar to computational databases such as SciDB [SBZ+13], MonetDB [IGN+12] or HypeR [HPS+17], which focus on providing support for efficient computation close to where the data resides, without limiting the concrete application too strictly. Such systems form an active research field [LGG+18; BGL+22] and might hold interesting benefits for the multimedia community. They constitute a stark contrast to systems with very focused applications such as Milvus [WYG+21], Vald[11] or Qdrant[12].

[11] See https://github.com/vdaas/vald/, Accessed August 2022
[12] See https://github.com/qdrant/qdrant/, Accessed July 2022

Since ADAM$_{pro}$[GS16] is Cottontail DB's predecessor and one of the inspirations for the work presented in this Thesis, we would also like to highlight the differences between its query model and our own. As outlined in Section 5.5, [Gia18] proposes a new similarity operator $\tau_{\delta(\cdot,\cdot),a,q}(\mathcal{R})$ that *"performs a similarity query under a distance $\delta(\cdot,\cdot)$ applied on an attribute $a$ of relation $\mathcal{R}$ and compared to a query vector $q$."* [Gia18] (p. 138). It then uses this operator to derive more restricted versions that basically consider NNS and range search ($\epsilon$NN) to be dedicated query primitives. As we have argued in Section 6.1, this model is very restrictive in its assumptions and to some extend violates the aspect of pragmatics as advertised by [Fer14].

In contrast, our model considers the distance computation to be the result of an arbitrary function invocation. We restrict ourselves to certain distance functions in practice but in theory, any function could be employed. Furthermore, we use the extended projection to make the derived distance explicit and we argue, that making the attribute explicit is what allows downstream processing such as normalisation (from distance to score), combination of multiple scores to express score fusion or variants thereof [BMS+01]. Ultimately, we consider it to be at the discretion of the user to decide whether and how calculated attributes should be used and included and it should be the task of the query optimiser to take these decisions into account when preparing an efficient execution plan.

Consequently, the relational algebra extensions we propose are rather limited and lie well within the capabilities most modern DBMS bring today. In line with [AAA+14; AAA+20], we identify the efficient execution of the functions and computations involved using the available means in terms of hard- and software (e.g., SIMD, GPU), as important, future research directions. In order to enable fast execution, functions must be enriched with metainformation that allow for optimisation at different stages of the query planning and execution process. In addition, we also believe the relationship between functions and indexing techniques to be a promising avenue to explore.

Nevertheless, other systems, such as Milvus [WYG+21], take a similar approach to [GS16] as they add query primitives for the different types of workloads they accomodate - leading to growing number of disjoint APIs for different purposes. We do not consider this a sustainable approach, since it ignores future workloads that may combine existing primitives in a different way (e.g., aggregating on distance or FNS). Furthermore, generating specialised blackbox operators for different types of high-level primitives unnecessarily limits the potential for optimisation by the DBMS and increases the complexity of the systems.

### 8.5.2 Query Planning and Cost Model

In Section 6.2 we have proposed a cost model that takes the expected quality of results into account and we have discussed methods that can be used to estimate that quality for different, known query plans. Our reference implementation Cottontail DB uses a query planner that leverages the aforementioned cost-model to determine the optimal execution path given a query and a cost policy specified by the user, as demonstrated in Section 8.2.2.

To the best of our knowledge, both aspects – the estimation of quality as well as its use in a cost model for query planning – are rather uncommon in multimedia search systems today. Even though the quality vs. execution time trade-off is an implicit consequence of approximate solutions to proximity-based search in general and NNS in particular, there is very little research that considers this issue systematically, with a few, notable exceptions [WB00; BHC+01; SJH+05]. This is reflected by the fact, that most state-of-the-art approximate index structures fail to provide any strong quality guarantees [EZP21]. Here we see a huge corpus for future research: On the one hand, we see a need for index structures that provide explicit quality guarantees such as [LWW+20]. On other the hand, we should continue to explore system-level mechanisms that continuously evaluate the efficiency and effectivenes by which results have been produced and adjust the cost model accordingly. Such an empirical approach to query planning has already been proposed by [Gia18], however, only taking execution time into account. This goes into the direction of adaptive query processing [DIR+07] and the cost-model we propose can be seen as a foundation for a continuous control loop between the query execution engine and the query planner.

### 8.5.3 High-Dimensional Index Maintenance

Maintaining a durable and consistent state and providing transactional guarantees for data structures required to execute queries – including secondary index structures – is a basic trait shared by most if not all traditional SQL and NoSQL systems. For some reason, however, durability and the notion of isolation and consistency in the face of dynamic data are aspects often overlooked for high-dimensional indexing [Ams14; HJB19], especially in combination [Ams14]. In Section 6.3, we have proposed a system-level mechanism for maintaining high-dimensional index structures on disk and for processing incremental changes to the data in a transactional manner. The viability of the proposed mechanism was demonstrated in Section 8.4, based on two concrete examples.

Despite an already huge corpus of reasearch that dates back to the 70s [Ben75; Gut84; BKS+90; IM98; WSB98; JDS10], high-dimensional indexing for NNS is still a very active research topic today [SOV+21; KBC+18]. Since different index structures provide different properties in terms of scalability, performance and quality (guarantees) and can potentially be employed for different tasks, we regard this work as pivotal for further extending the indexing capabilities of a multimedia DBMS such as Cottontail DB. One could argue that having a single index may be sufficient, however, we believe that in light of a DBMS's ability to plan queries and the different properties these indexes bring, having a choice is better. Furthermore, multiple indexes could also be combined to improve the quality of results and the speed of execution [Gia18].

In light of ever growing data collections, the topic of disk-based ANNS has been getting more attention lately, with a few, early examples [LJA11; LÁJ+09; GJA10] and new, emerging techniques such as DiskANN [JSDS+19] or SPANN [CZW+21], which both advertise a hybrid in-memory/on-disk processing model and report impressive numbers for both latency and recall. However, in contrast to [LJA11; LÁJ+09], both these state-of-the-art algorithms do not consider dynamic data at all and operate on the premise that the complete data collection can be (re-)indexed. This severely limits the usefulness of such approaches, since dynamic data is a fact that must be accepted. Nevertheless, such efforts lay important groundwork for disk-based multimedia index maintenance and must simply be extended in terms of their ability to process incremental changes.

In [LJA11; LÁJ+09], the authors propose the NV-tree, which is a disk-resident, high-dimensional index structure that provides transactional support [LÁJ+18] and thus supports inserts, updates and deletes. The NV-tree would make a great extension to Cottontail DB and we expect it to fully fit into the proposed model for index maintainence in that it is an index structure that does not fail per-se and should not exhibit deterioration due to dynamic data (i.e., its a corner case, like the $B^+$-tree). However, unfortunately the index structure itself seems proprietary and the available literature does not provide us with enough information to come up with out own implementation.

The subject of online indexing (i.e., processing incremental changes) has also been addressed by a few authors, for indexes such as PQ [XTZ18], LSH [CS15] and some graph-based methods [ZWN22]. While the proposed methods lack a notion of index state and transactionality and do not consider indexes to be durable data structures, these are still very important contributions towards a robust write-model for different types of indexing techniques.

We have also dicussed different vector database systems in Section 5.3, which all somehow face the issue of dynamic data. Milvus [WYG+21], for example, handles inserts by marking them with timestamps and it allows for configurable consistency levels that differ in the timestamps of entries that are being considered. Hence, it uses a kind of MVCC-based consistency and concurrency control model, similarly to Cottontail DB. Deletes are enabled by setting bit mask for deleted entries and filtering them on the fly at query time. However, for both mechanisms, it is not fully clear how they scale as changes accumulate. Nevertheless, some concepts employed by Milvus may be very interesting, e.g., the (re-)indexing of chunks rather than the entire data collection at once. Vald advertises a "asynchronous auto indexing" that uses "distributed index graphs" to avoid locking the entire graph during indexing. However, it is not clear what type of guarantees are provided by this mechanism other than preventing a system-wide halt. Unfortunately, for Qdrant , we could not find any information as to how dynamic data and concurrency is handled.

*Don't adventures ever have an end? I suppose not. Someone else always has to carry on the story.*

— J.R.R. Tolkien, 1954

# 9

# Conclusion & Outlook

Inspired by the questions raised in [JWZ+16] and the solutions presented by [Gia18], we set out to challenge some assumptions about the storage, management and processing of data used in multimedia retrieval and analytics workloads. Most importantly, we called into question that (i) multimedia retrieval and analytics only involves NNS in high-dimensional, real-valued vector spaces, (ii) that data collections remain static while they are being queried, and (iii) that the execution path of a query should be an explicit choice of the user. Throughout this Thesis, we tried to take the viewpoint of a database engineer who considers the problems at hand to be data management and processing issues that should be delegated to the underlying DBMS whenever possible [Fer14; Ams14]. At the (preliminary) end of our journey, this has led to several contributions:

1. We specified a query model based on the notion of generalised, proximity based operations to support multimedia retrieval and analytics workloads on top of relational algebra (see Section 6.1). In our model, we put a special emphasis on the role of functions and Distance Function Classes in query formulation, planning and execution and we explore the relationship between these functions and high-dimensional indexes.

2. We proposed a cost-model that takes the expected quality of results produced by approximate index structures into account (see Section 6.2). This cost model exposes the often implicit and opaque trade-off between accuracy and speed to the system and the user, who can act upon it.

3. We described a mechanism that can function as a template for integrating and maintaining high-dimensional indexes in a transactional and durable database system (see Section 6.3). Such an integration is crucial for supporting workloads that rely on changing data.

4. We introduced our open-source reference implementation Cottontail DB, which employs the aforenamed, theoretical contributions (and some more, see Chapter 7).

5. We presented an evalution that uses Cottontail DB to demonstrate the viability of our approach (see Chapter 8). One focus of this evaluation was a direct comparison of Cottontail DB to the vector database Milvus [WYG+21].

6. We identified important issues and research questions to further the fields of multimedia retrieval, analytics and database research as well as the convergence of the two.

Cottontail DB can be considered a classical database in many respects in that it offers traditional database features such as durability and transactionality. Yet its main purpose, until today, has been the processing of interactive multimedia retrieval workloads. Cottontail DB has been part of the *vitrivr* ecosystem [RGT+16; GRS19a] for several years now and its versatile query model has enabled successful participation of *vitrivr* in many iterations of events like the VBS [RAPG+19; SAPG+20; SGH+21a; HAG+22] or LSC [SGH+21b; HAPG+20; HGP+21; SGH+22]. In addition, Cottontail DB powered rather experimental systems such as LifeGraph [RBG+21], which speaks to its versatility.

The work on Cottontail DB and thus the work presented in this Thesis contributes towards integrating aspects of multimedia retrieval & analytics on the one hand and database research on the other. In its current form, it constitutes an important foundation for current and future research endeavours in various multimedia applications. In addition, we also see a huge potential for using both *vitrivr* and Cottontail DB for practical applications and for educational purposes.

## 9.1 Future Work

The process of writing this Thesis, the formulation of the theoretical models and the implementation of system implementations it comprises of, has answered several important questions but it has raised even more. Over the course of this project, we came-up with many (brave) new ideas on both the model and the implementation side of things. We will conclude this chapter and this Thesis, for that matter, by ellborating on these ideas.

### 9.1.1 Query Model

Decades of effort have been invested into the development of different query models for information retrieval and their combination with the Boolean model. However, only few have actually been implemented and put to the test. We consider the query model we have presented to be a first step in combining two worlds in a pragmatic way that opens new avenues to explore. On the one hand, there are many potentially relevant extensions that lead towards a full-fledged Extended Boolean Retrieval Model [SFW83] such as fuzzy queries [Uma83; BMS+01] or similarity joins [YLK10]. One step in realising this could be to consider distances and scores explicit data types that, in addition to basic arithmetics, allow for advanced operations required in this context. On the other hand, a general purpose multimedia DBMS must also consider advanced query primitives, such as RNNS [KM00], bichromatic RNNS [SRA+01] or clustering. Both could prove to be interesting extensions to the proposed model.

In light of such potential extensions we would like to briefly touch upon the question *"Is a novel multimedia query language needed for the database to fully support multimedia analytics, or is an extension of classic query languages sufficient?"* [JWZ+16] (p. 299, RQ2). As far as the work in this Thesis goes, we can state that all of the functionality required can be expressed by an extended relational algebra and therefore be easily translated into SQL. The idea of expressing advanced functionality as mere function invocations can be further extended to other concepts such as fulltext search, as, e.g., done by MS-SQL. However, advanced query paradigms such as clustering and RNNS are difficult or impossible to express in standard SQL and may therefore require language extensions similar to, e.g., common table expressions for recursive queries. While some authors may advertise a dedicated "information retrieval query language" [Fer14], we tend to argue from a personal experience, that extension of SQL would be preferable to coming up with a new language, simply because SQL is well established, well known and easy to use [1].

### 9.1.2 Multimedia Indexing

We have shown in our evaluation that the ability to cope with changes to the data at an index level may not be sufficient to maintain a constant quality of service, at least for the index structures considered in our experiments. However, we strongly believe that the argument can be extended to other indexes as well.

[1] As a matter of fact, a SQL interface is among the top feature requests for Cottontail DB.

We see a major challenge in the tuning of hyperparameters to the concrete collections at hand. As it seems, the factors that determine the right choice are not just tied to the dimensionality of the vectors but also the size of the collection and the distribution of data within it. Up and until now, this problem has not been systematically considered and finding the set of parameters that yields the best results in terms of performance and quality has been a manual task that relies on experience of the user and empirical analysis. This may work well if a collection remains static but fails once a collection is subject to accumulating changes that cannot be dealt with, given the set of hyperparameters. Hence, rather than regarding these parameters as fixed properties, a multimedia DBMS should move towards dynamically adapting them to its needs. In order for this to work, however, we require a better understanding of the relationship between those parameters, the properties of a collection and the quality of the results.

In addition, the traditional model applied to indexing is that of a system user or database administrator creating indexes based on their requirements. Due to the complex interdependencies we have observed, this might not work well for high-dimensional indexes which is why we also see multimedia indexing as a very interesting area for applying Database Cracking techniques [IKM07; SJD13] wherein the DBMS tries to learn indexes and optimised storage views based on query workloads. To some extent, our adapting the cost model and state of existing indexes based on the outcome of queries can be regarded as first baby-steps in such a direction.

### 9.1.3 Storage Model

Over the course of Cottontail DB's lifecycle, we have gone through three different storage engine versions (four, if one counts the experimental ones). One takeaway message from these projects are that the organisation and layout of data on disk has a great impact on performance and that storage of and access to large, complex data types comes with unique requirements. At a high level, there two ways to approach this issue, both equally interesting and challenging:

On the one hand, [Gia18] did propose and implement a polystore approach, which delegates storage to different systems that handle a certain type of data- or query-workload particularily well. Such an approach comes with the challenge that transactional guarantees must be provided accross different systems. This is a problem tackled by PolyDBMS systems such as Polypheny-DB [VSS18; VHS+20]. In fact, first steps in such a direction have already been taken with the integration of Cottontail DB into Polypheny-DB and they and could give rise to

interesting new insights for mixed query workloads in the *vitrivr* context. However, the disadvantage of such an approach is the inherent complexity incurred by relying on different systems, which may complicate the configuration and parametrisation of the individual components for a specific usecase.

On the other hand, it would also be worthwile to come-up with efficient low-level storage solutions for the different types of data and workloads faced by a multimedia DBMS as this would allow for tighter integration and optimisation of different parts of a system and reduce dependency on external components. The recent years have seen many interesting strategies to database storage that could be relevant for our purpose (e.g., [IGN+12; SR12]). One concrete step we would like to take is serialisation and storage of multiple values in a single chunk or segment. We know from experiments that storing and compressing multiple vectors together can lead to significantly better compression and thus reduced IO for read workloads. Such a storage scheme could also be beneficial for a batched execution model. Furthermore, several, specialised storage engines could and should be integrated by an abstraction layer as, e.g., as proposed by [DJ11; AIA14] and (to some extent) implemented by Cottontail DB, if only to be able to compare them for our purposes.

### 9.1.4 Cost Model

By making the cost of impaired quality explicit, we have opened an interesting field to explore. Thus far, we have only considered rather simple cases in which an approximate index structure is used to satisfy a NNS or range search query. In addition to making these estimates more accurate and exploring the impact of approximation on different types of primitive constructs, it is also very interesting see how errors introduced at some point in a complex execution plan propagate and influence a final result. Let us, for example, consider the case of a similarity-join, wherein both strands of execution are generated by an approximate method. How are the errors combined in resulting dataset? Is this something that can be predicted assuming an accurate mode of the initial error?

### 9.1.5 Execution Model

In its early days, Cottontail DB implemented a materialization model wherein each operator generated intermediate result sets in memory. This strategy was quickly dropped in favour of the much simpler and less memory-intensive iterator model. However, due to the great potential we see in vectorisation and

SIMD, especially with eyes to the recent launch of the Java Vector API[2], this model is no longer an optimal fit and a transition to a batch-model should be realised rather sooner than later. Such a transition must of course be accompanied and aligned with changes to the storage model, which should also be batched.

On the implementation side we see some room for work on the aspect of automated code generation. In order to attain the execution speed we need, a lot of logic is re-implemented over and over again to make use of optimisations for specific data types and to avoid type casting in tight loops. This process could and should be automated either at compile or runtime (code generation).

Another interesting aspect that falls in of execution is that of distributing workloads. $\text{ADAM}_{\text{pro}}$ used to be inherently distributed [GS16; Gia18] whereas Cottontail DB is inherently local. Adding the ability to distribute workloads among differet nodes might go a long way in achieving better execution performance, especially for analytical workloads. However, such a step will invitabely raise new research and engineering questions with respect to partitioning and distribution of the data, indexing and transactionality through, e.g., distributed snapshot isolation [BHF+14]. Generally speaking, though, we see it as a worthwile endevour to rethink Cottontail DB data partitioning model from storage to processing with regards to both local parallelisation as well as distribution.

[2] See https://openjdk.org/jeps/338 and JEPs 414 & 417, Accessed August 2022

# Appendix

# A

# Evaluation Supplementals

## Cottontail DB Entities & Configuration

### Configuration Parameters

```
{
    /* Path to data folder. */
    "root":"/mnt/raid0/evaluation/cottontail-data",
    "server": {
      /* Port used by gRPC query interface. */
      "port": 1865
    },
    "cost": {
      /* Allows for aggressive parallelisation. */
      "speedupPerWorker": 0.00,

      /* Portion of IO costconsidered parallelisable. */
      "parallelisableIO": 0.9
    },
    "execution": {
      /* Activate SIMD support through Vector API. */
      "simd": true
    }
}
```

Listing A.1: Cottontail DB configuration used during the evaluation (config.json).

## Entities

**Table A.1 State of the entity yandex_deep5m used during the evaluation.**

| DBO | Class | Type | Rows | Size | Info |
|---|---|---|---|---|---|
| yandex_deep5m | ENTITY | - | $5 \times 10^6$ | - | - |
| yandex_deep5m.id | COLUMN | INTEGER | $5 \times 10^6$ | 1 | NOT NULL |
| yandex_deep5m.feature | COLUMN | FLOATVEC | $5 \times 10^6$ | 96 | NOT NULL |
| yandex_deep5m.category | COLUMN | INTEGER | $5 \times 10^6$ | 1 | NOT NULL |
| yandex_deep5m.idx_feature_vaf | INDEX | VAF | $5 \times 10^6$ | - | CLEAN |
| yandex_deep5m.idx_category_btree | INDEX | BTREE | $5 \times 10^6$ | - | CLEAN |
| yandex_deep5m.idx_feature_pq | INDEX | PQ | $5 \times 10^6$ | - | CLEAN |
| yandex_deep5m.idx_feature_ivfpq | INDEX | IVFPQ | $5 \times 10^6$ | - | CLEAN |

**Table A.2 State of the entity yandex_deep10m used during the evaluation.**

| DBO | Class | Type | Rows | Size | Info |
|---|---|---|---|---|---|
| yandex_deep10m | ENTITY | - | $1 \times 10^7$ | - | - |
| yandex_deep10m.id | COLUMN | INTEGER | $1 \times 10^7$ | 1 | NOT NULL |
| yandex_deep10m.feature | COLUMN | FLOATVEC | $1 \times 10^7$ | 96 | NOT NULL |
| yandex_deep10m.category | COLUMN | INTEGER | $1 \times 10^7$ | 1 | NOT NULL |
| yandex_deep10m.idx_feature_vaf | INDEX | VAF | $1 \times 10^7$ | - | CLEAN |
| yandex_deep10m.idx_category_btree | INDEX | BTREE | $1 \times 10^7$ | - | CLEAN |
| yandex_deep10m.idx_feature_pq | INDEX | PQ | $1 \times 10^7$ | - | CLEAN |
| yandex_deep10m.idx_feature_ivfpq | INDEX | IVFPQ | $1 \times 10^7$ | - | CLEAN |

**Table A.3 State of the entity yandex_deep100m used during the evaluation.**

| DBO | Class | Type | Rows | Size | Info |
|---|---|---|---|---|---|
| yandex_deep100m | ENTITY | - | $1 \times 10^8$ | - | - |
| yandex_deep100m.id | COLUMN | INTEGER | $1 \times 10^8$ | 1 | NOT NULL |
| yandex_deep100m.feature | COLUMN | FLOATVEC | $1 \times 10^8$ | 96 | NOT NULL |
| yandex_deep100m.category | COLUMN | INTEGER | $1 \times 10^8$ | 1 | NOT NULL |
| yandex_deep100m.idx_feature_vaf | INDEX | VAF | $1 \times 10^8$ | - | CLEAN |
| yandex_deep100m.idx_category_btree | INDEX | BTREE | $1 \times 10^8$ | - | CLEAN |
| yandex_deep100m.idx_feature_pq | INDEX | PQ | $1 \times 10^8$ | - | CLEAN |
| yandex_deep100m.idx_feature_ivfpq | INDEX | IVFPQ | $1 \times 10^8$ | - | CLEAN |

**Table A.4 State of the entity yandex_deep1b used during the evaluation.**

| DBO | Class | Type | Rows | Size | Info |
|---|---|---|---|---|---|
| yandex_deep1b | ENTITY | - | $1 \times 10^9$ | - | - |
| yandex_deep1b.id | COLUMN | INTEGER | $1 \times 10^9$ | 1 | NOT NULL |
| yandex_deep1b.feature | COLUMN | FLOATVEC | $1 \times 10^9$ | 96 | NOT NULL |
| yandex_deep1b.category | COLUMN | INTEGER | $1 \times 10^9$ | 1 | NOT NULL |
| yandex_deep1b.idx_feature_vaf | INDEX | VAF | $1 \times 10^9$ | - | CLEAN |
| yandex_deep1b.idx_feature_pq | INDEX | PQ | $1 \times 10^9$ | - | CLEAN |
| yandex_deep1b.idx_feature_ivfpq | INDEX | IVFPQ | $1 \times 10^9$ | - | CLEAN |

# B

# Additional Results

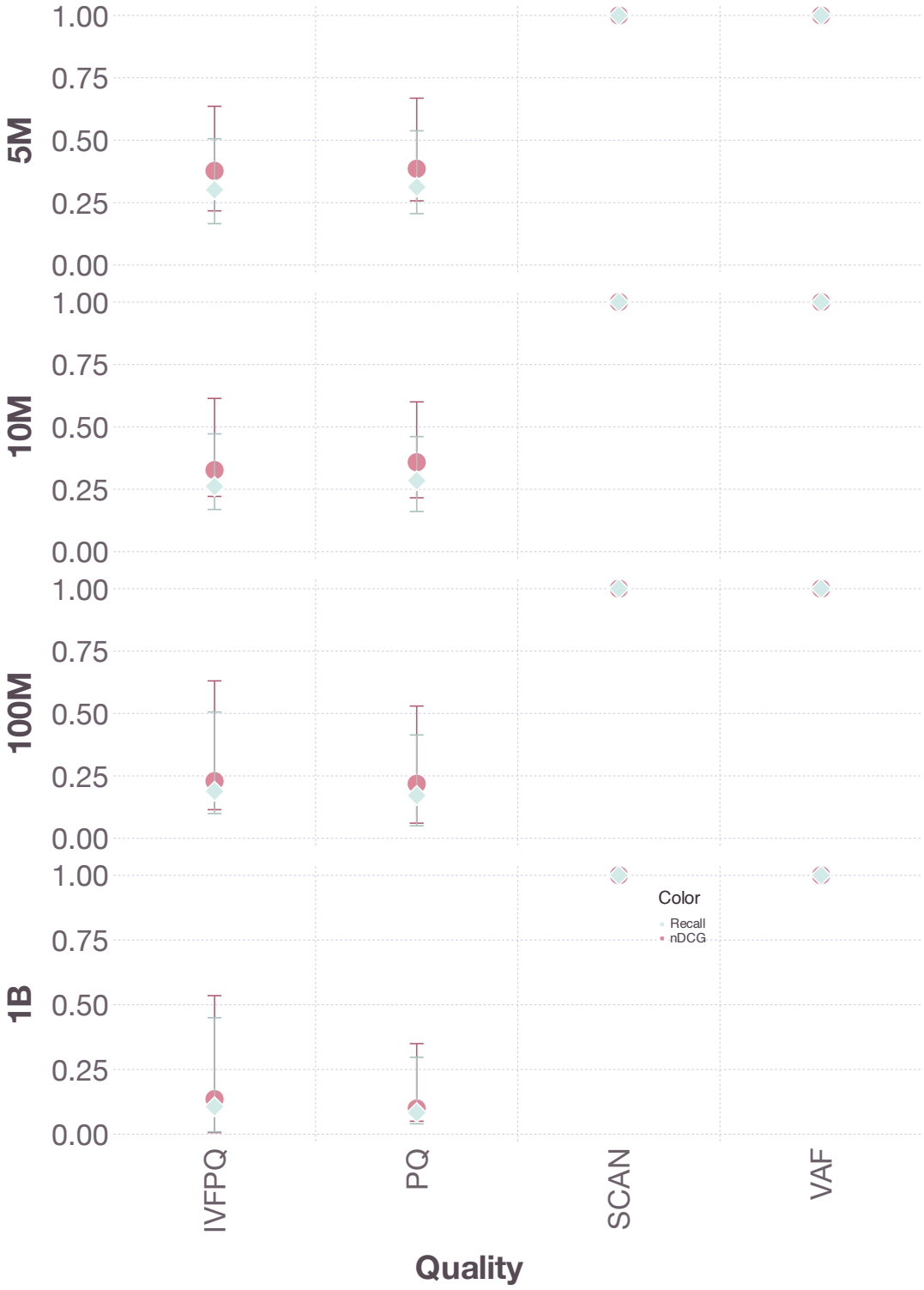



**Figure B.1** **Quality metrics for Cottontail DB during the simple NNS large-scale retrieval evaluation presented in Section 8.3.1. Recall and DCG are 1.0 for all execution strategies other than PQ, for which they exhibit rather low values with a large spread.**

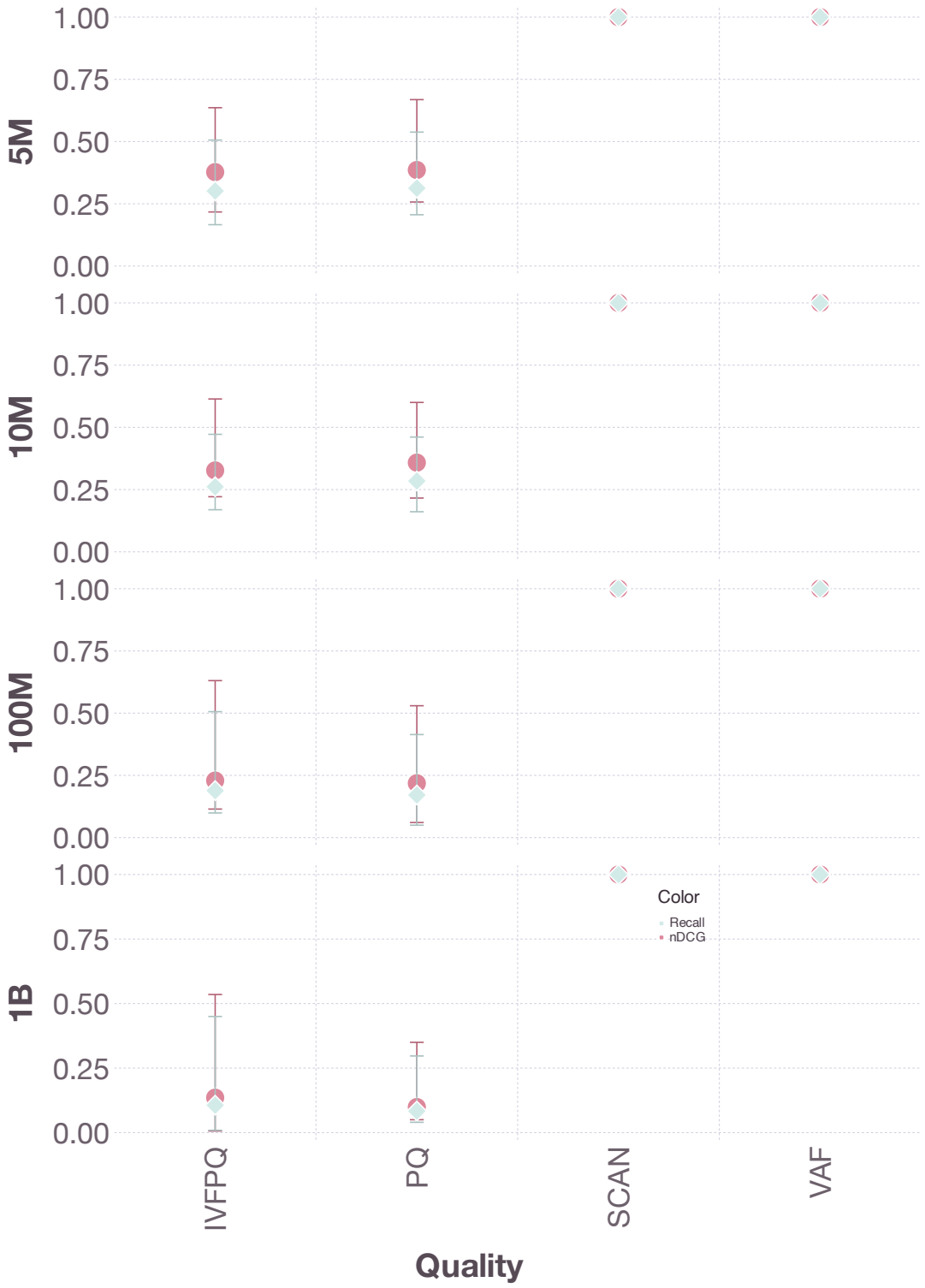



**Figure B.2** **Quality metrics for Cottontail DB during the NNS + fetch large-scale retrieval evaluation presented in Section 8.3.1. Recall and DCG are 1.0 for all execution strategies other than PQ, for which they exhibit rather low values with a large spread.**

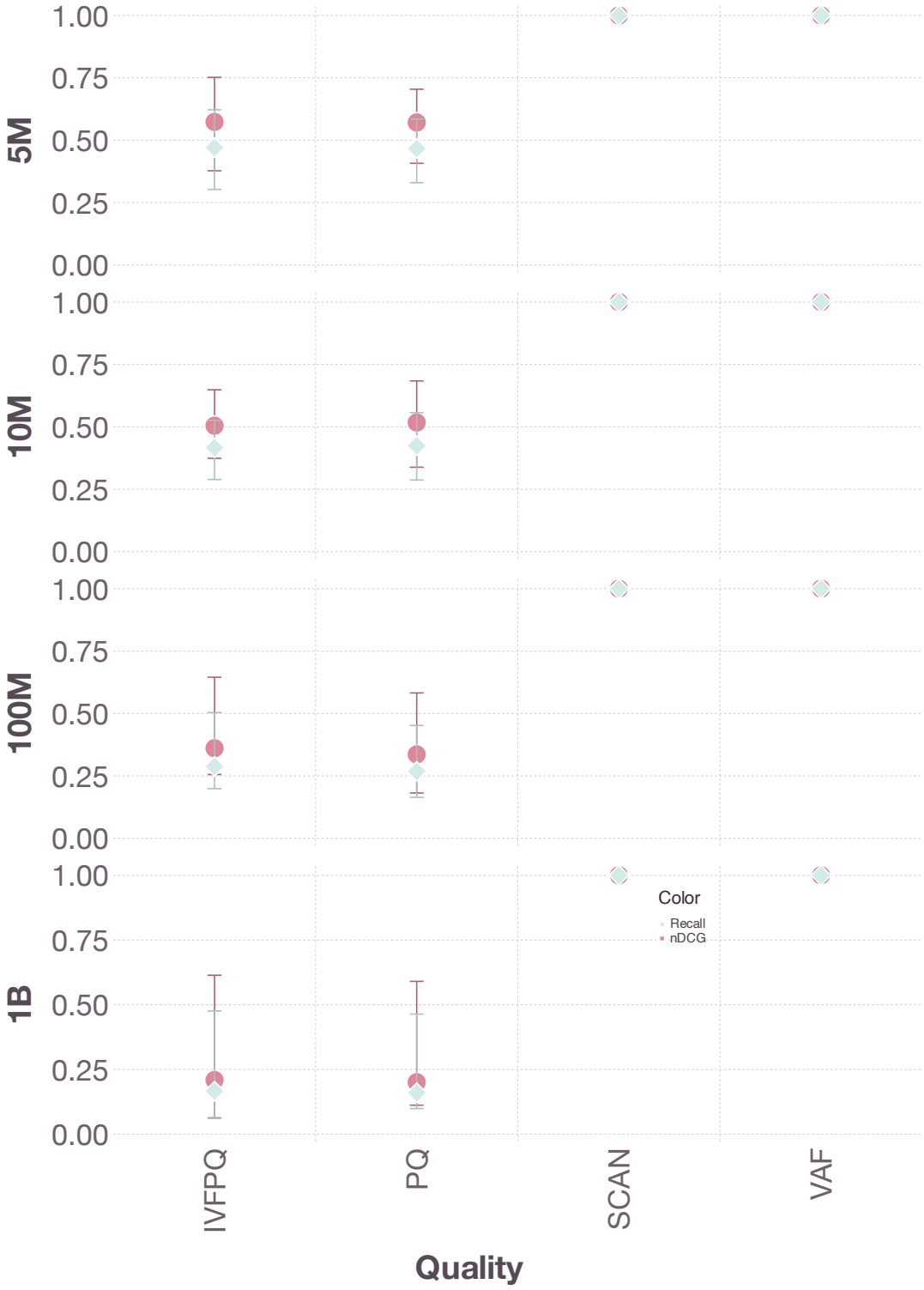



**Figure B.3** **Quality metrics for Cottontail DB during the hybrid query large-scale retrieval evaluation presented in Section 8.3.1. Recall and DCG are 1.0 for all execution strategies other than PQ, for which they exhibit rather low values with a large spread.**

# Bibliography


[AAA+14] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The Beckman Report on Database Research. *ACM Sigmod Record*, 43(3):61–70, December 2014. ISSN: 0163-5808. DOI: 10.1145/2694428.2694441.

[AAA+20] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. The Seattle Report on Database Research. *ACM Sigmod Record*, 48(4):44–53, February 2020. ISSN: 0163-5808. DOI: 10.1145/3385658.3385668.

[AAB+05] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The Lowell Database Research Self-Assessment. *Communications of The ACM*, 48(5):111–118, May 2005. ISSN: 0001-0782. DOI: 10.1145/1060710.1060718.

[AN97] D.A. Adjeroh and K.C. Nwosu. Multimedia Database Management-Requirements and Issues. *IEEE MultiMedia*, 4(3):24–33, 1997. DOI: 10.1109/93.621580.

[AAB+08] Rakesh Agrawal, Anastasia Ailamaki, Philip A Bernstein, Eric A Brewer, Michael J Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J Franklin, Hector Garcia-Molina, et al. The Claremont Report on Database Research. *ACM Sigmod Record*, 37(3):9–19, 2008.

[AIA14] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, New York, NY, USA. Association for Computing Machinery, 2014. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2610502.

[Ams14] Laurent Amsaleg. *A Database Perspective on Large Scale High-Dimensional Indexing*. PhD thesis, Université Rennes, 2014.

[AG08] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys*, 40(1):1–39, 2008.

[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.

[ABF17] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. In *Proceedings of the 2017 International Conference on Similarity Search and Applications*, pages 34–49, 2017.

[BL14] Artem Babenko and Victor Lempitsky. The Inverted Multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(6):1247–1260, 2014.

[BL16] Artem Babenko and Victor Lempitsky. Efficient Indexing of Billion-scale Datasets of Deep Descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

[BK17] Tobias Baer and Vishnu Kamalnath. Controlling Machine-Learning Algorithms and their Biases. *McKinsey Insights*, 2017.

[BPS+20] Abhishek Bagade, Ashwini Pale, Shreyans Sheth, Megha Agarwal, Soumen Chakrabarti, Kameswari Chebrolu, and S. Sudarshan. The Kauwa-Kaate Fake News Detection System: Demo. In *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*, CoDS COMAD 2020, pages 302–306, New York, NY, USA, 2020. ISBN: 978-1-4503-7738-6. DOI: 10.1145/3371158.3371402.

[BZ18] Pablo Barberá and Thomas Zeitzoff. The New Public Address System: Why do World Leaders Adopt Social Media? *International Studies Quarterly*, 62(1):121–130, 2018.

[BHM16] Kai Uwe Barthel, Nico Hezel, and Radek Mackowiak. Navigating a Graph of Scenes for Exploring Large Video Collections. In *Proceedings of the 2016 International Conference on Multimedia Modeling*, pages 418–423, 2016.

[BTVG06] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded Up Robust Features. In *European Conference on Computer Vision*, pages 404–417, 2006.

[BM70] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. In *Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA. Association for Computing Machinery, 1970. ISBN: 978-1-4503-7941-0. DOI: 10.1145/1734663.1734671.

[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.

[BCH+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230, New York, NY, USA.

Association for Computing Machinery, 2018. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190662.

[BOV07] Radim Belohlavek, Stanislav Opichal, and Vilem Vychodil. Relational Algebra for Ranked Tables with Similarities: Properties and Implementation. In Michael R. Berthold, John Shawe-Taylor, and Nada Lavrač, editors, *Advances in Intelligent Data Analysis VII*, pages 140–151, Berlin, Heidelberg. Springer Berlin Heidelberg, 2007. ISBN: 978-3-540-74825-0.

[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.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[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.

[BRS+19] Fabian Berns, Luca Rossetto, Klaus Schoeffmann, Christian Beecks, and George Awad. V3C1 Dataset: An Evaluation of Content Characteristics. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 334–338, 2019.

[BDO95] Michael W Berry, Susan T Dumais, and Gavin W O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, 37(4):573–595, 1995.

[BHF+14] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed Snapshot Isolation: Global Transactions Pay Globally, Local Transactions Pay Locally. *The VLDB journal*, 23(6):987–1011, 2014.

[BMCFM+19] Bhairav Bipin Mehta, Simone Coppo, Debra Frances McGivney, Jesse Ian Hamilton, Yong Chen, Yun Jiang, Dan Ma, Nicole Seiberlich, Vikas Gulani, and Mark Alan Griswold. Magnetic Resonance Fingerprinting: A Technical Review. *Magnetic Resonance in Medicine*, 81(1):25–46, 2019.

[BGL+22] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. Machine Learning, Linear Algebra, and More: Is SQL All You Need? In *Proceedings of the 11th Conference on Innovative Data Systems Research*, pages 9–12, 2022.

[BdB+07] Henk M Blanken, Arjen P de Vries, Henk Ernst Blok, and Ling Feng. *Multimedia Retrieval*. Springer, 2007.

[Blo46] Felix Bloch. Nuclear Induction. *Physical Review*, 70(7-8):460, 1946.

[BHC+01] Henk Ernst Blok, Djoerd Hiemstra, Sunil Choenni, Franciska de Jong, Henk M Blanken, and Peter MG Apers. Predicting the Cost-quality Trade-off for Information Retrieval Queries: Facilitating Dtabase Design and Query Optimization. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, pages 207–214, 2001.

[BMS+01] Klemens Böhm, Michael Mlivoncic, Hans-Jörg Schek, Roger Weber, et al. Fast Evaluation Techniques for Complex Similarity Queries. In *VLDB*, volume 1, pages 211–220, 2001.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM*, 51(12):77–85, December 2008. ISSN: 0001-0782. DOI: 10.1145/1409360.1409380.

[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.

[BGJ10] Nicolas Bruno, César Galindo-Legaria, and Milind Joshi. Polynomial Heuristics for Query Optimization. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, pages 589–600, 2010. DOI: 10.1109/ICDE.2010.5447916.

[BJZ13] Nicolas Bruno, Sapna Jain, and Jingren Zhou. Continuous Cloud-Scale Query Optimization and Processing. *Proceedings of the VLDB Endowment*, 6(11):961–972, 2013.

[BBZ12] Petra Budikova, Michal Batko, and Pavel Zezula. Query Language for Complex Similarity Queries. In Tadeusz Morzy, Theo Härder, and Robert Wrembel, editors, *Proceedings of the East European Conference on Advances in Databases and Information Systems*, pages 85–98, Berlin, Heidelberg, 2012. ISBN: 978-3-642-33074-2.

[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.

[CS15] Fatih Cakir and Stan Sclaroff. Adaptive Hashing for Fast Similarity Search. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1044–1052, 2015.

[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.

[CHS+95] Michael J Carey, Laura M Haas, Peter M Schwarz, Manish Arya, William F Cody, Ronald Fagin, Myron Flickner, Allen W Luniewski, Wayne Niblack, Dragutin Petkovic, et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management*, pages 124–131, 1995.

[CSM+15] Stephen F Cauley, Kawin Setsompop, Dan Ma, Yun Jiang, Huihui Ye, Elfar Adalsteinsson, Mark A Griswold, and Lawrence L Wald. Fast Group Matching for MR Fingerprinting Reconstruction. *Magnetic Resonance in Medicine*, 74(2):523–528, 2015.

[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.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA. Association for Computing

Machinery, 1998. ISBN: 0-89791-996-3. DOI: 10.1145/275487.275492.

[CHW+21] Aozhu Chen, Fan Hu, Zihan Wang, Fangming Zhou, and Xirong Li. What Matters for Ad-hoc Video Search? A Large-scale Evaluation on TRECVID. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2317–2322, 2021.

[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.

[CZW+21] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient Billion-Scale Approximate Nearest Neighborhood Search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

[CPR+07] Flavio Chierichetti, Alessandro Panconesi, Prabhakar Raghavan, Mauro Sozio, Alessandro Tiberi, and Eli Upfal. Finding Near Neighbors through Cluster Pruning. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 103–112, 2007.

[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.

[Ciu22] Dan Ciuriak. The Role of Social Media in Russia's War on Ukraine. *Available at SSRN*, 2022.

[Cod70] E. F. Codd. Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[CHN+95] William F Cody, Laura M Haas, Wayne Niblack, Manish Arya, Michael J Carey, Ronald Fagin, Myron Flickner, Denis Lee, Dragutin Petkovic, Peter M Schwarz, et al. Querying Multimedia Data from Multiple Repositories by Content: The Garlic Project. In *Proceedings of the 1995 Working Conference on Visual Database Systems*, pages 17–35, 1995.

[CZR18] Ouri Cohen, Bo Zhu, and Matthew S Rosen. MR Fingerprinting Deep Reconstruction Network (DRONE). *Magnetic Resonance in Medicine*, 80(3):885–894, 2018.

[Cre20] Stefano Cresci. A Decade of Social Bot Detection. *Communications of the ACM*, 63(10):72–83, 2020.

[CLP22] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are You Sure You Want to Use MMAP in Your Database Management System? In *Proceedings of the 2022 Conference on Innovative Data Systems Research (CIDR)*, Chaminade, CA, 2022.

[CSW+19] Limeng Cui, Kai Shu, Suhang Wang, Dongwon Lee, and Huan Liu. DEFEND: A System for Explainable Fake News Detection. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 2961–2964, New York, NY, USA. Association for Computing Machinery, 2019. ISBN: 978-1-4503-6976-3. DOI: 10.1145/3357384.3357862.

[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.

[Xopa] *Data Management: SQL Call Level Interface (CLI)*. X/Open Company Ltd., U.K. ISBN: 1-85912-081-4.

[Xopb] *Data Management: Structured Query Language (SQL) Version 2*. X/Open Company Ltd., U.K. ISBN: 1-85912-151-9.

[DVF+16] Clayton Allen Davis, Onur Varol, Emilio Ferrara, Alessandro Flammini, and Filippo Menczer. Botornot: A system to evaluate social bots. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 273–274, 2016.

[dSP+17] Jonathan de Oliveira Assis, Vanessa CO Souza, Melise MV Paula, and João Bosco S Cunha. Performance Evaluation of NoSQL Data Store for Digital Media. In *Proceedings of the 12th Iberian Conference on Information Systems and Technologies*, pages 1–6, 2017.

[dLG16] Gabriel de Oliveira Barra, Mathias Lux, and Xavier Giro-i-Nieto. Large Scale Content-based Video Retrieval with LIvRE. In *2016 14th International Workshop on Content-based Multimedia Indexing*, pages 1–4, 2016.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[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, 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.

[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.

[DIR+07] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. Adaptive Query Processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[Dha13] Vasant Dhar. Data Science and Prediction. *Communications of the ACM*, 56(12):64–73, 2013.

[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, 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.

[DJ11] Jens Dittrich and Alekh Jindal. Towards a One Size Fits all Database Architecture. In pages 195–198, January 2011.

[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.

[FKL+17] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, Andrew Pavlo, et al. Main Memory Database Systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[Far09] Hany Farid. Image Forgery Detection. *IEEE Signal Processing Magazine*, 26(2):16–25, 2009.

[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, 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 10th International Conference on Information and Knowledge Management*, pages 202–209, 2000.

[FCC+20] Emilio Ferrara, Herbert Chang, Emily Chen, Goran Muric, and Jaimin Patel. Characterizing Social Media Manipulation in the 2020 US Presidential Election. *First Monday*, 2020.

[Fer14] Nicola Ferro, editor. *Bridging Between Information Retrieval and Databases*, volume 8173 of *Lecture Notes in Computer Science*. Springer, 2014. ISBN: 978-3-642-54797-3.

[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.

[FC13] Fausto C Fleites and Shu-Ching Chen. Efficient Content-based Multimedia Retrieval using Novel Indexing Structure in PostgreSQL. In *Proceedings of the 2013 IEEE International Symposium on Multimedia*, pages 500–501, 2013.

[FSN+95] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Qian Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by Image and Video Content: The QBIC System. *Computer*, 28(9):23–32, 1995. DOI: 10.1109/2.410146.

[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, 2000.

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pages 1433–1445, 2018.

[FT21] Henning Funke and Jens Teubner. Low-latency compilation of SQL queries to machine code. *Proceedings of the VLDB Endowment*, 14(12):2691–2694, July 2021. ISSN: 2150-8097. DOI: 10.14778/3476311.3476321.

[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.

[GS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullmann, and Jennifer Widom. *DATABASE SYSTEMS The Complete Book, 2nd Edition*. second edition, 2009. ISBN: l.

[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 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.

[GRS19b] Ralph Gasser, Luca Rossetto, and Heiko Schuldt. Towards an All-purpose Content-based Multimedia Information Retrieval System. *CoRR*, abs/1902.03878, 2019.

[GHK+14] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, 2014. DOI: 10.1109/TPAMI.2013.240.

[GTK01] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity Estimation using Probabilistic Models. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 461–472, 2001.

[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.

[GAKS14] Ivan Giangreco, Ihab Al Kabary, and Heiko Schuldt. ADAM: A Database and Information Retrieval System for Big Multimedia Collections. In *Proceedings of the 2014 IEEE International Congress on Big Data*, pages 406–413, 2014.

[GSP+19] Ivan Giangreco, Loris Sauter, Mahnaz Amiri Parian, Ralph Gasser, Silvan Heller, Luca Rossetto, and Heiko Schuldt. VIRTUE: A Virtual Reality Museum Experience. In *Proceedings*

*of the 24th International Conference on Intelligent User Uterfaces*, pages 119–120, 2019.

[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, 2016.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[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.

[GJA10] Gylfi Þór Gudmundsson, Björn Þór Jónsson, and Laurent Amsaleg. A Large-scale Performance Study of Custer-based High-Dimensional Indexing. In *Proceedings of the International Workshop on Very-large-scale Multimedia Corpus, Mining and Retrieval*, VLS-MCMR '10, pages 31–36, New York, NY, USA. Association for Computing Machinery, 2010. ISBN: 978-1-4503-0166-4. DOI: 10.1145/1878137.1878145.

[GdR+09] Denise Guliato, Ernani V de Melo, Rangaraj M Rangayyan, and Robson C Soares. PostgreSQL-IE: An Image-Handling Extension for PostgreSQL. *Journal of Digital Imaging*, 22(2):149–165, 2009.

[GSL+20] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating Large-scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning*, pages 3887–3896, 2020.

[GHQ95] Ashish Kumar Gupta, Venky Harinarayan, and Dallan Quass. Generalized Projections: A Powerful Approach to Aggregation. In 1995.

[GJS+21] Cathal Gurrin, Björn Þór Jónsson, Klaus Schöffmann, Duc-Tien Dang-Nguyen, Jakub Lokoč, Minh-Triet Tran, Wolfgang Hürst, Luca Rossetto, and Graham Healy. Introduction to the Fourth Annual Lifelog Search Challenge, LSC'21. In *Proceedings of the 2021 International Conference on Multimedia Retrieval*, pages 690–691, 2021.

[GSJ+19] Cathal Gurrin, Klaus Schoeffmann, Hideo Joho, Andreas Leibetseder, Liting Zhou, Aaron Duane, Duc-Tien Dang-Nguyen, Michael Riegler, Luca Piras, Minh-Triet Tran, et al. Comparing Approaches to Interactive Lifelog Search at the Lifelog Search Challenge (LSC2018). *ITE Transactions on Media Technology and Applications*, 7(2):46–59, 2019.

[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.

[HR16] Hadi Hashem and Daniel Ranc. Evaluating NoSQL Document Oriented Data Model. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops*, pages 51–56, 2016.

[HAPG+20] Silvan Heller, Mahnaz Amiri Parian, Ralph Gasser, Loris Sauter, and Heiko Schuldt. Interactive Lifelog Retrieval with vitrivr. In *Proceedings of the 3rd Annual Workshop on Lifelog Search Challenge*, pages 1–6, 2020.

[HAG+22] Silvan Heller, Rahel Arnold, Ralph Gasser, Viktor Gsteiger, Mahnaz Parian-Scherb, Luca Rossetto, Loris Sauter, Florian Spiess, and Heiko Schuldt. Multi-Modal Interactive Video Retrieval with Temporal Queries. In *Proceedings of the 2022 International Conference on Multimedia Modeling*, pages 493–498, 2022.

[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 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.

[HGP+21] Silvan Heller, Ralph Gasser, Mahnaz Parian-Scherb, Sanja Popovic, Luca Rossetto, Loris Sauter, Florian Spiess, and Heiko Schuldt. Interactive Multimodal Lifelog Retrieval with Vitrivr at LSC 2021. In *Proceedings of the 4th Annual on Lifelog Search Challenge*, pages 35–39, 2021.

[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, 2020.

[HSH07] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a Database System*. Now Publishers Inc, 2007.

[Hen19] Gavin Henry. Howard Chu on Lightning Memory-Mapped Database. *Ieee Software*, 36(06):83–87, 2019.

[HSJ+22] Nico Hezel, Konstantin Schall, Klaus Jung, and Kai Uwe Barthel. Efficient Search and Browsing of Large-scale Video Collections with Vibro. In *International Conference on Multimedia Modeling*, pages 487–492, 2022.

[Hir77] Daniel S Hirschberg. Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[HJB19] Anders Munck Højsgaard, Björn Þór Jónsson, and Philippe Bonnet. Index Maintenance Strategy and Cost Model for Extended Custer Pruning. In Giuseppe Amato, Claudio Gennaro, Vincent

Oria, and Miloš Radovanović, editors, *Similarity Search and Applications*, pages 32–39, Cham. Springer International Publishing, 2019. ISBN: 978-3-030-32047-8.

[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.

[HPS+17] Nina Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. HyPerInsight: Data Exploration Deep Inside HyPer. In *Proceedings of the 2017 ACM Conference on Information and Knowledge Management*, CIKM '17, pages 2467–2470, New York, NY, USA. Association for Computing Machinery, 2017. ISBN: 978-1-4503-4918-5. DOI: 10.1145/3132847.3133167.

[Hür20] Manuel Hürbin. *Retrieval Optimization in Magnetic Resonance Fingerprinting*. PhD thesis, University of Basel, Basel, Switzerland, 2020.

[IGN+12] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. MonetDB: Two Decades of Research in Column-Oriented Database. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[IKM07] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Database Cracking. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, volume 7, pages 68–78, 2007.

[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.

[Ioa03] Yannis Ioannidis. The History of Histograms (abridged). In *Proceedings 2003 Conference on Very Large Databases*, pages 19–30, 2003.

[Iwa16] Masajiro Iwasaki. Pruned Bi-directed k-Nearest Neighbor Graph for Proximity Search. In *Proceedings of the 2016 International Conference on Similarity Search and Applications*, pages 20–33. Springer, 2016.

[JK84] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984. ISSN: 0360-0300. DOI: 10.1145/356924.356928.

[JK02] Kalervo Järvelin and Jaana Kekäläinen. Cumulated Gain-based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

[JLX21] Ehsan Javanmardi, Sifeng Liu, and Naiming Xie. Exploring the Philosophical Foundations of Grey Systems Theory: Subjective Processes, Information Extraction and Knowledge Formation. *Foundations of Science*, 26, June 2021. DOI: 10.1007/s10699-020-09690-0.

[JSDS+19] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. *Advances in Neural Information Processing Systems*, 32, 2019.

[JDS10] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2010.

[JDJ19] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[JWZ+16] Björn Þór Jónsson, Marcel Worring, Jan Zahálka, Stevan Rudinac, and Laurent Amsaleg. Ten Research Questions for Scalable Multimedia Analytics. In *Proceedings of the 2016 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.

[KCH95] Patrick M Kelly, T Michael Cannon, and Donald R Hush. Query by Image Example: The Comparison Algorithm for Navigating Digital Image Databases (CANDID) Approach. *Storage and Retrieval for Image and Video Databases III*, 2420:238–248, 1995.

[Kha21] Firas Layth Khaleel. An Overview on Multimedia Big Data Analytics. In *2021 International Conference on Electrical Engineering and Informatics*, pages 1–6, 2021. DOI: 10.1109/ICEEI52609.2021.9611118.

[KUG14] M. Ali-ud-din Khan, Muhammad Fahim Uddin, and Navarun Gupta. Seven V's of Big Data: Understanding Big Data to Extract Value. In *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education*, pages 1–5, 2014. DOI: 10.1109/ASEEZone1.2014.6820689.

[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.

[KCG+03] Serkan Kiranyaz, Kerem Caglar, Esin Guldogan, Olcay Guldogan, and Moncef Gabbouj. MUVIS: A content-based Multimedia Indexing and Retrieval Framework. In *Proceedings of the 7th International Symposium on Signal Processing and Its Applications*, volume 1, pages 1–8, 2003.

[Koh90] Teuvo Kohonen. The Self-Organizing Map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[KC01] Irena Koprinska and Sergio Carrato. Temporal Video Segmentation: A survey. *Signal processing: Image communication*, 16(5):477–500, 2001.

[KM00] Flip Korn and Suresh Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. *ACM Sigmod Record*, 29(2):201–212, 2000.

[KBC+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA. Association for Computing Machinery, 2018. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3196909.

[KVM+20] Miroslav Kratochvíl, Patrik Veselỳ, František Mejzlík, and Jakub Lokoč. SOM-hunter: Video Browsing with Relevance-to-SOM Feedback Loop. In *Proceedings of the 2020 International Conference on Multimedia Modeling*, pages 790–795, 2020.

[KVH21] Alexander Krolik, Clark Verbrugge, and Laurie Hendren. R3d3: Optimized Query Compilation on GPUs. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 277–288, 2021.

[LBB+18] David MJ Lazer, Matthew A Baum, Yochai Benkler, Adam J Berinsky, Kelly M Greenhill, Filippo Menczer, Miriam J Metzger, Brendan Nyhan, Gordon Pennycook, and David Rothschild. The Science of Fake News. *Science*, 359(6380):1094–1096, 2018.

[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.

[LCM+22] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W Lee, and Tae Jun Ham. ANNA: Specialized Architecture for Approximate Nearest Neighbor Search. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Achitecture*, pages 169–183, 2022.

[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, *Proceedings of the 2018 International Conference on*

*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*, 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.

[Lib03] Leonid Libkin. Expressive Power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003.

[LKD+88] Volker Linnemann, Klaus Küspert, Peter Dadam, Peter Pistor, R Erbe, Alfons Kemper, Norbert Südkamp, Georg Walch, and Mechtild Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *VLDB*, pages 294–305, 1988.

[LLN+20] Chong Liu, Defu Lian, Min Nie, and Xia Hu. Online Optimized Product Quantization. In *Proceedings of the 2020 IEEE International Conference on Data Mining*, pages 362–371, 2020.

[LBB+22] Jakub Lokoč, Werner Bailer, Kai Uwe Barthel, Cathal Gurrin, Silvan Heller, Ladislav Peška, Luca Rossetto, Klaus Schoeffmann, Lucia Vadicamo, Stefanos Vrochidis, et al. A Task Category Space for User-centric Comparative Multimedia Search Evaluations. In *International Conference on Multimedia Modeling*, pages 193–204, 2022.

[LBS+18] Jakub Lokoč, Werner Bailer, Klaus Schoeffmann, Bernd Münzer, and George Awad. On Influential Trends in Interactivev Video Retrieval: Video Browser Showdown 2015–2017. *IEEE Transactions on Multimedia*, 20(12):3361–3376, 2018.

[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.

[LWW+20] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. VHP: Approximate Nearest Neighbor Search via Virtual Hypersphere Partitioning. *Proceedings of the VLDB Endowment*, 13(9):1443–1455, 2020.

[LGG+18] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. Scalable Linear Algebra on a Relational Database System. *IEEE Transactions on Knowledge and Data Engineering*, 31(7):1224–1238, 2018.

[LC08] Mathias Lux and Savvas A. Chatzichristofis. LIRE: Lucene Image Retrieval: An Extensible Java CBIR Library. In *Proceedings of the 16th ACM International Conference on Multimedia*, pages 1085–1088, New York, NY, USA, 2008. ISBN: 978-1-60558-303-7. DOI: 10.1145/1459359.1459577.

[MGS+13] Dan Ma, Vikas Gulani, Nicole Seiberlich, Kecheng Liu, Jeffrey L Sunshine, Jeffrey L Duerk, and Mark A Griswold. Magnetic Resonance Fingerprinting. *Nature*, 495(7440):187–192, 2013.

[ML86] Lothar F Mackert and Guy M Lohman. R* Optimizer Validation and Performance Evaluation for Local Queries. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 84–95, 1986.

[MY18] Yu A Malkov and Dmitry A Yashunin. Efficient and Robust Approximate Nearest Neighbor Search using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2018.

[MBK02] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, and Dimitris Papadias, editors, *Proceedings of the 28th International Conference on Very Large Databases*, pages 191–202. San Francisco, 2002. ISBN: 978-1-55860-869-6.

[MCS88] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical Profile Estimation in Database Systems. *ACN Computing Surveys*, 20(3):191–221, September 1988. ISSN: 0360-0300. DOI: 10.1145/62061.62063.

[MS96] Sherry Marcus and V. S. Subrahmanian. Foundations of Multimedia Database Systems. *Journal of the ACM*, 43(3):474–523, May 1996. ISSN: 0004-5411. DOI: 10.1145/233551.233554.

[MPM+14] Debra F McGivney, Eric Pierre, Dan Ma, Yun Jiang, Haris Saybasili, Vikas Gulani, and Mark A Griswold. SVD Compression for Magnetic Resonance Fingerprinting in the Time Domain. *IEEE Transactions on Medical Imaging*, 33(12):2311–2322, 2014.

[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.

[MHL+92] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting fine-Granularity Locking and Partial Rollbacks using Write-ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[MT99] Danilo Montesi and Alberto Trombetta. Similarity Search Through Fuzzy Relational Algebra. In *Proceedings of the 10th International Workshop on Database and Expert Systems Applications*, pages 235–239, 1999.

[MSL+14] Hannes Mühleisen, Thaer Samar, Jimmy Lin, and Arjen De Vries. Old Dogs are Great at New Tricks: Column Stores for IR Prototyping. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 863–866, 2014.

[MMB+04] Henning Müller, Nicolas Michoux, David Bandon, and Antoine Geissbuhler. A Review of Content-Based Image Retrieval Systems in Medical Applications - Clinical Benefits and Future Directions. *International Journal of Medical Informatics*, 73(1):1–23, 2004.

[MU17] Henning Müller and Devrim Unay. Retrieval from and Understanding of Large-Scale Multi-Modal Medical Datasets: A Review. *IEEE Transactions on Multimedia*, 19(9):2093–2104, 2017. DOI: 10.1109/TMM.2017.2729400.

[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.

[NL14] Thomas Neumann and Viktor Leis. Compiling Database Queries into Machine Code. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 37(1):3–11, 2014.

[ÓÞJA+11] Arnar Ólafsson, Björn Þór Jónsson, Laurent Amsaleg, and Herwig Lejsek. Dynamic Behavior of Balanced NV-trees. *Multimedia Systems*, 17(2):83–100, March 2011. ISSN: 1432-1882. DOI: 10.1007/s00530-010-0199-4.

[OT93] Eitetsu Oomoto and Katsumi Tanaka. OVID: Design and implementation of a video-object database system. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):629–643, 1993.

[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, 2015.

[PST+04] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group Nearest Neighbor Queries. In *Proceedings of the 20th International Conference on Data Engineering*, pages 301–312, 2004.

[PTM+05] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate Nearest Neighbor Queries in Spatial Databases. *ACM Transactions on Database Systems*, 30(2):529–576, 2005.

[PLH+21] Johns Paul, Shengliang Lu, Bingsheng He, et al. Database Systems on GPUs. *Foundations and Trends in Databases*, 11(1):1–108, 2021.

[PW10] Ofir Pele and Michael Werman. The Quadratic-Chi Histogram Distance Family. In *European Conference on Computer Vision*, pages 749–762, 2010.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.

[Pet19] Alex Petrov. *Database Internals*. O'Reilly Media, Inc., 2019. ISBN: 978-1-4920-4034-7.

[PR19] Orestis Polychroniou and Kenneth A Ross. Towards Practical Vectorized Analytical Query Engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7, 2019.

[PR20] Orestis Polychroniou and Kenneth A Ross. VIP: A SIMD Vectorized Analytical Query Engine. *The VLDB Journal*, 29(6):1243–1261, 2020.

[PYC+18] Samira Pouyanfar, Yimin Yang, Shu-Ching Chen, Mei-Ling Shyu, and S. S. Iyengar. Multimedia Big Data Analytics: A Survey. *ACM Computing Surveys*, 51(1), January 2018. ISSN: 0360-0300. DOI: 10.1145/3150226.

[Pri08] Dan Pritchett. BASE: An ACID Alternative. *Queue*, 6(3):48–55, 2008.

[PBB+10] Piotr Przymus, Aleksandra Boniewicz, Marta Burzańska, and Krzysztof Stencel. Recursive Query Facilities in Relational Databases: A Survey. In *Database Theory and Application, Bio-Science and Bio-Technology*, pages 89–99. Springer, 2010.

[PTP46] Edward M Purcell, Henry Cutler Torrey, and Robert V Pound. Resonance Absorption by Nuclear Magnetic Moments in a Solid. *Physical Review*, 69(1-2):37, 1946.

[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.

[QFB+19] Thorsten Quandt, Lena Frischlich, Svenja Boberg, and Tim Schatto-Eckrodt. Fake News. *The International Encyclopedia of Journalism Studies*:1–6, 2019.

[RKH+21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning Transferable Visual Models from Natural Language Supervision. In *Proceedings of the 2021 International Conference on Machine Learning*, pages 8748–8763, 2021.

[RM97] Davood Rafiei and Alberto Mendelzon. Similarity-Based Queries for Time Series Data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 13–25, 1997.

[RDR+98] R. Ramakrsihnan, D. Donjerkovic, A. Ranganathan, K.S. Beyer, and M. Krishnaprasad. SRQL: Sorted Relational Query Language. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 84–95, 1998. DOI: 10.1109/SSDM.1998.688114.

[RDG+16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-time Object Detection. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[Rei81] Raymond Reiter. On Closed World Data Bases. In *Readings in Artificial Intelligence*, pages 119–140. Elsevier, 1981.

[RDJ12] George Ritzer, Paul Dean, and Nathan Jurgenson. The Coming of Age of the Prosumer. *American Behavioral Scientist*, 56(4):379–398, 2012.

[RJ10] George Ritzer and Nathan Jurgenson. Production, Consumption, Prosumption: The Nature of Capitalism in the Age of the Digital 'prosumer'. *Journal of Consumer Culture*, 10(1):13–36, 2010.

[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.

[RBM22] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Computing Surveys*, 55(1):1–38, 2022.

[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 vitrivr. 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.

[RBG+21] Luca Rossetto, Matthias Baumgartner, Ralph Gasser, Lucien Heitz, Ruijie Wang, and Abraham Bernstein. Exploring Graph-Querying Approaches in LifeGraph. In *Proceedings of the 4th Annual Workshop on Lifelog Search Challenge*, LSC '21, pages 7–10, New York, NY, USA. Association for Computing Machinery, 2021. ISBN: 978-1-4503-8533-6. DOI: 10.1145/3463948.3469068.

[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, 2021. ISBN: 978-3-030-67835-7.

[RGS19] Luca Rossetto, Ralph Gasser, and Heiko Schuldt. Query by Semantic Sketch. *arXiv preprint arXiv:1909.12526*, 2019.

[RGG+18] Luca Rossetto, Ivan Giangreco, Ralph Gasser, and Heiko Schuldt. Competitive Video Retrieval with vitrivr. In *Proceedings of the 24th International Conference on Multimedia Modelling*, MMM 2018, pages 403–406. Springer International Publishing, 2018. ISBN: 978-3-319-73600-6.

[RGT+16] Luca Rossetto, Ivan Giangreco, Claudiu Tanase, and Heiko Schuldt. Vitrivr: 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.

[RSB21] Luca Rossetto, Klaus Schoeffmann, and Abraham Bernstein. Insights on the V3C2 Dataset. *arXiv preprint arXiv:2105.01475*, 2021.

[RHM97] Yong Rui, Thomas S Huang, and Sharad Mehrotra. Content-Based Image Retrieval with Relevance Feedback in MARS. In *Proceedings of International Conference on Image Processing*, volume 2, pages 815–818, 1997.

[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.

[SFW83] Gerard Salton, Edward A Fox, and Harry Wu. Extended Boolean Information Retrieval. *Communications of the ACM*, 26(11):1022–1036, 1983.

[SWY75] Gerard Salton, Anita Wong, and Chung-Shu Yang. A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[SAPG+20] Loris Sauter, Mahnaz Amiri Parian, Ralph Gasser, Silvan Heller, Luca Rossetto, and Heiko Schuldt. Combining Boolean and Multimedia Retrieval in vitrivr for Large-scale Video Search. In *Proceedings of the 2020 International Conference on Multimedia Modeling*, pages 760–765, 2020.

[SGB+22] Loris Sauter, Ralph Gasser, Abraham Bernstein, Heiko Schuldt, and Luca Rossetto. An Asynchronous Scheme for the Distributed Evaluation of Interactive Multimedia Retrieval. In *Proceedings of*

*the 2nd International Workshop on Interactive Multimedia Retrieval*, October 2022. DOI: 10.1145/3552467.3554797.

[Sch19] Klaus Schoeffmann. Video Browser Showdown 2012-2019: A Review. In *2019 International Conference on Content-based Multimedia Indexing (CBMI)*, pages 1–4, 2019.

[SJD13] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the VLDB Endowment*, 7(2):97–108, 2013.

[SMM99] E Di Sciascio, G Mingolla, and Marina Mongiello. Content-Based Image Retrieval over the Web using Query by Sketch and Relevance Feedback. In *International Conference on Advances in Visual Information Dystems*, pages 123–130, 1999.

[SR12] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34. ACM, 1979. DOI: 10.1145/582095.582099.

[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.

[SHS+08] Jie Shao, Zi Huang, Heng Tao Shen, Xiaofang Zhou, Ee-Peng Lim, and Yijun Li. Batch Nearest Neighbor Search for Video Retrieval. *IEEE Transactions on Multimedia*, 10(3):409–420, 2008. DOI: 10.1109/TMM.2008.917339.

[SXJ21] Yijie Shen, Jin Xiong, and Dejun Jiang. Using Vectorized Execution to Improve SQL Query Performance on Spark. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP 2021, New York, NY, USA. Association for Computing Machinery, 2021. ISBN: 978-1-4503-9068-2. DOI: 10.1145/3472456.3472495.

[SOV+21] Larissa C Shimomura, Rafael Seidi Oyamada, Marcos R Vieira, and Daniel S Kaster. A Survey on Graph-based Methods for Similarity Searches in Metric Spaces. *Information Systems*, 95:101507, 2021.

[SJH+05] R. Siguroardottir, B.P. Jonsson, H. Hauksson, and L. Amsaleg. The Quality vs. Time trade-off for Approximate Image Descriptor Search. In *Proceedings of the 21st International Conference on Data Engineering Workshops (ICDEW'05)*, pages 1175–1175, 2005. DOI: 10.1109/ICDE.2005.294.

[SAA+10] Yasin N. Silva, Ahmed M. Aly, Walid G. Aref, and Per-Ake Larson. SimDB: A Similarity-aware Database System. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1243–1246, 2010. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807330.

[SWA+22] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, et al. Results of the NeurIPS'21 Challenge on Billion-scale Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2205.03763*, 2022.

[SZ03] Sivic and Zisserman. Video Google: A Text Retrieval Approach to Object Matching in Videos. In *Proceedings of the 9th IEEE International Conference on Computer Vision*, 1470–1477 vol.2, 2003. DOI: 10.1109/ICCV.2003.1238663.

[SB11] Tomáš Skopal and Benjamin Bustos. On Nonmetric Similarity Search Problems in Complex Domains. *ACM Computing Surveys (CSUR)*, 43(4):1–50, 2011.

[SWS+00] A.W.M. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content-Based Image Retrieval at the End of the Early Years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(12):1349–1380, 2000. DOI: 10.1109/34.895972.

[SAR+14] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos

Krikellas, and Rhonda Baldwin. Orca: A Modular Query Optimizer Architecture for Big Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 337–348, New York, NY, USA. Association for Computing Machinery, 2014. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595637.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. Compilation in Query Execution. In *Proceedings of the 7th International Workshop on Data Management on New Hardware*, pages 33–40, 2011.

[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.

[SGH+22] Florian Spiess, Ralph Gasser, Silvan Heller, Mahnaz Parian-Scherb, Luca Rossetto, Loris Sauter, and Heiko Schuldt. Multimodal Video Retrieval in Virtual Reality with vitrivr-vr. In *International Conference on Multimedia Modeling*, pages 499–504, 2022.

[SGH+21a] Florian Spiess, Ralph Gasser, Silvan Heller, Luca Rossetto, Loris Sauter, and Heiko Schuldt. Competitive Interactive Video Retrieval in Virtual Reality with vitrivr-VR. In *Proceedings of the 2021 International Conference on Multimedia Modeling*, pages 441–447, 2021.

[SGH+21b] Florian Spiess, Ralph Gasser, Silvan Heller, Luca Rossetto, Loris Sauter, Milan van Zanten, and Heiko Schuldt. Exploring Intuitive Lifelog Retrieval and Interaction Modes in Virtual Reality with VITRIVR-VR. In *Proceedings of the 4th Annual on Lifelog Search Challenge*, pages 17–22, 2021.

[Spi09] David I Spivak. Simplicial Databases. *arXiv preprint arXiv:0904.2012*, 2009.

[SRA+01] Ioana Stanoi, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, volume 2001, pages 99–108, 2001.

[Sto81] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, 1981.

[SBP+11] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The Architecture of SciDB. In *Proceedings of the 2011 International Conference on Scientific and Statistical Database Management*, pages 1–16, 2011.

[SBZ+13] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013. DOI: 10.1109/MCSE.2013.19.

[Swa89] Arun Swami. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 367–376, 1989.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[Tsa07] Chih-Fong Tsai. A Review of Image Retrieval Methods for Digital Cultural Heritage Resources. *Online Information Review*, 2007.

[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.

[TSF+12] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-Based Query Optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 324–335, 2012.

[Uma83] M Umano. Retrieval from Fuzzy Database by Fuzzy Relational Algebra. *IFAC Proceedings Volumes*, 16(13):1–6, 1983.

[Via19] Gregory Vial. Understanding Digital Transformation: A Review and a Research Agenda. *The Journal of Strategic Information Systems*, 28(2):118–144, 2019.

[VHS+20] Marco Vogt, Nils Hansen, Jan Schönholz, David Lengweiler, Isabel Geissmann, Sebastian Philipp, Alexander Stiemer, and Heiko Schuldt. Polypheny-DB: Towards Bridging the Gap Between Polystores and HTAP Systems. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 25–36. Springer, 2020.

[VSS18] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *Proceedings of the 2018 IEEE International Conference on Big Data*, pages 3364–3373, 2018.

[Vu19] Tin Vu. Deep Query Optimization. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1856–1858, New York, NY, USA. Association for Computing Machinery, 2019. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3300104.

[WYG+21] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[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.

[WOK+98] Yasuhiko Watanabe, Yoshihiro Okada, Kengo Kaneji, and Yoshitaka Sakamoto. Multimedia Database System for TV Newscasts and Newspapers. In *International Conference on Advanced Multimedia Content Processing*, pages 208–220, 1998.

[WMZ10] William Webber, Alistair Moffat, and Justin Zobel. A Similarity Measure for Indefinite Rankings. *ACM Transactions on Information Systems*, 28(4), November 2010. ISSN: 1046-8188. DOI: 10.1145/1852102.1852106.

[WB00] Roger Weber and Klemens Böhm. Trading Quality for Time with Nearest-Neighbor Search. In *International Conference on Extending Database Technology*, pages 21–35, 2000.

[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.

[WLK+10] Kyu-Young Whang, Jae-Gil Lee, Min-Soo Kim, Min-Jae Lee, Ki-Hoon Lee, Wook-Shin Han, and Jun-Sung Kim. Tightly-Coupled Spatial Database Features in the Odysseus/OpenGIS DBMS for High-Performance. *GeoInformatica*, 14(4):425–446, 2010.

[WLL+15] Kyu-Young Whang, Jae-Gil Lee, Min-Jae Lee, Wook-Shin Han, Min-Soo Kim, and Jun-Sung Kim. DB-IR Integration using Tight-Coupling in the Odysseus DBMS. *World Wide Web-internet and Web Information Systems*, 18(3):491–520, 2015.

[WZW85] S. K. M. Wong, Wojciech Ziarko, and Patrick C. N. Wong. Generalized Vector Spaces Model in Information Retrieval. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '85, pages 18–25, New York, NY, USA. Association for Computing Machinery, 1985. ISBN: 0-89791-159-8. DOI: 10.1145/253495.253506.

[WCZ+13] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable. In *Proceedings of the 29th IEEE International Conference on Data Engineering*, pages 1081–1092, 2013.

[XTZ18] Donna Xu, Ivor W. Tsang, and Ying Zhang. Online Product Quantization. *IEEE Transactions on Knowledge and Data Engineering*, 30(11):2185–2198, 2018. DOI: 10.1109/TKDE.2018.2817526.

[YPM+19] Fan Yang, Shiva K. Pentyala, Sina Mohseni, Mengnan Du, Hao Yuan, Rhema Linder, Eric D. Ragan, Shuiwang Ji, and Xia (Ben) Hu. XFake: Explainable Fake News Detector with Visualizations. In *Proceedings of the 2019 World Wide Web Conference*, WWW 2019, pages 3600–3604, New York, NY, USA. Association for Computing Machinery, 2019. ISBN: 978-1-4503-6674-8. DOI: 10.1145/3308558.3314119.

[YLF+20] Wen Yang, Tao Li, Gai Fang, and Hong Wei. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2241–2253, 2020.

[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.

[YLK10] Bin Yao, Feifei Li, and Piyush Kumar. K Nearest Neighbor Queries and kNN-joins in Large Relational Databases (almost) for Free. In *2010 IEEE 26th International Conference on Data Engineering*, pages 4–15, 2010.

[YHM15] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust Query Optimization Methods with Respect to Estimation Errors: A Survey. *Sigmod Record*, 44(3):25–36, December 2015. ISSN: 0163-5808. DOI: 10.1145/2854006.2854012.

[YCL+19] Lin Yue, Weitong Chen, Xue Li, Wanli Zuo, and Minghao Yin. A Survey of Sentiment Analysis in Social Media. *Knowledge and Information Systems*, 60(2):617–663, 2019.

[Zad96] Lotfi A Zadeh. Fuzzy Sets. In *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A Zadeh*, pages 394–432. World Scientific, 1996.

[ZW14] Jan Zahálka and Marcel Worring. Towards Interactive, Intelligent, and Integrated Multimedia Analytics. In *Proceedings of the 2014 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 3–12, 2014. DOI: 10.1109/VAST.2014.7042476.

[ZCB+18] Savvas Zannettou, Tristan Caulfield, Jeremy Blackburn, Emiliano De Cristofaro, Michael Sirivianos, Gianluca Stringhini, and Guillermo Suarez-Tangil. On the Origins of Memes by Means of Fringe Web Communities. In *Proceedings of the Internet Measurement Conference 2018*, pages 188–202, 2018.

[ZCL+10] Daniel Zeng, Hsinchun Chen, Robert Lusch, and Shu-Hsing Li. Social media Analytics and Intelligence. *IEEE Intelligent Systems*, 25(6):13–16, 2010. DOI: 10.1109/MIS.2010.151.

[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.

[ZHC+06] Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A Lang, and Yuan-chi Chang. Boolean+ ranking: Querying a Database by k-constrained Optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 359–370, 2006.

[ZWN22] Wanlei Zhao, Hui Wang, and Chong-Wah Ngo. Approximate k-NN Graph Construction: A Generic Online Approach. *IEEE Transactions on Multimedia*, 24:1909–1921, 2022.

[ZTL20] Weijie Zhao, Shulong Tan, and Ping Li. SONG: Approximate Nearest Neighbor Search on GPU. In *Proceedings of the 36th IEEE International Conference on Data Engineering*, pages 1033–1044, 2020.

[ZZ20] Xinyi Zhou and Reza Zafarani. A Survey of Fake News: Fundamental Theories, Detection Methods, and Opportunities. *ACM Computing Surveys (CSUR)*, 53(5):1–40, 2020.

[ZCW+15] Wenwu Zhu, Peng Cui, Zhi Wang, and Gang Hua. Multimedia Big Data Computing. *IEEE Multimedia*, 22(3):96–105, 2015.

[Zih21] Gabriel Zihlmann. *Magnetic Resonance Fingerprinting Reconstruction Using Methods from Multimedia Retrieval*. PhD thesis, University of Basel, Basel, Switzerland, 2021.

# 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 Tesearch 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

**2022**

- Loris Sauter, Ralph Gasser, Abraham Bernstein, et al. An Asynchronous Scheme for the Distributed Evaluation of Interactive Multimedia Retrieval. In *Proceedings of the 2nd International Workshop on Interactive Multimedia Retrieval*, October 2022. DOI: 10.1145/3552467.3554797

- Silvan Heller, Rahel Arnold, Ralph Gasser, et al. Multi-Modal Interactive Video Retrieval with Temporal Queries. In *Proceedings of the 2022 International Conference on Multimedia Modeling*, pages 493–498, 2022

- Florian Spiess, Ralph Gasser, Silvan Heller, et al. Multi-modal Video Retrieval in Virtual Reality with vitrivr-vr. In *International Conference on Multimedia Modeling*, pages 499–504, 2022

**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, 2021. ISBN: 978-3-030-67835-7

- Silvan Heller, Ralph Gasser, Mahnaz Parian-Scherb, et al. Interactive Multimodal Lifelog Retrieval with Vitrivr at LSC 2021. In *Proceedings of the 4th Annual on Lifelog Search Challenge*, pages 35–39, 2021

- Luca Rossetto, Matthias Baumgartner, Ralph Gasser, et al. Exploring Graph-Querying Approaches in LifeGraph. In *Proceedings of the 4th Annual Workshop on Lifelog Search Challenge*, LSC '21, pages 7–10, New York, NY, USA. Association for Computing Machinery, 2021. ISBN: 978-1-4503-8533-6. DOI: 10.1145/3463948.3469068

- Florian Spiess, Ralph Gasser, Silvan Heller, et al. Exploring Intuitive Lifelog Retrieval and Interaction Modes in Virtual Reality with VITRIVR-VR. in *Proceedings of the 4th Annual on Lifelog Search Challenge*, pages 17–22, 2021

- 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

– Florian Spiess, Ralph Gasser, Silvan Heller, et al. Competitive Interactive Video Retrieval in Virtual Reality with vitrivr-VR. in *Proceedings of the 2021 International Conference on Multimedia Modeling*, pages 441–447, 2021

– 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

– Silvan Heller, Mahnaz Amiri Parian, Ralph Gasser, et al. Interactive Lifelog Retrieval with vitrivr. In *Proceedings of the 3rd Annual Workshop on Lifelog Search Challenge*, pages 1–6, 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

– Loris Sauter, Mahnaz Amiri Parian, Ralph Gasser, et al. Combining Boolean and Multimedia Retrieval in vitrivr for Large-scale Video Search. In *Proceedings of the 2020 International Conference on Multimedia Modeling*, pages 760–765, 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)

– Ivan Giangreco, Loris Sauter, Mahnaz Amiri Parian, et al. VIRTUE: A Virtual Reality Museum Experience. In *Proceedings of the 24th International Conference on Intelligent User Uterfaces*, pages 119–120, 2019

– **Rossetto2019:Retrieval**

– Jakub Lokoč, Gregor Kovalčík, Bernd Münzer, et al. 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

– Luca Rossetto, Mahnaz Amiri Parian, Ralph Gasser, et al. Deep Learning-based Concept Detection in vitrivr. 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 vitrivr. In *Proceedings of the 24th International Conference on Multimedia Modelling*, MMM 2018, pages 403–406. Springer International Publishing, 2018. ISBN: 978-3-319-73600-6

## Advised Theses

– Colin Saladin. *SIMD Support for Proximity Based Queries in Cottontail DB*. Bachelor's Thesis, University of Basel

– 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 VBS and paper titled *Towards Explainable Interactive Multi-Modal Video Retrieval with vitrivr* 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 vitrivr* at the *ACM International Conference on Multimedia Retrieval (ICMR 2019)*

**Best VBS System Award** for the winning system of the VBS and paper titled *Deep Learning-based Concept Detection in vitrivr* 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 ETH Zürich) honours the best thesis at a Swiss University every year.

## Other Relevant Experience

– Technical Program Committee (PC) Member of the *International Workshop on Interactive Multimedia Retrieval (IMuR 2022)*

- Technical Program Committee (PC) Member of the *International Conference on Multimedia Modeling (MMM 2022)*
- Technical Program Committee (PC) Member of the *Fourth Annual Workshop on Lifelog Search Challenge (LSC 2021)*
- Technical Program Committee (PC) Member of the *Second International Workshop on Video Retrieval Methods and Their Limits (ViRaL 2021)*
- Technical Program Committee (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