Universität des Saarlandes

Challenging Traditional Views and Techniques in Relational Query Processing and Indexing

Joris Nix

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, 2024

Day of Colloquium

May 6, 2025

Dean of the Faculty

Prof. Dr. Roland Speicher

Examination Board

Chair

Prof. Dr. Markus Bläser

Reviewers

Prof. Dr. Jens Dittrich

Prof. Dr. Jana Giceva

Prof. Dr. Thomas Heinis

Academic Assistant

Dr. Marcel Maltry

Abstract

Query processing and indexing are fundamental components of every relational database management system. These areas are concerned with core aspects such as performance, design, and maintenance, which have been the subject of extensive, longstanding research. As a result, well-established methods and paradigms have emerged, particularly in query optimization, index construction, and the design and functionality of SQL. In this thesis, we aim to challenge and redefine some of these traditional views and techniques.

In the first part of this thesis, we question the translation of a logical plan to a physical plan on the granularity of complete operators during query optimization. Instead, we propose to deepen this process by breaking up the abstraction of an operator to consider more fine-granular subcomponents, enabling additional optimization potential. Our experimental validation demonstrates the impact of varying physical representations of a logical operator and highlights that a more holistic optimization approach can significantly improve estimated query plan quality.

In the second part of this thesis, we aim to apply this approach specifically to index structures, which are generally considered monolithic and hand-crafted entities tailored to specific use cases. We propose a generic indexing framework that breaks up index structures by separating a logical index from a physical index similar to the split into logical and physical operators. Furthermore, we formulate index construction as an optimization problem that we solve using genetic programming. Our experiments show that our approach successfully rediscovers existing baselines. In addition, an optimized index tailored to a specific dataset and workload not only matches, but in some cases, surpasses the performance of traditional indexes.

In the third part of this thesis, we propose a single keyword extension to SQL that breaks up the single-table result limitation by allowing to return a subdatabase. This subdatabase contains the tables that participate in the query, each reduced to those tuples that contribute to the traditional query result. We present four SQL-based rewrite methods and an efficient

native algorithm that we implemented in a database system with a state-of-the-art compiling query execution engine. The experimental evaluation shows that multiple individual result sets significantly reduce the overall result set size, with our methods adding minimal overhead to the query execution time and, in some cases, even outperforming traditional, single-table execution.

Zusammenfassung

Abfrageverarbeitung und Indizierung sind grundlegende Komponenten jedes relationalen Datenbankmanagementsystems. Diese Bereiche befassen sich mit zentralen Aspekten wie Leistung, Design und Wartung, die seit langem Gegenstand umfangreicher Forschung sind. Infolgedessen sind fest etablierte Methoden und Paradigmen entstanden, insbesondere in der Abfrageoptimierung, der Indexkonstruktion sowie im Design und der Funktionalität von SQL. In dieser Arbeit wollen wir einige dieser traditionellen Ansichten und Techniken infrage stellen und neu definieren.

Im ersten Teil dieser Arbeit hinterfragen wir die Übersetzung eines logischen Plans in einen physischen Plan auf der Granularität vollständiger Operatoren während der Abfrageoptimierung. Stattdessen schlagen wir vor, diesen Prozess zu vertiefen, indem wir die Abstraktion eines Operators aufbrechen, um feingranularere Teilkomponenten zu betrachten, die zusätzliches Optimierungspotenzial ermöglichen. Unsere experimentelle Validierung zeigt die Auswirkungen unterschiedlicher physischer Repräsentationen eines logischen Operators und hebt hervor, dass ein ganzheitlicherer Optimierungsansatz die geschätzte Qualität des Abfrageplans erheblich verbessern kann.

Im zweiten Teil dieser Arbeit wollen wir diesen Ansatz speziell auf Indexstrukturen anwenden, die im Allgemeinen als monolithische und handgefertigte Einheiten betrachtet werden, die auf spezifische Anwendungsfälle zugeschnitten sind. Wir schlagen ein generisches Index-Framework vor, das Indexstrukturen aufbricht, indem es einen logischen Index von einem physischen Index trennt, ähnlich wie bei der Aufteilung in logische und physische Operatoren. Darüber hinaus formulieren wir die Indexkonstruktion als ein Optimierungsproblem, das wir mithilfe genetischer Programmierung lösen. Unsere Experimente zeigen, dass unser Ansatz in der Lage ist, bestehende Referenzindexe wiederzuentdecken. Des Weiteren ist ein optimierter Index, der auf einen spezifischen Datensatz und Arbeitslast zugeschnitten ist, nicht nur in der Lage die Leistung traditioneller Indexe zu erreichen, sondern diese in bestimmten Szenarien

sogar zu übertreffen.

Im dritten Teil dieser Arbeit schlagen wir eine Erweiterung von SQL um ein einziges Schlüsselwort vor, die die Einschränkung des Ergebnisses auf eine einzige Tabelle aufbricht, indem sie die Rückgabe einer Teildatenbank ermöglicht. Diese Teildatenbank enthält die an der Abfrage beteiligten Tabellen, die jeweils auf die Tupel reduziert sind, die zum traditionellen Abfrageergebnis beitragen. Wir stellen vier SQL-basierte Umformungsmethoden und einen effizienten nativen Algorithmus vor, den wir in einem Datenbanksystem mit einer modernen kompilierenden Engine zur Ausführung von Abfragen implementiert haben. Die experimentelle Auswertung zeigt, dass mehrere individuelle Ergebnisse die Gesamtgröße des Ergebnisses erheblich reduzieren, wobei unsere Methoden nur minimalen zusätzlichen Aufwand bezüglich der Ausführungszeit der Abfrage verursachen und in einigen Fällen sogar die traditionelle Ausführung mit nur einer Ergebnistabelle übertreffen.

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Dr. Jens Dittrich, for giving me the opportunity to pursue my Ph.D. His inspiring and often unconventional ideas sparked my curiosity, leading to many invaluable discussions and exciting research topics. His experience, support, and the freedom he provided allowed me to tackle every challenge that arose during my studies.

I would also like to thank Prof. Dr. Jana Giceva and Prof. Dr. Thomas Heinis for reviewing this thesis. It is a great honor to have such distinguished researchers evaluate my thesis, and I am grateful for their time and expertise.

Furthermore, I want to sincerely thank my exceptional current and former colleagues for their support, both in overcoming the difficulties of work and in navigating everyday life. I am especially grateful to Marcel, Immanuel, Luca, and Simon who have also become wonderful friends. Special thanks go to my co-author, Christian, with whom I had the pleasure of collaborating.

I am also deeply thankful to the fellow students I met during my undergraduate and graduate studies, who helped me persevere through my academic journey, with some becoming dear friends, especially Christopher.

Finally, I feel incredibly fortunate and grateful for my amazing family and friends, who have supported me unconditionally throughout my doctoral studies and my entire life.

Additionally, I would like to acknowledge the use of ChatGPT for assistance with spelling, grammar, translation, and overall refinement of my writing.

Contents

Chapter 1

Introduction

In an ever increasingly digitalized world, database management systems (DBMSs) are an integral part to store and manage tremendous amounts of data. DBMSs are so widespread, that they can be found practically everywhere in everyday life. For example, each electronic device contains at least one database instance, often even several. The authors of SQLite, probably the most widely used database system in the world, claim that "there are over one trillion (1e12) SQLite databases in active use" [Hip], being used in smartphones, computers, smart TVs, and apps among others. On the one hand, this widespread highlights the exceptional importance of DBMSs today. On the other hand, it also draws attention to the many different challenges and requirements database systems face. In general, such systems require high availability and reliability, and have to ensure consistency and durability. Besides those fundamental requirements, database systems are confronted with additional overarching goals and objectives, two of which we examine more closely. First, *performance* is one of the core aspects and main concerns of most DBMSs, coming in many different flavors like throughput, latency, or memory consumption. Depending on the system requirements, a poor DBMS performance can render an application useless. Second, the *design and maintenance effort* required to build a DBMS is a crucial aspect, encompassing architectural decisions and essential components such as storage management, concurrency control, recovery mechanisms, and query processing, including optimization techniques. Obviously, a good design is essential for being able to continuously maintain and improve a system.

Certainly, the aforementioned objectives are deeply rooted in the database community and have been part of longstanding and extensive research. Therefore, each of those research areas has well-established solutions for various problems in the respective domain. This thesis

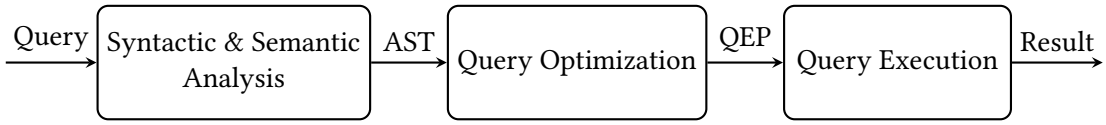


Figure 1.1: High-level overview of the different query processing steps.

aims to address specific parts of both areas from a different angle and propose novel solutions by breaking up traditional views and techniques in query processing and indexing. Specifically, we first tackle the translation of logical plans to physical plans on the granularity of complete operators during query optimization by breaking up traditional operators into more fine-granular subcomponents. Second, we break up monolithic index structures by separating a logical index from a physical index, and automatically generate index structures by formulating index construction as an optimization problem that we solve using genetic programming. Third, we address the limitation of single-table query results in SQL by breaking up the results into multiple individual result sets, each containing only the tuples contributing to the traditional query output.

1.1 Breaking Up Operators in Query Optimization

Query optimization (QO) is at the heart of every relational database system and probably the most important optimization problem in databases. It is part of a larger processing pipeline that consists of various steps. Figure 1.1 depicts a high-level overview of the main query processing stages. For a given query, a system first conducts a syntactic and semantic analysis of the query, resulting in an abstract syntax tree (AST) that represents the query to be executed. In the query optimization phase, based on this AST, a query execution plan (QEP) is computed. Finally, this QEP is executed by the query execution engine to compute the final result.

The underlying goal of QO is to enable fast and efficient query processing by determining the best *physical* query execution plan under a given cost model. To achieve this, QO is traditionally comprised of two steps [PHH92], *rule-based* optimization and *cost-based* optimization. Figure 1.2 gives a broad overview of the internals of the query optimization step.

Upon entering the QO phase, the AST is transformed into a logical plan consisting of logical operators like projections or joins as introduced by Codd [Cod70]. This logical plan is essentially a relational algebra expression in tree form. Based on this initial logical plan, a set of rules is applied to transform the represented query into a semantically equivalent one [Cod72].

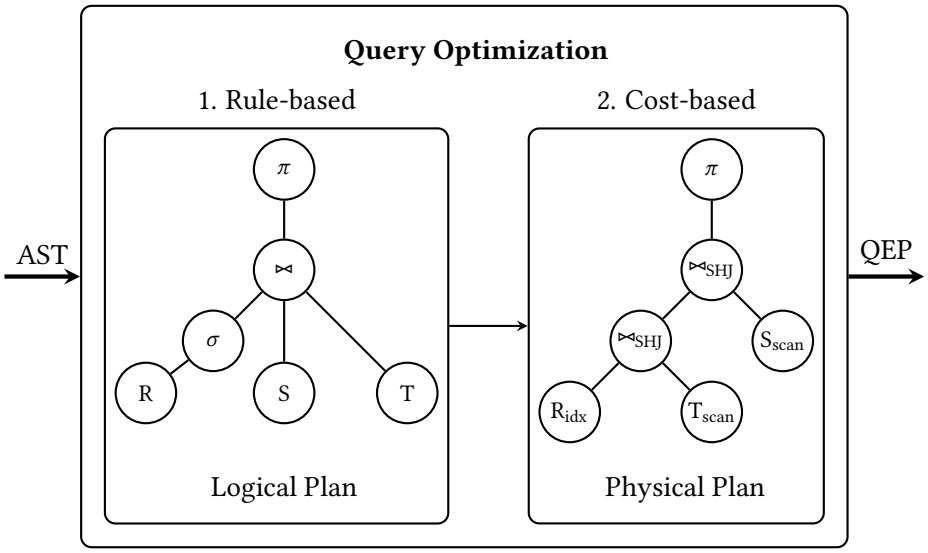


Figure 1.2: Query optimization steps.

These rules, for example, include pushing down predicates and projections, replacing Cartesian products through joins, unnesting of (sub-) queries [NK15], and fusing operators as in the groupjoin [MN11]. The common premise for these rewrite rules is that the transformed query plan heuristically improves performance based on the chosen metric.

After all suitable rules have been applied, the cost-based optimization step determines a join order and produces a physical plan by choosing a physical implementation for each logical operator. The most commonly used approaches to compute a join order are either bottom-up [Sel+79; VM96; Moe+06] or top-down [DT07; FM11] plan enumeration using dynamic programming. Constructing the physical plan primarily includes deciding how database tables are accessed, e.g., index-based or scan-based, and choosing a concrete physical implementation for the logical operators, e.g., simple hash join or index-nested loops join. Note that, depending on the specific implementation of a DBMS, determining the join order and selecting a physical representation for a logical operator can be closely interconnected. For example, considering so called 'interesting properties' like the sortedness of a relation on a specific attribute, can enable different physical implementations and in turn, favor a different join order. Furthermore, there might even be no clear separation between rule-based and cost-based optimization and the boundary becomes rather blurred.

In summary, the fundamental parts of QO are to apply certain rewrite rules, to determine a join order, and to select a physical implementation for each logical operator in the plan. In particular, the de facto standard since the '70s is to map these 'coarse-granular' logical opera-

tors in the logical plan to physical operators. However, we argue that this potentially misses significant optimization opportunities. For example, if we decide to implement a logical join operation using a simple hash join, the DBMS usually provides one specific implementation of this operator. As already observed by Richter et al. [Ric+15], a hash table has many different dimensions which significantly influence performance like the type of the hash table, the hashing scheme, or the hash function. Therefore, choosing the concrete physical implementation for a logical operation can be considered an optimization task in itself. However, in virtually all the aforementioned works, an operator serves as the fundamental unit of the optimization process and is regarded as an inseparable unit.

In Chapter 2, we introduce the concept of *Deep Query Optimization* (DQO), which essentially proposes to break up logical and physical operators into more fine-granular subcomponents. We argue that instead of a direct translation from a logical operator to a physical one, there should be a series of unnesting steps, each progressively increasing the physicality of the query plan. Through this approach, we aim to apply optimizations both within individual operators but also across typical operator boundaries.

1.2 Breaking Up Index Structures

Indexing is one of the most important and performance critical concepts in database systems. Each year a myriad of indexing papers is published. Figure 1.3 shows the cumulative number of papers that have the term 'index' in their title and were published in VLDB, SIGMOD, ICDE, and CIDR from 1975 until today. Importantly, by now, there are at least about 800 indexing or indexing-related papers, which only represents a lower bound. One of the reason for this high number of papers each and every year is that index structures are often built upon existing work and tailored for specific use cases. In particular, these index structures are carefully designed and implemented as immutable, monolithic entities.

Let us take the ubiquitous B-tree as an example, which was already introduced in 1972 by Bayer and McCreight [BM72]. Since then, there has been a tremendous amount of follow-up papers that either improve upon the original B-tree or adapt it for specific use cases. While the B-tree in its original form stores keys and payloads at all levels, the currently prevalent B^+-tree [Knu73] only uses separator values in internal nodes and stores all keys and corresponding payloads in the leaves. This especially improves performance when sequentially scanning the leaves which are usually linked. Among this multitude of works, Rao and Ross introduce CSB^+-trees [RR00], a cache-conscious variant that stores child nodes contiguously in memory, improving cache

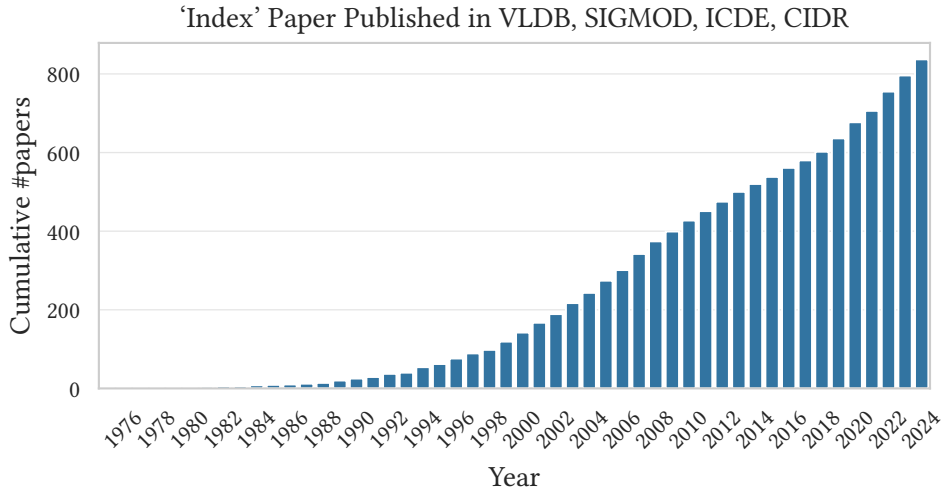


Figure 1.3: Cumulative number of papers that contain the term 'index' in their title and were published in VLDB, SIGMOD, ICDE, and CIDR. Data taken from https://dblp.org/.

locality and reducing the required amount of pointers. However, storing nodes contiguously also makes updates in the index much harder. In contrast to that, there is also work on improving B$^+$-trees in scenarios with high update rates, for example by Graefe [Gra04]. Even to this day, new variants like the DB$^+$-tree [Kwo+23] get published, which improves the branching behavior by redesigning the internal node structure.

All those extensions of the original B-tree commonly adapt the data structure based on underlying characteristics and requirements such as hardware, disk-based vs main-memory-based system, and the concrete workload. At the same time, they all use fundamentally the same 'hard-coded' node structures. In particular, they generally do not use different types of search methods or different data layouts within the same index structure. Furthermore, there are also many different other index structures like the Adaptive Radix Tree (ART) [Lei+13] or the Recursive Model Index (RMI) [Kra+18] that are applicable and show superior performance in certain situations.

In Chapter 3, we introduce a novel indexing framework called *Genetic Generic Generation of Index Structures* (GENE), which enables the automatic generation of index structures. GENE is based on the observation that most index structures are assembled along a few key dimensions, including structural building blocks, invariants, and decisions on the internal layout of nodes. Building on our DQO paper's idea of breaking up logical and physical operators into finer-grained subcomponents, we now aim to apply this approach specifically to index struc-

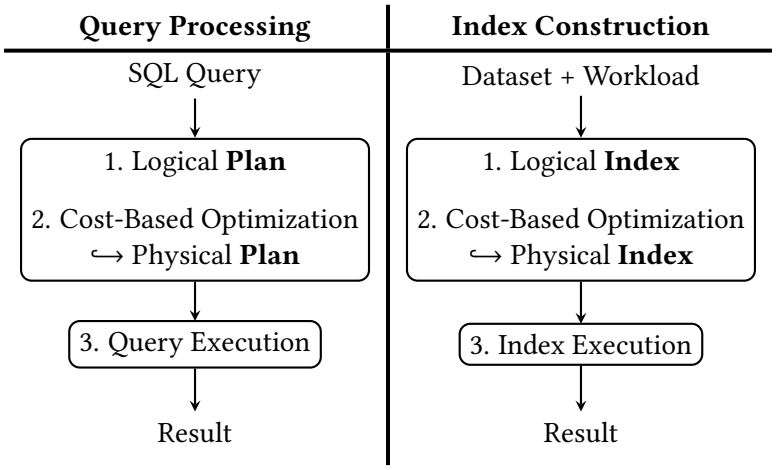


Figure 1.4: Analogy of index construction to query processing.

tures. Figure 1.4 shows a side-by-side comparison of the optimization process during query processing and index construction. Given a dataset and workload, our goal is to find the 'best' physical index in terms of throughput, memory consumption, or any other performance metric, resulting in an optimization problem. Concretely, we first define a logical index that is mostly concerned with partitioning the dataset. Afterward, we apply cost-based optimization to generate a physical index that decides on the specific search algorithm and data layout inside a node. Based on this, we either use interpretation during query execution to access our physical index, or we can even generate optimized code representing the final index structure. In summary, in GENE we clearly separate a logical from a physical index and formulate index construction as an optimization problem. We address this problem using genetic programming to automatically generate index structures.

1.3 Breaking Up SQL's Single-Table Results

Users interacting with a database management system fundamentally have two points of contact with that system. First, the database schema or external view that describes the structure and organization of the data. Second, the query language that uses this basic description of the data to query the database. When defining the database schema, one very important aspect is *data normalization* which aims at reducing data redundancy while establishing and improving data integrity. Regarding the query language, the *Structured Query Language* (SQL) invented by Chamberlin and Boyce [CB74] in the early 1970s is the most widespread language used for querying relational database systems. However, using SQL is limited to produce a single-table

result, even when querying multiple different relations. This introduces two major problems. First, denormalizing the original tables through joins is bound to introduce duplicated values, i.e., we have the problem of relational information redundancy. This conflicts with the initial goal of normalizing the data. Another problem is the loss of relational information. Since SQL is both, a data retrieval and a data transformation language, during the process of computing a query result, some information can get lost. For instance, the information where a specific attribute in the result table comes from or that duplicated values originate from the same tuple can get lost.

While significant efforts have been made to extend SQL, such as the introduction of the Data Cube [Gra+96] and Skyline [BKS01] operators, these works primarily focus on enhancing SQL with additional functionality. Despite these developments, the limitation of single-table queries has persisted since SQL's invention.

In Chapter 4, we overcome this single-table limitation of traditional SQL by introducing a new keyword, RESULTDB, that allows us to return multiple individual result sets, i.e., to essentially compute a subdatabase as a result. Specifically, instead of returning a single table, we return the set of tables that participate in a query, each containing only the tuples that contribute to the traditional query result. There a quite a few application areas and use cases where producing such individual result sets can be beneficial. In particular, it reduces data transfer by minimizing redundancies, and preserves relational information by keeping data retrieval and data transformation separate. Furthermore, it may even improve performance during query execution by avoiding the generation of large intermediate results. To compute a result subdatabase, we present several rewrite algorithms allowing us to use any SQL-92-compliant closed-source database system, and we propose an efficient native algorithm allowing us to extend query optimizers directly inside a database system.

1.4 Contributions

In this section, we summarize and list the contributions of the projects presented in the following chapters of this thesis.

Deep Query Optimization

Chapter 2 is based on the publication:

> Jens Dittrich and Joris Nix. "The Case for Deep Query Optimisation". *10th Conference on Innovative Data Systems Research (CIDR)*, 2020. [DN20]

In this work, we make the following contributions:

1. Concept of Deep Query Optimization. Current state-of-the-art query optimization usually translates a logical operator in a single step into a physical operator, missing interesting optimization potential. The underlying idea of DQO is to break up the abstraction of these holistic operators into more fine-granular subcomponents. Initially, we argue that the transformation from logical to physical plan represents more of a physiological design continuum. Afterward, we discuss the impact of local versus global effects, i.e., how the choice of specific subcomponents can influence the optimization process. Furthermore, we identify a set of meta-relational properties that present promising avenues in the context of DQO.

2. Materialized Algorithmic Views. We introduce the concept of materialized algorithmic views (MAVs) akin to traditional materialized views. The core concept behind MAVs is to materialize parts of a query execution plan at varying levels of granularity, ranging from individual components of traditional operators, to entire operators, and even to larger subplans. Furthermore, we introduce the algorithmic view selection problem, i.e., the problem of matching precomputed MAVs to suitable subplans during query optimization.

3. Experimental Evaluation. We showcase the impact of deep query optimization based on five different implementation variants of the traditional grouping operator. Furthermore, we highlight the potential improvements of DQO over traditional query optimization in terms of estimated query plan costs, particularly when taking meta-relational properties into account.

Genetic Generic Generation of Index Structures

Chapter 3 is based on the publication:

> Jens Dittrich, Joris Nix, and Christian Schön. "The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures". *Proc. VLDB Endow.*, 2021. [DNS21]

In this work, we make the following contributions:

1. Generic Indexing Framework. We introduce a generic index structure framework that makes a clear distinction between a logical and physical index akin to the split into logical and physical operators. A logical index handles the data partitioning, whereas a physical index defines the concrete data layout and search method inside each node.

2. Genetic Algorithm. We formulate index construction as an optimization problem that we solve using genetic programming. As part of the genetic algorithm, we present the core mechanism for automatically generating index structures, along with the mutations that define potential changes to both logical and physical indexes, and the fitness function used to evaluate the performance of physical index structures.

3. Experimental Evaluation. We conduct a comprehensive evaluation of our generic framework and demonstrate the ability of our genetic algorithm to rediscover existing, previously handcrafted indexes. Furthermore, we investigate the potential performance gain of an optimized GENE index compared to state-of-the-art index structures. Our results show that depending on the underlying dataset, our optimized index is able to match and even outperform the traditional indexes.

Personal Contributions and Involvement. My co-author Christian Schön primarily focused on the genetic algorithm, while I concentrated on the generic indexing framework. As a result, he was responsible for implementing the genetic algorithm and took charge of Section 3.4, Section 3.6.1, and Section 3.6.2, which are included in this thesis for completeness but not claimed as my contribution. However, for Section 3.4, I significantly contributed to the development of key components, including the initial population generation in Section 3.4.2, the specific mutations applied to logical and physical index structures in Section 3.4.3, and the fitness function used to evaluate the performance of the generated indexes in Section 3.4.4.

Extending SQL to Return a Subdatabase

Chapter 4 is based on the publication:

> Joris Nix and Jens Dittrich. "Extending SQL to Return a Subdatabase". *Proc. ACM Manag. Data*, 2025. [ND25]

In this work, we make the following contributions:

1. SQL Extension. We propose a backward-compatible SQL extension, `SELECT RESULTDB`, that returns a subdatabase instead of a single-table result. This subdatabase contains the tables that participate in the SQL query, each restricted to those tuples that contribute to the overall query result. We show that our approach has far-reaching consequences and a wide range of use cases, the most notable being that we avoid the denormalization redundancy typically introduced by joins.

2. Rewrite Methods. We present four SQL-based rewrite methods that can be used in any DBMS including closed-source systems to support our extension. These rewrite methods explore different semi-join and materialization strategies, showcasing their strengths and weaknesses as well as the trade-offs between them.

3. Native Algorithm. We present an efficient native algorithm to extend query optimizers in open-source database systems to compute a result subdatabase. This algorithm solves acyclic queries using semi-joins based on the well-known Yannakakis algorithm. For cyclic queries, we propose a new technique, called *folding*, to transform a cyclic query into an acyclic query. In this context, we introduce the *Tree Folding Enumeration Problem*, a new optimization problem on how to choose folds to transform cyclic queries to acyclic queries. We implement our algorithm in mu*t*able, a main memory DBMS with a state-of-the-art compiling query execution engine.

4. Experimental Evaluation. We conduct an extensive experimental study comparing traditional query processing producing a single-table result with our approaches producing a result subdatabase. In particular, we evaluate the performance of our rewrite methods and investigate the overhead they introduce over single-table processing. Additionally, we compare our native algorithm with traditional query processing and analyze the costs associated with joining a result subdatabase to obtain the single-table result. Our results show that multiple individual result sets significantly reduce the result set size, with our methods adding minimal overhead to the query execution time and, in some cases, even outperforming single-table execution.

Chapter 2

Deep Query Optimization

2.1 Introduction

Query Optimization (QO) is at the heart of any query engine. The core task of QO is to find an efficient plan under a given cost model. A major difficulty of this task is to compute not only a *logical plan* (an extended relational algebra expression, a DAG of logical operators) specifying which relations to join in which order but additionally a *physical plan* (a DAG of physical operators) that specifies which access methods (e.g., unclustered B-tree vs scan) and algorithms to use (e.g., sort-merge vs hash join). The physical plan can then either be interpreted or compiled.

A major problem of this approach is the hidden legacy of relational algebra. For example, at some point in QO, a logical join, i.e., $\bowtie(R, S)$, is translated to a physical join, e.g., `SortMergeJoin`(R, S) or `HashJoin`(R, S). In other words, the abstraction used during query optimization is the following: a physical operator in relational algebra receives one or two inputs from outside, does some well-defined processing *inside*, and produces one output dataset to the outside. In summary, a physical plan is a DAG-structured 'algorithmic recipe' where the nodes are physical operators and the edges symbolize producer-consumer relationships.

There has been considerable work on examining the pipelining aspects of this legacy, i.e., how to effectively implement the producer-consumer relationships from good old volcano-style ONC-iterators [Gra93], via vectorization [Bon+05] to breaking the boundaries of physical operators to run plans until the next pipeline breaker [Neu11], to include parallelism [Lei+14], and combinations thereof [Ker+18].

However, these works neglect that the algorithms used to implement an operator *can be considered a query plan in itself*. For instance, consider the physical grouping operator[1]. From a 10,000 feet perspective, it can be implemented either using a sort-based or a hash-based algorithm. In hash-based grouping, as shown in Algorithm 1, we initialize an empty hash table (line 2). Then we insert each tuple from the input into that hash table using the grouping key as the key to the hash table and the set of tuples having that key as the value to the hash table (lines 4–9). Afterward, for each existing key in the hash table, we compute the aggregate on the set of tuples pointed to (lines 11–13).

Algorithm 1 Textbook-style pseudocode for hash-based grouping.

```
1:  function HashBasedGrouping(Relation R, groupingKey)
2:      HashMap hm
3:      Result result = {}
4:      for each r ∈ R do                    ▷insert all tuples from input R into HashMap hm
5:          if r.groupingKey ∈ hm then
6:              hm.probe(r.groupingKey) ∪ = {}
7:          else
8:              hm.insert(r.groupingKey, {r})
9:          end if
10:     end for
11:     for each key ∈ hm.keySet() do        ▷build aggregates for each existing key in HashMap hm
12:         result ∪ = aggregate(hm.probe(key))
13:     end for
14:     return result
15: end function
```

This algorithm can be found in almost all database textbooks, lectures, and even conference talks. The problem with this algorithm is that it *implies a couple of algorithmic and physical design decisions*: (1) As an internal index structure a hash table is used, but which one exactly? As already observed in [Ric+15] a hash table has many different dimensions which influence performance dramatically. (2) The insert operations to the hash table are implicitly assumed to occur in serial. (3) The aggregation operations in the hash table are implicitly assumed to occur in serial, i.e., group-wise. (4) The 'Relation R' parameter in the function signature implies that the entire result set is passed to the algorithm fully materialized. Likewise, the final aggregates are collected in a result set before passing them outside which again implies that the result set is materialized. In summary, this induces two unnecessary pipeline breakers. (5) The two phases in the algorithm (first load the hash table, then compute the aggregates) forbids any

[1]We focus on grouping here as joins have very similar algorithmic issues and solutions. This becomes clear if you consider that a join is merely a co-group-operation with exactly two inputs followed by an aggregation on each co-group. Vice versa a grouping operation is merely a co-group operation with a single input.

kind of non-blocking behavior, e.g., like in any kind of online aggregation algorithm [Hel+97; Dit+02].

A much better description of this algorithm is shown in Algorithm 2.

Algorithm 2 Partition-based grouping. Here each 'line of code' is written as a producer-consumer pattern, i.e., a line of code consumes some input and creates one or multiple producers. This does not make *any* algorithmic decision whatsoever on how this producer-consumer pattern will be implemented physically. $\rightrightarrows$ denotes that an operation provides a bundle of independent producers.

```
1: function PartitionBasedGrouping(Producer R, Consumer R', groupingKey)
2:     R ⟶ partitionBy(groupingKey) ⇉ R_partitions        ▷partition the input
3:     R_partitions ⇉ aggregate(...) ⇉ R'                 ▷aggregate each partition
4: end function
```

The two lines of code basically say: in line 2, we will partition the data produced by R into a bundle of independent producers. If the input produces 42 different groups, `partitionBy` creates 42 different producers. Semantically each producer will deliver the tuples belonging to its group. Notice that there is no need here to shoehorn the result into one relation as in relational algebra or SQL. In line 3, we specify that a bundle of independent producers is aggregated with the same aggregation function, but possibly independently.

What if we depict this 'code', i.e., the insides of the logical operator, as a query plan itself? This is shown in Figure 2.1. Figure 2.1 (a) shows logical grouping as found in (extended) relational algebra. If we open up that box, i.e., if we *unnest* the operator into a more fine-granular query plan, we obtain Figure 2.1 (b). The latter corresponds to the pseudocode given in Algorithm 2. The more we unnest, we *increase the physicality* of the query plan as for each unnest we have to make a decision on *how exactly* to implement a certain 'bubble'. Only after some recursive unnesting, we eventually obtain Figure 2.1 (d) which corresponds to the pseudocode of hash-based grouping shown in Algorithm 1. In other words, hash-based grouping is just one of many special cases in a partition-based grouping algorithm. Figure 2.1 (e) shows another unnest using static perfect hashing (SPH, in the experiments we will even use minimal SPH) as well as a parallel load. And from that we could continue further...

In Figure 2.1, the arrows denote a specific path we followed at each unboxing step. In turn, at each unnest we discard several options. In this example we unnest four times. However, this figure just visualizes the principle. We do not imply that it is exactly four unnest operations to get from a logical operator to a 'physical' plan. However, what *we do imply is that the current state-of-the-art to translate from one extreme (a logical operator) in a single step to another extreme*

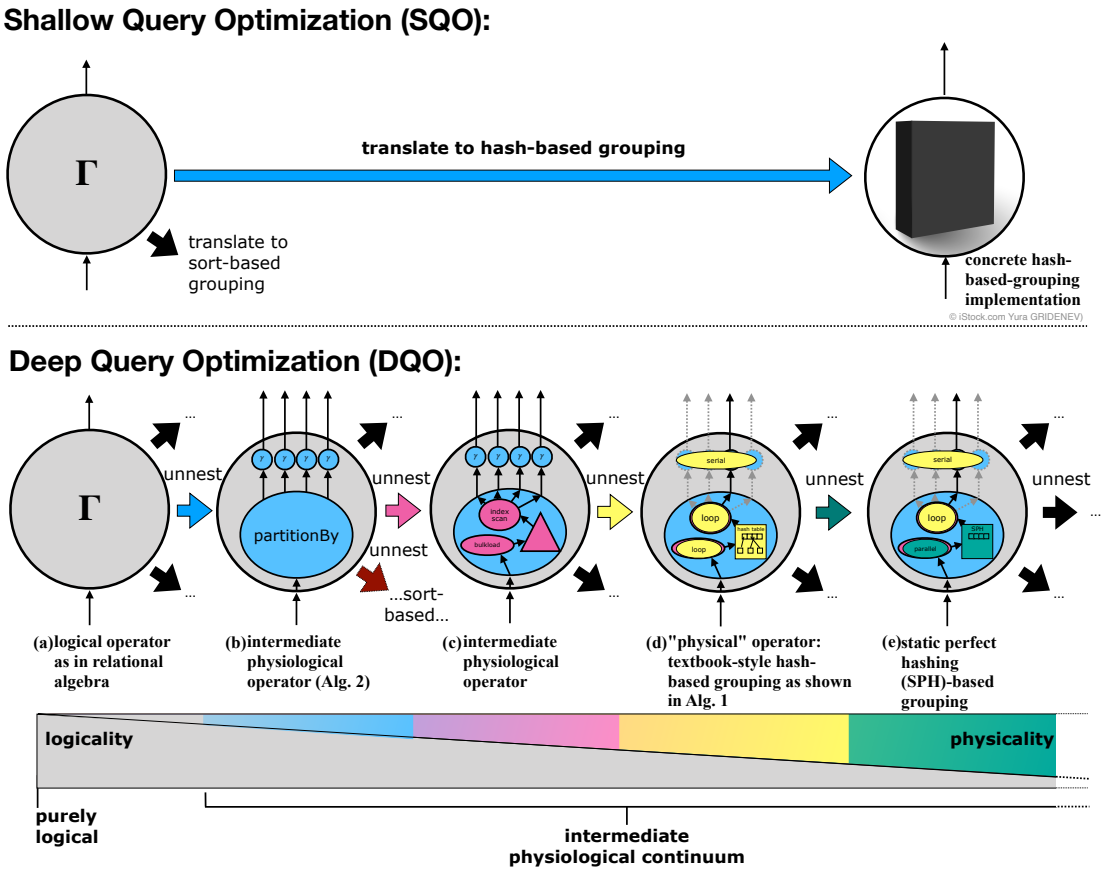


Figure 2.1: Standard ('shallow') query optimization (SQO) vs deep query optimization (DQO). Top: Different unnesting steps of the logical grouping operator: each color depicts yet another hidden nested query plan. Observe that every time we unnest, we make certain algorithmic decisions. Thus, *each unnest operation increases the physicality of the plan.* For instance, if we decide to implement `partitionBy` using any type of index, that decision will exclude other options like sort-based grouping. In turn, if we decide to implement that index using a hash table, we exclude other types of indexes. Notice that naming a plan 'physical' is actually misleading, as there is almost always yet another translation step underneath. For instance, the plan in (e) increases the physicality by choosing one particular type of hash table, hash function, etc. (SPH in this case). In further unnesting steps (not shown), code compilation by the database system and/or compiler will add more steps. Finally, compiler and hardware will make more decisions, e.g., reordering of code and memory accesses. Probably, only the final actions performed on the hardware (the *physics*) should be called a truly *physical plan.*
Bottom: the design space continuum starting on the left from purely logical to plans increasing the physicality of the plan. So far, we have mostly ignored the intermediate physiological plans in that middle-ground: that is what we consider in Deep Query Optimization (DQO).

(a blackbox 'physical operator') misses several interesting optimization opportunities (cf. Shallow Query Optimization in Figure 2.1).

This paper is structured as follows: in Section 2.2, we introduce Deep Query Optimization (DQO). In Section 2.3, we introduce Materialized Algorithmic Views (MAVs) and the Algorithmic View Selection Problem (AVSP). Section 2.4 discusses how to integrate DQO into existing systems. Section 2.5 presents early experimental results of our idea. Section 2.6 contrasts DQO to related work. Finally, Section 2.7 presents a research agenda.

2.2 Deep Query Optimization

The core idea of **Deep Query Optimization (DQO)** is the following: rather than computing a plan using only coarse-granular logical and/or 'physical' operators (as done in Shallow Query Optimization, SQO), in DQO we consider more fine-granular components in the optimization process.

As already shown in the previous sections, when inspecting a single group-by operator, there are many different hidden levels of nested query plans. Table 2.1 presents another view on these granularity levels using an example from biology, a living cell, as an analogy. A living cell is composed of organelles which consist of (macro-)molecules which consist of atoms. Using the living cell analogy, we can phrase the key idea of DQO as follows: *extend SQO to **also** assemble organelles and macro-molecules from molecules rather than only living cells from organelles.* Table 2.2 contrasts SQO, and machine-'learned' techniques [Wan+16] with DQO.

2.2.1 Local vs Global Effects of Deep Query Optimization

DQO can be considered *local* if the types of subcomponents used to assemble a specific granule do not have any effect on the context of that granule. In contrast, DQO must be considered *global* (or at least non-local) if the types of subcomponents used to assemble a specific granule do (or even may) have an effect on the context of that granule.

For instance, assume we want to find the optimal index to be used in hash-based grouping. Let us take a look at two major options for the indexes to use here:

1. We use an out-of-the-box hash-table. Then, the output of the operator will not be sorted. Technically, it is often sorted in the order as the groups appear in the hash table. That

Table 2.1: Granularity concepts in biology vs their counterparts in query optimization, their typical sizes, and example instances. The main difference of DQO over SQO is that we push the frontier of what can be optimized further down into 'physical' operators. Hence, a physical operator is not anymore a given thing optimized manually by some developer, but optimized and synthesized by the system, either at query time or beforehand.

Granularity Concept		**Typical size (LOC)**	**Optimized and synthesized how?**	
in Biology	**in Query Optimization**		**Shallow Query Optimization**	**Deep Query Optimization**
living cell	'physical' query plan	~10000	query optimizer	
organelle	'physical' operator	~1000	developer	query optimizer
macro-molecule	type of index structure (hash vs tree), scan method, high-level bulk loading, and probing algorithm	~100	developer	query optimizer
molecule	any subcomponent of an index, e.g., a node or leaf type, hash function used, particular probing implementation, low-level cache & SIMD tricks	~10	developer	
atom	assignment, loop initialization, arithmetic operation, matrix operation	~1	compiler	

order depends heavily on the hash function used. If we do not know exactly which order is produced by a blackbox hash table, we have to assume that the data is unordered to be on the safe side.

2. We use a static perfect hash-function (SPH). SPH can simply be an array of groups of tuples (or running aggregates in the case of distributive and/or decomposable aggregation functions). The grouping key then serves as the index into that array. Here, the linear array slot computation works like a perfect hash function. If all array slots are used, the SPH is even minimal. This is only applicable if the key domain of the grouping key is (relatively) dense. This situation is not as rare as one might think. For instance, the keys of a dictionary-compressed column are a natural candidate for this and can directly be used for SPH. Like in a hash-table, the grouped output will be sorted according to the order in the underlying array. Notice that in both options, the order on the probe input is preserved.

In summary, any optimization step in DQO can be considered local, if the subplan produced at that step has the same properties as any other plan at that granule. In contrast, if for a given granule different subplans have different properties, its optimization effects may be considered global.

Table 2.2: SQO, 'learned', and DQO in comparison: both SQO and DQO assume that we rewrite and enumerate possible plans. However, in contrast to SQO, in DQO we assume components at a much finer granule (as shown in Table 2.1). In contrast to 'learned' approaches, in DQO, we do not learn those functions from scratch but assume fine-granular building blocks to be available. Similar to SQO, we assume that rules are available that define how to combine those building blocks, i.e., we see this primarily as an optimization rather than a machine learning problem. Another important difference and advantage over several 'learned' approaches and a similarity to SQO is that we do not need extra mechanisms to correct errors due to the approximative nature of most ML-methods.

(a) SQO

function to optimize	who?	based on	how?
index_node()	human	code lines	trial & error
index()	human	index_node()	trial & error
hash_join()	human	index()	trial & error
query_plan()	machine	hash_join(), index(), etc.	rules & enumeration
database_system()	human	query_plan()	trial & error

(b) 'learned'

function to optimize	who?	based on	how?
index_node()	NA	NA	NA
index()	machine	nothing	ML + correction code
hash_join()	machine	nothing	ML + correction code
query_plan()	machine	nothing	ML + correction code
database_system()	machine	nothing	ML + correction code

(c) DQO

function to optimize	who?	based on	how?
index_node()	human	code lines	trial & error
index()	machine	index_node()	rules & enumeration
hash_join()	machine	index()	rules & enumeration
query_plan()	machine	hash_join(), index(), etc.	rules & enumeration
database_system()	machine	query_plan()	rules & enumeration

Table 2.3: Meta-relational properties for deep query optimization.

physical	structural	statistical
data layout	partitioning	distribution
compression	grouping	cardinality
location	sorting	uniqueness
hardware	correlation	density
clustered		# rows
		# columns
		# NULL-values
		min, max, avg, etc.

2.2.2 Meta-Relational Plan Properties

Throughout the years, database research literature proposed concepts like *interesting orders* in sort-based operators or physical properties like *compressed* or *partitioned* data [Gra+93]. However, in DQO we do not limit ourselves to this narrow set of plan properties and expand the *property space*. Therefore, we introduce the concept of *meta-relational properties*, i.e., properties that go beyond a pure relational model and reasoning about a query plan. Table 2.3 gives an overview of existing meta-relational properties including a first attempt to categorize them into various subcategories. Concepts like *interesting orders* are included as a special-case.

Physical meta-relational properties are those properties that are abstracted away by physical data independence, e.g., any kind of data layout, physical location, and/or internal representation of the data.

Structural meta-relational properties are related to ordering and/or partitioning of the data, e.g., *sorting* is referred to in database literature as *interesting order*. As these properties do not necessarily have to be represented physically they belong to a separate category. Still they *may additionally* also be reflected physically in which case they belong to the category *Physical meta-relational properties*.

Statistical meta-relational properties are statistical properties summarizing the data using statistical moments or any other useful (yet lossy) approximation.

As optimization in DQO happens on small granules, we have to identify the right set of meta-relational properties that allow for a deeper optimization on the corresponding granule, e.g., on an operator level or on the structure of the query plan. In addition, we have to consider that many of these meta-relational plan properties may have a non-local effect.

In a first shot, the properties with a non-global effect can be considered and handled very similarly to how interesting orders are handled in dynamic programming. If any subcomponent in DQO produces an output with such a property, we *must not* discard that information.

In the experiments in Section 2.5, we show how we modified dynamic programming to factor in the following meta-relational plan property: dense vs sparse and how that makes a difference in optimization and runtime performance. Examining the other properties in more detail is an avenue for future work.

2.3 Materialized Algorithmic View Selection

There is another interesting similarity of DQO to SQO when recalling materialized views (MVs, precomputed query results) and also prepared statements (preoptimized queries).

2.3.1 Materialized Algorithmic Views (MAVs)

In DQO, in particular for local effects but also for the non-local ones, it makes sense *to precompute certain granules offline (before a query comes in).* We coin these precomputed components **Materialized Algorithmic Views (MAV)**. Just like MVs prematerialize query results, MAVs prematerialize (i.e., optimize) parts of a deep query plan. An MAV may be a component of a physical 'operator', entire physical 'operators' or larger units, i.e., physical subplans.

In other words, a deep query plan as shown for instance in Figure 2.1 (b–e) may conceptually represent (parts of) an algorithm, an index node/leaf, an index, a physical operator like a very specific hash join algorithm implementation or even larger parts of a physical plan. However, as the search space for this plan is potentially large, we believe that MAVs may help in that they materialize certain subtrees of that plan into preoptimized units (again: just as prematerialized views are used as building blocks in shallow query optimization).

MAVs can be precomputed for any granularity level not only within the traditional boundaries of an operator. Like that MAVs can be used as building blocks for DQO at query time to speed-up plan enumeration.

2.3.2 When to Materialize MAVs?

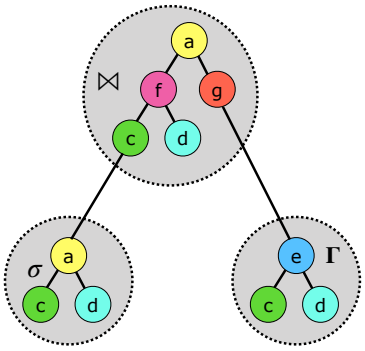
An interesting question is when to compute an MAV. Again, the duality to materialized view selection helps a lot here: MVs work well if the underlying data does not change too often, and we do not spend too much work updating those MVs. In addition, an MV should be useful in queries in the sense that it is used (matched) regularly and decreases the overall query runtime (including query optimization time). At the same time the MV should be storage efficient. For MVs these trade-offs naturally lead to a cost/benefit-calculation to decide which MVs to keep. We believe that very similar techniques will strike for MAVs. We envision a usefulness-ranking which quickly identifies components of a deep query plan that can be reused over and over again. This is up to future work to show.

There is a natural trade-off here: how much time do I want to spend on DQO offline vs at query time?

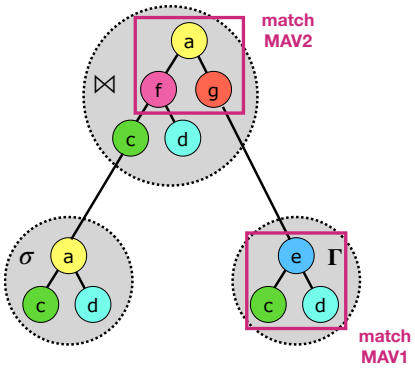
2.3.3 The Algorithmic View Selection Problem (AVSP)

At query time we have to face the challenge of matching those precomputed MAVs to suitable subplans. Again, inspired by the materialized view selection problem [Bar+97], we coin this the **Algorithmic View Selection Problem (AVSP)**. Like with MVs, there is no need in AVSP to make any manual decision about which granules to precompute. This decision can be automated. This is simply adding a new AVSP dimension to the physical design problem.

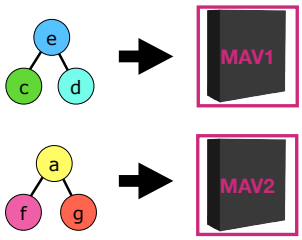
An example for algorithmic view matching is shown in Figure 2.2. Figure 2.2a shows an example of a fine-granular (deep) query plan that we need to optimize. In Figure 2.2b, we match the available MAVs as listed in Figure 2.2c against the plan in Figure 2.2a. In Figure 2.2b, we visualize two matches: both MAV1 and MAV2 can be matched against suitable subplans. In Figure 2.2d, we replace these matches with their corresponding preoptimized MAVs. Only after this matching step, plan enumeration needs to be invoked. Hence, the search space is diminished – just like in materialized view selection. Figure 2.2 also shows the relationship to MVs: an MV for a given subplan precomputes a query result. This only works if that subplan corresponds to a subtree of the plan, i.e., it does not depend on additional computations that need to be performed at query time. In contrast, an MAV materializes an algorithm. Still that algorithm may rely on inputs of other algorithms which are only optimized at runtime. In that sense, an MAV is also related to prepared statements which partially preoptimize a plan but still allows the database engine to insert variable assignments. In that sense, in order to map these ideas



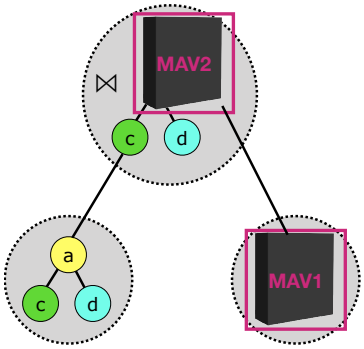
(a) Possible deep query plan.



(b) Algorithmic view matching.



(c) Available materialized algorithmic views.



(d) Deep query plan using MAV1 and MAV2.

Figure 2.2: Materialized Algorithmic Views (MAVs) in deep query planning and the Algorithmic View Selection Problem (AVSP).

back to the traditional database universe of logical and physical operators, an MAV that happens (incidentally) to prematerialize the internals of what we used to call an operator could be coined a **prepared operator**. For instance, in Figure 2.2d, MAV1 materializes the contents of a grouping operator. Again, MAVs do not have to match these traditional operator boundaries. In Figure 2.2d, the join operator is an example for a case where only parts of the 'operator' are replaced by MAV2.

Another relationship worth mentioning is plan caching: some database engines already keep subplans of previously optimized queries and reuse them. However, in contrast to DQO, these techniques work on the coarse-granular operator abstraction.

Also notice that there is **no need to fully preoptimize** an MAV. A 'materialization' simply means that we decrease the logicality and increase the physicality of this subplan in whatever way. Recall our discussion for Figure 2.1 which displays this effect as well: prematerializing an MAV simply means that we push a (sub-)plan further to the right.

MAVs and the AVSP trigger a couple of interesting research challenges and directions, in particular when applied to indexing. We will discuss them in Section 2.7.

2.3.4 But the Search Space Is Exponential!

It is obvious that DQO increases the search space of an already (but only possibly!) exponential search space even more. However, in SQO, it is well-known that the shape of the join graph may reduce the complexity of the search space dramatically, e.g., the search space is only polynomial for linear plans, see [Moe] for an overview. We believe that similar effects are present in DQO, i.e., the shape of the graph representing how components may be composed may reduce the search space a lot. This is up to future work to explore.

We also believe that similarly to how MVs helped to reduce the search space of a shallow query plan at query time MAVs help to reduce the search space over algorithmic components. So, in summary, the explosion of the search space may seem frightening in the beginning, however, that is not a reason to ignore DQO but rather a call to research.

2.4 System Integration

Our experiments in Section 2.5 consider the effects of DQO in an isolated way by examining a concrete example of different grouping implementations. The intention is to make the case for a deeper level of query optimization without having a complete system behind it. Our long term vision is to integrate this core idea into an existing database management system. Therefore, in this section we will discuss what the integration of DQO into an existing DBMS could look like. For this, we will briefly touch on the core components of query optimization like plan representation, plan space enumeration, cost models, and statistics. Ultimately, we are planning to integrate DQO into mu*t*able [HD23c], a query execution engine currently developed at Saarland University.

Plan representation. The most fundamental design decision, on which the other parts of the query optimization engine are building up, is the internal representation of a query. Since we aim to break up the current structure of operators and to optimize on a deeper level, a new physiological representation of queries might be necessary (analogue to one of the intermediate physiological operator representations in Figure 2.1). For instance, instead of having a logical grouping operator, we could introduce multiple physiological grouping operators which already decide on some implementation details like index-based or order-based grouping (see our algorithms in Section 2.5).

Plan space enumeration. The deeper we go into an operator, the more the search space and the complexity increases. Holistically enumerating the whole query plan including all possible subcomponents of every operator could easily exceed what is computationally possible right now. To overcome this, one possibility might be to do independent plan enumeration inside each operator using the structural components as the building blocks. With that, we can locally come up with the best possible operator given the underlying dataset and workload. Once we have the optimal physical operators, we can use classical plan enumeration algorithms like dynamic programming to find the final query execution plan. Of course, this probably does not find the overall best execution plan because we are missing global optimization effects. However, this could already improve performance significantly. To further prune the search space, we can fall back on materialized algorithmic views. We could essentially create a library of many different algorithmic implementations from which we could choose.

Cost models. Which cost models we are effectively going to use depends on the type of plan enumeration and which statistics are available to use. In case we want to enumerate the

structure of operators, we might need separate cost models for the different kinds of operators. In addition, we expect to come across new rules which we can incorporate into the optimization process. For example, using distribution information by knowing that the underlying data is sparse, we might discard all possible plans using static perfect hash-based algorithms because the performance will definitely be worse than classical hash-based implementations (see our experimental results in Section 2.5).

Statistics. To enable a fine-granular optimization of our query operators and with it, of the complete query plan, we need precise statistics about the underlying data and the workload. For queries processing large amounts of data, we assume the overhead for collecting additional statistics to be most likely negligible. Which statistics we need depends on the optimization decisions we want to make. For example, information about the density of the data allows the use of specialized algorithms like static perfect hash-based grouping.

As a closing thought, we expect that many design decisions will come naturally as we are implementing the system.

2.5 Experiments

DQO can be applied in all parts of query optimization that include algorithmic design decisions. In this section, we demonstrate in a small domain that DQO can have a significant impact on query execution.

2.5.1 Setup and Methodology

All experiments were conducted on a Linux machine with an AMD Ryzen Threadripper 1900X 8-Core processor with 32 GiB memory. All algorithms are implemented single-threaded in C++ and compiled with Clang 8.0.1, -O3.

We consider five different implementation variants of grouping[2]. Each implementation computes the aggregates `COUNT` and `SUM` on the fly and stores a mapping from grouping key to aggregate data inside an array. In our experiments, we assume the number of distinct values to be known. However, this assumption is not strictly required and can be relaxed by using dynamic data structures, techniques like rehashing, or arrays that cover the whole domain in

[2]The implementation of our algorithms is available on GitHub https://github.com/BigDataAnalyticsGroup/Deep-Query-Optimization

case it is relatively small and dense.

Hash-based Grouping (HG). We use `std::unordered_map` as the underlying hash table and the Murmur3 finalizer as hash function. Every input element is inserted individually into the hash table.

Static Perfect Hash-based Grouping (SPHG). We use the grouping key as offset into the array storing the groups, acting as a static and perfect hash function.

Order-based Grouping (OG). This implementation requires the input data to be partitioned by the grouping key. We iterate sequentially over the input data, create a group for the very first occurrence of a grouping key, and insert this group at the first empty slot in the array. As long as the grouping key remains the same, the corresponding aggregates are updated.

Sort & Order-based Grouping (SOG). We do not require that the input data is partitioned by the grouping key. Therefore, we first sort the data then we apply OG.

Binary Search-based Grouping (BSG). We store a mapping from grouping key to aggregate data inside a sorted array. This allows us to perform binary search to look up a group by its key.

The datasets consist of 100 million 4 byte unsigned integer values representing the grouping key. Each dataset is uniformly distributed and has two properties, sortedness and density. Taking all combination of those properties, we end up with four different datasets.

2.5.2 Performance of Physical Grouping Implementations

This section compares the execution time of the algorithms across the four datasets. Figure 2.3 shows one plot for each of the four different datasets. Each plot depicts the execution time in milliseconds for an increasing number of groups.

Sorted & Dense. For this combination, OG and SPHG exhibit the best performance at roughly 250 ms, being more than four times faster than HG. SOG performs even worse because the already sorted input data is again (unnecessary) sorted, incurring considerable overhead. Furthermore, the execution time of each algorithm is mostly independent of the number of groups.

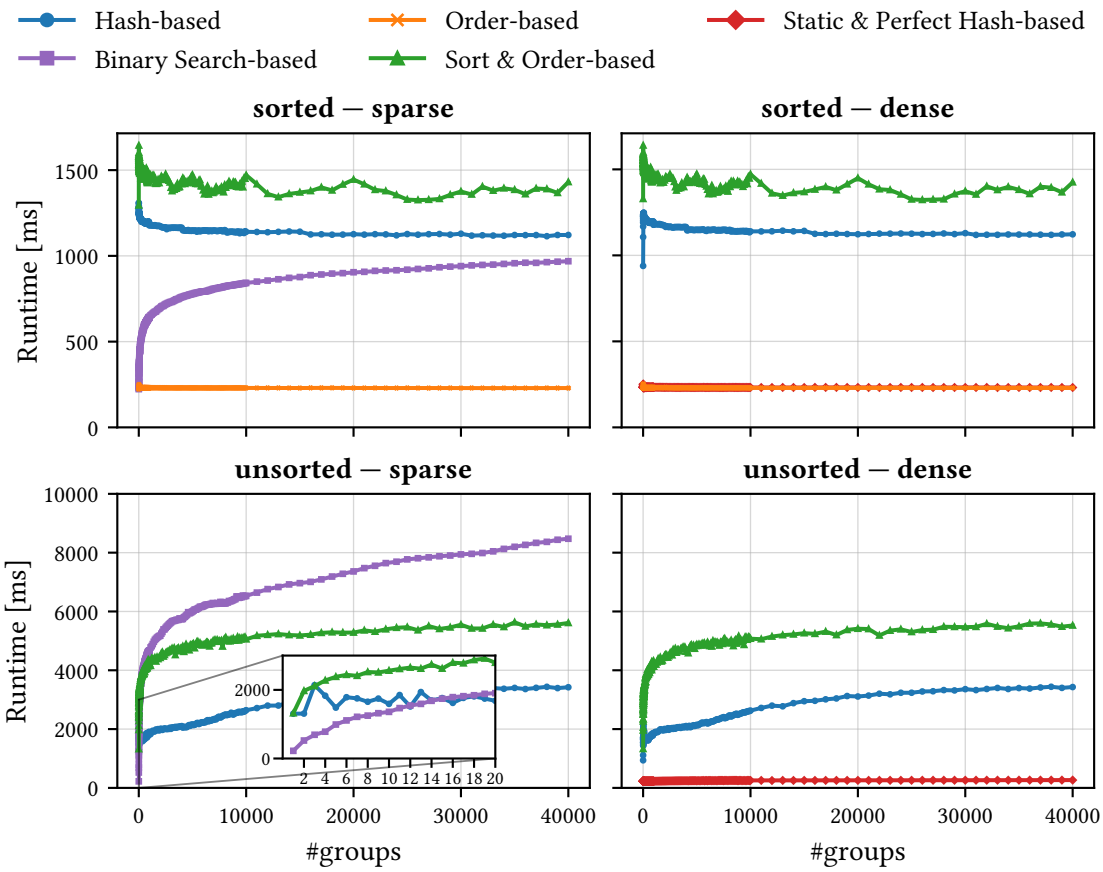


Figure 2.3: Grouping performance of four different grouping algorithms on four different input datasets.

Sorted & Sparse. This experiment shows that the execution time of HG, OG, and SOG is essentially independent of the data density. OG again performs best at around 250 ms. Since we have a sparse data domain, we cannot use SPHG. Instead, we use BSG, which incurs logarithmic costs relative to the number of groups. However, no algorithm comes close to the performance of OG.

Unsorted & Dense. In this case, SPHG is the best performing algorithm at a constant execution time of roughly 250 ms, being unaffected by the sortedness of the data. The execution time of HG grows with an increasing number of groups because of caching effects. For less than about 500 groups, SOG displays a steep rise in execution time, and afterward merely a modest increase.

Unsorted & Sparse. For this setting, we can neither exploit the sortedness nor the density of the data. Without these properties, HG is superior in a wide range of number of groups. However, for up to 14 groups (see zoom-in in the respective plot), BSG outperforms HG. This opens up another optimization dimension in which the number of distinct values should be considered.

In summary, this experiment shows that not only sortedness determines the fastest algorithm but also other properties like in this case density.

2.5.3 DQO-Enabled Dynamic Programming

This section shows how classical dynamic programming can lead to better query execution plans when extending it with DQO. Consider the query depicted in Listing 2.1.

Listing 2.1 Example query joining two relations R and S followed by a grouping operation.

```
1 SELECT R.A, COUNT(*)
2 FROM R JOIN S ON R.ID=S.R_ID
3 GROUP BY R.A;
```

For the physical implementations of the joins, we assume the algorithmic counterparts of our grouping implementations. The corresponding cost models are shown in Table 2.4. We assume the size of tables R and S to be 40, 000 and 90, 000, respectively. Further, we assume the output-size of the join to be 90, 000 because of the foreign-key constraint `R.ID=S.R_ID` and the output-size to be 20, 000. Figure 2.4 shows the logical plan of the query in Listing 2.1.

Table 2.4: Cost models for grouping and join algorithms.

	Grouping	**Join**
hash-based	$HG(R) = 4 \cdot \|R\|$	$HJ(R,S) = 4 \cdot (\|R\| + \|S\|)$
order-based	$OG(R) = \|R\|$	$OJ(R,S) = \|R\| + \|S\|$
sort & order-based	$SOG(R) = \|R\| \cdot log_2(\|R\|) + \|R\|$	$SOJ(R,S) = \|R\| \cdot log_2(\|R\|) + \|S\| \cdot log_2(\|S\|) + \|R\| + \|S\|$
static perfect hash-based	$SPHG(R) = \|R\|$	$SPHJ(R,S) = \|R\| + \|S\|$
binary search-based	$BSG(R) = \|R\| \cdot log_2(\#groups)$	$BSJ(R,S) = \|R\| \cdot log_2(\#groups) + \|S\| \cdot log_2(\#groups)$

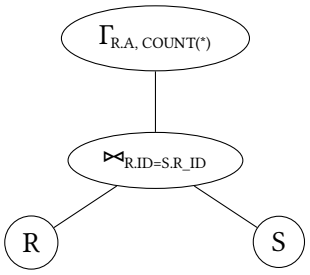


Figure 2.4: Logical plan of example query.

First, we join both input relations R and S and afterward, we compute the grouping result on the output of the join. However, we want to determine which physical implementations of the join and grouping operators lead to the overall best plan. While SQO only considers data sortedness, DQO also considers other DQO plan properties (cf. Section 2.2.2), here: the density of the grouping keys. This allows us to use static perfect hash-based algorithmic designs (SPH) for our join and grouping implementations. For such a small query, a classical dynamic programming algorithm is sufficiently fast. In case the search space gets too big, we believe that we can fall back on established dynamic programming variations [Moe+06; Moe+08].

Figure 2.5 presents the DQO dynamic programming tables for the *dense* property, considering the different combinations of input relations based on sortedness. Similarly, Figure 2.6 displays the corresponding dynamic programming tables for the *sparse* property under the same sortedness variations of input relations. We consider static perfect hash-based algorithms only for dense data and binary search-based algorithms only for sparse data. In addition, we assume sort and order-based algorithms to incur sorting cost only if the input is not sorted. Furthermore, if at least one of the inputs is not sorted, we do not consider order-based algorithms. For each combination, we compared the estimated costs of the best DQO plan to the estimated costs of the best SQO plan. In general, the dynamic programming results show that in a dense domain, using specialized algorithms like SPH leads to plans with the lowest estimated cost.

Table 2.5 shows the improvement factors for the estimated plan costs of DQO over SQO. Compared to DQO, the only difference is that SQO does not consider data density as a meta-relational property. Since SPH can only be used in a dense domain, for sparse data DQO generates the same plans as SQO, resulting in no improvement. In case both inputs are sorted, the order-based implementations achieve the cheapest plans regardless of the data density. However, if at least one input is unsorted, DQO generates plans with an improvement factor of up to 4x. In this case, DQO chooses plans that use the SPHJ and SPHG algorithms.

Table 2.5: Improvement factors for the estimated plan costs of DQO over SQO.

		sparse	**dense**
$\mathbf{R_{sorted}}$	$\mathbf{S_{sorted}}$	1x	1x
	$\mathbf{S_{unsorted}}$	1x	4x
$\mathbf{R_{unsorted}}$	$\mathbf{S_{sorted}}$	1x	2.8x
	$\mathbf{S_{unsorted}}$	1x	4x

In summary, the experiments show that depending on the underlying data properties, different physical implementations lead to the cheapest plans and achieve the best execution time. This supports our claim that more fine-granular optimization can lead to better query plans.

Figure 2.5: Dynamic programming tables for *dense* data and different combinations of input relations regarding sortedness.

(a) R_{sorted} & S_{sorted}

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40,000	✓	✓
S	scan(S)	0	90,000	✓	✓
R ⋈ S	HJ(R,S)	520,000	90,000	✓	✓
	OJ(R,S)	130,000		✓	✓
	SOJ(R,S)	130,000		✓	✓
	SPHJ(R,S)	130,000		✓	✓
Γ(R ⋈ S)	HG(OJ(R,S))	490,000	20,000	✓	✗
	OG(OJ(R,S))	220,000		✓	✓
	SOG(OJ(R,S))	220,000		✓	✓
	SPHG(OJ(R,S))	220,000		✓	✓

(b) R_{sorted} & $S_{unsorted}$

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40,000	✓	✓
S	scan(S)	0	90,000	✓	✗
R ⋈ S	HJ(R,S)	520,000	90,000	✓	✗
	SOJ(R,S)	1,611,187		✓	✓
	SPHJ(R,S)	130,000		✓	✗
Γ(R ⋈ S)	HG(SPHJ(R,S))	490,000	20,000	✓	✗
	SPHG(SPHJ(R,S))	220,000		✓	✓
	SOG(SPHJ(R,S))	1,701,187		✓	✓
	OG(SOJ(R,S))	1,701,187		✓	✓
	SOG(SOJ(R,S))	1,701,187		✓	✓

(c) $R_{unsorted}$ & S_{sorted}

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40,000	✓	✗
S	scan(S)	0	90,000	✓	✓
R ⋈ S	HJ(R,S)	520,000	90,000	✓	✓
	SOJ(R,S)	741,508		✓	✓
	SPHJ(R,S)	130,000		✓	✓
Γ(R ⋈ S)	HG(SPHJ(R,S))	490,000	20,000	✓	✗
	OG(SPHJ(R,S))	220,000		✓	✓
	SOG(SPHJ(R,S))	220,000		✓	✓
	SPHG(SPHJ(R,S))	220,000		✓	✓

(d) $R_{unsorted}$ & $S_{unsorted}$

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40,000	✓	✗
S	scan(S)	0	90,000	✓	✗
R ⋈ S	HJ(R,S)	520,000	90,000	✓	✗
	SOJ(R,S)	2,222,696		✓	✓
	SPHJ(R,S)	130,000		✓	✗
Γ(R ⋈ S)	HG(SPHJ(R,S))	490,000	20,000	✓	✗
	OG(SOJ(R,S))	2,312,696		✓	✓
	SOG(SPHJ(R,S))	1,701,187		✓	✓
	SOG(SOJ(R,S))	2,312,696		✓	✓
	SPHG(SPHJ(R,S))	220,000		✓	✓

Figure 2.6: Dynamic programming tables for *sparse* data and different combinations of input relations regarding sortedness.

(a) R_{sorted} & S_{sorted}

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40, 000	✗	✓
S	scan(S)	0	90, 000	✗	✓
R ⋈ S	HJ(R,S)	520, 000	90, 000	✗	✓
	OJ(R,S)	130, 000		✗	✓
	SOJ(R,S)	130, 000		✗	✓
	BSJ(R,S)	1, 987, 403		✗	✓
Γ(R ⋈ S)	HG(OJ(R,S))	490, 000	20, 000	✗	✗
	OG(OJ(R,S))	220, 000		✗	✓
	SOG(OJ(R,S))	220, 000		✗	✓
	BSG(OJ(R,S))	1, 415, 894		✗	✓

(b) R_{sorted} & $S_{unsorted}$

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40, 000	✗	✓
S	scan(S)	0	90, 000	✗	✗
R ⋈ S	HJ(R,S)	520, 000	90, 000	✗	✗
	SOJ(R,S)	1, 611, 187		✗	✓
	BSJ(R,S)	1, 987, 403		✗	✗
Γ(R ⋈ S)	HG(HJ(R,S))	880, 000	20, 000	✗	✗
	OG(SOJ(R,S))	1, 701, 187		✗	✓
	SOG(HJ(R,S))	2, 091, 187		✗	✓
	SOG(SOJ(R,S))	1, 701, 187		✗	✓
	BSG(HJ(R,S))	1, 805, 894		✗	✓

(c) $R_{unsorted}$ & S_{sorted}

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40, 000	✗	✗
S	scan(S)	0	90, 000	✗	✓
R ⋈ S	HJ(R,S)	520, 000	90, 000	✗	✓
	SOJ(R,S)	741, 508		✗	✓
	BSJ(R,S)	1, 987, 403		✗	✓
Γ(R ⋈ S)	HG(HJ(R,S))	880, 000	20, 000	✗	✗
	OG(HJ(R,S))	610, 000		✗	✓
	SOG(HJ(R,S))	610, 000		✗	✓
	BSG(HJ(R,S))	1, 805, 894		✗	✓

(d) $R_{unsorted}$ & $S_{unsorted}$

Subproblem	Possible Plans	Costs	Output Size	Properties	
				Dense	Sorted
R	scan(R)	0	40, 000	✗	✗
S	scan(S)	0	90, 000	✗	✗
R ⋈ S	HJ(R,S)	520, 000	90, 000	✗	✗
	SOJ(R,S)	2, 222, 696		✗	✓
	BSJ(R,S)	1, 987, 403		✗	✗
Γ(R ⋈ S)	HG(SPHJ(R,S))	880, 000	20, 000	✗	✗
	OG(SOJ(R,S))	2, 312, 696		✗	✓
	SOG(HJ(R,S))	2, 091, 187		✗	✓
	SOG(SOJ(R,S))	2, 312, 696		✗	✓
	BSG(HJ(R,S))	1, 805, 894		✗	✓

2.6 Related Work

Notice, that the bulk of related work has already been discussed inline above or will be discussed in the Research Agenda in Section 2.7.

Stratos Idreos' work [Idr+18b; Idr+18a] is the closest work to ours when it comes to physical 'indexing operators', see Section 2.7 (Materialized Algorithmic Index Views) for details.

Christoph Koch's work [Koc13] discusses the problems with missing abstractions in the database community, in particular when communicating our findings to the systems community. The Introduction of that paper is a must-read for every researcher: *"... What is frequently lacking is a conclusive deconstruction of the research contribution [of a paper] into first principles and fundamental patterns from which the contribution is composed. ..."*. In his work, Christoph Koch develops methods for synthesizing and composing algorithms [Koc14]. That work is however much more PL and compiler-oriented than DQO. In particular, we argue that database domain-specific decisions should not be delegated to the PL compiler.

Lohman introduced the concept of *LOw-LEvel Plan OPerators* (LOLEPOPs) [Loh88]. In this work, the author aims to enumerate different query execution plans by using a 'building block' approach to assemble query execution plans in a grammar-like style. Specifically, they propose to compose low-level database operations such as access, join, or sort into higher-level operations. Although this approach resembles the concept of DQO to some extent, the key difference is the level of granularity. LOLEPOPs essentially consider existing relational algebra operators such as joins as low-level operations and expand this set with additional operations such as sorting or shipping. In particular, they do not propose the idea to break up existing relational algebra operators. In contrast, we explicitly advocate for breaking up traditional logical operators and consider more fine-granular subcomponents during the query optimization process. This enables us to construct query plans by optimizing not only the subcomponents within individual logical operators but also across the conventional boundaries of operators.

The work by Kohn et al. [KLN21] is closely related to our work, as it also builds on the idea of assembling complex parts of a query plan from low-level operators, specifically focusing on SQL aggregations. Like DQO, their work is motivated by the observation that traditional relational algebra operators are too coarse-grained for a holistic optimization process. To address this, Kohn et al. propose to break up traditional SQL aggregation operators into a set of low-level operators, enabling the composition of more advanced SQL aggregates. Their experimental evaluation demonstrates that composing SQL aggregation computations from low-level opera-

tors can significantly improve performance compared to monolithic operators. This approach can be seen as an instantiation of a deep query plan specifically for aggregation.

The works by Bandle and Giceva [BG21] and Jungmair and Giceva [JG23] propose the use of declarative sub-operators for universal data processing systems. In both works, the authors advocate for making sub-operators first-class entities within data processing systems, allowing them to function as an intermediate representation or unified abstraction layer, respectively. While the first work presents a more visionary perspective, the second offers a concrete design and implementation based on specific requirements and novel design decisions. Jungmair and Giceva [JG23] establish a unified abstraction layer that enables the specification of complex, custom operators and algorithms. By decoupling the semantics of an operator from its concrete implementation, similar to the separation of logical and physical operators, they are able to reduce complexity and make the implementation of complex operators feasible. Further, the fine-granularity of these sub-operators also facilitates new automatic optimizations and transformations. While our approach specifically breaks up traditional relational algebra operators to unlock further optimization potential, their approach extends beyond query optimization. It focuses on efficiently supporting diverse workloads and reducing the internal complexity of data processing systems by introducing a unified abstraction layer based on declarative sub-operators. In general, their work can be seen as a concrete implementation and realization of the broader vision we advocate in our DQO work.

2.7 Research Agenda

Revisit SQO Algorithms. We envision that many existing SQO algorithms can easily be extended to support DQO. Just like extending SQO to large queries [Neu+18], the challenge will be to extend them to deep queries and find the right sweet spots. Recall that, in the history of SQO, initially only relatively small queries could be optimized, but over time the queries became bigger and bigger. We foresee the same to happen with DQO: over time deeper and deeper queries will become optimizable. As long as optimization time in DQO is an issue, we need MAVs to the rescue.

Physiological Algebra. An interesting research subspace will be to identify the right components to use in DQO, i.e., what are suitable organelles and macro-molecules to consider? We envision that this will lead to a physiological component set akin to relational algebra yet including both logical and physical aspects.

Algorithmic Views Selection. A promising direction is to systematically research and evaluate MAVs and the Algorithmic Views Selection Problem (AVSP). When to materialize which algorithm into an MAV? Beforehand or at query time? What do we possibly gain or lose at query time? These trade-offs have to be explored carefully. And, yes, for sure: these trade-offs are absolutely workload-dependent. For which parts of a query plan should we consider DQO?

Partial Algorithmic Views. Rather than fully materializing parts of a deep query plan into an MAV, or, if we pick the other extreme, not materializing it at all, there is an interesting middle-ground: It makes sense to *partially optimize an MAV offline* and leave some flexibility for DQO at query time. Which portions should be left up for DQO at query time? Again, these trade-offs have to be explored carefully. Actually, there are many interesting lessons here that can be adapted from compiler construction in this space.

Materialized Algorithmic Index Views. An entire interesting research subspace is to apply DQO to indexing. It would be exciting to explore DQO in that context. In database literature, we witness the birth of about a dozen index structures every year. Most indexes are basically composed of substructures (atoms in our analogy), i.e., different nodes and leaf types. Classbook index structures like a B-trees and binary search trees use a tiny set of node and leaf types, other indexes extend that set slightly allowing for more heterogeneous trees [Lei+13].

An extreme version of this is [Idr+18b; Idr+18a]. However, a synthesized data structure (SDS) is simply one special case of what we propose. Basically, in DQO, *a synthesized data structure is one particular type of MAV*. In addition, for an SDS all the optimization happens offline. That is an unnecessary restriction as already outlined in Section 2.3. In DQO we do not need to synthesize the entire index or any other MAV beforehand. Which implies the following:

Runtime-Adaptivity and Reoptimization of MAVs. So far we suggested optimizing deep query plans and *then* execute these plans. As with shallow query plans, the literature on reoptimization (during query time) as well as adaptivity should be revisited in the light of DQO. For instance, in traditional indexing, for each column, the decision whether to create an index is binary. What if we make that decision continuous? Like that different parts of a column are not, slightly, or fully indexed. That is the core idea of adaptive indexing [Ker+05; Sch+13]. An adaptive index has built-in heuristics to make these decisions at runtime based on the incoming queries. And even those heuristics may be meta-adapted [Sch+18]. In the DQO universe a (meta-)adaptive index is simply a partial MAV where some optimization decisions have been delegated to query time and baked into that MAV. This idea should be revisited for all physical components currently used in SQO; not only indexes.

Long term Vision. We are planning to integrate DQO into mutable [HD23c]. mutable is an extensible research database system built at Saarland University. In particular, we want to explore how to make a smooth transition from SQO to DQO and find the sweet spots for any given workload.

2.8 Conclusions

This paper made several contributions: we opened the book for Deep Query Optimization (DQO). We presented the general idea, contrasted it to SQO, and showed the high potential of DQO. In addition, we introduced the concept of Materialized Algorithmic Views and the Algorithmic View Selection Problem. We presented early experimental results with DQO. In addition, we compiled a research agenda.

Chapter 3

Genetic Generic Generation of Index Structures

3.1 Introduction

3.1.1 Problem 1: Indexes Are Considered Monolithic Entities

When we, as database researchers, talk about indexes, we use the term *index* like referring to an entity of its own. But is that the case? Let us look at our good old B-tree: A *B-tree index* consists of two different node types: *inner nodes* and *leaves*. Inner nodes keep pointers to other nodes. The main purpose of an inner node is to route incoming lookups to other nodes. Leaves keep pointers to data pages or the data itself. In addition, a B-tree index algorithmically preserves a couple of invariants, e.g., all paths from the root to a leaf have the same lengths, each node only has one parent node (i.e., nodes are structurally organized into a tree), and so forth. In addition, all nodes keep data in a specific layout (row or column layout, cache-and SIMD-efficient layouts, etc.) and define which search algorithm to use inside a node (binary search, interpolation, prediction, etc.). Since the publication of the original B-tree paper [BM72] almost 50 years ago, the physical organization of B-trees has been improved in a zillion different ways, e.g., [RR99; RR00; SGL09; Kim+10].

But what concretely is **the entity** 'the index' in here? So far we only defined two different node types pointing to each other, we added a couple of constraints (fan-outs, tree-structure, concrete physical organization of inner nodes and leaves). We may also add heuristics for

invariant maintenance (split and merge). But, if we change any aspect of this, do we receive a completely different index? When is it just a variant of an *existing* index? And when is it a *new* index? For instance, if we change constraints to allow nodes to have more than one parent, would that be a completely different index entity? Or is it just that one constraint that changes (with possible implications to other features of the index)?

In this paper, we will introduce the idea of logical and physical indexes. We will show that most existing indexes can be expressed as a specific *configuration* in a generic logical and physical indexing framework[1] including B-trees, radix-trees, learned indexes, and even extendible hashing. And those configurations can be combined almost arbitrarily *within the same configuration*. This opens the book for a myriad of hybrid 'indexes'. For instance, in our framework, one extreme of an index (say a single hash table) can smoothly be morphed into another extreme (say a B-tree style index with all kinds of different layouts and search algorithms inside its nodes).

3.1.2 Problem 2: Two Completely Different Methodologies to Solve a Similar Problem

It is remarkable that there is quite a divide in databases when it comes to designing efficient components of a database system like index structures as opposed to designing query plans. For index structures, the historic and state-of-the-art approach is to define some performance goals, reason about complexities, design something on a blackboard, and then implement it. Like that an index (much like any other system component) has to be designed from scratch and then implemented. Eventually, we receive a piece of software that then (hopefully) serves the original purpose. In sharp contrast to this, since the '70s and the seminal Selinger paper [Sel+79], database researchers follow a completely different, and rather successful, design path when it comes to designing query plans: we automatically assemble complex plans from logical and physical operators.

So why follow two completely different design approaches if at the core these are similar problems? Once we are in the position to express an 'index' as a configuration in a generic logical and physical indexing framework, there is one question left: Why should we configure indexes by hand anyway? Why should we handcraft which node type to use, which node-internal search algorithm to use, which data layout, tree-levels to use, etc.? If we have different components of an index which can be interchanged freely, plus options to play with, well, then we

[1] Note that we will not introduce this as a software framework as done in [HNP95; Ber+01] but rather on a conceptual level.

have an optimization problem!

For this reason, in this paper, we will propose a genetic algorithm that, given a dataset and workload, will automatically determine a suitable logical and physical index configuration.

3.1.3 Problem Statement

We summarize the two principal problems discussed above into the following problem statement that we will investigate in this work:

1. How can we generalize the most important index structures into a common conceptual indexing framework?
2. How can we automatically breed index structures using 1.

3.1.4 Contributions

In this paper, we make the following contributions:

1. We introduce a generic index structure framework that makes a clear difference between a logical and a physical indexing framework. This is inspired by the split into logical and physical operators in relational and physical algebras/operators.
2. We present a genetic algorithm which allows us to automatically generate (breed) efficient index configurations (aka indexes).
3. We present an extensive experimental evaluation of our approach, demonstrating that we can rediscover both existing, previously handcrafted indexes and new types of hybrid indexes.

The paper is structured as follows. In Section 3.2, we introduce our logical generic indexing framework. After that, in Section 3.3, we introduce our physical generic indexing framework. Both serve as the basis for Section 3.4 where we introduce our index breeding approach. Section 3.5 contrasts our approach to related work. Section 3.6 presents our experimental evaluation. We will conclude and point out a couple of exciting future research directions in Section 3.7.

3.2 Generic Logical Indexing Framework

In this section we introduce our generic logical indexing framework. The physical indexing framework is explained Section 3.3.

Descriptions of index structures tend to mix up logical (*what* is done) and physical aspects (*how* is that achieved). For instance, consider the following sentence taken from a popular textbook.

> "A sorted file, called the *data file*, is given another file, called the *index file*, consisting of key-pointer-pairs. A search key K in the index file is associated with a pointer to a data-file record that has search key K" [GUW02, Section 13.1].

In this sentence the logical aspects of the index (black underlines, e.g., sorted, key, record) and the physical aspects of the index (red underlines, e.g., file, pointer) are introduced *at the same time* and thus mix up both aspects in the same explanation. In a way this violates physical data independence of the index structure. We want to clearly separate the logical and physical aspects of an index. This is analogue to the separation of logical and physical operators in query processing and optimization.

Basic Definitions. Any expression $\sigma_P(R)$ where P is a predicate defined on a relational schema $[R] : \{[A_1 : D_1, \ldots, A_n : D_n]\}$, i.e., a function $P : [R] \mapsto \{\text{true,false}\}$, is called a *query* on R. The result of a query is $\sigma_P(R) \subseteq R$. Given $[R]$ with an attribute A_i with a corresponding non-categorical one-dimensional domain D_i, and two constants $l, h \in D_i, l \leq h$, $\sigma_{l \leq A_i \leq h}(R)$ is a *range query* on R. It selects all tuples $t = (a_1, .., a_i, .., a_n) \in R$ where a_i is contained in the interval $[l; h]$. A range query with $l = h$ is called a point query.

3.2.1 Logical Nodes and Logical Indexes

Definition 1 (*Logical Node.*)**.** A logical node is a tuple $(\text{p}, \text{RI}, \text{DT})$:

1. $\text{p} : [R] \rightarrow D$ is a **partitioning function** on the schema $[R]$ of the dataset to index, (p may be undefined),
2. RI is the **routing information**. It is a function $RI : D \rightarrow \mathcal{P}(N)$ where N is a set of nodes and $\mathcal{P}(N)$ is the power set of N. In other words, each element of D (the target domain of p) is mapped to a subset of the nodes in N. For each outcome of the partitioning function p we can find a set of associated nodes or the empty set. Notice that the routing

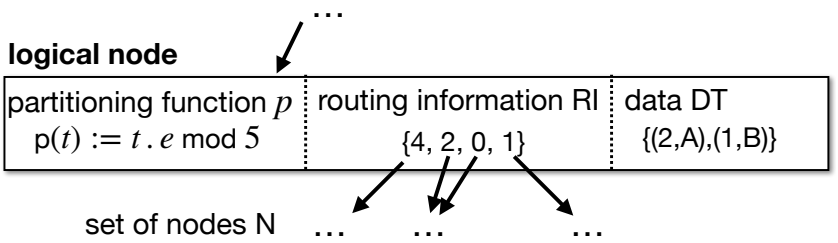


Figure 3.1: An example of a logical node with a hash-style partitioning function, four mappings in the routing information RI, and two tuples in the data part.

information does neither imply nor assume a specific physical organization including a sort order on its entries (like in B-trees). RI may be undefined. In the following, we use nodes(RI) for the set of nodes mapped to by RI.

3. DT is the **data**. It is a set of tuples with relational schema $[R]$, DT may be empty[2].

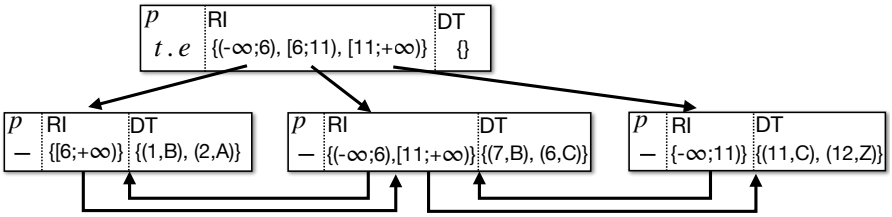
Figure 3.1 visualizes the principal structure of a logical node. The partitioning function p computes $t.e$ mod 5 which yields a domain $D = \{0, 1, 2, 3, 4\}$. Here, only a subset of D is shown in the visualization of RI, i.e., 3 is not shown as it maps to the empty set. In addition, RI maps 2 and 0 to the same node. Moreover, the data part DT contains two tuples $(2, A)$ and $(1, B)$.

Definition 2 (*Complete Logical Index.*)**.** Let LN be a set of logical nodes with $\forall_{n \in LN}$: $nodes(n.RI) \subseteq LN$. Then the graph $\lambda = (LN)$ is called a complete logical index.

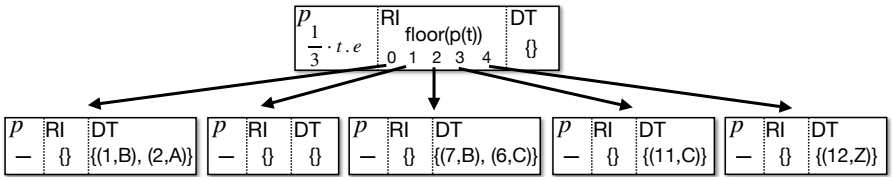
In other words, only if all routing information in the nodes of LN points to nodes contained in LN, we call LN a complete logical index. At first, this definition sounds a bit trivial, but this definition makes an important observation that is frequently overlooked: a logical index **is-a** graph of logical nodes — and **nothing else**.

Running Example. Figure 3.2 illustrates the modeling power of our framework and shows *four possible* logical indexes for $[R] = \{[e : \text{int}, g : \text{char}]\}$ and $R = \{(2, \text{A}), (7, \text{B}), (1, \text{B}), (6, \text{C}), (12, \text{Z}), (11, \text{C})\}$. Note that in these examples the DTs are empty for inner nodes. The implications of non-empty DTs are future work. Figure 3.3 demonstrates how we can model arbitrary 'hybrid' logical indexes.

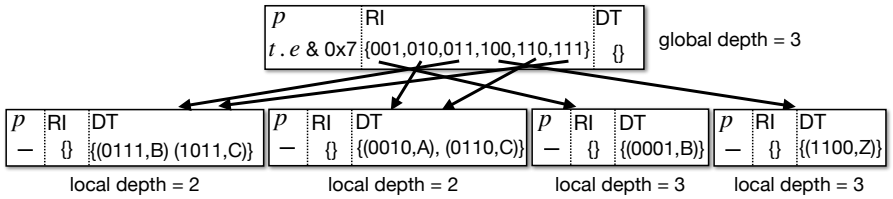
[2]In principle, DT could also be defined as a similar function as RI, the difference being that RI maps to nodes whereas DT maps to tuples. Also note that the DT-fields can be used to very naturally support buffer-tree-style indexes [Arg95], bulk loading mechanisms [BS01] as well as any form of recursive partitioning algorithm.



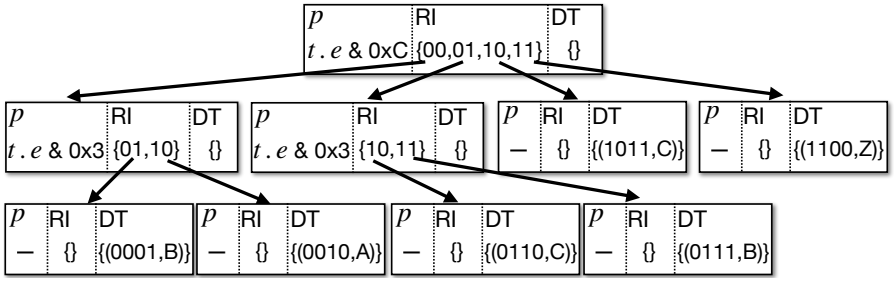
(a) **B-tree with ISAM:** Here the partitioning function returns $t.e$. The routing information maps ranges to nodes on the next level. This induces a B-tree-style partitioning. Notice that the common textbook explanation of B-trees showing k pivots and $k+1$ pointers is already a specific physical implementation of this logical index. In addition, this index contains entries on the leaf-level for backward and forward chaining of leaves as in ISAM.



(b) **RMI:** Here the partitioning function is a linear function $p(t) = \frac{1}{3} \cdot t.e + 0$ that squeezes the data into a smaller range ([0;12] → [0;4]). This is equivalent to a linear regression over the key space. RI groups the data into bins (corresponding to nodes on the next level). However, p and RI can be set to use any form of regression method and for any node independently.



(c) **Extendible Hashing:** Here the partitioning function only considers a suffix of the lowest three bits (&0x7) of $t.e$. This implies that it partitions exactly like an extendible hashing [Fag+79] directory with global depth of three.



(d) **Radix Tree:** Here the partitioning functions partition the dataset on two adjacent bits each: the root-node partitions on the first two bits of the prefix, the next level on the next two bits. This induces a radix-partitioning. Note that in this example the index is configured to keep at most one tuple per leaf. This can of course be configured. So alternatively, we could force a two-level tree just partitioning on the first two bits. The second level would then keep multiple entries in their DT-fields.

Figure 3.2: The modeling power of our logical indexing framework for traditional indexes. Four special cases of possible logical indexes for the running example. All examples mimic existing and handcrafted (physiological) index structures.

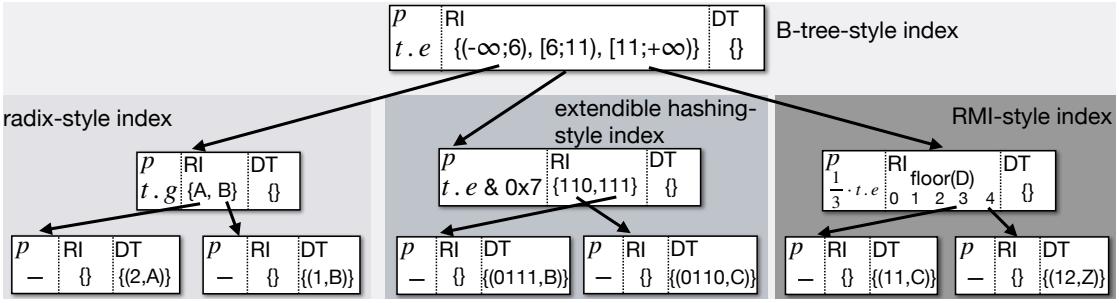


Figure 3.3: The modeling power of our logical indexing framework for any form of 'hybrid' index. The example combines properties from four different traditional index structures. Notice that there are countless examples: any node in this logical index may be exchanged by any other suitable logical node as long as the data in the index is partitioned in a way that all possible queries on the logical index return the correct result set. On this abstraction level it is still undefined *how* data is represented in the different nodes and in particular in the RI-function and the DT-set and *how* we search.

3.2.2 Logical Queries

Definition 3 (*RQ: Result of a Range Query on a Logical Index.*)**.** Given a range query with predicate $P := l \leq A_i \leq h$, a logical index λ build upon a relation R and a non-empty start node-set $SN \subseteq LN$, the result set of the range query is given by:

$$\mathrm{RQ}(P, SN) := \bigcup_{n \in SN} \left(\underbrace{\sigma_P(n.DT)}_{\text{data in } n} \cup \mathrm{RQ}\Big(P, \bigcup_{t \in R, l \leq t.A_i \leq h} n.\mathrm{RI}\big(n.p(t)\big) \Big) \right)$$

Notice that the set semantics will implicitly remove duplicates which in a physical graph-structured index (possibly not obeying set semantics) may result from visiting nodes multiple times. Also note that this query will recursively traverse the graph for all qualifying nodes in the RI-fields. This is fine for a strictly tree-structured index, however, as soon as we do not have a tree-structure anymore but a more general DAG, it may become possible that, given a set of start nodes SN, certain nodes are reachable via *multiple* paths. For a general graph, the implementing algorithm has to be modified to not visit nodes multiple times.

Definition 4 (*Correctness of a Logical Index.*)**.** Let $\lambda = (LN)$ be a complete logical index. Let SN be an arbitrary non-empty subset of start nodes: $SN \subseteq LN$. Let $DT_{\lambda:} = \bigcup_{n \in LN} n.DT$ be the

data contained in λ. Let $\sigma_{P:=l\leq A_i\leq h}(R)$ be a *range query* on R. If

$$\forall_{l,h} : \sigma_{l\leq A_i\leq h}(DT_\lambda) = \mathrm{RQ}(P, SN),$$

then λ is called a correct logical index w.r.t SN.

Notice that the correctness of an index depends on whether data is placed into the different DT-sets according to the properties of the different partitioning functions used at the various nodes. Furthermore, the start nodes SN must be chosen such that all qualifying data can be reached by the range query. For instance, in a tree-structured index picking the start node is trivial: we call it 'the root node'. In a general graph structure, which may even be disconnected, things can become more complex, i.e., we might have multiple 'root nodes', i.e., all nodes that cannot be reached from any other node of the index, or even no root nodes (in case of a cyclic graph). This discussion is beyond the scope of this paper and therefore in the following, we will only consider correct, DAG-structured indexes and assume that SN is chosen accordingly.

3.3 Generic Physical Indexing Framework

As we just have defined logical indexes (our counterparts to the logical relational algebra operators), now, we can proceed to devise physical indexes (our counterparts to physical operators).

For each logical node **and** *for each of its RI and DT-part* we eventually have to specify *how* to realize it. We do this by making a *physical* decision on the search algorithm (see Section 3.3.1) and the data layout to use for that set (see Section 3.3.2). Or, we delegate those decisions by using a nested index (see Section 3.3.3). Any index where for all its nodes the data layouts and algorithms are sufficiently specified, is called a *physical index*.

3.3.1 Specify Search Algorithm

We decide which search algorithm to use for searching (key/value)-pairs in RI and/or DT. Note that all search algorithms stop once a qualifying key was found, i.e., we found the corresponding entry in RI, or we have an exact key match in DT. The principal options are as follows:

1. **scan:** linear search through all entries, for each key check if it qualifies.
2. **binS:** binary search, iteratively check *key* in the middle of the data structure and reduce

the search area to [left, mid] or (mid, right] respectively until *key* qualifies.

3. **intS:** interpolation search, iteratively compute slope and intercept, i.e., a linear function, for *left* and *right* key, predict key location *pred* and reduce search area to [left, pred] or (pred, right] respectively until *key* qualifies.

4. **expS:** exponential search, start with the first entry, increase exponent i for key position specified by 2^i until *key* is greater than the search value, use binary search (or any other suitable method) inside range $[2^{i-1}$, end].

5. **hashS:** chained hashing (or any other suitable hashing variant), use the underlying hash function to compute the location of the *key* (and its associated mapping).

6. **linregS:** linear regression (or any other form of approximation and/or learning), compute slope and intercept, i.e., linear function, for all data points, compute error bounds, predict key location *pred* and use linear search (or any other suitable error correction method) inside [pred - lower error bound, pred + upper error bound].

7. **hybridS:** any suitable hybrid algorithm (i.e., a composite of the former options).

3.3.2 Specify Data Layout

We decide which data layout to use for representing the data from RI and/or DT. To define a data layout, we have to specify the following:

1. **col vs row:** key/value-pairs are in row or col layout.

2. **func:** we use a function to specify the RI and/or DT-mapping, thus we do not need to represent pivots and/or data and therefore do not need a data layout. As discussed in Definition 1 already, we assume the DT-fields to be actual sets even though they could be modeled as a more general mapping as well.

3. **unsorted vs sorted:** the entries are (or are not) sorted by their key.

4. **comp:** the entries are compressed (and how exactly, i.e., which compression method).

5. **hybridDL** any suitable hybrid data layout (i.e., any composite of the former options).

Notice that some of these data layout decisions cannot be made independently of the search algorithms to use, e.g., binary search implies a sorted data layout. Figure 3.4 shows an example

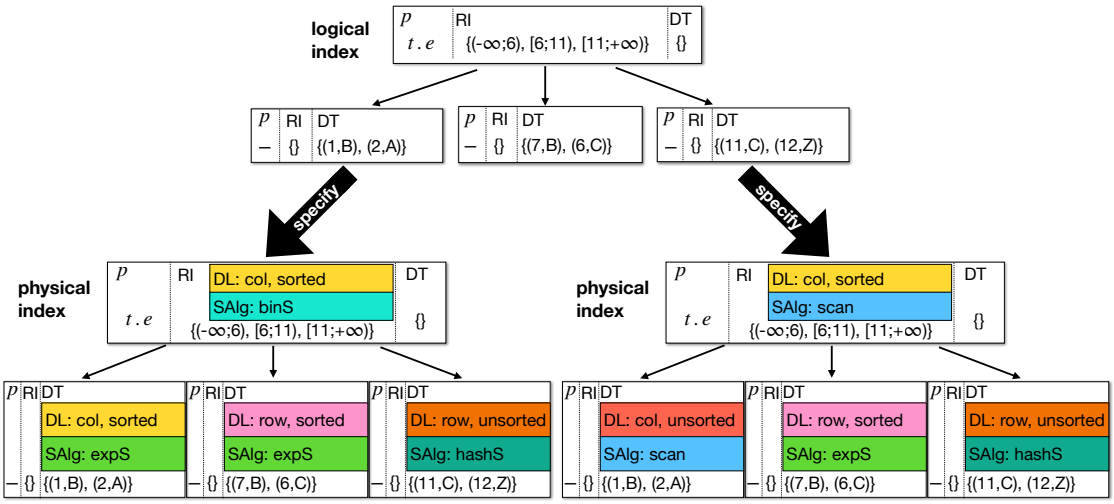


Figure 3.4: The arrows show some possible transitions from a logical to a physical index (we specify an algorithm and/or a data layout). Notice that neither the partitioning tree nor the assignment of data to nodes are changed in this process.

of a logical index that by specifying the search algorithms and data layouts may be transformed into different physical indexes.

3.3.3 Specify by Nested Logical or Physical Index

We make a decision to specify RI and DT by a nested physical index. Notice that this is not equivalent to the recursively reachable set of nodes pointed to by one particular RI. Nesting is about representing the key/value-lookup search algorithms and data layout **inside** a node by another index. For instance, consider a physical binary search tree (BST). If we use such BST to represent and search RI, we basically have a nested physical index in our node. However, this is just a special case, so in theory we can allow for arbitrary nested indexes at this point.

3.4 Genetic Index Breeding

As we just have defined our logical and physical generic indexing frameworks, we proceed to present our genetic algorithm allowing us to automatically generate indexes. This is structured as follows:

1. Core algorithm (Section 3.4.1)

2. Initial population generation (Section 3.4.2)
3. The set of applicable mutations describing possible changes to individual logical and physical index structures (Section 3.4.3)
4. The fitness function used to measure the performance of individual physical index structures (Section 3.4.4)

The major challenge with a generic indexing framework presented in Section 3.3 is the intractable search space. Therefore, we need an optimization method that can cope with such a huge search space. Notice that an intractable search space does not imply that we cannot find a good solution. In fact, entire research communities work on these kinds of problems including: planning, reinforcement learning, and genetic optimization. We decided to design our search algorithm based on genetic optimization. Genetic optimization algorithms have been developed for more than 40 years [Hol75], but recently gained a lot of attention due to growing computational resources. They allow researchers to effectively explore larger search spaces. Recent surprising, and not widely-known, results include: *genetic algorithms can rediscover state-of-the-art machine learning algorithms*(!) [Rea+20]. Furthermore, they can devise yet unknown mathematical equations [Cra+20]. Genetic optimization tasks are very domain specific as possible mutations and the performance measure depend heavily on the concrete task.

3.4.1 Core Algorithm

The general design for our algorithm follows the principal of evolution which is known from nature. Algorithm 3 shows our genetic search algorithm of GENE and Table 3.1 gives an overview of the corresponding symbols. We start with the main function GeneticSearch (line 15). We start by initializing a *population* of individuals (line 16), in our case a set of physical index structures $\Pi := \{\pi | \pi \text{ is a physical index}\}$ (see function InitPopulation, line 1). To create the initial population, we build and populate s_{init} physical index structures (line 4) and add them to the population Π (line 5). This build process is described in more detail in Section 3.4.2. Now, we enter the central iteration: we perform g_{max} iterations in genetic search (lines 17–31). We start by tournament selection (line 18), see function TournamentSelection (line 9). We select a sample of size s_{T} of the current population Π (line 10) from which we select the fittest index π_{min} (line 11). We keep a trace of the fitness of physical indexes to never evaluate indexes multiple times. We compute the median fitness $\tilde{t}$ of sample T (line 12) and return both π_{min} and

Algorithm 3 Genetic Search Algorithm of GENE

```
1:  function InitPopulation(DS, s_init)
2:      Π = ∅                                          ▷initialize population with empty set
3:      for (i = 0; i < s_init; i++) do                ▷create s_init initial indexes
4:          π = buildAndPopulateRandomIndex(DS)        ▷build and populate index
5:          Π = Π ∪ {π}                                ▷add index to population Π
6:      end for
7:      return Π                                       ▷return population Π
8:  end function

9:  function TournamentSelection(Π, s_T, W)
10:     T = sample_subset(Π, s_T)                      ▷draw random subset T ⊆ Π of size s_T
11:     π_min = arg min_{π∈T} f(π, W)                  ▷select fittest individual π_min in T under W
12:     t̃ = median_fitness(T)                          ▷compute median fitness of all π ∈ T
13:     return (π_min, t̃)                              ▷return fittest individual π_min and median fitness t̃
14: end function

15: function GeneticSearch(g_max, s_init, s_max, s_Π, s_T, s_ch, DS, MD, ND, W)
16:     Π = InitPopulation(s_init, DS)                 ▷initialize population
17:     for (i = 0; i < g_max; i++) do                 ▷perform r_max iterations/generations
18:         (π_min, t̃) = TournamentSelection(Π, s_T, W)   ▷run tournament selection
19:         for (j = 0; j < s_max; j++) do             ▷create s_max mutations
20:             m = draw_mutation(MD)                  ▷draw from mutation distribution
21:             n = draw_node(ND(π_min, m))            ▷draw from node distribution
22:             ph = draw_phys(PD(m, n))               ▷draw from phys distribution
23:             π_mut = m(π_min, n, ph)                ▷perform mutation
24:             if f(π_mut, W) ≤ t̃ then                ▷add π_mut to Π if fitter than median t̃
25:                 if |Π| ≥ s_Π then                  ▷if capacity exceeded
26:                     Π = Π \ arg max_{π∈T} f(π, W)  ▷remove unfittest individual
27:                 end if
28:                 Π = Π ∪ {π_mut}                    ▷add index to population
29:             end if
30:         end for
31:     end for
32:     π_min = arg min_{π∈Π} f(π, W)                  ▷return fittest individual of final population
33:     return π_min
34: end function
```

Table 3.1: Symbols.

Symbol	Meaning
λ	logical index
π	physical index
Π	population
s_{init}	initial size of the population
s_{Π}	maximum number of indexes in population
g_{max}	number of generations
s_{max}	number of mutations created and evaluated in a single iteration
s_{T}	size of sample in tournament selection
s_{ch}	maximum length of a mutation chain applied in one iteration
DS	dataset
π_{min}	best individual in tournament selection
π_{mut}	mutated element
$\tilde{t}$	median fitness
MD	probability distribution of mutations
m	a single mutation
$ND(\pi, m)$	probability distribution of nodes
$PD(m, N)$	probability distribution of physical implementations
W	workload of queries
$f(\pi, W)$	fitness of a physical index

$\tilde{t}$ (line 13) to the GeneticSearch function (line 18). Then, we enter the mutation loop (line 19). The core idea is to compute $s_{\text{max}} \geq 1$ mutations for index π_{min}. We draw a random mutation m from a precomputed distribution of mutations MD (line 20). For the mutation m we draw a start node n to be used for this mutation (line 21) as well as a physical implementation ph (line 22). The mutations and distributions are described in detail in Section 3.4.3.

Then, we perform the actual mutation on π_{min} (line 23) and receive π_{mut}. We originally also experimented with applying chains of mutations (lines 20 and 23) but it did not show any benefits. We check, whether the mutated index π_{mut} has a better fitness than the median $\tilde{t}$ (line 24). If it has a better fitness, we check if Π exceeds its capacity of maximum allowed physical indexes s_{Π} (line 25). If that is the case, we remove the physical index with the worst fitness from Π (line 26). Then we add π_{mut} to the population Π (line 28). Once the outer loop terminates, we determine the fittest index from Π (line 32) and return it.

3.4.2 Initial Population Generation

What is a good start population Π for the genetic algorithm? In Algorithm 3, function InitPopulation (line 1), we need to define an initial population of individual index structures. There are several possible dimensions to consider. First, we can change the initial number s_{init} of indexes in Π. This basically defines how diverse the initial set of indexes may be. Second, we should determine how to actually build and populate the initial physical index with data from dataset DS (line 4). There are several options:

1. We start with a single physical node that does not contain data, mutate it, and only then insert the actual data. We experimented with this approach initially but discarded it quickly due to its high training costs. Thus, we do not support it in our algorithm anymore.

2. We start with a single physical node containing all data. For data layout/search method we either randomly pick it or we pick one that we believe works well for the given workload.

3. We use bottom-up bulk loading with the difference that for all nodes the search algorithms and data layouts are picked randomly. In our current version we exclude hash nodes for inner nodes as we have not defined a radix-partition search method on this data layout yet. We will integrate this in future versions of our optimization framework. The resulting tree is logically similar to a standard B-Tree, the physical nodes however differ considerably.

4. We start with a population containing a physical index that resembles a state-of-the-art hand-tuned index, i.e., we define the logical index (including its partitioning functions) as well as the physical nodes. Then we check whether we can still improve that index through our genetic algorithm.

Notice that for options from 1. to 4. increasing, we postulate that we take away load from GENE, using it increasingly as a refinement tool. The more we start with something already representing a very efficient (or fit, however fitness is defined) index, the more we expect that only small mutations will be performed by GENE. At least that is what we would believe. In fact, even if we (non-randomly) specify an initial physical index to start with, recall, that GENE has all degrees of freedom to pick mutations, and may surprise us by taking unexpected turns and make different decisions.

3.4.3 Mutations and Their Distributions

In this section we introduce a suitable set of mutations and discuss how they are used in our algorithm.

Mutation. In our framework, a *mutation* is a function m : Index⟶ Index. A mutation takes a single index as input, mutates it, and returns a modified index. By 'Index' we mean, that either a logical index (λ) *or* a physical index (π) is given, and a mutated index is returned (λ_{mut} or π_{mut}). λ_{mut} and π_{mut} must preserve the correctness of λ and π. This is inspired by rewrite rules in classical query optimization: there we also only consider rules that are guaranteed to not change the query result. We will only consider mutations on tree-structured indexes. This is not a restriction of our generic framework but makes the following mutations a bit more digestible.

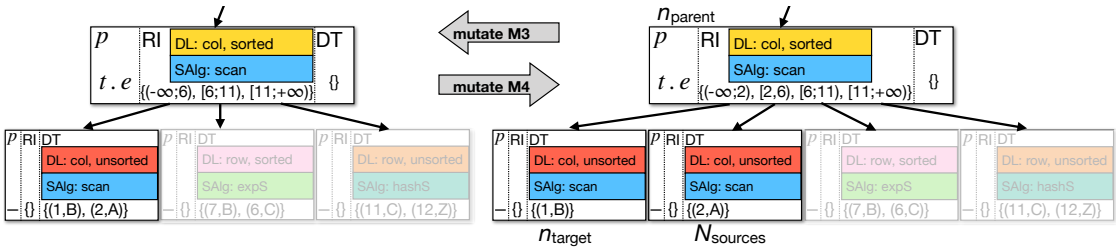
Mutation distributions. We use a probability distribution MD allowing us to assign different probabilities to the different mutations (line 20), e.g., we can prioritize certain mutations. Given a mutation m and a physical index π_{min} we draw from a second distribution $ND(\pi_{\text{min}}, m)$ to determine the nodes N for this mutation (line 21). Now, we draw from a third distribution $PD(m, N)$ to determine which physical implementation to use for this mutation and nodes. Setting probabilities to zero within this distribution $PD(m, N)$ excludes invalid combinations of physical data layout and search method, e.g., binary search on unsorted data layouts. Note that these distributions can be created based on microbenchmarks.

Fundamental Mutations. Our goal is to implement a minimal set of mutations allowing to create a huge variety of physical indexes.

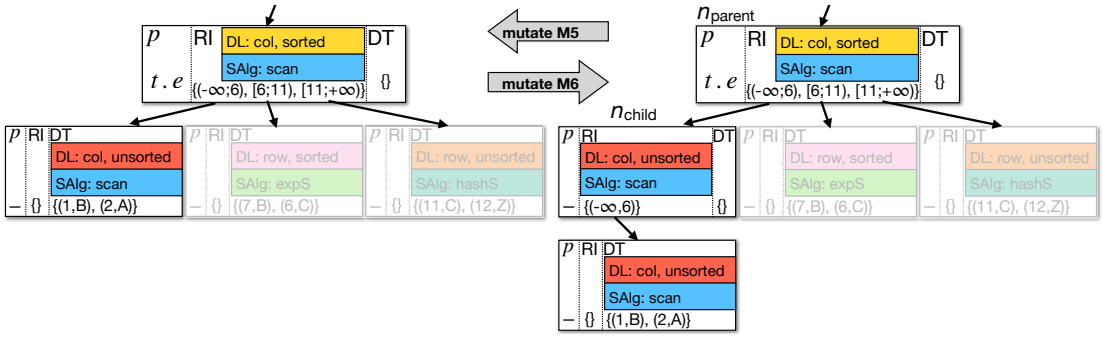
- $\mathbf{M_1}$ **Change data layout:** From n, we randomly select either its RI- *or* DT-part. Then we create a new physical node n' with data layout $n'.dl \neq n.dl$ drawn from $PD(m, N)$ with the same data and routing information as n: $n'.DT = n.DT \wedge n'.RI = n.RI$. The options for data layouts are described in Section 3.3.2. If n contains child partitions, we enforce the additional condition $n.dl' \neq$ hash, as our software framework does not (yet) support child partitions in nodes with a hash layout. In π, we replace n by n'. If $n'.s$ is incompatible with $n'.dl$, we draw a new method from $PD(m, N)$ to ensure correctness. Figure 3.5a shows an example: the input node n has a sorted column-layout. In the index, we replace n by n' which has a tree-layout.
- $\mathbf{M_2}$ **Change search method:** From n, we randomly select either its RI- *or* DT-part. Given

(a) **M_1 Change node type:** change data layout of RI.

(b) **M_2 Change search method:** change search of RI.



(c) **M_3 & M_4 Merge or split nodes horizontally:** merge left & middle child node (M3) or split leftmost child node (M4).



(d) **M_5 & M_6 Merge or split nodes vertically:** merge top-level node's left child (M5) or split it (M6).

Figure 3.5: Performing the mutations described in Section 3.4.3 on actual physical indexes.

the existing search method $n.s$, we draw an $s' \neq s$ from $PD(m, N)$. Then we create a new physical node n' with the new search method s' with the same data and routing information as n: $n'.DT = n.DT \wedge n'.RI = n.RI$. Figure 3.5b shows an example: the input node n uses a scan as search method. In the index, we replace n by n' using binary search.

M_3 Merge sibling nodes horizontally: We set node $n_{\text{parent}} := n$ whose RI maps to at least one other node in π, if not we abort this mutation. From the set of nodes mapped to by n_{parent} we randomly select a child node $n_{\text{target}} \in \text{nodes}(n_{\text{parent}}.\text{RI})$. We select a non-empty subset $N_{\text{sources}} \subseteq \text{nodes}(n_{\text{parent}}.\text{RI})$ of nodes to merge into n_{target} using the following restrictions: $n_{\text{target}} \notin N_{\text{sources}} \wedge \forall_{n \in N_{\text{sources}}} n.p = n_{\text{target}}.p$. This implies that the source domain of the routing information function D is equal for all nodes in $N_{\text{sources}} \cup \{n_{\text{target}}\}$. We then need to perform updates on two levels of the index: The node n_{target} that we merge with and the parent node n_{parent}. We start by describing the updates to the node n_{target}. First we update the data $n_{\text{target}}.\text{DT}$ and set it to the union of all data within the merged nodes:

$$n'_{\text{target}}.\text{DT} = n_{\text{target}}.\text{DT} \cup \bigcup_{n \in N_{\text{sources}}} n.\text{DT}.$$

In the following, we also update the routing information function $n_{\text{target}}.\text{RI}$ such that

$$\forall_{d \in D} n'_{\text{target}}.RI(d) = n_{\text{target}}.\text{RI}(d) \cup \bigcup_{n \in N_{\text{sources}}} n.\text{RI}(d),$$

where D is the common domain of the RIs in $N_{\text{sources}} \cup \{n_{\text{target}}\}$. This ensures that our target node n_{target} now maps to all child nodes that any node $n \in N_{\text{sources}}$ previously mapped to, i.e., we can still reach all child nodes. For the parent node n_{parent} we have to update the routing information $n_{\text{parent}}.\text{RI}$ such that

$$\forall_{d \in n_{D_{\text{parent}}}} \forall_{n \in N_{\text{sources}}} n \in n_{\text{parent}}.RI(d)$$

$$\Rightarrow n_{\text{parent}}.RI(d) = \{n_{\text{target}}\} \cup n_{\text{parent}}.RI(d) \setminus \{n\}.$$

In other words: We remove all mappings to merged nodes $n \in N_{\text{sources}}$ and replace them with a new mapping to the node n_{target}.

Notice that the merge operation performed in B-trees is essentially just a specialized version of this general merge mutation. In a B-tree the number of merged nodes k is typically set to $k = 2$ and the nodes must be directly neighboring due to the sorted key domain. For our actual implementation, we also restrict ourselves similarly to merges where $|N_{\text{sources}}| = 1$. Merge operations with larger source-sets can easily be achieved

by recursively executing the merge operation on the same node. Figure 3.5c shows an example: the set N_{sources} contains a single leaf that we want to merge into n_{target}. To achieve this we first merge all data contained in N_{sources}.DT into n_{target}.DT. As N_{sources}.RI is empty, we do not have to do anything here. In n_{parent}.RI, we need to remove the mapping to all nodes in N_{sources}, in this case the key-range $[2;6) \subset D$ must be changed to map to n_{target}. For this example this is equivalent to merging the old entry $(-\infty;2)$ with $[2;6)$ into $(-\infty;6)$. Now, all nodes in N_{sources} can be removed from the index.

M_4 Split child node horizontally into k nodes: This is the inverse mutation of M_3. Figure 3.5c shows an example.

M_5 Merge sibling nodes vertically: We set node $n_{\text{parent}} := n$ whose RI maps to at least one other node in π, if not we abort this mutation. From the set of nodes mapped to by n_{parent} we randomly select a child node $n_{\text{child}} \in \text{nodes}(n_{\text{parent}}.\text{RI})$ using the following restriction: $n_{\text{child}}.p = n_{\text{parent}}.p$. To merge n_{child} into n_{parent}, we then need to perform the following updates: First we need to move all data in n_{child}.DT to the parent node:

$$n_{\text{parent}}.\text{DT} = n_{\text{parent}}.\text{DT} \cup n_{\text{child}}.\text{DT}$$

In the following, we move potential child nodes n' of n_{child} to the parent node n_{parent}:

$$\forall_{d \in D_{\text{parent}}}\ n_{\text{child}} \in n_{\text{parent}}.\text{RI}(d)$$

$$\Rightarrow n_{\text{parent}}.\text{RI}(d) = n_{\text{parent}}.\text{RI}(d) \setminus \{n_{\text{child}}\} \cup n_{\text{child}}.\text{RI}(d)$$

where D_{parent} is the domain of n_{parent}.RI. In other words: We remove all mappings to the merged node n_{child} and replace them with mappings to the child nodes of n_{child}. For our actual implementation, we restrict ourselves to the merge of a single parent-child-pair during a single mutation. Merge operations for longer chains of nodes can easily be achieved by recursively executing the merge operation on the same node. Figure 3.5d shows an example: We select the root node as n_{parent} and its left child node as n_{child} which we want to merge into the root node. To achieve this we first merge all data contained in n_{child}.DT into n_{parent}.DT. In n_{parent}.RI, we need to remove the mapping to n_{child} and replace them with mappings to the children of n_{child}. In this case, we remove the key-range $(-\infty;6) \subset D$ and replace it with the corresponding entries of n_{child}.RI. For this example this is equivalent to re-inserting the entry $(-\infty;6)$ into n_{parent}.RI.

M_6 Split child node vertically into k nodes: This is the inverse operation of M_5. Figure 3.5d shows an example.

3.4.4 Fitness Function

The fitness function is used to measure the performance of a single physical index and describes what to optimize by the genetic algorithm (either by minimizing or maximizing its value). Its definition can be chosen freely depending on the optimization goal. We have chosen to optimize our index structures for the runtime given a specific workload consisting of point and range queries. We therefore define the fitness function f: Physical Index $\times$ Workload$\longrightarrow \mathbb{R}$ to be minimized in the following way: $f(\pi, W) = r(\pi, W)_c$. π denotes the physical index (the individual) to evaluate, W is a sequence of queries and denotes the workload of the specific experiment. $r(\pi, W)_c$ is the median runtime measured for this physical index on the workload over c runs. The fitness function can also easily be adapted to factor in other optimization goals like memory or energy efficiency. Other interesting extensions include regularization, i.e., index complexity could be punished (similar to model complexity in ML). Furthermore, we could punish or incentivize the filling grade of leaves, e.g., if leaves are fully packed, this is beneficial for read-optimized indexes but for inserts can quickly lead to structural modifications of the tree. However, if leaves are only partially filled, many inserts can be handled by leaf-local changes. All these requirements can be modeled into the fitness function.

3.5 Related Work

Handcrafted Indexes. Since the original B-tree-paper [BM72] in 1972, B-trees have become a workhorse in database systems. Since then a myriad of B-tree-variants and -improvements have been proposed [RR99; RR00; SGL09; Kim+10]. Moreover, considerable work has been done in the past years to better understand the performance of hash tables which are widely used in query processing [Alv+15; Ric+15].

Learned Indexes. The core task of a learned index [Kra+18] is to provide an index on a densely packed, sorted array. The main idea is to manually define an (outer) B-tree-like structure, typically a two-level tree (coined RMI by the authors). Then, inside each node, rather than performing a binary search on the keys contained in that node — as done in a textbook B-tree — a learned regression function is used to predict the position in the sorted array. Care has to be taken to avoid prediction errors. This is done through an error correction method: the prediction actually defines a range which must be post-filtered through a different algorithm like binary or interpolation search. The biggest advantage of a 'learned index' is that no space is required to store pivots in internal nodes thus allowing for high branching factors. Like our

work, the original work was a read-only index. It bulk loaded the index top-down, but as with any other B-tree like structure, bottom-up bulk loading up is also possible [Kip+20] and actually easier. Later on different proposals were made to use different regression techniques [Kip+19] and support inserts and deletes [Din+20; FV20]. Also note that the RMIs make a couple of other assumptions that may not always hold in practice [Cro21]. As illustrated in Figure 3.2b already, an RMI is just one special configuration in GENE: an RMI is (1) a logical index: classical B-tree (however, fixed number of layers, balancing enforced, high fan-out), (2) a physical index: node internal search constrained to use some form of linear regression. In other words, an RMI handcrafts its logical structure. Then, inside its nodes it uses a fixed physical regression method to learn a CDF. In contrast, we allow for optimizing the structure *and* the search methods and data layouts used inside nodes. Thus, we fully embrace the orthogonality of *learning a model only inside* a node vs *optimizing the entire index structure.* Our approach aims at optimizing the entire index structure not only learning weights in a handcrafted structure.

Periodic Tables and Data Calculator. The work by Stratos Idreos et al. on semi-automatic data structure design is truly inspiring. In their vision paper [Idr+18b] they aim at a complete dissection and classification of the individual primitives used to design data structures. They sketch the huge design space of indexes and conclude that many quadrants in that space are still unexplored. They also phrase the high-level vision to synthesize an index from a declarative specification. Their main idea is to use a fine-grained learned cost models to be able to cost the physical individual index primitives (like scans, binary search, etc.). However, they go not further to show how this can be achieved concretely. In addition, no split into logical and physical indexes is given which is the key enabler in our approach. The follow-up work [Idr+19b] is another vision paper which goes into somewhat more detail in describing the problem space of this endeavor and proposing a workbench like "'Data Alchemist' architecture" which is a semi-automatic design tool. However, again no experiments and/or results are shown. Then, [Idr+18a] explores a large set of physical index design primitives, benchmarks them, and uses the results to learn cost models for physical primitives. This is used to build synthesized cost models for the expected cost of a combination of those physical primitives. The authors show several indexes where these cost estimates match the actual runtimes very well. At the same time the paper emphasizes that many physical design primitives and their cost models are missing including compression, concurrency, updates, etc. In their most recent work [Idr+19a], they present the concept of design continuums, which unify different data structure designs by introducing common parameters, rules, and domains necessary to describe the underlying individuals. Using this design continuum, they show how to transition between known data structures, exposing also hybrid designs, and how to extend the continuum by new designs.

Their focus lies on the semi-automated construction of these design continuums which are supposed to support researchers and engineers in finding a close to optimal data structure for a given problem composed of workload and hardware by using it as an inference engine.

There are four important differences to our work: we focus on (1) fully automatic index structure construction, (2) we provide a clear separation into logical and physical index components, (3) we believe that the index design space is simply too big for a practical system to be comprehensively modeled by (learned) cost models one reason being that costs models of different physical primitives are often non-additive and hence not usable for an optimization process. (4) Optimization time is important but not as critical as in standard query optimization: recall that the creation of an index structure is an offline process (in contrast to the creation of an index instance at query time!). And therefore, it makes a lot of sense to define fitness via actual observed runtime measurements rather than cost models whenever possible.

Generic Frameworks. A couple of generic indexing frameworks have been proposed in the past, most notably GIST [HNP95] and XXL [Ber+01]. Those frameworks also aimed at generalizing presumably different index structures into a common software framework. This in turn allowed architects to implement important database algorithms for the generic index. The specialized indexes could then relatively easily be adapted to use the generic algorithms. Prominent examples include generic bulk loading [BSW97] and concurrency control [KMH97]. Though that work was inspiring to us, we stress that in our paper we argue on a conceptual level rather than an object-oriented-level. Moreover, we are primarily inspired by the analogue separation into logical and relational operators without immediately specifying how physical operators get implemented (ONC, vectorization, SIMD, whatever) or even how software interfaces need to be defined, as that is a tertiary concern.

DQO. Recently, we proposed Deep Query Optimization [DN20]. The core idea is to break operators into smaller components which can then possibly be optimized using traditional query optimization technique. This paper is another inspiration of our work. However, that work does not go into any detail on how such an idea can be realized in the context of indexing. It neither details how traditional operators can be split nor how this can be turned into an optimization problem for automatic index creation. We fill that gap.

Index Selection. Index Selection [LL71; Kos+20] operates on a completely different level as our approach. Instead of coming up with a concrete index structure, in index selection the goal is to determine a suitable set of attributes to index in order to improve the runtime of a workload. In contrast, in our work we consider how to devise efficient index structures in the

first place — which could then be leveraged in index selection algorithms.

Adaptive Indexing. As index selection is NP-hard, an interesting strategy is to not consider indexing a binary decision but rather allow indexes to become more and more fine-grained over time. That is at the heart of adaptive indexing [IKM07]. Several interesting proposals have been made in this space, see [Sch+13] for a survey. However, all these indexes are still handcrafted indexes. In future work, we are planning to revisit some of these techniques, as the DT-field of our logical nodes can be used to mimic many of those techniques.

Genetic Algorithms. Genetic algorithms are a long known search method for an infeasible search space and have been used in our database community for decades. Early work by Bennett et al. [BFI91] applied a genetic algorithm to search for efficient plans in a query optimizer. Other papers used similar approaches to improve database testing [Bat+07] or to perform index selection [Kor+04; Neu+19; FG89]. We are however not aware of papers tackling the problem of index creation using a genetic algorithm and therefore try to further extend the application area of these algorithms.

Decoupling Logical and Physical Indexes. Early work on partitioning schemes was done by Hellerstein et al. [Hel+02]. They represent data as a set of partitions where each partition is then (redundantly) mapped to at least one physical replica. In contrast to our work, they do not consider partitioning trees as in our logical indexes, and they also do not further detail how to physically implement each partition. In the field of structural indexing [Agt+16; Pic+14; Fle+09] introduce the idea to co-partition (or cluster) tuples in a relational schema using graph partitioning. These graph partitions can then be exploited to answer structural queries which could be difficult to compute using foreign key indexes only. Their work has a completely different goal: while we strive to create a single physical index, they strive to create a graph partitioning which can then be mapped to suitable existing indexes. Extending our logical index partitions to their graph co-partitions could be an interesting future extension to GENE. The GMAP project by Tsatalos et al. [TI94; TSI96] is another interesting work in the area of physical data independence and index design. In contrast to their work, we focus on the clear difference between a logical and physical index and not the schema and a physical index. Moreover, we automatically generate efficient index structures, while their work only allows the choice of one concrete physical index.

3.6 Experimental Evaluation

In our experiments, we first determine a suitable set of hyperparameters for our genetic framework. Based on those hyperparameters, we then carefully evaluate GENE. We highlight the cost for training and the ability to automatically reach a certain performance baseline. Finally, we show the capability of GENE to match and even beat the performance of several state-of-the-art index structures.

System. All experiments were executed on a machine with an AMD Ryzen Threadripper 1900X 8-Core processor with 32 GiB memory on Linux. Our framework and the respective experiments are implemented in C++ and compiled with Clang 8.0.1, -O3. All experiments are run single-threaded and in main-memory.

Datasets. We use three types of datasets. All datasets consist of unique 64-bit uint keys and a 64-bit payload. In the following, we refer to the keys as *data.keys*. The payload represents the offset of the corresponding key into a sorted array. Therefore, we refer to the payload as *data.offset*. The datasets exhibit a variety of different characteristics like *distribution*, *density*, *domain*, and *size*. The first dataset uni_{dense} contains keys that are uniformly distributed in a dense domain. Concretely, uni_{dense} contains keys in the range [0, n) where n is the size of the dataset. The other two datasets, *books* and *osm*, represent real-world datasets with complex distributions and are taken from [Kip+19]. The datasets are sampled-down to our specific data size by uniformly drawing elements without duplicates. We have two main dataset sizes 100K and 100M, depending on the concrete experiment. Table 3.2 gives an overview of the datasets.

Workloads. We use three classes of workloads: point, range, and mixed point and range query workloads. For the moment, all our workloads are read-only, i.e., we do not consider *insert*, *delete*, or *update* statements. Note however, that our generic framework still supports insertions and deletions. In addition, *update* statements would not alter the structure of the index, so we could easily integrate them into our framework. Table 3.3 summarizes the basic workload types. Point(data, idx_{min}, idx_{max}) represents a point query workload where the keys to lookup are taken from the keys in the dataset *data* by selecting indices in the subdomain $[\text{idx}_{\text{min}}, \text{idx}_{\text{max}}) \subseteq [0, n)$ with a uniform distribution. Likewise, Range_{sel}(data, idx_{min}, idx_{max}) describes a range query workload consisting of pairs specifying the lower bound and upper bound of the query. The lower bound is drawn with a uniform distribution in the index domain $[\text{idx}_{\text{min}}, \text{idx}_{\text{max}}$ - data.size * sel) $\subseteq [0, n)$ and the upper bound is set based on the dataset size and the given selectivity *sel*. If the domain is not explicitly specified, we assume it to cover the

Table 3.2: Datasets.

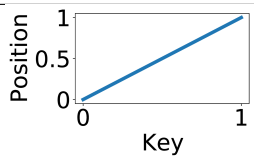
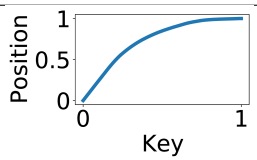
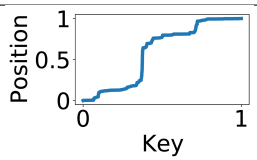
Dataset	CDF	Properties
$\text{uni}_{\text{dense}}$		n := # elements (100K, 100M) 64-bit unique unsigned integers
books		n := # elements (100K, 1M, 10M, 100M) 64-bit unique unsigned integer Dataset taken from [Kip+19]
osm		n := # elements (100K, 100M) 64-bit unique unsigned integers Dataset taken from [Kip+19]

Table 3.3: Workloads.

Workload	Characteristics	Parameters
Point(data, idx_{min}, idx_{max})	point queries in index domain $[\text{idx}_{\text{min}}, \text{idx}_{\text{max}})$ with uniform distribution	$[\text{idx}_{\text{min}}, \text{idx}_{\text{max}}) \subseteq [0, n)$
Range_{sel}(data, idx_{min}, idx_{max})	range queries in index domain $[\text{idx}_{\text{min}}, \text{idx}_{\text{max}})$ with uniform distribution and selectivity *sel*	$[\text{idx}_{\text{min}}, \text{idx}_{\text{max}}) \subseteq [0, n)$ $\text{sel} \in [0, 1]$
Mix(data, P, R)	mix of point and range query workloads with P and R being sets of respective workloads based on *data*	$P := \{p \mid p \text{ is Point(data, } \text{idx}_{\text{min}}, \text{idx}_{\text{max}})\}$ $R := \{r \mid r \text{ is } \text{Range}_{sel}\text{(data, } \text{idx}_{\text{min}}, \text{idx}_{\text{max}})\}$

whole dataset. Mix(data, P, R) represents a mix of point and range queries with P and R being sets of point and range query workloads, respectively, based on *data*. Note, that in contrast to the datasets, our workloads may contain duplicates.

As already showcased in Section 3.3 and Section 3.4, there is a huge search space in designing physical index structures. Consequently, in our experiments, we focus on the most important data layouts and search algorithms. We use the data layouts depicted in Table 3.4. As search algorithms, we use **scan**, **binS**, **intS**, **expS**, and **hashS** described in more detail in Section 3.3.1.

Table 3.4: Data layouts.

Data Layout	Characteristics	Implementation Detail
sorted_col	RI and DT have columnar layout for both keys and values. Sorted according to keys.	C++ standard library container `std::vector<Key>` and `std::vector<Value>`
hash	DT represents hash table mapping keys to their values. RI empty.	C++ standard library container `std::unordered_map<Key, Value>`
tree	RI and DT represent tree data structure mapping keys to their values. Sorted according to keys.	C++ standard library container `std::map<Key, Value>`

3.6.1 Hyperparameter Tuning

We use a uni_{dense} dataset of size 100K and vary five different parameters within this experiment:

1. Number of mutations per generation ($s_{\max}$): $s_{\max} \in \{10, 50\}$.
2. Maximum population size (s_Π): $s_\Pi \in \{50, 200, 1000\}$.
3. Tournament selection size (s_T): $s_T \in \{10\%, 50\%, 100\% \text{ of population size}\}$.
4. Initial population size (s_{init}): $s_{\text{init}} \in \{10, 50\}$.
5. Population insertion criterion (q): Instead of taking the median of the subset drawn during tournament selection, we define a percentile q to be reached for a mutated individual to be inserted into the population: $q \in \{0\%, 50\%, 100\%\}$. For the 0% percentile, we always insert the mutated individual, for the 100% percentile we only add it if it is better than the previous best individual within the tournament selection subset.

Table 3.5: Best Genetic Search Configurations (over 5 runs)

Rank	$s_{\max}$	s_Π	s_T	s_{init}	q	median runtime [s]	mean runtime [s]
1	10	200	100%	50	0%	13.72	91.72
2	10	1000	50%	50	50%	14.58	26.10
3	10	1000	100%	10	50%	16.71	24.94
4	10	1000	100%	50	0%	16.87	94.48
5	10	1000	50%	10	50%	18.21	158.49

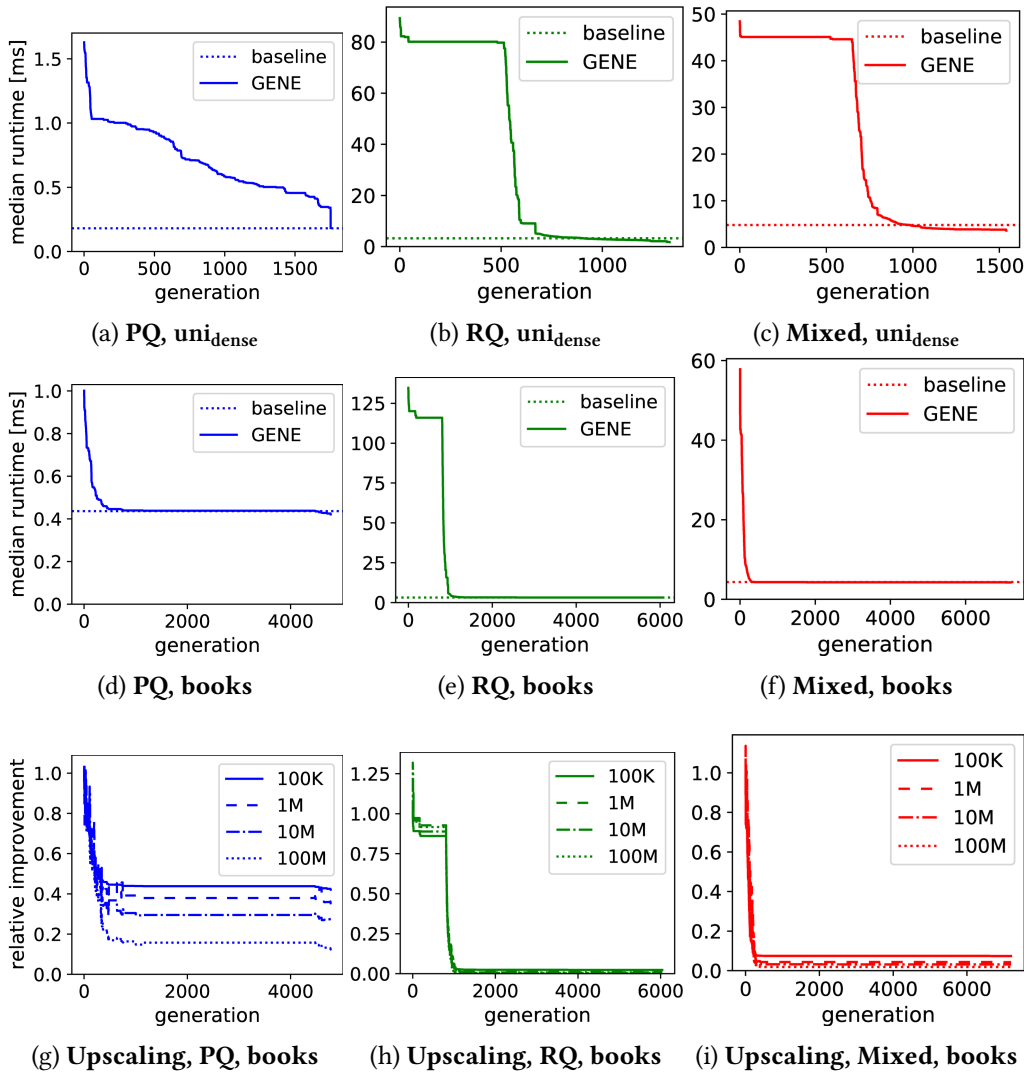
Table 3.5 shows the best configurations (based on the median of the 5 runs executed per configuration). Given a total number of mutations we want to perform, we conclude that it is more beneficial to use a smaller number of mutations per generation combined with a larger number

of generations. As the population size has a limited influence, we decided to keep it very small to reduce the overhead to maintain the population. We therefore used the following default parameters for the experiments in the following sections: $s_{\max} = 10, s_{\Pi} = 50, s_T = 25, s_{\text{init}} = 10$ and $q = 50\%$.

3.6.2 Rediscover Suitable Baseline Indexes

In this experiment, we will demonstrate that our genetic algorithm is capable of reproducing the performance of various baseline index structures as known from textbooks. We consider two different datasets: uni_{dense} of size 100K and *books* of sizes 100K, 1M, 10M, and 100M. We combine each of those two datasets with three different workloads of 10,000 queries each: $\text{Point}(\text{uni}_{\text{dense}})$, $\text{Range}_{0.001}(\text{uni}_{\text{dense}})$ and a $\text{Mix}(\text{uni}_{\text{dense}}, P, R)$ workload, with $\text{P} := \{\text{Point}(\text{uni}_{\text{dense}})\}$ and $\text{R} := \{\text{Range}_{0.01}(\text{uni}_{\text{dense}})\}$ consisting of 80% point and 20% range queries. For each workload, we define a baseline within our generic framework of which we believe it has a decent performance. For the point query only workload, we assume a simple hash table to perform best which is implemented as an index structure with a single node having the hash data layout. For the range query only and mixed workload, we assume a B-tree-like structure to offer a decent performance. We initialize the tree to have 100 fully filled leaves, each containing 1,000 elements and a fan-out of 10 for the internal nodes. Each node is configured to use the `sorted_col` layout and `binS`. We configured GENE to allow nodes to contain up to 100,000 key-value-pairs or 100,000 child partitions (potentially leading to solutions consisting of a single node or solutions with one node per element assembled under a single root node). In the initial population trees were bulk loaded with 100 equally filled leaves and a fan-out of 10, but with randomized data layouts and search methods. Each experiment is conducted for 8000 generations. The genetic search was run on the smallest sample size of 100K elements.

Figure 3.6 shows the results. The first two rows show the performance of the best individual of each generation found by the genetic algorithm compared to the respective baseline on the datasets with 100K elements. The third row shows the relative improvement compared to the initial index structure after upscaling the dataset sizes of 100K to 100M. Specifically, each time we found a new best individual, we checked whether the results carry over to the larger datasets, i.e., we created new index structures using the same routing information, data layouts, and search methods as found by GENE (i.e., using the exact same index structure), but bulk loaded them with the larger dataset, increasing leaf capacities if necessary. We then evaluated them using the exact same workload as used in the genetic search. We plot the curves up to the point of the last improvement.



(a) **PQ, uni$_{\text{dense}}$** (b) **RQ, uni$_{\text{dense}}$** (c) **Mixed, uni$_{\text{dense}}$**

(d) **PQ, books** (e) **RQ, books** (f) **Mixed, books**

(g) **Upscaling, PQ, books** (h) **Upscaling, RQ, books** (i) **Upscaling, Mixed, books**

Figure 3.6: (a-f): GENE approaching handcrafted baselines on three different workloads: A point query only workload (PQ), a range query only workload (RQ) and a mixed workload consisting of 80% point and 20% range queries. (g-i): Relative improvement compared to the initial index structure after upscaling to dataset sizes of 100K to 100M.

As we can clearly see, GENE rapidly approaches the baseline. This is mostly due to the fact that GENE can rather easily improve by mutating very inefficient nodes in the beginning. After getting close to the baseline, GENE only finds slight improvements, e.g., by changing search algorithms within nodes, which are hardly visible on the plot. The index structures found by GENE are very similar to the baselines: On the uni_{dense} dataset, GENE always returned a single node index structure. For the point query only workload, it came up with a single `hash`

node containing all entries, i.e., exactly the baseline we defined beforehand. For the range query only as well as mixed workload, GENE also reduced the index to a single node, but with `sorted_col` data layout and `intS` search method. This difference is due to the fact that range queries can not be executed efficiently on a hash node. This result is reasonable as a uniformly distributed, dense dataset can easily be modeled by an array with a linear model as search method. Considering the *books* dataset, the point query workload resulted in a tree with 68 nodes in total, 66 of them being leaves. All but one leaf are direct children of the `sorted_col` root node, the remaining leaf has a single `tree` node between itself and the root. With 48 nodes, the vast majority of the leaves has a `hash` data layout. The remaining leaves are of `sorted_col` (15) or `tree` data layout (3). The dominating search method for non-`hash` nodes is `binS`, with only 3 exceptions that use `expS`. The resulting index structure reminds of a partitioned hash map, indicating that GENE indeed approached the expected baseline. For the range query workload, we obtained an index with similar size, having 44 nodes with `sorted_col` data layout in total, 40 of them being leaves. The index has a height of three with the majority of the leaves (38) situated at depth two and only two leaves being one level below. `binS` is again the dominating search method for the leaves, with four nodes using `intS` and two using `expS` instead. The resulting index structure reminds of a shallow B-tree, indicating that GENE again approached the expected baseline. For the final mixed workload, the results are similar to the range only workload. We obtained an index of height three with 41 nodes in total (all with `sorted_col` data layout), 35 of them being leaves. The majority of the leaves is at depth two, with three leaves being one level above and one leaf being a level below. The dominating search method is again `binS`, with only 7 leaves using an `intS` instead. As for the range query only workload, GENE approached a shallow B-tree like index to match the performance of the baseline. The last line in Figure 3.6 shows the improvements of the scaled index structures for the books dataset. Each line represents the relative improvements compared to the best index structure of GENE's initial population, upscaled to the indicated dataset sizes of 100K (the size on which the search was conducted) up to 100M. We can clearly see that an improvement in the solid line representing the training data nearly always results in a very similar improvement for the upscaled index structures. The overall, relative improvement becomes even bigger with increasing dataset size, indicating that is most likely sufficient to run GENE on a sample of the data to obtain a decent index structure, highly reducing the necessary search time. If best possible performance is the ultimate goal, then GENE can again be applied to the upscaled index structure resulting from the sample to perform further fine-tuning.

We also experimented with an additional, mixed workload again consisting of 10,000 queries with an 80% / 20% point to range query ratio, based on the uni_{dense} dataset. However, this time

Table 3.6: Overview of different index types and representatives of each category.

Type	Index	Details
Tree	B-tree	TLX `btree_map` [Bin18]
Radix	ART	SOSD `ARTPrimaryLB` [Mar+20]
Learned	PGM	PGM `PGMIndex` [FV20]

we chose the queries to be normally distributed around key 75,000 with a standard deviation of 10,000, i.e., the queries were mainly focused on the upper half of the key domain. Our GENE algorithm again decided to shrink the initial index structures considerably, however it stopped after 3500 generations returning a tree with 4 levels and 25 nodes in total, 17 of them being leaves. The nodes containing the upper half of the key domain were again using the `sorted_col` layout and either `intS` or `binS`.

The total runtimes of GENE heavily depend on the concrete datasets and workloads. The fastest execution for uni_{dense} with point query only workload took less than 3 minutes until the last improvement was found. The longest run on the same dataset with range query only workload took about 122 minutes. Performing the additional upscaling steps further influenced the runtimes, leading to execution times of up to 30 hours for the *books* dataset in combination with range query only workload.

3.6.3 Optimized vs Heuristic Indexes

In this section, we will compare the performance of a GENE index with representatives of different prevalent heuristic index types. Table 3.6 gives an overview of the different index types and respective representatives. For the B+tree implementation we use the commonly used TLX baseline implementation by Bingmann [Bin18]. In particular, we use the specialized B+tree template class `btree_map` implementing STL's map container. The ART implementation is taken from the SOSD benchmark [Mar+20] by Marcus et al. and concretely, we use the implementation `ARTPrimaryLB` that supports lower bound lookups. PGM [FV20] by Ferragina et al. provides multiple implementations that support a variety of different functionalities like insertion and deletion support or compression to reduce space usage. Since we are only interested in the lookup performance, we use the default `PGMIndex` implementation. We purposely exclude hash tables since they do not support range queries efficiently.

We conduct our performance evaluation on the three different datasets, uni_{dense}, *books*, and *osm*, each with a size of $n = 100\text{M}$ data points. As for the workload, we are going to use a mixed

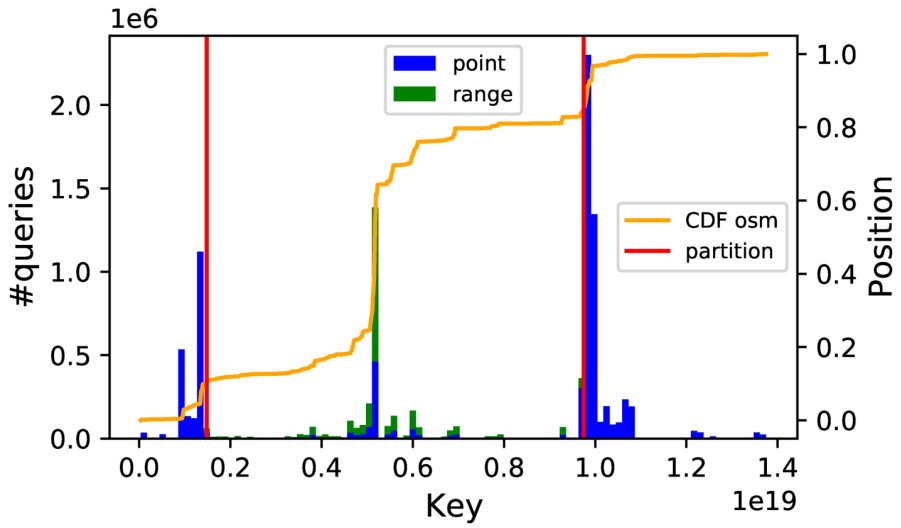


Figure 3.7: Visualization of the experimental setup. The *osm* dataset is shown as CDF while the point and range queries are illustrated as a stacked histogram. The red vertical lines highlight the partition borders.

workload consisting of multiple point and range query workloads. Concretely, the workload consists of 1M queries, divided in three point query workloads and one range query workload: Mix(data, P, R), with P := {Point(data, 0, 0.1 · n), Point(data, 0.1 · n, 0.85 · n), Point(data, 0.85 · n, n)} and R := {Range(data, 0.1 · n, 0.85 · n)}, where data ∈ {$\text{uni}_{\text{dense}}$, books, osm}. With that, the queried key domain is essentially split into three partitions at 10% and 85% of the data based on the different workloads. The first partition $[0, 0.1 \cdot n)$ exclusively receives point queries representing 20% of the total workload size. The second partition $[0.1 \cdot n, 0.85 \cdot n)$ receives a mix of both, 10% point and 20% range queries, and the third partition $[0.85 \cdot n, n)$ 50% point queries.

Figure 3.7 illustrates the workload based on the *osm* dataset. Since each data point maps a key to its position in a sorted data array, range queries can be translated to finding the position of the lower bound in the index and subsequently scanning the data array. This scan is independent of the underlying index type and can therefore be neglected. Thus, a range query in our evaluation is equivalent to a lower bound lookup in the index.

Our generic implementation allows us to easily replace specific parts of a physical index structure like the data layout or search method. However, this leads to a non-negligible performance overhead mainly due to repeated dynamic dispatches. To be competitive with the other baselines and state-of-the-art index structures, we provide an additional implementation that specifically contains the concrete physical index structures used in this experiment. Figure 3.8

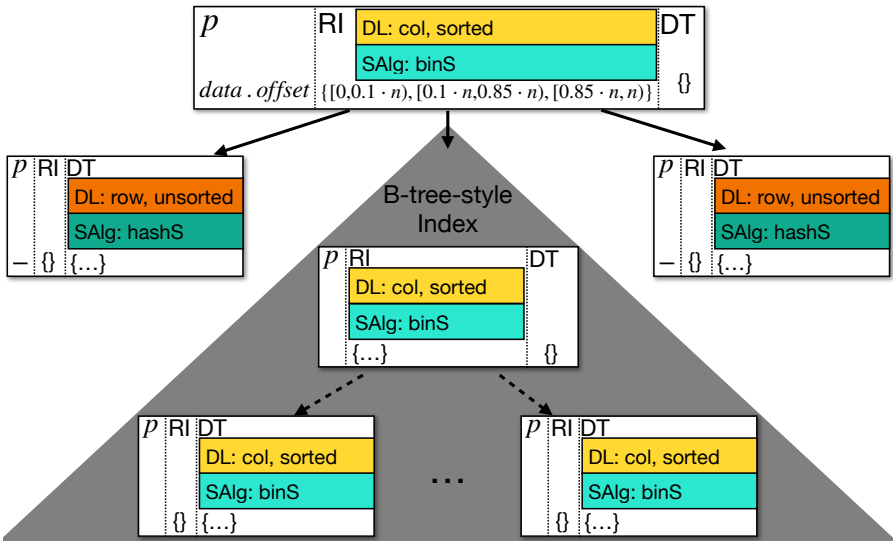


Figure 3.8: Physical index structure of the GENE index based on the workload partitioning.

shows the physical structure of our GENE index. Since the workload domain is split into three partitions with two exclusive point query regions, we bulk load our index structure accordingly. The first and third partition are hash nodes while the second partition represents a B-tree-style index. The root is a sorted array using binary search. We randomly shuffle the workload before each execution to avoid caching effects.

Figure 3.9 shows the results of the index structures for different datasets. We report the average index lookup time. Independent of the underlying dataset, the TLX B-tree requires around 700 ns and is not able to compete with the other indexes. On the uniform dense dataset, ART and PGM both achieve a lower lookup time than GENE. However, for both, a uniform dense dataset is close to the optimal use case. For the two real-world skewed and sparse datasets, our GENE index achieves a competitive or even faster lookup time than the other index structures of around 350 ns.

We are well aware that this is a very specific use case, however, it showcases that there are indeed scenarios where an optimized GENE index can outperform a state-of-the-art (heuristic) data structure. Expanding the covered design space by GENE, i.e., the available data structures and search algorithms, and automatically finding those scenarios is part of future work. In conclusion, our proof of concept emphasizes that there are use cases in which GENE is able to achieve a competitive or even superior performance than state-of-the-art index structures and therefore, confirms its validity.

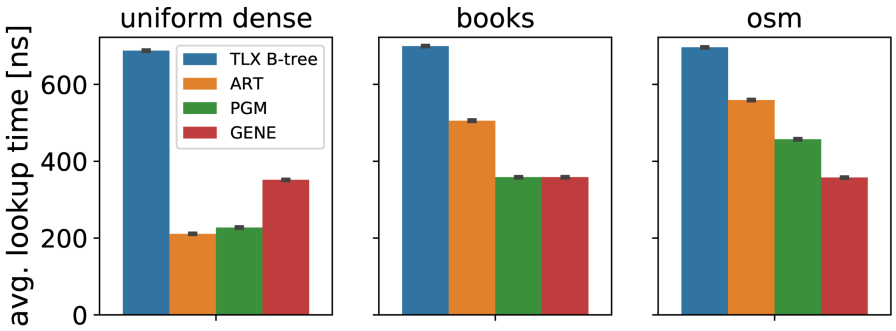


Figure 3.9: Average index lookup time comparison between three representative state-of-the-art index structures and our GENE index on three different datasets and workloads described in Section 3.6.3. The small black bars indicate the standard deviation of five runs, which is negligibly small.

3.7 Conclusion and Future Work

Conclusion. This paper has opened the book for automatically generated index structures. We have proposed a powerful generic indexing framework on the logical and physical level analogue to logical and physical operators in query processing and optimization. We have shown that by clearly separating the logical and physical dimensions of an index, a huge number of existing (physical) indexes can be represented in our generic indexing framework. Furthermore, we introduced *Genetic Generic Generation of Indexes (GENE)*. Given a workload, GENE can come up with an efficient physical index structure automatically. Our initial experimental results outline the potential and efficiency of our approach.

Future Work. This paper is obviously just a starting point and there are many possible exciting research directions ahead:

1. code-generation, similar to generating code for the most efficient physical *plan* found, generate code for the most efficient *physical index structure* found,
2. *The Index Farm*: we plan to open source our framework: the goal is that people submit a workload on a web page and the framework emits suitable source code for an index structure,
3. runtime adaptivity: how to mutate structurally, this can also simulate the adaptive indexing family of index structures,
4. updates: explore workloads with inserts, updates, and deletes,

5. scalability: extend our scalability experiments to evaluate workloads only on subtrees affected by mutations, using cost functions to prioritize expensive partitions when drawing nodes for mutations
6. effects of non-empty DT-fields in internal nodes,
7. extend GENE to support more data layouts, search algorithms, and hardware acceleration (SIMD).

Chapter 4

Extending SQL to Return a Subdatabase

4.1 Introduction

With the invention of the relational model by E. F. Codd [Cod70], *database normalization* was already identified as a key principle in database design. In that seminal paper, Codd introduced the first normal form with the aim of enabling the development of a 'universal data sublanguage'. In his follow-up work [Cod71], Codd went on to define the second and third normal form. These normal forms were designed to reduce data redundancies by eliminating undesirable dependencies between relations, improve maintenance and consistency of the data, enhance the extensibility of databases, and make the relational model more informative to users. Moreover, the logical database schema serves as the common interface for both applications and database developers. Therefore, carefully designing the logical database schema is crucial when creating a database. However, queries involving multiple relations often inherently and inevitably denormalize the underlying data when producing the result set, which contradicts the core principles of normal forms. This implies that significant effort is invested in normalizing data within the database system, yet the normalized state is often neglected once the data is queried using SQL. This occurs, for instance, when passing data to users or creating materialized views.

customers

id	*name*	*state*
0	cust_A	NY
1	cust_B	CA
2	cust_C	NY

order

cid	*pid*
0	1
1	1
2	2
2	1
0	2
1	3

products

id	*name*	*category*
0	smartphone	electronics
1	laptop	electronics
2	shirt	clothing
3	pants	clothing

Figure 4.1: Database tables with sample data. The gray rows indicate the tuples contributing to the result set in Figure 4.2.

customers ⋈ order ⋈ products

c.name	*p.name*	*p.category*
cust_A	laptop	electronics
cust_A	shirt	clothing
cust_C	laptop	electronics
cust_C	shirt	clothing

Figure 4.2: Relational result table. The colors represent attribute values of database entities (tuples) that get duplicated in the result of the query depicted in Listing 4.1.

4.1.1 Problem Statement

Consider the SQL query in Listing 4.1. Figure 4.1 depicts the corresponding database tables with sample data and Figure 4.2 shows the query result.

Listing 4.1 SQL statement.

```
1 SELECT c.name, p.name, p.category
2 FROM customers AS c, order AS o, products AS p
3 WHERE c.state = 'NY' AND
4       c.id = o.cid AND
5       p.id = o.pid;
```

When restricted to a single-table query output, computing the relational result shown in Figure 4.2 has two notable problems.

Problem 1: Relational information redundancy. The result table contains redundancies, with customer names and product information appearing multiple times due to denormalization caused by the underlying join operation. These data values are not only displayed multiple times but are also physically duplicated. In general, the larger the result table, the greater the

effort to transmit or store these result sets. This is particularly problematic for queries that produce large result sets, as it can lead to significant memory consumption during query processing when these redundancies have to be materialized at some point in the physical execution plan.

Problem 2: Relational information loss. In addition, relational information indicating that duplicated values originate from the same tuple is lost. For instance, there could be two customers named $cust_A$, and distinguishing between them in the result table would require projecting their primary keys. Another issue with the result table is that the concept of a 'key' is simply abandoned when processing data in SQL or relational algebra. For instance, in Figure 4.2, the combination of the customer and product name uniquely identifies each record. However, this information is not evident from the result set alone. SQL discards even more information from the underlying relations: where did a specific attribute in the result table originate from? Does that attribute value correspond to an attribute in one of the base relations, or was it computed? These are central questions in the area of 'data provenance' and emphasize that both SQL and relational algebra are not primarily data *retrieval* languages but rather data *transformation* languages. Both languages take a set of base relations as their input and transform them into **a single output relation**. However, considerable information regarding the relationship among the schema, keys, and data from the base relations is lost in the process.

Therefore, in this work, we argue that it is much more natural to return individual, reduced tables, i.e., a subdatabase, instead of a potentially denormalized result.

Definition 5 (Result Subdatabase)**.** Let Q be an arbitrary select-project-join query over a set of relations $R = \{R_1, \ldots, R_n\}$ that projects to a set of attributes $A = A_1 \cup \cdots \cup A_m$ where each A_i is a subset of the attributes of relation R_i and $m \leq n$. Let T be the single table result of Q over R. A result subdatabase is defined as:

$$Q_{\text{subdatabase}} := \{\pi_{A_1}(T), \ldots, \pi_{A_m}(T)\}$$

In other words, instead of returning a single table, we return the set of tables whose attributes are part of the projection of the original query, each containing only the tuples that contribute to the overall query result. This concept has a wide range of use cases where it enhances declarative simplicity and may even improve query performance, a selection of which are discussed in the following.

electronics

id	*pid*	*storage*
0	0	64 GB
1	0	32 GB
2	1	128 GB

products

id	*name*	*price*
0	smartphone	900
1	laptop	3500
2	shirt	40
3	pants	120

clothing

id	*pid*	*size*
0	2	L
1	3	XS
2	3	M

Figure 4.3: Database tables showing *electronics* and *clothing* as subtypes of *products*. The *pid* in both subtypes is a foreign key to the supertype.

4.1.2 Use Cases

1. Hierarchical Data. Consider the database tables shown in Figure 4.3, which shows a different way of modeling the products table. Instead of having the category as an attribute as shown in Figure 4.1, the products table is divided into multiple subtypes, each representing a specific category and containing category-specific information. Let us assume we are interested in all electronic and clothing products priced under 1000 Euros. Listing 4.2 shows the corresponding query. Note that, we cannot use `UNION` to compute the desired result due to the

Listing 4.2 Querying hierarchical data.

```
1 SELECT e.*, c.*
2 FROM products AS p
3      LEFT OUTER JOIN electronics AS e ON p.id = e.pid
4      LEFT OUTER JOIN clothing AS c ON p.id = c.pid
5 WHERE p.price < 1000;
```

different schemas of electronics and clothing. Furthermore, since we must merge the relevant tuples from both subtypes into a single output relation, we are forced to use `OUTER JOIN`s, which introduce `NULL` values as padding. However, with the `RESULTDB` extension, we can compute the same result split into multiple individual output relations, allowing us to eliminate the undesired and redundant `NULL` values. This use case applies more broadly to any scenario in which we need to retrieve data from multiple distinct relations that lack a direct relationship.

2. Views. Materialized views (MVs) are a powerful concept in database management systems, commonly used to precompute specific results and enhance query performance. However, MVs come with drawbacks, the most significant being storage overhead. Materializing query results as views requires physically replicating part of the underlying data, leading to additional storage costs. This issue is exacerbated by the fact that MVs often contain redundancies introduced by data denormalization through joins.

Therefore, applying the idea of materializing only the individual result sets – using the RESULTDB keyword to create the view – offers significant advantages by greatly reducing storage overhead, primarily by eliminating duplicated data. For example, assume we want to create a materialized view for the query given in Listing 4.1. Instead of materializing the single-table join result with redundancies, as shown in Figure 4.2, RESULTDB would only materialize the underlying database entities contributing to the query result, as illustrated in gray in Figure 4.1. Depending on the amount of redundancy in the data, this approach has the potential to considerably reduce storage overhead. The idea of storing only these filtered relations can also be naturally applied to *data provenance*, particularly in the context of view lineage [CWW00]. In that work, the authors propose several algorithms to reconstruct those tuples for a given data item that produce a materialized (aggregation) view. These computed sets of source data items essentially map one-to-one to our reduced base table views and can be used by the proposed algorithms to trace the lineage of data items.

The issue with only storing the filtered base relations is that a **post-join** might be required, i.e., we might have to join the individual tables again. However, this should not be viewed as a disadvantage but rather as an opportunity. On the one hand, if the materialized view contains a high amount of redundancy and the cost of executing the post-join is relatively low, it can be beneficial to send the individual result sets to the client and execute the post-join there, thereby reducing transfer overhead. On the other hand, without fully materializing the join result, we can apply filters and create index structures directly on the filtered base table views, which can be much more efficient than doing so on potentially large materialized views. Furthermore, our experiments (Section 4.6.4) show that the post-join overhead is in general extremely small.

Another advantage of computing a result subdatabase is that it allows users to conveniently define a completely customized view across multiple tables. With traditional SQL, users are limited to either defining a view for each table individually or combining data from different tables into a single table. Computing a subdatabase can be particularly useful, for instance, when defining a view related to *logical data independence* or *access control*. In addition, it is much more convenient to redefine the view if the requirements or specifications change.

3. Distributed Database Systems. In a distributed setting, it is often advantageous to process as much data as possible locally on a single node. Once processed, the results may need to be sent to another node for further computation, which can lead to significant data transfer overhead. Therefore, computing a result subdatabase locally, rather than a single-table result, can minimize the amount of data that needs to be transmitted, thereby reducing transfer time and potentially decreasing the overall computation time. This concept is similar to the general

idea of semi-join reductions in distributed settings [Ber+81], which can be seen as a subset of computing a result subdatabase but also applies to broader contexts, such as shipping the result of a query to an application server. In general, any scenario that involves transferring data over a potentially slow network can benefit from producing individual result sets. Naturally, we must consider the trade-off between reducing transfer costs and executing the post-join.

Overall, there are numerous use cases that could benefit from computing multiple individual result sets. The advantages are extensive, including smaller (intermediate) result sets, a more intuitive experience for users, and new opportunities for query optimization.

4.1.3 Contributions

We extend SQL to allow it to return a result subdatabase, i.e., only the tuples from those relations that are required to compute the query result. In summary, our contributions are as follows:

1. We introduce a backward-compatible SQL extension, `SELECT RESULTDB`, which enables `SELECT` statements in SQL to return a clearly defined subset of a database rather than just a single table. Although the introduction of this new keyword does not extend the expressive power of SQL, it fundamentally alters the underlying semantics of the computed result set. Note that, this work focuses on the *data retrieval* aspect and is therefore limited to select-project-join (SPJ) queries. However, we plan to address data transformation such as grouping and more complex operations such as set difference or anti-joins in future work. (Section 4.2)

2. We propose four rewrite algorithms that enable any SQL-92-compliant closed-source database system to support our extension. (Section 4.3)

3. We present an efficient native algorithm, that enables query optimizers to compute the result subdatabase efficiently directly inside a database system. We implement our algorithm in mu*t*able [HD23c], an open-source main memory DBMS featuring a state-of-the-art compiling query execution engine. (Section 4.4)

4. We conduct an extensive experimental study comparing traditional single-table query processing with our proposed approaches. We evaluate both our rewrite methods and the integration of our algorithm directly into a DBMS. Our results show that multiple individual result sets significantly reduce size, with our methods adding minimal overhead and, in some cases, even outperforming single-table execution. (Section 4.6)

4.2 Querying a Database to Return a Subdatabase

We propose to change SQL and relational algebra to return a *subdatabase*. That subdatabase is well-defined: for each relation in a query that is part of the final projection, we return the tuples that contribute to the query result.

4.2.1 Preliminaries

Let $\mathcal{R}$ be set of all relations and let $\mathcal{Q}$ be the set of all select-project-join queries that project to a set of attributes from its input relations.

Definition 6 (A Query Returning a Relation)**.** Let $Q \in \mathcal{Q}$ be a query. We define the evaluation of Q and its input relations $R \subseteq \mathcal{R}$ that produces a *single-table* (ST) result as follows:

$$Q_{\text{ST}} : \mathcal{Q} \times 2^{\mathcal{R}} \to \mathcal{R}, (Q, R) \mapsto T$$

Here, T is a relation with the schema from Q's final projection.

4.2.2 A Query Returning a Subdatabase

Definition 7 (A Query Returning a Subdatabase)**.** Let $Q \in \mathcal{Q}$ be a query over a set of input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$ that projects to a set of attributes $A = A_1 \cup \cdots \cup A_n$ where each A_i is a subset of the attributes of relation R_i. We define the evaluation of Q and its input relations $R \subseteq \mathcal{R}$ that produces a *subdatabase* as follows:

$$Q_{\text{RDB}} : \mathcal{Q} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}, (Q, R) \mapsto DB, \text{where}$$
$$DB := \{R'_i \mid R_i \in R \wedge A_i \neq \emptyset\}, \text{with } R'_i := \pi_{A_i}\big(Q_{\text{ST}}(Q, R)\big)$$

In other words, Q_{RDB} returns a *subdatabase*. This subdatabase contains a set of relations $R'_i \subseteq R_i$, where each relation is part of the projections in Q. The subset R'_i is defined as the result of the single-table execution of Q, projected to the attributes A_i.

To avoid confusion, the prefix 'sub' in subdatabase refers to the fact that (1.) Q_{DB} returns a subset of the input relations, and (2.) those relations contain a subset of the tuples from the input relations R. Notice that we assume set semantics of relation algebra, i.e., π returns a duplicate-free set. However, extending this to bag semantics is straightforward: our definitions

remain unchanged, with the sole adjustment being that the projection operation must preserve duplicates.

4.2.3 Relationship-Preserving Subdatabase

Definition 8 (Relationship-Preserving Subdatabase)**.** Let $Q \in \mathcal{Q}$ be a query over a set of input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$ that projects to a set of attributes $A = A_1 \cup \cdots \cup A_n$ where each A_i is a subset of the attributes of relation R_i. Let A_i^J be the subset of attributes of relation R_i that are part of the join predicates in Q. We then define $A_i' := A_i \cup A_i^J$ and the evaluation of Q and its input relations $R \subseteq \mathcal{R}$ that produces a *relationship-preserving subdatabase* as follows:

$$Q_{\text{RDB}_{RP}} : \mathcal{Q} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}, (Q, R) \mapsto DB, \text{where}$$
$$DB := \{R_i' \mid R_i \in R \wedge A_i' \neq \emptyset\}, \text{with } R_i' := \pi_{A_i'}\big(Q_{\text{ST}}(Q, R)\big)$$

In other words, we extend the set of projected attributes A_i by the attributes A_i^J required to compute the joins of Q. In particular, we require a relationship-preserving subdatabase to be able to obtain the original single-table result again. Specifically, we can reconstruct the original single-table result by computing Q_{ST} on the reduced database $Q_{\text{RDB}_{RP}}(Q, R)$:

$$Q_{\text{ST}}(Q, R) = Q_{\text{ST}}\big(Q, Q_{\text{RDB}_{RP}}(Q, R)\big).$$

Note that in some cases it is only necessary to consider a subset of the join predicates to recompute the single-table result, e.g., if a join does not contribute to the reconstruction of the original result.

4.2.4 Extending SQL: `SELECT RESULTDB`

Given a read-only query $Q \in \mathcal{Q}$ over a set of input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$. We propose to extend the `SELECT` clause of Q to

```
SELECT RESULTDB
```

This statement will return $Q_{\text{RDB}}(Q, R)$ as defined in Definition 7. In order to compute the post-join, i.e., to recompute the original single-table result again, we have to return $Q_{\text{RDB}_{RP}}(Q, R)$ as defined in Definition 8. Listing 4.3 shows an example SQL statement and Figure 4.4 the corresponding result sets produced by that statement.

Listing 4.3 Example SQL query showcasing our extension.

```
SELECT RESULTDB c.id, c.name, c.state,
                o.cid, o.pid,
                p.id, p.name, p.category
FROM customers AS c, order AS o, products AS p
WHERE c.state = 'NY' AND
      c.id = o.cid AND
      p.id = o.pid
```

customers

id	*name*	*state*
0	cust_A	NY
2	cust_C	NY

order

cid	*pid*
0	1
2	2
2	1
0	2

products

id	*name*	*category*
1	laptop	electronics
2	shirt	clothing

Figure 4.4: The computed subdatabase for the `SELECT RESULTDB` query in Listing 4.3.

Furthermore, `SELECT RESULTDB` does not require full materialization of the returned relations. Results can be returned through pipelines, akin to standard single-table query processing, with each relation mapping to an iterator (or cursor) that uses a pull- or push-based streaming interface.

4.3 SQL-Based Rewrite Methods

In this section, we discuss four SQL-based rewrite methods (RMs) that allow us to implement the query semantics introduced in Section 4.2. Since these rewrites happen entirely on the SQL level and do not require access to the source code of the database management system, this allows for very easy integration into every system, in particular closed source database systems. Our rewrite methods can be classified along two dimensions, as depicted in Figure 4.5.

Dimension 1: SQL semi-join strategy. SQL does not provide a keyword for semi-joins. Hence, we have to write semi-joins implicitly, either through an inner join followed by a `SELECT DISTINCT` on a single relation or through a subquery using the `IN` keyword. Semantically, both rewrites express the same query and return the same result set. Unfortunately, the two rewrites may yield very different plans in many database systems (breaking the strict separation of SQL's declarativeness from query optimization).

Dimension 2: Materialization strategy. Instead of *dynamically* computing join results for the semi-join strategies, we can *prematerialize* or cache data. For example, using materialized views (MVs), we can precompute the entire query result or relevant portions, achieving a trade-off between pre-materialization effort and query processing time.

	SELECT DISTINCT	Subquery
Dynamic	RM 1	RM 3
Materialized	RM 2	RM 4

Figure 4.5: SQL semi-join × materialization landscape.

In the following, we first present the intuition behind each RM, followed by a formal definition using relational algebra. Let $Q \in \mathcal{Q}$ over the input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$ be defined as:

$$Q := \pi_A\Big(\sigma_J\big(\sigma_F(R^\times)\big)\Big),$$

with $A = A_1 \cup \cdots \cup A_n$, where each A_i is a subset of the attributes of the relation R_i, $J = J_1 \wedge \cdots \wedge J_n$ where each J_i is the set of join predicates of R_i, $F = F_1 \wedge \cdots \wedge F_n$ where each F_i is the set of filter predicates of R_i, and $R^\times = R_1 \times \cdots \times R_n$ is the Cartesian product of all relations in R. Furthermore, we illustrate the application of each rewrite method using the query shown in Listing 4.1.

4.3.1 RM 1: Dynamic `SELECT DISTINCT`

The first rewrite method transforms a single query into multiple queries where each query projects to the unique attributes of exactly one relation that participates in the query output. Formally, RM 1 rewrites Q into multiple $Q_i := \pi_{A_i}(\sigma_J(\sigma_F(R^\times)))$, $\forall i$ where $A_i \neq \emptyset$. Listing 4.4 provides an example.

Listing 4.4 RM using multiple `SELECT DISTINCT`s.

```
1 BEGIN TRANSACTION;
2 SELECT DISTINCT c.name
3 FROM customers AS c, order AS o, products AS p
4 WHERE c.state = 'NY' AND c.id = o.cid AND p.id = o.pid;
5
6 SELECT DISTINCT p.name, p.category
7 FROM customers AS c, order AS o, products AS p
8 WHERE c.state = 'NY' AND c.id = o.cid AND p.id = o.pid;
9 COMMIT;
```

We use the `DISTINCT` keyword to prevent any duplicated entities due to the join and wrap the

queries into a transaction to guarantee correct results w.r.t. the current committed state of the database.

While RM 1 is relatively straightforward, the primary disadvantage is that it executes the same queries with slightly different `SELECT` clauses multiple times, including potentially costly joins.

4.3.2 RM 2: Materialized `SELECT DISTINCT`

To proactively circumvent that a potentially expensive join is executed multiple times, RM 2 makes use of MVs. In this rewrite method, we first explicitly create a (temporary and unmaintained) MV *once*. This represents a snapshot as of the time the MV is created. Second, we execute the individual queries against that MV and third, we drop the MV. Formally, RM 2 first creates $Q_{\text{MV}} := Q$ and then creates multiple $Q_i := \pi_{A_i}(Q_{\text{MV}})$, $\forall i$ where $A_i \neq \emptyset$. Listing 4.5 provides an example.

Listing 4.5 RM using an MV.

```
1 CREATE MATERIALIZED VIEW MV AS
2 SELECT c.name, p.name, p.category
3 FROM customers AS c, order AS o, products AS p
4 WHERE c.state = 'NY' AND c.id = o.cid AND p.id = o.pid;
5
6 SELECT DISTINCT c.name FROM MV;
7 SELECT DISTINCT p.name, p.category FROM MV;
8
9 DROP MATERIALIZED VIEW MV;
```

RM 2 has the advantage that it avoids the repeated computation of the same query. However, the disadvantage is that we need to materialize a potentially large query result within the DBMS, which can be particularly costly in a disk-based system. Overall, RM 2 trades upfront costs for materializing the join result versus repeated cost for the computation of the join result.

4.3.3 RM 3: Dynamic Subquery

The third rewrite method tries to steer the query optimizer into using semi-joins internally by providing a hint using subquery syntax. Formally, RM 3 rewrites Q into multiple Q_i using

semi-joins, $\forall i$ where $A_i \neq \emptyset$ as follows:

$$\begin{aligned} Q_i &:= \pi_{A_i}\Big(\sigma_J\big(\sigma_F(R^\times)\big)\Big) \\ &\overset{1.}{=} \pi_{A_i}\Big(\sigma_{F_i}(R_i) \bowtie_{J_i} \big(\sigma_{J\setminus J_i}\big(\sigma_{F\setminus F_i}(R^\times \setminus R_i)\big)\big)\Big) \\ &\overset{2.}{=} \pi_{A_i}\Big(\sigma_{F_i}(R_i) \ltimes_{J_i} \big(\sigma_{J\setminus J_i}\big(\sigma_{F\setminus F_i}(R^\times \setminus R_i)\big)\big)\Big) \end{aligned}$$

Transformation step 1. explicitly performs the join between R_i and the rest of the relations $R \setminus R_i$. In transformation step 2., the application of the left semi-join is equivalent to the join as we only project to the attributes of R_i anyway. Note that the subquery does not necessarily have to contain all other relations but only those required to compute the relevant primary key values, which depends on the specific join graph. Listing 4.6 provides an example.

Listing 4.6 RM using dynamic subqueries.

```
1  BEGIN TRANSACTION;
2  SELECT DISTINCT c.name
3  FROM customers AS c
4  WHERE c.state = 'NY' AND c.id IN -- c.id = o.cid
5      (SELECT o.cid
6       FROM order AS o, products AS p
7       WHERE o.pid = p.id);
8
9  SELECT DISTINCT p.name, p.category
10 FROM products AS p
11 WHERE p.id IN -- p.id = o.pid
12     (SELECT o.pid
13      FROM order AS o, customers AS c
14      WHERE c.state = 'NY' AND o.cid = c.id);
15 COMMIT;
```

With this rewrite, the query optimizer of a database system like PostgreSQL can efficiently execute a semi-join in certain cases without materializing a potentially large join result. However, the effectiveness of the query optimizer in utilizing a semi-join operator depends on various factors and may not always be guaranteed.

4.3.4 RM 4: Materialized Subquery

The fourth rewrite method essentially materializes a join index, i.e., for all participating relations, it only projects to their primary keys. Formally, RM 4 first creates $Q_{\text{MV}} := \pi_{A^{\text{PK}}}(\sigma_J(\sigma_F(R^\times)))$, with $A^{\text{PK}} = \{A_i^{\text{PK}} | 1 \leq i \leq n \wedge A_i \neq \emptyset\}$, where each A_i^{PK} is the set of pri-

mary key attributes of R_i. Afterward, RM 4 creates multiple $Q_i := \pi_{A_i}(R_i \ltimes_{J_i^{\text{PK}}} Q_{\text{MV}})$, $\forall i$ where $A_i \neq \emptyset$. J_i^{PK} denotes the join on the primary key attributes. Listing 4.7 provides an example.

Listing 4.7 RM using an MV and multiple subqueries.

```
1  CREATE MATERIALIZED VIEW MV AS
2  SELECT DISTINCT c.id, p.id
3  FROM customers AS c, order AS o, products AS p
4  WHERE c.state = 'NY' AND c.id = o.cid AND p.id = o.pid;
5
6  SELECT DISTINCT c.name
7  FROM customers AS c
8  WHERE c.id IN
9      (SELECT c.id from MV);
10
11 SELECT DISTINCT p.name, p.category
12 FROM products AS p
13 WHERE p.id IN
14     (SELECT p.id from MV);
15
16 DROP MATERIALIZED VIEW MV;
```

RM 4's rationale is that it typically requires significantly less storage than RM 2 and has the potential to leverage semi-joins internally.

4.4 ResultDB$_{\text{semi-join}}$ Algorithm

In Section 4.3, we focused on how we can implement queries returning a database only using SQL-based rewrites. While this approach is widely applicable due to its ease of implementation and because we do not need access to the source code, each presented method comes with its own drawbacks. We either perform a potentially expensive join multiple times, materialize a large join result, or try to enforce a semi-join optimization using subqueries.

Therefore, in this section, we present an algorithm that can be integrated into a DBMS to efficiently compute `SELECT RESULTDB` queries. Our algorithm allows us to fully reduce all relations of a join graph with an arbitrary topology. For this, we first discuss how we can leverage Yannakakis' algorithm for acyclic join graph topologies in Section 4.4.2. Next, in Section 4.4.3, we show how to transform cyclic queries into acyclic ones to reuse the algorithm from Section 4.4.2. Finally, we present our complete algorithm in Section 4.4.4.

4.4.1 Preliminaries

The core idea is to efficiently *reduce* each individual relation to the minimal set of tuples that participate in the result set as defined in Definition 7. For this, we are going to make use of *semi-joins*. The (left) semi-join between two relations R and S is defined as $R \ltimes S = \pi_{[R]}(R \bowtie S)$. Similar to previous work [Cod70; BC81], we use the term *semi-join reduction* or just *reduction* when performing a semi-join. In particular, we say 'R is reduced by S' if we perform $R \ltimes S$. Depending on the context, a semi-join reduction can also refer to the reduction of every relation that is part of a query.

As already shown in previous work [BC81], the shape of the join graph is an essential factor for computing a semi-join reduction. A join graph for some query Q is defined as $JG_Q = (R, J)$, where R is the set of relations and J is the set of joins in Q. We will always assume a connected join graph. In the following, we look separately at both *acyclic* and *cyclic* join graph topologies and discuss how we can algorithmically compute our result subdatabases.

Notion of Acyclicity. Note that *acyclicity* can be defined in various ways, with one common definition being α-acyclicity. Intuitively, it can be defined as follows (see Definition 18.2 of Arenas et al. [Are+22] for a more formal definition):

Definition 9 (α-acyclicity)**.** A query Q is acyclic iff there exists an equivalent query Q' whose join graph $JG_{Q'}$ is a tree.

However, in the scope of this work, we will use a simpler definition solely based on the structure of the join graph:

Definition 10 (JG-acyclicity)**.** A query Q is acyclic iff its join graph JG_Q is acyclic.

The problem with these different notions of acyclicity is that a query may be α-acyclic but not JG-acyclic. In this work, we decided to use JG-acyclicity due to the following two reasons. First, checking for JG-acyclicity can be done very efficiently. A join graph is cyclic iff the number of joins is equal to or greater than the number of relations. In contrast, checking for α-acyclicity requires for example the application of the GYO algorithm (see Proposition 18.6 of Arenas et al. [Are+22]), which is computationally more expensive. Second, under α-acyclicity, there can be multiple tree-shaped queries Q' for a given query Q. Deciding which Q' to consider for our algorithm basically represents another enumeration problem.

This presents a trade-off: identifying and constructing acyclic queries under α-acyclicity is computationally more expensive but may save the effort required to transform a cyclic query

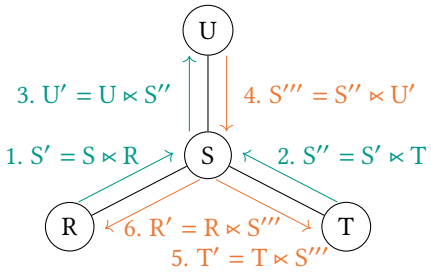


Figure 4.6: Acyclic join graph topology visualizing a possible semi-join order with U as root. $\rightarrow$ shows the bottom-up and $\rightarrow$ the top-down pass.

into an acyclic one (cf. Section 4.4.3). For the remainder of this paper, we use *(a)cyclicity* to refer specifically to *JG-(a)cyclicity*. However, exploring α-acyclicity will be considered in future work.

4.4.2 Acyclic Join Graph Topology

The Yannakakis algorithm [Yan81] provides an efficient way for solving acyclic (tree) conjunctive queries, i.e., select-project-join (SPJ) queries that have an acyclic join graph. The algorithm essentially works in three main steps. After choosing an arbitrary root node, we first perform consecutive semi-joins bottom-up from the *leaves* to the *root*. Second, we perform consecutive semi-joins top-down from the *root* to the *leaves*. After executing both bottom-up and top-down passes through the tree, all relations are reduced to their minimal set of tuples that participate in the join. Lastly, the Yannakakis algorithm joins the reduced relations to obtain a single output relation. The main motivation of this algorithm is to keep intermediate results as small as possible by reducing all relevant relations to their minimal set of tuples that participate in the join before actually joining the relations. Algorithm 4 shows the high level steps of this algorithm.

Algorithm 4 Yannakakis' Algorithm.

(0) Choose an arbitrary node in the join graph as root.
(1) Perform bottom-up semi-joins from leaves to root.
(2) Perform top-down semi-joins from root to leaves.
(3) Compute join result.

Figure 4.6 shows an acyclic join graph consisting of four relations and visualizes *one possible* semi-join order. In this example, we select U as the root and then compute a breadth-first search order for the edges starting at U. The reversed order now gives us a suitable sequence

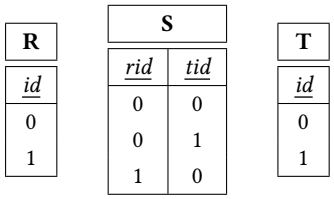
for the bottom-up semi-joins while we perform the top-down semi-joins in the original order. The semi-joins are performed in the direction of the arrows, i.e., an arrow from R to S represents the semi-join S $\ltimes$ R. Note, that for a specific node with multiple children, the order in which the semi-joins are applied does not matter for correctness but might have an impact on performance.

Algorithm 5 Reduce relations of a join graph using Yannakakis' algorithm.

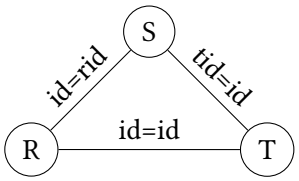
```
1:  function REDUCE_RELATIONS(G)                    ▷G is an acyclic join graph
2:      root = choose_node(G)                       ▷(0) root node
3:      edges_bfs_order = bfs_edges(G, root)
4:      for join ∈ reversed(edges_bfs_order) do     ▷(1) bottom-up
5:          semi_join(join.left, join.right)
6:      end for
7:      for join ∈ edges_bfs_order do               ▷(2) top-down
8:          semi_join(join.right, join.left)
9:      end for
10:     return G                                    ▷G contains reduced relations
11: end function
```

Algorithm 5 shows pseudocode for the computation of our result subdatabase based on the fundamental steps in Yannakakis' algorithm depicted in Algorithm 4. In line 2, we choose a root node of our tree-structured join graph (step 0). Instead of randomly selecting a root node, we employ a heuristic that favors relations included in the projections, prioritizing those with higher degrees when multiple such relations exist. This heuristic is based on two key reasons. First, since we only need to reduce relations that we eventually return, choosing a relation in the projections as root can reduce the number of semi-joins needed in the top-down pass. Second, choosing high-degree nodes typically leads to shallower trees, allowing us to perform subsequent semi-joins more effectively. The selection of the root node, which we refer to as the *Root Node Enumeration Problem*, can have a substantial impact on performance. In line 3, we order the edges of the join graph in a breadth-first-search order starting at the root node. Additionally, we assume the join operands are ordered relative to the root node, meaning there are directed edges from the root to the leaves. This is crucial because the semi-join operation is not commutative, and we must ensure that semi-joins are executed in the correct direction. In lines 4-5, we perform the bottom-up semi-joins (step 1), and in lines 6-7, we perform the top-down semi-joins (step 2). In contrast to Yannakakis' algorithm, we no longer need to execute step 3, as the reduced relations already form our desired result database. This is a very crucial characteristic of RESULTDB queries.

The most obvious reason Yannakakis' algorithm is not used in traditional query processing is



(a) Example data.



(b) Cyclic join graph topology.

R ⋈ S ⋈ T			
R.id	*S.rid*	*S.tid*	*T.id*
0	0	0	0

(c) Actual join result.

R	S		T
id	*rid*	*tid*	*id*
0	0	0	0

(d) Expected RESULTDB output.

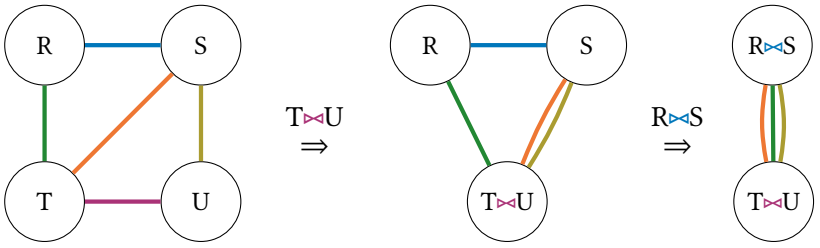
Figure 4.7: Example that shows why Yannakakis' algorithm cannot be applied to cyclic queries.

the uncertainty about whether the overhead of an initial semi-join reduction outweighs the benefits of smaller subsequent join results. However, we need not worry about this since we focus solely on the reduced relations, allowing us to fully leverage the efficiency of Yannakakis' algorithm while avoiding its potential drawbacks.

4.4.3 Cyclic Join Graph Topology

Yannakakis' algorithm is explicitly only defined for acyclic conjunctive queries. Furthermore, Bernstein et al. [BC81] show that for cyclic queries, semi-joins either cannot be used to *fully reduce* the relations or we need a very long sequence of semi-joins. 'Fully reduced' refers to a relation where only the minimal set of tuples needed for the final join result remains. In Figure 4.6 for example, after performing the first semi-join S ⋉ R, the relation S is only partially reduced. Once S is reduced by all neighboring relations, i.e., after the fourth semi-join, it cannot be further reduced anymore and thus, we call it fully reduced.

Figure 4.7 shows an example why Yannakakis' algorithm cannot be applied to cyclic queries. Assume we have the data given in Figure 4.7a and the join graph with a cyclic topology in Figure 4.7b. The actual join result only consists of a single row as depicted in Figure 4.7c. Figure 4.7d shows the corresponding expected result database. Applying Yannakakis' algorithm on this example has two fundamental problems. First, we cannot even choose a root node for this join graph. Second, no matter in which order we apply potential semi-joins, we will never filter out any of the tuples from our example data. The reason is that every row in each relation has a join partner in every other relation. As a result, Yannakakis' algorithm produces the original subdatabase shown in Figure 4.7a, and we cannot obtain our expected output.



(a) JG 1 (b) JG 2 (c) JG 3

Figure 4.8: Transformation of a cyclic join graph into an acyclic one by folding vertices, i.e., computing join results of neighboring vertices.

So, in order to leverage Yannakakis' algorithm, we need to 'transform' cyclic queries into tree queries. There is already some work on transforming graphs into a tree structure called *tree decompositions* [RS86]. We discuss this line of work in more detail in the related work Section 4.5. However, we decided to employ a more pragmatic and accessible way to get rid of cycles in a join graph.

Algorithm 6 Transform cyclic join graph into acyclic one.

```
1: function FOLD_JOIN_GRAPH(G)
2:     while G.is_cyclic() do                    ▷#joins >= #relations
3:         x = choose_node(G)
4:         y = choose_neighbor(G, x)
5:         G.replace(x, y, x ⋈ y)                ▷fold x and y; adjust affected joins
6:     end while
7:     return G                                  ▷acyclic join graph
8: end function
```

The general idea is to *fold* relations together such that the cycles in the join graph are resolved. Algorithm 6 shows pseudocode for the construction of an acyclic join graph. At the core, the algorithm consists of two steps. First, we choose a (random) node x in the join graph and one of its neighbors y (lines 3-4). Second, we replace those two nodes x and y with their join result and adjust other affected joins accordingly (line 5). We repeat these steps until our join graph is acyclic (line 2). Since our join graph is undirected and connected, we can easily check for cycles by comparing the number of joins to the number of relations. If the number of joins is equal to or greater than the number of relations, the join graph is cyclic.

Figure 4.8 shows an example transformation of a cyclic join graph into an acyclic one by folding multiple vertices. Our initial join graph (JG 1) has multiple cycles (e.g., R–S–U–T–R or R–S–T–R). In the first step, we choose the nodes T and U and replace them in the join graph with their

join result T $\bowtie$ U. Note that we now have a conjunctive join predicate between S and T $\bowtie$ U which is visualized by having multiple edges in their respective color between two nodes. After this transformation, we still have a cyclic join graph (JG 2). Therefore, we repeat the process and join the nodes R and S. The final join graph (JG 3) consists of two nodes connected by a single join containing three join predicates.

Note, that this is just *one* possible outcome because we did not specify how to choose either node x or y in Algorithm 6. Choosing nodes S and T initially would have yielded an acyclic join graph already after folding the first two nodes. The choice of folds is very likely to have a significant impact on the performance of the algorithm. This opens up a whole new optimization problem that we coin the *Folding Enumeration Problem*. Exploring this problem goes beyond the scope of this paper, and we will investigate it as part of future work. However, instead of just randomly picking the two nodes x and y, our implementation heuristically chooses the nodes with the highest degree. The rationale for this is that nodes with many join partners are more likely to be part of a cycle and with that, we might need fewer folds to reach an acyclic state.

Lemma 1. The folding process does not alter the join result and eventually results in an acyclic join graph.

Proof. Let $Q \in \mathcal{Q}$ be a cyclic query and $JG_Q = (R, J)$ be the corresponding connected join graph, where $R = \{R_1, R_2, \ldots, R_n\}$ is the set of relations and J is the set of joins in Q. Joining two arbitrary but connected relations R_i and R_j and replacing them with the join $R_i \bowtie R_j = R_{ij}$ in Q does not change the overall join result. This follows directly from the associativity of the join operation.

$$\begin{aligned} Q &= R_1 \bowtie \cdots \bowtie R_i \bowtie \cdots \bowtie R_j \bowtie \cdots \bowtie R_n \\ &= R_1 \bowtie \cdots \bowtie (R_i \bowtie R_j) \bowtie \cdots \bowtie R_n \\ &= R_1 \bowtie \cdots \bowtie R_{ij} \bowtie \cdots \bowtie R_n \end{aligned}$$

A connected, undirected join graph is acyclic iff the number of joins $|J|$ is equal to the number of relations $|R|$ minus 1, i.e., $|J| = |R| - 1$. Each folding step reduces the number of relations in the join graph by *exactly one* by merging two nodes. However, folding two nodes reduces the number of joins in the join graph by *at least one*, depending on the structure of the query graph and the relations to be joined. Therefore, we can successively join two relations until

the number of joins is less than the number of relations in the folded join graph. In the worst case, we apply the folding steps until we are left with exactly two nodes and one edge, which by definition is acyclic. From this, it follows that the folding process eventually results in an acyclic join graph. □

4.4.4 Putting It All Together

With the transformation of cyclic queries into acyclic queries complete, we can now present our final ResultDB$_{\text{semi-join}}$ Algorithm 7.

Algorithm 7 `SELECT RESULTDB` on arbitrary join graph topologies.

```
1:  function ResultDB_semi-join(G)
2:      if G.is_cyclic() then
3:          G = G.fold_join_graph(G)                ▷transform into acyclic JG
4:      end if
5:      G = reduce_relations(G)                     ▷use Yannakakis' algorithm
6:      for r ∈ G.relations do
7:          if r.is_fold() then                     ▷r could be a join result
8:              base_rels = G.decompose(r)          ▷split join into base relations
9:              G.deduplicate(base_rels)            ▷remove potential dups
10:         else
11:             G.deduplicate(r)                    ▷projection could introduce dups
12:         end if
13:     end for
14:     return G.relations                          ▷result database
15: end function
```

Given an arbitrary join graph G, we first check if the join graph is cyclic (line 2). In that case, we use our folding algorithm (line 3) to construct an acyclic join graph. Afterward, we can reuse Yannakakis' algorithm to reduce the relations (line 4). At this point, a relation can also represent a join result, and the algorithm performs the semi-joins based on the modified join predicates. These join predicates can now be conjunctions of multiple joins predicates (cf. Figure 4.8). Finally, we have to break up the joins again and remove potential duplicates. For this, we iterate over all nodes in the join graph (line 5) and check if this node is a fold (line 6). If so, we *decompose* this node again (line 7) by projecting each involved base relation and remove potential duplicates (line 8). In general, this operation can basically be seen as the inverse of the folding algorithm. We also remove duplicates due to the projection from base relations (line 10) and finally return the reduced relations (line 11).

Theorem 1. The algorithm ResultDB$_{\text{semi-join}}$ (Algorithm 7) is correct, i.e., we obtain the

correct and fully reduced relations.

Proof. Let $Q \in \mathcal{Q}$ be a query over a set of relations $R = \{R_1, R_2, \ldots, R_n\}$ and $Q_{\text{ST}}(Q, R)$ be the single-table evaluation of Q and its inputs R (see Definition 6). Further, let $\textsc{Yannakakis}_{\textsc{reduce}}$ be the reduction phase of Yannakakis' algorithm depicted in Algorithm 5, let $\textsc{Decomp}\big(Q_{\text{ST}}(Q,R)\big) = \big\{\pi_{R_1}\big(Q_{\text{ST}}(Q,R)\big), \ldots, \pi_{R_n}\big(Q_{\text{ST}}(Q,R)\big)\big\}$ be the operation that decomposes a query result into its base relations, and let $\textsc{Decomp}_{\textsc{folds}}$ analogously split all folds into their base relations.

We have to show that applying Yannakakis' reduction phase to an arbitrary join graph yields the same result as performing the decomposition operation on the single-table result. In Lemma 1, we have already shown that an arbitrary query Q can be transformed into an acyclic query Q' with its corresponding set of relations R' without changing the query result, i.e., $Q_{\text{ST}}(Q,R) = Q_{\text{ST}}(Q',R')$. If Q is already acyclic, the folding process is a no-op. Let $JG_{Q'}$ be the corresponding join graph instantiated with the underlying relations R'. Since Q' is acyclic, we can apply Yannakakis' algorithm. Therefore, $\textsc{Yannakakis}_{\textsc{reduce}}(JG_{Q'})$ produces the fully reduced relations R'_{reduced}. Since the folds in R'_{reduced} are fully reduced as well, applying the $\textsc{Decomp}_{\textsc{folds}}$ operator to those folds yields the fully reduced base relations in turn. Therefore, we can conclude:

$$\textsc{Decomp}\big(Q_{\text{ST}}(Q,R)\big) = \textsc{Decomp}_{\textsc{folds}}\big(\textsc{Yannakakis}_{\textsc{reduce}}(JG_{Q'})\big)$$

□

4.5 Related Work

SQL Flaws. SQL having some flaws is no secret. There are many publications [Dat84; TSV18; Gra08; DKK05] and blog posts [SQLFlaws14; Pra19] that present some major shortcomings of SQL as a language. For example, they criticize SQL's `NULL` value semantic, that SQL is hard to compose due to its different kinds of expressions (table vs scalar), or that it is generally very inconsistent and error-prone. In this work, we also discuss a number of problems which mainly stem from the single-table limitation causing the denormalization of tables through joins. In addition, we provide a single keyword extension that is able to fix all problems mentioned in this work and even addresses some of the flaws mentioned in related work.

SQL Extensions. Throughout the years, many different variations of SQL extensions have been introduced. Regarding single keyword extensions, the Data Cube [Gra+96] operator and the Skyline [BKS01] operator are probably the most well-known. The Data Cube operator enables N-dimensional aggregate computation while the Skyline operator allows for filtering out 'interesting' data points. Our work also introduces a new keyword that enables the computation of a result database instead of a single-table result. To the best of our knowledge, this is the first work with this contribution.

Another way of extending SQL is to move away from the traditional relational data model and introducing a query language tailored to semi-structured or unstructured data. SQL++ [OPV14] is a semi-structured query language of AsterixDB [Als+14] that basically represents a superset of the SQL and JSON data model. Extending SQL with JSON features enables the arbitrary nesting and composition of data values. While this is definitely a valid approach, our proposed extension explicitly stays in the relational word, keeping schema and relational information. In addition, it is minimally invasive since it only requires the addition of a single keyword instead of using a new query language with many different features.

Malloy [Tab21] is a new query language for describing data relationships and transformations. It compiles queries to SQL and excels at handling nested data, addressing one of our primary objectives. However, the key focus of this work is to generate multiple result sets without introducing an entirely new query language.

Semi-joins & Yannakakis' Algorithm. To implement our idea to return a result database, the fundamental idea is to use semi-joins to reduce all involved relations. Bernstein et al. [BC81] already introduces the idea to use semi-joins to solve relational queries. By 'solve relational queries' they refer to efficiently computing the reduced relations using semi-joins and subsequently computing the final join result using these reduced relations. This algorithm is also well-known as Yannakakis' [Yan81] algorithm. In their work, they show that queries with a tree-structured query graph can be solved using semi-joins. Conversely, queries with cyclic graphs generally require large *semi-join programs* or cannot be solved at all. Both works inspired our algorithmic solution.

Yang et al. [Yan+24] recently published another work utilizing Yannakakis' algorithm. The general idea is also to optimize join performance by reducing the corresponding relations as much as possible before joining them. However, instead of using semi-joins to filter the relations, they utilize Bloom filters, which turn out to be computationally more efficient. Their technique is called *predicate transfer* and can be seen as a generalization of Bloom joins. Unlike

their approach, we cannot use Bloom filters without further ado due to potential false positives. While they remove false positives in the final join, we return the filtered relations directly. We could use such a probabilistic data structure only if the post-join is always executed on the client.

Bitmaps, Bitvectors, Bitmap Join Indexes, & Bloom Filters. Value bitmaps [OQ97; CI98], bitvector filters [DCN20], bitmap join indexes [Aou+05], and their probabilistic counterparts like bloom filter [Blo70; PSS09; Lan+19] are useful optimization techniques for read-mostly scenarios. An early variant of this is the technique described by Graefe [Gra93]. Especially bitvector filters and bloom filters can be seen as a variant of sideways information passing (see paragraph below). Since ResultDB essentially only filters the base relations of a query, any existing DBMS already supporting these techniques can directly apply them to ResultDB queries. A detailed experimental evaluation of the different techniques and trade-offs is beyond the scope of this work, but we believe that it will considerably improve the performance of ResultDB queries.

Sideways Information Passing & Factorization. The underlying idea of sideways information passing (SIP) is to optimize query processing by exchanging information between arbitrary parts in a query plan. This optimization is often aimed at reducing intermediate (join) results by *reducing* relations early on. Therefore, a semi-join reduction [BC81] is actually a special case of SIP, whose application is very broad. Shrinivas et al. [Shr+13] present how sideways information passing in the context of materialization strategies can be used to improve the performance in the Vertica Analytical Database [Lam+12]. Neumann and Weikum [NW09], show how SIP can be used to speed up index scans at query runtime in RDF graphs. Another work [Zhu+17] by Zhu et al. is able to produce robust query plans in star schemas by making use of bloom filters, a SIP data structure. While SIP applies a certain reduction in a specific scenario, we unconditionally reduce all relations that are part of a query. Furthermore, we explicitly compute this reduced state of a relation whereas SIP only uses it as a means to speed up query processing.

Factorized Databases [OS16] is another concept that tries to minimize intermediate join results. However, in contrast to the aforementioned SIP methods, factorization is essentially a compression technique for join results that tries to get rid of redundancies. With that, it also targets the problem of relational information redundancies (cf. Problem 1 in Section 4.1). In contrast to factorized databases, we primarily try to avoid the redundancies in join results instead of hiding those redundancies using compression techniques.

Tree decompositions. As discussed in Section 4.4.3, tree decompositions [RS86] can be used to transform a cyclic graph into a tree structure by grouping nodes connected by edges into 'bags', adhering to specific properties. While tree decompositions are useful, particularly in database theory [Are+22; GMS09; Bod96; BCM22] for efficiently solving NP-hard problems like evaluating conjunctive queries with bounded treewidth, they have some practical issues. The main issue with tree decompositions is that they often result in trees where original nodes appear in multiple bags, increasing the join effort. To avoid this, we use a straightforward approach that joins (folds) relations until the graph is acyclic. This method provides better control over which relations are joined and allows for easier optimization with regard to the subsequent reduction phase.

Worst-case optimal joins. A key advantage of our algorithm presented in Section 4.4 is that we avoid computing potentially large intermediate results. Similarly, worst-case optimal joins (WCOJ) avoid producing large intermediate results by computing multi-way joins instead of binary joins [Ngo+12]. However, WCOJ come with some major downsides. First, they are predominantly useful in contexts like large graphs with many self-joins [Szá+22]. Second, they are mainly efficient for cyclic queries [WWS23]. Third, and most importantly, existing and well-known algorithms like the Leapfrog Triejoin [Vel12] come with impractical requirements such as the existence of ordered index structures on their input. Freitag et al. [Fre+20] provide an approach to integrate WCOJ in relational databases without such hard requirements. While this work makes WCOJ more accessible outside the database theory community, we argue that Yannakakis' algorithm is better suited, as we can use it to efficiently compute multiple result sets while avoiding large intermediate join results.

Data Provenance. There are fascinating relationships of ResultDB to lineage and provenance. For instance, the seminal work by Cui, Widom, and Wiener [CWW00] introduced query rewrites to track the *derivation set* of an output tuple t, i.e., all tuples from the input database that contributed to computing t (see Definition 8.2 of Cui, Widom, and Wiener [CWW00]). Consider any SPJ query Q and reduce its output to a single tuple by applying filters, resulting in Q^t. Then, the derivation set of t *is equal to the ResultDB query* of Q^t by definition (as long as ResultDB returns all attributes from all referred input relations)! This is because we return only those tuples that somehow contribute to at least one of the tuples in the single-table result. In a way, ResultDB queries can be seen as *multi-tuple derivation set queries*.

Similarly, the work by Niu et al. [Niu+17] and Arab et al. [Ara+14] provides techniques for optimizing provenance queries. All these techniques should be revisited to leverage ResultDB style query processing. However, this goes beyond the scope of this work.

4.6 Experiments

In this section, we first get an overview of the result set sizes of various SQL queries. Afterward, we investigate the rewrite methods discussed in Section 4.3 and our ResultDB$_{\text{semi-join}}$ algorithm presented in Section 4.4. Through our experiments, we address the following four research questions (RQs):

RQ 1 What is the level of data redundancy in the result sets of a real-world benchmark, and what is the potential data redundancy in a theoretical scenario? (Section 4.6.1)

RQ 2 How do the rewrite methods compare to each other, and what is their overhead relative to the single-table approach? (Section 4.6.2)

RQ 3 What is the query execution time of our ResultDB$_{\text{semi-join}}$ algorithm compared to the single-table approach? (Section 4.6.3)

RQ 4 How does the end-to-end runtime of our ResultDB approach compare to that of the single-table approach, including data transfer time and post-join time? (Section 4.6.4)

Setup. All experiments were conducted on an AMD Ryzen Threadripper 1900X 8-Core processor with 32 GiB main memory. The underlying operating system is Arch Linux with kernel version 6.8.2.arch2-1 on an x86_64 architecture.

Systems. We integrated our ResultDB$_{\text{semi-join}}$ algorithm from Section 4.4 into mutable [HD23c], an open-source relational DBMS with a compiling query engine [HD23a]. Currently, mutable is an early-stage research project supporting core functionality like query execution, different data layouts, and plan enumerators [HD23b]. However, mutable is still missing some functionality like indexes, multithreading, or advanced language features.

Due to the lack of certain features, we decided to use PostgreSQL for evaluating the rewrite methods presented in Section 4.3. In particular, we use PostgreSQL 16.2 and measure the client-side runtime using the `\timing` command of the accompanying interactive terminal *psql*. Furthermore, we increase the *shared_buffers* size to 16 GiB and the *work_mem* size to 1 GiB based on empirical analysis, as these parameters significantly impact performance, and use default settings otherwise.

Datasets & Workloads. To evaluate our algorithms, we use the *Join Order Benchmark* (JOB) [Lei+15], which is based on the real-world IMDb dataset and exclusively contains

SPJ queries with a variable number of joins. Some queries were slightly modified for use in mutable due to missing keywords like `IN` or `BETWEEN`, without changing their semantics.

Query Types. We distinguish the following query types:

1. *Single Table* (ST): refers to the SQL query that produces a potentially denormalized single-table join result and serves as a baseline (cf. Definition 6).
2. *ResultDB without post-join information* (RDB): refers to the query that returns only the individual relations that are part of the `SELECT` clause of the original query, i.e., we exclusively project those attributes (cf. Definition 7).
3. *ResultDB with post-join information* (RDB_{RP}): additionally projects the attributes necessary for the post-join, effectively creating a relationship-preserving query (cf. Definition 8).

4.6.1 Result Set Sizes

In this section, we first have a closer look at the result set sizes of several JOB queries. Afterward, we will examine a star schema dataset to illustrate the potential extent of data redundancy. We investigate the result set sizes for the three different approaches ST, RDB, and RDB_{RP}. We compute the size of a result set by adding up the individual sizes of all attributes that are returned. The size of an attribute is calculated by multiplying the datatype size by the number of tuples for numeric attributes, or by summing the actual string length of each tuple for character attributes.

JOB. Table 4.1 shows the result set sizes for a subset of the JOB queries in kilobytes (KiB), as well as the compression ratio defined as $^{\text{size(Single Table)}}/_{\text{size(subdatabase)}}$. We limit ourselves to those queries that we investigate in more detail later on in Section 4.6.4. In general, most JOB queries have relatively small result set sizes, typically just a few kilobytes, and also exhibit small compression ratios. This is due to the fact that those JOB queries often return a very small number of rows, project a limited set of attributes, or both. Note that, the result set sizes of some queries like `4a` for RDB_{RP} (19.07 KiB > 18.91 KiB) are even larger because they need to project additional attributes. The queries that stand out are `16b` with a very large result set size of roughly 130,376 KiB (≈ 127 MiB) and `11c` with a compression ratio of 1410. In general, most queries that return a subdatabase, regardless of whether they are relationship-preserving,

Table 4.1: JOB result set sizes in KiB (compression ratio).

Method	**3c**		**4a**		**9c**		**11c**	
ST	166.16	(1.0)	18.91	(1.0)	497.56	(1.0)	267.85	(1.0)
RDB_{RP}	101.66	(1.6)	19.07	(1.0)	57.07	(8.7)	0.19	(1409.7)
RDB	101.66	(1.6)	13.15	(1.4)	35.29	(14.1)	0.19	(1409.7)
Method	**16b**		**18c**		**21a**		**22c**	
ST	130376.72	(1.0)	686.85	(1.0)	68.06	(1.0)	783.46	(1.0)
RDB_{RP}	7884.8	(16.5)	627.69	(1.1)	0.61	(111.6)	122.68	(6.4)
RDB	3821.99	(34.1)	254.96	(2.7)	0.29	(234.7)	33.53	(23.4)
Method	**25b**		**28c**		**31a**		**33c**	
ST	0.25	(1.0)	310.66	(1.0)	45.76	(1.0)	9.32	(1.0)
RDB_{RP}	0.18	(1.4)	53.32	(5.8)	7.5	(6.1)	1.37	(6.8)
RDB	0.1	(2.5)	15.47	(20.1)	3.51	(13.0)	0.76	(12.3)

tend to produce significantly smaller result sets. This, in turn, leads to high compression ratios due to substantial redundancy in the output.

Star Schema. To illustrate the extent to which data redundancy can occur, let us consider a typical star schema consisting of a fact table and several dimension tables. In the worst case, each tuple from a specific dimension table joins with all tuples from the other dimension tables, i.e., the fact table contains the Cartesian product of the dimension tables. This results in maximum redundancy. Assume we query the star schema by joining all relations, selecting all attributes, and scaling the result set size using filters with varying selectivity across all dimension tables. Figure 4.9 shows the result set sizes of this dataset and workload. Note that, in this case, RDB only projects the payload of the dimension tables and the fact table, meaning that no primary or foreign keys are returned. In comparison, both Single Table and RDB_{RP} include this key information.

In general, the result set size for all three approaches increases quadratically with respect to the selectivity due to the characteristics of the fact table. However, the increase of the Single Table approach is much steeper since the data of all dimension tables gets repeated. The higher the selectivity value, the more tuples from the fact table participate in the join result and with that, more data from the dimension tables gets duplicated. As a result, the gap between Single Table and RDB_{RP} widens as the selectivity value increases. This redundancy is depicted in gray. Clearly, the gap widens significantly the more dimension tables we have and the larger the tuples in the dimension tables are.

Regarding **RQ 1**, these results demonstrate that denormalization through joins, whether in JOB

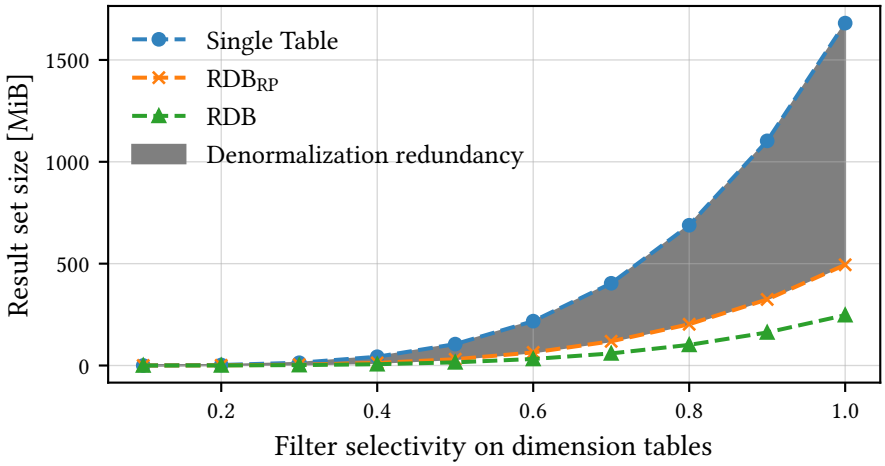


Figure 4.9: Theoretical star schema result set sizes.

Table 4.2: Overhead of the **best** rewrite method compared to the single-table execution time for the IMDb dataset and JOB queries.

Query	**1b**	**2a**	**3c**	**4a**	**5c**	**6a**	**7a**	**8a**	**9c**	**10c**	**11c**
Overhead	10.7%	0.8%	1.8%	12.9%	-2.0%	19.0%	0.6%	0.9%	23.4%	-0.6%	-84.1%
Best RM	RM 4	RM 3	RM 1	RM 4	RM 3	RM 4	RM 4	RM 4	RM 4	RM 4	RM 3
Query	**12a**	**13b**	**14a**	**15d**	**16b**	**17a**	**18c**	**19a**	**20b**	**21a**	**22c**
Overhead	3.0%	1.8%	-1.2%	43.2%	48.6%	-53.5%	4.7%	2.6%	-90.5%	-1.7%	3.1%
Best RM	RM 4	RM 4	RM 4	RM 4	RM 4	RM 3	RM 2	RM 4	RM 3	RM 4	RM 4
Query	**23a**	**24a**	**25b**	**26a**	**27a**	**28c**	**29a**	**30c**	**31a**	**32a**	**33c**
Overhead	-4.9%	0.9%	-0.0%	-75.8%	6.7%	1.2%	-0.0%	4.6%	-0.1%	9.5%	12.2%
Best RM	RM 4	RM 4	RM 4	RM 2	RM 4	RM 4	RM 4	RM 4	RM 4	RM 4	RM 4

queries or in a theoretical star schema workload, can introduce a significant amount of data redundancy. This motivates computing and returning a subdatabase instead of a potentially denormalized single-table result.

4.6.2 Rewrite Methods

In this section, we investigate the performance of our rewrite methods introduced in Section 4.3 with respect to the end-to-end client-side runtime in PostgreSQL. For each JOB query template (1–33), we show the results for one specific instantiation (a–d), selected based on whether the query is supported by mutable or chosen at random otherwise. We first compare the performance of the different rewrite methods against each other. Afterward, we examine the over-

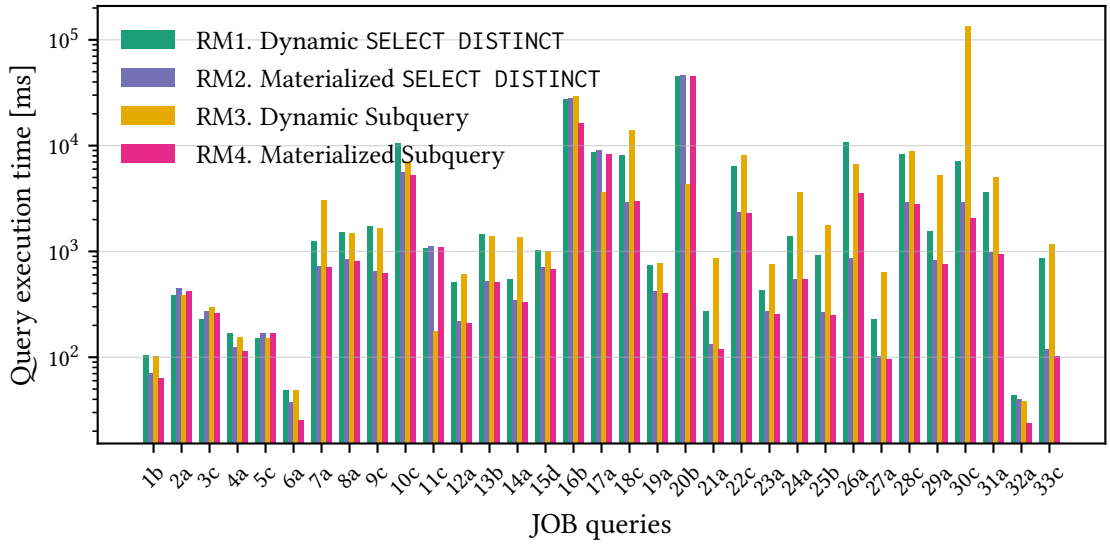


Figure 4.10: Query execution time of the rewrite methods in PostgreSQL for the IMDb dataset.

head or potential speed-up the rewrite methods introduce in comparison to the single-table execution. It is crucial to approach this comparison with caution, as the methods yield different results; it primarily aims to provide an intuition of the difference in query execution time. For this experiment, we use the RDB approach, i.e., we use the original query and do not add additional attributes for the post-join. To avoid data transfer overhead, we use `COUNT(*)` to aggregate the results. Since every rewrite method might consist of multiple queries, we summarize the measured runtime of the individual queries to obtain a single query execution time. For example, for RM 2 we first measure the time to create the MV and then the time to execute each `SELECT DISTINCT` query. We report the median query execution time of five runs.

Figure 4.10 shows the *query execution time* on a logarithmic scale for JOB. In general, it is evident that the rewrite methods exhibit significant differences in performance. No method is consistently better or worse than the others. The dynamic rewrites RM 1 and RM 3 exhibit very often a similar performance. While RM 1 is rather consistent in comparison to the other methods, RM 3 contains some noticeable outliers. For example, for `7a`, `21a`, `27a`, and `30c` the query execution time of RM 3 is up to an order of magnitude slower. However, RM 3 also has some positive applications, e.g., for `11c` or `20b`, where it outperforms the other RMs by far. Looking at the physical execution plan in PostgreSQL, we can see that RM 3 uses a different join order and essentially computes a semi-join between the outer relation and the subquery through the use of the `IN` keyword. In comparison to the other rewrites, RM 1 and RM 3

perform best if there is only a single relation referenced in the projections of the query, which is the case for 2a, 3c, 5c, 11c, 17a, and 20b. As soon as there are two or more relations, they fall behind. The materialization rewrites RM 2 and RM 4 exhibit very similar query execution times as well, with RM 4 being faster in almost all cases. This is most likely due to the fact that RM 4 simply requires less data to be materialized since it only materializes the join index instead of the complete join result (all required attributes).

Table 4.2 shows the *overhead* in percent for each query using the best performing rewrite method, comparing it to the single-table query execution time as baseline. A negative overhead represents an improvement. Except for a few queries like 15d (43.2%), and 16b (48.6%), at least one of the rewrite methods performs comparably to the single-table execution. For example, q2a and q31a have 0.8% and −0.1% overhead, respectively. However, there are also a few queries like 11c (−84.1%) and 20b (−90.5%) where our rewrite methods significantly outperform the single-table execution. In those cases, RM 3 is able to successfully apply a semi-join. This comparison underscores the small performance overhead of our approach, which is occasionally even faster.

Regarding **RQ 2**, we conclude that query execution times vary significantly across different rewrite methods depending on the type of query, with occasional outliers. However, most importantly, the best rewrite method often introduces only a marginal overhead over the single-table execution and sometimes even outperforms it. As a general rule, we recommend using RM 3 in case there is just a single output relation and RM 4 otherwise, as it is the best performing rewrite method in 75% of the cases.

4.6.3 ResultDB$_{\text{semi-join}}$ Algorithm

In this section, we evaluate the performance of our ResultDB$_{\text{semi-join}}$ algorithm described in Section 4.4 integrated into mu*t*able. Specifically, we compare the query execution time of our implementation with the single-table execution. To ensure a fair comparison despite the differing results of both approaches, we implemented a new logical/physical operator called *Decompose*. This operator is placed on top of the standard projection operator at the root of a plan. Instead of returning a single-table result set, it provides the ResultDB output by splitting the result into individual relations and removing duplicates. We briefly compare the ResultDB performance, in particular the post-join times, to the unchanged baseline in Section 4.6.4.

We implemented Algorithm 7 with one additional optimization. Once we fully reduced all relations that are part of the projections, we stop early and return the result, as there is no need

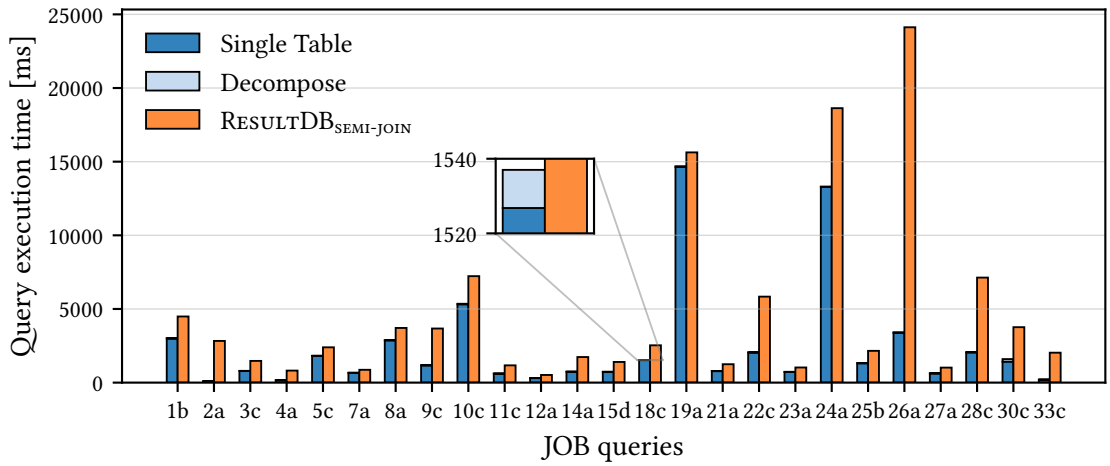


Figure 4.11: Query execution time of the ResultDB$_{\text{semi-join}}$ algorithm in mutable for the IMDb dataset. The zoom-in highlights the negligible overhead of the decompose operation.

to always reduce all relations. In addition, we minimize data transfer by returning only the number of qualifying rows instead of the complete result. As for the rewrite methods, we use the RDB approach without adding attributes for the post-join. Furthermore, we inject the real cardinalities of filtered relations and join results to avoid negative effects on the query execution time due to poor cardinality estimation. Note that, mutable's query engine is currently limited to 16 GiB and does not support varying-length character data types. To run a meaningful amount of JOB queries, we manually limit the corresponding attributes in the IMDb schema to a fixed length, determined empirically. Nevertheless, some queries still exhaust this memory limit and therefore, we were not able to include all queries. We report the median query execution time of five runs.

Figure 4.11 compares the query execution time of Single Table plus Decompose as a stacked bar plot with ResultDB$_{\text{semi-join}}$ for several JOB queries. The key observation is that ResultDB is consistently slightly slower than the Single Table plus Decompose approach. Notably, due to small result set sizes, the Decompose step introduces hardly any overhead, making it nearly imperceptible in our plot. For instance, as depicted in the zoom-in, the decomposition for 18c requires roughly just 10 ms, which is just a tiny fraction of the overall execution time. The same applies to all other queries in this experiment. Note that, preliminary experiments show that the Decompose step can be quite time-consuming for sufficiently large data sets. In general, for most queries the overhead introduced by our prototype implementation is still manageable. For example, for q5c, ResultDB$_{\text{semi-join}}$ has a runtime of 2396 ms, compared to 1821 ms for the

single-table execution. However, there are also a few outliers were ResultDB$_{\text{semi-join}}$ performs considerably worse like for 26a or 33c.

Regarding **RQ 3**, we conclude that the current implementation of the ResultDB$_{\text{semi-join}}$ algorithm does not yet match the single-table execution times, even when factoring in the decomposition. However, despite our implementation lacking many optimization opportunities, such as improved folding or an optimized semi-join reduction order, these experiments yield promising results. Furthermore, the very small Decompose overhead essentially provides us with the possibility to compute a result subdatabase very efficiently.

4.6.4 Runtime with Data Transfer & Post-join

In this section, we compare the rewrite methods and our ResultDB$_{\text{semi-join}}$ algorithm to the single-table execution in terms of *end-to-end* runtime, including query execution, data transfer, and post-join time. For this experiment, we use RDB$_{\text{RP}}$ queries to be able to compute the post-join after transferring the data. We focus on a subset of the JOB queries, aiming to cover a diverse range of query types. The data transfer rate (DTR) – the time required to transfer a query result – plays a critical role in overall performance. For the following experiments, we assume a DTR of 100 Mbps, a speed commonly regarded as reliable for general use. While data centers and cloud-based networks often achieve DTRs in the tens or even hundreds of gigabits per second, these environments typically handle significantly larger datasets. As a result, our chosen DTR provides a meaningful basis for evaluating performance in more conventional settings. To measure the post-join times, we compute and materialize the reduced relations and use the respective system, PostgreSQL or mu*t*able, to compute the final join result.

Rewrite Methods. Table 4.3 shows the end-to-end performance of the best rewrite method (RM) in PostgreSQL compared to the single-table (ST) approach. In general, for queries with larger result set sizes and high compression ratios (e.g., 9c, 16b, and 22c), the transfer time is noticeably higher for the single-table result than for the result subdatabase. For example, transferring the single-table result of 16b takes approximately 10,186 ms, whereas transferring the result subdatabase requires only around 616 ms. Furthermore, the post-join times are almost negligible in the overall runtime and in comparison to the single-table execution. In particular, many JOB queries have post-join times of just a few milliseconds, which typically represents only a small fraction of the original single-table query execution time. However, some queries, such as 16b, have high post-join times.

In general, most queries exhibit similar performance across both approaches due to the small

Table 4.3: End-to-end performance of the best rewrite method (RM) compared to the Single Table (ST) execution on JOB.

JOB Query	**3c**		**4a**		**9c**	
Approach	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**
Query Execution [ms]	226.48	367.33	101.97	173.89	505.92	656.25
Data Transfer [ms]	12.98	7.94	1.48	1.49	38.87	4.46
Post-join [ms]	-	4.93	-	5.6	-	13.87
$\sum$ [ms]	239.46	380.2	103.44	180.98	544.79	674.58
JOB Query	**11c**		**16b**		**18c**	
Approach	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**
Query Execution [ms]	1098.98	195.59	10959.35	21559.84	2783.83	3217.37
Data Transfer [ms]	20.93	0.01	10185.68	616.0	53.66	49.04
Post-join [ms]	-	0.78	-	314.06	-	35.67
$\sum$ [ms]	1119.9	196.38	21145.03	22489.9	2837.49	3302.08
JOB Query	**21a**		**22c**		**25b**	
Approach	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**
Query Execution [ms]	121.85	100.94	2219.48	2356.31	250.47	360.65
Data Transfer [ms]	5.32	0.05	61.21	9.58	0.02	0.01
Post-join [ms]	-	1.85	-	6.16	-	2.45
$\sum$ [ms]	127.17	102.83	2280.69	2372.05	250.49	363.12
JOB Query	**28c**		**31a**		**33c**	
Approach	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**
Query Execution [ms]	2792.73	2886.34	953.03	931.46	90.56	123.34
Data Transfer [ms]	24.27	4.17	3.58	0.59	0.73	0.11
Post-join [ms]	-	6.83	-	2.37	-	11.22
$\sum$ [ms]	2817.0	2897.34	956.61	934.42	91.29	134.66

overhead introduced by the data transfer and the post-join. Note that, the query execution times are slightly higher than the ones reported in Figure 4.10 because we compute a relationship-preserving subdatabase. However, there are cases where the high data transfer time has a significant impact. For instance, the time for computing a result subdatabase for 16b is almost twice as high as computing the single-table result, with an execution time of 10,959 ms compared to 21,560 ms. However, due to the reduced transfer time of approximately 616 ms – compared to nearly 10,186 ms – and the small post-join time of 314 ms, the rewrite method almost performs comparably to the single-table execution in terms of overall execution time with 21,145 ms versus 22,490 ms.

ResultDB$_{\text{semi-join}}$. Because mutable's query execution is significantly faster compared to PostgreSQL, the post-join times are also considerably reduced. For most JOB queries in this experiment (excluding 16b as it does not run in mutable), the post-join times consistently fall below 1 ms, with the highest one being 4.65 ms for 18c. The transfer times are the same as the ones in Table 4.3. As a result, the overall query execution times are similar or higher (due to computing RDB$_{\text{RP}}$) to those in Figure 4.11, leading to the decision to omit the corresponding visualization.

Regarding **RQ 4**, we conclude that in terms of end-to-end runtime, ResultDB approaches the single-table execution and that fast data transfer times can have a significant impact as for 16b.

4.7 Future Work

In the following, we suggest possible topics for further investigation that were beyond the scope of this initial work.

Query Optimization. As discussed throughout this work, there is significant optimization potential yet to be explored, which our research group is actively investigating. We are working on several different algorithms that address the *Root Node Enumeration Problem* and the *Folding Enumeration Problem*.

Data Transformations. This initial paper focuses on the *data retrieval* aspect of SQL, i.e., SPJ queries. However, we also have ongoing research regarding *data transformation*. In this context, we plan to extend our definitions in Section 4.2 accordingly. We envision to integrate arbitrary data transformation alongside data retrieval, including grouping on different criteria on different relations **at the same time**. Consequently, our approach naturally supports true

grouping sets semantics without shoehorning the different results into the same single output table.

Subqueries. For non-correlated subqueries, we already explored the idea to naturally extend table-valued subqueries to *subdatabase-valued* subqueries. Instead of a single-table result, the outer query receives multiple reduced tables. For correlated subqueries, we plan to explore rewrite techniques similar to [NK15] and investigate how this affects query optimization.

Views. In contrast to traditional views, a subdatabase view offers a view on a set of relations rather than a single relation. Similar to correlated subqueries, we want to investigate which optimization potential querying a subdatabase view exhibits.

Subdatabase Snapshot. In this paper, returning a subdatabase is restricted to the relevant tuples of a subset of the tables. However, a result subdatabase could also include metadata, statistics, indexes, or the query execution plan for performing the post-join. For instance, the query execution plan could be sent in WebAssembly and executed within a sandbox by the client, eliminating the need for users to perform the post-join manually.

API Integration. Current database APIs like JDBC drivers, expect a single-table result, i.e., cursor of tuples, to be returned. We propose a minimally invasive extension, enabling to return a set of cursors, with each cursor corresponding to a distinct result set. Further, we aim to explore the feasibility of a cursor that iterates over the join co-groups of multiple result sets, reducing the user's burden of performing the post-join on the client.

4.8 Conclusion

SQL comes with the very hard limitation that each query result is shoehorned into a single table. In this work, we initially discuss the fundamental problems, data redundancy and information loss, that stem from this limitation. To address these problems, we propose to extend the SQL `SELECT` clause by a single keyword: `RESULTDB`. This extension enables us to return a subdatabase, i.e., a subset of tables, each containing only the tuples that contribute to the overall query result, instead of a single, potentially denormalized, table. Furthermore, we introduce a formalization of our SQL extension showing that it is well-defined and has clear semantics.

We present two classes of approaches to support our new functionality. First, we propose four SQL-based rewrite methods allowing us to transform traditional SQL queries into semantically

equivalent queries returning a result subdatabase. Second, we propose an efficient algorithm that can be integrated directly into a DBMS. We also show promising experimental results. Computing individual result sets can significantly reduce the result set size, leading to faster data transfer times and lower storage requirements for materialized views. Further, our rewrite methods and algorithm introduce only minimal overhead and can sometimes even outperform the single-table execution.

Limitations As already pointed out in Section 4.7, this work opens the book for a lot of exciting follow-up works that should investigate some of its limitations. Our experiments show that our native algorithm is slower than the single-table execution. However, we also showed that we can use our decompose operator, a simple extension to any traditional query execution plan, to efficiently compute a result subdatabase. Although ResultDB is slower, the comparison is not entirely fair. Single-table queries benefit from 50 years of extensive research, while the application of Yannakakis' algorithm in real systems experienced little optimization effort so far. However, we are confident that future work can address many of these challenges. In addition, this work is currently limited to SPJ queries. However, we already have ongoing research that tackles data transformation like aggregation and arithmetic expressions.

Chapter 5

Conclusion

This thesis reexamines key areas of relational query processing and indexing, approaching them from a fresh perspective to reformulate problem statements and introduce novel solutions. By challenging conventional methods, we aim to offer new insights and techniques within these domains.

5.1 Deep Query Optimization

Summary. In the first part of this thesis, we introduce the concept of deep query optimization (DQO). The idea involves breaking up logical and physical operators into more fine-granular subcomponents during the query optimization process. Rather than constructing a query execution plan on the granularity of complete operators, we envision a series of incremental unnesting steps that progressively increase the physicality of the query plan and enable additional optimizations. To manage the expanding search space, we introduce materialized algorithmic views (MAVs) alongside the algorithmic view selection problem (AVSP). MAVs allow for prematerializing parts of the query plan, whereas the AVSP addresses the challenge of deciding which parts of a query plan can be effectively replaced by MAVs, optimizing both performance and resource utilization.

In our experimental evaluation, we demonstrate the effectiveness of DQO using the logical grouping operator. Depending on the characteristics of the input data, different physical implementations of the logical grouping operation yield substantial variation in execution times. Additionally, we show that DQO-enabled dynamic programming – where meta-relational prop-

erties and different physical representations of logical operators are considered – can produce query plans with significantly improved estimated plan quality.

Limitations & Future Research Directions. DQO is a concept that critically challenges the existing way of optimizing queries. As a visionary work, it opens possibilities for a range of applications within and beyond query processing. However, it is important to note that there is currently no concrete implementation or integration of DQO into an existing system. The experimental evaluation was conducted in an isolated environment, highlighting the potential of DQO. However, this setup inevitably lacks the internal influences of a full system, making it difficult to assess the real impact. Consequently, it is essential to integrate DQO into a system and examine the trade-off between optimization time and execution time improvement for deep query plans.

We argue that MAVs are a powerful tool for tackling the search space explosion that inevitably accompanies a deeper optimization process. While promising, a fundamental analysis on how MAVs impact the search space and how much overhead the AVSP introduces, is necessary.

We propose that the transformation from logical to physical plan can be seen as a physiological design continuum, motivating the development of a physiological algebra. This requires identifying the appropriate granularity and determining which logical and physical aspects should be included in a new subcomponent.

Lastly, the idea to approach the query optimization process more holistically is widely discussed, and there exists a plethora of research in this area, some of which is based on our DQO work. It is important to investigate the overlaps among these studies and to explore the most effective strategies for deepening the optimization process.

5.2 Genetic Generic Generation of Index Structures

Summary. In the second part of this thesis, we present a framework for the genetic generic generation of index structures (GENE). Instead of treating index structures as monolithic and hand-crafted entities, we propose to split an index into two parts: a logical and a physical index. Based on this split, we formulate index construction as an optimization problem that we solve using genetic programming.

In our experimental evaluation, we demonstrate that GENE is able to rediscover existing baseline indexes. Furthermore, an optimized GENE index that is tailored to a specific dataset and

workload is able to match and, in some cases, even outperform state-of-the-art index structures.

Limitations & Future Research Directions. To represent a wide variety of index structures, our generic framework has to offer the flexibility to support any node type and to combine arbitrary node types. This flexibility is achieved through object-oriented design, which, however, introduces a considerable runtime overhead when evaluating the performance of these indexes during the genetic search. In contrast, state-of-the-art indexes are typically highly tuned and optimized. Therefore, implementing some form of code generation for the index structures resulting from our genetic search is essential to reduce interpretation overhead and achieve competitive performance.

Currently, the range of supported data layouts and search methods defining our physical nodes is limited to the most fundamental building blocks of traditional index structures. To explore a broader portion of the search space and increase the potential for discovering improved index structures, it is important to extend the generic indexing framework with additional node types.

The initial implementation of our genetic algorithm serves as a proof of concept, primarily demonstrating the feasibility of our approach. While considerable effort has gone into developing and refining the mutations and the fitness function, there remains substantial room for improvement. For instance, evaluating the current fitness function is very time-consuming. The performance could be significantly improved by selectively reevaluating only those parts of the index affected by recent mutations. In general, revising and maturing our genetic optimization algorithm may unlock significant potential to generate higher-quality index structures more efficiently.

5.3 Extending SQL to Return a Subdatabase

Summary. In the third part of this thesis, we address a fundamental limitation of SQL: its traditional design restricts it to returning only a single result set. Specifically, we introduce a novel keyword extension, `RESULTDB`, that modifies SQL's output semantics. This extension ensures that the tables that are part of the query – reduced to the tuples contributing to the traditional query result – are returned. This has a wide variety of benefits, including reduced result set sizes, enhanced declarative simplicity, and new opportunities for query optimization. To compute a result subdatabase, we propose four SQL-based rewrite methods and a native algorithm, the latter of which has been integrated into a state-of-the-art compiling query execution engine.

In our experimental evaluation, we show that returning a result subdatabase can significantly reduce the result set size, which in turn leads to faster data transfer times and lower storage requirements for materialized views. Additionally, our rewrite methods and algorithm introduce minimal overhead to the query execution time compared to single-table execution. Most importantly, when accounting for data transfer and post-join time, the end-to-end time required to compute a result subdatabase is, in a fair number of cases, lower than the traditional query execution time.

Limitations & Future Research Directions. Our initial work on implementing the computation of a result subdatabase establishes a foundational framework but does not focus on optimizing the computation process. To address this, we introduce two significant optimization problems. Although some follow-up has already begun to tackle these issues, further research in that domain is necessary. In particular, optimizing `RESULTDB` queries – including data transfer and post-join – offers substantial opportunities for improving the overall execution process.

We describe the inherent issue in SQL where data retrieval and data transformation are intertwined, leading to a loss of relational information. In its fundamental form, `SELECT RESULTDB` focuses solely on the data retrieval aspect, leaving data transformation unaddressed. However, ongoing research already addresses this issue. In particular, we aim to integrate arbitrary data transformation alongside data retrieval, including grouping on different criteria on different relations at the same time.

Currently, returning a subdatabase is limited to data – specifically, the relevant tuples from a subset of the queried tables. A natural next step would be to expand this approach to include additional information such as metadata, statistics, indexes and the query execution plan. For instance, including the query execution plan could eliminate the need for users to implement the post-join logic manually. In general, expanding the data or information that is returned could unlock entirely new possibilities in query processing.

5.4 Concluding Remarks

This thesis addresses three specific issues with well-established solutions in query processing and indexing, offering novel perspectives and ideas to reframe and solve these issues. While we outline several interesting and promising directions for future research within each project, we also advocate for a critical examination of other areas in database systems by challenging traditional views and techniques.

Bibliography

[Agt+16] Erik Agterdenbos et al. "Empirical evaluation of guarded structural indexing." In: *Proceedings of the 19th International Conference on Extending Database Technology, EDBT*. 2016, pp. 714–715. URL: https://doi.org/10.5441/002/edbt.2016.101.

[Als+14] Sattam Alsubaiee et al. "AsterixDB: A Scalable, Open Source BDMS." In: *Proc. VLDB Endow.* 7.14 (2014), pp. 1905–1916. DOI: 10.14778/2733085.2733096. URL: http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf.

[Alv+15] Victor Alvarez et al. "A comparison of adaptive radix trees and hash tables." In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1227–1238. DOI: 10.1109/ICDE.2015.7113370. URL: https://doi.org/10.1109/ICDE.2015.7113370.

[Aou+05] Kamel Aouiche et al. "Automatic Selection of Bitmap Join Indexes in Data Warehouses." In: *Data Warehousing and Knowledge Discovery, 7th International Conference, DaWaK 2005, Copenhagen, Denmark, August 22-26, 2005, Proceedings*. Ed. by A Min Tjoa and Juan Trujillo. Vol. 3589. Lecture Notes in Computer Science. Springer, 2005, pp. 64–73. DOI: 10.1007/11546849_7. URL: https://doi.org/10.1007/11546849%5C_7.

[Ara+14] Bahareh Arab et al. "A Generic Provenance Middleware for Queries, Updates, and Transactions." In: *6th Workshop on the Theory and Practice of Provenance, TaPP'14, Cologne, Germany, June 12-13, 2014*. Ed. by Adriane Chapman, Bertram Ludäscher, and Andreas Schreiber. USENIX Association, 2014. URL: https://www.usenix.org/conference/tapp2014/agenda/presentation/arab.

[Are+22] Marcelo Arenas et al. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[Arg95] Lars Arge. "The Buffer Tree: A New Technique for Optimal I/O-Algorithms (Extended Abstract)." In: *Algorithms and Data Structures, 4th International Workshop, WADS '95, Kingston, Ontario, Canada, August 16-18, 1995, Proceedings*. Vol. 955. Lecture Notes in Computer Science. Springer, 1995, pp. 334–345. DOI: 10.1007/3-540-60220-8_74. URL: https://doi.org/10.1007/3-540-60220-8%5C_74.

[Bar+97] Elena Baralis et al. "Materialized Views Selection in a Multidimensional Database." In: *VLDB* (1997). URL: http://www.vldb.org/conf/1997/P156.PDF.

[Bat+07] Hardik Bati et al. "A genetic approach for random testing of database systems." In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. 2007, pp. 1243–1251. URL: http://www.vldb.org/conf/2007/papers/industrial/p1243-bati.pdf.

[BC81] Philip A. Bernstein and Dah-Ming W. Chiu. "Using Semi-Joins to Solve Relational Queries." In: *J. ACM* 28.1 (1981), pp. 25–40. DOI: 10.1145/322234.322238. URL: https://doi.org/10.1145/322234.322238.

[BCM22] Karl Bringmann, Nofar Carmeli, and Stefan Mengel. "Tight Fine-Grained Bounds for Direct Access on Join Queries." In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Leonid Libkin and Pablo Barceló. ACM, 2022, pp. 427–436. DOI: 10.1145/3517804.3526234. URL: https://doi.org/10.1145/3517804.3526234.

[Ber+01] Jochen Van den Bercken et al. "XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries." In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001, pp. 39–48. URL: http://www.vldb.org/conf/2001/P039.pdf.

[Ber+81] Philip A. Bernstein et al. "Query Processing in a System for Distributed Databases (SDD-1)." In: *ACM Trans. Database Syst.* 6.4 (1981), pp. 602–625. DOI: 10.1145/319628.319650. URL: https://doi.org/10.1145/319628.319650.

[BFI91] Kristin P. Bennett, Michael C. Ferris, and Yannis E. Ioannidis. "A Genetic Algorithm for Database Query Optimization." In: *Proceedings of the 4th International Conference on Genetic Algorithms*. 1991, pp. 400–407.

[BG21] Maximilian Bandle and Jana Giceva. "Database Technology for the Masses: Sub-Operators as First-Class Entities." In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2483–2490. DOI: 10.14778/3476249.3476296. URL: http://www.vldb.org/pvldb/vol14/p2483-bandle.pdf.

[Bin18] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. https://github.com/tlx/tlx, accessed November 8, 2021. 2018.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. "The Skyline Operator." In: *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*. Ed. by Dimitrios Georgakopoulos and Alexander Buchmann. IEEE Computer Society, 2001, pp. 421–430. DOI: 10.1109/ICDE.2001.914855. URL: https://doi.org/10.1109/ICDE.2001.914855.

[Blo70] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors." In: *Commun. ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[BM72] Rudolf Bayer and Edward M. McCreight. "Organization and Maintenance of Large Ordered Indices." In: *Acta Informatica* 1 (1972), pp. 173–189. DOI: 10.1007/BF00288683. URL: https://doi.org/10.1007/BF00288683.

[Bod96] Hans L. Bodlaender. "A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth." In: *SIAM J. Comput.* 25.6 (1996), pp. 1305–1317. DOI: 10.1137/S0097539793251219. URL: https://doi.org/10.1137/S0097539793251219.

[Bon+05] Peter A. Boncz et al. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR* (2005). URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

[BS01] Jochen Van den Bercken and Bernhard Seeger. "An Evaluation of Generic Bulk Loading Techniques." In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001, pp. 461–470. URL: http://www.vldb.org/conf/2001/P461.pdf.

[BSW97] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. "A Generic Approach to Bulk Loading Multidimensional Index Structures." In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997, pp. 406–415. URL: http://www.vldb.org/conf/1997/P406.PDF.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes*. Ed. by Gene Altshuler, Randall Rustin, and Bernard D. Plagman. ACM, 1974, pp. 249–264. DOI: 10.1145/800296.811515. URL: https://doi.org/10.1145/800296.811515.

[CI98] Chee Yong Chan and Yannis E. Ioannidis. "Bitmap Index Design and Evaluation." In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. Ed. by Laura M. Haas and Ashutosh Tiwary. ACM Press, 1998, pp. 355–366. DOI: 10.1145/276304.276336. URL: https://doi.org/10.1145/276304.276336.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[Cod71] E. F. Codd. "Further Normalization of the Data Base Relational Model." In: *Research Report / RJ / IBM / San Jose, California* RJ909 (1971).

[Cod72] E. F. Codd. "Relational Completeness of Data Base Sublanguages." In: *Research Report / RJ / IBM / San Jose, California* RJ987 (1972).

[Cra+20] Miles D. Cranmer et al. "Discovering Symbolic Models from Deep Learning with Inductive Biases." In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/c9f2f917078bd2db12f23c3b413d9cba-Abstract.html.

[Cro21] Andrew Crotty. "Hist-Tree: Those Who Ignore It Are Doomed to Learn." In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021%5C_paper20.pdf.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. "Tracing the lineage of view data in a warehousing environment." In: *ACM Trans. Database Syst.* 25.2 (2000), pp. 179–227. DOI: 10.1145/357775.357777. URL: https://doi.org/10.1145/357775.357777.

[Dat84] C. J. Date. "A Critique of the SQL Database Language." In: *SIGMOD Rec.* 14.3 (1984), pp. 8–54. DOI: 10.1145/984549.984551. URL: https://doi.org/10.1145/984549.984551.

[DCN20] Bailu Ding, Surajit Chaudhuri, and Vivek R. Narasayya. "Bitvector-aware Query Optimization for Decision Support Queries." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier et al. ACM, 2020, pp. 2011–2026. DOI: 10.1145/3318464.3389769. URL: https://doi.org/10.1145/3318464.3389769.

[Din+20] Jialin Ding et al. "ALEX: An Updatable Adaptive Learned Index." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 969–984. DOI: 10.1145/3318464.3389711. URL: https://doi.org/10.1145/3318464.3389711.

[Dit+02] Jens-Peter Dittrich et al. "Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm." In: *VLDB* (2002). DOI: 10.1016/B978-155860869-6/50034-2. URL: http://www.vldb.org/conf/2002/S09P03.pdf.

[DKK05] Jens-Peter Dittrich, Donald Kossmann, and Alexander Kreutz. "Bridging the Gap between OLAP and SQL." In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm et al. ACM, 2005, pp. 1031–1042. URL: http://www.vldb.org/archives/website/2005/program/paper/tue/p1031-dittrich.pdf.

[DN20] Jens Dittrich and Joris Nix. "The Case for Deep Query Optimisation." In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p3-dittrich-cidr20.pdf.

[DNS21] Jens Dittrich, Joris Nix, and Christian Schön. "The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures." In: *Proc.*

VLDB Endow. 15.3 (2021), pp. 527–540. DOI: 10.14778/3494124.3494136. URL: http://www.vldb.org/pvldb/vol15/p527-dittrich.pdf.

[DT07] David DeHaan and Frank Wm. Tompa. "Optimal top-down join enumeration." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. Ed. by Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou. ACM, 2007, pp. 785–796. DOI: 10.1145/1247480.1247567. URL: https://doi.org/10.1145/1247480.1247567.

[Fag+79] Ronald Fagin et al. "Extendible Hashing - A Fast Access Method for Dynamic Files." In: *ACM Trans. Database Syst.* 4.3 (1979), pp. 315–344. DOI: 10.1145/320083.320092. URL: https://doi.org/10.1145/320083.320092.

[FG89] Farshad Fotouhi and Carlos E. Galarce. "Genetic Algorithms and the Search for Optimal Database Index Selection." In: *Computing in the 90's, The First Great Lakes Computer Science Conference*. Vol. 507. Lecture Notes in Computer Science. 1989, pp. 249–255. URL: https://doi.org/10.1007/BFb0038500.

[Fle+09] George H. L. Fletcher et al. "A methodology for coupling fragments of XPath with structural indexes for XML documents." In: *Inf. Syst.* 34.7 (2009), pp. 657–670. URL: https://doi.org/10.1016/j.is.2008.09.003.

[FM11] Pit Fender and Guido Moerkotte. "A new, highly efficient, and easy to implement top-down join enumeration algorithm." In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. Ed. by Serge Abiteboul et al. IEEE Computer Society, 2011, pp. 864–875. DOI: 10.1109/ICDE.2011.5767901. URL: https://doi.org/10.1109/ICDE.2011.5767901.

[Fre+20] Michael J. Freitag et al. "Adopting Worst-Case Optimal Joins in Relational Database Systems." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904. URL: http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf.

[FV20] Paolo Ferragina and Giorgio Vinciguerra. "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds." In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1162–1175. URL: http://www.vldb.org/pvldb/vol13/p1162-ferragina.pdf.

[GMS09] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. "Generalized hypertree decompositions: NP-hardness and tractable variants." In: *J. ACM* 56.6 (2009), 30:1–30:32. DOI: 10.1145/1568318.1568320. URL: https://doi.org/10.1145/1568318.1568320.

[Gra+93] Goetz Graefe et al. "The Volcano Optimizer Generator: Extensibility and Efficient Search." In: *ICDE* (1993). DOI: 10.1109/ICDE.1993.344061. URL: https://doi.org/10.1109/ICDE.1993.344061.

[Gra+96] Jim Gray et al. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total." In: *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*. Ed. by Stanley Y. W. Su. IEEE Computer Society, 1996, pp. 152–159. DOI: 10.1109/ICDE.1996.492099. URL: https://doi.org/10.1109/ICDE.1996.492099.

[Gra04] Goetz Graefe. "Write-Optimized B-Trees." In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento et al. Morgan Kaufmann, 2004, pp. 672–683. DOI: 10.1016/B978-012088469-8.50060-7. URL: http://www.vldb.org/conf/2004/RS18P2.PDF.

[Gra08] John Grant. "Null values in SQL." In: *SIGMOD Rec.* 37.3 (2008), pp. 23–25. DOI: 10.1145/1462571.1462575. URL: https://doi.org/10.1145/1462571.1462575.

[Gra93] Goetz Graefe. "Query Evaluation Techniques for Large Databases." In: *ACM Comput. Surv.* 25.2 (1993). DOI: 10.1145/152610.152611. URL: https://doi.org/10.1145/152610.152611.

[GUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - the Complete Book (International Edition)*. Pearson Education, 2002.

[HD23a] Immanuel Haffner and Jens Dittrich. "A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries." In: *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*. Ed. by Julia Stoyanovich et al. OpenProceedings.org, 2023, pp. 1–13. DOI: 10.48786/EDBT.2023.01. URL: https://doi.org/10.48786/edbt.2023.01.

[HD23b] Immanuel Haffner and Jens Dittrich. "Efficiently Computing Join Orders with Heuristic Search." In: *Proc. ACM Manag. Data* 1.1 (2023), 73:1–73:26. DOI: 10.1145/3588927. URL: https://doi.org/10.1145/3588927.

[HD23c] Immanuel Haffner and Jens Dittrich. "mutable: A Modern DBMS for Research and Fast Prototyping." In: *13th Conference on Innovative Data Systems Research,*

CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023. www.cidrdb.org, 2023. URL: https://www.cidrdb.org/cidr2023/papers/p41-haffner.pdf.

[Hel+02] Joseph M. Hellerstein et al. "On a model of indexability and its bounds for range queries." In: *J. ACM* 49.1 (2002), pp. 35–55. URL: https://doi.org/10.1145/505241.505244.

[Hel+97] Joseph M. Hellerstein et al. "Online Aggregation." In: *SIGMOD* (1997). DOI: 10.1145/253260.253291. URL: https://doi.org/10.1145/253260.253291.

[Hip] Richard Hipp. *SQLite: Most Widely Deployed and Used Database Engine*. https://www.sqlite.org/mostdeployed.html. Accessed: 2024-07-26.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. "Generalized Search Trees for Database Systems." In: *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*. Morgan Kaufmann, 1995, pp. 562–573. URL: http://www.vldb.org/conf/1995/P562.PDF.

[Hol75] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1975.

[Idr+18a] Stratos Idreos et al. "The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models." In: *SIGMOD* (2018). DOI: 10.1145/3183713.3199671. URL: https://doi.org/10.1145/3183713.3199671.

[Idr+18b] Stratos Idreos et al. "The Periodic Table of Data Structures." In: *IEEE Data Eng. Bull.* 41.3 (2018). URL: http://sites.computer.org/debull/A18sept/p64.pdf.

[Idr+19a] Stratos Idreos et al. "Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn." In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf.

[Idr+19b] Stratos Idreos et al. "Learning Data Structure Alchemy." In: *IEEE Data Eng. Bull.* 42.2 (2019), pp. 47–58. URL: http://sites.computer.org/debull/A19june/p47.pdf.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Database Cracking." In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 68–78. URL: http://cidrdb.org/cidr2007/papers/cidr07p07.pdf.

[JG23] Michael Jungmair and Jana Giceva. "Declarative Sub-Operators for Universal Data Processing." In: *Proc. VLDB Endow.* 16.11 (2023), pp. 3461–3474. DOI: 10.14778/3611479.3611539. URL: https://www.vldb.org/pvldb/vol16/p3461-jungmair.pdf.

[Ker+05] Martin L. Kersten et al. "Cracking the Database Store." In: *CIDR* (2005). URL: http://cidrdb.org/cidr2005/papers/P18.pdf.

[Ker+18] Timo Kersten et al. "Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask." In: *PVLDB* 11.13 (2018). DOI: 10.14778/3275366.3275370. URL: http://www.vldb.org/pvldb/vol11/p2209-kersten.pdf.

[Kim+10] Changkyu Kim et al. "FAST: fast architecture sensitive tree search on modern CPUs and GPUs." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010, pp. 339–350. DOI: 10.1145/1807167.1807206. URL: https://doi.org/10.1145/1807167.1807206.

[Kip+19] Andreas Kipf et al. "SOSD: A Benchmark for Learned Indexes." In: *CoRR* abs/1911.13014 (2019). arXiv: 1911.13014. URL: http://arxiv.org/abs/1911.13014.

[Kip+20] Andreas Kipf et al. "RadixSpline: a single-pass learned index." In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*. ACM, 2020, 5:1–5:5. DOI: 10.1145/3401071.3401659. URL: https://doi.org/10.1145/3401071.3401659.

[KLN21] André Kohn, Viktor Leis, and Thomas Neumann. "Building Advanced SQL Analytics From Low-Level Plan Operators." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li et al. ACM, 2021, pp. 1001–1013. DOI: 10.1145/3448016.3457288. URL: https://doi.org/10.1145/3448016.3457288.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. "Concurrency and Recovery in Generalized Search Trees." In: *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 1997, pp. 62–72. DOI: 10.1145/253260.253272. URL: https://doi.org/10.1145/253260.253272.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN: 0-201-03803-X.

[Koc13] Christoph Koch. "Abstraction without regret in data management systems." In: *CIDR* (2013). URL: http://cidrdb.org/cidr2013/Papers/CIDR13%5C_Paper149.pdf.

[Koc14] Christoph Koch. "Abstraction Without Regret in Database Systems Building: a Manifesto." In: *IEEE Data Eng. Bull.* 37.1 (2014). URL: http://sites.computer.org/debull/A14mar/p70.pdf.

[Kor+04] Marcin Korytkowski et al. "Genetic Algorithm for Database Indexing." In: *Artificial Intelligence and Soft Computing - ICAISC*. Vol. 3070. Lecture Notes in Computer Science. 2004, pp. 1142–1147. URL: https://doi.org/10.1007/978-3-540-24844-6%5C_179.

[Kos+20] Jan Kossmann et al. "Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2382–2395. URL: http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf.

[Kra+18] Tim Kraska et al. "The Case for Learned Index Structures." In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 2018, pp. 489–504. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

[Kwo+23] Yongsik Kwon et al. "DB+-tree: A new variant of B+-tree for main-memory database systems." In: *Inf. Syst.* 119 (2023), p. 102287. DOI: 10.1016/J.IS.2023.102287. URL: https://doi.org/10.1016/j.is.2023.102287.

[Lam+12] Andrew Lamb et al. "The Vertica Analytic Database: C-Store 7 Years Later." In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1790–1801. DOI: 10.14778/2367502.2367518. URL: http://vldb.org/pvldb/vol5/p1790%5C_andrewlamb%5C_vldb2012.pdf.

[Lan+19] Harald Lang et al. "Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput." In: *Proc. VLDB Endow.* 12.5 (2019), pp. 502–515. DOI: 10.14778/3303753.3303757. URL: http://www.vldb.org/pvldb/vol12/p502-lang.pdf.

[Lei+13] Viktor Leis et al. "The adaptive radix tree: ARTful indexing for main-memory databases." In: *ICDE* (2013). DOI: 10.1109/ICDE.2013.6544812. URL: https://doi.org/10.1109/ICDE.2013.6544812.

[Lei+14] Viktor Leis et al. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *SIGMOD* (2014). DOI: 10.1145/2588555.2610507. URL: https://doi.org/10.1145/2588555.2610507.

[Lei+15] Viktor Leis et al. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (2015), pp. 204–215. DOI: 10.14778/2850583.2850594. URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

[LL71] Vincent Y. Lum and Huei Ling. "An Optimization Problem on the Selection of Secondary Keys." In: *Proceedings of the 1971 26th Annual Conference*. ACM '71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 349–356. ISBN: 9781450374842. DOI: 10.1145/800184.810505. URL: https://doi.org/10.1145/800184.810505.

[Loh88] Guy M. Lohman. "Grammar-like Functional Rules for Representing Query Optimization Alternatives." In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*. Ed. by Haran Boral and Per-Åke Larson. ACM Press, 1988, pp. 18–27. DOI: 10.1145/50202.50204. URL: https://doi.org/10.1145/50202.50204.

[Mar+20] Ryan Marcus et al. "Benchmarking Learned Indexes." In: *Proc. VLDB Endow.* 14.1 (2020), pp. 1–13.

[MN11] Guido Moerkotte and Thomas Neumann. "Accelerating Queries with Group-By and Join by Groupjoin." In: *Proc. VLDB Endow.* 4.11 (2011), pp. 843–851. URL: http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf.

[Moe] Guido Moerkotte. *Building Query Compilers*. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf. Accessed: 2019-12-16.

[Moe+06] Guido Moerkotte et al. "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products." In: *VLDB* (2006). URL: http://dl.acm.org/citation.cfm?id=1164207.

[Moe+08] Guido Moerkotte et al. “Dynamic programming strikes back.” In: *SIGMOD* (2008). DOI: 10.1145/1376616.1376672. URL: https://doi.org/10.1145/1376616.1376672.

[ND25] Joris Nix and Jens Dittrich. “Extending SQL to Return a Subdatabase.” In: *Proc. ACM Manag. Data* 3.3 (2025). DOI: 10.1145/3725291. URL: https://doi.org/10.1145/3725291.

[Neu+18] Thomas Neumann et al. “Adaptive Optimization of Very Large Join Queries.” In: *SIGMOD* (2018). DOI: 10.1145/3183713.3183733. URL: https://doi.org/10.1145/3183713.3183733.

[Neu+19] Priscilla Neuhaus et al. “GADIS: A Genetic Algorithm for Database Index Selection (S).” In: *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE*. 2019, pp. 39–54. URL: https://doi.org/10.18293/SEKE2019-135.

[Neu11] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware.” In: *PVLDB* 4.9 (2011). DOI: 10.14778/2002938.2002940. URL: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[Ngo+12] Hung Q. Ngo et al. “Worst-case optimal join algorithms: [extended abstract].” In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini. ACM, 2012, pp. 37–48. DOI: 10.1145/2213556.2213565. URL: https://doi.org/10.1145/2213556.2213565.

[Niu+17] Xing Niu et al. “Provenance-Aware Query Optimization.” In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 2017, pp. 473–484. DOI: 10.1109/ICDE.2017.104. URL: https://doi.org/10.1109/ICDE.2017.104.

[NK15] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries.” In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*. Ed. by Thomas Seidl et al. Vol. P-241. LNI. GI, 2015, pp. 383–402. URL: https://dl.gi.de/handle/20.500.12116/2418.

[NW09] Thomas Neumann and Gerhard Weikum. “Scalable join processing on very large RDF graphs.” In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June*

29 - July 2, 2009. Ed. by Ugur Çetintemel et al. ACM, 2009, pp. 627–640. DOI: 10.1145/1559845.1559911. URL: https://doi.org/10.1145/1559845.1559911.

[OPV14] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. "The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases." In: *CoRR* abs/1405.3631 (2014). arXiv: 1405.3631. URL: http://arxiv.org/abs/1405.3631.

[OQ97] Patrick E. O'Neil and Dallan Quass. "Improved Query Performance with Variant Indexes." In: *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. Ed. by Joan Peckham. ACM Press, 1997, pp. 38–49. DOI: 10.1145/253260.253268. URL: https://doi.org/10.1145/253260.253268.

[OS16] Dan Olteanu and Maximilian Schleich. "Factorized Databases." In: *SIGMOD Rec.* 45.2 (2016), pp. 5–16. DOI: 10.1145/3003665.3003667. URL: https://doi.org/10.1145/3003665.3003667.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. "Extensible/Rule Based Query Rewrite Optimization in Starburst." In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*. Ed. by Michael Stonebraker. ACM Press, 1992, pp. 39–48. DOI: 10.1145/130283.130294. URL: https://doi.org/10.1145/130283.130294.

[Pic+14] François Picalausa et al. "Principles of Guarded Structural Indexing." In: *Proc. 17th International Conference on Database Theory (ICDT)*. 2014, pp. 245–256. URL: https://doi.org/10.5441/002/icdt.2014.26.

[Pra19] Elvis Pranskevichus. *We Can Do Better Than SQL*. https://www.edgedb.com/blog/we-can-do-better-than-sql. Accessed: 2023-11-28. 2019.

[PSS09] Felix Putze, Peter Sanders, and Johannes Singler. "Cache-, hash-, and space-efficient bloom filters." In: *ACM J. Exp. Algorithmics* 14 (2009). DOI: 10.1145/1498698.1594230. URL: https://doi.org/10.1145/1498698.1594230.

[Rea+20] Esteban Real et al. "Automl-zero: Evolving machine learning algorithms from scratch." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 8007–8019.

[Ric+15] Stefan Richter et al. "A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing." In: *PVLDB* 9.3 (2015). DOI: 10.14778/2850583.2850585. URL: http://www.vldb.org/pvldb/vol9/p96-richter.pdf.

[RR00] Jun Rao and Kenneth A. Ross. "Making B^+-Trees Cache Conscious in Main Memory." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 2000, pp. 475–486. DOI: 10.1145/342009.335449. URL: https://doi.org/10.1145/342009.335449.

[RR99] Jun Rao and Kenneth A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Morgan Kaufmann, 1999, pp. 78–89. URL: http://www.vldb.org/conf/1999/P7.pdf.

[RS86] Neil Robertson and Paul D. Seymour. "Graph Minors. II. Algorithmic Aspects of Tree-Width." In: *J. Algorithms* 7.3 (1986), pp. 309–322. DOI: 10.1016/0196-6774(86)90023-4. URL: https://doi.org/10.1016/0196-6774(86)90023-4.

[Sch+13] Felix Martin Schuhknecht et al. "The Uncracked Pieces in Database Cracking." In: *PVLDB* 7.2 (2013). DOI: 10.14778/2732228.2732229. URL: http://www.vldb.org/pvldb/vol7/p97-schuhknecht.pdf.

[Sch+18] Felix Martin Schuhknecht et al. "Adaptive Adaptive Indexing." In: *ICDE* (2018). DOI: 10.1109/ICDE.2018.00066. URL: https://doi.org/10.1109/ICDE.2018.00066.

[Sel+79] Patricia G. Selinger et al. "Access Path Selection in a Relational Database Management System." In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*. ACM, 1979, pp. 23–34. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. "k-ary search on modern processors." In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*. ACM, 2009, pp. 52–60. DOI: 10.1145/1565694.1565705. URL: https://doi.org/10.1145/1565694.1565705.

[Shr+13] Lakshmikant Shrinivas et al. “Materialization strategies in the Vertica analytic database: Lessons learned.” In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, 2013, pp. 1196–1207. DOI: 10.1109/ICDE.2013.6544909. URL: https://doi.org/10.1109/ICDE.2013.6544909.

[SQLFlaws14] *Sql Flaws*. https://wiki.c2.com/?SqlFlaws. Accessed: 2023-11-28. 2014.

[Szá+22] Gábor Szárnyas et al. “The LDBC Social Network Benchmark: Business Intelligence Workload.” In: *Proc. VLDB Endow.* 16.4 (2022), pp. 877–890. DOI: 10.14778/3574245.3574270. URL: https://www.vldb.org/pvldb/vol16/p877-szarnyas.pdf.

[Tab21] Lloyd Tabb. *Malloy*. https://www.malloydata.dev. Accessed: 2024-06-27. 2021.

[TI94] Odysseas G. Tsatalos and Yannis E. Ioannidis. “A Unified Framework for Indexing in Database Systems.” In: *Database and Expert Systems Applications, 5th International Conference, DEXA*. Vol. 856. Lecture Notes in Computer Science. 1994, pp. 183–192. URL: https://doi.org/10.1007/3-540-58435-8%5C_183.

[TSI96] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. “The GMAP: A Versatile Tool for Physical Data Independence.” In: *VLDB J.* 5.2 (1996), pp. 101–118. URL: https://doi.org/10.1007/s007780050018.

[TSV18] Toni Taipalus, Mikko T. Siponen, and Tero Vartiainen. “Errors and Complications in SQL Query Formulation.” In: *ACM Trans. Comput. Educ.* 18.3 (2018), 15:1–15:29. DOI: 10.1145/3231712. URL: https://doi.org/10.1145/3231712.

[Vel12] Todd L. Veldhuizen. “Leapfrog Triejoin: a worst-case optimal join algorithm.” In: *CoRR* abs/1210.0481 (2012). arXiv: 1210.0481. URL: http://arxiv.org/abs/1210.0481.

[VM96] Bennet Vance and David Maier. “Rapid Bushy Join-order Optimization with Cartesian Products.” In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. Ed. by H. V. Jagadish and Inderpal Singh Mumick. ACM Press, 1996, pp. 35–46. DOI: 10.1145/233269.233317. URL: https://doi.org/10.1145/233269.233317.

[Wan+16] Wei Wang et al. “Database Meets Deep Learning: Challenges and Opportunities.” In: *SIGMOD Record* 45.2 (2016).

[WWS23] Yisu Remy Wang, Max Willsey, and Dan Suciu. “Free Join: Unifying Worst-Case Optimal and Traditional Joins.” In: *Proc. ACM Manag. Data* 1.2 (2023), 150:1–150:23. DOI: 10.1145/3589295. URL: https://doi.org/10.1145/3589295.

[Yan+24] Yifei Yang et al. “Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries.” In: *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. URL: https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf.

[Yan81] Mihalis Yannakakis. “Algorithms for Acyclic Database Schemes.” In: *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 1981, pp. 82–94.

[Zhu+17] Jianqiao Zhu et al. “Looking Ahead Makes Query Plans Robust.” In: *Proc. VLDB Endow.* 10.8 (2017), pp. 889–900. DOI: 10.14778/3090163.3090167. URL: http://www.vldb.org/pvldb/vol10/p889-zhu.pdf.