# Declarative Sub-Operators for Universal Data Processing

Michael Jungmair
Technische Universität München
jungmair@in.tum.de

Jana Giceva
Technische Universität München
jana.giceva@in.tum.de

## ABSTRACT

Data processing systems face the challenge of supporting increasingly diverse workloads efficiently. At the same time, they are already bloated with internal complexity, and it is not clear how new hardware can be supported sustainably.

In this paper, we aim to resolve these issues by proposing a unified abstraction layer based on declarative sub-operators in addition to relational operators. By exposing this layer to users, they can express their non-relational workloads declaratively with sub-operators. Furthermore, the proposed sub-operators decouple the semantic implementation of operators from the efficient imperative implementation, reducing the implementation complexity for relational operators. Finally, through fine-grained automatic optimizations, the declarative sub-operators allow for automatic morsel-driven parallelism. We demonstrate the benefits not only by providing a specific set of sub-operators but also implementing them in a compiling query engine. With thorough evaluation and analysis, we show that we can support a richer set of workloads while retaining the development complexity low and being competitive in performance even with specialized systems.

**PVLDB Reference Format:**
Michael Jungmair and Jana Giceva. Declarative Sub-Operators for Universal Data Processing. PVLDB, 16(11): 3461 - 3474, 2023.
doi:10.14778/3611479.3611539

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/lingo-db/subop-vldb-2023-reproducibility.

## 1 INTRODUCTION

Data processing engines are increasingly expected to support a richer set of workloads beyond relational SQL queries. If engines can not meet users' processing demands, standard extension mechanisms like user-defined operators or procedural extensions are used, but often at the cost of usability and performance [9, 13]. At the same time, existing engines are already bloated with internal complexity [45], which is further exacerbated by adding more complex features to the SQL standard [7]. Additionally, efficiently implementing operators for modern hardware leads to a lot of redundancy that decreases developer productivity and maintainability [40], and it is still unclear how data processing systems can keep up with the increased complexity of modern hardware.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611539

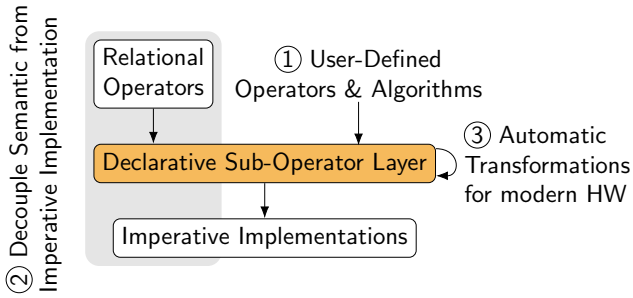


**Figure 1: Overview of our approach**

A few in our community have already identified that relational algebra should not be the *only* abstraction level exposed to users [1], or used internally [8, 17, 25, 45]. More specifically, there have been different proposals for abstractions that either focus on implementing complex operators [25], representing different workloads [39], or supporting modern hardware [2, 6, 41].

Fortunately, we argue in this paper that all of these problems can be addressed with a unified abstraction layer of declarative sub-operators as shown in Figure 1, if one were to follow a number of design principles. ① By exposing this new lower-level abstraction level, users and frontend libraries can use sub-operators to efficiently specify custom operators and algorithms. Thus, traditional extension mechanisms like UDFs and their tradeoffs can be replaced, and more workloads can be executed efficiently by the query engine. ② Furthermore, by using declarative sub-operators, we can decouple the semantic implementation of complex operators from the efficient implementation with imperative code. This helps to reduce internal complexity and make the implementation of complex operators like window functions more feasible. ③ Finally, because this new abstraction layer is based on fine-granular yet declarative sub-operators, one can perform new automatic optimizations and transformations. For example, instead of hard-coding operator-specific implementations for morsel-driven parallism [29] we can then automatically apply workload-agnostic transformations to reach the same result.

Designing such an abstraction layer requires balancing of *expressability* (i.e., can all desired algorithms be expressed efficiently) and *declarativity* (i.e., how easy can transformations be realized). In this paper, we propose a novel design for such an abstraction layer that is more expressible and explicit than prior work and can thus not only express complex relational operators efficiently but also user-defined operators and algorithms. However, we also demonstrate that despite the expressibility, many optimizations and transformations can still be applied effectively, including automatically introducing morsel-driven parallelism. We successfully implemented a concrete set of sub-operators in our compiling database

system LingoDB (cf. Section 6) and refined it based on the gained experience. Using this implementation[1], we demonstrate that we can not only support a richer set of workloads like complex analytical queries (full TPC-H and TPC-DS), user-defined operators, iterative algorithms (e.g., k-Means, PageRank), and numerical workloads, but also that we can achieve this without incurring performance penalties and while keeping the system's internal complexity low. We note that further optimizations like auto-vectorization could help to improve LingoDB's performance further, especially for numerical workloads, but we leave that to future work.

## 2 THE NEED FOR SUB-OPERATORS

Relational operators have been the cornerstone of relational database systems for decades. They are declarative and abstract away many implementation details. This allows for effective, automatic query optimization and enables transparent innovations under the hood. However, in the last years, increasingly more research works started questioning if perhaps the abstraction of relational operators is too high-level for many modern workload requirements. Thus, we argue that *in addition to* relational operators, we need another lower layer of *sub-operators* to solve the following problems.

**P1: Reducing Implementation Complexity** The efficient implementation of relational operators is complex and often requires a high implementation effort. Even basic operators such as aggregations can require complex, monolithic implementations (`select count(distinct a),count(distinct b)...`). In addition, databases frequently add support for new operators for e.g., array processing [31] or spatial data [14]. Furthermore, fused operators (e.g., GroupJoin, TopK) are introduced during query optimization to improve performance. Finally, increasingly more complex operators are being added to the SQL standard: From Window Functions and Ordered Set Aggregates (2003) to Row Pattern Matching (2016) and Property Graph Queries (2023)[7]. We argue that this is hardly manageable for development and maintenance without relying on *reusable and composable abstractions* below relational algebra.

**P2: Expressing Computations beyond SQL** Database systems are increasingly expected to support computations beyond what could be expressed with simple SQL queries. Especially iterative algorithms are hard to implement efficiently with SQL, but they are commonly required for graph analytics (e.g., page rank, shortest path) or data mining (e.g., k-means, dbscan). As already proposed by Bandle et al. [1], a reusable sub-operator layer can allow for efficient implementations of such algorithms. Furthermore, by using the same building blocks as relational operators, we can integrate complex algorithms seamlessly in queries. This avoids expensive materializations and allows the query optimizer to perform optimizations that e.g., push selections through user-defined operators.

**P3: Execution on Modern Hardware** Modern hardware is increasingly complex and often requires a high implementation effort in order to fully utilize it. For example, using morsel-driven parallelism [29] for efficient multi-threading requires adapting the implementation of every operator. In contrast, with sub-operators, we can automatically apply workload-agnostic transformations to reach the same effect (e.g., using thread-local states) without having to hard-code an implementation for every relational operator.

[1] https://github.com/lingo-db/lingo-db

**Resulting Requirements** In summary, we argue that we need a *unified* sub-operator layer that helps with *at least* the three problems outlined above. From these problems, we can derive six requirements for such a sub-operator layer

- R1 **Universal:** Sub-Operators should be designed universally to allow for reusability and composability. Then, new complex operators (P1) and new algorithms (P2) can be implemented by users without adapting the database engine.
- R2 **Declarative:** The design should be declarative to allow for optimizations and hardware-tailored implementations (P3)
- R3 **Control Flow Mechanisms:** Iterative algorithms require control flow e.g., in the form of loops (P2)
- R4 **Embeddable:** The sub-operator layer should be embeddable inside a tree of relational operators to integrate custom algorithms in the form of user-defined operators in a way that also query optimization can work well (P2)
- R5 **Explicit State:** State and its accesses should be explicit to enable automatic transformations for modern hardware environments (P3)
- R6 **Mutable Data Structures:** Efficient implementations of relational operators require mutable data structures (P1)

In addition, the sub-operator layer should be designed to allow for an *efficient* execution of relational and non-relational workloads.

## 3 CONCEPTUAL DESIGN

We propose a concrete abstraction layer built from sub-operators that meets all requirements discussed in Section 2. We start by identifying five major design principles as sketched in Figure 2:

(1) **Using tuple streams** for connecting different sub-operators enables us to embed sub-operator programs in relational operator trees for user-defined operators (R4). Furthermore, this leads to highly declarative, reusable, and composable sub-operators as we abstract from concrete input and output formats (R1, R2).

(2) **Explicit Declarative State** State is a first-class citizen and explicitly accessed (R5). By decoupling operators and states, logic can be reused with other data structures (R1). Furthermore, states and state accesses are declarative (R2) through *entry schemas* to allow for automatic optimizations.

(3) **References** To avoid redundancies and keep sub-operators reusable (R1), sub-operators should either perform state navigation or access a specific entry. For example, we do not include an operation that updates values in a hashtable, as this would perform both the lookup and the modification. Instead, lookups are explicit and return an *abstract reference*. This reference is then consumed by a second operation (e.g., scatter) that updates the referenced entry.

(4) **Explicit control flow** Sub-operators can either create a tuple stream by performing at most one logical loop, or consume a tuple stream. Control flow is explicitly implemented to ensure that sub-operator stay simple and reusable (R1) and to expose control-flow mechanisms (R3)

(5) **Views** offer additional properties on top of other states. This makes the design more declarative (R2) and especially enables the implementation of efficient and complex data structures (R6).

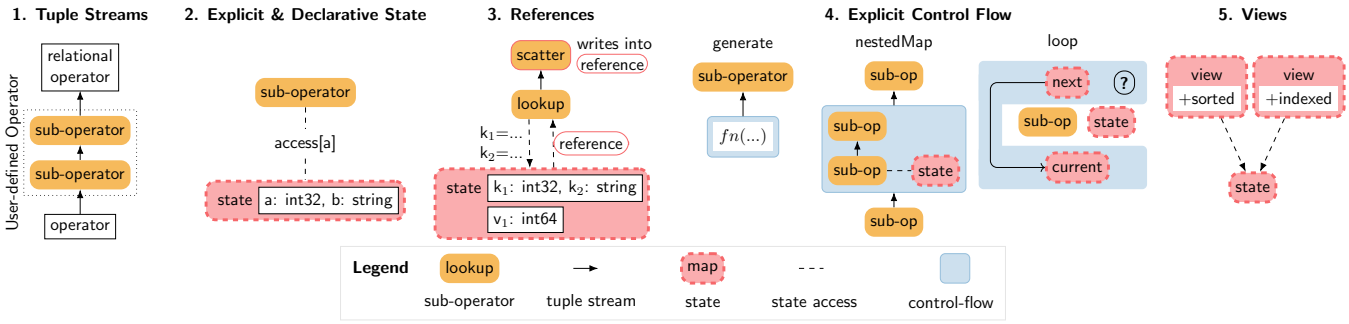


**Figure 2: The five major design principles for the proposed sub-operator layer**

## 3.1 Detailed Design

Following these design decisions, and based on the insights and challenges we faced during the implementation of the design in LingoDB (cf. Section 6), we developed a concrete design for an abstraction layer based on sub-operators. Note that we do not claim that it is the only possible design, but a concrete, implementable alternative that follows the design principles, meets the requirements, and *actually works* as we show in Section 7. More specifically, we did not aim for completeness (as the design can be further extended following the design principles) or minimality (as fused sub-operators can simplify optimizations: e.g., reduce instead of gather+map+scatter, scan instead of scanRefs +gather), but for a practical design. In addition, the detailed design may also vary depending on the concrete environment (e.g., compiling vs interpreting engine, target hardware architectures) and the workloads that should be supported.

**Table 1: Available State and View types**

| | Type / Sub-Operator | Description |
|---|---|---|
| states | table | relational table managed by the database |
| | single | holds exactly one entry |
| | buffer | stores entries linearly, but non-continuous |
| | map | for each key, exactly one entry is stored |
| | multimap | for each key, multiple entries can exist |
| | heap | stores only the smallest k entries (w.r.t ordering) |
| | array | stores a fixed amount of entries continuously |
| | resultTable | table that can be returned to the user |
| views | continuousView | entries can be accessed continuously |
| | sortedView | entries can be accessed in sorted order |
| | hashIndexedView | hash-based index enables lookups |
| | segmentTreeView | maintains partial aggregate values |
| | list | non-materialized list (e.g., result of lookup) |

More specifically, we propose a concrete set of state and view types as explained in Table 1. This includes external state types (e.g., table) with a fixed layout, for which we assume that the query engine can obtain a handle using some identifier (e.g., table name). Furthermore, seven intermediate data structures and four view types can be used. Finally, for handling variable-length results (e.g., for lookups), we introduce an abstract list type.

Similarly, Table 2 lists and specifies the semantics for all available sub-operators. Sub-operators form pipelines that process tuples following a tuple-at-a-time model. Only after every tuple is fully processed by all sub-operators, the next tuple is produced by e.g., a table scan. The proposed set of sub-operators includes different kinds of scan operators, operators for basic tuple stream manipulation, and operators for accessing states. Additionally, we propose sub-operators dedicated to references (produced by e.g., scanRefs, lookupOrInsert). The first category loads (gather) or updates (scatter, reduce) values from the referenced entry. The second category manipulates references by e.g., performing basic arithmetic operations for references to a continuous state. Such operations are necessary for implementing window functions and algorithms on arrays. Finally, three different, intentionally limited control-flow mechanisms are available: generate, nestedMap, and loop. The concept of nestedMap is sufficient for typical intra-pipeline control flow that is necessary for e.g., implementing nested loop joins or handling hash collisions. However, expressing iterative algorithms with nestedMap is hard, especially for algorithms that have no fixed number of iterations but e.g., terminate when the required accuracy is reached. Such iterative algorithms can be expressed with a loop. Finally, generate is useful for decomposing scalar values (e.g., splitting strings into words), or generating sequences (e.g., unique random numbers).

## 3.2 Using States and Sub-Operators

In the following, we discuss how the sub-operators can be used to implement a full SQL query as shown in Figure 3. Before starting with sub-operators, we assume that the database frontend has already parsed the SQL query and performed standard query optimization techniques, yielding a physical plan as shown in Figure 3.

Then, step by step, this physical plan is translated into a sub-operator plan. Both table scans are translated to a scan sub-operator on an explicit table (1), (5). The selection $\sigma_{a.x<5}$ is translated into a map operator (2) that evaluates the expression and a filter (3) that filters the tuple stream.

For the left outer join, we assume that the left side is smaller and, thus, used for building the multimap. Hence, each tuple on the left is inserted explicitly into the multimap (4). For every tuple on the right side, a lookup on the multimap is performed (6). The list of references to matching entries for the current tuple is passed through the tuple stream to the nestedMap as an argument. Inside the nestedMap, the list is scanned (7), and the required values are loaded from the reference pointing to the multimap entry (8) and

**Table 2: Available Sub-Operators with their semantic and exemplary use cases**

| | | | | |
|---|---|---|---|---|
| $t : T = \langle a := x, b := y \rangle$ | Tuple with columns a (value x) and b (value y) | $t.c_x$ | get column value of tuple by column name $c_x$ |
| $ts : [T] = [\langle ... \rangle \vert condition]$ | Definition of Tuplestream | $[\![m_1 := x, m_2 := y]\!] \in s$ | Entry of state $s$ with members $m_1, m_2$ and values $x, y$ |
| $t : T = t_1 \circ t_2$ | tuple concatenation | $(m, c) \in mapping$ | Mapping of member $m$ to column $c$ |
| $X...$ | variadic number of values of type $X$ | $c : column[x]$ | c is a column such that tuple.$c$ is of type $x$ |

| | Sub-Operator | Semantics | Exemplary Usage |
|---|---|---|---|
| scans | scan$(s : State, mapping) \rightarrow [T]$ | $[\langle c_1 := x_1, ... \rangle \vert [\![m_1 := x_1, ...]\!] \in s \wedge \forall i : (m_i, c_i) \in mapping]$ | tablescan |
| | scanRefs$(s : State, c_{ref}) \rightarrow [T]$ | $[\langle c_{ref} := r \rangle \vert r$ is reference pointing to $x \wedge x \in s]$ | window |
| | scanList$(list, c_{ref}) \rightarrow [T]$ | $[\langle c_{ref} := r \rangle \vert r \in list]$ | hash-join |
| stream | union$(l : [T]...) \rightarrow [T]$ | $[t \vert \exists ts \in l : t \in ts]$ | union all, outer join |
| | map$(t : T, f : T \rightarrow T) \rightarrow T$ | $t \circ f(t)$ $\quad f(t)$ computes a tuple with new columns | map, selection |
| | filter$(t : T, c_{pred} : column[bool]) \rightarrow [T]$ | $[t \vert t.c_{pred}]$ | selection, join |
| | rename$(t : T, rename)$ | $\langle c := t.c \vert \nexists (c, c_{new}) \in rename \rangle \circ \langle c_{new} := t.c \vert (c, c_{new}) \in rename \rangle$ | rename, |
| state access | materialize$(t : T, s : State, mapping)$ | $s.append([\![m := t.c \vert (m, c) \in mapping]\!])$ | nl-join, sorting |
| | insert$(t : T, s : State, mapping, eqFn)$ | $s.insert([\![m := t.c \vert (m, c) \in mapping]\!], eqFn)$ | hash-join |
| | lookup$(t : T, s : State, c_{ref}, eqFn)$ | $t \circ \left\langle c_{ref} := \begin{cases} r & \text{if } r = s.lookup(t, eq) \\ invalid & \text{otherwise} \end{cases} \right\rangle$ | hash-join, groupjoin |
| | lookupOrInsert$(t : T, s : State, c_{ref}, eqFn, initFn)$ | $t \circ \left\langle c_{ref} : \begin{cases} r & \text{if } r = s.lookup(t, eqFn) \\ s.insert(initFn) & \text{otherwise} \end{cases} \right\rangle$ | aggregation, window, set operations |
| access ref | gather$(t : T, c_{ref}, mapping) \rightarrow T$ | $t \circ \langle c := load(t.c_{ref}, m) \vert (m, c) \in mapping \rangle$ | hash-join |
| | scatter$(t : T, c_{ref}, mapping)$ | $store(t.c_{ref}, [\![m := t.c \vert (m, c) \in mapping]\!])$ | outerjoin, semijoin |
| | reduce$(t : T, c_{ref}, mapping, f)$ | scatter(map(gather$(t, c_{ref}, mapping), f), c_{ref}, mapping)$ | aggregation,window |
| references | unwrapFilter$(t : T, c_{ref}) \rightarrow [T]$ | $[t \vert t.c_{ref}$ is valid reference$]$ | groupjoin |
| | entriesBetween$(t : T, c_{refbegin}, c_{refend}, c_{between}) \rightarrow T$ | $t \circ \langle c_{between} := \Vert \{r \vert t.c_{refbegin} < r \leq t.c_{refend}\} \Vert \rangle$ | window, pagerank |
| | getStart$(t : T, s : State, c_{ref}) \rightarrow T$ | $t \circ \langle c_{ref} := r \rangle$ and r is reference to *first* entry $\in s$ | window, pagerank |
| | getEnd$(t : T, s : State, c_{ref}) \rightarrow T$ | $t \circ \langle c_{ref} := r \rangle$ and r is reference to *last* entry $\in s$ | window |
| | offsetBy$(t : T, c_{ref}, c_{resref}, c_{offset}) \rightarrow T$ | $t \circ \langle c_{resref} := t.c_{ref} + t.c_{offset} \rangle$ | window, pagerank |
| control flow | generate$(f : () \rightarrow [T]) \rightarrow [T]$ | $f()$ | intersect all, values() |
| | nestedMap$(t : T, nested : T \rightarrow [T]) \rightarrow [T]$ | $nested(t)$ | hash-join, window |
| | loop$(s_i : State..., n : State... \rightarrow State... \times bool) \rightarrow State...$ | $\begin{cases} s_{i+1} & n(s_i) = s_{i+1} \times false \\ \text{loop}(s_{i+1}, n) & n(s_0) = s_{i+1} \times true \end{cases}$ | k-means, pagerank, iterative algorithms |

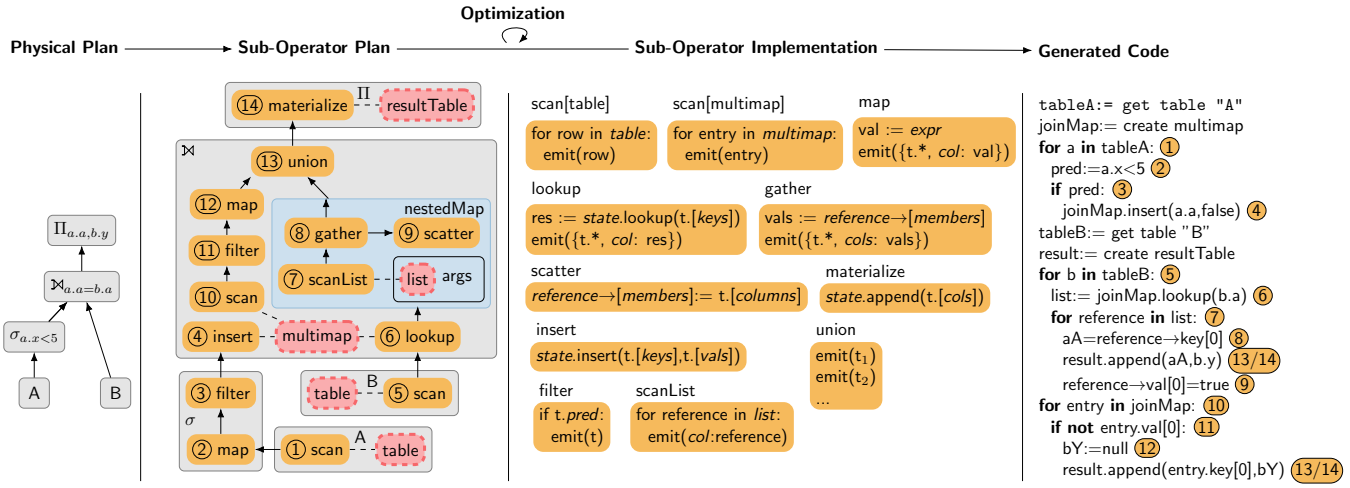


**Figure 3: Example: A query containing a left outer join is implemented with sub-operators**

returned as the result of the nestedMap. In addition, a scatter (9) explicitly writes into each entry (using the reference produced by scanList), setting a marker value to true to indicate that this entry had a join partner. Finally, all left tuples without join partner are computed: The multimap is scanned (10) and filtered for unmarked entries (11). Then all columns from $B$ are mapped to null (12) and unioned with the result of the nestedMap (13). Last, each tuple

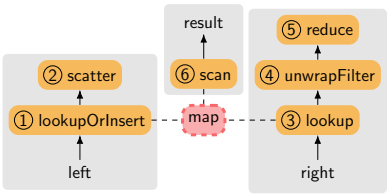


**Figure 4: Sub-Operator-based GroupJoin**

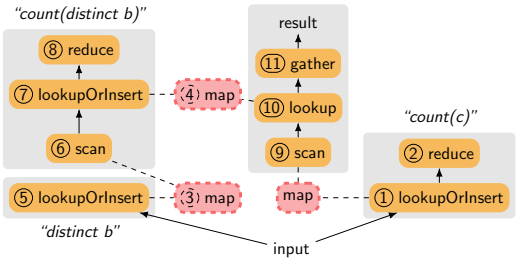


**Figure 5: Sub-Operator-based complex aggregation**

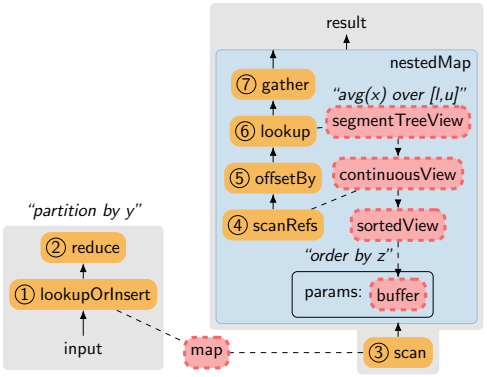


**Figure 6: Sub-Operator-based complex window function**

emitted by the sub-operators responsible for implementing the outer join is appended to a resultTable ⑭.

The resulting sub-operator plan also specifies the order in which the sub-operators should be executed. They may be, in principle, reordered if there is no read/write conflict. Next, automatic transformations and optimizations rewrite the sub-operator plan into another, semantically equivalent sub-operator plan for e.g., automatic parallelization as discussed in Section 5. Afterward, the sub-operators are implemented with imperative code. Because of our design decisions (state and control flow are explicit), the complexity of sub-operators is very limited. Thus, as sketched in the figure, the implementation for each sub-operator only consists of small code snippets that are then fused into efficient, tight loops, similar to code generated by data-centric code generation [36].

# 4 IMPLEMENTATION OF COMPLEX OPERATORS AND ALGORITHMS

Efficiently implementing complex (relational) operators can be challenging within existing systems: With increasingly complex operators being introduced (e.g., window functions), even implementing the correct semantics is non-trivial. In this section, we show that the proposed design significantly reduces this complexity by (1) leveraging the reusable sub-operators and (2) decoupling the semantic implementation from the applied optimizations. More specifically, in this section, we discuss the implementation of three different relational operators (groupjoin, complex aggregations, window functions) and show how user-defined operators and iterative algorithms can be implemented.

## 4.1 Complex Relational Operators

**GroupJoin** Fusing joins and aggregations to so-called GroupJoins has been proposed as optimization for more than 30 years [3, 32, 34, 48], as they improve performance by using the same hash-table for join and aggregation. Despite the performance improvements, not many systems implement GroupJoins because of the additional implementation effort. However, we can just reuse the existing sub-operators (already in use for aggregations and joins) to implement GroupJoins as shown in Figure 4. The build side first ensures that one map entry for every group-by key exists through a lookupOrInsert ① that inserts a new entry if necessary and returns a reference to the relevant entry. Then, additionally required column values are persisted into the referenced entry using the scatter operation ②. Next, for every tuple, the probe side performs a *lookup* ③ that returns an unsafe reference because potentially no entry was found. Because of the GroupJoin's semantic, we only have to consider tuples that match a group on the left. Thus, we filter the tuple stream to only include safe references ④ and perform the reduction ⑤ as specified by the GroupJoin (e.g., sum(x)). Finally, after every tuple on the right was processed, the map contains the result of the GroupJoin. Thus, a scan ⑥ is performed to produce the resulting tuple stream that can be consumed by following operators.

**Complex Aggregations** Even basic aggregations can sometimes become challenging. Take for example this query: `select a, count( distinct b), count(c)from R group by a`. Because of the *distinct*, we can not perform a simple aggregation using a single hashtable but also require duplicate elimination. Figure 5 sketches our implementation based on sub-operators in such cases: The input stream is processed twice: Once for the count(distinct) aggregate and once for the other, non-distinct aggregate. For the non-distinct aggregate, we create a map, and for each input tuple, the relevant bucket is looked up or created ①. On the returned reference, the reduction ② is performed according to the aggregate (e.g., $entry.count := entry.count + 1$). For the distinct aggregate, we first create two maps: One for deduplication ③ and one for the actual aggregation ④. A lookupOrInsert ⑤ on the deduplication map is performed for each input tuple, ensuring that each value is exactly contained once. Afterward, the map is scanned ⑥ to produce the distinct values used by the aggregation implemented with lookupOrInsert ⑦ and reduce ⑧. Finally, we have to join the aggregates from both maps to produce the final tuple stream. Thus, the first aggregation map is scanned ⑨, and for every tuple, a lookup ⑩ on the other map is performed. The corresponding aggregate values are then loaded ⑪ from the reference produced by the lookup.

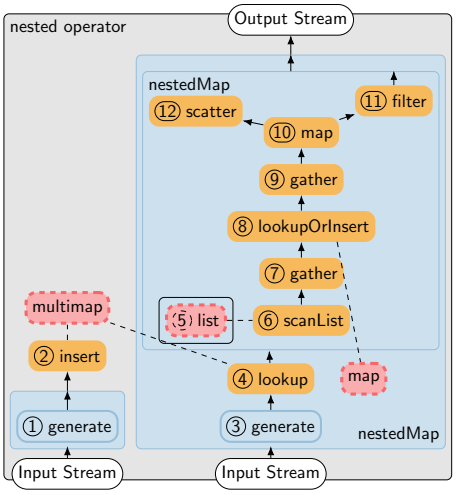


**Figure 7: User-Defined Operator for a similarity join**

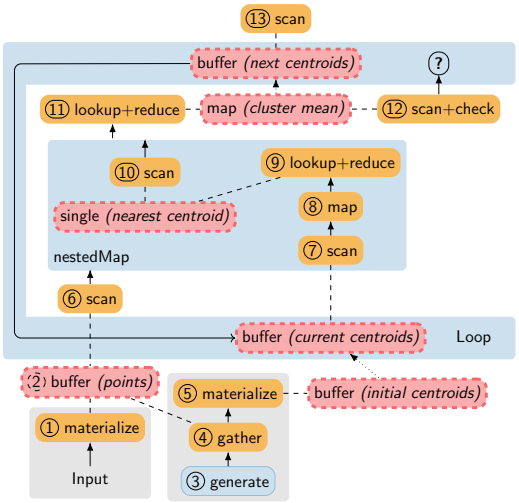


**Figure 8: K-Means realized with Sub-Operators**

**Window Functions** Window functions like `avg(x) over (partition by y order by z rows between N preceding and current row)` require not only partitioning and sorting but also efficient evaluation of aggregate functions on sub-ranges. Using sub-operators, we can first focus on specifying the correct semantics and then later perform optimizations. Figure 6 sketches how such a window function can be implemented. First, we partition the data using a map that contains a buffer for each entry: For each input tuple, the relevant entry is looked up or created ① and the required columns are persisted in the buffer with the reduction ②. After the partitioning is complete, we scan the map ③ and for each partition process the corresponding buffer in a nestedMap. Inside the nestedMap, we create a sortedView according to the specified order on top of the corresponding buffer. On top of the sortedView, two additional views are created: a continuousView and a segmentTreeView according to the aggregation function. The continuousView is then scanned ④, producing a reference for each entry. This reference is then offset by $N$ ⑤, to produce the lower bound. Finally, a lookup ⑥ using lower and upper bound (current reference) is performed on the segmentTreeView to obtain the aggregate for the window⑦.

## 4.2 User-Defined Operators

In addition to efficiently implementing complex but well-known relational operators, sub-operators are also useful for realizing *user-defined operators*. With user-defined operators, users can extend the functionality of a database engine by implementing custom operators that operate on one or more tuple streams. However, until now, user-defined operators either come with high overhead or require complex mechanisms inside the database engine [46]. We propose a *zero-overhead, low-effort* approach. Similar to how it is possible to use inline assembly in C programs, we can now embed sub-operator programs inside a relational query, as both work on tuple streams. For this, a new relational operator is introduced, whose semantics is specified by a nested sub-operator plan and can be analyzed by the query optimizer.

For example, let us assume that we want to perform a join based on string similarity. Figure 7 shows how such a join can be implemented as a user-defined operator with nested sub-operators, similar to the approach of Chaudhuri et al. [4]. For each string on the left side, we first ① generate all $n$-grams using a generator inside a nestedMap and then, for each n-gram, ② insert the current tuple into a multimap. For the right side, we also generate all $n$-grams for the current tuple ③. Then, we perform a lookup ④ in the multimap to find all possible candidates. Using a map, we now count how many n-grams are shared with each candidate and only emit candidates that have enough n-grams in common. This is implemented by ⑥ scanning the list ⑤ of all candidates and ⑦ loading the tuple id from the multimap entry. Then, the corresponding map entry is ⑧ looked up or inserted, and the current count is ⑨ loaded. A map sub-operator ⑩ computes $count + 1$ and $count = threshold$. If the second expression evaluates to true ⑪, the current candidate is emitted. In any case, the map entry is updated to contain the increased count ⑫.

## 4.3 Iterative Algorithms

Iterative algorithms are common in graph analytics, data mining, or machine learning. Current database systems support iterative algorithms, if at all, through (1) recursive SQL or (2) user-defined functions. However, neither is optimal: Recursive SQL is hard to formulate and can be inherently inefficient because of the lack of mutable data structures. User-defined functions are treated as a black box and can not be easily optimized, parallelized, or distributed. In contrast, by using sub-operators, iterative algorithms can be implemented declaratively and efficiently. Iteration is implemented by the *loop* operator, which iterates as long as a dynamic condition is fulfilled. Inside the loop, all sub-operators and state types can be used for efficiently executing the current iteration.

More specifically, we now discuss how k-means can be implemented with sub-operators. Figure 8 sketches the implementation, but smaller details are omitted to keep the figure reasonably small. First, all input points are ① materialized into a ② buffer. Then, $k$

random points are sampled from the input. Unique random offsets are emitted by the ③ *generator*, ④ the corresponding points are loaded and then ⑤ materialized into a second buffer. This buffer holding the initial centroids is the initial argument of the loop, which computes a new set of centroids in each iteration.

For each iteration, ⑥ all points are scanned and for each point, the nearest centroid is computed in a nestedMap. The current centroids are ⑦ scanned, and the distance function to the current point is computed ⑧. Then a ⑨ reduction (*nearestCluster* := *arg min dist*, *minDist* := *min*(*dist*)) is performed on a state of type *single*. After all centroids have been processed, the nearest centroid for the current point is emitted through a scan ⑩.

Afterward, a ⑪ reduction on a map is performed to compute the new average coordinates for each centroid. These new coordinates are then ⑫ compared with the previous coordinates to decide if the loop should continue. Finally, after the centroids converged (or an iteration limit is reached), the final centroids can be ⑬ scanned and e.g., used for prediction.

## 5 AUTO-PARALLELIZATION

We just showed how to implement the semantics of complex operators and algorithms using sub-operators. We now discuss how they can be automatically executed multi-threaded to fully benefit from modern many-core systems. In general, two main approaches are used for intra-query parallelism: Plan-driven parallelism hides parallelism in exchange operators [10], which allows keeping all other operators unmodified but requires static scheduling. In contrast, morsel-driven parallelism [29] dynamically schedules input tuples to be processed by pipelines of parallel-aware operators. While morsel-driven parallelism offers high scalability, performance, and elasticity, it also comes with a higher implementation effort as each operator needs to be implemented for parallel execution on top of the potentially complex operator semantics.

Instead of adapting every operator implementation, we propose to automatically transform sub-operator programs into parallel programs. By performing these transformations workload-agnostic on the sub-operator layer, not only SQL queries profit but also any other workload expressed as sub-operators.

**General Strategy** Algorithm 1 depicts the general algorithm for auto-parallelizing sub-operator programs. For every scan that neither guarantees a specific order of tuples nor is nested inside a nestedMap, we analyze the corresponding pipeline of sub-operators and try to parallelize it. First, all sub-operators of the current pipeline are collected (Line 4) and analyzed regarding Read-Write, Write-Read, and Write-Write conflicts (Line 5). Next, we try to eliminate each conflict with a transformation and store the transformations without applying them directly (Lines 8-18). Afterward, if no conflicts remain, the scan is marked as parallel (Line 20), and all saved transformations are stored in a global set (Line 21). Finally, after all scan operators have been processed, all transformations are applied (Line 25), and the pipelines can be executed safely in parallel.

**Thread-Local States** The main conflict resolution strategy as proposed by Leis et al. [29], transforms a global state into thread-local states that can be accessed and updated without any synchronization. Before executing subsequent pipelines that require a consistent global state, the thread-local states are merged to produce the final

**Algorithm 1** Auto-Parallelization Strategy

```
1:  transforms := {}
2:  for scan in program do
3:     if ¬nested(scan) ∧ ¬ordered(scan) then
4:        operations := getPipelineOperations(scan)
5:        conflicts := analyzeConflicts(operations)
6:        localTransforms := {}
7:        remainingConflicts := {}
8:        for op in conflicts do
9:           if state partitioning solves conflict then
10:             localTransforms.insert(partition_state(op))
11:          else if atomic execution solves conflict then
12:             localTransforms.insert(mark_atomic(op))
13:          else if state supports entry locking then
14:             localTransforms.insert(wrap_lock(op))
15:          else
16:             remainingConflicts.insert(op)
17:          end if
18:       end for
19:       if remainingConflicts is empty then
20:          mark scan as parallel
21:          transforms:= transforms ∪ localTransforms
22:       end if
23:    end if
24: end for
25: apply transforms
```

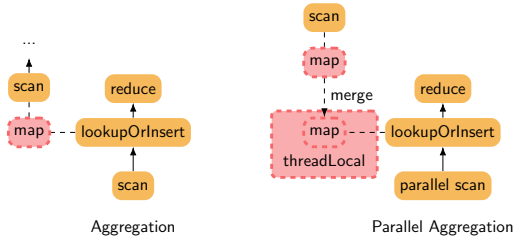


**Figure 9: Parallelization of an aggregation**

global state. This strategy works well, especially when new states are constructed, like materializing elements or performing aggregations. Figure 9 sketches this strategy for a simple aggregation. As the lookupOrInsert and reduce operators can lead to Write-Write and Read-Write conflicts, the map is wrapped in a thread-local state to eliminate the conflicts without incurring a synchronization overhead. Thus, the main pipeline can be executed in parallel. Afterward, the thread-local maps are merged based on the reduce operator into a map that is scanned to produce the final result.

**Atomic Execution** In many cases, no new states are constructed, but existing ones are updated. Then, the previous transformation can not be applied. However, if both the update function and the affected types are simple enough, sub-operators like scatter and reduce can be executed atomically on modern hardware. In such cases, we resolve conflicts by marking the sub-operator as *atomic* to enforce an atomic execution.

**Locking** If neither of the previous transformations can be applied, state entries can be locked before being updated. A lock sub-operator is, thus, introduced that first locks an entry through the provided reference before nested sub-operators are executed.

**Specialized Map Implementation** Making a map (e.g., used for aggregation) thread-local allows us to execute the pipeline in parallel. However, if we just use a regular hashmap, a lot of synchronization is required during the merge phase. Thus, following the idea of

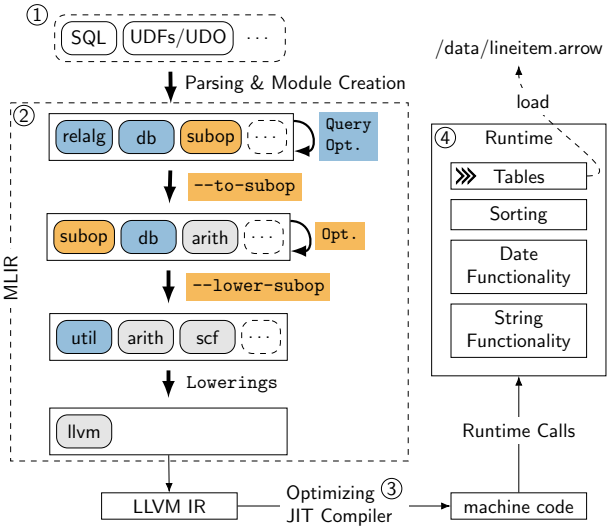


**Figure 10: Integration of the proposed Sub-Operator design in LingoDB ( MLIR builtin , LingoDB , sub-operators )**

Leis et al., we automatically switch to a special 2-phase map implementation. In the first phase, thread-local preaggregation fragments reduce the heavy hitters and partition all entries into $N$ buckets. During the merge phase, $N$ hash-tables can be constructed fully parallel without requiring synchronization over the entries of the corresponding buckets across all thread-local fragments.

## 6 IMPLEMENTATION IN LINGODB

With this paper, we not only want to propose a theoretical idea and design principles but also show that it can be implemented and applied in practice. In principle, the proposed ideas can be applied to systems that follow a push-based tuple-at-a-time model and do not suffer from more operator boundaries compared to relational operators. We implemented the proposed sub-operator design into LingoDB [17], our prototype system based on MLIR [28].

**MLIR** aims to reduce the engineering effort for developing domain-specific compilers. It features a single, standardized intermediate representation (IR) format that allows for introducing new abstraction levels through so-called dialects. Dialects bundle custom types, operations operating on these types, and attributes used for annotating compile-time constants and metadata. Compared to other IRs, MLIR is not flat, and operations can contain nested operations, which for example, allows for modeling explicit high-level for loops. MLIR further provides an infrastructure to define optimization passes that transform MLIR programs and comes with built-in passes (e.g., common subexpression elimination). Finally, high-level dialects are lowered progressively over multiple abstraction levels until a low-level dialect is reached (e.g., llvm), which can be executed.

**LingoDB** The overall goal of LingoDB is to blur the lines between databases and compilers by implementing database abstractions on top of MLIR and implementing query optimization as compiler passes. By building on MLIR and using *open* intermediate representations, relational operators can now be freely combined with other users of MLIR, such as machine-learning frameworks, and cross-domain optimization becomes viable. To achieve this, LingoDB introduces new high-level database-specific dialects to MLIR: The *relalg* dialect contains relational operators processing tuple streams and *db* specifies database-specific scalar types and operations. After the SQL query has been parsed and translated into an MLIR module, query optimization is performed through a *set of compiler passes* that rewrite the MLIR module accordingly. Finally, the operators are lowered to imperative code using push-based data-centric code generation [36] and further lowerings are applied until the llvm dialect is reached.

**Introducing Sub-Operators** For this paper, we added a new *subop* dialect that contains the sub-operators and the data structure types as proposed in Section 3. Additionally, we implemented a new *nested* operator in the *relalg* dialect that allows for embedding nested sub-operators in a relational query. This nested operator is also handled during query optimization to e.g., pushdown predicates through user-defined operators if possible. Figure 10 shows how LingoDB now processes queries using the implemented sub-operators: SQL queries are still translated into a high-level MLIR module that contains mainly operations of the *relalg* and *db* dialects. However, if user-defined operators are used, operations of the *subop* dialect are already present at this stage. Next, query optimization is performed on relational operators and sub-operators with a set of optimization passes. Afterward, a pattern-based lowering of relational operators into sub-operators is performed. The resulting sub-operator plan can then be further optimized by several new MLIR passes, including a *Parallelize* pass. Finally, a lowering pass generates imperative MLIR operations from declarative sub-operators by following the idea of data-centric code generation. Since we designed the sub-operators to be of low complexity, this lowering can also be performed by matching and applying lowering patterns. The generated, imperative MLIR operations are further lowered by the existing LingoDB infrastructure until the low-level *llvm* dialect is reached, which can be executed using LLVM [27].

**Implemented Optimizations** We apply the following optimizations on the sub-operator layer, some of which have already been discussed in prior work [23, 45].

(1) **GlobalSharing (GS)**: Share scans and states between independent pipelines.
(2) **ReuseLocal (RL)**: Avoid materializations and locally reuse already existing data structures e.g., reuse an aggregation hashtable for a subsequent join.
(3) **SpecializeStates**: State is specialized based on data properties and usage patterns. For example, multimaps can often be rewritten into a buffer for materialization and a hashIndexedView to allow for lookups, saving on costly resize operations and synchronization overhead.
(4) **DeferLoading (DL)**: By splitting gather operations and pushing them up as far as possible, we can defer loading and, thereby, improve performance.
(5) **EntryCompression (EC)**: By storing null indicators efficiently in one single integer, the entry size can be reduced by up to a factor of two by reducing padding requirements.

# 7 EVALUATION

In this section, we evaluate our proposed design and its implementation in LingoDB. First, we show that our approach significantly reduces the effort for implementing both complex relational operators and user-defined operators and algorithms. Then, we show that, despite these improvements, our implementation offers competitive performance for complex analytical workloads by comparing the overall system performance of LingoDB to state-of-the-art systems using the TPC-H and TPC-DS benchmarks. Next, we show that also workloads beyond relational queries can be executed efficiently in three case studies. Finally, we perform a more detailed ablation study on the implemented optimizations and analyze the scalability of LingoDB when using multiple threads.

If not noted otherwise, experiments were run multi-threaded on an AMD Ryzen 9 5950X CPU with a base frequency of 3.4 GHz and a maximal frequency of 4.9 GHz, and 64 GiB of main memory. The system runs Ubuntu 22.04.

## 7.1 Supported Workloads & Complexity

In Section 2, we argued that sub-operators can reduce the complexity to efficiently implement complex relational operators and support non-relational workloads.

**Supported Relational Workloads** Implementing complex relational operators in LingoDB has become much easier with sub-operators. Now we only have to express the operator's semantics using suitable reusable sub-operators. This effect can also be observed in Table 3, which shows lines of code for implementing selected relational operators for both our approach and in DuckDB. Even for operators like complex aggregations, window functions, or group joins, only a few hundred lines are required. Often, DuckDB requires around 4x more code, especially for implementing parallel execution, which we deal with automatically. Also, DuckDB does not implement some more complex operators like GroupJoins or intersect all, probably due to the required implementation effort. Altogether, the reduced implementation effort makes it possible for us to support a wide range of relational operators in LingoDB:

- 10 basic operators (e.g., tablescan, selection, map, sort)
- all set operations with set and multi-set semantics
- 7 different join operators that can be executed as nested-loop join, index-nested-loop join, or hash-based,
- complex aggregations and window functions
- fused operators for optimization: GroupJoin and TopK

Because of this extensive support for complex relational operators, LingoDB can now run even complex analytical benchmarks like TPC-DS [35], which require all of the listed operators (except for GroupJoin and TopK).

**Beyond Relational Workloads** We designed our approach such that we can also support other workloads, potentially embedded into a relational query. To demonstrate this, we fully implemented three different algorithms *outside of the database engine*: a custom similarity join algorithm as discussed in Section 4.2 and two well-known iterative algorithms: k-means and pagerank. This also includes pre- and post-processing phases that are typically ignored but are non-trivial to implement (e.g., sampling k random points for k means or mapping vertex identifiers to dense integers for pagerank). As discussed in Section 9, we are still working on user-facing frontends to make the developer experience more convenient. Thus, we implemented these algorithms by specifying the sub-operators directly in the highly verbose IR format of MLIR. But even with this more verbose style, we can specify all of these algorithms in a few lines as shown in Table 3.

**Table 3: Lines of Code required for operators in the database engine and different user-defined algorithms.**[2]

| operator | LingoDB | DuckDB |
|---|---|---|
| tablescan | 36 | 111 |
| sort | 52 | 202 |
| aggregation | 384 | 1358 |
| window | 349 | 524 |
| GroupJoin | 149 | n/a |
| except [all] & intersect [all] | 192 | n/a |

| algorithm | MLIR |
|---|---|
| similarityJoin | 94 |
| k-means | 214 |
| pagerank | 162 |

**Table 4: Overall lines of code for the integration in LingoDB**[3]

| Component | C++ |
|---|---|
| Sub-Operator Dialect | 2074 |
| Implementation of Relational Operators | 2143 |
| Implementation of Sub-Operators | 3203 |
| Optimization Passes | 1392 |
| Parallelization Pass | 347 |
| Pushdown through Nested Sub-Operators | 43 |

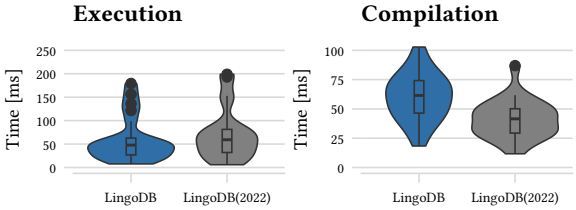


**Figure 11: Comparison to old LingoDB (TPC-H SF1, 1 Thread)**

**System Complexity** Despite the wide range of supported operators and workloads, the total amount of code is fairly small, as shown in Table 4. Especially, the amount of code required for automatic parallelization is small even though it spares a lot of effort and manages to fully parallelize every TPC-H and TPC-DS query.

## 7.2 Comparison with LingoDB$_{2022}$

Since we integrated our approach into LingoDB, we compare both execution and compilation times to the version of 2022 [17] single-threaded for TPC-H with scale-factor 1 as shown in Figure 11. We can observe that the execution performance improved slightly. This is not directly related to using sub-operators but to optimizations like introducing group joins that are now feasible and further improvements in query optimizations. The compilation latency slightly increased due to three factors: We added additional query

[2] window functions and groupjoin reuse logic of the aggregation operator
[3] whitespace and comments are excluded

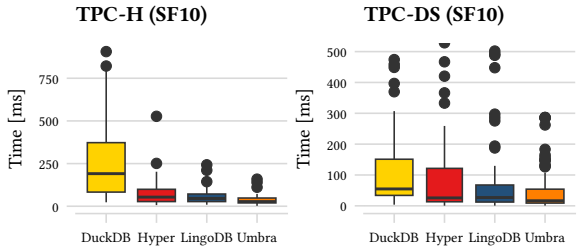


**Figure 12: Execution times for TPC-H and TPC-DS queries**

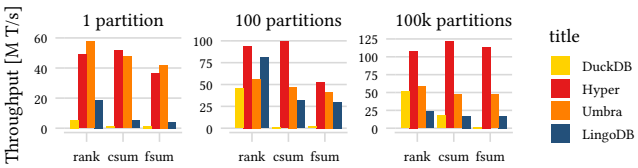


**Figure 13: Micro-benchmark for window functions**

optimization techniques since 2022, introduced the sub-operator layer, and generate more complex code for parallel execution even though we only execute it here with one thread. Still, the compilation latencies remain reasonable and could be further improved by orthogonal approaches like adaptive query compilation [24].

## 7.3 Performance for SQL Benchmarks

Many problems can be solved by introducing another layer of abstraction. However, in many cases, this additional abstraction layer has a negative impact on performance. To show that our system using sub-operators is competitive for analytical workloads, we compare LingoDB with the vectorized database system DuckDB [43], the commercial version of Hyper [20] published by Tableau [16], and the research database system Umbra [37]. Figure 12 depicts the overall execution times for both TPC-H and TPC-DS on scale factor 10. LingoDB significantly outperforms DuckDB by an average factor of 4.8 (TPC-H) and 3.9 (TPC-DS) and is, on average, 10% faster than Hyper for TPC-H and TPC-DS. Umbra remains faster because of orthogonal optimizations like better query optimization.

## 7.4 Micro Benchmark for Window Functions

In this paper, we claim that we can *efficiently* implement complex operators using sub-operators. We perform a microbenchmark to evaluate the performance of window functions similar to the ones conducted by Leis et al. [29]. We compare the execution times of LingoDB, DuckDB, Hyper, and Umbra for three different queries, each featuring a single window operator over 10M tuples while we vary the number of partitions. The three queries compute the rank, the cumulative sum (csum), and the sum of the preceding and following 100 values (fsum). The measured execution times are shown in Figure 13. Overall, we observe that LingoDB's performance is competitive as LingoDB is almost always faster than DuckDB but falls behind in performance to Umbra and Hyper.

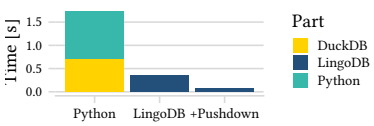


**Figure 14: Execution times for Similarity Join Example**

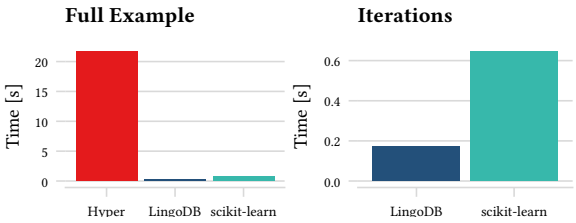


**Figure 15: Execution times for k-Means Example**

## 7.5 Case Study: Similarity-Join

As discussed in Section 4.2, sub-operators can be used for extending the functionality of database systems by declaratively specifying user-defined operators. We evaluate this feature based on the string similarity join introduced in Section 4.2. More specifically, we perform the evaluation based on an example query that finds similar movie titles from a certain genre in the IMDB-based JOB dataset [30]. For LingoDB, we implemented the proposed algorithm with sub-operators inside a *nested operator* embedded in the remaining query. This tight integration already avoids materialization barriers and benefits from parallel execution and generating data-centric machine code. Furthermore, using declarative sub-operators enables the query optimizer to perform advanced optimizations such as *pushing selections through* the user-defined operator.

The implementation in LingoDB is compared with a combination of DuckDB (standard query parts) and python (similarity join) in Figure 14. We can observe that already by avoiding materialization barriers, parallelization, and code generation leads to a speedup of more than 4 times. When adding a selective selection after the similarity join, the speedup increases further, as LingoDB can automatically push the selection through the user-defined operator.

## 7.6 Case Study: Iterative Algorithms

In the following subsection, we show that also complex iterative algorithms can be implemented efficiently outside of the database engine. We compare the performance of an implementation with sub-operators embedded inside a relational query against using recursive SQL in Hyper, a combination of DuckDB and optimized python packages (both scikit-learn and scikit-network use hand-optimized C implementations under the hood), and Weld [39].

We start by evaluating an implementation of the k-means algorithm as already discussed in Section 4.3. For this, we use an end-to-end example based on the New York taxi data set [47], which for a given time of the day, clusters all recorded trips based on the pickup location into 30 clusters and only reports the 5 clusters with

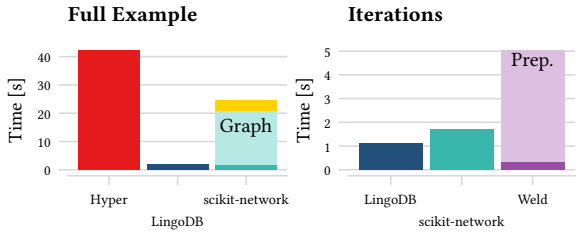


**Figure 16: Execution times for PageRank Example**

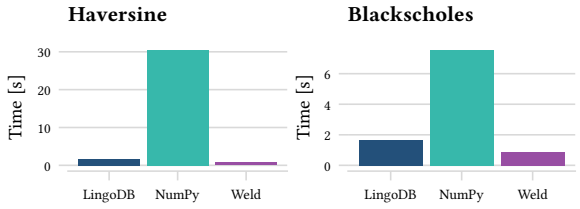


**Figure 17: Execution Times for numerical workloads**

the highest average lucrativity. As shown in Figure 15 LingoDB is faster in both the full example and the iterative part.

We also implemented the page-rank algorithm with sub-operators. For a concrete end-to-end example, we aim to identify successful actors using the page-rank algorithm on a graph $G$ and only emit the top ten actors. $G$ is computed from the IMDB-based JOB dataset [30] as follows: A directed edge $(x, y)$ is contained in $G$ for every movie that lists $y$ in the actor list before $x$. The execution times for both the full example and the iterative parts are plotted in Figure 16. We can observe that for the full query, LingoDB is faster than Hyper and DuckDB/scikit-network, which especially suffers from having to construct a certain graph format in python. When just comparing the iterative part, Weld is faster than LingoDB and scikit-network but also expects the graph to be in a certain format that matches Weld's execution paradigms. If we account for the preprocessing time required by Weld, LingoDB is significantly faster.

## 7.7 Case Study: Numerical Workloads

Since relational workloads are increasingly combined with numerical workloads, we also evaluate the performance for two numerical workloads already used for evaluating Weld [38]. Similar to Weld, we implemented a small python library that emits sub-operators, and profit from fusing pipelines using the *ReuseLocal* pass. The resulting execution times for LingoDB, Weld, and NumPy are shown in Figure 17. We observe that our approach is significantly faster than NumPy through optimizations and using parallelism. Weld has superior performance, most probably due to using vectorization, which especially helps with numerical workloads.

## 7.8 Effect of Optimizations and Parallelization

Finally, we analyze the speedup gained through the implemented optimizations and the auto-parallelization as shown in Figure 18. In an ablation study, we activate the different optimizations step by step, excluding specialization because of the high correlation

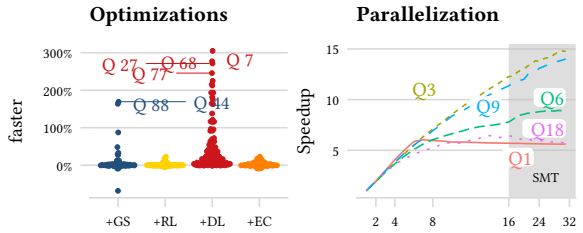


**Figure 18: Effect of optimizations and parallel execution.**

**Table 5: Comparison of related work with declarative IRs**

| | Voodoo [41] | Lolepops [25] | Weld [39] | Modularis [26] | Spark [49] | SDQL [44] | DBLAB [45] | **Ours** |
|---|---|---|---|---|---|---|---|---|
| Explicit States | ✓ | ~ | ✓ | ✗ | ~ | ✓ | ✓ | ✓ |
| Control Flow | ✗ | ✗ | ✓ | ~ | ✗ | ~ | ✓ | ✓ |
| Embeddable | ~ | ✓ | ✗ | ✓ | ~ | ~ | ~ | ✓ |
| Mutable DS | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ |
| Universal IR | ✗ | ✗ | ✓ | ~ | ✓ | ~ | ✓ | ✓ |
| Parallelization | ✓ | ~ | ✓ | ✓ | ✓ | ✓ | ~ | ✓ |
| Distribution | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Het. HW | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Vectorization | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |

with the parallelization strategy. Figure 18 shows the measured speedups of the execution time for all TPC-DS queries (SF 10).

We can observe that the *GlobalSharing* pass does not improve many queries. But when it does, it results in speedups of up to 2x. Similarly, only a few queries profit slightly from the *ReuseLocal* pass as the reused data structures are often small and do not impact the overall execution time much. In contrast, almost all queries benefit from the *DeferLoading* pass, up to a speedup of 3x. Finally, entry compression has a moderate effect: many queries profit slightly, but only some significantly (up to 25%). While theoretically, these optimizations can both decrease or increase compilation times significantly, we observed that most optimizations do not consistently speed up or slow down compilation. Only the *GlobalSharing* optimization consistently reduces the compilation time.

Furthermore, Figure 18 also displays the scaling behavior for selected TPC-H queries [21] as we increase the number of threads. It can be observed that our parallelization strategy works and scales roughly linear until the memory bandwidth is saturated or SMT is used. A similar scaling behavior can also be observed for other (compiling) systems like Umbra.

# 8 RELATED WORK

In this paper, we proposed a new design for an abstraction layer below relational operators. In the following, we discuss related work and contrast the most relevant *declarative* approaches with our approach as summarized in Table 5.

**Intermediate representations for compiling query engines** Over the last ten years, lower-level abstractions have especially been proposed to simplify the development of compiling query engines[17, 22, 44, 45]. We see this line of work orthogonal to our approach of introducing declarative sub-operators, as prior work focuses on decreasing complexity by introducing new high-level imperative abstraction layers.

**Unified Representations for Big Data Systems** Big data systems like Spark [49] have also introduced *unified* abstractions to abstract from challenges in distributed computing (e.g., resiliency) and make distributed data processing available for different kinds of workloads. At first glance, also Spark's operators seem similar to the sub-operators we proposed. However, only a few operators like map, filter, or nestedMap/flatMap are equivalent to the corresponding sub-operators, and most operators are much more high-level and complex: Control-flow for a `k-means` implementation and state such as hash-tables for `reduceByKey` are not exposed but are internally managed. Users can only choose to either use one of the provided complex operators (e.g., window functions) or implement their custom operator in imperative code using Spark's framework. Conceptually, Modularis [26], a distributed execution layer for data analytics, is quite similar to Spark. The main differences are that Modularis declarative sub-operators are designed to have limited complexity and be portable and operate on tuple streams instead of RDDs. Furthermore, many different *unified representations* aim to support different workloads, usually relational workloads combined with other high-level domains [5, 15, 18, 33].

**Weld** With Weld [39], Palkar et al. proposed a runtime for high-performance data analytics based on a common intermediate representation for a range of data-parallel workloads such as simple SQL queries, machine learning, and graph analytics. Two main differences distinguish Weld from our approach: First, Weld's IR centers around immutable data structures (e.g., *vectors* and *dictionaries*) created by merging values into *builders* in parallel loops. While this design simplifies dealing with automatic vectorization and parallelization, it also significantly reduces the expressibility of the IR, as many *efficient* implementations of algorithms require mutable state (e.g., SemiJoins). In contrast, even though our design allows for mutable states, we are still able to effectively auto-parallelize formulated programs. Even more important, every Weld program can be reduced to a sub-operator program that can be fully parallelized. Second, even though Weld can serve as a backend for relational queries, *embedding* Weld programs as *user-defined operator* inside a relational query remains difficult: (1) It would require a materialization and (2) relational optimizations such as pushing selections through a user-defined operator would remain difficult.

**Targeting Heterogeneous Hardware** Additionally, special intermediate representations have been introduced for (compiling for) modern hardware [2, 6, 11, 26, 41]. In 2016, Pirk et al. proposed Voodoo [41], a vector algebra for portable database logic across modern hardware. For generating code for different hardware (e.g., CPU, GPU), they introduce the Voodoo Algebra as an intermediate representation that can be produced from physical query plans. It features declarative stateless operators that describe data flow and state is managed explicitly. By design, Voodoo does not contain control-flow mechanisms and complex data structures beyond linear vectors to simplify the compilation for GPUs. Also, work on heterogeneous hardware uses the name "sub-operators" for pre-compiled kernels that are not declarative but can still solve problems like specialization for specific hardware [42] or placing OpenCL kernels across devices [19].

**Composing Relational Operators** Dittrich et al. made the case for *Deep Query Optimization* [8] and envisioned that relational operators should be broken up into more fine-granular sub-components to perform *deeper* query optimization, e.g., automatically specializing state types. In 2021, Kohn et al. proposed using *Low-Level Plan Operators* for implementing otherwise monolithic and complex statistical operators like aggregations and window functions [25] and optimizations such as reusing and specializing state. Similarly, Excalibur [12] also splits relational operators into low-level operators before implementing them with Voila [11].

In addition, we share many goals and basic ideas with the vision of Bandle et al. [1]. However, our concrete, implemented design is based on many novel design decisions, such as using references to decouple state lookup from state updates and views to make state transformations more declarative. Furthermore, we focus on implementing complex operators, user-defined operators, and parallelization and, most importantly, show that this approach is viable.

## 9 FUTURE WORK

The proposed sub-operator design opens up a lot of opportunities for future work.

**User-Facing Frontends and Libraries** We just showed that the proposed sub-operator design can be used to implement data processing algorithms beyond SQL. In that case, end-users still have to write sub-operator programs themselves. Thus, we plan on working on user-facing frontends and libraries similar to the approach taken by Weld [39]. Then, users could benefit from the tight integration and all the optimizations by just switching to a different library.

**Vectorized Execution** Gubner et al. demonstrated with VOILA [11] that a suitable *database intermediate representation* can be efficiently executed both in a vectorized and data-centric way. We believe that this is also feasible with our design: We could generate vectorized code for certain sub-operators but also execute sub-operator pipelines using pre-compiled vectorized functions to save on compilation times or profit from specialized implementations.

**Distribution and Heterogeneous Hardware** In Section 5, we already showed how we can automatically parallelize sub-operator programs. Similarly, we also plan to leverage the sub-operator design to automate the distribution of sub-operator programs across multiple machines and heterogeneous hardware. Since automating parallelism already solves similar problems (distributing work on different threads), we believe that our design will also be applied for distribution and heterogeneous hardware.

**Adaptive Pipelines** Currently, we simultaneously lower all sub-operators to imperative code. In the future, we could perform the lowering on pipeline granularity and thereby adapt to sometimes unexpected behavior.

## 10 CONCLUSION

This paper proposes a novel design for declarative sub-operators that supports increasingly diverse workloads, reduces the implementation complexity for relational operators, and allows for automatic transformations to e.g., introduce parallelism. We proposed five main design principles that help us balance expressibility and declarativity. Following these design principles, we proposed a concrete set of sub-operators and state types which were fully implemented in LingoDB. We showed that LingoDB can now support a richer set of workloads while keeping the development complexity low and being competitive in performance.

# REFERENCES


[1] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proc. VLDB Endow.* 14, 11 (2021), 2483–2490. https://doi.org/10.14778/3476249.3476296

[2] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.* 27, 6 (2018), 797–822. https://doi.org/10.1007/s00778-018-0512-y

[3] Damianos Chatziantoniou, Michael O. Akinde, Theodore Johnson, and Samuel Kim. 2001. The MD-join: An Operator for Complex OLAP. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 524–533. https://doi.org/10.1109/ICDE.2001.914866

[4] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 5. https://doi.org/10.1109/ICDE.2006.9

[5] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie J. Hendren. 2018. HorseIR: bringing array programming languages together with database query processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018*, Tim Felgentreff (Ed.). ACM, 37–49. https://doi.org/10.1145/3276945.3276951

[6] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[7] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258. https://doi.org/10.1145/3514221.3526057

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

[9] Yannis E. Foufoulas, Alkis Simitsis, Eleftherios Stamatogiannakis, and Yannis E. Ioannidis. 2022. YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proc. VLDB Endow.* 15, 10 (2022), 2270–2283. https://www.vldb.org/pvldb/vol15/p2270-foufoulas.pdf

[10] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 102–111. https://doi.org/10.1145/93597.98720

[11] Tim Gubner and Peter A. Boncz. 2021. Charting the Design Space of Query Execution using VOILA. *Proc. VLDB Endow.* 14, 6 (2021), 1067–1079. https://doi.org/10.14778/3447689.3447709

[12] Tim Gubner and Peter A. Boncz. 2022. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. *Proc. VLDB Endow.* 16, 4 (2022), 829–841. https://www.vldb.org/pvldb/vol16/p829-boncz.pdf

[13] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their usage in the wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391. https://doi.org/10.14778/3457390.3457402

[14] Ralf Hartmut Güting. 1994. An Introduction to Spatial Database Systems. *VLDB J.* 3, 4 (1994), 357–399. http://www.vldb.org/journal/VLDBJ3/P357.pdf

[15] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, Foto N. Afrati and Jacek Sroka (Eds.). ACM, 2:1–2:10. https://doi.org/10.1145/3070607.3070608

[16] Tableau Software Inc. 2023. Tableau Hyper API. https://tableau.github.io/hyper-db/docs/. Accessed: 2023-07-10.

[17] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401. https://www.vldb.org/pvldb/vol15/p2389-jungmair.pdf

[18] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[19] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (2017), 733–744. https://doi.org/10.14778/3067421.3067423

[20] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[21] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[22] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905. https://doi.org/10.1007/s00778-020-00643-4

[23] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *Proc. VLDB Endow.* 7, 10 (2014), 853–864. https://doi.org/10.14778/2732951.2732959

[24] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 197–208. https://doi.org/10.1109/ICDE.2018.00027

[25] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1001–1013. https://doi.org/10.1145/3448016.3457288

[26] Dimitrios Koutsoukos, Ingo Müller, Renato Marroquín, Ana Klimovic, and Gustavo Alonso. 2021. Modularis: Modular Relational Analytics over Heterogeneous Distributed Platforms. *Proc. VLDB Endow.* 14, 13 (2021), 3308–3321. https://doi.org/10.14778/3484224.3484229

[27] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. IEEE, 2–14.

[29] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[31] David Maier, Peter Baumann, Martin Kersten, Kian-Tat Lim, and Michael Stonebraker. 2013. ArrayQL algebra: version 3. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf. Accessed: 2023-07-10.

[32] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *Proc. VLDB Endow.* 4, 11 (2011), 843–851. http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf

[33] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. 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, 7:1–7:10. https://doi.org/10.1145/3399666.3399911

[34] Ryohei Nakano. 1990. Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions. *ACM Trans. Database Syst.* 15, 4 (1990), 518–557. https://doi.org/10.1145/99935.99943

[35] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1049–1058. http://dl.acm.org/citation.cfm?id=1164217

[36] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[37] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[38] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[39] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analysis. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf

[40] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384. https://www.vldb.org/pvldb/vol15/p3372-pedreira.pdf

[41] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (2016), 1707–1718. https://doi.org/10.14778/3007328.3007336

[42] Orestis Polychroniou and Kenneth A. Ross. 2020. VIP: A SIMD vectorized analytical query engine. *VLDB J.* 29, 6 (2020), 1243–1261. https://doi.org/10.1007/s00778-020-00621-w

[43] Mark Raasveldt and Hannes Mühleisen. 2019. 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*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[44] Hesam Shahrokhi and Amir Shaikhha. 2023. Building a Compiled Query Engine in Python. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, Clark Verbrugge, Ondrej Lhoták, and Xipeng Shen (Eds.). ACM, 180–190. https://doi.org/10.1145/3578360.3580264

[45] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1907–1922. https://doi.org/10.1145/2882903.2915244

[46] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131. https://www.vldb.org/pvldb/vol15/p1119-sichert.pdf

[47] New York City Taxi and Limousine Commission (TLC). [n.d.]. *TLC Trip Record Data*. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm

[48] Günter von Bültzingsloewen. 1989. Optimizing SQL Queries for Parallel Execution. *SIGMOD Rec.* 18, 4 (1989), 17–22. https://doi.org/10.1145/74120.74123

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.