

mutable: A Modern DBMS for Research and Fast Prototyping

Immanuel Haffner

Saarland University, Saarland Informatics Campus
Germany
immanuel.haffner@bigdata.uni-saarland.de

ABSTRACT

Few to zero DBMSs provide extensibility together with implementations of modern concepts, like query compilation for example. We see this as an impeding factor in academic research in our domain. Therefore, in this work, we present *mutable*, a system developed at our group, that is fitted to academic research and education. *mutable* features a modular design, where individual components can be composed to form a complete system. Each component can be replaced by an alternative implementation, thereby mutating the system. Our fine-granular design of components allows for precise mutation of the system. Metaprogramming and just-in-time compilation are used to remedy abstraction overheads. In this demo, we present the high-level design goals of *mutable*, discuss our vision of a modular design, present some of the components, provide an outlook to research we conducted within *mutable*, and demonstrate some developer-facing features.

1 INTRODUCTION

Bobby Tables is a young Ph.D. student in the field of database systems, who just recently started doing his own research. Bobby has an idea how to compute join orders more efficiently. At some point, Bobby has to implement his idea to perform an empirical evaluation. Bobby now has to make an important decision: He can (1) implement his idea in an existing (open-source) system or (2) implement his idea stand-alone, i.e. not integrating it into a system. Bobby asks for guidance from his Ph.D. advisor, who tells him 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 his approach in an existing system. He will be able to rely on a rich infrastructure taking care of many things unrelated to his 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 his 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

Jens Dittrich

Saarland University, Saarland Informatics Campus
Germany
jens.dittrich@bigdata.uni-saarland.de

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 his join ordering algorithm in an existing system. But which system should he 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 he will find a suitable system for his implementation among these many candidates. He uses the filter to refine his 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 he spots some famous research projects within the list: PostgreSQL, Hypér, MonetDB, DuckDB, and NoisePage, to name a few. Bobby takes a closer look at the open-source systems to understand how he can implement and integrate his join ordering algorithm. He 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 [3, 9, 20, 23]. 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 he 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.



github.com/mutable-org/mutable



1.1 Outline

In Section 2, we present our design goals, contrast to prior work, and propose our approach with *mutable*. Section 3 to Section 7 present some of *mutable*'s features, both conceptually and by example.

2 DATABASE SYSTEM DESIGN

To help Bobby out of his 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.

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.

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.

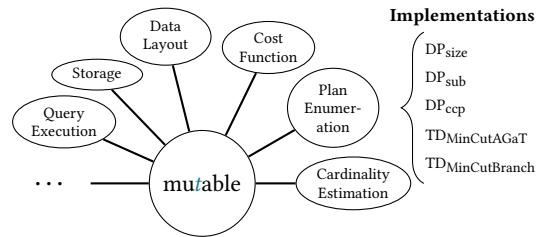


Figure 1: Components of *mutable* with multiple, interchangeable implementations of plan enumeration.

Systems belonging to this group are ADT-INGRES [28], R²D² [14], POSTGRESQL [25, 27], Starburst [12, 26], and DUCKDB [24].

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 academic background, are GENESIS [5] and EXODUS [8]. 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. [5]

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 [13, 15, 16, 18, 21]. 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.

2.3 Our Approach: The *mutable* 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 *mutable* [/mju:təbl/]. 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, mutable is a system composed of independent components. We provide a visualization of this concept in Figure 1. This design is very similar to that of GENESIS’ storage system JUPITER (Section 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 [6]. To us, 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 mutable. 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 mutable aims to provide both, an implementation of this imperative shell and the development of components. In Section 2.4 we elaborate the concept of imperative shell, and in Section 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. [5]. 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 mutable. The developer need not be concerned with this process. By compiling (and potentially optimizing) this code, mutable 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 mutable, 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 mutable 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.

2.4 mutable: 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 / 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.

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

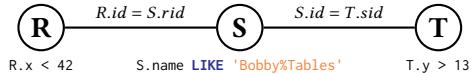


Figure 2: A query graph with three relations and two join predicates. Each relation has one selection predicate.

Listing 1 Interface of the PlanEnumerator component.

```

1 struct PlanEnumerator {
2     /** Enumerate feasible plans for query \p G.
3      * \param G graph representation of the query
4      * \param CE cardinality estimator component of the
5      *       queried database
6      * \param CF cost function to minimize
7      * \param PT table of best plans found, with one
8          entry per feasible partial plan
9     */
10    virtual void enumerate_plans(
11        const QueryGraph &G,           // value (in)
12        const CardinalityEstimator &CE, // component
13        const CostFunction &CF,        // component
14        PlanTable &PT,              // value (in & out)
15    ) const = 0;
16 };
  
```

Table 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 2: Final PlanTable after the invocation of the PlanEnumerator. Note that there is no entry for {R, T} since our implementation in Listing 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 ⚡ S
{S, T}	13	13	S ⚡ T
{R, S, T}	7	20	R ⚡ (S ⚡ T)

the world”. Because of the common interface, all implementations are completely interchangeable.

Our approach. The mutable system is composed of many components, as illustrated in Figure 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 [6]. 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.

Example. To exemplify our approach, let us look at mutable’s interface for (logical) join order optimization. In mutable, join ordering computes for a given query graph a partial order in which sets of relations are joined. Figure 2 shows an example of a query graph. To compute a join order for this query, mutable invokes the PlanEnumerator component through the interface presented in Listing 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 $\text{cardinality}(\{R, S\}) = 17$. 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

Listing 2 DP_{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,
5             Subproblem right)
6         { PT.update(G, CE, CF, left, right); };
7         M.for_each_CSG_pair_undirected(handle_CSG_pair);
8     }
9 };
  
```

for computing a join order via dynamic programming, it can also be populated with entries in any other fashion. The PlanTable is expected to be populated with entries for single relations, as exemplified in Table 1. The result of invoking the PlanEnumerator is a PlanTable populated with entries that form a valid logical plan. Table 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_{ccp} by Moerkotte and Neumann [19]. To do so, let us go through the *actual* implementation of DP_{ccp} in mutable, given in Listing 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 2.

Our example demonstrates how concise mutable’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_{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 plans. This design lets a researcher easily exchange this particular implementation for another in the 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 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 [2], a rich compiler infrastructure, simply cannot achieve peak compilation speed as LLVM is not built for JIT compilation [11]. (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 researchers that are not compiler experts [10, 15].

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.

Our 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 [15], Hyper [22], Umbra [15], and Flounder IR [10] provide DSLs for 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 [13].

Example. In our example in Listing 3, we will implement code generation for selection with 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 5. Lines 5 to 7 declare 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 5.

Listing 3 Implementation of selection via conditional branching and short-circuit evaluation of the selection predicate.

```

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

```

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 [21]: 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 [29], that was later adopted by HyPER [22]. In line 9, 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 11, the lambda uses the CodeGenContext to compile the selection predicate to an *abstract syntax tree* (AST) in the underlying IR. In line 13, the lambda performs a conditional branch based on the compiled predicate. Note the particular uppercase IF and the semicolon after the closing brace in line 16. This is our DSL, that mimics C++ in its syntax. Behind the scenes, the IF generates code with a conditional branch and performs short-circuit evaluation of pred. DSL code in the *then*-block emits IR that is only executed when the predicate is satisfied. In line 15, 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 3, the actual code generation happens in lines 11 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.

5 PHYSICAL OPTIMIZATION

Problem. After implementing physical selection Sel in Section 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

Listing 4 Register Sel with the physical optimization process.

```

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

```

Table 3: Physical implementations of logical patterns.

Pattern	C++ Code	Algorithm
	SelectionLop	branching selection
		predicated selection
	JoinLop	simple-hash join
		sort-merge join
	pattern_t<GroupingLop, JoinLop>	groupjoin

that assigns physical implementations to logical operators. However, a one-to-one assignment of physical to logical operators is insufficient: Menon et al. [18] 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.

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.

Our approach. In Section 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 [7, p. 11, Section 2.5.2].

Example. After implementing selection in Sel, we must register Sel with the physical optimization process. In Listing 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 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 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.

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 [17], while others implement a particular physical data layout directly, e.g., NOISEPAGE [4]. 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.

Listing 5 Implementation of a PAX layout.

```

1 DataLayout layout;
2 auto &block = layout.add_inode(/* num_tuples= */ 128,
3                               /* stride_in_bits= */ 12288);
4 block.add_leaf( /* INT(4) PRIMARY KEY
5   /* type= */ Type::Get_Integer(Type::TY_Vector, 4),
6   /* idx= */ 0,
7   /* offset= */ 0,
8   /* stride= */ 32);
9 block.add_leaf( /* CHAR(6)
10  /* type= */ Type::Get_Char(Type::TY_Vector, 6),
11  /* idx= */ 1,
12  /* offset= */ 4096, /* 128 * 32
13  /* stride= */ 48); /* 6 * 8
14 block.add_leaf( /* BOOL
15  /* type= */ Type::Get_Boolean(Type::TY_Vector),
16  /* idx= */ 2,
17  /* offset= */ 10240, /* 4096 + 128 * 48
18  /* stride= */ 1);
19 block.add_leaf( /* NULL bitmap
20  /* type= */ Type::Get_Bitmap(Type::TY_Vector, 3),
21  /* idx= */ 3,
22  /* offset= */ 10368, /* 10240 + 128 * 1
23  /* stride= */ 3);

```

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.

Our approach. In `mutable`, 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.

Example. In Listing 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 [18] 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 lines 2 and 3. In lines 4 to 18, 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 19 to 23, 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.

`mutable` encapsulates the concept of computing `DataLayouts` for a table schema in a component. Such a component acts as a factory

for creating DataLayouts. 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 a subset of the Apache Arrow [1] 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.

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.

Our 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.

Listing 6 Sketch of the YAML file for TPC-H Q1.

```

1  description: TPC-H Query 1 # Description. Mandatory
2  suite: TPC                  # Mandatory
3  benchmark: TPC-H            # Mandatory
4  name: Q1                   # Optional, defaults to path
5  readonly: true              # Optional, defaults to false
6  chart: # Chart configuration. Every field is optional
7    x:
8      scale: linear           # One of "linear", "log"
9      type: Q                 # One of Q, O, N, or T
10     label: 'Scale_factor'   # Axis label
11    y:
12      scale: linear           # One of "linear", "log"
13      type: Q                 # One of Q, O, N, or T
14     label: 'Time_ms'          # Axis label
15   data:
16     - table:
17       name: Lineitem
18       attributes:
19         l_orderkey: INT(4)
20         l_partkey: INT(4)
21     ...
22   file: benchmark/tpc-h/data/lineitem.tbl
23   format:                      # Format of the file
24     filetype: DSV
25     delimiter: '|'
26     header: false
27   systems:
28     'mutable':                # Spec. of experiment
29     ...
30     'PostgreSQL':             # Spec. of experiment
31     ...
32     'Hyper':                  # Spec. of experiment
33     ...
34     'DuckDB':                 # Spec. of experiment
35     ...

```

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 6. The specification of the systems is particular to 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 shown in Figure 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.

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 shows the visualization of an experiment for one-sided range queries on integer columns. 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

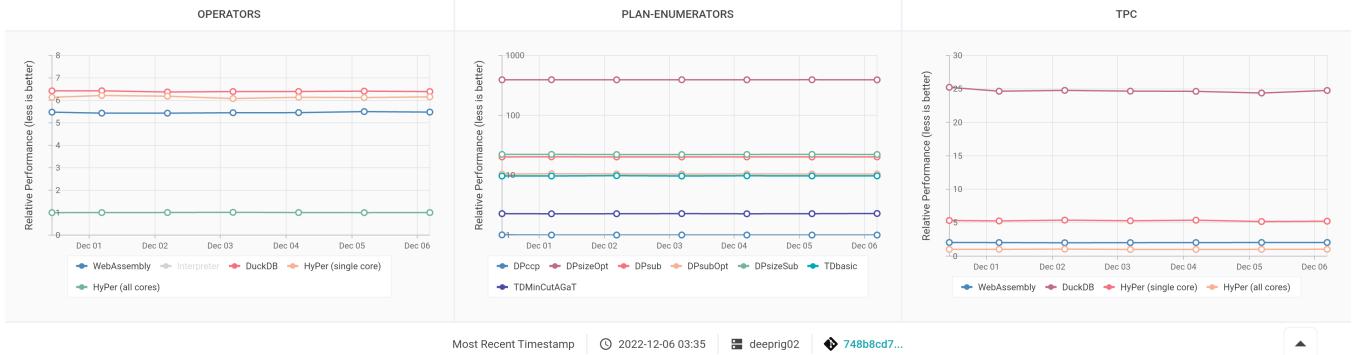


Figure 3: Aggregated performance statistics for selected benchmark suites, as visible on our dashboard.

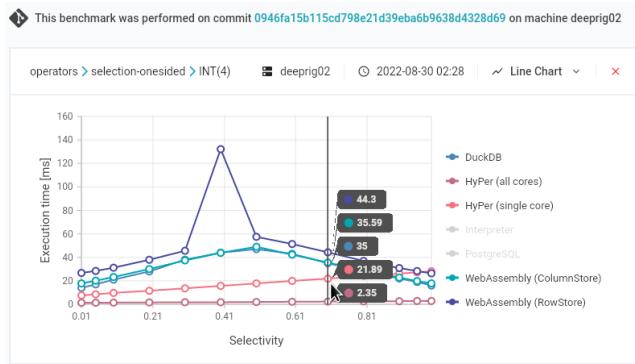


Figure 4: Visualization of recent benchmark results.

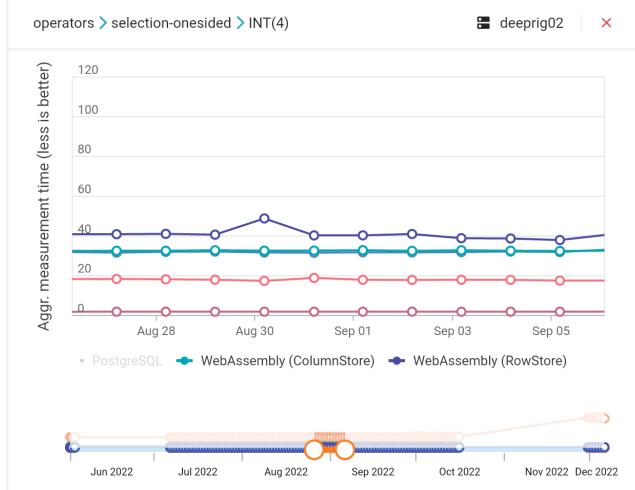


Figure 5: Visualization of benchmark results over time.

scatter, line, and bar charts. We can hide single entries by toggling them in the chart legend. Hovering over the chart shows the precise measurements.

We can see in Figure 4 that there appears to be an outlier for DuckDB at around 40% selectivity. We have two options to determine whether this is an outlier or reproducible behavior. In the “Recent Experiments” tab, we can select the date of the experiment.

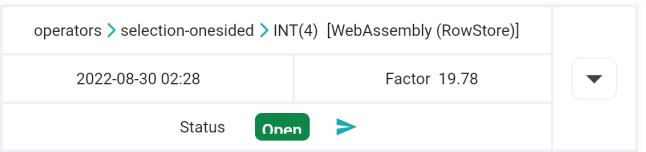


Figure 6: A detected performance anomaly.

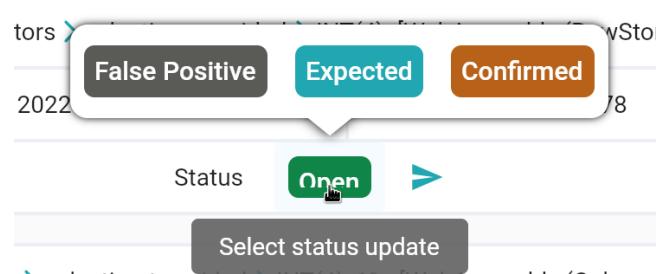


Figure 7: Updating the status of a performance anomaly report.

We can hence have a look at prior runs to see whether this occurred before. Alternatively, we can go to the “Continuous Benchmarking” tab. It provides aggregated performance statistics over time for every single experiment. Figure 5 shows the continuous benchmarking chart for the experiment. First, we should zoom in on the date of the experiment, i.e. August 30, 2022, to relax the visualization. Then, we can observe that on Aug 30, there was indeed a spike in execution time. This suggests that this is a performance anomaly.

Manually detecting such performance anomalies would be very cumbersome, particularly with hundreds of experiments being performed every single day. We have therefore integrated into the DJANGO REST API a mechanism to detect and report performance anomalies. Detected anomalies are reported on the “Dashboard”. For the particular experiment of Figure 4, a performance anomaly was detected, as can be seen in Figure 6. The report shows in which experiment an anomaly was detected, the date of the anomaly, and by which factor the performance exceeds a certain threshold. The threshold is derived from the standard deviation of the performance of the past two weeks. Clicking on the down arrow reveals the recent experiment and the continuous benchmarking charts immediately below the 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, as we exemplify in Figure 7. 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 “Looking At” or “Closed”.

As we can see in the continuous benchmarking report in Figure 5, this anomaly occurred just once. It is likely that this was due to unexpected load on the benchmarking server affecting the measurements. It is inconvenient if this anomaly will remain on the “Dashboard”. We can therefore easily update the status to “False Positive” and the anomaly will disappear from the “Dashboard”.

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

REFERENCES

- [1] The Apache Software Foundation 2022. *Arrow - A cross-language development platform for in-memory analytics*. The Apache Software Foundation. <https://arrow.apache.org>
- [2] 2022. *The LLVM Compiler Infrastructure*. <https://llvm.org>
- [3] 2022. *NoisePage*. <https://github.com/cmu-db/noisepage/tree/master/docs>
- [4] 2022. *NoisePage - Database Management System Project*. <https://noise.page>
- [5] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. 1988. GENESIS: An Extensible Database Management System. *IEEE Trans. Softw. Eng.* 14, 11 (1988).
- [6] G. Bernhardt. 2012. *Boundaries*. <https://www.destroyallsoftware.com/talks/boundaries>
- [7] R. T. E. Bruns. 2007. Instruction selection on directed acyclic graphs. (2007).
- [8] M. J. Carey and D. J. DeWitt. 1987. An Overview of the EXODUS Project. *IEEE Data Eng. Bull.* 10, 2 (1987).
- [9] DuckDB Foundation 2022. *DuckDB*. DuckDB Foundation. <https://duckdb.org/>
- [10] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. 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*, Danica Porobic and Thomas Neumann (Eds.). ACM, 6:1–6:7. <https://doi.org/10.1145/3399666.3399925>
- [11] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*.
- [12] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. 1989. Extensible Query Processing in Starburst. In *SIGMOD*.
- [13] I. Haffner and J. Dittrich. 2023. A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *EDBT*.
- [14] A. Kemper and M. Wallrath. 1987. An Analysis of Geometric Modeling in Database Systems. *CSUR* 19, 1 (1987).
- [15] T. Kersten, V. Leis, and T. Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *The VLDB Journal* (2021).
- [16] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. 2014. Building efficient query engines in a high-level language. *PVLDB* 7, 10 (2014).
- [17] Geoffrey Lentner. 2019. Shared Memory High Throughput Computing with Apache Arrow™. In *PEARC*.
- [18] P. Menon, T. C. Mowry, and A. Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB* 11, 1 (2017).
- [19] G. Moerkotte and T. Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *PVLDB*.
- [20] MonetDB B.V. 2022. *MonetDB*. MonetDB B.V. <https://www.monetdb.org/documentation-Jan2022/dev-guide/>
- [21] T. Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011).
- [22] Thomas Neumann. 2021. Evolution of a Compiling Query Engine. *Proc. VLDB Endow.* 14, 12 (2021), 3207–3210. <https://doi.org/10.14778/3476311.3476410>
- [23] PostgreSQL 2022. *PostgreSQL*. PostgreSQL. <https://www.postgresql.org/docs/14/index.html>
- [24] M. Raasveldt and H. Mühlisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*.
- [25] L. A. Rowe and M. R. Stonebraker. 1987. *The POSTGRES Data Model*. Technical Report, UC Berkeley, Dept of Electrical Engineering and Computer Science.
- [26] P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. 1986. Extensibility in the Starburst Database System. In *OODS*.
- [27] M. Stonebraker and L. A. Rowe. 1986. The Design of Postgres. (1986).
- [28] M. Stonebraker, W. B. Rubenstein, and A. Guttmann. 1983. Application of Abstract Data Types and Abstract Indices to CAD Data Bases. In *Engineering Design Applications, Database Week*.
- [29] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. 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*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 307–322. <https://doi.org/10.1145/3183713.3196893>