Universität des Saarlandes

Optimization and Processing of Relational Database Queries

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

by

Immanuel Haffner

Saarbrücken, 2024

Abstract

The purpose of this Ph.D. thesis is to advance the state of the art of query processing in relational database systems.

As first contribution, I present a reduction of the classical, NP-hard join order optimization problem to a shortest path problem. Consequently, I develop a heuristic search algorithm for solving this reduced problem. I provide a strong theoretical foundation for the reduction and the search. A thorough evaluation shows improvements in optimization time of orders of magnitude.

My second contribution is a simplified design for query execution by just-in-time compilation to machine code. Architecturally simple solutions such as query compilation with LLVM suffer from unacceptably high compilation times. Modern just-in-time query compilers with significantly reduced compilation times, on the other side, are extravagantly hand-crafted. Rather than reinventing compiler technology in a DBMS, I propose to embed an off-the-shelf just-in-time compiling engine that is designed, built, and tested by compiler experts. I am able to achieve the lowest compilation times and competitive execution performance in all experiments.

As my third contribution, I design a relational DBMS as a software system that is composed of individual components, each implementing an isolated logical task. For example, join ordering is one such component of a composable DBMS. I describe the design principles that guide my and my colleagues' efforts towards implementing such a DBMS. Most importantly, my thesis is accompanied by the release of our open-source research DBMS mu*t*able, that implements this design of composition.

Zusammenfassung

Ziel dieser Dissertation ist es, den Stand der Technik bei der Verarbeitung von Anfragen in relationalen Datenbanksystemen zu verbessern.

Als ersten Beitrag präsentiere ich eine Reduktion des klassischen, NP-schweren Optimierungsproblems der Join-Reihenfolge auf ein Kürzester-Pfad-Problem. Folglich entwickle ich einen heuristischen Suchalgorithmus zur Lösung dieses reduzierten Problems. Ich biete eine starke theoretische Grundlage für die Reduktion und die Suche. Eine gründliche Auswertung zeigt eine Verbesserung der Optimierungszeit um Größenordnungen.

Mein zweiter Beitrag ist ein vereinfachtes Design für die Ausführung von Anfragen durch just-in-time Kompilierung in Maschinencode. Architektonisch einfache Lösungen wie die Anfragekompilierung mit LLVM leiden unter inakzeptabel hohen Kompilierungszeiten. Moderne just-in-time Anfrage-Compiler mit deutlich reduzierten Kompilierzeiten sind dagegen aufwendig selbst entwickelt. Anstatt die Compilertechnologie in einem DBMS neu zu erfinden, schlage ich vor, eine gebrauchsfertige just-in-time Compiler-Engine einzubinden, die von Compiler-Experten entworfen, gebaut und getestet wurde. Ich bin in der Lage, in allen Experimenten die niedrigsten Kompilierzeiten und eine konkurrenzfähige Ausführungsleistung zu erzielen.

In meinem dritten Beitrag entwerfe ich ein relationales DBMS als ein Softwaresystem, das aus einzelnen Komponenten zusammengesetzt ist, von denen jede eine isolierte logische Aufgabe implementiert. Zum Beispiel ist die Join-Anordnung eine solche Komponente eines zusammensetzbaren DBMS. Ich beschreibe die Designprinzipien, die mich und meine Kollegen bei der Implementierung eines solchen DBMS leiten. Am wichtigsten ist, dass meine Arbeit von der Veröffentlichung unseres Open-Source-Forschungs-DBMS mu*t*able begleitet wird, das dieses Design der Zusammensetzung implementiert.

Acknowledgement

I thank my Ph.D. advisor Prof. Dr. Jens Dittrich for introducing me to the fascinating topic of database systems during his core lecture, for offering me a researcher position at his chair, for supervising my research for many years, for providing me with insightful feedback through which I was able to improve the quality of my work, for reviewing all my scientific work, and finally for reviewing my Ph.D. thesis.

I thank Prof. Dr. Sebastian Hack for supervising and reviewing both my Bachelor's and my Master's thesis, for introducing me to scientific work, for teaching me thoroughly about compilers (which was of significant value for my thesis), and for being a reliable source of information and guidance. I also thank him for being second reviewer of my Ph.D. thesis.

I thank Prof. Dr. Guido Moerkotte for his research and publications, which were always a valuable source of information during my thesis, for his extensive book "Building Query Compilers"[1], which came in handy several times for fact checking and validation, and for being third and external reviewer of my Ph.D. thesis.

In addition to thanking my reviewers, I would like to thank all the other wonderful people who supported me in my pursuit of a Ph.D. degree. While I cannot mention all of them here, I want to explicitly thank a few that provided exceptional support.

Special thanks are due to my family, that always believed in me and motivated me in pursuing a Ph.D. degree. I especially thank my grandmother Christel for finding calming and conciliatory words whenever a heated debate about politics was about to divide the family. Your world-class cooking was also a great relief from Mensa cuisine and provided for excellent recreation.

With great joy, I can say that my coworkers during my time as a Ph.D. student also became dear friends. I want to thank them for being such kind and supportive colleagues. You have made work and research a fantastic experience.

I thank my friends for bearing with me through this time, even though I was frequently

[1]At the time of writing this thesis, this document is still a draft.

indisposed and couldn’t participate in activities as often as I would have liked.

Last but not least, I thank Prof. Dr. Roland Leißa, who was the primary supervisor of my Bachelor’s and Master’s theses. From him, I learned to love (and hate) . He introduced me to C++ and taught me good style in programming and software design by having me work on LLVM and Clang, and pedantically reviewing my work. The hours we spent together thinking about and writing code contributed significantly to the skills I have today.

Contents

Chapter 1

Introduction

In today's software landscape, *database management systems* (DBMSs) are an indispensable component of the software stack. They are found in virtually every application or device. SQLite, for example, is delivered by default on Android, iOS, and Windows from Windows 10 onwards – and this list is not exhaustive. All major browsers ship with built-in DBMSs and most Android/iOS apps manage their data within a DBMS. Large businesses manage millions to billions of records of data – e.g. user information, goods in stock, or sensor measurements – with the help of large-scale and sometimes distributed DBMSs.

To understand why DBMSs take such a crucial role in software systems and what difficult tasks they solve for their users, I will briefly survey the history of DBMSs and highlight the motivations that led to their development and success. I will then present some of the most difficult tasks DBMSs have to solve. Along this brief survey, I will outline the contributions of my thesis, that advance the current *state of the art* (SotA) of DBMSs.

1.1 A Brief History of Database Management Systems

Simultaneously to the mass fabrication of *integrated circuits* in the 60s and 70s and the advent of silicon computer chips in the 70s, the term '*database*' began to appear in literature. This temporal correlation can be observed in Figure 1.1, depicting the frequency of occurrences in literature over the past decades. We can see that the term 'database' was steadily used more often in the 70s and gained a lot of momentum during the 80s and 90s. The first use of the term in a technical sense can be dated back to 1962, according to the Oxford English Dictionary [OED]. To understand that this correlation is in fact causal, let us look up the definition of 'database' in the Oxford English Dictionary: a database is "a structured set of data held in a computer, especially one that is accessible in various ways" [OED]. This definition is arguably so generic,

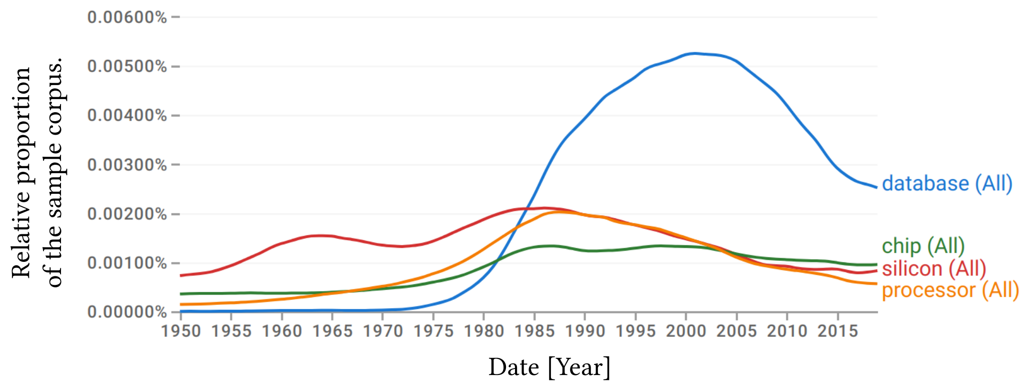


Figure 1.1: The Google *Ngram* of the word 'database' (case-insensitive).

that there is hardly any purposeful software that *does not* operate on a database of some form. As a natural consequence, the dawn of the computer age, which created an entire industry of software development, was thus also the dawn of databases.

With virtually every software using databases in one form or another, operating databases became a very frequently recurring task. This fact necessitated the development of reusable and generalized software to work with databases. DBMSs were born! With the first general-purpose DBMSs becoming popular and even being used commercially, an interest in a standard for such systems started to emerge. The *Database Task Group* (DBTG) was founded in 1967 to devise such a standard. In 1971, they delivered their standard, that then led to the development of *navigational* DBMSs and became known as the *CODASYL model* – that is because DBTG was founded within CODASYL. Navigational DBMSs expose their internal organization of data to the user and provide means to "navigate" through the database, e.g. from one record to another or from a key attribute to the respective record. A drawback of the navigational approach is that it leaks – to some extent – the internal representation of the data. More so, the logical process of querying a database must be intermixed with navigational commands to retrieve all required records. In 1970, E. F. Codd raised concerns about the CODASYL model and advocated for a strict separation between the DBMS user and the DBMS' internal organization of data. To this end – and in stark contrast to the navigational model – Codd presented the *relational* model in his seminal work "A Relational Model of Data for Large Shared Data Banks", which should become the foundation of relational database management and spark a vast area of research for generations to come [Cod70].

Students		
matriculation	name	major
1234567	Charles W. Bachman	Mech. Eng.
1234568	Edgar F. Codd	Math
1234571	Jim Gray	Math & Stats
1234572	Michael Stonebraker	Mech. Eng.

Subjects	
tag	full name
Mech. Eng.	Mechanical Engineering
Math	Mathematics
Math & Stats	Mathematics & Statistics

Table 1.1: Two tables of ‘Student’ and ‘Subject’ entities. The underlined attributes ‘matriculation’ and ‘tag’ are the primary keys of the respective tables.

1.2 The Relational Model in a Nutshell

In the relational model, data is organized in relations of entities, with one relation per type of entity. Each relation contains a fixed number of attributes storing the attributes of the entities. Each tuple of a relation must be *uniquely* identified (within this relation) by its *primary key*, which is a composition of one or more attributes of the entity. Such primary keys are a means to user-oriented identifiers of records. In contrast to the navigational model, where primary keys are system-defined artificial values and relationships between records are expressed through physical properties such as disk addresses, relationships in the relational model are expressed through primary keys composed of attributes of entities: For two related relations, one relation will be extended by artificial attribute(s) storing primary keys *of the related relation*, thereby expressing the relationship between the entities of these relations. Such an attribute referencing a primary key is called a *foreign key*. Queries to the database can then join two relations based on this relationship expressed through the foreign key. Consider, for example, the two relations in Table 1.1. We can see that the attribute ‘tag’, that is the primary key of relation ‘Subjects’, is referenced by foreign key attribute ‘major’ of relation ‘Students’. Consider further the following natural language query.

“Which students major in ‘Mechanical Engineering’?” (Q_1)

Query Q_1 can be answered by joining the relations and selecting only those tuples where the ‘full name’ is ‘Mechanical Engineering’, yielding ‘Charles W. Bachman’ and ‘Michael Stonebraker’.

An important aspect of the relational model is that it is based on a strong mathematical foundation. Relations and operations thereon build on set theory and are formalized in relational calculus and relational algebra. There are two variants of relational calculus:

1. The tuple relational calculus was created by Codd to provide a declarative query language

for the relational model [Cod70; Cod72].

2. The domain relational calculus, that is more akin to first-order logic, was later proposed by Lacroix and Pirotte. Together with Codd they showed that it has expressive power equivalent to the domain-independent tuple relational calculus and relational algebra [LP77].

For example, we can express Q_1 in relational calculus as

$$\left\{ s \;\middle|\; \begin{array}{c} s \in \mathit{Students} \wedge u \in \mathit{Subjects} \wedge \\ s_{[\text{Students.major}]} = u_{[\text{Subjects.tag}]} \wedge \\ u_{[\text{Subjects.full name}]} = \text{‘Mechanical Engineering’} \end{array} \right\}$$

The representation of queries in relational calculus is almost a *normal form.* All queries have the shape $\{\langle X_1, \ldots, X_n \rangle \mid p\,(\langle X_1, \ldots, X_n \rangle)\}$, where X_i are attributes or constants and p is a boolean formula. If we required p to have normal form, e.g. *conjunctive normal form* (CNF), then we would have a normal form for queries.

To express Q_1 in relational algebra, we have multiple choices, as the following expressions (A)-(D) demonstrate.[1]

$$\sigma_{\text{full name = ‘Mechanical Engineering’}}\left(\sigma_{\text{major = tag}}\left(\mathit{Students} \times \mathit{Subjects}\right)\right) \tag{A}$$

$$\sigma_{\text{full name = ‘Mechanical Engineering’} \wedge \text{major = tag}}\left(\mathit{Students} \times \mathit{Subjects}\right) \tag{B}$$

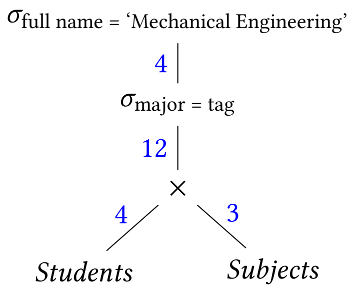
$$\sigma_{\text{full name = ‘Mechanical Engineering’}}\left(\mathit{Students} \bowtie_{\text{major = tag}} \mathit{Subjects}\right) \tag{C}$$

$$\mathit{Students} \bowtie_{\text{major = tag}}\left(\sigma_{\text{full name = ‘Mechanical Engineering’}}\left(\mathit{Subjects}\right)\right) \tag{D}$$

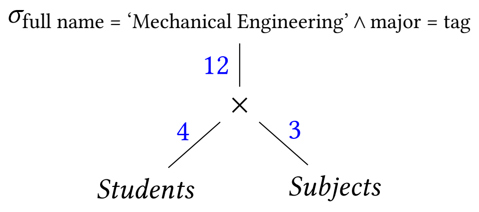
We can clearly see that relational algebra queries can be formulated quite differently despite being semantically equivalent. The key distinguishing factor between relational calculus and relational algebra is that relational algebra expresses the operators to use and a (partial) order of evaluation by which to compute the query result. The order of evaluation is induced by the precedence of the relational operators. In laymen's terms, a query formulated in relational calculus expresses *what* to compute whereas a query formulated in relational algebra expresses *how* to compute the query result.

We know that the relational algebra expressions (A)-(D) are semantically equivalent, i.e. always computing the same result, yet computing this result in different ways. This begs the question: When and how does the order of computation matter? To exemplify the impact of different algebraic expressions for the same query, I depict in Figure 1.2 the expressions (A)-(D) as trees representing the induced order of evaluation. I annotate these trees with the

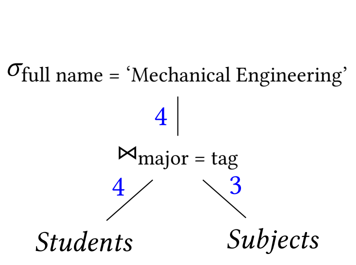
[1] To keep the expressions short and easier to read, we completely omit projections.



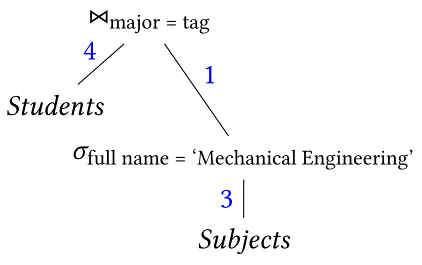
Query plan (A) with $\Sigma = 23$.



Query plan (B) with $\Sigma = 19$.



Query plan (C) with $\Sigma = 11$.



Query plan (D) with $\Sigma = 8$.

Figure 1.2: Query plans (A)-(D), represented as trees and annotated with the sizes of intermediate results.

sizes of the intermediate results in blue. We can observe that the different expressions yield different amounts of intermediate tuples. If we consider that these expressions are actually used to steer the computations of the query results, then we can argue that the amount of intermediate tuples produced is directly or indirectly proportional to the amount of work required to compute the query result. From now on, we refer to expressions steering the computation of a query result as *query plans* (or simply *plans*). To make the prior argument more concrete, we conduct a detailed comparison of plans (A) and (C) – that is the plans induced by expressions (A) and (C). Consider first in (A) the Cartesian product $\mathit{Students} \times \mathit{Subjects}$ yielding 12 intermediate tuples that are the input to the selection $\sigma_{\text{major = tag}}$. Computing the Cartesian product takes time $O(|\mathit{Students}| \cdot |\mathit{Subjects}|)$. The result of the Cartesian product, i.e. 12 tuples, is then processed further by the selection. Let us compare this to the join $\mathit{Students} \bowtie_{\text{major = tag}} \mathit{Subjects}$ in expression (C). For joins on equality predicates we know efficient algorithms that perform in time

$$O(|\mathit{Students}| + |\mathit{Subjects}| + |\mathit{Students} \bowtie_{\text{major = tag}} \mathit{Subjects}|)$$

Further, we know that $|\mathit{Students} \bowtie_{\text{major = tag}} \mathit{Subjects}| = |\mathit{Students}|$ because attribute major is a foreign key to primary key tag. We derive

$$\begin{aligned} & O\left(|Students| + |Subjects| + |Students \bowtie_{\text{major = tag}} Subjects|\right) \\ = {} & O\left(|Students| + |Subjects|\right) \end{aligned}$$

Consequently, the join asymptotically reduces the amount of work done in (C) compared to (A). Very importantly, this asymptotic improvement is reflected in the number of intermediate tuples produced. The amount of intermediate tuples of a query plan is hence a strong indicator of the amount of work necessary to compute the query result by this plan. For the example in Figure 1.2 we can argue that plan (D) is likely the most efficient of the four plans, with nearly just $\frac{1}{3}$ the amount of the intermediate tuples of (A). While this relative improvement may appear small, keep in mind that the example uses relations of 4 and 3 tuples, respectively. With larger tables, the *relative* improvement of (C) and (D) over (A) would grow; (C) and (D) produce *asymptotically* fewer intermediate tuples.

1.3 Towards Query Optimization

My example in Figure 1.2 demonstrates that different yet semantically equivalent expressions in relational algebra yield different intermediate results and intermediate result sizes can be related to the work required to answer a query. Consequently, when relational algebra is used to steer the computation of a query result, then the choice of the query plan significantly affects the performance of query answering. This leads us to the question: Is there a "best" or "optimal" query plan to compute the result of a query, and can it be computed (efficiently)? Before we can answer this question, we must first define optimality. The definition commonly accepted by researchers is that optimality is relative to some *cost model* and the optimal plan is the one (or one of many) that minimizes the cost under said cost model. With this definition of optimality, our quest for an optimal plan for a query becomes a classical optimization problem: Given some query Q and a cost model C, choose from the set of all relational algebra expressions that are semantically equivalent to Q, say R_Q, the expression minimizing C. Formally,

$$\mathop{\mathbf{arg\,min}}_{r \in R_Q} C(r)$$

The process of finding such a plan r is commonly called *query optimization*. At this point, I would like to emphasize that Codd already recognized in 1972 the optimization potential for queries [Cod72]. He states that a query expressed in relational calculus is an ideal starting point for optimization. In contrast to that, he further argues that when queries are expressed in relational algebra, the “properties of desired data tend to get hidden in the particular operation sequence” and the semantics of the query are thereby obfuscated [Cod72]. Consequently, when tasked with optimizing a query expressed in relational algebra, one can content oneself with locally optimizing the expression – e.g. through algebraic rewrites – and potentially missing the globally best plan. Alternatively, to guarantee finding the best plan eventually, one must enable transformations that deteriorate the cost of the plan in order to *escape* local optima. However, allowing for such transformations introduces difficult problems into the optimization process, as the following approach exemplifies.

1.3.1 Rule-Based Transformative Query Optimization

In the work of Graefe and DeWitt [GD87], the authors build a transformative query optimizer with rewrite rules on relational algebra. All transformations are directed and assigned an improvement factor. A factor < 1 means a cost reduction is achieved through transformation. A factor of 1 means no change in cost; this is used to have the optimizer exploit commutativity of joins. A factor > 1 means a cost increase, i.e. the plan becomes more costly. At a first glance, one might argue that transformations with factors > 1 should generally be avoided. However, this would get the optimizer stuck in local optima, make the search result depend heavily on the initial plan where search starts, and it can have the optimizer miss the global optimum. Therefore, the authors include transformations that deteriorate the plan cost in order to escape local optima. However, this leads to another problem: transformations could now be applied cyclically, repeatedly reproducing the same plans over and over. To remedy this problem, memoization is added to track plans that have already been generated. This then leads to the next problem: the memory demand for memoization is too high, causing early optimization aborts that yield suboptimal plans. To work around this limitation, the authors restrict the optimizer to apply only transformations with a factor less than a certain threshold, e.g. < 1.05. This threshold causes the optimizer to produce significantly fewer plans and converge faster towards an optimum. However, the optimization is not guaranteed to find the global optimum anymore, which the authors also validate empirically.

We conclude that optimal transformative query optimization is practically infeasible because of the memory requirements for memoization. It also has the undesired property that the optimization process depends heavily on the initial plan where the search begins. On the one side, a plan that is “far” from an optimal plan leads to long optimization times. On the other side, if optimization is not optimal, the choice of the initial plan affects what final plan is used to answer the query; and this has unexpected and undesired effects on query runtime. To circumvent these innate problems of transformative query optimization, alternative approaches

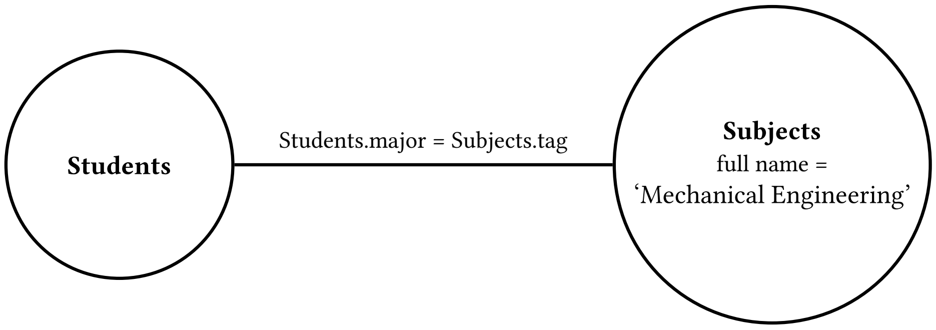


Figure 1.3: Query graph for query Q_1.

have been studied. The most prominent, most successful, and widely in use alternative being query optimization based on combinatorics.

1.3.2 Combinatorial Query Optimization

Instead of starting from an initial plan, as in transformative query optimization, we can start with a query formulated in relational calculus. Recall, that relational calculus is almost a normal form and that there is no order among operations. Hence, starting from the formulation in relational calculus makes optimization independent of any initial plan or formulation of the query by a user.

In combinatorial query optimization, the relational calculus query is represented as a graph with relations as vertices and join predicates as edges connecting the joined relations. The relations are additionally annotated with selection predicates applying to the relation. We call this representation of a query the *query graph*. Figure 1.3 shows the query graph for our very simple query Q_1. The query graph maintains the important property of relational calculus of not defining an order among the join operations. This can better be seen in the example in Figure 1.4, showing a more complex query graph G with four relations connected by four join predicates and no order among the joins.

Recall, that query plans, i.e. relational algebra expressions, define a partial order among the joins. With the query graph at hand, we can reinterpret a query plan as a *complete recursive cut* of the query graph, that is a recursive application of graph cuts such that eventually each relation forms its own partition. To demonstrate this, consider the following plan P for the query graph G in Figure 1.4.

$$(A \bowtie_{p_1} B) \bowtie_{p_2 \wedge p_4} (C \bowtie_{p_3} D) \tag{P}$$

The top-level join $\bowtie_{p_2 \wedge p_4}$ of P cuts G into the two subgraphs induced by the subsets of relations

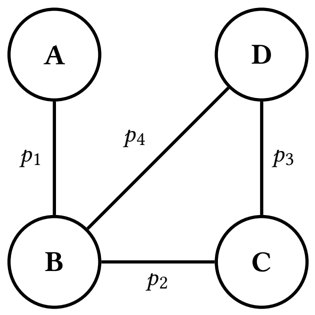


Figure 1.4: A more complex query graph G with four relations A, B, C, D and four join predicates p_1, p_2, p_3, p_4.

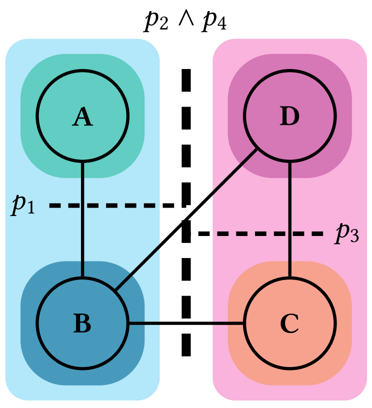


Figure 1.5: Partition of graph G by recursive graph cutting according to plan P.

$\{A, B\}$ and $\{C, D\}$, formally $G_{[\{A,B\}]}$ and $G_{[\{C,D\}]}$, respectively. These two subgraphs are further cut recursively. Join $\bowtie_{p_1}$ cuts $G_{[\{A,B\}]}$ into $G_{[\{A\}]}$ and $G_{[\{B\}]}$. Likewise, join $\bowtie_{p_3}$ cuts $G_{[\{C,D\}]}$. Figure 1.5 shows the partition of graph G of Figure 1.4 via recursive cuts by plan P.

We now have established an understanding of a query plan, or more precisely the joins within the plan, as a graph partition via recursive graph cuts. If we can systematically and efficiently enumerate all such recursive cuts of a given query graph, we will be able to enumerate all different join orders. From these join orders we can then construct query plans, hence allowing us to systematically and exhaustively explore the space of alternative plans.

So far, we neglected selections and Cartesian products. Selections, i.e. predicates on a single relation, are generally applied as early as possible in a plan, i.e. directly to the relation and before any join. Selections on multiple relations are always treated as join predicates and hence lead to join operations in the plan. Cartesian products are often ignored by combinatorial query optimization. Practice has shown that plans with Cartesian products are almost always suboptimal and considering them significantly expands the search space for alternative plans [Sel+79; MN06]. However, there are particular cases where Cartesian products are desirable to consider [CM95; NR18]. Including Cartesian products into combinatorial query optimization is very easy: to consider a Cartesian product of two relations with no join connecting them, we connect these relations by an edge with predicate *True*. To generally allow for Cartesian products in combinatorial query optimization, the query graph can be augmented by edges with *True* predicates until it is a complete graph.

It is also important to note, that it is sufficient to consider only cuts, i.e. partitions into exactly two subsets, out of all possible graph partitions. That is, because joins and Cartesian products are binary operations, combining two input relations to one result relation. Should we want to consider n-ary operations, e.g. n-way joins or n-way Cartesian products, then we must also consider the respective graph partitions.

We have now reformulated the search for the optimal query plan to a combinatorial partition problem of the query graph. As we build the query graph from the relational calculus formulation of the query, this approach is independent of any initial plan. This is an improvement over transformative query optimization as done by Graefe and DeWitt [GD87] and presented in Section 1.3.1. However, there is another very significant improvement. Graefe and DeWitt face the problem of exhausting the available memory with their transformative query optimizer when allowing for optimality-degrading transformations (factor > 1) [GD87]. This is because they must memoize every plan ever seen in order to prevent cyclically regenerating the same plans indefinitely. The amount of distinct plans for a query is *at least* the amount of different join orders. For n relations, the amount of different join orders is $n!$ when only linear plans are considered, with linear plans being trees where each inner node is parent to at least one leave. If bushy plans are considered as well, then the amount of join orders becomes $n! \cdot C_{n-1}$ in the worst case, where C_i is the i-th Catalan number:

$$C_n := \frac{1}{n+1}\binom{2n}{n} \qquad \text{(Catalan numbers)}$$

Either way, the amount of different join orders for n relations is *superexponential* in n. Hence, optimal transformative query optimization (i.e. with any factor allowed) will require memory for memoization superexponential in the number of relations, which is practically infeasible. Contrary to that, enumerating all join orders by enumerating all recursive graph cuts of the query graph can be done with dynamic programming and requiring *just* exponentially much memory, i.e. $\mathcal{O}(2^n)$ for n relations.

1.3.3 Query Optimization by Dynamic Programming

Enumerating all recursive graph cuts can be done efficiently and without producing duplicates. Even better, it can be done in an order that permits dynamic programming. Two properties must be satisfied for dynamic programming to be applicable:

(1) *Optimal substructure:* A problem can be solved optimally by dividing it into smaller subproblems and solving these subproblems optimally.

(2) *Overlapping subproblems:* A problem can be broken down into subproblems which are reused several times.

These two properties are satisfied in our recursive graph cutting problem and I will elaborate this along Figure 1.6.

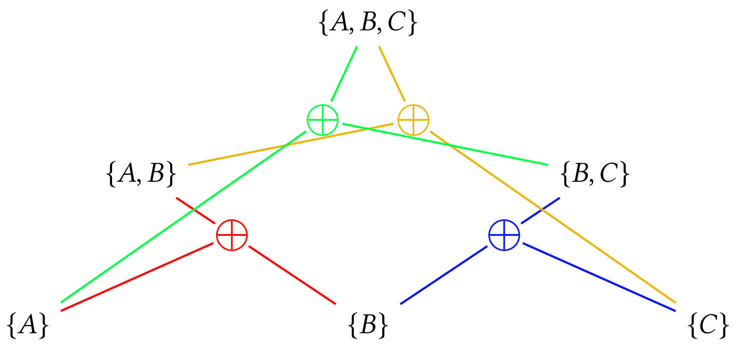


Figure 1.6: Recursive decomposition of subproblem $\{A, B, C\}$ into smaller subproblems as done by dynamic programming. Operator $\bigoplus$ shows where dynamic programming combines the optimal solutions of subproblems to compute a solution of a larger problem.

With regard to (1), we can make the following observation. For computing the optimal plan for $\{A, B, C\}$, we can enumerate all cuts of $G_{[\{A,B,C\}]}$, here $(\{A, B\}, \{C\})$ and $(\{A\}, \{B, C\})$. These two partitions yield four subsets $\{A\}$, $\{C\}$, $\{A, B\}$, and $\{B, C\}$. We can compute the optimal plan for each subset recursively. From the solutions to these subproblems, we can construct two candidate plans for $\{A, B, C\}$: we can join the best plan for $\{A, B\}$ with the best plan for $\{C\}$ and we can join the best plan for $\{A\}$ with the best plan for $\{B, C\}$. The better of those two plans is the best plan for $\{A, B, C\}$.

With regard to (2), we can observe that, for example, subproblem $\{A\}$ occurs twice as a subproblem: once for $\{A, B, C\}$ and once for $\{A, B\}$. Consequently, the optimal solution for $\{A\}$ – albeit being trivial in this example – is reused multiple times.

The graph partitions can be enumerated by dynamic programming in various orders, depending on the particular algorithm. For example, the algorithm by Selinger et al. [Sel+79] first enumerates all partitions of induced subgraphs with two relations, then all partitions of induced subgraphs with three relations, and so on until eventually all partitions for the entire query graph are enumerated. We generally distinguish two classes of dynamic programming algorithms for enumerating all graph partitions, namely *bottom-up* and *top-down* algorithms. Algorithms in the bottom-up class enumerate and optimally solve subproblems in an ascending order, starting with individual relations and then advancing to ever more complex subproblems. Algorithms in the top-down class start with the entire query graph and then recursively cut it until all relations form individual partitions. When all alternative cuts of a subproblem have been explored and solved optimally, an optimal solution for that subproblem is found.

One important aspect distinguishing the two classes is that bottom-up algorithms produce complete recursive graph cuts relatively late in their execution. That is, because many small subproblems are solved optimally before the first partition of the entire query graph is ever produced. In contrast to that, top-down algorithms find complete recursive graph cuts occasionally during the entire execution. That is, because top-down algorithms enumerate all cuts of the entire graph in the outermost level of recursion and every such cut leads to one complete recursive graph cut. Therefore, and quite naturally, top-down algorithms are susceptible to pruning. The idea is that partial recursive graph cuts, that are more costly than the best complete graph cut found yet, need not be recursively cut any further as they cannot yield any better plan.

For historical reasons, the bottom-up algorithms are commonly referred to as dynamic programming algorithms, abbreviated DP, whereas the top-down algorithms are really called just that and abbreviated TD. The reason is that the first algorithms performing combinatorial query optimization all fall into the bottom-up class [Sel+79; VM96; MN06]. Already back in 1979, when Selinger et al. [Sel+79] presented the first algorithm for combinatorial query optimization, their algorithm was performing bottom-up dynamic programming.

Even though the first solutions to query optimization already appeared in the late 1970s, this problem has kept researchers busy to that very day. Besides transformative and combinatorial query optimization, other approaches have been developed over the past decades. These alternatives build on genetic algorithms [BFI91; Vel08; FWY08], greedy algorithms [Feg98; WP00; Neu09; NR18], adaptive re-optimization [KD98; Ng+99; WNS16; Per+19], or machine learning [Mar+19; Mar+21; Neg+21]. For the special case of *provably optimal* join ordering, however, no other feasible solution than combinatorial join ordering via dynamic programming has been found yet.

In Chapter 2, I present a reduction of join order optimization to a shortest path problem and, consequently, how to solve it efficiently with heuristic search. Particular configurations of this heuristic search yield provably optimal join orders while efficiently pruning the search space, thereby finding optimal solutions several orders of magnitude faster than current SOTA.

1.4 From Query Plan to Query Execution

In Section 1.3, I introduced the abstract concept of a cost function for query plans, and consequently I motivated query optimization for finding a query plan that minimizes such a cost function. The cost function is usually designed to correlate to execution time, but other optimization goals such as maximum memory consumption or time-to-first-tuple are also possible. With query optimization, we are able to compute for a given query an optimal plan, i.e. a plan that minimizes the cost function. My contribution in Chapter 2 enables us to compute such an optimal plan more efficiently for a broad spectrum of queries. But what are we going to do

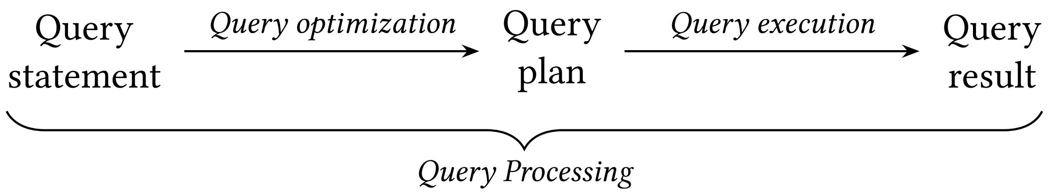


Figure 1.7: The phases of query processing, from query statement to query result.

with the optimal query plan? How can we actually execute this query plan? And, staying in the mindset of striving for optimal efficiency, how can we make the execution *efficient*?

This is the task of query execution. In literature, one often finds the term "query compilation" in this context. Generally, compilation describes the process of translating a program in a source language to an equivalent program in a target language. In most cases, when one speaks of compilation, translation of a program in a high-level language to an executable machine-code program is meant. Compilation often includes performing efficiency-increasing program transformations, also called optimizations. In the context of database queries, query compilation refers to the process of translating a query statement to a query plan. This process can involve an optimization phase, as explained in Section 1.3, to increase the efficiency of the resulting query plan. The term "query compilation" can, however, be ambiguous and lead to misunderstandings. When the query plan is being compiled further in order to execute it, e.g. to machine code, then the overall processing of a query involves another compilation step. In this case, does "query compilation" still only describe translating from query statement to query plan or does it now involve the subsequent compilation of the query plan? To avoid any potential ambiguities, I will use the nomenclature given in Figure 1.7.

The various methods to query execution can be dissected into just two classes, namely *interpreting* and *compiling* methods. Interpreting methods employ an interpreter that directly executes the operations of a given query plan to compute the query result. Compiling methods compile a given query plan into an executable program (e.g., machine code) and execute that program in order to compute the query result. These two methods are not mutually exclusive. However, combinations of interpretation and compilation in the context of query execution have been explored only very recently, e.g. by Kohn, Leis, and Neumann [KLN18] and Kersten, Leis, and Neumann [KLN21] and by myself together with my Ph.D. advisor Jens Dittrich [HD23a].

Both interpretation and compilation have their individual benefits and drawbacks. The very first relational DBMS, System R, already employed compilation of SQL queries to machine code [Sel+79]. This approach was later abandoned in favor of interpreting query plans, as compiling query plans directly to machine code requires a lot of development and maintenance effort and requires additional work when targeting new architectures [TER18]. Interpreting query plans only requires a single implementation of an interpreter in a high-level language, that can then be compiled once per target architecture. That is why interpreting query plans was the dominant procedure for executing query plans for a long time. This paradigm shifted towards compiling query plans when main memory DBMSs appeared. With all or most of the frequently accessed (hot) data held in main memory, data accesses suddenly became much faster than in prior disk-based systems. The interpretational overheads, that were dwarfed by costs for disk I/O, suddenly take a significant share in query execution times [Pad+01; BZN05a; Rao+06]. This development led to a comprehensive body of work aiming to improve CPU core and cache utilization [Ail+99; Pad+01; BMK09]. A part of these works pursued compiling entire queries into a sequence of tight loops in some low-level executable language, e.g. machine code or Java bytecode. There are two major benefits to compiling queries: First, interpretational overheads, e.g. dynamic dispatches based on operand types, can be eliminated at compilation time. Second, operator boundaries are eliminated, thereby *fusing* sequences of operators into single tight loops and enabling passing data in registers rather than sending data between operators through memory.

While the benefits of compiling queries to machine code might sound very convincing at first glance, there is an undeniable downside to this approach: Queries must be compiled before they can be executed, thereby delaying query execution. Query compilation can therefore potentially deteriorate performance, e.g. by increasing the time-to-first-tuple or – even worse – by increasing query processing time, if the compilation delay outweighs the performance improvement achieved over interpreted execution. While initial works on query compilation focused on maximizing the performance of the compiled query [KVC10; Dia+13; LZF13; FIL14; Klo+14], follow-up works acknowledged the downside of compilation delays and investigated techniques to circumvent long compilation times [Neu11; KLN18; KLN21].

A critical observation about this recent line of research is that it is very similar to *just-in-time (JIT)* compilation, which is a very well-studied problem in the compiler community. In Chapter 3, I study how we can leverage the compiler community's results from decades of research on JIT compilation to build a compiling query execution engine with relatively low effort that is competitive with the most recent achievements in query compilation. In particular, my approach achieves low compilation times, high query execution performance, and adaptive (re-)optimization during query execution. While all these desirable properties have already been achieved in recent DBMSs, the benefit of my approach is that we can achieve these properties with ease by cleverly building on top of existing JIT compilation infrastructure.

1.5 A New DBMS for Research and Fast Prototyping

To evaluate my contributions, that I hinted at in the prior sections, I had to implement my approaches in some DBMS. Chronologically, my first project – that resulted in the paper that forms Chapter 3 – was on JIT compiling SQL queries to machine code as described in Section 1.4. When I was about to start this project, I searched for an open-source DBMS that would fit my purpose and enable me to implement my approach into that system. Surprisingly, there was only a single open-source DBMS that supported query compilation and code generation, namely NoisePage [22b]. After looking at NoisePage more closely, I realized that its design was very rigid; modifying it to my needs would be a very tedious task let alone the lack of documentation. This was the point where I decided to build a new DBMS – or at least those parts of a DBMS that I would need to realize my approach. Effectively, this meant building an SQL-to-machine-code compiler plus a storage layer. In retrospective, *not* using NoisePage turned out to be the right decision, as the project was discontinued in July 2021.

A few months into building this new DBMS, I was facing the problem of adding a storage layer to the just built SQL parser, semantic analysis, and compiler. By looking at open-source projects for inspiration, I realized that all DBMSs I found had a fixed storage layout that was baked into the system, e.g. a hard-coded row or PAX layout. After some discussion with my advisor about how to proceed, we came to the conclusion that we wanted to do things differently. We wanted to build a generic solution that can be adapted to different – hopefully any – layout with ease while still providing performance that is competitive to hard-coded layouts. This approach becomes feasible through the efficient code generation and the following efficient and effective compilation to machine code. In particular, we built a data structure for *describing* the desired data layout to the DBMS. The DBMS takes this description to generate at query compilation time the necessary code to access the data in the way that is described by our data structure. A very nice side effect of our approach is that it separates storage and data layout from the remainder of the system. Where other compiling systems expose internal data layouts in other components, e.g. query execution, our system achieves a clean separation of concerns. While a similar separation of concerns is achieved in DBMSs with interpreting query execution, we are able to eliminate any interpretational overheads through JIT compilation.

After successfully separating storage and data layout from the remainder of the system, in particular query execution, we decided that we want to achieve the same degree of separation of concerns and encapsulation for all parts of our DBMS. This lead to our abstract concept of building our DBMS as a composition of individual components. While this is not a new idea at all – and was already explored three decades ago by Batory et al. [Bat+88], Carey and DeWitt [CD87], and Carey et al. [Car+91] – we are able to overcome abstraction overheads through our approach of JIT compiling not only queries but also any code that is part of query execution, e.g. storage access. The fact that our DBMS is composable and hence can be configured – or *mutated* – on a fine granule to one's demands explains our choice for its name: mu*t*able, a play on words from *mutation* and *table* (to emphasize our focus on relational data).

Chapter 4 provides a more extensive motivation for building a new DBMS and our design. This section also dives deeper into our design considerations and the goals we have set for the project. Further, in that chapter, I provide an extensive list of components of mu*t*able, describe their interfaces, and provide examples.

Chapter 2

Query Optimization

This chapter is based on my publication "Efficiently Computing Join Orders with Heuristic Search" [HD23b]. This work was published in the research track of SIGMOD 2023. This work was co-authored by my Ph.D. advisor Prof. Dr. Jens Dittrich.

2.1 Introduction

The *Structured Query Language* (SQL) is the dominant programming language to query and transform relational data, that is usually stored in *(relational) database management systems* ((R)DBMS). SQL is a declarative language: it only expresses *what* to compute without specifying *how* to compute. This declarative style of expressing operations burdens a DBMS with determining a query execution plan (or simply query plan) that defines how the computations required by a query are done. A crucial part of determining a query plan is determining a join order, i.e. the order in which individual relations are joined by the respective join predicates of the query. The join order has a major impact on the performance of the query plan and hence it is of utmost importance to a DBMS to compute a *"good"* join order – or at least to avoid *"bad"* join orders [BC05; Lei+15]. This problem is known as the *join order optimization problem* (JOOP) and it is generally NP hard [IK84; CM95]. There exists a comprehensive body of work on computing join orders. It can be divided into work on computing optimal join orders [IK84; Sel+79; CM95; MN06; DT07; FM11a; FM11b], work on greedy computation of potentially sub-optimal join orders [Feg98; WP00; Neu09; NR18], work on adaptive re-optimization of join orders [KD98; Ng+99; WNS16; Per+19], and recent work based on machine learning [Mar+19; Mar+21; Neg+21].

Ono and Lohman [OL90] derive analytically the number of distinct plans w/o Cartesian products, showing that the amount of plans is generally exponential in the number of relations.

For queries with many relations, the search space of plans quickly becomes too large to explore exhaustively. DBMSs therefore define a threshold beyond which suboptimal but faster algorithms are used [NR18]. Interestingly though, optimal algorithms need not be exhaustive.

In the domain of AI planning, searching extremely large search spaces is a frequent task and research in that area has brought forth algorithms to efficiently explore such search spaces. An important class of such algorithms is *best-first search* (BFS). BFS enables efficiently finding optimal or nearly optimal solutions without exhaustively exploring the entire search space. It has proven itself useful in a wide range of applications [ES11; Sal17]. The question arises whether and how BFS can be applied to JOOP.

Contributions. In this work, we present a new approach to join order optimization that is based on heuristic search, an important subset of BFS. In particular, we make the following contributions.

1. To the best of our knowledge, we present the first formal reduction of JOOP to shortest path. We present formalizations for both bottom-up and top-down join ordering and investigate their dualism. (Section 2.2)

2. We define heuristic search, perform a theoretical analysis of heuristic search applied to our shortest path problem, and elaborate the general search procedure. (Section 2.3)

3. We present an efficient search space representation for both bottom-up and top-down search. Additionally, we devise two crucial optimizations, one of which is highly particular to the search space of JOOP. (Section 2.4)

4. We identify and circumvent a potential pitfall when incorporating a DBMS cost model into heuristic search, that severely limits the efficiency of the search. (Section 2.5)

5. We experimentally evaluate our approach and compare it to state-of-the-art algorithms. (Section 2.7)

6. We propose a new benchmark that systematically explores the Query Graph Exploration Landscape (QGraEL) along the three query graph dimensions number of relations, graph density, and edge distribution. (Section 2.7.3)

This chapter is organized in the order of contributions. We discuss related work in Section 2.6.

2.2 Join Order Optimization as a Shortest Path Problem

In this section, we formalize join order optimization as a shortest path problem. We begin with a brief excursion to shortest path and graph search. We then reduce bottom-up join order optimization to shortest path. We investigate the dualism of bottom-up and top-down join order optimization when expressed as a shortest path problem. Lastly, we analyze the time complexity of solving JOOP via shortest path.

2.2.1 The Shortest Path Problem

We formally define the shortest path problem on directed graphs as follows. Let $G := (V, E)$ be a graph with vertices V and directed, weighted edges $E \subseteq \{(u, v, w) \mid u, v \in V, w \in \mathbb{R}^+\}$. For an edge $e = (u, v, w)$, we call u the *tail*, v the *head*, and w the *weight* of e. A path $P = e_1 \dots e_k \in E^k$ is a sequence of edges with $\forall\, i \in \{1, \dots, k-1\}.\ head(e_i) = tail(e_{i+1})$. We say that P *starts* in $tail(e_1)$, *ends* in $head(e_k)$, and has length $|P| = k$. The weight of a path is defined as $weight(P) := \sum_{i=1}^{|P|} weight(e_i)$. Let $n_0, n_* \in V$ be the start and goal of a search problem, respectively, and further let $\mathbb{P}(n_0, n_*) := \{P \mid P \text{ starts in } n_0 \wedge P \text{ ends in } n_*\}$ be the set of all paths from n_0 to n_*. We then define the shortest path as

$$\mathop{\mathbf{arg\,min}}_{P \in \mathbb{P}(n_0, n_*)} weight(P) \tag{2.1}$$

A shortest path algorithm computes a solution for the shortest path problem, that is a shortest path algorithm computes for some $G := (V, E)$ and $n_0, n_* \in V$ a path P according to Definition 2.1.

2.2.2 Reducing JOOP to Shortest Path

We will now formulate join ordering as a shortest path problem. To do so, we need to formalize JOOP and then reduce it to shortest path. Note, that this requires a reduction from NP-hard JOOP to PTIME shortest path where the size of the search space for shortest path is exponential in the size of the query graph G_Q (in the worst case). We use the query graph in Figure 2.1 as a running example throughout this section. For some query Q, let $G_Q := (R, J)$ be the *query graph* of Q, with relations R as vertices and joins $J \subseteq \binom{R}{2}$[1] as edges. For Figure 2.1, we have $R = \{A, B, C, D\}$ and $J = \big\{\{A, B\}, \{B, C\}, \{B, D\}, \{C, D\}\big\}$. The

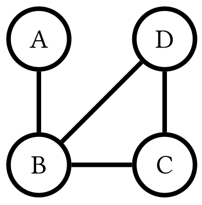


Figure 2.1: Example query graph.

[1]The notation $\binom{S}{k}$, read "from set S choose k", denotes all subsets of S of size k, i.e. $\{s \subseteq S \mid |s| = k\}$. Observe, that $\left|\binom{S}{k}\right| = \binom{|S|}{k}$.

goal of join order optimization is to order the joins in J such that the induced *plan* for Q has minimal cost. In this work, we restrict ourselves to binary, inner joins. We call a subset $S \subseteq R$ a *subproblem* of Q, e.g. $\{A, B, D\}$ is one subproblem of Q. We say that a join $j = \{r_1, r_2\} \in J$ joins two disjoint subproblems S_1, S_2 if $r_1 \in S_1 \wedge r_2 \in S_2$. For example, join $j_{BD} = \{B, D\} \in J$ joins subproblems $\{A, B\}$ and $\{D\}$, written $j_{BD}(\{A, B\}, \{D\})$. Every join $j \in J$ can hence be treated as a partial function $2^R \times 2^R \rightharpoonup 2^R$:

$$j : (S_1, S_2) \quad \mapsto \quad S_1 \uplus S_2 \quad \text{if} \quad j = \{r_1, r_2\} \wedge r_1 \in S_1 \wedge r_2 \in S_2$$

The precondition of j ensures that j is only applicable to two disjoint subproblems that are joinable by j. For example, $j_{BD}(\{A, B\}, \{D\}) = \{A, B, D\}$ while $j_{BD}(\{A, B\}, \{C\})$ is undefined. Note, that this definition of j requires specifying the two subproblems $S1, S2$ to join. However, we want to represent a query plan as a sequence of joins, i.e. without explicitly specifying for each join what subproblems are joined. We can leverage a join j's precondition to formulate a *hoisted* definition j^* that operates on sets of subproblems. In particular, j^* automatically selects from a set of pairwise disjoint subproblems $S_1, \dots, S_n$ the two subproblems S_i, S_k for which $j(S_i, S_k)$ is defined:

$$j^* : \{S_1, \dots, S_n\} \quad \mapsto \quad \Big(\{S_1, \dots, S_n\} \setminus \{S_i, S_k\}\Big) \cup \{j(S_i, S_k)\}$$

where $j = \{r_i, r_k\}$, $r_i \in S_i$, and $r_k \in S_k$. For example,

$$j^*_{BD}\big(\big\{\{A, B\}, \{C\}, \{D\}\big\}\big) = \big\{\{C\}\big\} \cup \big\{j_{BD}(\{A, B\}, \{D\})\big\}$$

With the hoisted definition of joins, we can define a plan as a sequence of all joins in J. Let $p = j_1 \dots j_{|J|}$ be a plan. Then $p(\mathcal{S}) := j^*_{|J|} \circ \dots \circ j^*_1(\mathcal{S})$ denotes the sequential application of joins to some set of subproblems $\mathcal{S}$. Any plan p joining all relations R of query Q, formally $p\left(\binom{R}{1}\right) = \{R\}$, is a feasible plan for Q. Note, that "feasible" effectively means "without Cartesian products". Consider, for example, the plan $p = j_{BD}\, j_{AB}\, j_{CD}\, j_{BC}$. We have

$$\begin{aligned} p(\binom{R}{1}) &= j^*_{BC} \circ j^*_{CD} \circ j^*_{AB} \circ j^*_{BD}\big(\big\{\{A\}, \{B\}, \{C\}, \{D\}\big\}\big) \\ &= j^*_{BC} \circ j^*_{CD} \circ j^*_{AB}\big(\big\{\{A\}, \{B, D\}, \{C\}\big\}\big) = j^*_{BC} \circ j^*_{CD}\big(\big\{\{A, B, D\}, \{C\}\big\}\big) \\ &= \big\{\{A, B, C, D\}\big\} = \{R\} \end{aligned}$$

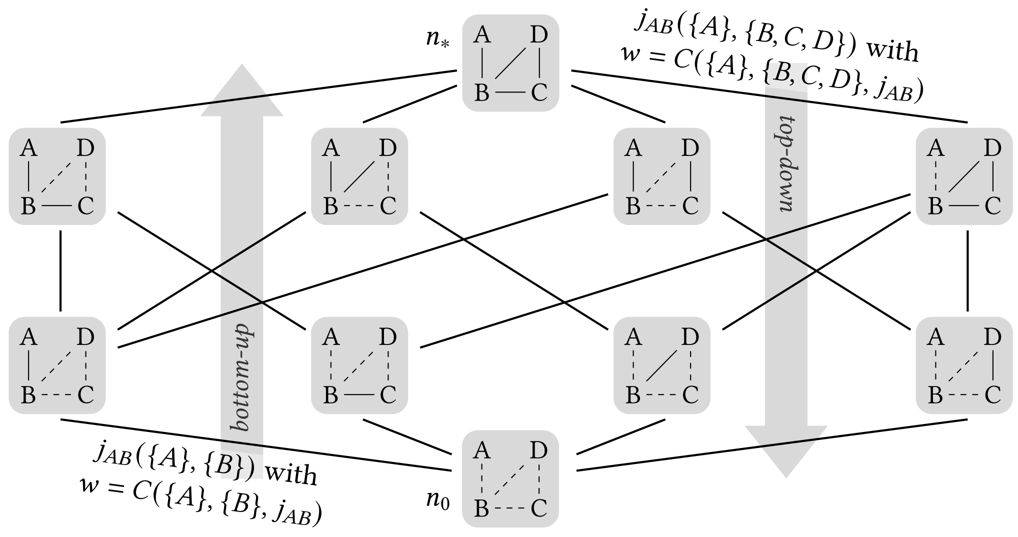

n_*
$j_{AB}(\{A\},\{B,C,D\})$ with
$w = C(\{A\},\{B,C,D\},j_{AB})$
top-down
bottom-up
$j_{AB}(\{A\},\{B\})$ with
$w = C(\{A\},\{B\},j_{AB})$
n_0
A
B
C
D

Figure 2.2: Search space for the shortest path. Sets are represented by their induced subgraph, e.g. n_0 is $\big\{\{A\},\{B\},\{C\},\{D\}\big\}$.

Observe, that query Q from Figure 2.1 is cyclic. Therefore, in this plan p, j_{BC} is subsumed[2] by applying both j_{BD} and j_{CD} beforehand. The goal of join order optimization according to our definition is to compute a feasible plan p of minimal cost.

We now reduce JOOP to shortest path. The idea of our reduction is that every application of a join j to two subproblems S_1, S_2 forms an edge in the search space for shortest path. The weight of this edge is the cost of performing this join. Search starts in the initial vertex $n_0 = \binom{R}{1}$. The search space consists of all vertices reachable from n_0 through successive application of the joins in J. Figure 2.2 shows the search space for the query graph in Figure 2.1. The search space is a directed graph, with edges directed *away* from n_0 (read bottom to top). The vertices of the search space are sets of subproblems yet to be joined together. Each subproblem of a vertex is drawn as a *connected subgraph* (csg) with solid edges; dashed edges represent joins not yet applied. Every path from the start $n_0 = \binom{R}{1}$ to the goal $n_* = \{R\}$ is a sequence of joins joining all relations in R and therefore a feasible plan according to our definition. Further, the weight of such a path equals the cost of the corresponding plan. We call the search space constructed by this reduction of JOOP to shortest path $\mathrm{SP}_{\mathrm{JOOP}}$. We can now solve JOOP by computing a shortest path according to 2.1 in $\mathrm{SP}_{\mathrm{JOOP}}$.

So far, we did not explain how weights are computed. It is fair to assume that a DBMS can provide a cost model to predict the cost of joining two subproblems $C : 2^R \times 2^R \times J \longrightarrow \mathbb{R}^+$. With

[2]Either j_{BC} degrades to a selection predicate or is applied simultaneously with j_{CD} by conjunction of j_{BC}'s and j_{CD}'s join predicates.

cost model C, we can define the weight of an edge as

$$weight\big((u,v)\big) := \min \left\{ C(S_1, S_2, j) \;\middle|\; \begin{array}{l} j \in J \land S_1, S_2 \in u \land \\ v = (u \setminus \{S_1, S_2\}) \cup \{j(S_1, S_2)\} \end{array} \right\}$$

A join subsumed by other joins can be evaluated in two ways: either by a join algorithm that supports a conjunction of multiple predicates or by a selection succeeding the subsuming joins. Either way, we expect C to compute the costs accordingly.

2.2.3 The Dualism of Bottom-Up and Top-Down Join Order Optimization

The reduction in Section 2.2.2 is for *bottom-up* join ordering: initially all relations are disjoint in n_0 and then joins are applied to join subproblems until *all* relations are joined together in n_*. In *top-down* join ordering, we start with all relations already joined together and *"undo"* joins until all relations are pairwise disjoint. Undoing joins means partitioning a subproblem S into smaller subproblems S_1, S_2 with $\exists\, j \in J.\ j(S_1, S_2) = S$. Observe in Figure 2.2 that *top-down* join ordering corresponds to a search starting in n_* with goal n_0 and edges directed *towards* n_0. The search space of top-down join ordering is dual to that of bottom-up join ordering. Hence, top-down join ordering is the dual problem of bottom-up join ordering.

2.2.4 Complexity of $\mathrm{SP}_{\mathrm{JOOP}}$

Join order optimization is well-known to be NP hard [IK84; CM95]. This means that solving JOOP requires time exponential in the size of the query graph G_Q. Since our reduction of JOOP to shortest path preserves optimality, solving JOOP by computing a shortest path in $\mathrm{SP}_{\mathrm{JOOP}}$, that is constructed by our *exponential-time* reduction in Section 2.2.2, must have worst-case time *exponential* in the size of the query graph. To prove that this is indeed the case, we give the following constructive argument.

Ono and Lohman [OL90] show that queries whose query graph $G_Q = (R, J)$ is a clique have $\Theta(3^{|R|})$ many *connected complement pairs (CCP)*, where a CCP is a pair of subproblems (S_1, S_2) s.t. $\exists\, j \in J.\ j(S_1, S_2)$ and S_1, S_2 induce csgs in G_Q. We show that for each CCP in G_Q there exists *at least* one vertex in $\mathrm{SP}_{\mathrm{JOOP}}$: For every CCP (S_1, S_2) in G_Q there exists at least one set of subproblems $\mathcal{S}$, s.t. $\mathcal{S}$ contains the CCP. Exactly one such $\mathcal{S}$ contains the CCP and otherwise only base relations, i.e. $\mathcal{S} = \big\{S_1, S_2\big\} \cup \binom{R \setminus (S_1 \cup S_2)}{1}$. This $\mathcal{S}$ is a vertex in $\mathrm{SP}_{\mathrm{JOOP}}$. Hence, $\mathrm{SP}_{\mathrm{JOOP}}$ has $|V| \in \Omega(3^{|R|})$ many vertices. Because every vertex in $\mathrm{SP}_{\mathrm{JOOP}}$ (except the goal) has at least one outgoing edge, there are $|E| \in \Omega(3^{|R|})$ many edges.

For computing a shortest path, we can choose from a broad set of shortest path algorithms. Because we are only interested in shortest paths from n_0 to n_*, our problem is the special case *single-pair* shortest path, with pair (n_0, n_*). Schrijver [Sch04] gives an extensive survey of shortest path algorithms. In the class of *uninformed* (or *blind*) search, algorithms only have information of the start n_0 and the search space (cf. Figure 2.2). This effectively means that the knowledge of n_* is of no use to uninformed search. Of this class of algorithms, even the asymptotically best have a worst-case time complexity that is at least linear in the size of the search space, i.e. $\Omega(|V|)$ or $\Omega(|E|)$. Since both $|V|$ and $|E|$ of $\mathrm{SP}_{\mathrm{JOOP}}$ are exponential in the size of the query graph G_Q (in the worst case), computing a shortest path in $\mathrm{SP}_{\mathrm{JOOP}}$ requires time exponential *in the size of* G_Q (in the worst case). However, the mentioning of uninformed search suggests that there must be *informed* search. Informed search, or *heuristic* search, has additional knowledge beyond the problem description, that allows for a goal-oriented search. We discuss this in the following Section 2.3.

2.3 JOOP as a Heuristic Search Problem

After reducing JOOP to $\mathrm{SP}_{\mathrm{JOOP}}$ in Section 2.2, we explore how to solve $\mathrm{SP}_{\mathrm{JOOP}}$ with heuristic search. We therefore extend search by a *heuristic function*. The heuristic function (or just heuristic) estimates for a given vertex in the search space the weight of a shortest path from that vertex to a goal. The heuristic enables the search to focus on vertices that it deems to lead to shorter paths. We can apply heuristic search to our shortest path problem if we can define a heuristic for our search space. We discuss important properties of heuristic functions in Section 2.3.1 and their impact on heuristic search in Section 2.3.2. We motivate that heuristic search enables us to gradually sacrifice optimality, in terms of plan cost, for efficiency. In Section 2.3.3, we then describe conceptually how we apply heuristic search to $\mathrm{SP}_{\mathrm{JOOP}}$ with a potentially exponentially large search space. We present proof sketches for completeness, soundness, and optimality in Section 2.3.4 and study different performance criteria of heuristic search in Section 2.3.5. We discuss how to find a heuristic for $\mathrm{SP}_{\mathrm{JOOP}}$ in Section 2.5.

2.3.1 Properties of Heuristic Functions

A heuristic function h estimates for some vertex v the weight of a shortest path from v to a goal. The optimal heuristic h^* returns for each vertex v *exactly* the weight of a shortest path from v to a goal. A heuristic h is *goal-aware* if the heuristic value of any goal is 0, formally v *is goal* $\Rightarrow h(v) = 0$. In $\mathrm{SP}_{\mathrm{JOOP}}$, checking whether a vertex is a goal is simple and we therefore assume all heuristics to be goal-aware. A heuristic *underestimates* if there exists a vertex for which the heuristic underestimates the weight of a shortest path to goal, i.e. $\exists v \in V.\ h(v) < h^*(v)$. Likewise, a heuristic *overestimates* if $\exists v \in V.\ h(v) > h^*(v)$. A heuristic that *never overestimates* is called *admissible*. Admissibility becomes important when we discuss optimality of heuristic search. A heuristic h is called *consistent* if the heuristic never overestimates the weight of a single edge, i.e. $\forall\,(u,v,w) \in E.\ h(u) \le h(v) + w$. Every consistent and goal-aware heuristic is also admissible and h^* is consistent.

2.3.2 Properties of Heuristic Search

To exploit a heuristic we need to perform heuristic search. In particular, we will focus on Dijkstra's algorithm [Dij+59] and famous A^* [HNR68]. There are two interesting properties of A^*, that we will mention here, as they will guide us when we design and evaluate heuristics.

Optimality. Algorithm A^* is optimal, that is it computes the shortest path from start to goal, if the heuristic h is admissible [HNR68].

Time Complexity. Dechter and Pearl [DP85] have shown that if the heuristic h is consistent, algorithm A^* is optimally efficient, i.e., there exists no BFS algorithm that finds a shortest path with traversing fewer vertices of the search space.

According to these two properties, if we are able to devise an admissible heuristic for SP_{JOOP}, we are guaranteed that A^* will find a shortest path, which corresponds to an optimal plan of JOOP. Further, if we are able to devise a consistent heuristic for SP_{JOOP} that is efficiently computable, i.e. in PTIME, we know that we can efficiently solve SP_{JOOP} (even if not in PTIME). A naïve attempt would be to devise an optimal heuristic for SP_{JOOP}. However, an optimal heuristic for SP_{JOOP} cannot be computed in PTIME:

Theorem 1. Unless P = NP, any optimal heuristic for SP_{JOOP} is not in PTIME, formally: $\forall\,h.\ \big(\forall\,v.\ h(v) = h^*(v)\big) \Rightarrow h \notin \text{PTIME}$.

We will prove Theorem 1 by contradiction, showing that if an optimal heuristic h were computable in PTIME, we could solve JOOP in PTIME, contradicting the fact that JOOP is NP hard [IK84; CM95]. Our proof relies on the conjecture P $\neq$ NP and on bounding the complexity of shortest path. With a depth d, where d is the minimal length of any path from n_0 to n_*, and a maximum branching factor b, the complexity of shortest path is in $O(b^d)$ [RN20]. In SP_{JOOP}, b is bounded by the number of joins $|J|$ and d is exactly $|R| - 1$. Hence, we can bound the time complexity by $O(|J|^{|R|-1})$. So far, this bound is not really helpful. However, for optimal h, b becomes 1. We prove this by contradiction (compare [ES11, Theorem 2.9 on p. 72]):

Lemma 1. Assume an edge $(u,v,w) \in E$ and further $\forall\,(u,v',w') \in E.\ h^*(v) + w \le h^*(v') + w'$. Then v lies on a shortest path from u to a goal.

Algorithm 1 BFS with on-demand search space computation.

```
function BFS($n_0$ : start vertex)
    $\mathcal{L} \leftarrow [n_0]$                                  ▷ initialize open list
    while $\mathcal{L}$ not empty do
        $u, g_u \leftarrow$ EXTRACT-BEST($\mathcal{L}$)             ▷ extract next best vertex with its cost
        if $u$ is goal then
            return Success                                         ▷ found path from $n_0$ to $n_*$
        end if
        for each $(u, v, w) \in E$ in EXPAND($u$) do              ▷ expand $u$
            ADD($\mathcal{L}, v, g_u + w$)                         ▷ add successors of $u$ to $\mathcal{L}$
        end for
    end while
    return Failure                                                 ▷ no path from $n_0$ to $n_*$ was found
end function
```

Proof of Lemma 1 by contradiction. Assume v does not lie on a shortest path from u to goal. Then $\exists\,(u, v', w') \in E$ with $v \neq v'$ and v' lies on a shortest path from u to goal. By optimality of h^*, it holds $h^*(u) = h^*(v') + w'$. Because v does not lie on a shortest path from u to goal, it holds $h^*(u) < h^*(v) + w$. Hence, $h^*(v') + w' < h^*(v) + w$. ↯ □

By Lemma 1, if h is optimal, it is sufficient for a search algorithm to pursue only a single edge minimizing $h(v) + w$. This means, with an optimal heuristic the branching factor b becomes 1 and our time complexity bound collapses to $\mathcal{O}(1^{|R|-1}) = \mathcal{O}(1)$. However, our bound does not account for the evaluation of h.

Proof of Theorem 1 by contradiction. With $d = |R| - 1$ and $b \leq |J|$, h is evaluated at most $(|R| - 1) \cdot |J|$ times. This term is polynomial in the size of the query graph $G_Q = (R, J)$. Assume for the sake of contradiction that we have an optimal heuristic $h \in$ PTIME. We would then be able to compute a shortest path in SP$_{\text{JOOP}}$ in PTIME. Since a shortest path in SP$_{\text{JOOP}}$ is an optimal plan for JOOP, we would be able to solve JOOP in PTIME. ↯ □

We now know that an optimal heuristic cannot be computed efficiently, i.e. in PTIME. However, if we were able to approximate h^* by some heuristic $h \in$ PTIME and h were consistent, then we could compute an optimal solution optimally efficiently with A^*. Interestingly, for practical purposes, the heuristic need not be consistent to achieve an effective branching factor close to 1.

2.3.3 Searching an Exponentially Large Space

In Section 2.2.4 we learned that the number of vertices $|V|$ and the number of edges $|E|$ of SP_{JOOP} are exponential in the size of the query graph G_Q. Hence, finding a shortest path in SP_{JOOP} requires time exponential *in the size of* G_Q. Constructing the *entire* search space a priori to the actual search would render our approach inefficient if not infeasible. Therefore, in our algorithm we will explore the search space on demand only.

We describe conceptually how the search space is computed successively and on demand by BFS and provide pseudo-code in Figure 1. We assume that BFS uses a container of vertices, usually called *open list*; some BFS algorithms implement this open list as a priority queue (e.g. A^*), some implement it as a single vertex (e.g. hill climbing). In each iteration of BFS, the *best* vertex is extracted from the open list to be *expanded* next (line 4). The definition of best depends on the search algorithm and is usually based on some combination of the weight g of the path by which the vertex was reached from the start n_0 and the heuristic value h of the vertex. For example, A^* defines best as the vertex minimizing $g + h$. When a vertex is expanded, some or all successors (but at least one) of this vertex are computed and added to the open list (line 9). BFS proceeds until one of two termination criteria is met: (1) When a goal is extracted from the open list (line 5), BFS has found a path from start to this goal and the search terminates successfully. (2) When the open list runs empty, no path from start to a goal was found and BFS terminates unsuccessfully (line 12).

Depending on the BFS algorithm, we may in the worst case explore the entire search space before reaching a goal. For A^*, this may happen when all joins have nearly the same cost, making edge weights nearly uniform and degrading A^* to breadth-first search. We believe, that situations in which (almost) the entire search space is explored are highly unlikely in the real world. We provide Table 2.1 in Section 2.3.5, which supports our claim with empirical data.

Figure 2.3 shows an example run of A^* on the search space in Figure 2.2. The algorithm starts in vertex $n_0 = ①$. We annotate each vertex with the weight g of the path from start ①. Additionally, we annotate vertices with their heuristic value h. For example, start ① has $g = 0, h = 50$. Vertex ① represents the four subproblems $\{A\}, \{B\}, \{C\}$, and $\{D\}$. We expand ① by applying any one of the four possible joins in J, thereby generating four successors. For each such successor we compute the weight g of the path from the start to that successor as well as the heuristic h. We label each edge with its weight, as resulting from expansion. For the expansion of ①, edges are labeled 20, 70, 120, and 10. The vertex of minimal $g + h$ that was not yet expanded is expanded next. Here, ② with $g + h = 10 + 40 = 50$ is expanded. ② represents the three subproblems $\{A\}, \{B\}$, and $\{C, D\}$. From ②, we can either join $\{A\}$ with $\{B\}$ or $\{B\}$ with $\{C, D\}$. Hence, the two successors ③ & ④ are generated by expansion of ②. We again compute g

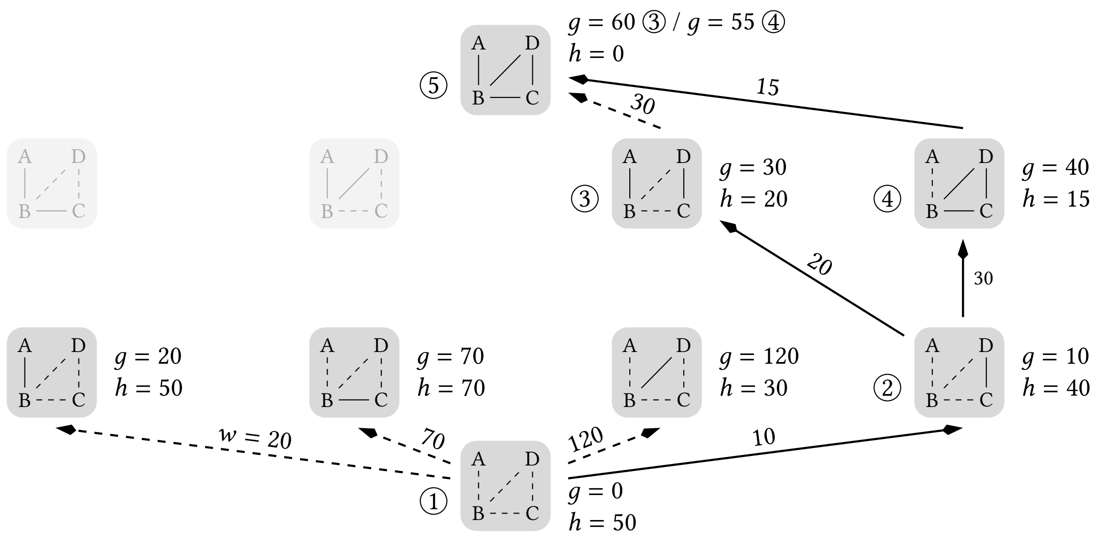


Figure 2.3: Search tree for bottom-up A^*. Vertices are labeled ① with their order of expansion. Dashed edges ⇢ mark vertices generated by expansion and are labeled with the weight of the edge. Solid edges → additionally were pursued by the search. Two vertices are never generated and the goal is generated twice, first with $g = 60$ by ③ and then with $g = 55$ by ④. The final plan is $A \bowtie (B \bowtie (C \bowtie D))$.

and h of these successors. Next, ③ with $g + h = 50$ is expanded into goal ⑤ with $g = 60, h = 0$. Notice, that at this point, the algorithm does *not* terminate yet, as we only *added* a goal ⑤ (line 9 in Figure 1). The next vertex to expand is ④ and expansion yields ⑤, again. However, this time ⑤ is reached by a path with weight $g = 55$. The next vertex to expand is ⑤ with $g + h = 55 + 0$. Since ⑤ is a goal, the search terminates successfully with a shortest path of weight 55.

2.3.4 Completeness, Soundness, Optimality

In this section, we show that the successive computation of the search space during BFS does not harm completeness, soundness, and optimality. We therefore sketch the proofs for these properties, assuming a BFS algorithm that is complete, sound, and optimal. However, before we do so, let us highlight three properties of the search space, that will help us with the proofs. (1) The search space is *acyclic*: there is no non-empty path that starts and ends in the same vertex. (2) The search space has *no dead-ends*: every vertex except the goal has at least one successor. (3) The *goal* n_* *has* depth $|R| - 1$: every path from n_0 to n_* has *exactly* length $|R| - 1$. These properties follow directly from the construction of the search space in Section 2.2.2.

Completeness. As the search space has no dead-ends, EXPAND yields for each non-goal vertex at least one successor. Because the search space is acyclic, each EXPAND reduces the distance (in edges) to n_* by one. Therefore and because the goal has finite depth, BFS will find a path from n_0 to n_*, if one exists.

Soundness. Soundness of our heuristic search follows from soundness of the reduction in Section 2.2.2, soundness of the BFS algorithm, and soundness of EXPAND. The latter is sound if it yields for a given vertex u only successors v w.r.t. E, i.e. $\exists\,(u, v, w) \in E$.

Optimality. In Section 2.2.2 we have shown that an optimal solution of the problem reduced to shortest path corresponds to an optimal solution of JOOP. Assuming optimality of BFS, what remains to show is that EXPAND preserves optimality. This is the case if for every vertex u, EXPAND(u) yields at least one vertex v s.t. v lies on a shortest path from u to goal. If EXPAND yields all successors, this is trivially the case.

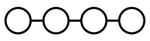
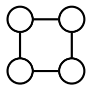
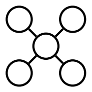
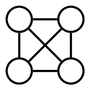
We now have a profound understanding of heuristic search and its applicability to our shortest path problem. Before we dive into the algorithmic challenges in Section 2.4, we present different performance criteria for evaluating heuristic search.

2.3.5 Performance Criteria

There are different measures to assess the performance of heuristic search. Of course, we look at running times in our evaluation in Section 2.7. If we allow for suboptimal solutions, an additional measure is how far off a computed solution is from an optimal solution (i.e. a shortest path). Another measure, that we already learned about in Section 2.3.2, is the effective branching factor b^*, which allows us to evaluate how informative a heuristic is to the search. However, b^* cannot be measured directly but is derived from the depth of the goal (which is $|R| - 1$) and another important measure for heuristic search: the number of generated vertices [RN20]. Because the depth of the goal is fixed, the only remaining variable for computing b^* is the number of generated vertices. Therefore, this number alone already allows us to compare different heuristics by how informative they are to the search. It also allows us to compare different search algorithms by how goal-oriented they explore the search space. We can even compare our approach to classical dynamic programming (DP): the number of vertices generated corresponds to the number of ccps joined and can directly be compared to the number of ccps enumerated by DP.

With this knowledge, we conduct our first experiment. We compare classical DP implemented by DP_{CCP} [MN06] to blind and heuristic search. As blind search we perform Dijkstra's

Table 2.1: Comparison of DP_{CCP} to search with Dijkstra and $A^*_{\downarrow} + h_{sum}$ by the number of CCPs that are enumerated to compute an optimal plan. We consider four different graph topologies and vary the number of relations. Less is better and the best per column is underlined.

	chain			**cycle**			**star**			**clique**		
												
	5	10	15	5	10	15	5	10	15	5	10	15
DP_{CCP}	20	165	560	36	321	1106	32	2304	114 688	90	28 501	7 141 686
Dijkstra↑	$\underline{10}$	438	2702	$\underline{13}$	130	2875	18	$\underline{218}$	$\underline{404}$	$\underline{22}$	$\underline{2884}$	$\underline{26\,992}$
Dijkstra↓	20	202	2629	44	650	6288	24	1026	61 739	91	191 313	>30 000 000
$A^*_{\downarrow} + h_{sum}$	$\underline{10}$	$\underline{53}$	$\underline{522}$	16	$\underline{118}$	$\underline{557}$	$\underline{10}$	265	12 568	29	20 969	7 050 206

algorithm in both directions: bottom-up, labeled Dijkstra↑, and top-down, labeled Dijkstra↓. As heuristic search we perform $A^*_{\downarrow}$ (top-down) with admissible heuristic h_{sum} (cf. Section 2.5). We compare the three algorithms by the number of ccps joined in Table 2.1. Because heuristic h_{sum} is admissible, $A^*_{\downarrow}$ computes an optimal plan, like DP_{CCP} and Dijkstra's algorithm. The best result in each column is underlined. Before we draw conclusions from our experiment, we want to emphasize that DP_{CCP} enumerates *all* ccps *exactly once* without duplicates [MN06]. The proportion of unique ccps to the number of relations $|R|$ is polynomial for chain and cycle topologies and exponential for star and clique topologies [OL90; MN06]. From the results in Table 2.1, we can make two key observations: (1) On star and clique topologies, Dijkstra↑ enumerates significantly less ccps than DP_{CCP}. On chain and cycle topologies, $A^*_{\downarrow} + h_{sum}$ enumerates significantly less ccps than DP_{CCP}. We conclude that heuristic search is able to find an optimal plan without enumerating *all* ccps; sometimes only a fraction of all ccps is required. (2) When we compare our two top-down searches, Dijkstra↓ and $A^*_{\downarrow} + h_{sum}$, we observe how much of an impact the heuristic has on the search: the heuristic sometimes reduces the number of ccps enumerated by more than one order of magnitude.

According to Observation 1, search and particularly heuristic search finds a provably optimal plan without the need to enumerate *all* ccps. In contrast, traditional algorithms for computing an optimal solution enumerate all ccps (with or without duplicates) [Sel+79; VM96; Van98; MN06; FM11a; FM11b]. Whether it is possible to compute an optimal plan via search in less time depends on whether search can be implemented efficiently. We describe the algorithmic challenges we face in the following Section 2.4. Observation 2 exemplifies the impact the heuristic has on the search's performance. Therefore, in Section 2.5, we explore and evaluate different heuristics.

2.4 Algorithmic Challenges

To be able to efficiently search for a shortest path from n_0 to n_*, we must be able to efficiently explore the search space. In Section 2.3.3, we argue that we must not compute the entire search space a priori to the search but instead compute the explored regions successively. Exploring the search space is done by successively expanding vertices to their successors, as exemplified in Figure 2.3.

In Section 2.4.1, we present a vertex representation that enables efficient generation of successors via EXPAND and efficient evaluation of a heuristic function h. Consequently, in Section 2.4.2 we show how to efficiently compute successor vertices for this representation in bottom-up and top-down search. As the search's performance also heavily depends on the implementation of the open list, we present in Section 2.4.3 an implementation that supports fast insertion of generated successors, fast extraction of the next best vertex, and efficient handling of duplicates. As we shall see in Section 2.4.4, some duplicates are actually desired while others are undesired. We develop an algorithm to suppress the generation of undesired duplicates already when expanding a vertex, thereby preventing attempts to insert undesired duplicates into the open list.

2.4.1 Vertex Representation

The vertices of our search space are sets of subproblems, i.e. sets of sets of relations. We can incrementally assign to each relation in the query graph a unique index, starting at 1. For the query graph in Figure 2.1, we could assign indices $A \mapsto 1$, $B \mapsto 2$, $C \mapsto 3$, $D \mapsto 4$. Each subproblem can then be represented as a bit vector $b_1 \dots b_{|R|}$ with bit b_i set if relation with index i is within the subproblem. For example, the subproblem $\{C, D\}$ is represented by the bit vector `0011`. A vertex is then represented as a sequence of bit vectors $\mathcal{V}$, with one bit vector per subproblem. Additionally, the bit vectors are kept sorted lexicographically to allow for hashing and efficient equality testing. For example, the vertex with label ② in Figure 2.3, $\big\{\{A\},\{B\},\{C,D\}\big\}$, is represented by $\mathcal{V} = [$`1000, 0100, 0011`$]$. To efficiently store and operate on bit sets, we employ the same case distinction as Neumann and Radke [NR18], using a 64 bit integer for queries with up to 64 relations, a 128 bit integer for up to 128 relations, a dynamic array of 64 bit vectors for up to 1024 relations, and a dynamic array of sorted relations for more than 1024 relations.

Algorithm 2 Bottom-up vertex expansion.

```
function EXPANDBottomUp($\mathcal{V}_u$ : representation of vertex $u$)
    for $i = 1$ to $|\mathcal{V}_u| - 1$ do
        for $k = i + 1$ to $|\mathcal{V}_u|$ do
            $\overline{b_i} \leftarrow \mathcal{V}_u[i]$                                  ▹representation of subproblem $S_1$
            $\overline{b_k} \leftarrow \mathcal{V}_u[k]$                                  ▹representation of subproblem $S_2$
            for each $j \in J$ joining $\overline{b_i}$ and $\overline{b_k}$ do
                $l \leftarrow \mathcal{V}_u[1 : i-1] \circ \mathcal{V}_u[i+1 : k-1]$       ▹slice $\mathcal{V}_u$ to replace $\overline{b_i}$ …
                $r \leftarrow \mathcal{V}_u[k+1 : |\mathcal{V}_u|]$                     ▹… and $\overline{b_k}$ by $(\overline{b_i} \mid \overline{b_k})$ …
                $\mathcal{V}_v \leftarrow l \circ \left(\overline{b_i} \mid \overline{b_k}\right) \circ r$        ▹… and maintain lexicographical order
                yield $\mathcal{V}_v$ by $j(S_1 = \overline{b_i}, S_2 = \overline{b_k})$        ▹emit how to generate successor
            end for
        end for
    end for
end function
```

2.4.2 Vertex Expansion

Given a vertex u, EXPAND(u) must compute all outgoing edges of that vertex. The term "outgoing" is now relative to the direction of join ordering: outgoing edges in bottom-up join ordering are incoming edges in top-down join ordering and vice versa, cf. Figure 2.2. Given the representation $\mathcal{V}_u$ of a vertex u in the search space, the task is to compute all edges $(u, v, w) \in E$, i.e. the outgoing edges of u. We now make a case distinction about the search direction.

Bottom-Up Search

In bottom-up search, there exists an edge $(u, v, w) \in E$ if there is a join $j \in J$ s.t. $j^*(u) = v$. Expanding the definition of hoisted joins from Section 2.2.2, we get

$$j^*(u) = v \quad \Leftrightarrow \quad \exists\, S_1 \neq S_2 \in u.\ v = \big(u \setminus \{S_1, S_2\}\big) \cup \big\{j(S_1, S_2)\big\}$$

From this definition we can derive Figure 2 to compute all outgoing edges of u: we simply need to test for all pairs of subproblems (S_1, S_2) whether there exists a join $j \in J$ s.t. $j(S_1, S_2)$.

Time Complexity. Figure 2 iterates over all pairs of subproblems (skipping symmetric pairs) and over all joins. Note that there can be at most $|R|$ many subproblems in $\mathcal{V}_u$. In the innermost loop, $\mathcal{V}_u$ is sliced to construct $\mathcal{V}_v$. This can be done in a single pass over $\mathcal{V}_u$. We can therefore bound Figure 2's time by $O(|R|^3 \cdot |J|)$. Judging by the asymptotic runtime, our algorithm appears to be very inefficient. However, our experiments in Section 2.7 reveal that expansion makes up

Algorithm 3 Top-down vertex expansion.

```
function EXPANDTopDown($\mathcal{V}_u$ : representation of vertex $u$)
    for $i = 1$ to $|\mathcal{V}_u|$ do                                  ▹partition each subproblem of $\mathcal{V}_u$
        for each ccp $\left(\overline{b_1}, \overline{b_2}\right)$ in PARTITION_MinCutAGaT($\mathcal{V}_u[i]$) do    ▹[FM11b, Fig. 6]
            $\mathcal{V}_v \leftarrow \mathcal{V}_u[1 : i-1] \circ \mathcal{V}_u[i+1 : |\mathcal{V}_u|]$    ▹remove subproblem at index i
            InsertSortedLex($\mathcal{V}_v, \overline{b_1}$)                    ▹insert $\overline{b_1}$ lexicographically
            InsertSortedLex($\mathcal{V}_v, \overline{b_2}$)                    ▹insert $\overline{b_2}$ lexicographically
            yield $\mathcal{V}_v$ by $j(S_1 = \overline{b_1}, S_2 = \overline{b_2})$          ▹emit how to generate successor
        end for
    end for
end function
```

for only a small fraction of overall search time.

Top-Down Search

Analogously to bottom-up search, in top-down search, there exists an edge $(u, v, w) \in E$ if there is a join $j \in J$ s.t. $j^*(v) = u$. Again, by expanding the definition of hoisted joins from Section 2.2.2, we get

$$\begin{aligned} j^*(v) = u \quad &\Leftrightarrow \quad \exists\, S_1 \neq S_2 \in v.\ u = v \setminus \{S_1, S_2\} \cup \left\{j(S_1, S_2)\right\} \\ &\Leftrightarrow \quad \exists\, S_1 \neq S_2 \in v\ \exists\, S \in u.\ j(S_1, S_2) = S \end{aligned}$$

This means, there exists an edge $(u, v, w) \in E$ if there is a subproblem S in u that can be partitioned into subproblems S_1, S_2 in v such that there is a join $j \in J$ with $j(S_1, S_2) = S$. Enumerating these partitions is exactly the problem of top-down join ordering [VM96; DT07; FM11a; FM11b]. We select an existing partitioning algorithm to enumerate all ccps of a subproblem, here PARTITION$_{\text{MinCutAGaT}}$ [FM11b], to implement top-down vertex expansion in Figure 3.

Time Complexity. Fender and Moerkotte [FM11b] analyze the time complexity of MincutLazy and find that it is worst for clique queries with $O(|R|^2)$ time. MinCutAGaT, which is based on MincutLazy, exhibits the same asymptotic runtime behavior. Analyzing our Figure 3, we see that the outer-most loop performs no more than $|R|$ iterations. Further, slicing $\mathcal{V}_u$ to construct $\mathcal{V}_v$ and the two invocations of InsertSortedLex require at most $O(|R|)$ time. We conclude that Figure 3's time complexity is bounded by $O(|R|^4)$. Note that MinCutBranch [FM11a] exhibits better asymptotic runtime behavior than MinCutAGaT for cycle and clique queries but is significantly more complex. As our evaluation in Section 2.7 shows, vertex expansion takes only a small share of overall search time and we therefore opt for MinCutAGaT in this work.

2.4.3 Open List and Duplicate Detection

As shown in Figure 1, BFS extracts in each iteration of the outer loop the next best vertex from the open list with extract-best, expands it into its successors, and adds the successors to the open list with add. To efficiently implement extract-best and add, we require a data structure that efficiently supports (1) finding the next best vertex, (2) removing the next best vertex, and (3) adding newly generated successor vertices. There is one more operation that the data structure should support. To motivate this, let us look again at the example in Figure 2.3. The goal ⑤ is generated twice, first by ③ with $g = 60$ and afterwards by ④ with $g = 55$. When ⑤ is generated the second time, it is already present in the open list. One way to support this, is by allowing for duplicates in the open list. This is safe, since duplicates have the same heuristic value h and hence the duplicate with smaller g is extracted first from the open list. However, if duplicates occur frequently, they cause the open list to grow unnecessarily large, thereby degrading performance. A better way to cope with duplicates is to detect them while they are being added to the open list. We therefore devise a scheme for the *detection of duplicates* (DeDup): A *new* vertex is immediately added to the open list. When a *duplicate* vertex is being added to the open list, we compare the g values of this duplicate and the vertex already in the open list. A duplicate with equal or greater g is discarded, as it cannot lead to finding shorter paths. A duplicate with smaller g means that we have found a shorter path from n_0 to this vertex. Instead of adding the duplicate to the open list, we reduce g of the vertex *already within the open list* to g of the duplicate that is being added. The data structure should therefore also support an operation to (4) reduce g of an already incorporated vertex.

Data structures fit for this task are heaps. They provide exactly the aforementioned required operations (1) find-min, (2) delete-min, (3) insert, and (4) decrease-key. There are many different implementations of heaps, e.g. binary heap, binomial heap, Fibonacci heap, and pairing heap, to name a few [Cor+16, 6.1 Heaps on p. 151]. Some heaps do not support decrease-key. In that case, the operation can be emulated by first deleting the vertex of old g and then re-inserting the vertex with new g. However, we shall use `boost::heap::fibonacci_heap`, which efficiently supports the decrease-key operation.[3]

DeDup requires that we can search for a particular vertex in the heap – an operation that is usually not (efficiently) supported. We therefore naïvely use a hash table in addition to the heap. The hash table serves two purposes: (1) It stores for each seen vertex a handle. If the vertex is currently in the heap, the handle references the heap entry. Otherwise, the vertex has

[3]Note that Boost implements max-heaps.

Algorithm 4 Add vertex to open list with duplicate handling.

```
function ADD($\mathcal{L}$ : open list, $v$ : vertex to add, $g_v$ : cost of $v$)
    if $v$ in $\mathcal{L}$.table then                                              ▷ is $v$ a duplicate?
        if $g_v < \mathcal{L}$.table$[v].g$ then                                    ▷ reached $v$ on a cheaper path?
            if $\mathcal{L}$.table$[v]$.handle is None then                         ▷ not in open list?
                $\mathcal{L}$.table$[v]$.handle ← INSERT($\mathcal{L}$.heap, $v$, $g_v + h(v)$)
                $\mathcal{L}$.table$[v].g \leftarrow g_v$                            ▷ remember cost of $v$
            else
                DECREASE-KEY($\mathcal{L}$.heap, $\mathcal{L}$.table$[v]$.handle, $g_v + h(v)$)   ▷ update cost
                $\mathcal{L}$.table$[v].g \leftarrow g_v$                            ▷ remember updated cost
            end if
        end if
    else
        $\mathcal{L}$.table$[v]$.handle ← INSERT($\mathcal{L}$.heap, $v$, $g_v + h(v)$)    ▷ insert and save handle
        $\mathcal{L}$.table$[v].g \leftarrow g_v$                                    ▷ remember cost of $v$
    end if
end function
```

Algorithm 5 Extract best vertex from open list.

```
function EXTRACT-BEST($\mathcal{L}$ : open list)
    $v, g_v$ ← FIND-MIN($\mathcal{L}$.heap)                    ▷ get best vertex and its cost
    DELETE-MIN($\mathcal{L}$.heap)                            ▷ remove best vertex from heap
    $\mathcal{L}$.table$[v]$.handle ← None                    ▷ update handle
    return $v, g_v$
end function
```

been extracted from the heap and the handle is **None**. (2) The hash table stores for each seen vertex the weight g of the cheapest path by which the vertex was reached. With the handle we are able to perform a DECREASE-KEY operation when a vertex is reached on a cheaper path. It also enables us to identify whether a vertex is currently in the heap or has already been deleted. Storing g inside the hash table enables us to discard duplicates not reached on a cheaper path, even when the vertex has already been extracted from the open list. DeDup implicitly requires that the heap can provide handles to entries. This is usually the case when the heap provides *referential stability*[4] of its elements. We implement ADD and EXTRACT-BEST with DeDup in Figure 4 and Figure 5, respectively. Note, that in the pseudo-code the open list $\mathcal{L}$ contains both the heap and the hash table. Further, Figure 4 defines the *best* vertex as the one minimizing $g_v + h(v)$, as is required by A^*; any other definition of *best* is possible.

We demonstrate the necessity of DeDup with a small experiment. We compare two implementations of the open list: one implementation with DeDup and one implementation that

[4]Referential stability is also called *pointer stability*.

Table 2.2: The impact of DeDup. We run Dijkstra↑ on queries of 10 relations and count new and duplicate vertices.

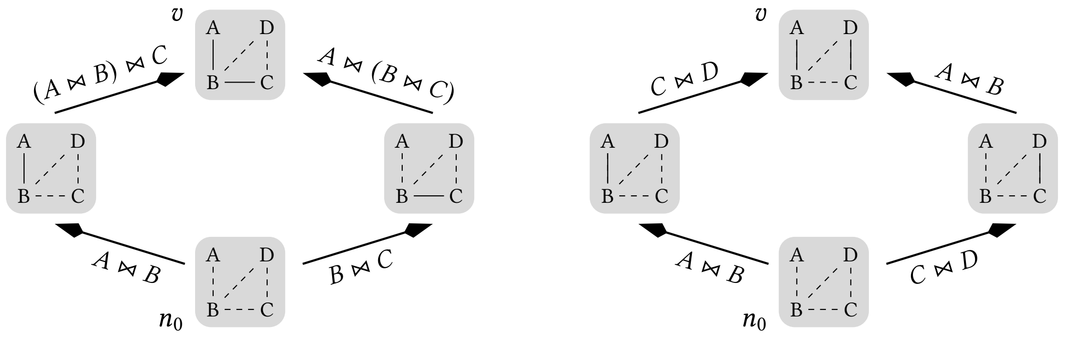
		chain	cycle	star	clique
without DeDup	#new	497	819	115	1497
	#duplicates	182 577	224 726	72	13 710
with DeDup	#new	497	819	115	1497
	#duplicates	<u>1140</u>	<u>1408</u>	<u>22</u>	<u>191</u>

simply allows for duplicates and performs no duplicate checking at all. We compute optimal plans for queries of 10 relations using bottom-up search with Dijkstra's algorithm and we count the new and duplicate vertices generated by vertex expansion. We present our findings in Table 2.2. Note, that whether we allow for duplicates in the open list only affects how many duplicates are generated and how often the same vertex is expanded but it does not affect which *unique* vertices are expanded or generated. The explored region of the search space remains the same. Hence, and as we expect, the amount of newly generated, unique vertices is independent of whether we allow for duplicates. In stark contrast, the amount of duplicates generated when allowing for duplicate vertices in the open list is up to two orders of magnitude larger. To understand why allowing for duplicates in the open list has such a devastating effect on the amount of vertices generated, we have to understand that every single duplicate extracted from the open list is expanded and hence all its generated successors become duplicates in the open list, leading to an exponential blow-up of duplicates. In the following Section 2.4.4, we discuss how we further prevent some duplicates of ever being generated.

2.4.4 Duplicate Prevention

In previous Section 2.4.3, we proposed DeDup to efficiently eliminate duplicates in the open list. However, DeDup does not prevent the *generation* of duplicates during vertex expansion. In the following, we identify two classes of duplicates: desired and undesired duplicates. We then devise a scheme to *prevent the generation of undesired duplicates* during vertex expansion (PreDup). We consequently extend vertex representation from Section 2.4.1 and expansion from Section 2.4.2.

To introduce the notion of desired and undesired duplicates, let us consider the two examples in Figure 2.4. Both examples show a fraction of the search space of Figure 2.2. In Figure 2.4a, we see two paths leading from the start n_0 to a vertex v, where relations A, B, and C have been joined. These two paths, despite leading to the same vertex, correspond to *two different partial plans*: one plan joins A and B first, the other joins B and C first. In contrast, in Figure 2.4b, we



(a) Desired duplicate, with each of the two paths corresponding to a unique partial plan.

(b) Undesired duplicate, with both paths corresponding to the same partial plan.

Figure 2.4: Example of desired versus undesired duplicates.

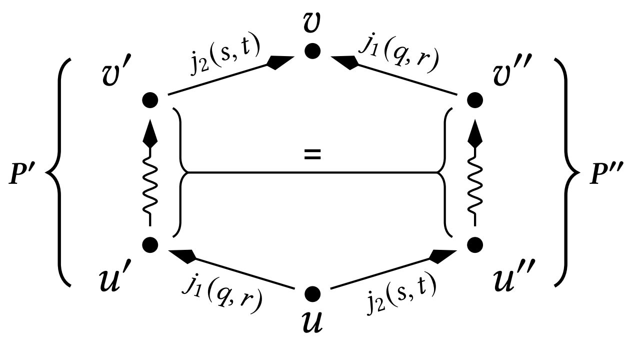


Figure 2.5: General pattern of undesired duplicates. Vertex u must contain four subproblems q, r, s, t, s.t. q can be joined with r and s can be joined with t. The order of the two joins can be permuted, resulting in two paths P' and P'' of exact same weight, formally $weight(P') = weight(P'')$. Hence, v is generated twice with the same cost.

see two distinct paths from n_0 to v that correspond to *the exact same partial plan*: although the two paths order the joins $A \bowtie B$ and $C \bowtie D$ differently, this ordering has no semantics in the corresponding partial plan. In the example of Figure 2.4a, we do want to generate the duplicate of v, as it may reveal a shorter path to v. If the duplicate does not reveal a shorter path, it will be discarded by DeDup (cf. Section 2.4.3). In the example of Figure 2.4b, however, we would be wise not to generate the duplicate of v. Its path corresponds to an already considered partial plan. Therefore, v has already been generated with the exact same cost g and hence the duplicate of v will definitely be discarded by DeDup.

We devise a scheme to prevent the generation of such undesired duplicates during vertex expansion, named PreDup. We say that a duplicate vertex is undesired, if the vertex was reached before on some path P' and is now reached on a different path P'' and it can be shown

Algorithm 6 Extension of Figure 2 to prevent redundant paths.

```
function EXPAND_BottomUp($\mathcal{V}_u$ : representation of vertex $u$)
    for $i = 1$ to $|\mathcal{V}_u| - 1$ do
        for $k = i + 1$ to $|\mathcal{V}_u|$ do
            $\overline{b_i} \leftarrow \mathcal{V}_u[i]$                          ▹representation of subproblem $S_1$
            $\overline{b_k} \leftarrow \mathcal{V}_u[k]$                          ▹representation of subproblem $S_2$
            if $\left(\overline{b_i} \mid \overline{b_k}\right) <_{\text{lex}} \nabla(\mathcal{V}_u)$ then      ▹undesired duplicate?
                continue                                       ▹skip
            end if
            for each $j \in J$ joining $\overline{b_i}$ and $\overline{b_k}$ do
                $l \leftarrow \mathcal{V}_u[1 : i-1] \circ \mathcal{V}_u[i+1 : k-1]$
                $r \leftarrow \mathcal{V}_u[k+1 : |\mathcal{V}_u|]$
                $\mathcal{V}_v = l \circ \left(\overline{b_i} \mid \overline{b_k}\right)^{\nabla} \circ r$            ▹maintain lexicographical order
                yield $\mathcal{V}_v$ by $j(S_1 = \overline{b_i}, S_2 = \overline{b_k})$         ▹emit how to generate successor
            end for
        end for
    end for
end function
```

that – independent of cardinalities – $weight(P') = weight(P'')$. In that case, we say path P'' is *redundant*. We provide a general pattern of undesired duplicates in Figure 2.5. There is a redundant path between vertices u and v, if u has four subproblems q, r, s, t such that q can be joined with r and s can be joined with t, i.e. $\exists\, j_1, j_2 \in J.\ j_1(q, r), j_2(s, t)$. In that case, any path from u to v that includes both these joins can be transformed into another, valid path from u to v by exchanging the order of the two joins. These two paths have the exact same weight, hence v is generated twice with exact same cost g. The idea of PreDup is to prevent the generation of undesired duplicates by preventing the search from pursuing redundant paths. More precisely, in Figure 2.5, the search may *either* perform $j_2(s, t)$ after $j_1(q, r)$ *or* $j_1(q, r)$ after $j_2(s, t)$ but not both. To implement this, we exploit the fact that our vertex representation in Section 2.4.1 keeps the sequence of bit vectors $\mathcal{V}$ sorted lexicographically. We have $j_1(q, r) = q \cup r$ and $j_2(s, t) = s \cup t$, with *either* $q \cup r <_{\text{lex}} s \cup t$ *or* $s \cup t <_{\text{lex}} q \cup r$. We store in $\mathcal{V}$ the subproblem that was the result of the most recent join, indicated with ∇. By storing the most recently joined subproblem, we can suppress the generation of undesired duplicates during EXPAND: we skip joins whose join result is lexicographically smaller than the stored subproblem of the expanded vertex. For example, let $q <_{\text{lex}} r <_{\text{lex}} s <_{\text{lex}} t$. When expanding u we get u' with $\nabla(\mathcal{V}_{u'}) = q \cup r$

and u'' with $\nabla(\mathcal{V}_{u''}) = s \cup t$ and

$$\begin{array}{llcll} \text{for } \mathcal{V}_{u'} : & j_2(s,t) = s \cup t & \not<_{\text{lex}} & q \cup r = \nabla(\mathcal{V}_{u'}) & \checkmark \\ \text{for } \mathcal{V}_{u''} : & j_1(q,r) = q \cup r & <_{\text{lex}} & s \cup t = \nabla(\mathcal{V}_{u''}) & \text{✗} \end{array}$$

Because we suppress join results that are lexicographically smaller than the stored subproblem, during vertex expansion the stored subproblem can only become larger w.r.t. the lexicographical ordering. Hence, no matter how v'' is reached from u'' in Figure 2.5, we know that $\nabla(\mathcal{V}_{v''}) \geq_{\text{lex}} \nabla(\mathcal{V}_{u''})$ and therefore $j_1(q,r)$ is suppressed when expanding v''. We extend our algorithm for bottom-up vertex expansion of Section 2.4.1 accordingly in Figure 6 and highlight the necessary modifications. In the initial vertex $n_0 = \binom{R}{1}$, the lexicographically smallest subproblem is marked. An extension of top-down expansion, as in Figure 3, would be analogous: the most recently partitioned subproblem is stored and subproblems lexicographically smaller than the most recently partitioned subproblem are not further partitioned. In the initial vertex of top-down search, i.e. $n_0 = \{R\}$, the single subproblem R is marked. We need to show that BFS with EXPAND as in Figure 6 is still complete, sound, and optimal.

Completeness. Looking at Figure 2.5, we see that our scheme only prevents expanding v'' to v, i.e. the edge $v'' \rightarrow v$, still leaving an alternative path from u to v. More generally, all vertices that were reachable from n_0 before our modification are still reachable from n_0. This particularly holds true for n_*.

Soundness. Our scheme neither introduces new edges into the search space nor does it alter the weights of existing edges. Therefore, any path found from n_0 to n_* corresponds to a feasible plan.

Optimality. In Figure 2.5, the paths P' and P'' have the exact same weight. If either of the paths, say P'', is eliminated by our scheme, then v is still reached by P' of exact same weight as P''. If P'' was an optimal path, so is P', and hence *an* optimal path to v is found.

To evaluate the gain of PreDup, we rerun the same experiment as in Table 2.2. This time, we use DeDup (cf. Section 2.4.3) and compare bottom-up search with and without PreDup. We present our findings in Table 2.3. Let us first look at star topology: We see that the number of generated new and duplicate vertices does not change. This is expected, as in star topology there are no bushy plans and therefore there are no redundant paths and no undesired duplicates. Next we look at chain, cycle, and clique topologies: We see that the number of generated duplicates is significantly reduced, as we expected. However, and maybe to your surprise, we

Table 2.3: The impact of PreDup on the experiment of Table 2.2. Both configurations include DeDup (cf. Section 2.4.3).

		chain	cycle	star	clique
DeDup only	#new	497	819	115	1497
	#duplicates	1140	1408	22	191
DeDup + PreDup	#new	$\underline{377}$	$\underline{626}$	115	$\underline{891}$
	#duplicates	$\underline{299}$	$\underline{419}$	22	$\underline{109}$

can also see that the number of generated new vertices shrunk. To understand why that is the case, let us reconsider Figure 2.3. With our scheme for preventing undesired duplicates, we have $\nabla(②) = \{C, D\}$. When ② is expanded, our scheme prevents the generation of ③ as successor, since $\{A, B\} <_{\text{lex}} \{C, D\}$. However, successor ④ is still generated and consequently a shortest path is found, with one unique vertex less generated.

2.5 Heuristic Functions for JOOP

So far, we presented how to solve join order optimization by heuristic search and the algorithmic challenges that arose. In this section, we present three heuristics for the search problem and their respective heuristic properties. However, we must first understand how to derive edge weights from a DBMS' cost model and how heuristics depend on the cost model.

2.5.1 From DBMS Cost Model to Edge Weights

In Section 2.2.2 we introduced a cost model C and used it to define the weights of the edges of the search space. The weight of a path is defined as the sum of the weights of its edges. A heuristic estimates the weight of a shortest path to the nearest goal. Therefore, heuristics depend on the cost model. It is hard, if not infeasible, to define an informative heuristic independent of the cost model. In this work, we focus on the well-known and frequently used cost model C_{out}, that assesses a plan by the sum of the cardinalities of all intermediate results [CM95; Neu09; NR18; FM11a; FM11b]. It is recursively defined as

$$C_{out}(T) := \begin{cases} 0 & \text{if } T \in R \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases} \tag{2.2}$$

Table 2.4: The impact of the definition of edge weights on the number of vertices generated. We run Dijkstra↑ on queries of 10 relations. Both approaches compute a solution that is optimal w.r.t. Definition 2.2.

		chain	cycle	star	clique
#vertices generated	with Definition 2.3	1014	2484	344	1864
	with Definition 2.4	<u>676</u>	<u>1045</u>	<u>137</u>	<u>1000</u>

In order to weigh the edges of the search space according to C_{out}, we must recursively decompose Definition 2.2 to match our definition of cost model C from Section 2.2.2:

$$C(S_1, S_2, j) \coloneqq |S_1 \bowtie S_2| \tag{2.3}$$

While this definition of C is coherent with C_{out}, it has one major pitfall that significantly hurts bottom-up heuristic search: According to Definition 2.2, the cardinality of the result set of the query is always included in the total cost. When comparing entire plans by their cost, the cardinality of the result set always cancels out. This is, however, not the case when comparing two entries of the open list in bottom-up heuristic search: Every entry for the goal already includes in its g the cardinality of the result set while entries for non-goal vertices do not include this cardinality in their g yet, despite the fact that this cardinality occurs as edge weight on *any* path to goal. When the heuristic frequently underestimates, this leads to goals added to the open list being artificially pushed towards the end, delaying their expansion and ultimately delaying finding a plan. We therefore devise a variant of C that simply excludes the cardinality of the result set:

$$C(S_1, S_2, j) \coloneqq \begin{cases} 0 & \text{if } j(S_1, S_2) = R \\ |S_1 \bowtie S_2| & \text{otherwise} \end{cases} \tag{2.4}$$

Plans optimal w.r.t. Definition 2.4 are also optimal w.r.t. C_{out} of Definition 2.2. We evaluate the impact of the two definitions of C on bottom-up heuristic search by comparing the number of vertices generated in Table 2.4. Our experiment demonstrates that with Definition 2.4 bottom-up heuristic search converges faster towards a goal. Note that this pitfall does not exist in top-down heuristic search, as the cardinality of the result set is immediately incorporated in *all* vertices when the initial vertex is expanded.

2.5.2 Four Simple Heuristics

The following heuristics are designed particularly for C_{out} of Definition 2.2, with edge weights computed according to Definition 2.4. In addition to the cost model, heuristics depend on the direction of the search, i.e. bottom-up vs. top-down, because heuristics estimate the distance to goal and the goal depends on the search's direction.

The *zero* heuristic. The simplest heuristic is the one assigning the same constant value to any vertex. This heuristic provides no additional information to the search. We define the particular constant heuristic with constant zero $h_{zero}(v) = 0$. Observe, that h_{zero} is goal-aware, consistent, and admissible. When used as heuristic for A^*, the search degrades into Dijkstra's algorithm. Naturally, h_{zero} can be used for both bottom-up and top-down search.

The *sum* heuristic. This heuristic provides a lower bound for the remaining cost to reach the goal in top-down search. All subproblems that are not base relations are yet to be partitioned. Looking at Definition 2.4, each subproblem that is not a base relation or R will add its cardinality to the overall cost C_{out}. We can therefore calculate a lower bound for the remaining cost by summing up the cardinalities of all subproblems that are neither base relations nor R:

$$h_{sum}(v) := \begin{cases} 0 & \text{if } v = \{R\} \\ \sum\limits_{S \in v \setminus \binom{R}{1}} |S| & otherwise \end{cases}$$

Since h_{sum} is a lower bound of the remaining cost, it never overestimates and therefore it is admissible and can be used to compute an optimal plan. Note that h_{sum} only accounts for the *current* subproblems, i.e. the subproblems in v. It does not consider any subproblems formed by partitioning subproblems in v. Therefore, h_{sum} often underestimates the remaining cost dramatically and the error grows with the distance (in #edges) of v to the goal.

The *GOO* heuristic. We devise a heuristic $h_{GOO\uparrow}$ for bottom-up search by greedily computing a *reasonable* path from the current vertex to goal using *greedy operator ordering* (GOO) [Feg98]. GOO iteratively selects and joins two subproblems until only a single subproblem remains. The two subproblems to join S_1, S_2 are chosen to minimize $|S_1 \bowtie S_2|$. As $h_{GOO\uparrow}$ estimates the remaining distance to goal by computing an actual path, the heuristic never underestimates. However, because the subproblems to join S_1, S_2 are chosen greedily, the heuristic often overestimates. Hence, BFS with $h_{GOO\uparrow}$ does not guarantee finding an optimal plan. We also devise a variant of this heuristic for top-down search, named $h_{GOO\downarrow}$. This variant partitions each subproblem $S = S_1 \bowtie S_2$ s.t. $|S_1| + |S_2|$ is minimized.

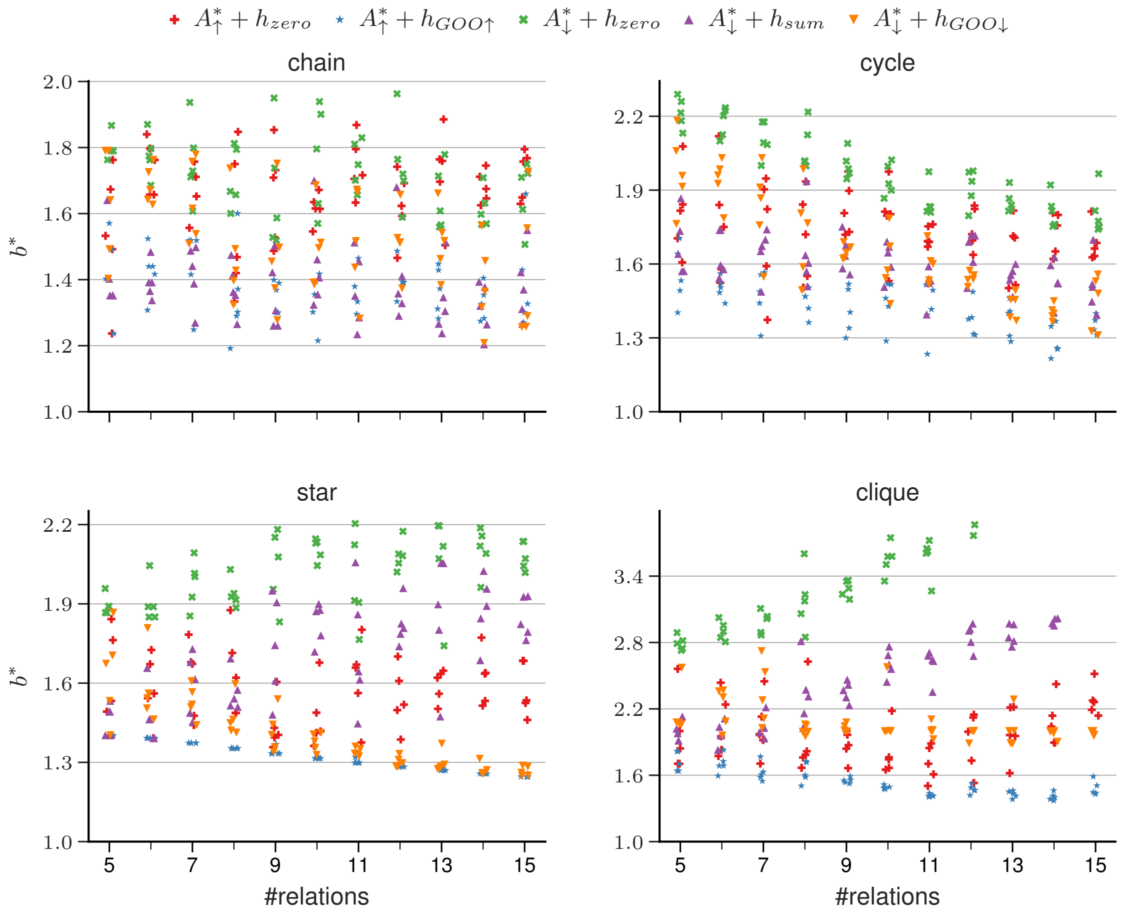


Figure 2.6: Information value of different heuristics.

2.5.3 Informative Value of Heuristic Functions

We compare the four heuristics h_{zero}, h_{sum}, $h_{GOO\downarrow}$, and $h_{GOO\uparrow}$ by how informative they are to heuristic search. We assess the heuristics by their effective branching factor b^* (cf. Section 2.3.2). Note, that A^* with h_{zero} is exactly Dijkstra's algorithm. We calculate b^* – the *average* branching factor *per* vertex expansion – from the depth d of the goal and the number N of generated vertices by solving the following equation [RN20].

$$N = b^* + (b^*)^2 + \cdots + (b^*)^d \tag{2.5}$$

We experimentally determine b^* for the four heuristics on the four topologies chain, cycle, star, and clique. We consider both bottom-up and top-down search with A^*. We vary the number of relations from 5 to 15, count the generated vertices, and from that derive the actual branching factor b^* according to Definition 2.5. We repeat each experiment five times with different seed (for details see Section 2.7). We present our results in Figure 2.6. We can generally observe that different heuristics provide different information value to the search and that their information value varies between the query topologies. In particular, we make five important observations: (1) $A^*_\uparrow + h_{zero}$ is generally more informative than $A^*_\downarrow + h_{zero}$. This is due to Definition 2.4, where in bottom-up search the cost g of a vertex already includes the cardinality of the current subproblems, which is not the case in top-down search. (2) h_{sum} corrects the aforementioned deficiency of h_{zero} and significantly reduces b^*. (3) For chain and cycle topology, $A^*_\downarrow + h_{sum}$ results in smaller b^* than $A^*_\uparrow + h_{zero}$, while for star and clique topology, exactly the opposite is the case. This observation suggests that different types of queries are better solved by bottom-up or top-down search. (4) Both $A^*_\uparrow + h_{GOO\uparrow}$ and $A^*_\downarrow + h_{GOO\downarrow}$ achieve least b^* for cycle, star, and clique topologies. This inadmissible heuristic causes the search to quickly converge towards a goal, but the solution can be suboptimal. (5) We can observe for each heuristic how b^* evolves with growing number of relations. When b^* grows with increasing number of relations, then the information value of the heuristic shrinks. Contrary, if b^* shrinks with increasing number of relations, the information value of the heuristic grows. For example, $h_{GOO\downarrow}$'s information value for star topology grows the more relations the query involves. The information value of h_{sum} for clique topology shrinks with increasing number of relations.

2.6 Related Work

2.6.1 Classical Join Ordering

Ibaraki and Kameda [IK84] prove that the problem of join order optimization is generally NP hard, even when allowing for only a single join method (i.e. nested-loops join). The authors provide a polynomial-time greedy algorithm, that computes an optimal plan if the query graph is a tree, e.g. a star query. The algorithm requires that the cost function, under which optimization is performed, satisfies the *adjacent sequence interchange* (ASI) property. The ASI property requires that a cost-benefit ratio, named *rank*, can be computed for each join. The work was further extended by Krishnamurthy, Boral, and Zaniolo [KBZ86] and the algorithm is sometimes referred to as IK/KBZ. Cluet and Moerkotte [CM95] show that summarizing the cardinalities of intermediate results serves as a good cost model, named C_{out}, that also satisfies the ASI property. We do not require ASI for heuristic search.

Selinger et al. [Sel+79] were the first to use DP to compute an optimal join order. Their algorithm, frequently referred to as $\mathrm{DP}_{\mathrm{size}}$, enumerates all (partial) plans in increasing number of relations, until a final, optimal plan is found. Cartesian products are performed as late as possible, i.e. never when the query graph is connected. Ono and Lohman [OL90] derive analytically for different topologies the number of distinct plans, excluding Cartesian products. For both star and clique topology, the number of plans is exponential in the number of relations. Vance and Maier [VM96] and Vance [Van98] improve upon $\mathrm{DP}_{\mathrm{size}}$ by devising a more efficient enumeration scheme following Gray code order [Gra53]. This algorithm is frequently referred to as $\mathrm{DP}_{\mathrm{sub}}$. Moerkotte and Neumann [MN06] further improve plan enumeration via DP. Their algorithm $\mathrm{DP}_{\mathrm{CCP}}$ enumerates all connected pairs of connected subgraphs without duplicates by traversing the query graph in a particular order. Chaudhuri et al. [Cha+95] invent top-down plan enumeration by decomposing a set of relations into two smaller sets and recursively computing optimal plans for these sets. In their work, the authors only consider linear plans. DeHaan and Tompa [DT07] generalize top-down plan enumeration to bushy plans and exclude Cartesian products. Their algorithm builds upon efficiently finding minimal graph cuts by computing the biconnected components of the query graph, that are organized in the *biconnection tree*. The authors show that top-down planning integrates well with cost-based branch-and-bound pruning, however the benefit is limited when Cartesian products are excluded. Fender and Moerkotte [FM11a; FM11b] further improve top-down plan enumeration with two algorithms: (1) $\mathrm{TD}_{\mathrm{MinCutAGaT}}$ extends the work of DeHaan and Tompa [DT07] by replacing the biconnection tree with an *advanced generate and test* routine. (2) $\mathrm{TD}_{\mathrm{MinCutBranch}}$ avoids connectedness checks by ensuring that only ccps are generated, thereby improving the complexity of finding a cut.

While the aforementioned algorithms enumerate *all* ccps, heuristic search is often able to find a *provably optimal* plan without enumerating all ccps. On the contrary, when the heuristic is uninformative, duplicates occur frequently. Branch-and-bound pruning is implicitly performed by the open list when ordering by g or $g + h$ and h is goal-aware. In contrast to prior work, heuristic search pursues those joins *first* that it deems to lead to cheaper plans.

2.6.2 Greedy Join Ordering

Fegaras [Feg98] presents *greedy operator ordering* (GOO), a greedy algorithm that repeatedly joins in each iteration the two subproblems leading to the smallest result size, until all relations are joined. GOO is a BFS with the heuristic defined as the result size of the most recent join and greedy BFS as search.

Neumann [Neu09] proposes query simplification to reduce the complexity of plan enumeration until it becomes tractable with DP. Simplification introduces ordering constraints, reducing the considered plans but sacrificing optimality. Neumann and Radke [NR18] propose *linearization* of the query graph. Their algorithm, named LinearizedDP, precedes $\mathrm{DP}_{\mathrm{CCP}}$ with a linearization phase based on IK/KBZ. Linearization, similarly to query simplification, greedily reduces the amount of plans considered by DP, potentially pruning the optimal plan and hence rendering DP suboptimal.

2.6.3 Heuristic Search

To the best of our knowledge, Sellis [Sel88] work on MQO is the first to apply heuristic search in the context of query optimization. In this particular work, multiple plans are generated for each query and a heuristic search algorithm then selects for each query exactly one plan, considering common intermediate results and minimizing the overall cost of executing all queries. Note, that this is a different optimization problem than join order optimization, where a single plan for a single query is computed.

Marcus et al. [Mar+19] train an ML model to predict the cost of the best plan constructible from a given partial plan. They use this model as heuristic for BFS. An argument is missing as to why the problem can be solved by search and whether the learned model has good heuristic properties. A general analysis of the search problem is lacking. Since the learned model may overestimate plan costs, the heuristic is inadmissible and hence search is suboptimal.

2.7 Evaluation

2.7.1 Setup

System. We implement our heuristic search and related state-of-the-art join ordering algorithms in mu*t*able [Haf+23], a main-memory database system currently developed at our group. Queries are provided to mu*t*able as SQL statements, for which mu*t*able computes a query plan with one of the join ordering algorithms. We use cost model C_{out} in all experiments. mu*t*able provides an interface to read cardinalities from a file. We use this feature to provide exact cardinalities to the process of join order optimization. We further exploit this feature to simulate queries with varying selection and join selectivities without the need to generate actual data.

Data. We evaluate all algorithms on the four query topologies chain, cycle, star, and clique, as they are a de-facto standard for evaluating join order optimization [DT07; FM11b; MN06; Neu09; Fen14; SMK97]. In addition, with this work we introduce a new benchmark which includes the former four topologies as special cases (Section 2.7.3). We vary number of relations and to simulate varying selection and join selectivities, we randomly generate 10 cardinality files per query. A file assigns to each subproblem a cardinality, where the cardinality assigned to a base relation represents the cardinality after selection (i.e. including selection selectivities) and

Algorithm 7 Generation of cardinalities.

```
function CARDINALITY-GEN($G_Q$ : query graph of query $Q$, $c_{min}$ : minimum cardinality, $c_{max}$ : maximum cardinality)
    $C \leftarrow$ new HashMap()                                   ▷ cardinalities of subproblems
    for each $r$ in $G_Q.R$ do                                   ▷ initialize cardinalities of base relations
        $C[r] \leftarrow$ RAND($c_{min}, c_{max}$)                 ▷ random cardinality in range $c_{min}$ to $c_{max}$
    end for

    $C' \leftarrow$ new HashMap()                                  ▷ maximum possible cardinality per subproblem
    for each csg $(S_1, S_2)$ of $G_Q$ do                          ▷ enumerated in $DP_{CCP}$ [MN06] order
        $c_1 \leftarrow$ CARDINALITY($S_1, C, C', c_{min}, c_{max}$)     ▷ get cardinality of $S_1$
        $c_2 \leftarrow$ CARDINALITY($S_2, C, C', c_{min}, c_{max}$)     ▷ get cardinality of $S_2$
        if $S_1 \cup S_2$ not in $C'$ then
            $C'[S_1 \cup S_2] \leftarrow c_1 \cdot c_2$              ▷ set max. cardinality of $S_1 \cup S_2$
        else
            $C'[S_1 \cup S_2] \leftarrow$ MIN($C'[S_1 \cup S_2], c_1 \cdot c_2$)    ▷ update max. cardinality
        end if
    end for

    $C[G_Q.R] \leftarrow$ CARDINALITY($G_Q.R$)                     ▷ cardinality of result
    return $C$
end function

function CARDINALITY($S, C, C', c_{min}, c_{max}$)
    if $S$ not in $C$ then                                       ▷ no fixed cardinality for $S$ yet?
        $c' \leftarrow$ MIN($C'[S], {c_{max}}^2$)                  ▷ max. cardinality of $S$, bounded by ${c_{max}}^2$
        $C[S] \leftarrow c_{min} + (c' - c_{min}) \cdot$ RAND(0, 1)   ▷ random cardinality of $S$
    end if
    return $C[S]$
end function
```

the cardinality assigned to a subproblem of multiple relations represents the cardinality after selection *and* join (i.e. including selection and join selectivities). We randomly generate these cardinalities with our algorithm CARDINALITY-GEN, given in Figure 7. Note that our algorithm produces *correlated* selectivities, i.e. it does not hold in general that $sel(A \bowtie B \bowtie C) = sel(A \bowtie B) \cdot sel(B \bowtie C)$.

Hardware. We run our experiments on a desktop computer with an AMD Ryzen Threadripper 1900X CPU at 3.8 GHz and 32 GiB DDR4 main memory. We disable the CPU's dynamic frequency scaling to reduce noise in our measurements.

Visualization. In line charts, the lines connect the arithmetic means and are highlighted by their 95% confidence interval. In box plots, the boxes show the interquartile range (25% - 75%) with a horizontal bar at the median (50%) and whiskers range from min to max – hence there

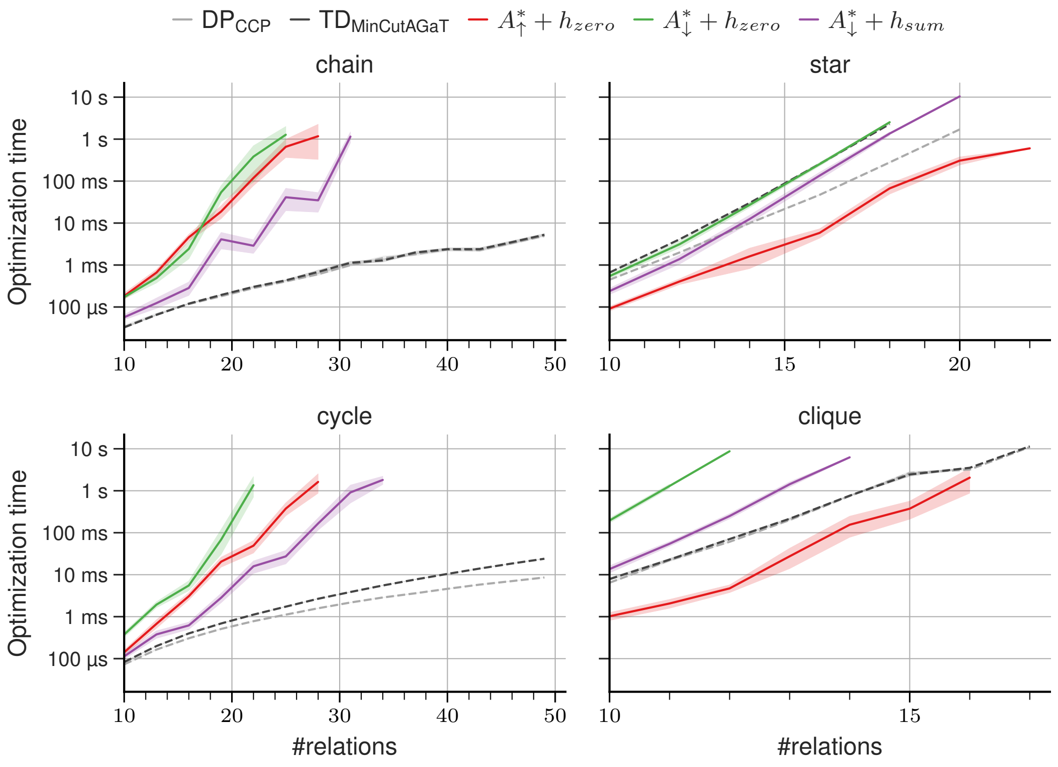


Figure 2.7: Comparison of our heuristic search to baselines DP_{CCP} and $\text{TD}_{\text{MinCutAGaT}}$ w.r.t. optimization time required to compute an *optimal* plan (less is better).

are no outliers.

2.7.2 Comparison to State of the Art

We compare our join order optimization via heuristic search to state-of-the-art algorithms. We distinguish between optimal and potentially suboptimal algorithms. We first compare by optimization time and then, for the potentially suboptimal algorithms, we compare the computed plans by their normalized cost.

We compare the optimization times of optimal join ordering algorithms in Figure 2.7. We make 4 key observations: (1) For chain and cycle, the fastest heuristic search is $A^*_\downarrow + h_{sum}$. (2) For star and clique, the fastest heuristic search is $A^*_\uparrow + h_{zero}$. (3) $A^*_\downarrow + h_{sum}$ always outperforms $A^*_\downarrow + h_{zero}$, emphasizing the importance of an informative heuristic. (4) Both DP_{CCP} and $\text{TD}_{\text{MinCutAGaT}}$ are unmatched by our heuristic search on the chain and cycle topologies. On the star and clique topologies, however, our $A^*_\uparrow + h_{zero}$ performs best, at roughly 10x faster than DP_{CCP} or $\text{TD}_{\text{MinCutAGaT}}$.

We compare the optimization times of suboptimal algorithms in Figure 2.8. For all four

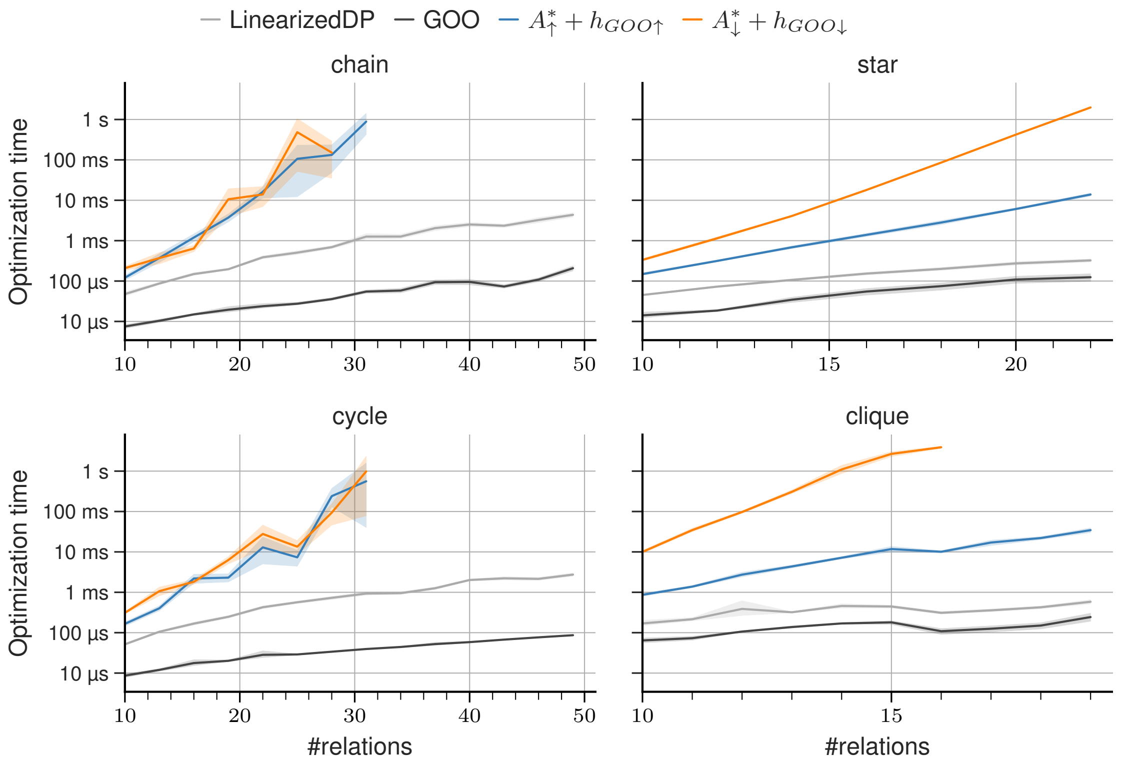


Figure 2.8: Comparison of our heuristic search to baselines LinearizedDP and GOO w.r.t. optimization time required to compute a *potentially suboptimal* plan (less is better).

topologies, we make the same observation: GOO is fastest, heuristic search is slowest, and LinearizedDP lies in between. On chain and cycle, both heuristic searches are equally fast, whereas on star and clique, $A^*_\uparrow + h_{GOO\uparrow}$ is significantly faster than $A^*_\downarrow + h_{GOO\downarrow}$.

In addition to the optimization times, we evaluate in Figure 2.9 the cost of the plans computed by suboptimal algorithms, normalized to the cost of an optimal plan. We can see that $A^*_\uparrow + h_{GOO\uparrow}$ generally produces the best plans. In particular, $A^*_\uparrow + h_{GOO\uparrow}$ improves over GOO in almost all cases. Surprisingly, $A^*_\downarrow + h_{GOO\downarrow}$ produces significantly worse plans than $A^*_\uparrow + h_{GOO\uparrow}$ on star and clique topologies. LinearizedDP produces exceptionally costly plans on the star and clique topologies. This is due to LinearizedDP's greedy linearization step, that is based on IK/KBZ, a greedy algorithm to compute *optimal linear* plans [IK84; KBZ86; NR18]. The problem with IK/KBZ is that it assumes uncorrelated join selectivities, an artificial constraint that is not provided by our data generation (cf. Figure 7). Therefore, IK/KBZ computes a suboptimal linearization, which rules out many good plans for LinearizedDP.

A general observation that we can make for heuristic search is that both optimization time and plan cost are correlated to the effective branching factor b^* of Figure 2.6: a smaller b^* leads

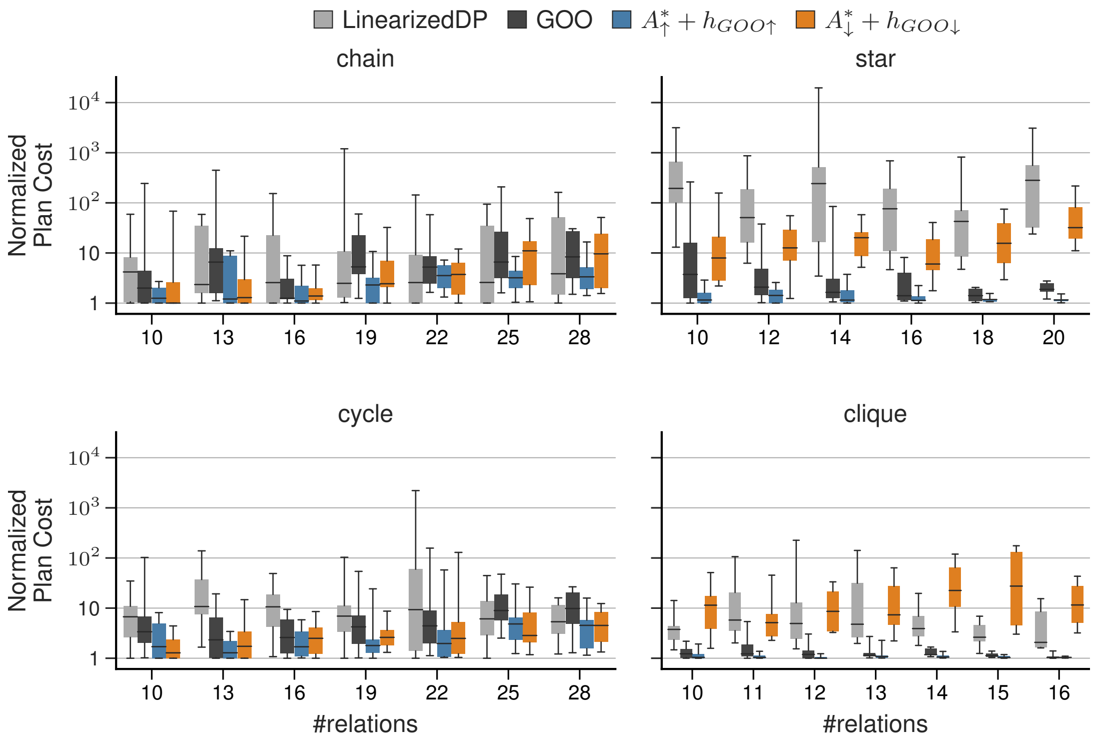


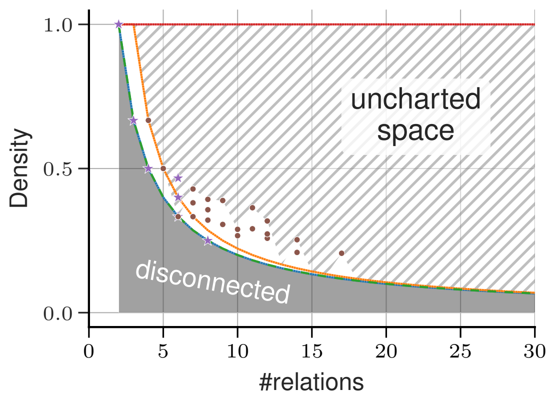
Figure 2.9: Comparison of our heuristic search to baselines LinearizedDP and GOO w.r.t. plan cost of the computed plan, normalized to the optimal plan as computed by any optimal algorithm (less is better, 1 is optimal).

to less optimization time and a better plan. This general rule does not apply, however, when comparing two searches of opposite direction, e.g. on star topology, $A^*_\downarrow + h_{GOO\downarrow}$ achieves smaller b^* than $A^*_\uparrow + h_{zero}$, but the latter is always faster. This supports our hypothesis from Section 2.5.3, that different types of queries are better solved by bottom-up or top-down search.

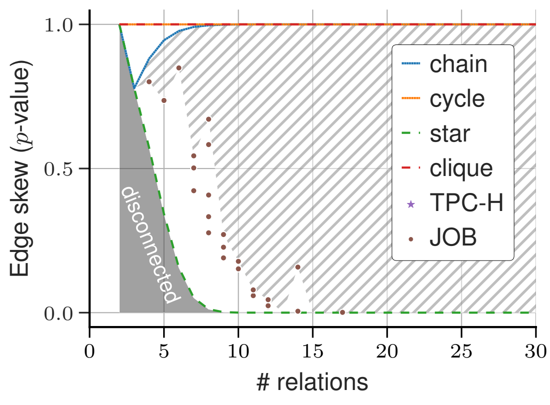
2.7.3 QGraEL: A New Benchmark for JOOP

Motivation

We analyze the four topologies studied in Section 2.7.2 together with the queries of the TPC-H and JOB benchmarks. For our analysis we introduce two measures on the query graph: *density* and *edge skew*. Density is simply defined as the graph density $D(G_Q) := \frac{2|J|}{|R|(|R|-1)}$ with $G_Q := (R, J)$, and it captures the ratio between actual edges and maximally possible edges in G_Q. We define edge skew as a measure for the distribution of degrees in G_Q, where the degree of a vertex is simply the number of edges at this vertex. We calculate edge skew as the p-value of the χ^2 test of the actual distribution of degrees in G_Q and expecting a uniform distribution



(a) By density.



(b) By edge skew (p-value).

Figure 2.10: Landscape of possible query graphs.

of degrees. For example, a cycle has a uniform distribution of degrees, i.e. every relation has a degree of 2. The χ^2 test will then compute $p = 1$ for *no edge skew*. For star, one relation has high degree while all other relations have degree 1 and p will be close to 0, signaling *high edge skew*.

With measures density and edge skew, we draw the entire landscape of queries in Figure 2.10. For density in Figure 2.10a, clique is at the upper limit with a density of 1, i.e. every possible join exists in G_Q, and chain and star are at the lower limit, with exactly $n - 1$ joins for n relations; graphs with fewer joins are disconnected. For edge skew in Figure 2.10b, clique and cycle have a uniform distribution of degrees and are at the upper limit of 1. With increasing number of relations, the edge skew of star increases and p converges towards 0.

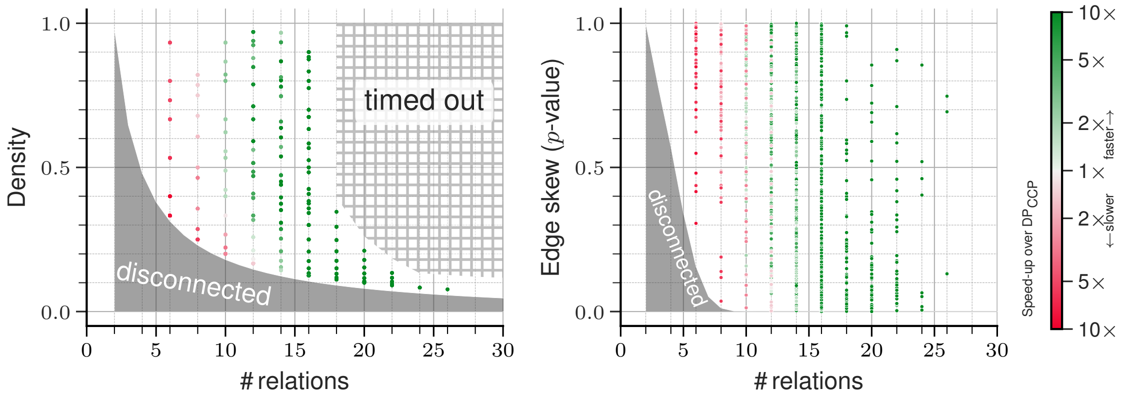
We additionally draw the queries of TPC-H and JOB into the landscape in Figure 2.10. We observe that all those queries have close to minimal density and high edge skew, leaving large uncharted spaces in both dimensions.

A New Benchmark

With this work we propose the new benchmark $\underline{\text{Q}}$uery $\underline{\text{Gra}}$ph $\underline{\text{E}}$xploration $\underline{\text{L}}$andscape (QGraEL). It systematically explores query graphs in three dimensions: number of relations, density, and edge skew. We evaluate every query in QGraEL with both DP$_{\text{CCP}}$ and $A^*_{\uparrow} + h_{zero}$ and compare their optimization times. This enables us to evaluate for which graph properties which algorithm performs better.

Results

Figure 2.11 shows our results, depicted along the three dimensions number of relations, density, and edge skew. We explore the landscape as much as possible, i.e. until either algorithm reaches



(a) By density. (b) By edge skew (p-value).

Figure 2.11: Speed-up of $A^*_\uparrow + h_{zero}$ over DP$_{\text{CCP}}$ in QGraEL. The color encodes the relative improvement of optimization time, and it is capped at 10 in both directions. Note, that in the worst case, $A^*_\uparrow + h_{zero}$ is less than 10x slower than DP$_{\text{CCP}}$ while in the best case we achieve speed-ups >1000x.

a fixed timeout. The color encodes the improvement or deterioration of heuristic search over DP$_{\text{CCP}}$. We observe that large spaces of the landscape that have been unexplored so far are clearly dominated by heuristic search. While the color coding in Figure 2.11 is clamped to 10x in both directions, Figure 2.12 visualizes the full range of relative improvement, with exceptional speed-ups of up to 1000x.

2.7.4 Detailed Evaluation of Heuristic Search

We perform an in-depth evaluation of heuristic search to understand how much the different operations contribute to the overall optimization time. We measure how much time is spent in each function via statistical profiling with the Linux PERF tool. We profile two runs: $A^*_\downarrow + h_{sum}$ on a chain query of 40 relations (`chain-40`) and $A^*_\uparrow + h_{zero}$ on a star query of 22 relations (`star-22`). We visualize the collected profiling data as a *flame graph*, that is a stacked horizontal bar chart, where one bar corresponds to one function and the width of the bar corresponds to the time spent within this function. When one function is called from another function, their bars are vertically stacked from bottom to top and in the order of the call stack. Figure 2.13 presents our findings. For search with $A^*_\uparrow + h_{zero}$, the heuristic is optimized out during compilation and hence does not appear in the flame graph. We can see that search spends a large share of its optimization time in EXPAND(), however only a fraction of time is spent inside the function itself. This observation supports our claim in Section 2.4.2, that vertex expansion makes up for only a small fraction of overall search time. On `chain-40`, most time is spent on evaluating

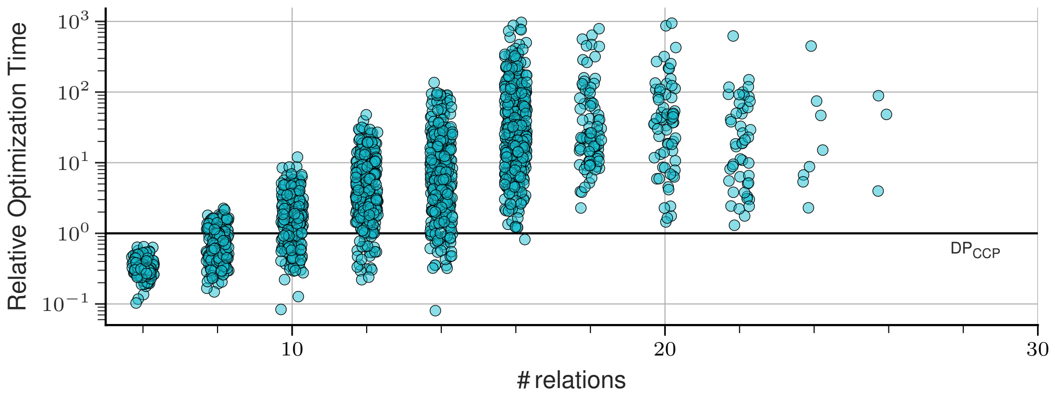


Figure 2.12: Relative improvement in optimization time of optimal heuristic search over DP_{CCP}. There are fewer data points towards higher number of relations as DP_{CCP} times out more frequently, making a relative comparison impossible.

the heuristic and on extracting the top element from the heap. On `star-22`, more than half of the time is spent on cardinality estimation. Note, that the optimization times are relative: on `star-22` search does not spend more time on cardinality estimation than on `chain-40` but instead search on `star-22` spends *less* time on managing the open list. This is due to search on `star-22` being more goal-oriented than search on `chain-40`, hence generating less duplicate vertices. The generation of duplicates on `chain-40` leads to frequent recalculation of heuristic values without progressing further towards the goal.

2.8 Conclusion

With this work, we provide a sound and generic framework for join order optimization via heuristic search. Our optimizations make heuristic search practical for application in a real DBMS, as our evaluation confirms. Figure 2.14 shows that we are able to extend the Pareto frontier of optimization time vs. plan cost. Our optimal solution $A^*_\uparrow + h_{zero}$ outperforms SOTA by up to 2 orders of magnitude on star and clique topologies. Our suboptimal solution $A^*_\uparrow + h_{GOO\uparrow}$ provides a middle ground between GOO, which is fast but shows high variance in plan quality, and our optimal solution. While this paper aims to be self-contained, there are many aspects or variations to our approach that did not fit into a single paper. We would like to give a glimpse of what future research may focus on: • designing more informative heuristics, potentially tuned for different query topologies, • applying different search strategies, e.g. *beam search*, *iterative deepening* A^*, *fringe search*, • *anytime search*, where search proceeds until a resource is exhausted or search is stopped and then retrieving the best plan found so far, or • *bidirectional heuristic search*, where search is simultaneously performed bottom-up and top-down and when

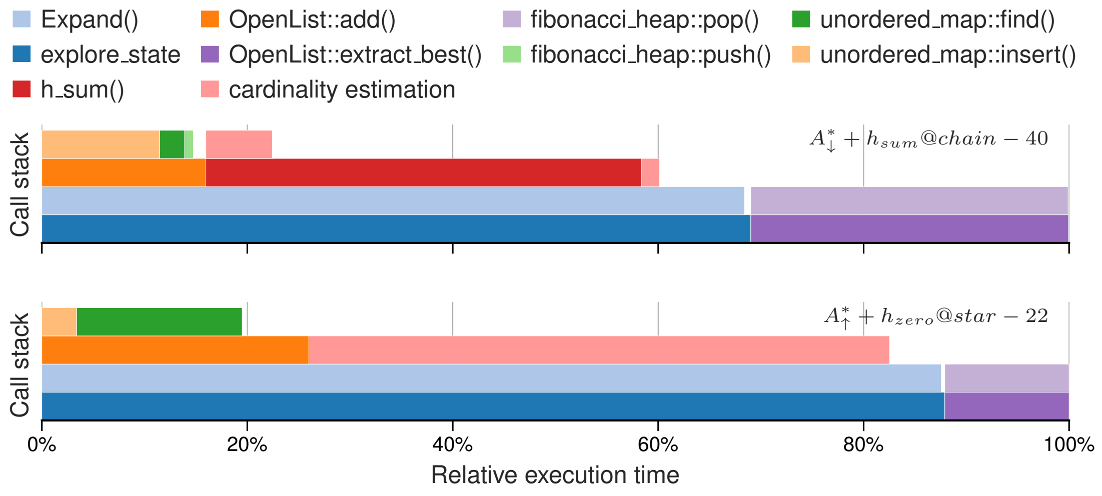


Figure 2.13: Detailed running time analysis of heuristic search.

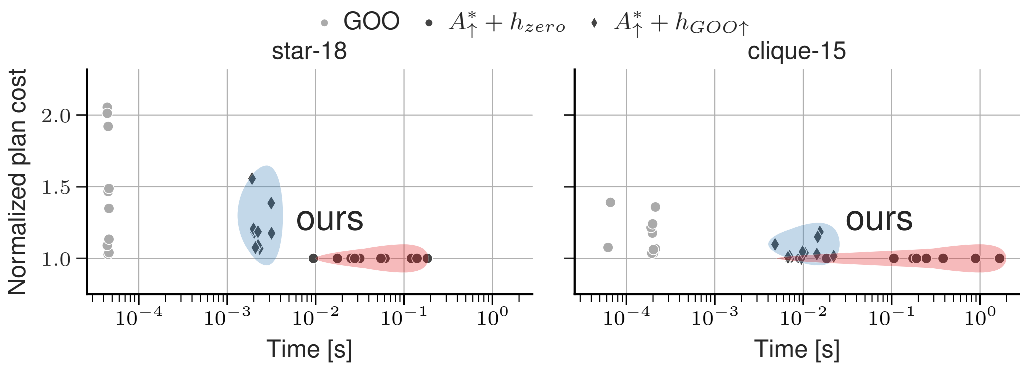


Figure 2.14: Pareto frontier of optimization time vs. plan cost.

both searches meet, a plan is found. We believe that our work serves as a foundation for and enables future research in the direction of computing join orders with heuristic search.

We would like to thank Karl Bringmann (MPII) for clarifying Section 2.2.4 & Section 2.3.2 and proofreading Theorem 1, Daniel Gnad (Linköping University) for supporting this work with his expertise in AI planning, and Felix Brinkmann for conducting an initial investigation on the applicability of AI planning to JOOP as part of his Bachelor's thesis.

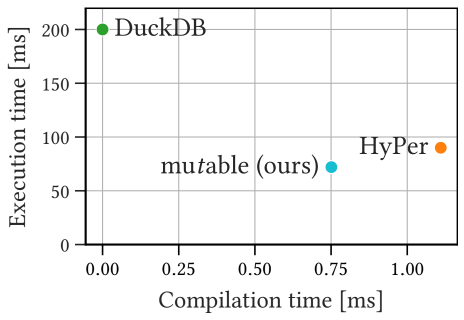


Figure 3.1: Design space of query execution engines, based on TPC-H Q1 benchmark results. The compilation time is the time to translate a QEP to machine code. The execution time is the time to execute the machine code and does not include the compilation time.

Chapter 3

Query Compilation

This chapter is based on my publication "A Simplified Architecture for Efficiently Compiling SQL Queries with WebAssembly" [HD23a]. This work was published in the research track of EDBT 2023. This work was co-authored by my Ph.D. advisor Prof. Dr. Jens Dittrich.

3.1 Introduction

To execute SQL queries, database systems must determine for each query a *query execution plan* (QEP) that defines how to execute the query. The QEP is then executed by either interpretation or compilation. Many early database systems used an interpreter for query execution, as it is easy to maintain and portable [Klo+14]. The Volcano model presented a generic and extensible design, adopted by many database systems that followed [Gra94]. The induced overhead of interpretation was dwarfed by the high costs for data accesses in disk-based systems [BZN05a; Neu11; Ker+18]. However, in modern main memory systems data accesses are significantly faster and the interpretation overhead suddenly takes a large share in query execution costs [Ail+99; Neu11]. Therefore, main memory systems must keep any overheads during query execution at a minimum to achieve peak performance. This development was the reason for an extensive body of work on query interpretation and compilation techniques and sparked a seemingly endless debate which of the two approaches to prefer [Rao+06; KVC10; Neu11; SZB11; Klo+14; Ker+18; KLN21]. Recently, Kohn, Leis, and Neumann proposed an adaptive approach to query execution, where the database system can seamlessly transition from interpreted to compiled query execution [KLN18]. This approach requires *both* a query interpreter *and* a query compiler that must be interoperable, which is achieved by a particular execution mode named *morsel-wise execution* [Lei+14]. Kersten, Leis, and Neumann followed up on this work and present adaptive execution by switching from non-optimized to optimized code during query processing [KLN21]. Despite the promising results of both works, we believe that implementing either approach requires expertise in interpreter and compiler design and poses an immense development effort, ultimately preventing wide-spread adoption.

In this work, we propose a new architecture for query execution engines of database systems. Rather than reengineering compiler technology, we suggest to employ a suitable and – most importantly – existing execution engine that takes care of *just-in-time* (JIT) compilation and adaptive execution. We dramatically reduce the complexity of the system by relying on existing infrastructure. By translating the QEP to an interchange format and delegating execution to an underlying engine, we are able to drastically reduce compilation times while maintaining competitive execution performance, as exemplified in Figure 3.1.

Contributions In this work, we present a new approach to JIT compilation of SQL queries to efficient machine code by building on existing JIT compiler infrastructure and the new low-level language WebAssembly.

1. We present a new, simplified, conceptual architecture of a query execution engine that allows us to delegate JIT compilation, optimization, and adaptive execution to an underlying engine. Like that, we avoid reengineering techniques researched and developed by the compiler community for decades. (Section 3.2)

2. We demonstrate how to implement this architecture in a real database system: mu*t*able. We use WebAssembly as intermediate representation and Google's V8 as backend. However, any other backend with similar properties as V8 conceptually works as well. mu*t*able supports the full pipeline of compiling SQL queries to executable code. (Section 3.3)

3. We discuss in detail the pros and cons over compiling with LLVM, adaptive compilation, and vectorized execution. (Section 3.4)

4. We discuss current limitations of our approach and how they will get resolved in the (near) future. (Section 3.5 & Section 3.6)

5. We provide an extensive experimental study, showcasing that even though we use an architecturally much simpler approach than state-of-the-art, we are able to match or even outperform state-of-the-art query compilers like HyPer. (Section 3.8)

We discuss related work in Section 3.7. We conclude our work in Section 3.9.

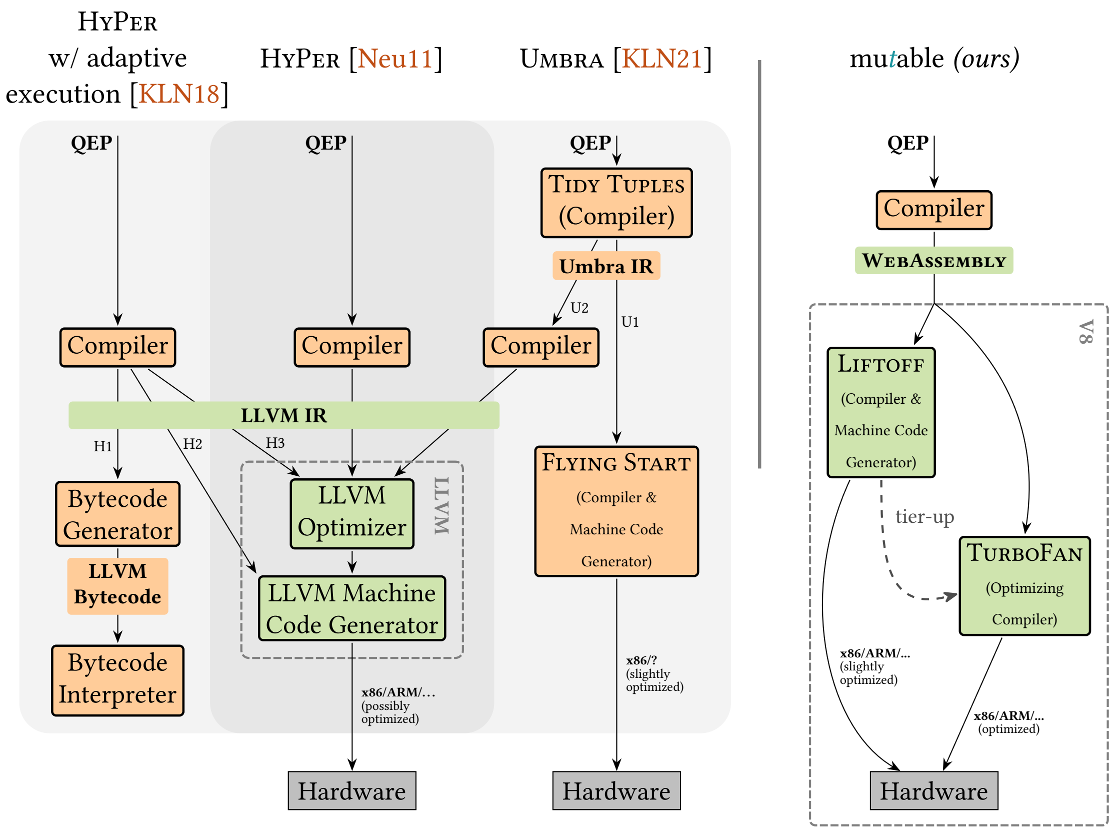
3.2 A New Architecture for Compiling Query Engines

We begin by motivating the need for an architectural simplification of query engines. We then propose our architecture and discuss pros and cons. In Figure 3.2a we present an overview of the architectures of prominent compiling query engines.

3.2.1 Other Architectures

HyPer. Although the very first relational database system, System R, already compiled queries to machine code [Cha+81], compiling queries only really became maintainable with the use of a compilation framework, such as LLVM used in HyPer [Neu11]. The original architecture of HyPer is shown in the second column of Figure 3.2a. HyPer translates the QEP with its own compiler to LLVM IR, the *intermediate representation* (IR) of LLVM . LLVM provides a large set of optimization passes that can be applied to the IR, potentially transforming the IR and increasing program efficiency. HyPer applies a fixed, handpicked subset of LLVM 's optimization passes [Neu11]. This subset was chosen such that optimization time is balanced with optimization gain. After applying the optimization passes to the IR, HyPer runs LLVM's machine code generation to obtain executable code. HyPer then runs this code on the hardware.

HyPer w/ adaptive execution. Potentially long-running compilation with LLVM delays query execution. Therefore, Kohn, Leis, and Neumann propose in a follow-up work to extend the compilation pipeline by interpretation and adaptively switching from interpreted to compiled execution as soon as compilation completes [KLN18]. This architecture is shown in the first column of Figure 3.2a and is an extension of the original architecture of HyPer. It uses the same compiler to translate QEPs to LLVM IR. At this point, there are three paths to proceed with. The first path H1 translates the LLVM IR with their bytecode generator to LLVM bytecode, an IR developed by the authors that is similar to LLVM IR yet optimized for interpretation. This LLVM bytecode is then interpreted by their bytecode interpreter. The second and third path, H2


HyPer w/ adaptive execution [KLN18]
HyPer [Neu11]
Umbra [KLN21]
mutable (ours)
QEP
QEP
QEP
QEP
Tidy Tuples (Compiler)
Umbra IR
U2
U1
Compiler
Compiler
Compiler
Compiler
WebAssembly
V8
LLVM IR
H1
H2
H3
Bytecode Generator
LLVM Bytecode
Bytecode Interpreter
LLVM Optimizer
LLVM Machine Code Generator
LLVM
Flying Start (Compiler & Machine Code Generator)
Liftoff (Compiler & Machine Code Generator)
tier-up
TurboFan (Optimizing Compiler)
x86/ARM/... (possibly optimized)
x86/? (slightly optimized)
x86/ARM/... (slightly optimized)
x86/ARM/... (optimized)
Hardware
Hardware
Hardware

(a) Architectural overview of compiling query engines.

	HyPer w/ adaptive execution [KLN18]	**HyPer [Neu11]**	**Umbra [KLN21]**	**mutable *(ours)***
Interpretation	🟧 ✓	✗	✗	🟩 ✓
Fast JIT Compilation	✗	✗	🟧 ✓	🟩 ✓
Optimizing Compilation	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✓
Adaptive Execution	🟧 ✓	🟥 ✗	🟧 ✓	🟩 ✓
Different HW (x86, ARM, etc.)	🟩 ✓	🟩 ✓	🟧 ✓(?)	🟩 ✓

(b) Feature matrix for the architectures in Figure 3.2a.

Figure 3.2: Orange 🟠 means (potentially re-) implemented by the system itself, green 🟢 means used off the shelf, and red 🔴 means desirable but lacking.

and H3, both rely on the original architecture of HyPer: H2 directly translates the LLVM IR to machine code, producing an *"O0"* executable. H3 incorporates LLVM optimizations, eventually producing an *"O2"* executable. While the query is being executed by interpretation of the LLVM bytecode, the LLVM IR is optimized and compiled to machine code in the background. Once this process completes (and the query has not yet terminated), the system switches from interpreted to compiled execution. Switching is enabled by morsel-wise execution [Lei+14].

Umbra. Although the architecture of HyPer with adaptive execution reduces query latency without sacrificing performance for long-running queries, initial interpretation is still slow and compilation with LLVM takes relatively long. This observation lead to another follow-up work by Kersten, Leis, and Neumann, in which the authors drop interpretation entirely in favor of fast JIT compilation [KLN21]. This architecture, shown in the third column of Figure 3.2a, is implemented in Umbra. In this architecture, the QEP is first translated by Tidy Tuples into their own Umbra IR. Umbra provides two compilation paths, U1 and U2, for this IR. U1 translates the IR directly to machine code with their JIT compiler Flying Start. This compiler performs only a fixed amount of passes over the IR, employs only a few fast optimizations, and generates slightly optimized *"O1"* machine code. Its purpose is to produce machine code fast while exploiting some potential for optimization. U2 translates the Umbra IR further to LLVM IR and follows the LLVM compilation pipeline as in HyPer, eventually producing a fully optimized *"O2"* executable. Similar to HyPer with adaptive execution, Umbra uses morsel-wise execution to switch from the code produced by Flying Start to the fully optimized code produced by LLVM .

Criticism

The original design of HyPer is clean and simple: use an existing compilation framework to compile QEPs to efficient machine code. However, the choice for LLVM introduces the deficiency of long compilation times delaying execution. Both HyPer with adaptive execution as well as Umbra work around this deficiency by introducing an alternative, much faster path to begin query execution and combine this with adaptively switching to optimized code when available. By inspecting our overview in Figure 3.2a, we can observe the sheer engineering effort that both systems undertake to enable this alternative path. We argue that neither approach will find wide-spread adoption as both require expert knowledge in interpreter and compiler design as well as immense development efforts. We therefore present a new architecture, that is as clean and simple as the original architecture of HyPer, yet brings the same benefits as Umbra.

3.2.2 Our Architecture

Requirements. To make justified decisions for our architecture, we first establish a common notion of our requirements: (1) We want to minimize the latency of query execution. (2) At the same time, we want to maximize the throughput of long-running queries. (3) Any kind of optimization should not add to the latency, meaning that optimization must be interweaved with execution. (4) Rather than solving (1)-(3) ourselves, we want to build on existing infrastructure.

Towards a solution. (1) To minimize latency of query execution, we can use interpretation or fast compilation of QEPs. (2) To increase throughput, we can apply crucial optimizations when compiling, e.g. register allocation. (3) To avoid optimization delaying query execution, optimization and query execution can happen in parallel. Query execution should switch to execution of the optimized code as soon as it becomes available. To increase adaptivity, optimizations should be applied on a fine granule: rather than waiting for the entire QEP to be optimized, we can compile and optimize individual pipelines and immediately make use of the optimized code. (4) Existing infrastructure providing the desired traits comes in the shape of JIT compilation frameworks or entire engines, controlling compilation, execution, and re-optimization.

Implementation. With a suitable JIT infrastructure at hand, the architecture of the query engine becomes surprisingly simple: translate the QEP to the interchange format and submit it to the infrastructure implementing (1)-(3) for execution. To our satisfaction, there is a plethora of projects implementing requirements (1)-(3) in an off-the-shelf engine. We give an overview of available projects and potential interchange formats in Section 3.7. Our choice for implementing this architecture is as follows: We translate QEPs to WebAssembly and delegate execution to the V8 engine. V8 is Google's JavaScript and WebAssembly engine and it fulfills all our aforementioned requirements. The fourth column of Figure 3.2a shows how V8 embeds into our proposed architecture. V8 provides two compilation tiers: fast compilation with Liftoff [Ham18] and optimizing compilation with TurboFan [V8 08]. Although initially WebAssembly is compiled with Liftoff to quickly start execution, V8 gradually replaces code during execution by optimized code produced by TurboFan as soon as it becomes available [Ham18; Nie+20]. V8 hence not only compiles WebAssembly but also takes care of adaptive execution. V8's Liftoff fulfills the same purpose as Umbra's Flying Start while V8's TurboFan can be seen as an optimizing compiler like LLVM with optimization passes, yet it is designed for a JIT environment and hence much faster. While Umbra has to implement and steer switching from non-optimized to optimized code, we can rely on V8 gradually optimizing the code during execution. Further, V8 provides fine-granular control over which optimizations to perform, whether to optimize adaptively during execution, and whether to enable the Liftoff compiler. One more benefit particular to V8 is that it compiles WebAssembly, which is an excellent interchange format between QEP and V8 as we elaborate in Section 3.3. We provide a summarized feature comparison in Figure 3.2b.

3.3 WebAssembly

Having a fast JIT compiler is inevitable to reducing latency in a compiling query engine, but it is certainly not enough. A (JIT) compiler takes as input the program, encoded in text or some kind of bytecode. We hence must translate the QEP to a suitable format accepted by the compiler. This step adds to the overall compilation time. It is therefore necessary to choose a fitting interchange format to enable fast translation of QEPs.

WebAssembly, or short *Wasm*, is "a low-level assembly-like language with a compact binary format that runs with near-native performance" [MDN]. Among the many high-level goals of WebAssembly, we see three key features that make it the instrument of choice for JIT compiling QEPs. The first key feature is that WebAssembly is size and load-time efficient, allowing for fast code generation, fast JIT compilation to machine code, and resource-friendly caching of already compiled code [Haa+17; Jan+19]. Second, WebAssembly is a *virtual instruction set architecture* (ISA) and therefore hardware independent and embeddable in many environments. Third, WebAssembly can be compiled to execute at near native speed [Jan+19] and make use of modern hardware capabilities, e.g. SIMD [V8 08]. Many WebAssembly engines offer debugging interfaces. The V8 engine provides an interface using the Chrome DevTools protocol over web socket. A developer can launch Google Chrome and connect to the V8 instance. The developer then has access to a wide range of debugging tools, including breakpoints, watchpoints, and memory inspection.

Although the name "WebAssembly" suggests that it was developed for the web, WebAssembly is primarily a virtual ISA that can be embedded in an execution environment. We highly recommend to the curious reader the work of Haas et al. [Haa+17], where the design of WebAssembly is elaborated in great detail and advantages over other low-level IRs are discussed.

3.3.1 Embedding WebAssembly

Despite its many benefits, WebAssembly comes with two significant limitations. The first limitation is that WebAssembly does not provide a standard library. Data structures like hash tables, algorithms like sorting, and even basic routines such as `memcpy` are not available out of

Listing 3.1 Example query to demonstrate the pipeline model.

```sql
SELECT R.x, MIN(S.x)
FROM R, S
WHERE R.x < 42 AND R.id = S.rid
GROUP BY R.x;
```

the box. The second limitation is that WebAssembly does not support generic programming. Hence, we cannot simply implement a library with generic algorithms and data structures ourselves. However, we shall work around these limitations by building on the ability to rapidly generate and compile WebAssembly. We solve the entire problem of not having a library by doing ad-hoc code generation: *Every algorithm and data structure required by a QEP is generated during compilation.* We do this in such a way, that we provide the concrete types of generic components, as required in the QEP, to the code generation process, which directly produces *monomorphic* code. Our approach allows us to rapidly generate code that is already fully inlined and specialized for the data types used in the QEP. We are able to achieve performance improvements that, in some cases, can have a tremendous impact. We elaborate our approach of ad-hoc library code generation in Section 3.5.

3.4 Compiling SQL to WebAssembly

In this section, we elaborate how to compile QEPs of SQL queries to WebAssembly. We dissect a QEP into pipelines, for which we generate code in topological order. We briefly revisit the pipeline model in Section 3.4.1. In Section 3.4.2 we sketch how we compile simple relational operators to WebAssembly. In Section 3.4.3 we explain how we compile complex operators without relying on an existing library by integrating ad-hoc generation of algorithms and data structures into the compilation process.

3.4.1 Pipeline Model

A QEP is – in its most essential form – a tree with tables or indexes at the leaves and relational operators at the inner nodes.[1] Figure 3.3 shows a QEP for the query in Listing 3.1. The edges between nodes of the tree point in the direction of data flow.

The tree structure of a QEP can be dissected into *pipelines* [BFV96]. A pipeline is a linear sequence of operators that does not require materialization of tuples. To identify the pipelines of a QEP, we hence must identify all operators that require materialization, named *pipeline*

[1]The authors are aware that a QEP need not strictly be a tree and in some situations a representation as directed acyclic graph is desirable [NK15].

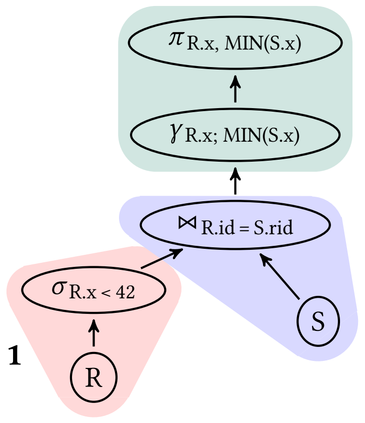


Figure 3.3: A QEP for the query in Listing 3.1, containing three pipelines enumerated in topological order.

breakers [Neu11]. The most common pipeline breakers are grouping, join, and sorting; table scan, index seek, selection, and projection are not pipeline breakers.

In Figure 3.3, we have colored and enumerated the three pipelines of the QEP. Pipeline 1 scans table `R`, selects all tuples where `R.x < 42`, and inserts all qualifying tuples into a hash table for the join. Pipeline 2 scans table `S` and probes all tuples against the hash table constructed by pipeline 1. Every pair of tuples from `R` and `S` that satisfies the condition `R.id = S.rid` is joined and inserted into another hash table where groups of `R.x` are formed. Pipeline 3 iterates over these groups and performs the final projection.

After dissecting the QEP into pipelines, each pipeline is compiled separately. However, we must order the pipelines such that all data dependencies of the QEP are satisfied. For example, pipeline 3 iterates over all groups produced by grouping. Hence, pipeline 2 that forms those groups must be executed *before* pipeline 3. By topologically sorting the pipelines we compute an order that satisfies all data dependencies in the QEP.

The pipeline model allows us to dissect a QEP into linear sequences of operators that process tuples without need for intermediate materialization. The pipeline model does *not* dictate whether to push or pull tuples, whether to process tuples one at a time or in bulk, or whether to execute the QEP by compilation or interpretation. In this work, we compile the pipelines of a QEP such that a single tuple is pushed at a time through the entire pipeline until it is materialized in memory.

3.4.2 Compiling Simple Operators

To compile simple operators to WebAssembly, we follow the approach of Neumann [Neu11], i.e. we generate data-centric code. We do not yet implement advanced code generation techniques, such as relaxed operator fusion [MMP17] or access-aware code generation [CGK20]. However, the approach of Neumann [Neu11] does not work for complex operators, as we will outline in Section 3.4.3. Generating WebAssembly code is very similar to generating LLVM code. In the following, we briefly sketch how we compile simple operators of a QEP.

Table scan, index seek, and pipeline breakers. The start of a pipeline – which is either a table scan, an index seek, or a pipeline breaker – is translated to a loop construct. For a table scan, we emit code to access all tuples of the respective table. For an index seek, we emit code to iterate over all qualifying entries in the respective index. For a pipeline breaker, e.g. grouping, we emit code to iterate over all materialized tuples, e.g. groups. The remainder of the pipeline is compiled into the loop's body.

Selection. A selection is compiled to a conditional branch. It is debatable whether to prefer short-circuit evaluation. For "simple" predicates, short-circuit evaluation is likely a bad choice: it introduces a conditional branch that unnecessarily stresses branch prediction [SZB11]. It may further lead to a conditional load from memory, which may negatively impact prefetching [Ker+18]. For "complex" predicates, short-circuit evaluation likely pays off: a conditional branch can bypass costly evaluation of the right hand side of a logical conjunction or disjunction [Ros02]. This transformation is a part of *if-conversion* [All+83]. We perform the aforementioned transformations during query optimization and *before* compilation. This optimization relies on domain-specific knowledge, e.g. predicate selectivities, that is inaccessible to the Wasm compiler. We do not implement predication in this work: every selection is compiled into one or more conditional branches.

Projection. The projection of an attribute or aggregate does not require an explicit operation. The code necessary to access the attribute's or aggregate's value has already been generated when compiling the beginning of the pipeline. To compile the projection of an expression, we compile the expression and assign the result to a fresh local variable. In contrast to interpretation, projecting attributes *away* is performed implicitly and requires no further code. Because the attribute that is projected away is not used further up in the QEP, no code using the attribute is generated. The register or local variable holding the attribute's value is automatically reclaimed during compilation to machine code [Boi+08].

3.4.3 Compiling Complex Operators

Compiling complex operators that need sophisticated algorithms and data structures is particularly difficult in our setting, as we cannot rely on an existing standard library providing generic implementations. We solve this deficiency by the ad-hoc generation of required algorithms and data structures during compilation of the QEP. We will explain this in detail in Section 3.5.

Hash-based Grouping & Aggregation. Hash-based grouping is a pipeline breaker: the incoming pipeline to the grouping operator assembles the groups in a hash table and updates the group's aggregates. The pipeline starting at the grouping operator iterates over all assembled groups as explained above.

An important distinction between our work and previous work is how inserts and updates to the hash table are performed. Previous work – including both interpretation- and compilation-based execution – relies on the existence of a pre-compiled library that provides a hash table implementation [Neu11; KLN18; KVC10]. There, operations on the hash table must use a type-agnostic interface, ruling out effective implementations of certain has table designs. The major issue is that the type of hash table entry is unknown at the time when the library is compiled. To look up a key, the key's hash is required. The hash can be computed *outside* the library and the computed hash value can be passed through the hash table's interface, as done by Neumann [Neu11]. However, hash collisions must be resolved and duplicates must be detected. Because of the type-agnostic interface, the hash table has no means to compare two keys. Hence, a callback function for pair-wise comparison is passed to the hash table's lookup function. Note, that looking up n keys requires *at least* n such callbacks! The situation gets worse if the hash table must be able to grow dynamically. To grow a hash table, all elements of the table must be rehashed. Again, because the hash table is type-agnostic, it has no means to compute the hashes. Hence, a callback function for hashing must be provided in addition to the comparison callback or the computed hash values must be stored within the hash table. Another downside of using a pre-compiled library is that calls to the library cannot be inlined at the call site: every access to the hash table requires a separate function call.

We resolve these issues by generating and JIT compiling the code for the hash table during compilation of the QEP. Although sounding expensive and prohibitive, we show in Section 3.8 that generating and compiling WebAssembly is affordable at running time. We explain the generation of library code in detail in Section 3.5.

Simple Hash Join. A simple hash join is a pipeline breaker for one of its inputs: the incoming pipeline, by convention the left subtree of the join, inserts tuples into a hash table. The pipeline of the join probes its tuples against that hash table to find all join partners. The same distinction between our work and previous work as for Hash-based Grouping & Aggregation applies here. To avoid artificial constraints on hash table design and to avoid issuing a function call per access to a hash table, we generate and JIT compile the required hash table code during compilation of the QEP. This approach is elaborated in Section 3.5.

Sorting. Sorting is a pipeline breaker and very similar to Grouping & Aggregation. Before the sorting operator can produce any results, all tuples of the incoming pipeline must be produced and materialized. After the incoming pipeline has been processed entirely, the sorting operator can output tuples in the specified order.

We implement the sorting operator by collecting all tuples from the incoming pipeline in an array and sorting the array with Quicksort. The way we integrate sorting into the compiled QEP is an important distinction between our work and previous work. In previous work that performs compilation, a sorting algorithm already exists as part of a pre-compiled library that is invoked to sort the array. The interface to this sorting algorithm is type-agnostic, i.e. the sorting algorithm does not know what it is sorting. In order to compare and move elements in the array, additional information must be provided when invoking the sorting algorithm. For comparison-based sorting, the size of an element in the array and a function that computes the order of two elements must be provided. This is very well exemplified by `qsort` from libc. This design leads to two severe performance issues. First, because the size of the elements to sort is not known when the library code is compiled, a generic routine such as `memcpy` must be used to move elements in the array. This may result in suboptimal code to move elements or even an additional function call per move. Additionally, values cannot be passed through registers and must always be read from and written to memory, obstructing optimization by the compiler. Second, to compute the order of two elements an external function must be invoked. This means, for every comparison of two elements the sorting algorithm must issue a separate function call. (To sort n elements, at least $\Theta\left(n \log n\right)$ such calls are necessary!)

When the QEP is being interpreted, e.g. in the vectorized execution model, similar problems emerge. Although tuples need not be moved if an additional array of indices is used, the sorting algorithm must delegate the comparison of two tuples to the interpreter, where the predicate to order by is dissected into atomic terms that are evaluated separately. This leads to significant interpretation overhead at the core of the sorting algorithm.

We resolve the aforementioned issues by generating and JIT compiling the library code during compilation of the QEP. Our generated sorting algorithm is precisely tuned to the elements to sort and the order to sort them by. In particular, the comparison of two elements is fully inlined into the sorting algorithm. We explain this approach in detail in Section 3.5.

Listing 3.2 Vectorized processing example. A selection vector is successively refined to compute `R.x < 42 AND R.y > 13`.

```
/* Create a fresh vector with indices
 * from 0 to VECTOR_SIZE - 1. */
sel0 = create_selection_vector(VECTOR_SIZE);
/* Evaluate LHS of conjunction. */
sel1 = cmp_lt_i32_imm(sel0, vec_R_x, 42);
/* Evaluate RHS of conjunction. */
sel2 = cmp_gt_i64_imm(sel1, vec_R_y, 13);
```

3.5 Ad-hoc Library Code Generation

In Section 3.3.1 and Section 3.4.3 we already motivated ad-hoc generation of specialized library code during compilation of a QEP. In this section, we elaborate our technique along the example of generating specialized Quicksort. While other building blocks of QEPs, e.g. hash join, would also suit as interesting example of our approach, we choose Quicksort for two reasons: (1) Quicksort is a recursive algorithm. We demonstrate that our ad-hoc generation is not limited to mere code fragments but can generate entire recursive functions ad-hoc. (2) At its core, Quicksort repeatedly performs pair-wise comparison of elements when partitioning the data set. This part of the algorithm benefits most from specialization to a particular element type and sort order, demonstrating the significant impact specialization can have on performance (cf. Section 3.8.2). We begin with partitioning and inlined comparison of elements before we explain how we generate Quicksort. Note that we need not generate code for an entire library (e.g. libc or STL) but we generate code for only those routines required by the QEP. Therefore, ad-hoc generation must be implemented only for those parts of libraries used by QEPs.

3.5.1 Conceptual Comparison

Before diving into the code generation example, let us reconsider our approach on a conceptual level and compare it with alternatives. A problem that is inherent in all query execution engines is that their supported operations must be polymorphic. Joins, grouping, sorting, etc. must be applicable to attributes of any type and size. We aim to provide this polymorphism at query compilation time by generating specialized library code. To understand how other systems solve this task, let us look at state-of-the-art solutions.

Vectorized Interpretation. In the vectorized processing model, operations are specialized for the different types of vectors. In Listing 3.2, we provide an example for the evaluation of a selection with a conjunctive predicate. The initial selection vector `sel0` is successively refined by calls to vectorized comparison functions `cmp_*` and eventually `sel2` contains all indices where the selection predicate is satisfied. A vectorized query interpreter executes a QEP by calling these vectorized functions and managing the data flow between function calls. To achieve short-circuit evaluation of the condition, the selection vector `sel1` is passed to the second comparison, such that the right-hand side of the conjunctive predicate is only evaluated for elements that also satisfy the left-hand side. In a compiling setting, short-circuit evaluation is usually implemented as a conditional branch. In the vectorized processing model, that control flow is converted to data flow. Conditional control flow can benefit from branch prediction, which works well in either case when the selectivity is very high or very low. However, when the control flow is converted to data flow, the benefit on low selectivities is lost [SZB11; Pir+16; Ker+18], as we exemplify in our example in Listing 3.2. Assume that the left-hand side of the condition is barely selective. Although the outcome of evaluating the left-hand side can be well predicted, evaluation of the right- hand side in line 7 can only start once the comparison in line 5 completes. Hence, this design completely eliminates the processors ability to predict the outcome of evaluating the left-hand side and executing the right-hand side unconditionally and out of order, as opposed to how it would be in a data-centric setting. A drawback of interpretation is that operations must be specialized and compiled ahead of time. It is infeasible to provide vectorized operations for arbitrary expressions, as there are infinitely many. Therefore, the interpreter dissects expressions into atomic terms for which a finite set of vectorized operations is pre-compiled. For our example in Listing 3.2, this means that the interpreter must *always* evaluate one side of the conjunction after the other and cannot evaluate both sides at once.

Linking with pre-compiled library. In a compilation-based processing model, e.g. HyPer, every operation in the QEP is compiled to a code fragment. The produced code is specific to the types of the operation's operands. Arbitrarily complex expressions are compiled directly rather than taking a detour through pre-compiled functions for expression evaluation, like in the interpretation model. Thereby, the compiler can choose to implement short-circuit evaluation by conditional control flow.

The biggest drawback of compiling QEPs is the time spent compiling. While direct compilation to machine code could be done rapidly, the produced code would certainly be of poor quality. Therefore, compilation-based systems employ compiler frameworks like LLVM to perform optimizations on the code. While these optimizations can greatly improve the performance of the code, they require costly analysis and transformation. Hence, compilation of queries can easily take more than a hundred milliseconds [KLN18].

To reduce the amount of code to compile, recurring routines like hash table lookups or sorting are pre-compiled and shipped in a library. During compilation of a QEP, when an operation can be delegated to a pre-compiled routine, the compiler simply produces a respective

Listing 3.3 Demonstration of a compilation-based processing model with calls to a pre-compiled library. Every insertion into the hash table requires a separate function call.

```
/* Initialize hash table. */
HT *ht = lib_HT_create();
/* Iterate over all rows of table R. */
for (auto row : tbl_R) {
    /* Evaluate selection predicate. */
    if (row.x < 42 and row.y > 13) {
        /* Compute hash of R.id. */
        auto hash = ... row.id ...;
        /* Insert into hash table. */
        char *ptr = lib_HT_insert(ht, hash, /* #bytes= */ 8);
        *(int*) ptr     = row.id; // key
        *(int*)(ptr+4) = row.x;  // value
    }
}
```

function call to the library. This is a trade-off between compilation time and running time and the biggest drawback of this approach. Function calls to a pre-compiled library prevent inlining and obstruct further optimization, thereby potentially leading to sub-optimal performance. We demonstrate this in Listing 3.3, where every insertion into a hash table requires a separate function call. The library code for probing the hash table can be compiled and optimized thoroughly ahead of time. Because the size of a hash table entry is unknown when the library is compiled, the size must be provided at running time when inserting an entry. In the example, the hash table must allocate 8 bytes per entry to store `R.id` and `R.x` and it is the task of the caller to assign those values to the entry.

Full compilation. In this approach, code for the entire QEP with all required algorithms and data structures is generated and compiled just in time. By generating the code just in time, it is possible to produce highly specialized code, target particular hardware features, and enable holistic optimization. One example for full compilation is template expansion, as done in the HIQUE system [KVC10]. HIQUE provides a set of generic algorithms and data structures that are instantiated and compiled to implement the QEP. Another example is code generation via *staging*, as done in LEGOBASE. Here, metaprogramming is used to write a query engine in Scala LMS, that when partially evaluated on an input QEP outputs specialized C code that implements the query [Klo+14]. While full compilation can achieve the highest possible throughput, both HIQUE and LEGOBASE take considerable compilation time with hundreds of milliseconds for single TPC-H queries.

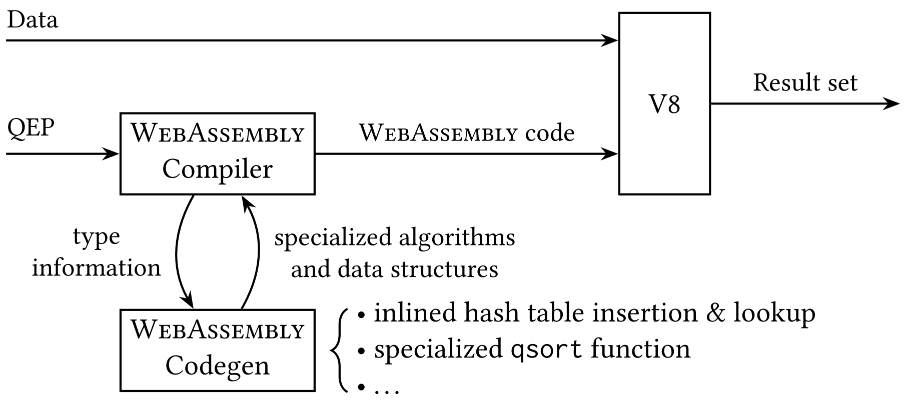

Data
QEP
WebAssembly Compiler
WebAssembly code
V8
Result set
type information
specialized algorithms and data structures
WebAssembly Codegen
• inlined hash table insertion & lookup
• specialized qsort function
• …

Figure 3.4: Compilation with on-demand code generation in mu*t*able. QEPs are compiled to WebAssembly and dispatched to V8. Specialized code is generated on demand for algorithms and data structures required by the QEP.

3.5.2 Our Approach: JIT Code Generation

Full compilation is very similar to our approach of generating required library routines just in time and JIT compiling the QEP. The key distinction is how code is generated. Previous work generates code in a high-level language. This code must then go through parsing and semantic analysis before it is translated to a lower level IR where optimizations are performed before executable machine code is produced. Going through the entire compiler machinery takes a lot of time. Our approach, depicted in Figure 3.4, bypasses most of these steps. We generate specialized algorithms and data structures directly in WebAssembly. By picking a suitable WebAssembly engine, e.g. V8, we fulfill all the requirements given in Section 3.2.2. Our approach is able to produce highly specialized algorithms and data structures and enables holistic optimization without the drawback of long code generation and compilation times.

3.5.3 Code Generation by Example

To provide the reader with a better understanding of how we generate library code just in time, let us exemplify our code generation along the example of Quicksort. We build the example bottom up, beginning with code generation for partitioning and the comparison of two elements before we explain code generation of the recursive Quicksort algorithm. We use pseudocode, as it is easier to read and understand than Wasm and because our approach of ad-hoc code generation is independent of a particular language.

Hoare's partitioning scheme. Hoare's partitioning scheme creates two partitions from a sequence of elements based on a boolean predicate such that all elements in the first

Listing 3.4 Pseudocode for the generation of specialized code that implements Hoare’s partitioning.

```
 1: function Partition($order$, $begin$, $end$, $pivot$)
 2:     Emit($l \leftarrow begin$)
 3:     Emit($r \leftarrow end$)
 4:     Emit(while $l < r$)
 5:     EmitSwap($l$, $r - 1$)
 6:     $cl \leftarrow$ EmitCompare($order$, $l$, $pivot$)
 7:     $cr \leftarrow$ EmitCompare($order$, $pivot$, $r - 1$)
 8:     Emit($l \leftarrow l + cl$)
 9:     Emit($r \leftarrow r - cr$)
10:     Emit(end while)
11:     return $l$
12: end function
```

partition do not satisfy the predicate and all elements in the second partition satisfy the predicate. We apply Hoare’s partitioning in our generated Quicksort algorithm, that in turn is used to implement sorting of tuples. In our setting, the sequence of tuples to partition is a consecutive array.

We provide pseudocode for the generation of specialized partitioning code in Listing 3.4. The function Partition takes four parameters: the *order* is a list of expressions to order by, *begin* and *end* are variables holding the address of the first respectively one after the last tuple in the array to partition, and *pivot* is a variable holding the address of the pivot to partition by. The pivot must not be in the range $[begin, end)$. First, the algorithm copies the values of *begin* and *end* by introducing fresh variables l and r in lines 2 and 3 and then emitting code that assigns the value of *begin* to l in line 4 and the value of *end* to r in line 5. Next, in line 6, a loop header with the condition $l < r$ is emitted. The code emitted thereafter forms the loop body. In line 7, EmitSwap is called to emit code that swaps the tuples at the addresses l and $r - 1$. Note that this is a function call *during code generation*. The call will emit code directly into the loop body, as if inlined by an optimizing compiler, and there will be no function call during execution of the generated code. In lines 8 and 9, EmitCompare is called to emit code that compares the tuples at addresses l and $r - 1$ to the tuple at address *pivot* according to the order specified by *order*. Each call returns a fresh boolean variable that holds the outcome of the comparison. Just like EmitSwap, calls to EmitCompare emit code directly into the loop body without the need for a function call in the generated code. The value of variable cl will be true if the tuple at address l compares less than the tuple at address *pivot* w.r.t. the specified *order*. Line 10 emits code that advances l to the next tuple if cl is true, otherwise l is not changed. Similarly, line 11 emits code to advance r to the previous tuple if cr is true. This is a means of implementing branch-free partitioning. In line 12, the loop body for the loop emitted in line 6 is

Listing 3.5 Pseudocode for the generation of code that compares two elements based on a specified order.

```
 1: function EmitCompare($order$, $l$, $r$)
 2:     Emit($v \leftarrow 0$)
 3:     for each $expr$ in $order$ do
 4:         $vl \leftarrow$ Compile($expr$, $l$)
 5:         $vr \leftarrow$ Compile($expr$, $r$)
 6:         switch $type(expr)$ do
 7:             case int
 8:                 Emit($lt \leftarrow vl <_{\text{int}} vr$)
 9:                 Emit($gt \leftarrow vl >_{\text{int}} vr$)
10:             case float
11:                 Emit($lt \leftarrow vl <_{\text{float}} vt$)
12:                 Emit($gt \leftarrow vl >_{\text{float}} vt$)
13:             ...                                    ▹cases for remaining types
14:         end switch
15:         Emit($v \leftarrow 2 \cdot v + gt - lt$)
16:     end for
17:     Emit($c \leftarrow v < 0$)
18:     return $c$
19: end function
```

finished. Eventually, Partition returns the variable *l*, which will point to the beginning of the second partition once the loop of line 6 terminates.

The code presented in Listing 3.4 looks almost like a regular implementation of partitioning. However, the function *emits* code that will perform partitioning. An important part of partitioning, that we skipped in Listing 3.4, is how the code to compare two tuples based on a given order is generated. Therefore, we also provide pseudocode for EmitCompare in Listing 3.5.

First, EmitCompare creates a fresh variable v in line 2 and initializes it to 0 in line 3. Then, in line 4, the function iterates over all expressions in *order*. The call to Compile in line 5 emits code to evaluate *expr* on the tuple pointed to by l and returns a fresh variable holding the value of the expression. Analogously, line 6 evaluates *expr* on the tuple pointed to by r. Next, type-specific code to compare the values vl and vr of the evaluated expression is emitted. Because the particular code to emit depends on the type of *expr*, line 7 performs a case distinction on the type. This case distinction is performed *during code generation* and the generated code will only contain the emitted, type-specific code. In case the expression evaluates to an `int`, lines 9 and 10 emit code to perform an integer comparison of vl and vr. The cases for other types are analogous. After emitting type-specific code for the comparison of vl and vr, line 16 emits code to update v based on the outcome of the comparison. After generating code to evaluate all expressions in *order* and updating v accordingly, lines 18 and 19 introduce a fresh boolean

Listing 3.6 Generated partitioning code for the order $[R.x + R.y, R.z]$.

```
Input: b, e, p
Output: $p_l$
1:  var $p_l \leftarrow$ b                                          ▷ Initialize pointers to the first and
2:  var $p_r \leftarrow$ e                                          ▷ one after the last tuple, respectively.
3:  while $p_l < p_r$ do
4:      ▷ EmitSwap($p_l$, $p_r - 1$)
5:      var $v_{tmp} \leftarrow *p_l$                                ▷ Use temporary variable
6:      $*p_l \leftarrow *(p_r - 1)$                                 ▷ to swap tuples
7:      $*(p_r - 1) \leftarrow v_{tmp}$                              ▷ at $p_l$ and $p_r - 1$.
8:      ▷ EmitCompare($[R.x + R.y, R.z]$, $p_l$, p)
9:      var $v_{l,pivot} \leftarrow 0$
10:     var $v_l \leftarrow p_l.x +_{\texttt{int}} p_l.y$           ▷ Compile($R.x + R.y$, $p_l$)
11:     var $v_{pivot} \leftarrow \mathrm{p}.x +_{\texttt{int}} \mathrm{p}.y$   ▷ Compile($R.x + R.y$, p)
12:     var $v_{lt} \leftarrow v_l <_{\texttt{int}} v_{pivot}$
13:     var $v_{gt} \leftarrow v_l >_{\texttt{int}} v_{pivot}$
14:     $v_{l,pivot} \leftarrow 2 \cdot v_{l,pivot} + v_{gt} - v_{lt}$
15:     var $v_l \leftarrow p_l.z$                                  ▷ Compile($R.z$, $p_l$)
16:     var $v_{pivot} \leftarrow \mathrm{p}.z$                      ▷ Compile($R.z$, p)
17:     var $v_{lt} \leftarrow v_l <_{\texttt{int}} v_{pivot}$
18:     var $v_{gt} \leftarrow v_l >_{\texttt{int}} v_{pivot}$
19:     $v_{l,pivot} \leftarrow 2 \cdot v_{l,pivot} + v_{gt} - v_{lt}$
20:     var $v_{cl} \leftarrow v_{l,pivot} < 0$
21:     ▷ EmitCompare($[R.x + R.y, R.z]$, p, $p_r - 1$)
22:     ...                                                         ▷ Code omitted for brevity.
23:     var $v_{cr} \leftarrow v_{pivot,r} < 0$
24:     $p_l \leftarrow p_l + v_{cl}$                                ▷ Advance left cursor.
25:     $p_r \leftarrow p_r - v_{cr}$                                ▷ Advance right cursor.
26: end while
```

variable c that will be set to $v < 0$, which evaluates to true if the tuple at l is strictly smaller than the tuple at r.

To put it all together, let us exercise an example. We invoke Partition with the *order* $[R.x + R.y, R.z]$, *begin* 'b', *end* 'e', and *pivot* 'p'. The generated code is given in Listing 3.6. Initially, in lines 1 and 2, the addresses of the first and one after the last tuple are stored in fresh variables. Then the loop in line 3 repeats as long as pointer p_l points to an address smaller than p_r. Lines 5 to 7 show the code produced by EmitSwap, that swaps two tuples using a temporary variable. In lines 9 to 20, the tuple at p_l is compared to the pivot according to the specified order. Variable v_{cl} is true if the tuple at p_l compares less than the pivot, false otherwise. Analogously, the tuple at $p_r - 1$ is compared to the pivot. To keep the example short and because the code is very similar, we omit this code and only show a place holder in line 22. At the end of the loop, in lines 24 and 25, the pointers p_l and p_r are advanced depending on the outcome of the comparisons.

Listing 3.7 Pseudocode to generate specialized Quicksort.

```
 1: function Quicksort(order)
 2:     Emit(function qsort(begin, end))
 3:     Emit(while $end - begin > 2$)
 4:     Emit($mid \leftarrow begin + (end - begin)/2$)
 5:     $m \leftarrow$ EmitMedianOf3($begin, mid, end - 1$)
 6:     EmitSwap($begin, m$)
 7:     $mid \leftarrow$ Partition($order, begin + 1, end, begin$)
 8:     EmitSwap($begin, mid - 1$)
 9:     Emit(if $end - mid \geq 2$)
10:     Emit(qsort($mid, end$))
11:     Emit(end if)
12:     Emit($end \leftarrow mid - 1$)
13:     Emit(end while)
14:     Emit(end function)
15: end function
```

The generated code will partition the range [b, e) such that the first partition contains only tuples that compare less than p and the second partition contains only tuples greater than or equal to p, w.r.t. the specified order. Note that the generated code is not a function. Instead, this code can be generated into a function where partitioning is needed. Hence, the entire code for partitioning will always be fully inlined and specialized for the order to partition by.

Quicksort. Quicksort sorts its input sequence by recursive partitioning. In our implementation of Quicksort, we compute the pivot to partition by as a *median-of-three*. With our code generation for partitioning at hand, generating Quicksort is relatively simple. We provide pseudocode in Listing 3.7. Line 2 defines a new function `qsort`, line 3 emits a loop that repeats as long as there are more than two elements in the range from *begin* to *end*. Inside this loop, lines 4 to 7 emit code to compute the median of three and bring the median to the front of the sequence to sort. Line 8 emits the code to partition the sequence $begin + 1$ to *end* using as pivot the median of three. After partitioning, the median must be swapped back into the partitioned sequence, which is done by line 9. Line 10 checks whether to recurse into the right partition. Line 11 emits a recursive call to sort the right partition with `qsort`. Afterward, in line 13, code is emitted to update *end* to the end of the left partition.

We can see that by executing our Quicksort code generation, we obtain a specialized, fully inlined `qsort` function that can be called to sort a sequence by the *order* specified during code generation.

3.6 Executing WebAssembly in a Database System

In the preceding sections, we explained how to compile a QEP and its required libraries to WebAssembly. In this section, we elaborate how we execute WebAssembly in an embedded engine. Although this approach works with any embeddable engine, we describe the process of embedding and executing WebAssembly in V8.

The WebAssembly specification requires that each *module* – think of translation unit in C – operates on its personal memory. This memory is provided by the engine, here V8. To execute a compiled QEP inside the engine, all required data (tables, indexes, etc.) must reside in the module's memory. One way to achieve this is by copying all data from the host to the module's memory. However, this incurs an unacceptable overhead of copying potentially large amounts of data before executing the QEP. An alternative is to use callbacks from the module to the host to transfer single data items on demand. For such a purpose, V8 allows for defining functions in the embedder that can be called from the embedded code. However, such callbacks also incur a tremendous overhead, because the VM has to convert parameters and the return value from the representation in embedded code to the representation in the embedder and vice versa. At the time of writing, V8 provides no method to use pre-allocated host memory as a module's memory. Therefore, we patch V8 to add a function for exactly that purpose: `SetModuleMemory()` sets the memory of a WebAssembly module to a region of the host memory. While this function enables us to provide a single consecutive memory region from the host to the module, it is not sufficient to provide multiple tables or indexes (which need not reside in a single consecutive allocation) to a module. The problem is that WebAssembly – in its current version – only supports 32 bit addressing. Hence, we cannot simply assign the entire host memory to the module. Instead, we are limited to 4 GiB of addressable *linear memory* inside the module. In the following, we describe pagination implemented with a technique named *rewiring* to work around this limitation. However, we want to stress that 64 bit addresses in WebAssembly are on the way and pagination will become obsolete eventually.

3.6.1 Accessing Data by Rewiring

Rewiring [SDS16] allows for manipulating the mapping of virtual address space to physical memory from user space. In particular, it enables us to map the same *physical* memory at two distinct *virtual* addresses. We exploit this technique to have data structures residing in distinct allocations appear consecutively in virtual address space and then use this address range as the module's memory.

We exemplify this technique in Figure 3.5. Assume a query accessing two tables `A` and `B`. The tables reside in completely independent memory allocations, hence there is no single

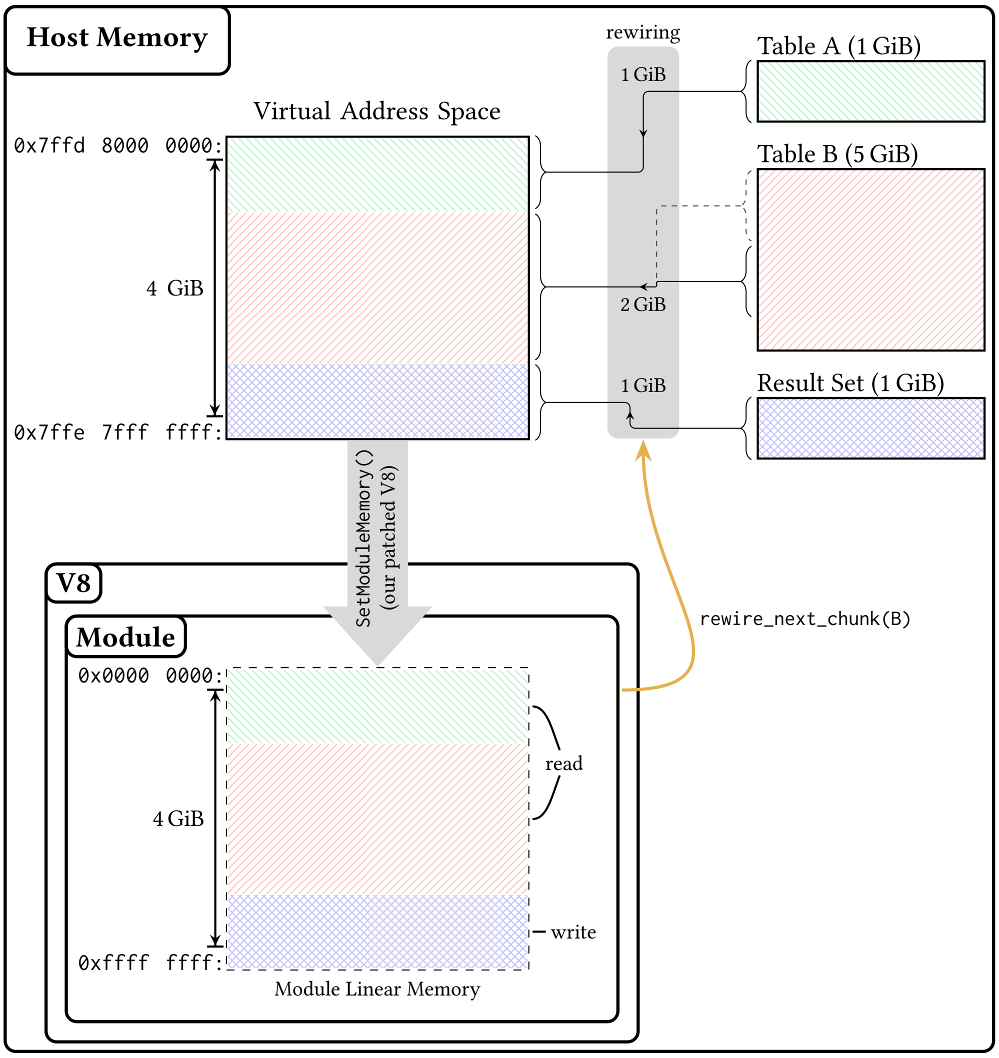


Figure 3.5: Example of mapping tables and output to a module's memory. The module can callback to the host to request mapping the next 2 GiB chunk of table B.

4 GiB virtual address range that contains both A and B entirely. Further, the query computes some results, and we therefore allocate 1 GiB of memory to store the query's result set. To give the module access to all required memory, we first allocate consecutive 4 GiB in virtual address space. Then we *rewire* table A, a portion of table B, and the memory for the result set into the freshly allocated virtual memory. Finally, we call `SetModuleMemory()` with the freshly allocated virtual memory. The module now has access to both tables and can write its results to the memory allocated for the result set. Note that table B is 5 GiB and cannot be rewired entirely into the virtual memory for the module. To give the module access to the entire table,

we install a callback `rewire_next_chunk()` that lets the host rewire the next 2 GiB chunk of table `B`, thereby allowing the module to iteratively process entire `B`.

3.6.2 Result Set Retrieval

Similarly to how data is made available to the module, we use rewiring to communicate the result set back to the host. As can be seen in Figure 3.5, the module writes the result set to a rewired allocation of 1 GiB. If the module produces a result set of more than 1 GiB, it produces the result set in chunks and issues a callback in between to have the host process the current chunk of results.

3.7 Related Work

In addition to our motivation for using V8 and WebAssembly in Section 3.2.2 and Section 3.3, respectively, we present in Section 3.7.1 JIT frameworks and engines that might be used alternatively. In Section 3.7.2, we augment the comparison with related work that is conducted throughout Section 3.2.1, Section 3.4.3, and Section 3.5.1.

3.7.1 JIT Frameworks & Engines

The idea to build query execution on top of a JIT engine occurred as early as 1997 in the context of JIT compilation in the Java Virtual Machine (JVM) [Cra+97]. These thoughts were later implemented in Java's HotSpot VM [Kot+08] and are still being developed today in GraalVM [Ora12]. Let us have a quick look at JIT compilers and engines we considered: MIR [Mak] provides an IR with an interpreter and fast JIT compiler, however it is still in early development by only a small community and does not provide adaptive execution out of the box. LibJIT [Fre] provides a framework for on-demand JIT compilation and reoptimization. However, it is lacking automation of adaptive execution and inlining. When we focus on executing WebAssembly, we have access to a rich set of engines. For brevity, we only mention a few examples: Wasmer [Was] is a feature-rich WebAssembly runtime but does not provide efficient embedding. SpiderMonkey is Mozilla's JavaScript and WebAssembly engine and quite similar to Google's V8. We chose V8 over SpiderMonkey because of its better documentation and because it is written in C++, like our database system mu*t*able.

3.7.2 Query Execution

Interpretation. Graefe [Gra94] proposes a unified and extensible interface for the implementation of relational operators in Volcano, named *iterator* interface. Ailamaki et al. [Ail+99] analyze query execution on modern CPUs and find that poor data and instruction locality as well as frequent branch misprediction impede the CPU from processing at peak performance. Boncz, Zukowski, and Nes [BZN05b] identify tuple-at-a-time processing as a limiting factor of the VOLCANO iterator design, that leads to high interpretation overheads and prohibits data parallel execution. To overcome these limitations, Boncz, Zukowski, and Nes [BZN05b] propose *vectorized* query processing, implemented in the X100 query engine within the column-oriented MONETDB system. Menon, Mowry, and Pavlo [MMP17] build on the vectorized model and introduce *stages* to dissect pipelines into sequences of operators that can be *fused*. By fusing operators, Menon, Mowry, and Pavlo are able to vectorize multiple sequential relational operators. Their implementation in PELOTON [Pav+17] shows that operator fusion increases the degree of inter-tuple parallelism exploited by the CPU.

Compilation. **Rao2006compiled** explore compilation of QEPs to JAVA and having the JVM JIT-compile and load the generated code. However, their approach sticks to the VOLCANO iterator model, restricting compilation from unfolding its full potential. Follow-up work explores compiling QEPs to vectorized Java code in Spark [ALX16]. Potential compilation overheads are not discussed. Schiavio, Bonetta, and Binder [SBB21] and Grulich, Zeuch, and Markl [GZM21] both recently explored the support of polyglot UDFs. Both approaches build on TRUFFLE, GRAALVM's compiler-compiler. We believe the JAVASCRIPT + WEBASSEMBLY eco system could very well support polyglot programming, too. There already exists a wide range of transpilers, e.g. TRANSCRIPT translates PYTHON to JAVASCRIPT or EMSCRIPTEN compiles LLVM-based languages to WEBASSEMBLY. With HIQUE, Krikellas, Viglas, and Cintra [KVC10] propose query compilation to C++ code by dynamically instantiating operator templates in topological order. They report query compilation times in the hundreds of milliseconds. Neumann [Neu11] presents compilation of pipelines in the QEP to tight loops in LLVM. Complex algorithms are implemented in C++ and pre-compiled, to be linked with and used by the compiled query. With the implementation in HYPER, Neumann achieves significantly reduced compilation times in the tens of milliseconds. Klonatos et al. [Klo+14] address the system complexity and the associated development effort of compiling query engines in their LEGOBASE system, where metaprogramming is used to write a query engine in Scala LMS that, when partially evaluated on an input QEP, yields specialized C code that implements the query. Despite the clean design, the code generation through partial evaluation as well as the compilation of the generated code leads to compilation times in the order of seconds.

Adaptive. A recent advancement in query execution is adaptive execution by Kohn, Leis, and Neumann [KLN18] and Kersten, Leis, and Neumann [KLN21], that we already discussed in great detail in Section 3.2.1.

3.8 Evaluation

In this section, we want to experimentally verify that our architecture of embedding an off-the-shelf JIT engine provides competitive performance to state-of-the-art systems. We want to stress that our goal is *not* to outperform existing systems but to demonstrate that our approach enables us to achieve similar performance at much lower engineering costs.[2] We begin by evaluating the performance of QEP building blocks, then we survey TPC-H queries, and finally we examine compilation times.

3.8.1 Experimental Setup

We implement our approach in mu*t*able [Haf+23], a main-memory database system developed by the Big Data Analytics Group at Saarland University. Although mu*t*able supports arbitrary data layouts, we conduct all experiments using a PAX layout with 4 MiB block. Since mu*t*able does not yet support multi-threading, all queries run on a single core.

We compare to three systems: (1) PostgreSQL 15.4 as representative for Volcano-style tuple-at-a-time processing, (2) DuckDB v0.8.1, implementing the vectorized model as in MonetDB/X100, and (3) HyPer, an adaptive system performing interpretation and compilation of LLVM bytecode, as provided by the `tableauhyperapi` Python package in version 0.0.17360. For PostgreSQL, we *disable* JIT compilation of expressions. Enabling JIT compilation in PostgreSQL did deteriorate execution times in all of our experiments. Further, we disable *write-ahead logging (WAL)* completely and explicitly create all tables with parameter `UNLOGGED` to remedy *ACID*-related overheads to query execution within PostgreSQL. While earlier versions of HyPer implemented the approach of Kohn, Leis, and Neumann [KLN18], recent versions of HyPer implement the approach of Kersten, Leis, and Neumann [KLN21], which cannot be disabled.[3] Further, with a recent update, HyPer's log file does not contain detailed information on time spent on interpretation, non-optimizing compilation, or optimizing compilation anymore. HyPer only reports compilation time and execution time, and it is not possible anymore to observe whether interpreted execution happened before compiled execution. We therefore report the overall execution time as given in HyPer's log file, and, if present, the compilation time. We run all our experiments on a machine with an AMD Ryzen 7800X3D with 8 physical

[2]Please be aware that performance differences are not only due to architectural differences but – much more likely – due to different implementations of the same algebraic operations. In mu*t*able we only use text-book implementations of algebraic operations.

[3]This is in stark contrast to the evaluation conducted in my paper corresponding to this chapter [HD23a].

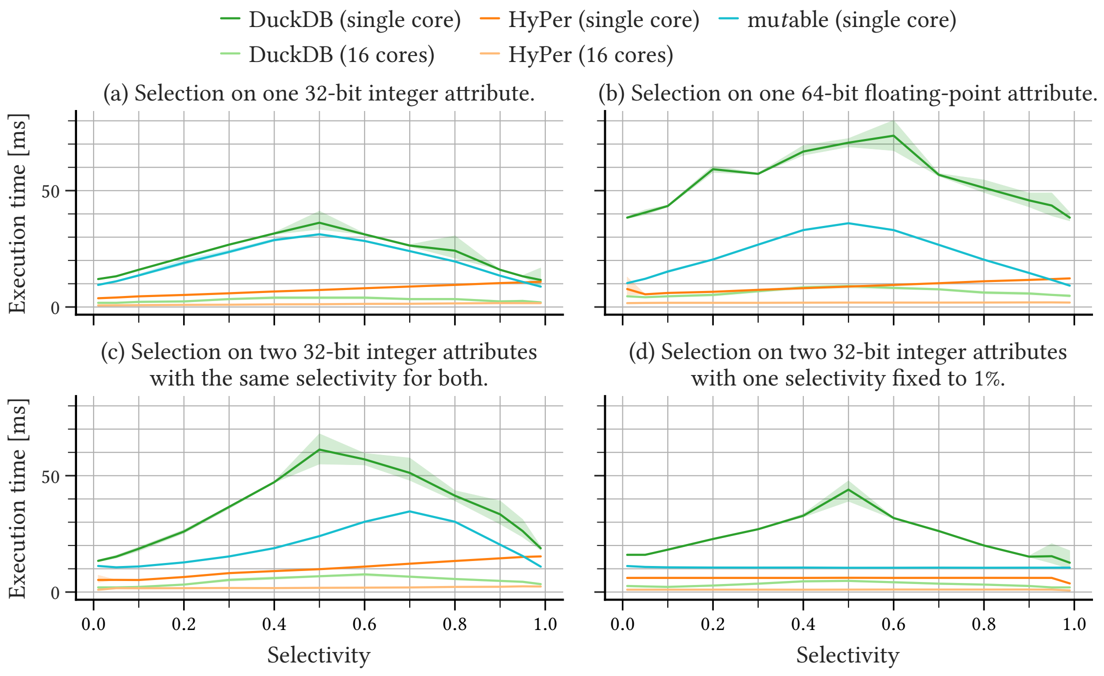


Figure 3.6: Evaluation of selection with one and two one-sided range predicates.

cores at 4.2 GHz base clock, 96 MiB L3 cache, and 32 GiB main memory. All data accessed in the experiments is memory resident. We repeat each experiment five times.

3.8.2 Performance of Query Building Blocks

With our first set of experiments, we evaluate the performance of individual query building blocks across different systems. We use a generated data set with multiple tables and 10 million rows per table. Tables contain only integer and floating-point columns, where integer values are chosen uniformly at random from the entire integer domain $[-2^{31}; 2^{31} - 1]$ and floating-point values are chosen uniformly at random from the range $[0; 1]$. All columns are individually shuffled and hence all columns are pairwise independent (uncorrelated). For HyPer and mu*t*able, we report only execution times without compilation times. We further enforce compilation with the optimizing TurboFan compiler.

Selection. In our first set of experiments, we run several `COUNT(*)`-queries with different `WHERE`-clauses to evaluate the performance of selection. In Listing 3.8, we exemplify the structure of these queries. Figure 3.6 (a) and (b) show our measurements for selection on a 32-bit integer and a 64-bit floating-point column, respectively. We omit our findings for PostgreSQL, as the times are above 200 ms. Both mu*t*able and DuckDB implement selection by conditional

Listing 3.8 Example queries for our experiments in Figure 3.6. The numeric constants are varied to achieve the desired selectivity of the selection predicate.

```
SELECT COUNT(*) FROM T WHERE i32 < 42;                          -- Fig. 3.6 (a)
SELECT COUNT(*) FROM T WHERE f64 < 0.42;                        -- Fig. 3.6 (b)
SELECT COUNT(*) FROM T WHERE i32a < 42 AND i32b < 42;           -- Fig. 3.6 (c)
SELECT COUNT(*) FROM T WHERE i32a < 42 AND i32b < -2104533975;  -- Fig. 3.6 (d)
```

branching. Therefore, both systems suffer from frequent branch misprediction at selectivities around 50% [SZB11; Ros02]. The execution time of HyPer remains unaffected by varying selectivity; our educated guess is that HyPer compiles branch-free code. We can see that mu*t*able outperforms DuckDB on all selectivities and for both integer and floating-point columns. This is likely the case because DuckDB, which implements the vectorized execution model, has the overhead of maintaining a selection vector [SZB11; Pir+16].

We conduct two additional experiments, where we perform a selection on two, pairwise-independent integer columns. In our third experiment, shown in Figure 3.6 (c), both conditions are varied with equal selectivity. This means, the overall selectivity of the selection is the squared selectivity of either condition. Since mu*t*able does not implement short-circuit evaluation and instead evaluates the selection predicate as a whole, a selectivity of $\sqrt{50\%} \approx 71\%$ per condition presents the worst-case for branch prediction with a time of 35 ms. DuckDB, which implements the vectorized model, must first evaluate one condition to a selection vector before evaluating the second condition on the selected rows. Because the conditions are evaluated individually, branch misprediction occurs up to twice as often and branch prediction is worst at a selectivity of 50% per condition with an execution time around 61 ms. As the selectivity grows, the second condition must be evaluated more often. This can be seen in the slight asymmetry of the execution time curve. HyPer's execution time significantly grows with the selectivity from around 5 ms at 0% to 17 ms at 100%. HyPer again produces branch-free code. However, it appears that the second column is only accessed if the first condition is satisfied, causing an increase in execution time with increasing selectivity.

In our fourth experiment, shown in Figure 3.6 (d), only one condition is varied while the other is fixed to a selectivity of 1%. The overall selectivity of the selection is hence in the range from 0% to 1%. Since mu*t*able evaluates the entire selection predicate as a whole, branch prediction works reliably well, and we observe a constant execution time of around 11 ms. DuckDB appears to evaluate the condition with varying selectivity first, leading to branch misprediction and execution times of up to 45 ms.

Listing 3.9 Example queries for our experiments in Figure 3.7.

```
SELECT COUNT(DISTINCT i32) FROM T;                     -- Fig. 3.7 (a) & (b)
SELECT COUNT(DISTINCT (i32a, i32b, i32c)) FROM T;      -- Fig. 3.7 (c)
SELECT SUM(i32a), SUM(i32b), SUM(i32c) FROM T GROUP BY i32; -- Fig. 3.7 (d)
```

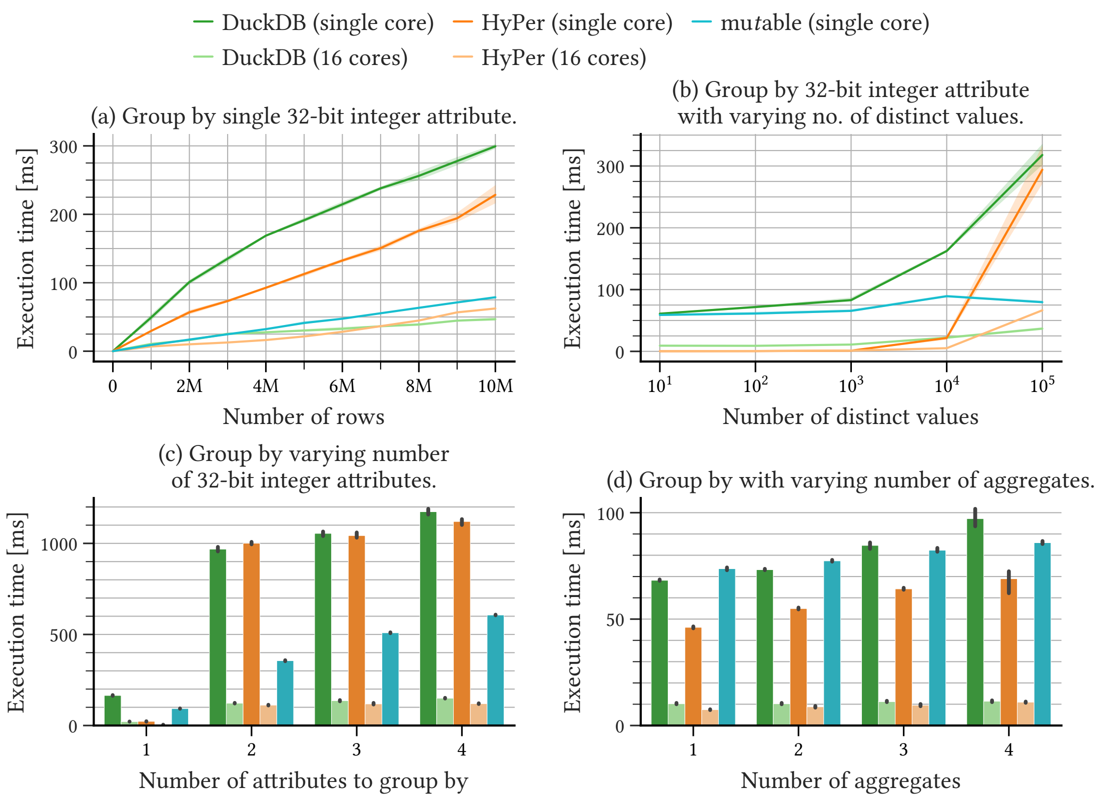


Figure 3.7: Evaluation of grouping & aggregation.

Grouping & Aggregation. Our next experiment evaluates the performance of grouping and aggregation. We run several `COUNT(DISTINCT ...)` and aggregate queries. In Listing 3.9, we exemplify the structure of these queries. We vary the experiment in several dimensions: (a) the number of rows in the table, (b) the number of distinct values in the column being grouped by, (c) the number of attributes to group by, and (d) the number of aggregates to compute. Figure 3.7 presents our findings.

In Figure 3.7 (a) to (c), a lion share of execution time is spent on hash table operations. mu*t*able generates a specialized hash table implementation per query, with all hash table operations fully inlined into the query code, and is thereby able to gain an advantage over the other systems. We must note that HyPer achieves *impossible* execution times in Figure 3.7 (b)

Listing 3.10 Example queries for our experiments in Figure 3.8.

```sql
SELECT id FROM T ORDER BY i32;              -- Fig. 3.8 (a) & (b)
SELECT id FROM T ORDER BY i32a, i32b, i32c; -- Fig. 3.8 (c)
```

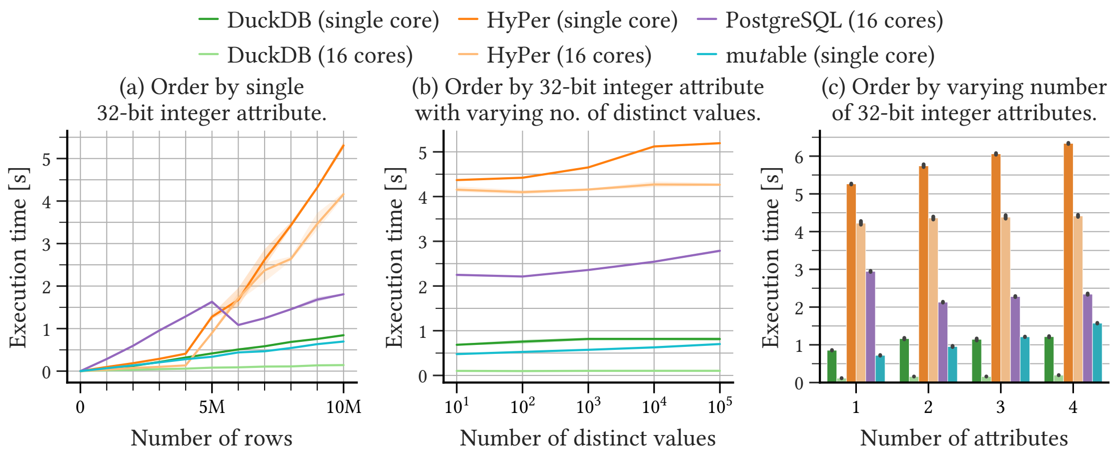


Figure 3.8: Evaluation of sorting.

for 10 to 1000 distinct values: We believe that HyPer answers our queries from internal statistics rather than actually executing the query.

In Figure 3.7 (d) we evaluate the performance of aggregation by altering the SELECT-clause to compute a varying number of aggregates. The time for hash table lookup is dwarfed by the time to compute the aggregates.

Sorting. Our next experiment evaluates the performance of sorting, as needed in ORDER BY-clauses or for merge-join. Similar to the experiment on grouping, we vary the experiment in several dimensions: the number of rows in the table, the number of attributes to order by, and the number of distinct values in the column to order by. For evaluation, we execute queries as exemplified in Listing 3.10, and Figure 3.8 presents our findings. We can observe a slight improvement over DuckDB and a significant improvement over HyPer and PostgreSQL, that we credit to mu*t*able's generation and consequent holistic optimization of the sorting operation, described in detail in Section 3.5.3. In contrast to interpretation or the use of a pre-compiled library, in Quicksort generated by mu*t*able the pair-wise comparison of elements and the entire routine for partitioning are fully inlined and specialized to the elements' types. No callbacks are required to compare elements and no dynamic dispatches are required to determine an element's runtime type.

Listing 3.11 Example queries for our experiments in Figure 3.9.

```sql
SELECT COUNT(*) FROM T JOIN S ON T.id = S.tid; -- Fig. 3.9 (a)
SELECT COUNT(*) FROM T JOIN S ON T.x = S.x;    -- Fig. 3.9 (b)
```

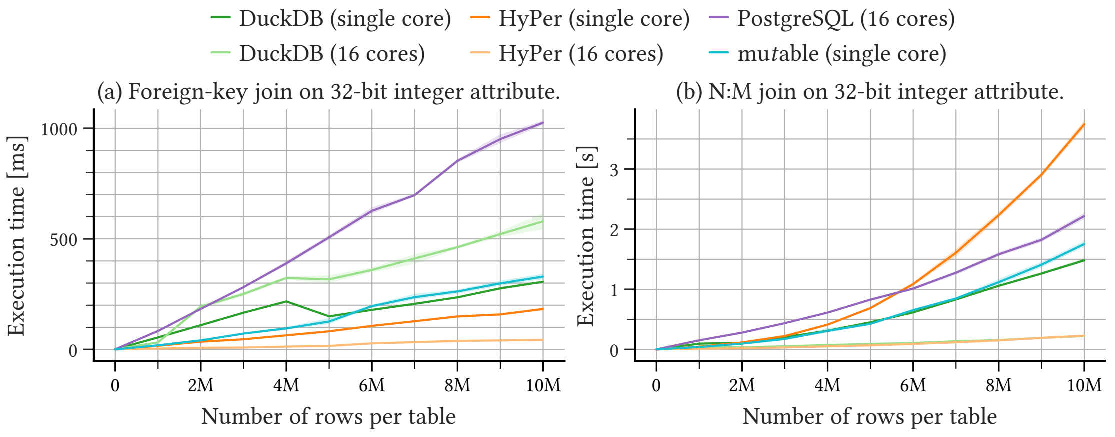


Figure 3.9: Evaluation of equi-join.

Equi-Join. In this experiment, we evaluate the performance of a foreign key join and an *n*:*m* equi-join. We sketch the structure of our queries in Listing 3.11. We perform the *n*:*m* join on non-key columns to avoid the systems using a pre-built index, since mu*t*able does not yet support indices.[4] We fix the selectivity of the joins to 10^{-6} and vary the size of the input relations. We present our findings in Figure 3.9. In (a), we see the expected linear behavior for all systems. There is a slight decrease in execution time from 4M to 5M rows for DuckDB. This is likely due to pre-allocation of an internal hash table for the hash join and the resulting load factor of that hash table. In (b), we observe the expected quadratic behavior. Notable here is that HyPer exhibits the strongest curvature and becomes the slowest of the systems from around 4M rows onwards. Our educated guess is that duplicates w.r.t. the join predicate lead to long collision chains in HyPer's hash table.

3.8.3 TPC-H

So far, our experiments only focus on individual query building blocks. Next, we conduct an experimental evaluation of TPC-H queries. By the time of writing, mu*t*able – and in particular our WebAssembly backend – only supports a subset of SQL and hence we are only able to evaluate the TPC-H queries 1, 3, 6, 12, and 14. We are further constrained to SF 1, as for

[4]In particular, mu*t*able cannot map non-consecutive data structures like indices from process memory into the WebAssembly VM. This is future work.

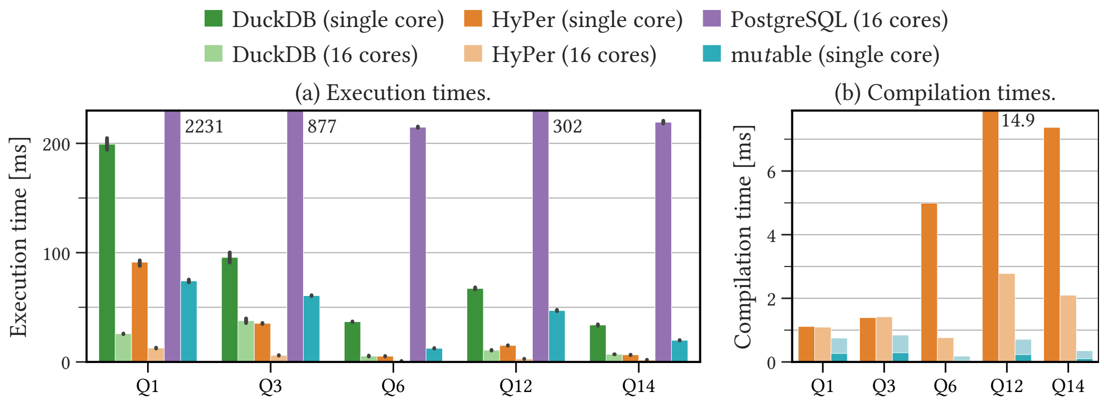


Figure 3.10: Evaluation of TPC-H queries on an SF 1 database. Compilation times for mu*t*able are split into the time to compile SQL to WebAssembly (● dark) and the time to compile WebAssembly to machine code (● light).

larger scale factors the data structures constructed during execution exceed the Wasm linear memory. We are currently unable to exploit the technique described in Section 3.6 for data structures; it is only applied to rewire single, consecutive memory regions. For HyPer, we extract compilation and execution times from its log file. For mu*t*able, we provide detailed timings for the translation of the QEP to WebAssembly and the compilation and execution of WebAssembly with TurboFan. We present our findings in Figure 3.10.

With regard to compilation times, mu*t*able's optimizing compilation with TurboFan is up to 27.4x faster than HyPer's compilation (Q6). Note that mu*t*able's compilation times include the *generation and compilation* of required algorithms and data structures, e.g. hash table operations. At the same time, mu*t*able's execution times are competitive to HyPer's – except for Q14 where HyPer on a single core significantly outperforms all other systems. It is worth mentioning that the reason why HyPer compiles faster on 16 cores is likely that no optimizing compilation is performed. This behavior matches our observations in our paper [HD23a], where we evaluated HyPer when it was still implementing the approach of Kohn, Leis, and Neumann [KLN18] and the produced log file contained detailed timings.

3.9 Conclusion

Our goal was to simplify the architecture of query execution engines while fulfilling the high-level requirements of low latency, high throughput, and adaptive execution. We proposed to embed a suitable off-the-shelf JIT engine into the database system to delegate the execution of QEPs to. By compiling QEPs to an efficient IR and delegating execution to said engine, we have met that goal. Our architecture requires much less engineering and maintenance than previous solutions. At the same time, our experimental evaluation confirms that we achieve low latency because of fast JIT code generation and high throughput because of adaptive (re-) optimization during query execution – all fully handled by the embedded engine.

We are convinced that our approach is considerably simpler to understand and implement than current state-of-the-art. By relying on successful, battle-tested infrastructure for JIT compilation and execution, we significantly reduce the required development effort to build an adaptive yet highly efficient query execution engine. With the ongoing standardization of WebAssembly [MDN; W3C] and the immense interest and amount of ongoing work in engines supporting this language [V8 08; Moz; Byt; Was; App], our approach provides a reliable and future-proof solution to adaptive query execution.

Further, we think that our ad-hoc generation of specialized algorithms and data structures shapes a new path for query compilation, potentially leading to much more efficient query processing than was possible so far.

Chapter 4

mu*t*able - A Modern Database System for Research & Fast Prototyping

This chapter is based on my publication "mutable - A Modern DBMS for Research and Fast Prototyping" [HD23c]. This work was published in the research track of CIDR 2023. This work was co-authored by my Ph.D. advisor Prof. Dr. Jens Dittrich.

4.1 Introduction

Bobby Tables is a young Ph.D. student in the field of database systems, who just recently started doing her own research. Bobby has an idea how to compute join orders more efficiently. At some point, Bobby has to implement her idea to perform an empirical evaluation. Bobby now has to make an important decision: She can (1) implement her idea in an existing (open-source) system or (2) implement her idea stand-alone, i.e. not integrating it into a system. Bobby asks for guidance from her Ph.D. advisor, who tells her that there are strengths and weaknesses to either approach and presents the following arguments regarding implementation in an existing system:

+ Bobby can save some development effort by implementing her approach in an existing system. She will be able to rely on a rich infrastructure taking care of many things unrelated to her topic of research, e.g., parsing and semantic analysis, concurrency control, buffer management, storage, or query execution.

+ When related algorithms have been implemented in the same system, Bobby can use them "off the shelf" for her evaluation.

+ There is *always* this one reviewer that expects you to evaluate your approach in a *real database system*. Bobby can consider it done.
– System-specific design decisions may (negatively) affect experiments. In the case of join ordering, a system with particularly slow query execution may dwarf Bobby's improvements over related works when comparing end-to-end workload execution times.
– Bobby may have to re-implement related algorithms in the chosen system, somewhat contradicting the argument of saving development effort.
– Alternatively, Bobby can evaluate implementations in other systems. However, this has the significant downside of leading to an "apples to oranges" comparison. In the case of join ordering, different systems may use different cost models, different cardinality estimation, or simply deploy more or less efficient data structures. All these factors can obfuscate Bobby's empirical findings.

After some consideration, Bobby decides to implement her join ordering algorithm in an existing system. But which system should she choose? Bobby searches online for some candidates and quickly finds the *Database of Databases* (dbdb.io). The site lists a whopping **875 database systems** as of December 2022. Bobby is convinced that she will find a suitable system for her implementation among these many candidates. She uses the filter to refine her search to systems of "Academic" or "Educational" type and having a relational data model. Surprisingly, from the initial 875 systems **only 34** remain. Still, Bobby is confident as she spots some famous research projects within the list: PostgreSQL, HyPer, MonetDB, DuckDB, and NoisePage, to name a few. Bobby takes a closer look at the open-source systems to understand how she can implement and integrate her join ordering algorithm. She finds that all systems provide some form of (online) documentation. However, the documentation is mainly targeted at database users and administrators. Sometimes, the documentation also includes a description and motivation of internal design decision, e.g., what algorithm for join ordering is implemented in the system. Some systems provide APIs and accompanying documentation for extending the supported data types or implementing user-defined functions or for embedding a system. Sadly, **no system** provides a documentation targeted at database researchers and developers, that would explain how new algorithms for solving a particular database problem – like join ordering, for example – can be implemented in the system [Duc; CMU; Pos; Mon]. From Bobby's point of view, it is unclear whether such APIs even exist and proper documentation is just lacking. In addition to these problems, the aforementioned systems generally do not ship with alternative implementations of the same system component, e.g., different algorithms for computing join orders. This is cumbersome for Bobby, as she cannot easily pick up and evaluate algorithms of related works.

The tragedy of Bobby Tables is a story many Ph.D. students and post-doctoral researchers are familiar with. For this reason, in this paper, we propose a new database system that is *designed for* researchers, scholars, and developers. We present a system that can serve as a universal framework for implementing and experimenting with new database technology and that can serve as a common test bed for experimental evaluations. What would such a system have to look like? How would one design such a system? This work is a mix of a system and a vision paper.

4.1.1 Outline

In Section 4.2, we present our design goals, contrast to prior work, and propose our approach with mu*t*able. Section 4.3 to Section 4.7 present some of mu*t*able's features, both conceptually and by example.

4.2 Database System Design

To help Bobby out of her misery, we want to design a database system that is mainly targeted at academic research. We work out the following design goals that we deem inevitable to foster efficient research and education.

4.2.1 Design Goals

Extensibility. The DBMS should be easily extensible to augment it with new algorithms. There should be as little obstacles as possible to get started developing with the system. Proper documentation and clean APIs will go a long way to fulfill this design goal.

Separation of Concerns. When implementing a new algorithm in the DBMS, one should not need to know about implementation details of the remainder of the system. The system should be split into individual components. Each component will fulfill a single purpose and is independent of other components. In particular, components shall appear to the outside as stateless and hence make it impossible to rely on internal state. This principle must guide the design process of the API.

Abstraction… To enable easy adoption by practitioners and researchers, abstractions are necessary to focus attention on details of importance to our community. With abstraction, we can form a common "language of symbols" we operate with. To achieve this, types, functions, classes, and methods should be named and designed in consistence with academic terminology and usage. Complex theoretical constructs need to be broken down into atoms and the system should be designed around these atoms.

…without regret. Traditionally, abstraction in software design comes at a cost. For example, abstraction through interfaces (or abstract classes) comes at the cost of dynamic dispatches, posing a considerable overhead for frequently called functions. Abstractions, in general, pose artificial boundaries that hinder a compiler from specializing and aggressively optimizing code. Enabling aggressive optimization by the compiler and avoiding overheads from dynamic dispatches will be absolutely necessary to achieve maximum performance.

These design goals are very broad and are fitting to any software system. In the following, we will elaborate in more depth how these goals apply to a database system and how we aim to achieve them.

4.2.2 Related Work

Certainly, each of the aforementioned design goals has already been studied by our community. We therefore briefly revisit prior work and emphasize potential shortcomings in their design.

Extensible DBMSs can be dissected into two groups. The first group contains "complete" DBMSs with support for extending the system by a user. Such systems may allow for introducing *abstract data types (ADT)* into the language by implementing them in a *domain-specific language (DSL)*. Other common extensibility features are custom storage techniques or data access methods. Systems belonging to this group are ADT-Ingres [SRG83], R^{2}D^2 [KW87], PostgreSQL [SR86; RS87], Starburst [Sch+86; Haa+89], and DuckDB [RM19].

The second group is formed of systems providing "DBMS building blocks" to ease the construction of specialized, purpose-built systems. Two ambassadors of this group, that have a strong academical background, are GENESIS [Bat+88] and EXODUS [CD87; Car+91]. GENESIS provides a "file" (storage) management system, named JUPITER, that is composed of several layers. JUPITER provides several implementations for each layer and can be arbitrarily composed by selecting one implementation for each layer. For example, to implement a new buffer management strategy in GENESIS, one must first implement JUPITER's buffer management layer and then configure JUPITER to use this implementation for buffer management. While GENESIS' extensibility evolves around storage management, EXODUS aims to provide a framework for building a custom DBMS. EXODUS ships with a generic storage manager with support for concurrent and recoverable operations on objects of any size, a library of type-agnostic access methods, a query optimizer *generator*, and tools for constructing query language front-ends. While EXODUS provides much flexibility, it does not provide a complete, off-the-shelf DBMS. Of all these extensible systems, none fulfills our goal of abstraction without regret. The authors of GENESIS not only acknowledge this fact, but even envision how to resolve this issue in the future:

> From the side of software development, a technology is needed to compose layers of software at compile time (not at run-time as we are now doing). Compile-time composition has the potential of eliminating unneeded generality [. . .] through code simplification.
>
> — Batory et al. [Bat+88]

Consequently, systems that compile queries naturally fall into this category: While allowing for composition of (physical) operators to form a query plan is an abstraction, compiling the final plan produces code free of (or with less) abstractions [Neu11; Klo+14; MMP17; KLN21; HD23a]. However, dbdb lists only a single database system of "Academic" or "Educational" type that performs code generation, namely NoisePage. It appears that NoisePage is the only open-source, relational database system capable of achieving abstraction without regret. However, this system appears not to be designed for extensibility or exchangeability of components.

4.2.3 Our Approach: The mu*t*able System

As we did not see a single DBMS sufficiently satisfying our aforementioned design goals, we decided to start building a new database system, named mu*t*able [/'mjuːtəbl/]. Our system aims to fulfill our design goals, as we elaborate next, and it is particularly fitted for academic research and education.

Extensibility. To achieve extensibility, mu*t*able is a system composed of independent components. We provide a visualization of this concept in Figure 4.1. This design is very similar to that of GENESIS' storage system JUPITER (Section 4.2.2). Each component in the system fulfills a single, logically isolated task, e.g., plan enumeration for query optimization. Different implementations of the same component can easily be interchanged to *mutate* system behavior. Therefore, the system can easily be extended by providing a new implementation of a component.

Separation of concerns. A separation of concerns is achieved through a paradigm named "The Value is the Boundary" that was proposed by Gary Bernhardt at SCNA 2012 [Ber12]. To us,

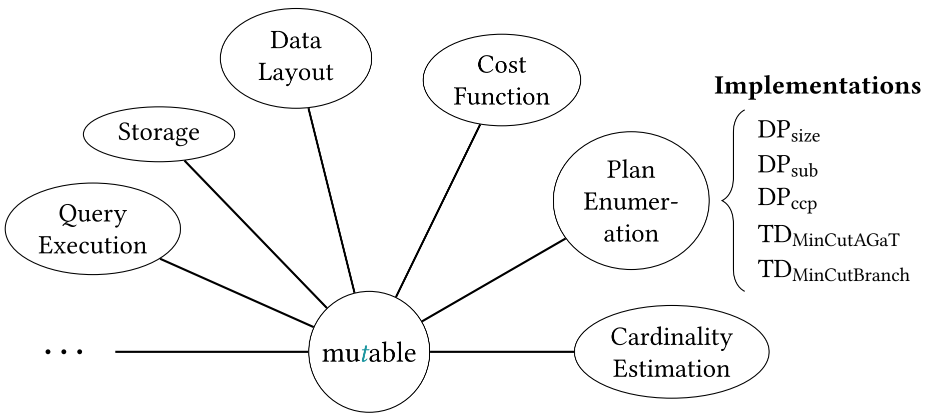


Figure 4.1: Components of mu*t*able with multiple, interchangable implementations of plan enumeration.

this paradigm means that the components appear to the outside world as stateless. They take values (potentially in the shape of data structures) as input, and they produce values (again, potentially data structures). Components shall not be dependent on internal state of other parts of the system. This design guarantees that a developer of one component need not be concerned with the implementation of any other component in the system. Dissecting the intrinsic logic into separate components and defining the "values" that need to be communicated in between is one of the crucial design processes involved in building mu*t*able. Naturally, we must represent state at some point. This is what Gary Bernhardt named the *imperative shell.* It is an imperative layer that connects the individual components, communicates the values between them, and holds the state of the system. Our work on mu*t*able aims to provide both, an implementation of this imperative shell and the development of components. In Section 4.2.4 we elaborate the concept of imperative shell, and in Section 4.3 and following we elaborate the design and development of components.

Abstraction. . . Abstraction is achieved by designing types, classes, methods, functions etc. in consistence with their academic usage and using the nomenclature common in our academic research.

. . . without regret. To achieve abstraction without regret, we envision a development process that heavily exploits metaprogramming to eliminate abstractions, as envisioned by Batory et al. [Bat+88]. For this purpose, be build on specialization through template-based metaprogramming and *just-in-time* (JIT) compilation. More precisely, we are developing a DSL that gives developers the impression of writing regular, imperative code. In the background, execution of that DSL code actually *produces code that is compiled and executed.* Compilation and execution of such generated code is handled implicitly by mu*t*able. The developer need not be concerned with this process. By compiling (and potentially optimizing) this code, mu*t*able is able to avoid abstraction overheads when executing the compiled code. Consider, for example, the development of a *multi-version concurrency control* (MVCC) algorithm, where each executing query must produce its read- and write-sets. In a traditional system, the MVCC component might have to register callbacks at the storage component to be informed of any read and write operations. Each such callback is an indirect function call, introducing unnecessary overhead to query execution. If this approach were implemented in mu*t*able, however, the MVCC component would register callbacks at the storage component that are implemented in our DSL. As a consequence, when the query is compiled, the code for data access is directly augmented by code to generate the read- and write-sets. Thereby, the indirect call that was originally necessary to achieve abstraction has been eliminated.

In the following sections, we present several features of mu*t*able in detail. We believe that these features make the system an appealing choice for researchers. We accompany our feature presentation with examples, such that the reader can observe how our implementation fulfills our design goals.

4.2.4 mu*t*able: The Imperative Shell

As we explained above, components are designed to provide no observable state to the outside. This guarantees that no part of the system may rely on implementation details exposed through internal state. Further, this level of encapsulation guarantees that components can safely be exchanged. However, a database system is an inherently stateful system. This state must be represented somewhere. Observe that, though the components themselves are stateless, they consume and produce values that represent state. These values must be communicated (or passed) between the components of the system, and they may be stored to persist state. This is exactly the task of the imperative shell. It connects the components, it controls the flow of data between them, and it represents state by storing values produced by components. Thereby, the imperative shell defines the interfaces of components, in terms of values consumed and produced.

Let us make this more concrete with an example. The tables of a database system represent state and are held within the imperative shell. However, the logic that operates on tables may be implemented in an execution backend component. Different backends may implement operations differently, yet they must all adhere to some common specification.

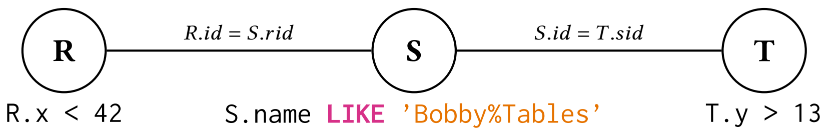


Figure 4.2: A query graph with three relations and two join predicates. Each relation has one selection predicate.

4.3 Components

Problem. In systems research – like our domain – it is necessary to conduct empirical studies to evaluate novel approaches. More so, we must also evaluate prior work to enable comparison and to contrast to our own work. In this process, it is important to perform the evaluations of both our and prior work under the exact same conditions. Only then can we truly compare our experimental findings and draw conclusions. However, this process is frequently disturbed. Consider, for example, the scenario that the original implementation of an algorithm is completely inaccessible. In this case, we must *reimplement* this algorithm to conduct our evaluation. Thereby, we may unknowingly improve or deteriorate the original algorithm. Nevertheless, reimplementation causes delay to our research. In an alternative scenario, the algorithm might be available, but as part of a complete system. In that case, we can fit our evaluation to the respective system. We must take great care to replicate the conditions under which the algorithm is evaluated. And yet, specifics to the system may unwittingly alter the experimental results.

Vision. Ideally, all algorithms of all related works are directly available for evaluation within a unifying system. Further, these algorithms all implement a common interface. This common interface guarantees that the conditions, under which evaluations are performed, are identical: all implementations share the same "view of the world". Because of the common interface, all implementations are completely interchangeable.

Approach. The mu*t*able system is composed of many components, as illustrated in Figure 4.1. For each component, we have devised an interface that describes precisely what information a component receives as input and what information a component produces as output. The process of designing component interfaces follows the principle "The Value is the Boundary", proposed by Bernhardt [Ber12]. As we shall see in the following, this design principle allows for clean separation of concerns, enables isolated testing of components, and further enables experimentally evaluating components in isolation of the remainder of the system.

Listing 4.1 Interface of the `PlanEnumerator` component.

```
struct PlanEnumerator {
  /** Enumerate feasible plans for query \p G.
   * \param G graph representation of the query
   * \param CE cardinality estimator component of the
   *           queried database
   * \param CF cost function to minimize
   * \param PT table of best plans found, with one
   *           entry per feasible partial plan
   */
  virtual void enumerate_plans(
    const QueryGraph &G,              // value (in)
    const CardinalityEstimator &CE, // component
    const CostFunction &CF,         // component
    PlanTable &PT                   // value (in & out)
  ) const = 0;
};
```

Table 4.1: Incoming `PlanTable` for the invocation of the `PlanEnumerator`. The `PlanTable` has been populated with entries for single relations.

Relations	**Cardinality**	**Cost**	**Plan**
$\{R\}$	50	0	R
$\{S\}$	20	0	S
$\{T\}$	35	0	T

Table 4.2: Final `PlanTable` after the invocation of the `PlanEnumerator`. Note that there is no entry for $\{R, T\}$ since our implementation in Listing 4.2 does not consider Cartesian products.

Relations	**Cardinality**	**Cost**	**Plan**
$\{R\}$	50	0	R
$\{S\}$	20	0	S
$\{T\}$	35	0	T
$\{R, S\}$	17	17	$R \bowtie S$
$\{S, T\}$	13	13	$S \bowtie T$
$\{R, S, T\}$	7	20	$R \bowtie (S \bowtie T)$

Example. To exemplify our approach, let us look at mu*t*able's interface for (logical) join order optimization. In mu*t*able, join ordering computes for a given query graph a partial order in which sets of relations are joined. Figure 4.2 shows an example of a query graph. To compute a join order for this query, mu*t*able invokes the `PlanEnumerator` component through the interface presented in Listing 4.1. The first argument to invocation is the query graph. The second argument is the `CardinalityEstimator` component. Its task is to estimate the cardinality of any given set of relations of the query graph, e.g., to estimate $\mathit{cardinality}(\{R, S\}) = 17$. The third argument is the cost function to minimize with optimization. The fourth argument is a `PlanTable`, a data structure similar to a *dynamic programming* (DP) table. Although the `PlanTable` can be used for computing a join order via dynamic programming, it can also be

Listing 4.2 DP$_{\text{ccp}}$ implementation of a `PlanEnumerator` component.

```
1 struct DPccp : PlanEnumerator {
2   void enumerate_plans(...) const override {
3     const AdjacencyMatrix &M = G.adjacency_matrix();
4     auto handle_CSG_pair = [&](Subproblem left, Subproblem right) {
5         PT.update(G, CE, CF, left, right); // update PlanTable for each CCP
6     };
7     M.for_each_CSG_pair_undirected(handle_CSG_pair);
8   }
9 };
```

populated with entries in any other fashion. The `PlanTable` is expected to be populated with entries for single relations, as exemplified in Table 4.1. The result of invoking the `PlanEnumerator` is a `PlanTable` populated with entries that form a valid logical plan. Table 4.2 shows the final `PlanTable` after join ordering.

We can observe how the `PlanEnumerator` component fulfills "The Value is the Boundary": To the outside world, a `PlanEnumerator` instance appears stateless. It consumes and produces values but it does not leak any state information, thereby preventing other components from relying on internal state. At the same time, the `PlanEnumerator` uses other components, namely `CardinalityEstimator` and `CostFunction`. These components appear stateless to the outside, too. To evaluate a `PlanEnumerator` – be it for testing or benchmarking – it suffices to (1) construct the `QueryGraph`, (2) provide `CardinalityEstimator` and `CostFunction` components, and (3) initialize a `PlanTable`. The ease with which we can isolate a `PlanEnumerator` from the remainder of the system makes testing, debugging, and benchmarking very accessible.

Now that we have seen the conceptual design of the interface, we will actually implement a `PlanEnumerator`. We will implement algorithm DP$_{\text{ccp}}$ by Moerkotte and Neumann [MN06]. To do so, let us go through the *actual* implementation of DP$_{\text{ccp}}$ in mu*t*able, given in Listing 4.2. In line 3, we get a handle on the graph's adjacency matrix. This data structure enables us to efficiently enumerate all pairs of *connected subgraphs* (CSGs) that are connected to one another. In line 4 and 5, we define a lambda, that takes a connected CSG pair as parameters `left` and `right`, and forwards it as a newly found plan to the `PlanTable`. Finally, in line 7, we let the adjacency matrix `M` enumerate all connected CSG pairs and provide the lambda of line 4 as callback. Eventually, when the `PlanEnumerator` returns after enumerating all plans, the `PlanTable` will contain an entry with the final plan, e.g., as in Table 4.2.

Our example demonstrates how concise mu*t*able's API is. With only 5 lines of code we are able to implement a state-of-the-art algorithm for join ordering. Of course, the complex graph traversal of DP$_{\text{CCP}}$ is realized by the adjacency matrix and remains completely hidden through the use of a callback function. Nonetheless, the code strongly expresses intent and almost appears to be a conceptual description of the algorithm. Also observe that our implementation does not rely on any implementation details of other components and fulfills only a single, isolated task: enumerating join orders. Our design lets a researcher easily exchange this particular implementation for another in our system.

With respect to abstraction overheads, we should note that `PlanTable` is not an abstract type. Further, `PlanTable::update()` is implemented in a header file and its implementation resides within the same translation unit as our DP_{CCP} implementation. The compiler will therefore inline the call to `PT.update()` and abstraction overheads are eliminated through compile-time composition. The same holds true for `for_each_CSG_pair_undirected()`.

4.4 Code Generation

Problem. In the previous section, we avoided abstraction overheads by relying on the compiler's ability to inline calls. However, this technique is not always applicable. Abstract types with virtual methods are sometimes necessary to achieve extensibility or composability. This is particularly true for query execution, where the query plan is a tree composed of abstract nodes and even within nodes we have abstractions, e.g., for expressions. To remedy abstraction overheads, queries can be compiled to machine code. However, we still see three problems impeding research in that direction: (1) Systems building on LLVM [LLV22], a rich compiler infrastructure, simply cannot achieve peak compilation speed as LLVM is not built for JIT compilation [Haa+17]. (2) Many compiling systems are not openly accessible, preventing extending, modifying, or even properly evaluating the query compilation process. (3) Systems that are openly accessible usually provide a low-level interface to code generation, that is similar to LLVM. Such an interface exacerbates adoption by DBMS researches that are not compiler experts [KLN21; FMT20].

Vision. Adoption of query compilation should not be any harder than directly implementing an algorithm in the programming language the DBMS is written in. The implementation should express the intent of the algorithm and must not be strictly coupled to the code generation process. Code generation should be designed with JIT compilation in mind. A suitable *intermediate representation* (IR) and compiler infrastructure should be provided.

Approach. We believe that a key technique to realizing our vision is metaprogramming. It allows us to pretend to the developers that they are writing regular code while code generation is performed in the background. This technique is becoming increasingly popular, e.g. LegoBase [KLN21], HyPer [Neu21], Umbra [KLN21], and Flounder IR [FMT20] provide DSLs for

Listing 4.3 Implementation of selection via conditional branching w/o short-circuit evaluation of the selection predicate.

```
 1  struct Sel : PhysicalOperator<
 2    Sel, // type for the curiously-recurring template pattern (CRTP)
 3    SelectionLOp // pattern of logical operator(s) to match
 4  > {
 5    static void execute(const Match<Sel> &M, CodeGenContext &Ctx, consumer_t consume)
 6    {
 7      /* Inject our code generation into that of our child. */
 8      M.child.execute([&M, &Ctx, consume=std::move(consume)](){
 9        /* Compile selection predicate w/o short-circuit evaluation. */
10        _Boolx1 pred = Ctx.compile(M.selection.predicate());
11        /* Conditional branching w/o short-circuit evaluation. */
12        IF (pred) {
13          /* Emit code for the remainder of the pipeline. */
14          consume();
15        };
16      });
17    }
18  };
```

code generation through metaprogramming. We have therefore developed a deeply-embedded DSL in C++ with a similar syntax to C++, that makes transitioning back and forth between DBMS code and generated query code seamless. We provide an implementation of the backend component with WebAssembly as IR and Google's V8 engine for JIT compilation and adaptive execution. In this work, we will only superficially describe our approach and focus on examples. Please refer to our separate work on JIT compiling SQL through WebAssembly to machine code with Google's V8, that is published at EDBT'23 [HD23a].

Example. In our example in Listing 4.3, we will implement code generation for selection w/o short-circuit evaluation of the selection predicate. For this section, it is sufficient to focus on lines 5 to 16. We will explain the remainder in the following Section 4.5. Line 5 declares the `execute()` method that "executes" the operator. In the context of code generation, this method actually generates the code for this operation. In an interpreting execution backend, this method would indeed execute the operator. This method's first parameter is the match, describing what part of the logical plan to execute. We elaborate this further in the following Section 4.5. The second parameter is the code generation context. It holds information necessary for code generation, e.g., an environment of named variables required to compile SQL predicates. The third parameter is a callable that "executes" the remainder of the pipeline. Note, that our model works slightly different from Neumann's produce/consume model, that was initially used in HyPer [Neu11]: rather than having `produce()` and `consume()` methods, we use a lambda to inject the consuming code into the child's code generation. Our approach is very similar to the approach proposed in the LB2 query compiler [TER18], that was later adopted by HyPer [Neu21]. In line 8, the handle M.child points to the physical operator implementing the logical child of the matched selection. On this child, we invoke execute() and pass as argument a lambda, that is defined in the following lines. In line 10, said lambda uses the CodeGenContext to compile the selection predicate to an *abstract syntax tree* (AST) in the underlying IR. Observe, that the return type is _Boolx1, which is defined by our DSL and represents a value at query runtime of boolean type. Additionally, the leading underscore _ expresses that the value may be **NULL** and the trailing x1 expresses that this value is *scalar*. (Our DSL by now supports code generation of SIMD code. To distinguish between scalar and vectorial runtime values, we use suffixes xN, where N is a suitable SIMD vector width.) In line 12, the lambda performs a conditional branch based on the compiled predicate. Note the peculiar syntax with an uppercase IF and the semicolon after the closing brace in line 15. This is our DSL, that mimics C++ in its syntax. Behind the scenes, the IF generates code with a conditional branch based on condition pred. DSL code in the *then*-block emits IR that is only executed when the predicate is satisfied. In line 14, the lambda invokes consume() to emit the consuming code of Sel's parent. DSL code executed by consume() emits code within the *then*-block. This means, code generated further up in the same pipeline will only be executed if the selection predicate is satisfied.

While there is some boilerplate code in Listing 4.3, the actual code generation happens in lines 10 to 15. Because of our DSL, that code is understandable by developers unfamiliar with code generation or compilation. Even more, we believe that with little practice developers will be able to benefit from compilation through metaprogramming with our DSL without necessarily having to understand the processes behind it.

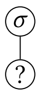
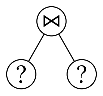
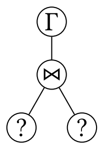
4.5 Physical Optimization

Problem. After implementing physical selection Sel in Section 4.4, we must inform the optimizer somehow that this is a suitable implementation for logical selection. While this step might appear trivial at first glance, there may be multiple physical implementations of the same logical operator, each fitted for a particular situation and hence with dependent cost. This fact calls for an optimization step that assigns physical implementations to logical operators. However, a one-to-one assignment of physical to logical operators is insufficient: Menon, Mowry, and Pavlo [MMP17] propose to *fuse* operators to produce more specialized implementations that can improve performance over naïve sequential application. This raises the question of how such fused operators can be considered in the optimization step.

Listing 4.4 Register `Sel` with the physical optimization process.

```
PhysicalOptimizer &PO = ...;
PO.register_operator<Sel>();
```

Table 4.3: Physical implementations of graph patterns.

Graph Pattern	Pattern C++ Code	Algorithm
	`Scan`	scan
	`SelectionLOp`	branching selection
		predicated selection
	`JoinLOp`	simple-hash join
		sort-merge join
	`pattern_t<GroupingLOp, JoinLOp>`	groupjoin

Vision. A DBMS researcher should be free to provide multiple physical implementations of any composition of logical operators. The optimization step that assigns physical implementations to the logical plan must consider all implementations and find the best of all possible physical plans.

Approach. In Section 4.3, we mentioned that the logical plan only induces a partial order and is represented as a tree. We implement a second optimization phase that enumerates physical implementations of the logical plan to find the best physical implementation. We treat physical operator implementations as partial graph covers and enumerate all possible coverings of the logical plan. This can be done in linear time [Bru07, p. 11, Section 2.5.2].

Example. After implementing selection in `Sel`, we must register `Sel` with the physical optimization process. In Listing 4.4, we invoke method `register_operator()` and pass as template argument the concrete type `Sel`. The method extracts from `Sel` the pattern to match, that was provided in line 3 of Listing 4.3. While the pattern to match a single `SelectionLOp` is trivial, our mechanism allows for more complex patterns to be declared. We provide some examples in Table 4.3. The helper class `pattern_t` allows for recursively composing patterns.

In addition to execution, physical operators can define custom *physical* cost functions as well as pre- and post-conditions to be considered in optimization. For example, it is possible to provide different cost functions for sort-merge join vs. simple-hash join and to formulate a post-condition for sort-merge join informing the optimizer that the join result is sorted on the join key.

4.6 Physical Data Layout Independence

Problem. The ability to decouple the physical data layout from the logical schema is a central building block of DBMSs. Some systems delegate this task to frameworks, like Apache Arrow [Len19], while others implement a particular physical data layout directly, e.g., NoisePage [22b]. Delegating this task to a framework introduces a framework's overheads into query processing. Directly implementing usually leads to hard-coding the data layout into the DBMS.

Vision. It should be possible to provide different strategies for mapping from a logical table schema to physical data layouts. A compiling backend should compile these mappings to direct data accesses, embedded in the compiled query, to avoid interpretation overheads.

Approach. In mu*t*able, we provide a concise method of describing the mapping from schema to data layout. Our method is generic enough to support arbitrary layouts of finite size. More precisely, our method allows for arbitrarily nested structures composed of various types, supporting even bit addressing and alignment. For example, the `BOOL` and `BITMAP` types need not be aligned to a whole multiple of a byte nor does their size need to be a whole multiple of a byte. A current limitation of our method is that we do not support variable-sized fields or pointers.

To efficiently access data through the description provided by a `DataLayout`, we translate `DataLayouts` in our interpreter and WebAssembly-based backends. The latter we present in Section 4.4.

Example. In Listing 4.5, we construct a `DataLayout` for a table with attributes `INT(4) PRIMARY KEY`, `CHAR(6)`, and `BOOL`. We lay out the data in PAX layout [MMP17] with PAX blocks of 128 tuples. The entire PAX layout is conceptually an indefinite sequence of PAX blocks. We first create an empty `DataLayout` in line 1. Then, we create a PAX block of 128 tuples and a stride of 12.288 bits in line 2. In lines 3 to 17, we add the attributes to the PAX block. To add an attribute to the PAX block, we specify the type of the attribute together with the offset of the attribute's column within the PAX block and the stride of a single attribute. In lines 18 to 22,

Listing 4.5 Implementation of a PAX layout.

```
DataLayout layout;
auto &block = layout.add_inode(/* num_tuples= */ 128, /* stride_in_bits= */ 12288);
block.add_leaf( // INT(4) PRIMARY KEY
    /* type=   */ Type::Get_Integer(Type::TY_Vector, 4),
    /* idx=    */ 0,
    /* offset= */ 0,
    /* stride= */ 32);
block.add_leaf( // CHAR(6)
    /* type=   */ Type::Get_Char(Type::TY_Vector, 6),
    /* idx=    */ 1,
    /* offset= */ 4096, // 128 * 32
    /* stride= */ 48); // 6 * 8
block.add_leaf( // BOOL
    /* type=   */ Type::Get_Boolean(Type::TY_Vector),
    /* idx=    */ 2,
    /* offset= */ 10240, // 4096 + 128 * 48
    /* stride= */ 1);
block.add_leaf( // NULL bitmap
    /* type=   */ Type::Get_Bitmap(Type::TY_Vector, 3),
    /* idx=    */ 3,
    /* offset= */ 10368, // 10240 + 128 * 1
    /* stride= */ 3);
```

we add the NULL bitmap to the PAX block. The NULL bitmap contains one bit per attribute, indicating whether the corresponding attribute is `NULL`.

In our WebAssembly-based execution backend, a scan of a table using the given `DataLayout` is compiled to a single loop iterating with four pointers – one per column and one for the NULL bitmap. On every 128-th iteration, the pointers are advanced to the next PAX block.

mu*t*able encapsulates the concept of computing `DataLayout`s for a table schema in a component. Such a component acts as a factory for creating `DataLayout`s. We have implemented one component for row layout and one for PAX layout. For PAX layout, one can specify either the number of tuples per block or the size in bytes of a single PAX block.

We see two limitations in our current implementation of this approach: (1) We do not support variable-length structures, e.g. arrays of variable length. (2) We do not support pointers to connect sequences of data, e.g. we cannot represent linked lists. Currently, all data must be finite and stored consequently in memory. Because of these limitations, we are currently only compatible with the full Apache Arrow [22a] specification. However, we are convinced that our approach can be extended by dynamically sized structures and pointers, and eventually it can support the full Arrow specification. The major obstacle we see here at the moment is the JIT code generation of data accesses from a `DataLayout` specification with dynamically sized structures or pointers.

4.7 Automated Evaluation

Problem. Evaluating DBMS algorithms or entire systems usually means running benchmarks. Writing benchmarks is therefore an inevitable task in our research. The results of benchmarks must be gathered, organized, and visualized to be easily interpretable. To enable comparison to related works, multiple algorithms or systems must be evaluated. Since evaluation is a process that is interleaved with research, it must be done repeatedly. Repeating evaluation by hand is tedious and automating evaluation for multiple algorithms or systems can be very cumbersome.

Vision. We envision a unifying evaluation framework, that researchers can easily implement experiments in and augment by new algorithms or systems to evaluate. The system should automate the process of repeated evaluation, gathering results, storing results persistently, and even visualizing results.

Approach. For this purpose, we have built a toolkit, that is composed of three tools: (1) The benchmarking tool runs a set of declaratively formulated experiments and collects results. The experiments are specified as YAML files and in such a way that we can run the same experiment on various database systems for comparison. The benchmarking tool can be set up to run repeatedly, e.g., daily or after each new commit to the `main` branch. Gathered experimental results are stored persistently in a database server. (2) A web server provides a REST API to read the gathered data from the database server. It provides both the original data and some pre-defined aggregated values. (3) An app that we developed for this purpose visualizes the results and monitors the benchmarking results over time. The app raises alerts when benchmarks were not run or when performance anomalies occurred. Our app is integrated with GitLab so that one can sign in to an administrative console through one's GitLab account. Once signed in, our app offers to directly create a GitLab issue from a raised alert. The issue is filled with a description of the alert as well as a breadcrumb link to directly go from GitLab issue to our app. Our app also tracks throughout the lifetime of an issue whether it has been resolved or rejected. Alerts can also be marked as expected, e.g. when performance improved expectedly because of an optimization in the code, or they can be marked as false positive, e.g. when the server running the benchmarks had unexpected load from other sources.

Example. The experiments are written in a descriptive YAML file, providing a textual description of the experiment, how the measurement data is to be interpreted in a chart, what data to load before the benchmark, and how to run the workload on each system. We provide an example for TPC-H query 1 in Listing 4.6. The specification of the systems is particular to

Listing 4.6 Sketch of the YAML file for TPC-H Q1.

```yaml
description: TPC-H Query 1 # Description.  Mandatory
suite: TPC                 # Mandatory
benchmark: TPC-H           # Mandatory
name: Q1                   # Optional, defaults to path
readonly: true             # Optional, defaults to false
chart: # Chart configuration.  Every field is optional
  x:
    scale: linear          # One of "linear", "log"
    type: Q                # One of Q, O, N, or T
    label: 'Scale␣factor'  # Axis label
  y:
    scale: linear          # One of "linear", "log"
    type: Q                # One of Q, O, N, or T
    label: 'Time␣[ms]'     # Axis label
data:                      # Data to load before benchmark
  - table:                 # Specification of a table
      name: 'Lineitem'
      attributes:
        l_orderkey: INT(4)
        l_partkey: INT(4)
        ...
    file: benchmark/tpc-h/data/lineitem.tbl
    format:                # Format of the file
      filetype: DSV
      delimiter: '|'
      header: false
systems:
  'mutable':
    ...                    # Spec. of experiment
  'PostgreSQL':
    ...                    # Spec. of experiment
  'HyPer':
    ...                    # Spec. of experiment
  'DuckDB':
    ...                    # Spec. of experiment
```

the respective system. We provide database connectors for several database systems, with the option to provide one's own connector. Queries must also be re-written per system because of potential SQL dialects or varying feature support.

Our benchmarking tool picks up the YAML file and runs the experiment, gathers the experimental results, and inserts all information related to the experiment to a relational database. A REST API written in Django provides easy access to the data.

Our browser app, written in Dart with Flutter, provides interactive visualization of the data. On the landing page – the "Dashboard" – we show an aggregated view of three hand-picked benchmark suites, namely "operators", "plan-enumerators", and TPC-H, as depicted in Figure 4.3. This is a heavily aggregated view of performance over the past days and intended to provide quick information on system behavior. Though the y-axis is value-less, it is linear and less is better.

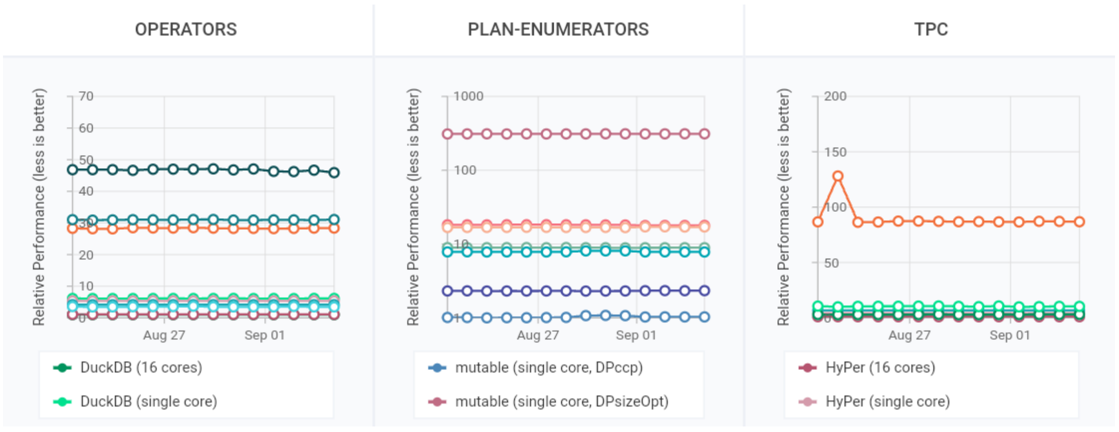


Figure 4.3: Aggregated performance statistics for selected benchmark suites, as visible on our dashboard. Provides an easily accessible overview over performance evolution.

The “Dashboard” provides a very brief overview over the performance. Our app provides a detailed visualization of single experiments in the “Recent Experiments” tab. Figure 4.4 shows the visualization of an experiment for one-sided range queries on integer columns.

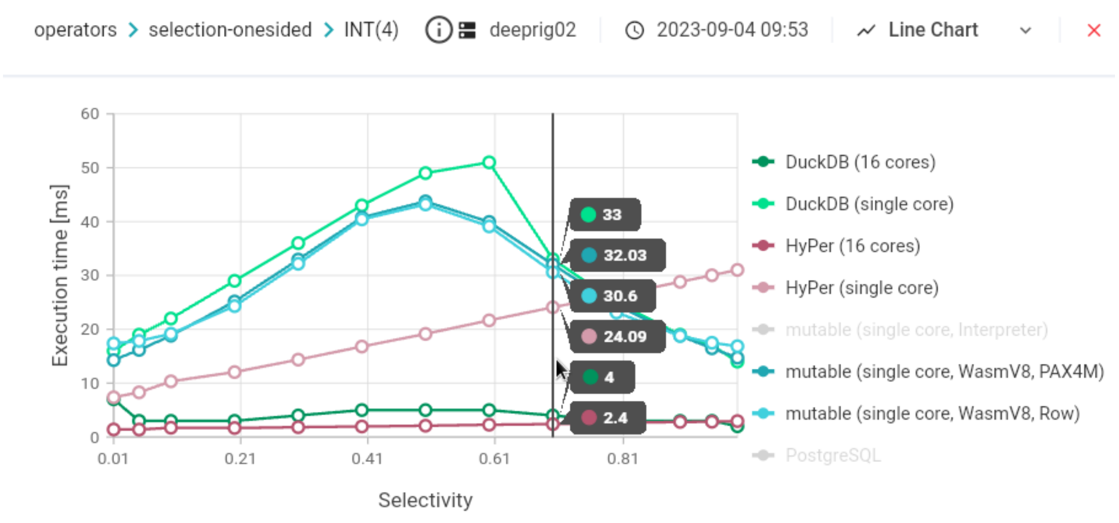
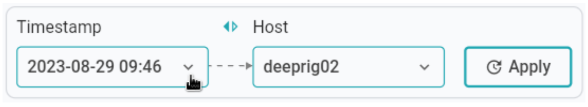


Figure 4.4: Interactive visualization of collected measurements for experiment operators ❯ selection-onesided ❯ INT(4).

The chart description from the YAML file is used to label the axis, select the scale of the axis, and automatically select the most appropriate style for visualization. We currently provide scatter, line, and bar charts displaying the median result of multiple runs as well as a scatter plot showing all measurements of all individual runs. We can hide single entries by toggling them in the chart legend. Hovering over the chart shows the precise measured values. Additional information is displayed above the chart: the location of the experiment, the machine the experiment was run on, and the exact time and date of the experiment.

We can manually select a different timestamp to inspect experimental results from another benchmark run. This can come in handy whenever we witness unexpected measurements in our experiments: we can go back in time to observe how exactly our measurements changed.



However, to understand how the performance of our system or that of other systems we compare to evolved over time, manually going back and forth through our timestamps is not very convenient. We therefore provide a visualization of experiment results over time in our “Continuous Benchmarking” tab. This tab provides aggregated performance statistics over time for every single experiment. Figure 4.5 shows the continuous benchmarking chart for the experiment in Figure 4.4.

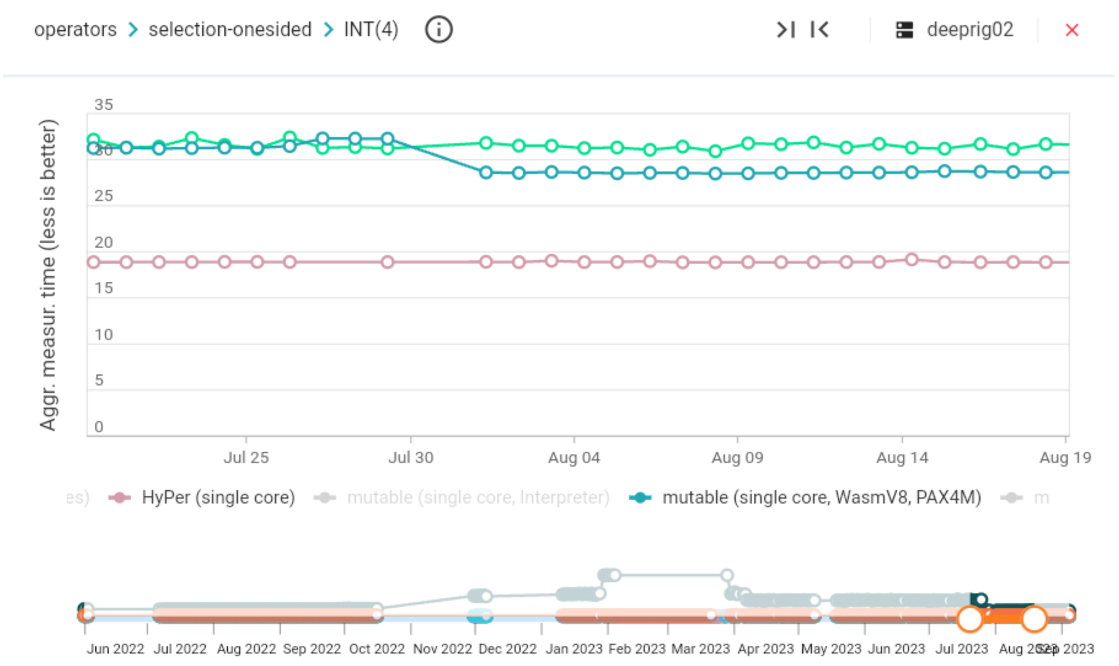
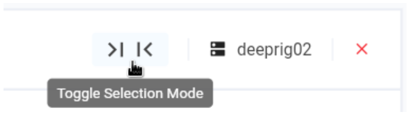


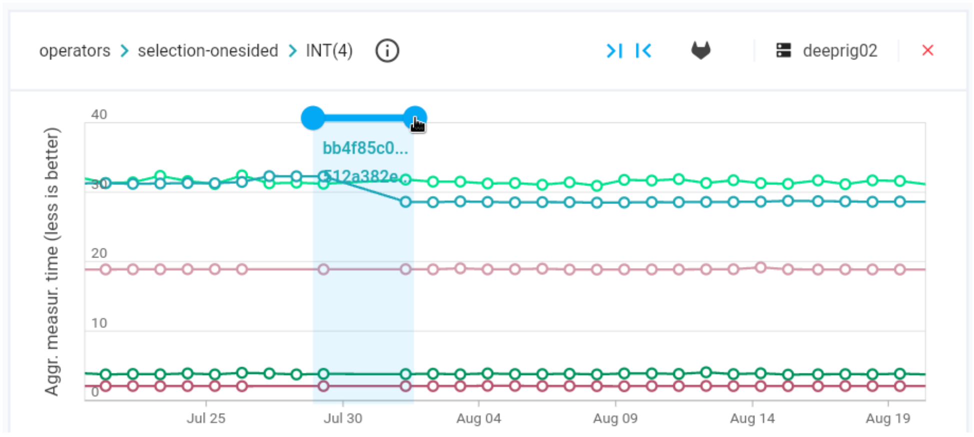
Figure 4.5: Visualization of experiment results over time.

In Figure 4.5, we can actually see two charts. The chart at the top shows the aggregated performance of the several systems over time. The smaller chart below allows us to select the exact range of time to visualize in the top chart. This feature lets us interactively zoom in on time ranges of particular interest. To aggregate the experimental results and associate a single numeric value to each timestamp and system, we decided to compute the Riemann sum of each experiment run per system and timestamp. The Riemann sum approximates the area under the curve. For example, the aggregated value for DuckDB (single core) at timestamp 2023-09-04 09:53, as depicted in Figure 4.4, would be the Riemann sum approximation of the area under the bright green curve. Since less execution time means better performance, consequently less area under the curve means better aggregated performance. Hence, in the "Continuous Benchmarking" charts, a smaller value on the y-axis is better.

Looking at Figure 4.5, we observe a performance improvement for mu*t*able around July 30. While performance improvements are appreciated, we may want to ensure that the witnessed performance improvement can be attributed to a performance-enhancing change in our code. Our continuous benchmarking visualization provides an easy facility for this purpose. First, we click on the "range selection" icon in the header of the experiment chart.



A blue range selection overlay appears in the chart. We drag the start and end of that overlay to precisely cover the time range in which we witness a performance anomaly.



After selecting the appropriate range, we click on the GitLab icon, that appeared in the header of the chart. This button will open a new browser tab showing the history of commits within the selected time range together with the actual changes to the code. We find this feature to be invaluable for diagnosing the cause of a drift in performance over time.

Manually detecting such performance anomalies would be very cumbersome, particularly with hundreds of experiments being performed every single day. We have therefore integrated into our DJANGO REST API a mechanism to automatically detect and report performance anomalies. Detected anomalies are reported on our "Dashboard". For the particular performance change shown in Figure 4.5, a performance anomaly was detected. The report shows in which experiment an anomaly was detected and the date of the anomaly. Further, we show a *likelihood* of this anomaly being a *true* anomaly rather than being noise. In fact, this likelihood is used in the first place to decide whether a new measurement is considered a *potential* anomaly.

To compute the likelihood of a measurement being an anomaly, we first compute the standard deviation and the mean of the measurements of the past 10 days before the current measurement. Then, we compute the delta of the computed mean and the current measurement and divide this delta by the standard deviation. The resulting factor is the likelihood of the current measurement being an anomaly. If this likelihood exceeds a hard-coded threshold, our system will consider this a potential anomaly and report it on the "Dashboard". The proposed method mainly aims to filter out noise. We chose the hard-coded threshold for when the likelihood is considered a potential anomaly empirically. Anyway, the threshold should be chosen such that it suppresses false positives, i.e. noise remains below said threshold, but never such that it suppresses true positives, i.e. the threshold being higher than the likelihood of an actual performance anomaly.

Our anomaly detection is designed as an interactive protocol with a developer validating each of the reported *potential* anomalies. A developer can then classify each anomaly as one of: • *false positive*, the anomaly is credited to noise in the measurements, or • *expected*, the anomaly was expected because of a prior change to the code, or • *confirmed*, the anomaly cannot be credited to noise and was not expected and hence needs further investigation. This process is depicted in Figure 4.6.

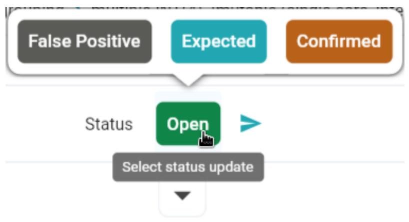


Figure 4.6: Updating the status of a performance anomaly report.

Our app supports updating the status of an anomaly report by first selecting a new status and then clicking on the blue right arrow. A very important feature here is that by confirming a performance anomaly, our app will create an issue in our GitLab project and fill the issue with all information available on the anomaly. Once the issue is tagged as being looked into or is closed in the GitLab project, our app recognizes this and presents the issue as either “Being Looked At” or “Closed”.

To see the full tool in action, visit our benchmark website at cb.mutable.uni-saarland.de.

Chapter 5

Conclusion

With my Ph.D. thesis, I made three major contributions in the domain of DBMS research.

My first contribution, presented in Chapter 2, approaches the prominent and important problem of join ordering from a new angle. I reformulate the join order optimization problem as a graph search problem, provide a strong theoretical foundation, and then take on an extensive effort to construct an efficient algorithm for solving this new optimization problem. Through several experimental evaluations, I was able to confirm that my novel approach improves over SOTA *asymptotically*, with speed-ups of 1000x or more for certain problem instances.

Future Work. To my belief, reformulating join ordering as a graph search problem creates a new angle to tackle the join order optimization problem. My work, presented in Chapter 2, forms the foundation for what may become a large body of follow-up work. Many choices that I made in Chapter 2 have alternatives that I was not able to explore due to a lack of time or space.

For example, different search algorithms may reveal new benefits of join ordering by graph search. As another example, bidirectional search may provide a solution that is naturally adaptive to the given query shape, preferring bottom-up search for star- or clique-shaped queries and preferring top-down search for chain- or cycle-shaped queries. I am currently supervising a Bachelor's thesis, where my student and I extend A^* in various ways. We added cost-based pruning, enabling more aggressive pruning of the search space while still maintaining optimality. While we only have preliminary results at this time, in some early experiments we witnessed a significant reduction in the number of explored vertices and hence a tremendous speed-up over search as presented in Chapter 2. We also extended A^* by anytime search, allowing for specifying a budget for the search. When that budget is exhausted, search terminates early, and potentially without finding a complete plan. We then *greedily* complete the partial solution, that is closest to the goal, to a complete plan. While this approach is not optimal anymore, it enables the application of heuristic search in time-constrained settings.

These early results already suggest that heuristic search still has potential that we have yet to explore.

My second contribution, presented in Chapter 3, proposes a simplified design of JIT compiling queries to machine code. By building on existing JIT engines, that are developed by a large community of compiler experts, well-funded by big tech companies, and widely applied in production, we are able to lift a tremendous burden off of DBMS developers' shoulders. Our results show that it is feasible to embed general-purpose JIT engines into a DBMS. Further, our approach achieves query compilation and query execution times that are on par with SOTA. Our particular choice for using WebAssembly– which does not support generic programming and does not provide a standard library – as intermediate representation therefore forces us to reimplement some library routines through metaprogramming and JIT code generation. However, we are able to show that the combination of WebAssembly and V8 enables such fast code generation and compilation to machine code, that generating all parts of a query – even library routines such as hashing or sorting – becomes worthwhile.

Future Work. I was supervising a Master's thesis, where my student and I extend my approach from Chapter 3 in various ways. We include the *relaxed operator fusion* proposed by Menon, Mowry, and Pavlo [MMP17] and generalize it to allow for repacking data into different formats at intermediate points of materialization. We further turn the placement of artificially introduced points of materialization, named "soft pipeline breakers", into an optimization problem. Additionally, we implement a physical plan optimization that exploits our ability to rapidly generate and compile code. Our physical optimization is able to generate special-purpose data structures or algorithms as they seem fit for a particular query. For example, if the query requires us to build a hash index on some intermediate result, we can exploit if the hash key is unique and generate specialized probing code that avoids a duplicate check.

I believe that rapid code generation and compilation is the key technology to query compilation through metaprogramming. Generating and composing code, that is tailor-made for a single particular query to achieve a maximum throughput, will be necessary to build a DBMS that competes for peak performance. The granule at which we compose this code can become more and more fine-granular, as we are able to properly phrase the composition as an optimization problem. Ultimately, query compilers may become special-purpose program synthesizers, that treat an SQL query as a program specification.

My third contribution, presented in Chapter 4, presents a novel DBMS that I built to support the research of my thesis, but that has become something more. My design of mu*t*able aims towards easy extension and modification. mu*t*able should support future DBMS researchers in quickly implementing and evaluating their research ideas. By licensing mu*t*able under AGPL, we guarantee that any evolution of that system must be made publicly available. I hope that this eventually contributes towards fair and reproducible evaluations and simplifies the process of conducting experiments. The extensibility of mu*t*able is such a central aspect of the system, that we are – at the time of finishing this thesis – building *multi-version concurrency control* as well as buffer management and persistent storage as plug-in features. Usually, MVCC and buffer management are both deeply embedded into a DBMS' design. With mu*t*able, we want to walk the extra mile and make every layer of a DBMS an exchangeable component. While such exchangeability and flexibility usually comes at the cost of performance, we make heavy use of the metaprogramming introduced in Chapter 3 to mitigate runtime overheads.

Bibliography

[22a] *Arrow - A cross-language development platform for in-memory analytics*. The Apache Software Foundation. 2022. URL: https://arrow.apache.org.

[22b] *NoisePage - Database Management System Project*. 2022. URL: https://noise.page.

[Ail+99] Anastassia Ailamaki et al. "DBMSs on a Modern Processor: Where Does Time Go?" In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Ed. by Malcolm P. Atkinson et al. Morgan Kaufmann, 1999, pp. 266–277. URL: http://www.vldb.org/conf/1999/P28.pdf.

[All+83] John R Allen et al. "Conversion of control dependence to data dependence." In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1983, pp. 177–189.

[ALX16] Sameer Agarwal, Davies Liu, and Reynold Xin. *Apache Spark as a compiler: Joining a billion rows per second on a laptop*. 2016.

[App] Apple Inc. *WebKit*. URL: webkit.org (visited on 04/30/2021).

[Bat+88] Don S. Batory et al. "GENESIS: An Extensible Database Management System." In: *IEEE Trans. Software Eng.* 14.11 (1988), pp. 1711–1730. DOI: 10.1109/32.9057. URL: https://doi.org/10.1109/32.9057.

[BC05] Brian Babcock and Surajit Chaudhuri. "Towards a robust query optimizer: a principled and practical approach." In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 2005, pp. 119–130.

[Ber12] G. Bernhardt. *Boundaries*. 2012. URL: https://www.destroyallsoftware.com/talks/boundaries.

[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, San Diego, CA, USA, July 1991.* Ed. by Richard K. Belew and Lashon B. Booker. Morgan Kaufmann, 1991, pp. 400–407.

[BFV96] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. "Dynamic load balancing in hierarchical parallel database systems." PhD thesis. INRIA, 1996.

[BMK09] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. "Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct." In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1648–1653. DOI: 10.14778/1687553.1687618. URL: http://www.vldb.org/pvldb/vol2/vldb09-10years.pdf.

[Boi+08] Benoit Boissinot et al. "Fast liveness checking for SSA-form programs." In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.* 2008, pp. 35–44.

[Bru07] R. T. E. Bruns. "Instruction selection on directed acyclic graphs." In: (2007).

[Byt] Bytecode Alliance. *Wasmtime.* URL: http://wasmtime.dev (visited on 09/06/2021).

[BZN05a] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings.* www.cidrdb.org, 2005, pp. 225–237. URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

[BZN05b] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings.* www.cidrdb.org, 2005, pp. 225–237. URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

[Car+91] Michael J. Carey et al. "The Architecture of the EXODUS Extensible DBMS." In: *On Object-Oriented Database Systems.* Ed. by Klaus R. Dittrich, Umeshwar Dayal, and Alejandro P. Buchmann. Topics in Information Systems. Springer, 1991, pp. 231–256.

[CB05] Thomas M Connolly and Carolyn E Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management.* Pearson Education, 2005.

[CD87] Michael J. Carey and David J. DeWitt. "An Overview of the EXODUS Project." In: *IEEE Data Eng. Bull.* 10.2 (1987), pp. 47–54. URL: http://sites.computer.org/debull/87JUN-CD.pdf.

[CGK20] Andrew Crotty, Alex Galakatos, and Tim Kraska. “Getting swole: Generating access-aware code with predicate pullups.” In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1273–1284.

[Cha+81] Donald D. Chamberlin et al. “A History and Evaluation of System R.” In: *Commun. ACM* 24.10 (1981), pp. 632–646. DOI: 10.1145/358769.358784. URL: https://doi.org/10.1145/358769.358784.

[Cha+95] Surajit Chaudhuri et al. “Optimizing queries with materialized views.” In: *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE. 1995, pp. 190–200.

[CM95] Sophie Cluet and Guido Moerkotte. “On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products.” In: *International Conference on Database Theory*. Springer. 1995, pp. 54–67.

[CMU] CMU Database Group. *NoisePage*. URL: https://github.com/cmu-db/noisepage/tree/master/docs (visited on 2022).

[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.

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

[Cor+16] Thomas H. Cormen et al. *Introduction to Algorithms*. The MIT Press, 2016.

[Cra+97] Timothy Cramer et al. “Compiling Java Just in Time.” In: *IEEE Micro* 17 (3 1997), pp. 36–43.

[Dia+13] Cristian Diaconu et al. “Hekaton: SQL server’s memory-optimized OLTP engine.” In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias. ACM, 2013, pp. 1243–1254. DOI: 10.1145/2463676.2463710. URL: https://doi.org/10.1145/2463676.2463710.

[Dij+59] Edsger W Dijkstra et al. “A note on two problems in connexion with graphs.” In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[DP85] Rina Dechter and Judea Pearl. “Generalized best-first search strategies and the optimality of A.” In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 505–536.

[DT07] David DeHaan and Frank Wm Tompa. "Optimal top-down join enumeration." In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, pp. 785–796.

[Duc] DuckDB Foundation. *DuckDB*. URL: https://duckdb.org/ (visited on 2022).

[ES11] Stefan Edelkamp and Stefan Schrodl. *Heuristic search: theory and applications*. Elsevier, 2011.

[Feg98] Leonidas Fegaras. "A new heuristic for optimizing large queries." In: *International Conference on Database and Expert Systems Applications*. Springer. 1998, pp. 726–735.

[Fen14] Pit Fender. "Algorithms for Efficient Top-Down Join Enumeration." In: (2014).

[FIL14] Craig Freedman, Erik Ismert, and Per-Åke Larson. "Compilation in the Microsoft SQL Server Hekaton Engine." In: *IEEE Data Eng. Bull.* 37.1 (2014), pp. 22–30. URL: http://sites.computer.org/debull/A14mar/p22.pdf.

[FM11a] Pit Fender and Guido Moerkotte. "A new, highly efficient, and easy to implement top-down join enumeration algorithm." In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 864–875.

[FM11b] Pit Fender and Guido Moerkotte. "Reassessing top-down join enumeration." In: *IEEE Transactions on Knowledge and Data Engineering* 24.10 (2011), pp. 1803–1818.

[FMT20] Henning Funke, Jan Mühlig, and Jens Teubner. "Efficient generation of machine code for query compilers." In: *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. Ed. by Danica Porobic and Thomas Neumann. ACM, 2020, 6:1–6:7. DOI: 10.1145/3399666.3399925. URL: https://doi.org/10.1145/3399666.3399925.

[Fre] Free Software Foundation, Inc. *LibJIT*. URL: https://www.gnu.org/software/libjit (visited on 09/06/2021).

[FWY08] Liying Fang, Pu Wang, and Jianzhuo Yan. "A multi-copy join optimization of information integration systems based on a genetic algorithm." In: *2008 The Third International Multi-Conference on Computing in the Global Information Technology (iccgi 2008)*. IEEE. 2008, pp. 223–228.

[GD87] Goetz Graefe and David J. DeWitt. "The EXODUS Optimizer Generator." In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Ed. by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pp. 160–172. DOI: 10.1145/38713.38734. URL: https://doi.org/10.1145/38713.38734.

[Gra53] Frank Gray. *Pulse code communication.* US Patent 2,632,058. Mar. 1953.

[Gra94] Goetz Graefe. “Volcano - An Extensible and Parallel Query Evaluation System.” In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135. DOI: 10.1109/69.273032. URL: https://doi.org/10.1109/69.273032.

[GZM21] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. “Babelfish: efficient execution of polyglot queries.” In: *Proceedings of the VLDB Endowment* 15.2 (2021), pp. 196–210.

[Haa+17] Andreas Haas et al. “Bringing the web up to speed with WebAssembly.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2017, pp. 185–200.

[Haa+89] Laura M. Haas et al. “Extensible Query Processing in Starburst.” In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989.* Ed. by James Clifford, Bruce G. Lindsay, and David Maier. ACM Press, 1989, pp. 377–388. DOI: 10.1145/67544.66962. URL: https://doi.org/10.1145/67544.66962.

[Haf+23] Immanuel Haffner et al. mu*t*able. 2023. URL: https://mutable.uni-saarland.de.

[Ham18] Clemens Hammacher. “Liftoff: a new baseline compiler for WebAssembly in V8.” In: *V8 JavaScript engine* (2018).

[HD23a] Immanuel Haffner and Jens Dittrich. “A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries.” In: *Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28 - March 31, 2023.* OpenProceedings.org, 2023.

[HD23b] Immanuel Haffner and Jens Dittrich. “Efficiently Computing Join Orders with Heuristic Search.” In: *Proc. ACM Manag. Data* (2023).

[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.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths.” In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[HR83] Theo Härder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery.” In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. "Architecture of a Database System." In: *Found. Trends Databases* 1.2 (2007), pp. 141–259. DOI: 10.1561/1900000002. URL: https://doi.org/10.1561/1900000002.

[IK84] Toshihide Ibaraki and Tiko Kameda. "On the optimal nesting order for computing n-relational joins." In: *ACM Transactions on Database Systems (TODS)* 9.3 (1984), pp. 482–502.

[Jan+19] Abhinav Jangda et al. "Not so fast: Analyzing the performance of WebAssembly vs. native code." In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 107–120.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. "Optimization of Nonrecursive Queries." In: *VLDB*. Vol. 86. 1986, pp. 128–137.

[KD98] Navin Kabra and David J DeWitt. "Efficient mid-query re-optimization of suboptimal query execution plans." In: *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 1998, pp. 106–117.

[Ker+18] Timo Kersten et al. "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask." In: *Proceedings of the VLDB Endowment* 11.13 (2018), pp. 2209–2222.

[KLN18] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive execution of compiled queries." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 197–208.

[KLN21] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra." In: *The VLDB Journal* (2021), pp. 1–23.

[Klo+14] Yannis Klonatos et al. "Building efficient query engines in a high-level language." In: *Proceedings of the VLDB Endowment* 7.10 (2014), pp. 853–864.

[Kot+08] Thomas Kotzmann et al. "Design of the Java HotSpot client compiler for Java 6." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5.1 (2008), pp. 1–32.

[KVC10] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation." In: *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. Ed. by Feifei Li et al. IEEE Computer Society, 2010, pp. 613–624. DOI: 10.1109/ICDE.2010.5447892. URL: https://doi.org/10.1109/ICDE.2010.5447892.

[KW87] Alfons Kemper and Mechtild Wallrath. "An Analysis of Geometric Modeling in Database Systems." In: *ACM Comput. Surv.* 19.1 (1987), pp. 47–91. DOI: 10.1145/28865.28866. URL: https://doi.org/10.1145/28865.28866.

[Lei+14] Viktor Leis et al. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 743–754.

[Lei+15] Viktor Leis et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[Len19] Geoffrey Lentner. "Shared Memory High Throughput Computing with Apache Arrow™." In: *PEARC*. 2019.

[LLV22] LLVM Team. *The LLVM Compiler Infrastructure*. 2022. URL: https://llvm.org.

[LP77] Michel Lacroix and Alain Pirotte. "Domain-Oriented Relational Languages." In: *Proceedings of the Third International Conference on Very Large Data Bases, October 6-8, 1977, Tokyo, Japan*. IEEE Computer Society, 1977, pp. 370–378.

[LZF13] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. "The Hekaton Memory-Optimized OLTP Engine." In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 34–40. URL: http://sites.computer.org/debull/A13june/Hekaton1.pdf.

[Mak] Vladimir Makarov. *MIR*. URL: https://github.com/vnmakarov/mir (visited on 09/06/2021).

[Mar+19] Ryan Marcus et al. "Neo: A Learned Query Optimizer." In: *Proceedings of the VLDB Endowment* 12.11 (2019).

[Mar+21] Ryan Marcus et al. "Bao: Making learned query optimization practical." In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 1275–1288.

[MDN] MDN Web Docs. *WebAssembly Developer Reference*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly (visited on 04/30/2021).

[MMP17] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. "Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last." In: *Proceedings of the VLDB Endowment* 11.1 (2017), pp. 1–13.

[MN06] Guido Moerkotte and Thomas Neumann. "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products." In: *Proceedings of the 32nd international conference on Very large data bases*. Citeseer. 2006, pp. 930–941.

[Mon] MonetDB B.V. *MonetDB*. URL: https://www.monetdb.org/documentation-Jan2022/dev-guide/ (visited on 2022).

[Moz] Mozilla Developer Network. *SpiderMonkey*. URL: developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey (visited on 04/30/2021).

[Neg+21] Parimarjan Negi et al. “Steering query optimizers: A practical take on big data workloads.” In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 2557–2569.

[Neu09] Thomas Neumann. “Query simplification: graceful degradation for join-order optimization.” In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 2009, pp. 403–414.

[Neu11] Thomas Neumann. “Efficiently compiling efficient query plans for modern hardware.” In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 539–550.

[Neu21] Thomas Neumann. “Evolution of a Compiling Query Engine.” In: *Proc. VLDB Endow.* 14.12 (2021), pp. 3207–3210. DOI: 10.14778/3476311.3476410. URL: http://www.vldb.org/pvldb/vol14/p3207-neumann.pdf.

[Ng+99] Kenneth W Ng et al. “Dynamic query re-optimization.” In: *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*. IEEE. 1999, pp. 264–273.

[Nie+20] Tobias NieSSen et al. “Insights into WebAssembly: compilation performance and shared code caching in Node.js.” In: *CASCON ’20: Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering, Toronto, Ontario, Canada, November 10 - 13, 2020*. Ed. by Lily Shaddick et al. ACM, 2020, pp. 163–172. DOI: 10.5555/3432601.3432623. URL: https://dl.acm.org/doi/10.5555/3432601.3432623.

[NK15] Thomas Neumann and Alfons Kemper. “Unnesting arbitrary queries.” In: *Datenbanksysteme für Business, Technologie und Web (BTW)* (2015).

[NR18] Thomas Neumann and Bernhard Radke. “Adaptive optimization of very large join queries.” In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 677–692.

[OED] OED Online, ed. *Oxford English Dictionary*. Oxford University Press. (Visited on 07/12/2013).

[OL90] Kiyoshi Ono and Guy M. Lohman. “Measuring the Complexity of Join Enumeration in Query Optimization.” In: *VLDB*. Vol. 97. 1990, pp. 314–325.

[Ora12] Oracle. *OpenJDK: Graal project.* 2012. URL: openjdk.java.net/projects/graal/ (visited on 04/30/2021).

[Pad+01] Sriram Padmanabhan et al. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures." 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. 567–574. DOI: 10.1109/ICDE.2001.914871. URL: https://doi.org/10.1109/ICDE.2001.914871.

[Pav+17] Andrew Pavlo et al. "Self-Driving Database Management Systems." In: *CIDR.* Vol. 4. 2017, p. 1.

[Per+19] Matthew Perron et al. "How I learned to stop worrying and love re-optimization." In: *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE. 2019, pp. 1758–1761.

[Pir+16] Holger Pirk et al. "Voodoo – A Vector Algebra for Portable Database Performance on Modern Hardware." In: *Proceedings of the VLDB Endowment* 9.14 (2016), pp. 1707–1718.

[Pos] PostgreSQL. *PostgreSQL.* URL: https://www.postgresql.org/docs/14/index.html (visited on 2022).

[PWZ96] Marko Petkovek, Herbert S Wilf, and Doron Zeilberger. *A = B.* A K Peters. Ltd., 1996.

[Rao+06] Jun Rao et al. "Compiled Query Execution Engine using JVM." In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA.* Ed. by Ling Liu et al. IEEE Computer Society, 2006, p. 23. DOI: 10.1109/ICDE.2006.40. URL: https://doi.org/10.1109/ICDE.2006.40.

[RM19] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* Ed. by Peter A. Boncz et al. ACM, 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212. URL: https://doi.org/10.1145/3299869.3320212.

[RN20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 4th ed. Prentice Hall, 2020.

[Ros02] Kenneth A Ross. "Conjunctive selection conditions in main memory." In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* 2002, pp. 109–120.

[RS87] Lawrence A. Rowe and Michael Stonebraker. "The POSTGRES Data Model." In: *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*. Ed. by Peter M. Stocker, William Kent, and Peter Hammersley. Morgan Kaufmann, 1987, pp. 83–96. URL: http://www.vldb.org/conf/1987/P083.PDF.

[Sal17] Saïd Salhi. *Heuristic search: The emerging science of problem solving*. Springer, 2017.

[SBB21] Filippo Schiavio, Daniele Bonetta, and Walter Binder. "Language-agnostic integrated queries in a managed polyglot runtime." In: *Proceedings of the VLDB Endowment* 14.8 (2021), pp. 1414–1426.

[Sch+86] Peter M. Schwarz et al. "Extensibility in the Starburst Database System." In: *1986 International Workshop on Object-Oriented Database Systems, September 23-26, 1986, Asilomar Conference Center, Pacific Grove, California, USA, Proceedings*. Ed. by Klaus R. Dittrich and Umeshwar Dayal. IEEE Computer Society, 1986, pp. 85–92. URL: http://dl.acm.org/citation.cfm?id=318842.

[Sch04] Alexander Schrijver. "Combinatorial optimization: Polyhedra and efficiency (algorithms and combinatorics)." In: *Journal-Operational Research Society* 55.9 (2004), pp. 1018–1018.

[SDS16] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. "RUMA has it: rewired user-space memory access is possible!" In: *Proceedings of the VLDB Endowment* 9.10 (2016), pp. 768–779.

[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*. Ed. by Philip A. Bernstein. ACM, 1979, pp. 23–34. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

[Sel88] Timos K Sellis. "Multiple-query optimization." In: *ACM Transactions on Database Systems (TODS)* 13.1 (1988), pp. 23–52.

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. "Heuristic and randomized optimization for the join ordering problem." In: *The VLDB Journal* 6.3 (1997), pp. 191–208.

[SR86] Michael Stonebraker and Lawrence A. Rowe. "The Design of Postgres." In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*. Ed. by Carlo Zaniolo. ACM Press, 1986, pp. 340–355. DOI: 10.1145/16894.16888. URL: https://doi.org/10.1145/16894.16888.

[SRG83] Michael Stonebraker, W. Bradley Rubenstein, and Antonin Guttman. “Application of Abstract Data Types and Abstract Indices to CAD Data Bases.” In: *Engineering Design Applications, Database Week, May 1983.* IEEE Computer Society, 1983, pp. 107–113.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. “Vectorization vs. compilation in query execution.” In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware.* ACM. 2011, pp. 33–40.

[TER18] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. “How to Architect a Query Compiler, Revisited.” In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 307–322. DOI: 10.1145/3183713.3196893. URL: https://doi.org/10.1145/3183713.3196893.

[V8 08] V8 Project Authors. *V8: Googles open source high-performance JavaScript and WebAssembly engine.* 2008. URL: https://v8.dev/.

[Van98] Bennet Vance. *Join-order optimization with Cartesian products.* Oregon Graduate Institute of Science and Technology, 1998.

[Vel08] Stoyan Vellev. “An adaptive genetic algorithm with dynamic population size for optimizing join queries.” In: (2008).

[VM96] Bennet Vance and David Maier. “Rapid bushy join-order optimization with cartesian products.” In: *ACM SIGMOD Record* 25.2 (1996), pp. 35–46.

[W3C] W3C WebAssembly Community Group. *WebAssembly.* URL: www.webassembly.org (visited on 04/30/2021).

[Was] Wasmer, Inc. *Wasmer.* URL: https://wasmer.io (visited on 09/06/2021).

[WNS16] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. “Sampling-based query re-optimization.” In: *Proceedings of the 2016 International Conference on Management of Data.* 2016, pp. 1721–1736.

[WP00] Florian Waas and Arjan Pellenkoft. “Join order selection (good enough is easy).” In: *British National Conference on Databases.* Springer. 2000, pp. 51–67.

Glossary

Abstract Data Type, ADT Formally, a class of objects whose logical behavior is defined by a set of values and a set of operations. 100

ACID ACID (atomicity, consistency, isolation, durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps [HR83]. 89

CNF Conjunctive normal form. 14

CODASYL Conference/Committee on Data Systems Languages 12, 137

Connected complement pair, CCP A pair of disjoint *connected subgraphs (CSG)* that are connected to each other by at least one edge. 32, 39

Connected Subgraph, CSG A subgraph of an undirected graph that is connected, i.e. each two vertices of the subgraph are connected by a sequence of edges of the subgraph. 137

Data Base Task Group, DBTG The working group founded within CODASYL to devise a standard for communicating with DBMSs. 12

Database Management System, DBMS A software system that enables users to define, create, maintain, and control access to the database [CB05]. 11, 12, 23–28, 31, 49, 62, 99–101, 107, 108, 110, 111, 113, 121–123, 137, 138

Database, DB A structured set of data held in a computer, especially one that is accessible in various ways. 11, 12, 137

Domain-specific language, DSL A computer language specialized to a particular application domain. 100

Just-in-time, JIT 24–26, 66, 122

multi-version concurrency control, MVCC A concurrency control scheme for DBMSs. 123

Ngram, Google Ngram The Google Books Ngram Viewer (Google Ngram) is a search engine that charts word frequencies from a large corpus of books and thereby allows for the examination of cultural change as it is reflected in books. The corpus is a sample of books written in English and published in the United States. 12

Normal form "A normal form is a way of representing objects such that although an object may have many names, every possible name corresponds to exactly one object." [PWZ96, p. 7] 14, 137

SOTA State of the Art. 11, 22, 62, 121, 122

Write-ahead logging, WAL A family of techniques for providing atomicity and durability in DBMSs [HSH07]. 89