View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Charting the Design Space of Query Execution using VOILA

Tim Gubner
CWI
tim.gubner@cwi.nl

Peter Boncz
CWI
peter.boncz@cwi.nl

ABSTRACT

Database architecture, while having been studied for four decades now, has delivered only a few designs with well-understood properties. These few are followed by most actual systems. Acquiring more knowledge about the design space is a very time-consuming processes that requires *manually* crafting prototypes with a low chance of generating material insight.

We propose a framework that aims to accelerate this exploration process significantly. Our framework enables synthesizing many different engines from a description in a carefully designed domain-specific language (VOILA). We explain basic concepts and formally define the semantics of VOILA. We demonstrate VOILA's flexibility by presenting translation back-ends that allow the synthesis of state-of-the-art paradigms (data-centric compilation, vectorized execution, AVX-512), mutations and mixes thereof.

We show-case VOILA's flexibility by exploring the query engine design space in an automated fashion. We generated thousands of query engines and report our findings. Queries generated by VOILA achieve similar performance as state-of-the-art hand-optimized implementations and are up to 35.5× faster than well-known systems.

PVLDB Reference Format:
Tim Gubner and Peter Boncz. Charting the Design Space of Query Execution using VOILA. PVLDB, 14(6): 1067 - 1079, 2021.
doi:10.14778/3447689.3447709

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/t1mm3/vldb_voila.

1 INTRODUCTION

Analytical queries process large data sets and their performance matters to users. Low runtimes can be achieved by a combination of efficient query execution operators that are arranged in an efficient plan, i.e. query optimization. While query optimization regressions or improvements strongly influence runtime, the execution engine can also significantly influence runtime [1]and, thus, cannot be ignored. The search for the best physical execution method features a very unfortunate risk-reward trade-off: (a) it requires exploring an extremely large space, (b) exploration itself is extremely work- and time-intensive and (c) has a very low success rate.

[1]Using tuple-at-a-time iterator-based interpretation – that was the dominant execution engine architecture up until the first decade of this millennium – as the baseline, we roughly estimate the overall performance improvement on TPC-H Q1 using results from papers: Vectorized execution [8] improves runtime by 40×. Data-centric compilation [22] is 2× faster than Vectorized execution. BiPie [32] further improves runtime by 3×. Then one can add morsel-driven parallelism [26] and gain another 48× (48 cores); or ≈ 10.000× in total without changing the plan.

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. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447709

Table 1: VOILA-synthesized queries are significantly faster than other open-source systems. DuckDB & LegoBase do not support parallelism. Runtimes in s, on TPC-H SF 10.

System	Q1	Q3	Q6	Q9
Single-threaded				
VOILA	0.59	0.93	0.15	2.07
DuckDB	5.71 (9.5×)	2.25 (2.4×)	0.64 (4.3×)	36.26 (17.5×)
LegoBase	0.78 (1.3×)	5.19 (5.5×)	0.29 (1.3×)	32.69 (15.8×)
24 threads				
VOILA	0.05	0.25	0.03	0.19
MonetDB	1.15 (24.3×)	0.36 (1.5×)	0.72 (28.8×)	0.30 (1.6×)
Weld	0.39 (8.3×)	3.05 (12.3×)	0.11 (4.3×)	6.90 (35.5×)

Vast Design Space. For a non-trivial query, the physical execution design space easily exceeds the logical query plan space (every operator can have multiple implementations) and grows, at least, exponentially (tree of operators with many possible implementations). The space is theoretically infinite, as, regardless of the actual space, the program can be inflated via No-Ops, or operations that compensate each other. In practice, due to memory constraints, the design space is finite but of astronomical size.

Slow Exploration. Exploring a single point (or a handful of points) typically requires either (a) engineering rather generic engine prototypes or (b) simulating an engine by hand-writing specific benchmark queries. The optimal choice depends on the scenario, e.g. for many queries it would advisable to rather have an engine instead of hand-rolling each query. In either case, for non-trivial benchmarks the exploration process requires tedious and time-intensive manual labor. For instance, implementing a query by hand typically takes days whereas developing an engine will require weeks to months.

To judge the relative performance, related work – the competitors – also have to be implemented. Often, the competitors are not readily available or contain slight nuances (e.g. different plans, different data structures, hash table sizes, SIMD) that render a fully fair ("apples-to-apples") comparison impossible. Consequently, more prototypes have to be implemented, increasing the development time even further.

Low Success Rate. Once a prototype has been created, its performance (runtime, memory footprint, energy-efficiency, etc.) can be measured and investigated. An exploration can be declared as a success, if it yields new insights into the design space. The initial bar is high, as reasonably "good" points have already been discovered (e.g. column-at-a-time [20], vectorized execution [8], data-centric compilation [31]), and a new, successful, exploration would need to improve upon them, e.g. by reducing runtime. Alternatively, one can declare success, when a new method has been discovered improves our understanding of the space.

Consequences. The unfortunate risk-reward ratio and high initial cost dis-incentives prototyping revolutionary new ideas. To reduce

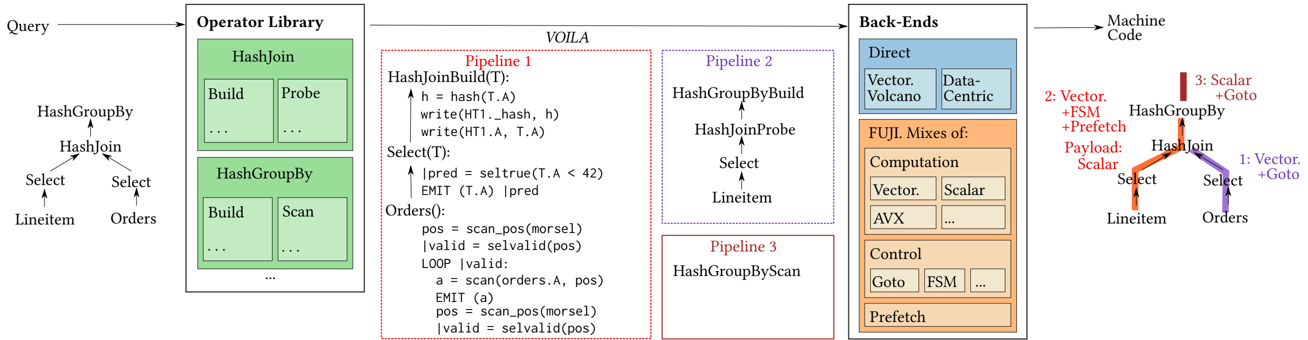


Figure 1: Architecture of our VOILA-based synthesis framework with various back-ends.

risk, new database architecture design points are typically chosen in proximity to already well-known points, i.e. improving a method. This leads to a vicious cycle where, we as a community, are *stuck* with further evolving already explored points in the design space.

Contributions. This project attempts to break this vicious cycle by making exploration significantly faster and far less work-intensive. Automating the exploration process can reduce months of development time into seconds and is, thus, literally millions of times faster than the state-of-the-art, which means: manually engineering prototypes. We present an extensible framework for exploring the design space of query execution. Figure 1 illustrates our framework. It is based on a novel domain-specific language (VOILA) that abstracts physical (low-level) query execution. VOILA is prudently designed to hit the sweet spot between performance (data-parallelism) and abstraction. It allows describing algorithms (e.g. a hash join probe) in a abstract fashion while achieving *competitive performance*. We demonstrate that state-of-the-art techniques (data-centric compilation [31] and vectorized execution [8]) can be synthesized from our high-level description. However, the flexibility of our language VOILA, is not limited to only these two techniques. Therefore, we present our novel and flexible code generation layer: FUJI. Through its artful design, FUJI can not only generate a plethora of different flavors as well as mixes thereof, but also reduces the manual effort required to add new code generators. We show VOILA's flexibility, by exploring the design space of well-known queries.

Results. Queries synthesized from VOILA, perform on par with optimized implementations of state-of-the-art query execution paradigms (data-centric compilation [31], vectorized execution [8], Interleaved Multi-Vectorization [13] , Relaxed Operator Fusion [29]). Compared to other state-of-the-art open-source systems optimized for fast data analytics, VOILA-generated queries are up 35.5× faster (see Table 1). VOILA's code generation modules are up to 33× more than well-known compilers and up to 18× faster to implement than engineering prototypes by hand. Consequently, VOILA can be translated into extremely efficient code and opens the way to quick prototyping and testing of new ideas, with limited manual effort.

Structure. The remaining paper is structured as follows: First, we introduce our novel language VOILA and, afterwards, formally define its semantics. With the basics of VOILA at hand, we describe how to translate VOILA to state-of-the-art paradigms data-centric compilation and vectorized execution. Afterwards, we present our novel flexible translation layer (FUJI). Finally, we evaluate the flexibility of our framework and discuss its performance. Then, we discuss related work and, afterwards, conclude the paper.

2 VOILA

During query evaluation, database systems often apply the same algorithms and data structures i.e. the same physical operators (hash join, hash group-by etc.). The major difference lies in their physical execution strategy (compiled/interpreted tuple/vector/column-at-a-time). We argue that operators should be described in a way such that we, later, can synthesize different execution strategies. Our domain-specific language VOILA (Variable Operator Implementation LAnguage) is tailored for this purpose. It describes the algorithmic details of an operator while abstracting the physical execution strategy away.

2.1 Core Concepts

High-level Languages are not well-suited. Various high-level languages can express algorithms relevant to database engines for which one can generate many different implementations (flavors). Notable examples are MIL/MAL [7], Voodoo [34], QMonad [40] and Weld [33]. However, these languages lack the ability of describing algorithmic details. Suppose we want to express a hash table lookup. Due to their level of abstraction, MAL, QMonad and Voodoo are unable to represent a simple hash table lookup. Instead, they enforce the usage of higher level concepts, like a hash join. Consequently, when generating many flavors from this description, one would have to re-implement many different joins. Being slightly more low-level, Weld can represent dictionary lookups via a primitive building block. This has two major disadvantages: (a) It is not easily possible to optimize specific fragments of the hash table lookup (e.g. fusing key checks for composite keys) without requiring a new lookup implementation. (b) When synthesizing different execution strategies, new black-box hash table lookups have to be re-implemented for each strategy.

Besides their inability to express algorithmic details, high-level languages often introduce complex nested data structures for intermediate values. Compared to simple intermediates (scalars, arrays), complex intermediates are typically slower to access (e.g. compare accessing a nested linked-list to a flat array) and consume more memory. To mitigate this problem, high-level languages have optimization passes that deal with removing complex intermediates. However removing these intermediates is a very costly process (in NP-time), called deforestation [43].

Both properties, the inability to express algorithm details and the problems caused by complex intermediates, render high-level languages unsuited for our purpose.

VOILA to the rescue. To mitigate these disadvantages, we propose to decompose data-structure specific operations (e.g. hash table lookup) into multiple simpler operations. In VOILA, we decompose a hash table lookup into: hash computation, bucket lookup, key check (gather & equality check), navigation to the next bucket and a loop to iterate over hash collisions. This decomposition describes *how* a hash table lookup is to be performed, but still leaves out specific details such as hash table design (bucket-chained, linear probing, ...), data layout (row-wise, columnar, hybrid) or execution strategies (data-centric, vectorized, interleaved prefetching, ...). From the example it is evident that VOILA is more high-level than low-level languages (e.g. C, LLVM IR [25] or ScalLite [40]), but more low-level than languages such as Voodoo and MIL/MAL. Thus, it bridges the gap between typical high- and low-level languages. After decomposing algorithms into high-level primitives, we can synthesize a specific implementation for each primitive and, thus, the whole algorithm.

Block-wise Execution. To abstract the physical execution from the logical description, operations in VOILA always operate on multiple values (vectors) at once. Scaling the vector size allows covering value-at-a-time (vector size 1) to column-at-a-time (vector size ∞) as well as the design points in between (e.g. SIMD).Most operations on vectors can happen completely data-parallel. The rationale of first-class support of block-wise execution is that rediscovering data-parallelism from sequential code is very hard, e.g. neither GCC nor LLVM properly vectorize loops with branches.

Inferred Types. Expressions and variables in VOILA do not specify data types, because they can be inferred automatically, from schema and program. This provides additional flexibility to add type-based optimizations (e.g. thinner data types for faster arithmetic [18]).

Predication. In our use-case, it is common to only process parts of the data, e.g. when tuples get logically removed (filtered out). In VOILA, filters create predicates. Predicates can be attached to operations, but can also be inferred from expressions. Note that we need predicates to synthesize efficient vectorized implementations á la Vectorwise [8].

Operator Context. Commonly, when lowering a higher-level into a lower-level language, e.g. physical plan into LLVM IR, the operator context disappears. After lowering, it will be very hard or impossible to determine which operations belong to which operators. To be able to, later, synthesize iterator-based operators from VOILA, we chose to keep the operator context.

2.2 Language

VOILA describes relational operators as imperative programs with high-level primitives. Each description consists of a list of statements (evaluation, assignment, `LOOP`, `EMIT`). Statements contain expressions. Expressions can either be literals (variables, constants) or functions on expressions (see Table 2). Statements, as well as expressions, logically operate on data vectors. Filters typically remove tuples from the flow. Instead of forcing materialization of vectors i.e. physically removing the deselected items, VOILA allows augmenting statements and expressions with predicates (`op |predicate`). Predicates can be thought of as (bit)masks that annotate whether an operation can safely be applied. If the predicate yields `true` for the tuple, the operation can be applied. Otherwise, the result of the operation is undefined. Though, that representation is conceptual,

Listing 1: Hash group-by operator in VOILA.

```
LOCAL HASHTABLE ht(k1 KEY, sum1 VALUE)                         1

GroupBy(T) {                                                   3
  h = hash(T[0])
  |miss = seltrue(true)                                        5

  LOOP |miss { // repeat until every tuple is processed        7
    bucket = bucket_lookup(ht, h)
    empty = eq(bucket, 0)                                      9
    |hit = selfalse(empty)
    |miss = seltrue(empty)                                     11

    LOOP |hit { // hash probing                                13
      htkey = gather(ht.k1, bucket)
      equal = eq(htkey, T[0])                                  15
      |found = seltrue(equal)
                                                               17
      // compute aggregates
      aggr_sum(ht.sum1, bucket |found, T[1])                   19

      // continue with non-matching tuples                     21
      |hit = selfalse(equal)
      bucket = bucket_next(ht, bucket |hit)                    23
      empty = eq(bucket, 0)
      |miss = selunion(|miss, seltrue(empty))                  25
      |hit = selfalse(empty)
    }                                                          27

    // optimistically insert non-matching tuples               29
    new_pos = bucket_insert(ht, h |miss)
    // copy key T[0] into column 'k1'                          31
    |can_scatter = selvalid(new_pos)
    scatter(ht.k1, new_pos |can_scatter, T[0])                 33
} }
```

in the data-centric VOILA back-end, no such bitmasks exist and the predicate will be translated into a branch. In VOILA, operators, as well as statements therein, are stateful, they e.g. maintain a hash table. The `EMIT` statement moves tuples of vectors, resulting from expressions, to the following operator. Logically, all expressions and statements operate on data vectors. Expressions either, in case of functions, apply the function element-wise on the data vector(s) – all vectors are required to have the same length – or, in case of a constant, broadcast a constant to all elements. The result of expressions can be stored in variables. Assignments behave similarly to other imperative languages and, logically, copy all values of data vector into the destination variable. This allows updating the same variable. Using different predicates, one can overwrite different positions of the same variable. Besides assignments, VOILA also allows fixed-point iteration via loops, similar to C's `while` statement. Different is that in VOILA the loop condition is a predicate and is only true, as long as at ≥ 1 items in the vector qualify.

Example. We explain VOILA using the hash-based group-by in Listing 1 as an example. First we declare the required data structures (line 1), then we describe the operator: Commonly an operator receives an input (`T`) which is a tuple of vectors. In this case, we use the `T` to find the final hash bucket, and to directly compute aggregates. We first extract the key (`T[0]`) and hash the value. Afterwards we initialize the predicate `miss` to select all tuples. As long as there are `misses`, we repeat the following process (7):

We lookup the first bucket and check whether it is 0, i.e. empty. For buckets that are $\neq 0$, we repeatedly follow the bucket chain (13), check the keys (13 & 14) and compute the aggregate(s) using these positions (19). Afterwards, we continue with the buckets that did not match any keys (21) and follow the bucket chain (23). If we noticed the end of the bucket chain, we have new `misses` (values that have to be inserted) and append them to the existing misses

Table 2: Expressions & Statements in VOILA. `x`, `y` **denote values or expressions.** `c` **denotes a table column.** `ht` **denotes a hash table.** `b` **denotes a hash bucket, also an expression.**

Operation	Description
Comparison/arithmetic/logic	
`eq(x, y)`	`x == y`
`not(x)`	`!x`
`add(x, y)`	`x + y`
`cast_i32(x)`	Casts `x` to (signed) 32-bit integer
`hash(x)`	Computes hash of `x`
...	
Hash Table	
`bucket_insert(ht, b)`	Create new bucket(s) in hash table
`bucket_lookup(ht, b)`	Given hash values, find initial buckets
`bucket_next(ht, b)`	Given bucket(s), find next bucket(s) in chain
Table/Array	
`scatter(c, b, x)`	Scatter values into bucket(s)
`gather(c, b)`	Gather values from buckets
`read_pos(t)`	Allocate next consecutive read position from table `t`
`write_pos(t)`	Allocate next consecutive write position from table `t`
`write(c, p, x)`	Consecutive write data to column starting from position `p`
`read(c, p)`	Consecutive Read from column starting from position `p`
Table Aggregates	
`aggr_count(c, b)`	Count active values (via predicate) in table's column `c` at index `b`
`aggr_sum(c, b, x)`	Sum values in table's column `c` at index `b`
...	
Data Inflow	
`scan_pos(t)`	Allocate next consecutive scan position
`scan(c, p)`	Returns column chunk from position `p`
Predicate	
`seltrue(x)`	Selected if `x` is true
`selfalse(x)`	`seltrue(not(x))`
`selvalid(x)`	Selected if `x` is valid. `[read, write, scan]_pos` can return an invalid position
`selunion(x, y)`	Selected if `x` is true or if `y` is true (`x` and `y` are both predicates)

(25). Then, we try to insert the `misses` (30) and copy the keys (33) which might fail. Finally, we repeat the insertion process until we found a bucket for every tuple.

3 FORMAL SEMANTICS OF VOILA

We formally define VOILA's semantics bottom-up: We start with basic expressions, afterwards step-wise broaden the semantics to statements, operators and, finally, the query.

3.1 Expressions

In VOILA, expressions logically apply functions to values. For example in Listing 1 (line 9), the expression `eq(bucket, 0)` applies the equality comparison on its arguments (`bucket` and `0`). Expressions can have predicates attached. Predicates indicate which values inside vectors are valid. In case there is no predicate attached, we attach a predicate that will return `true` for every value. We only define the actual result of an expression, when predicate is `true` (1), otherwise we define it as undefined ($\perp$).

Predicates. To conveniently apply predicates to functions, we define ϕ as the application of function f on its arguments $a_1, a_2, \ldots$ in the presence of a predicate $p \in \{0, 1\}$:

$$\phi(p, f, a_1, a_2, \ldots) := \begin{cases} f(a_1, a_2, \ldots) & \text{if } p = 1 \\ \perp & \text{otherwise} \end{cases}$$

Element-wise Application. To apply regular functions onto vectors, we define element-wise application (π). Let vectors be functions from an index set I to the result set R ($I \longrightarrow R$). For a vector $\vec{v}$, we can define $I = \{1, 2, \ldots, dim(\vec{v})\}$. Let $I_{\vec{a}}$ denote the index set of vector $\vec{a}$. We define the trivial element-wise application (π') as:

$$\pi'(i, f, a_1, a_2, \ldots) := \begin{cases} f(a_1(i), a_2(i), \ldots) & \text{if } c(i, a_1, a_2, \ldots) \\ \perp & \text{otherwise} \end{cases}$$

with $\begin{aligned} c(i, a_1, a_2 \ldots) :=& \big((a_1(i) \neq \perp) \wedge (a_2(i) \neq \perp) \wedge \ldots\big) \\ & \wedge\big((i \in I_{a_1}) \wedge (i \in I_{a_2}) \wedge \ldots\big) \end{aligned}$.

The function c defines which elements of the vectors are valid. An element is valid, if it is defined in terms of (a) $\perp$ and (b) the index set. The resulting index set of π' is the intersection of all input index sets: $I_{\pi'} = I_{a_1} \cap I_{a_2} \ldots$.

We define the element-wise application (π) as the inverse transformation of vectors to functions applied on π'.

Expressions apply a function f to its arguments and a global state W. We define the result of an expression η as the composition of predication (ϕ) and application (π):

$$\eta(W, f, \vec{p}, \vec{a_1}, \vec{a_2} \ldots) := \pi(\phi(f, W), \vec{p}, \vec{a_1}, \vec{a_2}, \ldots)$$

Using η, we define the expressions in VOILA as the element-wise application of a function f. Some expressions, namely `read` and `scan`, allow sequential access. These expressions behave similarly semantics to `gather` with the difference that the indices are sequential starting from a scalar offset (o). We transform expressions with sequential access into the same shape as normal expressions in VOILA described by a function f. Therefore, we add the identity vector $\vec{id}$ as an additional argument to π and, consequently, $f(a_1, a_2 \ldots)$. We denote such modified functions as $f_{seq}(a_1, a_2, \ldots, i)$ where a_1, a_2 are the arguments and i the i-th position of the identity vector id (i.e. $id_i = i$).

Table 3 defines most expressions as such a function f. To keep the notation concise, we implicitly broadcast scalar expressions (`ht`, `c`, `p`) to a constant vector with infinite dimension. The remaining expressions (`read_pos`, `scan_pos` and `write_pos`, `bucket_insert`) can have side-effects. Thus, we postpone their definition to the next section by rewriting them into statements: Let e be such an expression and x some unique identifier, then we rewrite $e(\vec{a}_1, \vec{a}_2, \ldots)$ into `assign`(x, $e(\vec{a}_1, \vec{a}_2, \ldots)$) and reference x in the expression(s) referring to the result of e.

3.2 Statements

As opposed to expressions, statements do not directly produce a result, but produce a side-effect (e.g. modifying a data structure, assigning variable) instead. For example in Listing 1 (line 4), the result of an expression is assigned to a variable i.e. modifies the variable's state.

Table 3: Expressions in VOILA defined as function f.

Operation	Semantics
Comparison/arithmetic/logic	
`eq(x, y)`	$f(W, x, y) = x = y$
`eq(x, y)`	$f(W, x) = \neg(x)$
`add(x, y)`	$f(W, x, y) = x + y$
`cast_i32(x)`	$f(W, x) = x$
`hash(x)`	$f(W, x) = hash(x)$
...	
Hash Table	
`bucket_lookup(ht, b)`	$f(W, ht, b) = get(W, D_{ht;bucket}[b])$
`bucket_next(ht, b)`	$f(W, ht, b) = get(W, D_{ht;next}[b])$
Table/Array	
`gather(c, b)`	$f(W, c, b) = get(W, D_c[b])$.
`read(c, p)`	$f_{seq}(W, c, p, i) = get(W, D_c[p + i])$.
Data Inflow	
`scan_pos(t)`	`read_pos(t)`
`scan(c, p)`	`read(c, p)`
Predicate	
`seltrue(x)`	$f(W, x) = x$
`selvalid(x)`	$f(W, x) \neq \bot$
`selunion(x, y)`	`or(x, y)`
...	
Other	
Variable v	$f(W, v) = get(W, V_v)$
Constant c	$f(W, c) = bcast(c)$

Side-Effects. To formally encapsulate side-effects, we define a "world state" W. Each statement $S :: (W, A) \rightarrow W$ has input arguments A and "modifies" W. A chain of two statements S_1, S_2 would hand though the world state W: $W_{final} = S_2(S_1(W_{init}, A_1), A_2)$. By induction we can construct chains of arbitrary length.

Global State. Our constructed world, or global state, W contains mappings for variables ($W.V_{var\ name}$), as well as mappings for data structures ($W.D_{struct\ name}$). Further we define getters and setters: A getter returns the specified value i.e. $get(W, m) := W.m$. A setter "modifies" the global state W by creating a new state W'. $W' := set(W, m, v)$ creates a copy of W, namely W', with $W'.m = v$. Using both, getters and setter, we can modify data structures and values assigned to variables in the global state W.

Element-wise Application. Similar to Expressions, we define the element-wise application for statements. The difference is that statements and (rewritten) expressions have side-effects and, thus, we need to (a) carry the global state around and (b) specify an evaluation order. Therefore, we define the element-wise application until an index m as:

$$A'_m(W, s, i, \vec{p}, \vec{a_1}, \vec{a_2}, \dots) := \begin{cases} A'_m(W', s, i+1, & \text{if } i \leq m \\ \quad \vec{p}, \vec{a_1}, \vec{a_2}, \dots) & \\ W' & \text{otherwise} \end{cases}$$

$$\text{with } W' = \begin{cases} s(W, i, \vec{a_{1,1}}, \vec{a_{2,1}}, \dots) & \text{if } (p = 1) \wedge (i \leq m) \\ W & \text{otherwise} \end{cases}.$$

Using A'_m, we define the element-wise application A as:

$$A(W, s, p, a_1, a_2, \dots) := A'_m(W, s, 1, \vec{p}, \vec{a_1}, \vec{a_2}, \dots)$$

with $m = min\{dim(\vec{p}), dim(\vec{a_1}), dim(\vec{a_2}), \dots\}$, $\vec{p} = \theta(W, p)$, $\vec{a_1} = \theta(W, a_1)$, $\vec{a_2} = \theta(W, a_2)$ ... Using A, we define statements as the application of a function s onto each element. With the element-wise application and the following helpers, most statements can be defined as in Table 4:

Table 4: Statements in VOILA. i is the index inside vectors.

Pattern	Semantics
$\mathtt{assign}(W, v, \vec{e})$	$s(W, i, v, \vec{e}) = set(W, V_v, \theta(W, \vec{e})_i)$
Writers	
$\mathtt{write}(c, e_{pos}, \vec{e_{val}})$	$\mathtt{scatter}(c, bcast(e_{pos}), \vec{e_{val}})$
$\mathtt{scatter}(c, \vec{e_{idx}}, \vec{e_{val}})$	$s(W, i, c, \vec{e_{idx}}, \vec{e_{val}}) = set(W, D_c[\theta(W, \vec{e_{idx}})[i]], \theta(W, \vec{e_{val}})[i])$
Aggregates	
Let f be an aggregation function, e.g. $f_{\text{sum}}(e_{old}, e_{val}) := e_{old} + e_{val}$	
$\mathtt{aggr_*}(c, \vec{e_{idx}}, \vec{e_{val}})$	$s(W, i, c, \vec{e_{idx}}, \vec{e_{val}}) = W'$ with $k = \theta(W, \vec{e_{idx}})[i]$ $W' = set(W, D_c[k], f(get(W, D_c[k]), \theta(W, \vec{e_{val}})))$
(Rewritten) Position Allocators	
For $*\mathtt{_pos} \in \{\mathtt{read_pos}, \mathtt{scan_pos}, \mathtt{write_pos}\}$	
$\mathtt{assign}(W, v, *\mathtt{_pos(t)})$	$W' = set(T, V_v, \theta(W, r))$ with $r = get(W, D_{t;*_pos})$ $T = set\Big(W, D_{t;*_pos}, \theta\big(W, \mathsf{add}(r, dim(r))\big)\Big)$

- $\theta(W, e)$ evaluates an expression e and returns its value.
- $C[i]$ accesses a value inside C at position i. This makes C a mapping from i to values.
- $\vec{e} = bcast(d)$ broadcasts d to all entries of $\vec{e}$ ($\forall i \vec{e_i} = d$).

`bucket_insert` tries to create $k = dim(b)$ buckets. It is possible that collisions happen (conflicting indices inside a vector), therefore the result can be (a) a successful insertion of the bucket or (b) a failure to insert. In the latter case, the insertion procedure will have to be repeated. `bucket_insert` either returns a bucket index, or $\bot$ for failure. We define $\mathtt{assign}(W, v, \mathtt{bucket_insert}(ht, b))$ as:

$$s(W, i, ht, b) = \begin{cases} set(W_3, V_v, r) & \text{if } conflict(\vec{b}, i) = 0 \\ set(W, V_v, \bot) & \text{otherwise} \end{cases}$$

with:

- $W_1 := set\big(W, D_{ht;next}[b], get(W, D_{ht;bucket}[b])\big)$,
- $(W_2, r) := allocate(W_1)$ allocates a position in the hash table and returns new state W_2 and position r.
- $W_3 := set(W_2, D_{ht;bucket}[b], r)$

$$conflict(\vec{b}, i) := \begin{cases} \left|\{\vec{b}[i]\} \cap \{\vec{b}[1], \dots, \vec{b}[i-1]\}\right| & \text{if } i > 1 \\ 0 & \text{otherwise} \end{cases}$$

`LOOP` repeats a statement S until a fixed-point is reached i.e. condition is not satisfied. We define `LOOP` as:

$$L(W, S, P) = \begin{cases} L(S(W), S, P) & \text{if } count(\theta(W, P), 1) > 0 \\ W & \text{otherwise} \end{cases}$$

3.3 Operators

In VOILA an operator is described by a function `op(input)`. For example in Listing 1 (line 3), this is `GroupBy(T)`.

`EMIT`. The `EMIT` statements transports tuples from the current operator to the next operator in the pipeline, or outputs tuples, last operator in the last pipeline, to the user. With knowledge of the current pipeline P, the following operator is known and static (will

Translate(Operator o, Input T):

(1) Loops: `LOOP |p ...` → `while (p) { ... }`

(2) Remove predicates:
- `r = x |p` → `if (p) { r = x }`
- `p = seltrue(x)` → `p = x`
- `p = selfalse(x)` → `p = !x`

(3) Implement operations:
- `r = hash(x)` → `r = HASH(x)`
- `r = read(col, idx)` → `r = col[idx]`
- `scatter(col, idx, val)` → `col[idx] = val`
- `EMIT x` → Translate($o + 1$, x)
- ...

Figure 2: Translation of statements/expression from VOILA to data-centric program. Order of application matters.

not change). This allows rewriting the VOILA program into a program without `EMIT`. To handle `EMIT`, we fully transform the VOILA program. Let the current pipeline be P, $|P|$ the number of operators in P and P_i the specific function of operator i. For every operator o (with operator function $P_o(W, x)$, with x being its input, or $\emptyset$ for scans), we replace every occurrence of `EMIT(x)` (typically only one), with $P_{o+1}(x)$ when $o + 1 < |P|$ (otherwise it would print tuples to the user). We repeat this process until no `EMIT` can be replaced. As a consequence, there will only be *one* remaining operator function (every operator has been inlined into the scan operator P_1) and the remaining `EMIT`s add tuples to the result R. We define the result R as a list, part of the global state W, and let $append(L, x)$ a function that appends a value x at the end of the list L. Like regular statements, we define (the remaining) `EMIT` as:

$$W' = \begin{cases} set(W, R, append(get(W, R), x)) & \text{if } p = 1 \\ W & \text{otherwise} \end{cases}$$

Execution & Termination. With the above toolkit, we can answer a query Q. Let the query consist of multiple pipelines $Q_1, Q_2, \ldots$. We construct an initial state W_0 with empty result R, empty set of variables V and set of data structures D consisting of base tables. Using the initial state W_0, we can evaluate each pipelines consecutively, resulting in a new state (a W') and feed W' into the next pipeline, and so on. We evaluate a pipeline using $eval(W, Q_n) := eval(W, Q_{n;1})$ which evaluates the first operator function ($Q_{n;1}$, the P_1 of a Q_n) of the pipeline until the end of the operator function is reached. We repeat this process until all pipelines have been evaluated. Afterwards the final result of the query is stored as $W.R$, in the final state W.

4 DIRECT SYNTHESIZER BACK-ENDS

Using descriptions in VOILA, we can synthesize different execution styles. We created simple back-ends that directly generate the state-of-the-art paradigms in C++: data-centric as used in HyPer [31], and iterator-based vectorized [8] code, as used in Vectorwise.

4.1 Data-Centric Program

We first re-cap data-centric compilation and afterwards describe how to synthesize data-centric code.

Data-Centric Compilation, first, splits query plans into pipelines. Pipelines start from scans (base table, group-by etc.) and end in a materializing operator. For each pipeline, data-centric compilation fuses all operators into one loop while, typically, generating scalar code for the operator's body. Thus, we focus on synthesizing scalar code, but our synthesis strategy is not limited to it.

Synthesis. Similar to the original approach by Neumann [31], we first split the query plan into pipelines and inline all operators in one pipeline into one loop. This gives us data-centric pipelines in VOILA, which we then lower to executable code. During the translation, we assume a vector size of 1 (i.e. scalar) and directly expand operations in VOILA using the set of rules listed in Figure 2.

4.2 Iterator-based Vectorized Program

Alternatively, one can also translate VOILA into an iterator-based and vectorized program. Currently, our framework can only synthesize unary operators and, thus, we split binary operators into pipelines as for data-centric code.

Iterator-based Operators. It is a traditional approach to implement physical operators as iterators by providing an `open-next-close` interface. Using this interface, operators pull the next tuple from the child operator(s) by calling `next`. This both reduces the size of intermediates materialized in memory (one tuple) as cache-efficiency (tuple is produced and immediately consumed).

Vectorized Interpretation. Traditionally, the iterator-based model only returns one tuple at a time. This leads to high interpretation overhead [8]. As a mitigation, the iterator-model can be extended to return multiple tuples. To further cut down interpretation costs, basic expressions also need to operate on multiple tuples. This is known as "vectorized interpretation" [8].

Synthesis. To generate an iterator-based vectorized program, we synthesize an operator implementation that implements the `open-next-close` interface. Inside the operator we need to construct expressions (`open`), evaluate them (`next`), deallocate resources (`close`), as well as, maintain the operator's state. Since we keep the operator context for each operator, we expand a basic operator template.

For each expression in VOILA, we need to generate code that constructs an expression (`Expr a_plus_b("add", a, b);`). At a later point, the top-most expression will trigger the recursive evaluation of its input expressions (lazy evaluation). For statements we generate specialized code:

- `EMIT` is translated into code that moves tuples (expressions) into the operator's output and returns tuples.
- `LOOP` translates into a loop including evaluating the loop predicate. It can happen that loops refer to in-flight, not yet evaluated, expressions either from out-side the loop, or from a previous iteration. In such cases, we need to evaluate these in-flight expressions.
- All remaining statements enforce the evaluation of their respective input expressions.

For each operator, we build expression trees which are evaluated either via statements, or when the result of the operator is requested (`next`).

5 FUJI – THE FLEXIBLE BACK-END

Our two direct back-ends can generate executable code for queries, using operators described in VOILA according to two completely different state-of-the-art flavors: data-centric and vectorized. However, many more flavors can be generated from VOILA. Therefore, we designed a third, more generic, back-end: Flexible Unified JIT

Table 5: Known points in the FUJI's design space

Flavor	Computation	Control	Prefetch	Buffer
Data-centric [31]	Scalar	Goto	✗	✗
Vectorized [8]	Vec. Primitive	Goto	✗	✗
AMAC [24]	Scalar	Conc. FSMs	✓	✗
ROF [29]	Scalar	Goto	✓	✓
IMV [13]	SIMD	Conc. FSMs	✓	✓

Infrastructure (FUJI). FUJI makes code generation (1) more flexible and (2) easier to extend and debug. It (a) decomposes code generation into components, (b) allows freely mixing them, and (c) is logically splitting code generation into different modules serving different purposes.

5.1 Component-based Flavor-Generation

We decompose code generation into basic components: Computation, Prefetching, Control and Buffering.

Computation. The computation component translates expression trees into scalar operations (e.g. Hyper), SIMD operations or calls to vectorized primitives (functions that process column chunks in a tight loop).

Prefetching. Recent work has shown that software prefetching can significantly improve performance [13, 24, 29] and therefore should be part of modern query engines.

Control. The Control component decides how `EMIT` and `LOOP` statements are translated. This can be a goto-based program or multiple finite state machines (FSMs). Multiple FSMs have the advantage that each FSM can run concurrently and allow overlapping prefetching with other operations, e.g. one FSM issues memory loads via prefetch instructions, while waiting the other FSMs can proceed.

Buffering. Selective operators (e.g. filters) or predicates can remove tuples from the flow. However, to achieve full utilization of SIMD lanes (or ALU in general) it is, in some cases, advisable to physically eliminate filtered-out tuples (i.e. typically materializing a chunk of the relation). Typically, buffering becomes more expensive with more columns, but leads to gains in subsequent operators/operations.

This decomposition covers the state-of-the-art, as Table 5 shows, as well as the space in between.

5.2 Flexible Unified JIT Infrastructure (FUJI)

Typically, code generators tend to be huge monoliths. For example, the early code generator of Hyper was about 10K lines of code [31]. To increase flexibility and extensibility, we split the logic of our FUJI back-end into multiple modules: Target Codegen, Generic Codegen and CLite.

Target Codegen. The target specific code generators synthesize optimized CLite code for specific implementations for expressions, buffering of intermediates and prefetching. We implemented 3 targets: `scalar`, `vector` and `avx512`. The `scalar` target generates data-centric code [31] similar to Section 4.1. The `vector` target generates calls to vectorized primitives which is an alternative implementation of vectorized execution, compared to iterator-based vectorized program (Section 4.2). In addition, FUJI currently provides an `avx512` target, which processes blocks of 8 values-at-a-time and, if possible, uses AVX intrisincs to process data. Note, not all operators are (a) possible using only the AVX instruction sets (e.g. `bucket_insert`), nor (b) benefit from it (e.g. aggregations or sub-word gathers). Selective processing is implemented using AVX-512 bit-masks. If AVX cannot be used, we check the mask and generate scalar code for each of the 8 values.

Generic Codegen. The generic code generator simplifies the target code generators by synthesizing code for frequently occurring patterns i.e. operator handling (generic operator template, transporting tuples i.e. `EMIT`), `LOOP`, buffering logic (buffer refill and flushing), variables and position allocators. This removes repetitive code from the target code generators, consequently, making them easier to engineer. Together with the target code generator, the generic code generator provides one set of translation rules.

CLite. Both, generic and target code generators, generate CLite code, our second domain-specific language. CLite is a simplified version of C without infix operators, macros, loops (go-to instead) i.e. VOILA can be lowered into a simpler language than C++ and C. It constitutes a lightweight abstraction that (a) provides a convenient interface to construct programs, (b) allows specific optimizations and (c) helps synthesizing different control-flow techniques. From the program in CLite, we generate a C++ program, but plan to compile to LLVM IR or machine code directly.

Synthesizing Control Flow. When synthesizing code, we differentiate between two control flow strategies: (a) simple goto-based programs and (b) finite state machines (FSMs). Both can easily be synthesized from CLite which can be seen in Figure 3: For a goto-based program, CLite blocks are lowered into a labels (`LABEL: BODY`) and branches into go-to statements (`goto NEXT;`). To generate a FSM, CLite blocks are lowered into a FSM state (`case STATE_ID: BODY`) and branches schedule the next states (`state.state = NEXT; break;`). The generated code is, then, wrapped into a loop and switch statement. To generate concurrent FSMs, we extend the FSM-code by wrapping local variables into a per-FSM state (`S`), and adding scheduling logic (`lwt`).

Minimizing State `S`. Especially the performance of concurrent FSMs is very sensitive to the number of variables stored in the per-FSM state `S`, as additional overheads occur when variables are accessed: (a) indirection overhead because, instead of in CPU registers, variables are stored inside an array of `struct` and (b) additional cache pressure with an increasing size of `S`. Therefore, we added an optimization pass, that promotes CLite variables to regular variables (can be stored in regular CPU registers). This is possible whenever variables are only read/written inside the same block, and, obviously, for constants. This optimization pass minimizes indirections as well as eases cache pressure.

5.3 Mixing Flavors

To finally generate an astronomical number of engines, the FUJI back-end should allow combining different flavors. Therefore, we extended VOILA with operations that allow changing the generated flavor, and extended FUJI with the ability to generate transitions between flavors.

One Flavor per Pipeline. One of the easiest ways of mixing flavors in a query is to choose a different flavor per pipeline. In FUJI, this is trivial because it just means instantiating a different code generator for each pipeline.

```
Fragment f;
Block l1(f), l2(f);
Builder b(l1);

Var x = f.var("int", "x");
Var y = f.var("int", "y");
Expr c = f.literal(42);
b.assign(y,
  b.func("+", b.ref(x), c));
b.effect(
  b.func("print", b.ref(y)));
b.branch(l2);
```
(a) Simple CLite example

```
int x,y;


{
  l1:
    y = x + 42;
    print(y);
    goto l2;
  l2:
    ...
}
```
(b) Goto Program

```
int state=1,x,y;

while (1) {

  switch (state) {
  case 1:
    y = x + 42;
    print(y);
    state = 2; break;
  case 2:
    ...
} }
```
(c) Finite State Machine

```
struct {int state=1,x,y;} S[N];
unsigned lwt = 0;
while (1) {
  auto& s = S[(lwt++) % N];
  switch (s.state) {
  case 1:
    s.y = s.x + 42;
    print(s.y);
    s.state = 2; break;
  case 2:
    ...
} }
```
(d) N Concurrent FSMs

Figure 3: Synthesizing Control-Flow from CLite.

Table 6: Highly diverse runtimes. SF 100. 24 threads.

#BLENDs	#Queries	Runtime (s)				
		Min	$Q_{0.25}$	Median	$Q_{0.75}$	Max
5	1	7.22	7.22	7.22	7.22	7.22
6	11	3.87	6.54	8.03	9.86	17.85
7	85	5.18	7.20	8.16	10.14	311.77
8	449	4.81	8.30	10.06	13.32	353.42
9	1511	4.70	9.05	10.98	15.51	318.32
10	9216	3.90	9.75	12.19	17.21	347.92
Total	11273	3.87	9.63	11.92	16.85	353.42

Mixing Flavors in a Pipeline. A more flexible method is to combine multiple flavors inside a pipeline what we call *blending*. We extended VOILA with `BLEND`, a statement which defines a flavor for a scope (sub-program with its statements and in-flight variables). Then, we can create different blends by setting a default/main flavor and introducing `BLEND`s which define flavors for program fragments. Note that this allows recursive stacking of `BLEND`s.

Translating `BLEND`. A `BLEND` defines a child flavor within a parent flavor. When translating a `BLEND`, we compose a new code generator (as described in Section 5.1) and buffer in-flight data during the transition from parent to child flavor, and back. Buffering can be done in many ways. One can imagine buffering columnar, row-wise buffers or mixes. To allow different buffering implementations, we construct an extremely versatile, yet simple, buffering interface: (1) `buffer_read_pos`/`buffer_write_pos` allocate slots for reading/writing, (2) `buffer_read`/`buffer_write` read/write data from/to the buffer, and (3) `buffer_read_commit`/`buffer_write_commit` commit used slots. For example, to write a row to the buffer, we first allocate a destination slot using `p = buffer_write_pos`. Then, we write each attribute/cell `a` to the slot `p` via `buffer_write(p, a)`. Afterwards, we complete the write via `buffer_write_commit(p)`. As this interface allows many possible buffer implementations, FUJI leaves specific implementation choices to the code generators.

Using the buffering interface, we implement `BLEND`. A `BLEND` introduces two buffers: an input and an output buffer. Data flows from a source into the input buffer. When the input buffer is full, we read values from the input buffer, and run the code inside `BLEND` (generated in a different flavor). This is producing output values which are then written into the output buffer. When the output buffer is full, we read values from the output buffer which then flow towards the sink. We use source and sink rather generically: In the trivial case, one `BLEND` inside a pipeline, the source refers to the VOILA code before the `BLEND` whereas the sink is the code after the `BLEND`. However, when nesting or chaining `BLEND`s, source/sink can as well read/write another `BLEND`'s buffer.

Buffering Design Choices. To minimize allocation overheads, we use fixed-sized ring buffers. Typically, when wrapping around, vectorized writes can become non-contiguous. In that case, we leave empty space at the end, wrap around and write contiguously. The buffer size impacts performance significantly. A buffer that is too small will be flushed too often, incurring branch miss-predictions. If the buffer is too large, additional cache misses can have a negative impact. We differentiate between the physical buffer size and a high watermark, the effective size. The buffer size ensures the writes fit, whereas the high watermark controls buffer performance. We use high watermark of `max(2*n, 2k)` and a size of `max(2*s*n, 64k)` with `n` being the input vector size (e.g. 8 for AVX-512) and `s` the number of concurrent FSMs.

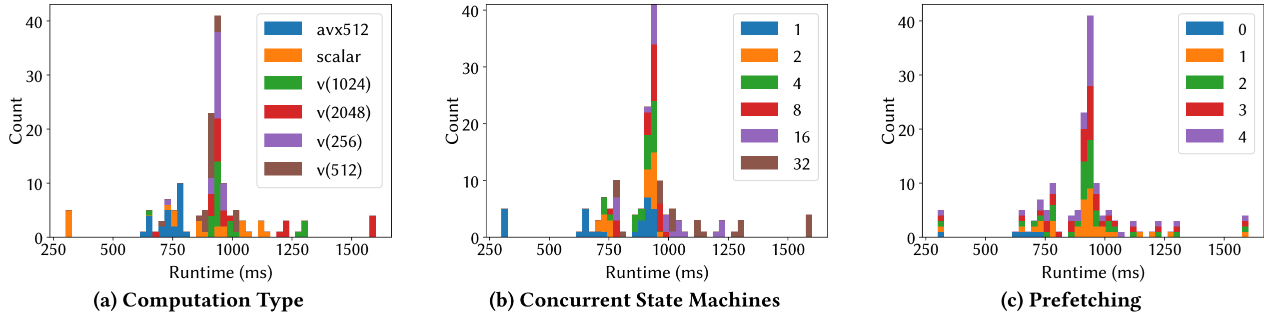
6 EVALUATION

We implemented the VOILA compiler with the two direct back-ends and FUJI in C++. All queries use the TPC-H dataset with varying scale factors (SF). The experiments were performed on a dual-socket Intel Xeon Gold 6126 with 24 SMT cores (12 physical cores) and 19.25 MB L3 cache each. The system is equipped with 384 GB of main memory.

6.1 Design Space Exploration

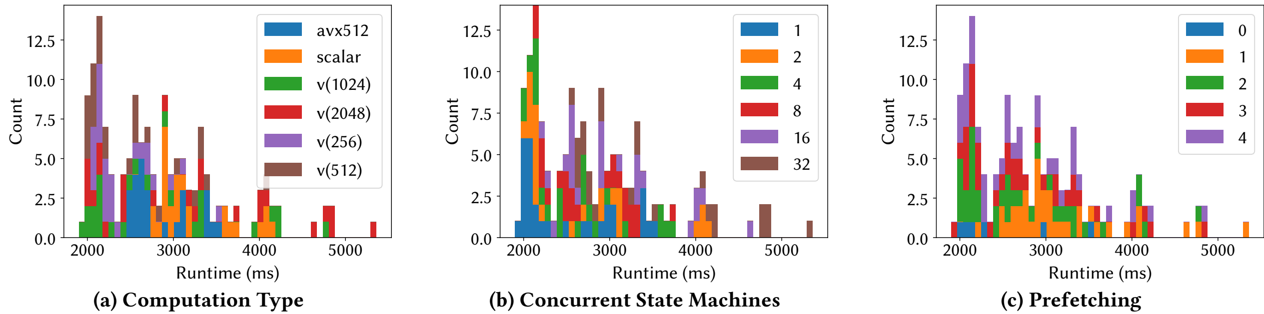
We explored the design space of TPC-H Q9 span through mixing different flavors per pipeline and `BLEND`ing different flavors inside the same pipeline. Instead of allowing fully flexible `BLEND` operations, we limited them to specific points: (a) hash join key check, (b) hash join payload gather, (c) projections/arithmetic and (d) filters. Further, we limited base flavors to the pipelines that contribute > 15% to total performance. We further restricted the space by restricting the essential parameters: computation type to the basic types (`scalar`, `avx512` and `vector`), prefetching to a boolean (`0`, `1`) and the number of concurrent state machines to reasonable small values (`1`, `2`, `4`, `8`). Since the design space of Q9 is too large for full exploration, we used sampling of the design space, in spirit of [15], as a robust way of exploring its performance diversity. We synthesized roughly 10,000 queries from VOILA using uniformly random combinations of base flavors (data-centric, prefetching, state machines etc.) as well as mixes of them, inside the same pipeline and between pipelines. This covers roughly $4 * 10^{-4}$% of the described space. The runtimes are summarized in Table 6.

Our uniform space sample frequently contains many `BLEND`s. Compared to the best runtime found (3.87), many queries perform worse ($\geq$ 2× higher median). There is a tail of runtimes > 4× slower than best time and extreme outliers that are ≈ 100× slower.



(a) Computation Type **(b) Concurrent State Machines** **(c) Prefetching**

Figure 4: Q1: Breakdown of *the same* histogram into computation type, prefetching and concurrent state machines. Many flavors are far from optimal. No benefit from prefetching. 24 threads, SF 100



(a) Computation Type **(b) Concurrent State Machines** **(c) Prefetching**

Figure 5: Q9: Breakdown of *the same* histogram into computation type, prefetching and concurrent state machines. Runtimes vary. `scalar` **flavors tend to lead to worse performance. 24 threads, SF 100**

With an increasing amount of `BLEND`s (mixes) the space increases exponentially. Further, there is a tendency that more `BLEND`s lead to higher runtime (increasing median, increasing 25- and 75-percentile $Q_{0.75}$). This can be explained by the increasing buffering effort per `BLEND`. Indicating that further methods to reduce buffering overhead are required. Besides that tendency, there are still positive outliers i.e. using 10 `BLEND` operations the minimum runtime is competitive to the best runtime using 5 mixes.

6.2 Impact of Components on Runtimes

Given a specific query, we investigate the impact of specific flavors and their components on the total runtime. Therefore, we generated roughly 150 base flavors (combinations of paradigms, prefetching and concurrent state machines) and ran the compiled query. The option we call prefetching encodes different prefetching localities as follows: `0` means no prefetching, `1` locality 0 (prefetch into all cache levels up to L1), `2` locality 1 (prefetch up to L2), `3` locality 2 (prefetch into L3), `4` non-temporal (short-term/evict soon) [2]. The precise meaning of the locality hints depends on the hardware. In particular, we analyze the impact of FUJI's components on two queries: TPC-H Q1 and Q9.

Q1. The results for Q1 are visualized in Figure 4. On first sight, the plot reveals that most flavors are sub-optimal, but outliers, positive as well as negative, exist. The best flavors are roughly 3× better than average. These are based on `scalar` processing and do not use concurrent state machines. There is no clear benefit of prefetching as Q1 fits into cache, but incurs no significant overhead either. Vectorized (`vector`) flavors tend to perform worse than `scalar` and `avx512` due to (a) materialization overhead (reading/writing vectors) and (b) in-efficient access to row-wise data when updating the aggregates. Further overhead is introduced by adding concurrent state machines, leading to the worst flavors being up to 50% worse than average. `avx512`'s performance is in between `scalar` and `vector`.

Q9 paints a different picture, as Figure 5 shows. In general, block-based flavors (`avx512`, `vector`) tend to outperform `scalar`. The best flavors are `vector`ized ones, with ≤ 8 concurrent state machines and prefetching. As opposed to Q1, we observed benefit from using concurrent state machines and prefetching. Similar to Q1, `avx512` tends to be in the middle between `vector` and `scalar`.

Summary. From both queries, we can see that neither flavor is optimal for all queries. The benefit from elaborate prefetching techniques (using concurrent state machines) on non-trivial queries appears to be rather limited and can even be detrimental to performance due to the overhead for small vector sizes (`scalar`, `avx512`).

6.3 VOILA vs. Hand-Optimized Code

We compare the runtime of VOILA-synthesized queries to state-of-the-art paradigms (a) data-centric compilation and (b) vectorized execution. As a baseline, we used the *hand-written* and *optimized* implementations by Kersten et al. [22]. Kersten et al. have shown that their implementations behave similar to the original systems Hyper and Vectorwise. We synthesized code for the basic data-centric and vectorized flavors (no prefetching, only one state-machine/goto-based) (a) `(scalar, 1, 0)`, (b) `(vector(1024), 1, 0)`, (c) using the direct hyper back-end and (d) using the Iterator-based vectorized back-end. Table 7 shows our results.

Besides the vectorized Q6, we observed similar performance over all queries in a range of ±30%.

Table 7: Competitive performance. Runtimes of flavors generated from VOILA are comparable with recent hand-written implementations. Runtimes in s.

Flavor	Q1	Q3	Q6	Q9
SF 10				
Typer [22]	0.5	1.1	0.2	3.1
Direct Hyper	0.6 (0.9×)	1.2 (0.9×)	0.3 (0.9×)	3.1 (1.0×)
FUJI Scalar	0.5 (1.1×)	1.2 (0.9×)	0.2 (1.3×)	3.1 (1.0×)
Tectorwise [22]	1.0	0.7	0.2	1.6
Direct Vector	1.0 (1.0×)	0.8 (0.8×)	0.2 (1.1×)	2.0 (0.8×)
FUJI Vector	1.0 (1.0×)	0.7 (0.9×)	0.3 (0.7×)	1.8 (0.9×)
SF 100				
Typer [22]	5.5	13.3	1.8	40.9
Direct Hyper	5.9 (0.9×)	12.2 (1.1×)	2.8 (0.6×)	31.9 (1.3×)
FUJI Scalar	6.0 (0.9×)	11.5 (1.2×)	1.8 (1.0×)	32.7 (1.2×)
Tectorwise [22]	9.2	8.0	1.7	21.2
Direct Vector	10.6 (0.9×)	7.6 (1.1×)	2.3 (0.7×)	17.7 (1.2×)
FUJI Vector	10.4 (0.9×)	6.7 (1.2×)	3.6 (0.5×)	16.0 (1.3×)

Vectorized Q6. Q6 is slower, because our synthesized code diverges from the Tectorwise implementation. Our implementation, first, computes all predicates and, afterwards, builds the selection vector from the conjunction of the predicates. For very selective queries, Q6 in particular, this introduces overhead for eliminated rows. Tectorwise builds a selection vector for every predicate and, therefore, can avoid this additional computational effort. Therefore, we modified the plan to build the selection vector for every predicate, similar to Tectorwise. With the modified plan, the FUJI-generated vectorized Q6, runs in 1.6s and performs roughly on par with Tectorwise (1.7s).

6.4 VOILA vs. State-of-the-Art Prefetching

The recent re-emergence of prefetching methods highlighted the importance of intelligent data structure access for overall query performance. In this experiment, we compare VOILA-synthesized queries to hand-optimized implementations such as Interleaved Multi-Vectorization (IMV) [13] and Relaxed Operator Fusion (ROF) [29].

The source code of IMV [12] revealed an already allocated perfectly sized hash table (size taken from a previous run). At runtime IMV, just inserts values into that hash table and builds bucket chains on-the-fly. Therefore, we implemented a flavor of IMV that can build a hash table of unknown size (*HyperBuild*). Our current implementation of VOILA lacks filter buffering and produces more intermediate states in concurrent state machines than strictly necessary. To enable an "apples-to-apples", in IMV, we disabled buffering (*NoBuffering*) and added 3 additional states (*Indirections*).As baseline we chose a FUJI flavor that resembles IMV: `(avx512, 8, 1)`. Both are using prefetching, multiple concurrent state machines and feature an implementation in AVX-512. We ran all queries single-threaded and used SF 10. Table 8 shows our results.

Compared to IMV, the VOILA-generated query achieves a similar performance. Once we remove certain factors that ensured a fair comparison (*Indirections*, *NoBuffering*, *HyperBuild*), IMV becomes up to 60% faster. We see this as an indication that future versions of VOILA should include (a) buffering and (b) further measures to minimize the number of states (in concurrent state machines). Compared to ROF, VOILA is roughly 20% slower. A crucial difference is that VOILA does not support buffering yet.

Table 8: VOILA can compete with hand-optimized prefetching, with further optimizations. Time in ms.

Name	Time	Speedup over `FUJI (avx512, 8, 1)`
`FUJI (avx512, 8, 1)`	1358	
Interleaved Multi-Vector. (IMV) [13]	1297	1.1 ×
-Indirections	912	1.5 ×
-HyperBuild	825	1.6 ×
-NoBuffering	800	1.7 ×
Relaxed Operator Fusion (ROF) [29]	1141	1.2 ×

6.5 VOILA vs. State-of-the-Art Open-Source

We compare VOILA to high-performance open-source systems: Weld [33], a domain-specific language for data analytics, DBLAB/LegoBase [1, 40], an elaborate query compiler, DuckDB [37], a vectorized in-memory DBMS, and MonetDB [20], an in-memory DBMS executing queries column-at-a-time. In the process, we had to make adjustments to the queries in Weld and LegoBase. Weld does not support group-by on strings – required for Q9 – therefore, we gave Weld the unfair advantage of using string dictionaries. We translated `n_name` into an integer and resolve the string at the end of the query. LegoBase allows many different query-specific optimizations, e.g. string compression and partitioning, that are not "TPC-H compliant" [40]. To enable a fair comparison, we used the TPC-H compliant settings proposed by Shaikhha et al. [40]. We compare the performance of queries generated by these systems to the best flavor synthesized from VOILA. Table 1 shows the results.

Queries generated from VOILA are up to 17.5× faster, without parallelism, and up to 35.5× faster, using all cores.

Single-threaded, VOILA-synthesized queries ran, across the board, 30% – 17.5× faster than DuckDB and LegoBase. Due to its early stage, DuckDB does not extensively focus on query performance, which explains its 4.3× – 9.5× slower performance on simple aggregation queries (Q1 & Q6). However, performance is the main focus of LegoBase compiler. LegoBase performs similarly (±30%) on simple aggregation queries. For complex join queries, LegoBase performs significantly worse (5.5× – 15.8× slower), than VOILA. LegoBase tends to produce sub-optimal code, partly caused by complex intermediate structures resulting from expressing a join (`dict[(key1, key2), list[val]]`) which are hard to remove [43].

Multi-threaded. All queries generated using Weld are 4.3 – 35.5× slower than queries generated from VOILA. Weld tends to perform better on simple aggregation queries and ran only 4.3× – 8.3× slower than VOILA. Weld performs worse on complex join queries, 12.3× – 35.5× slower than VOILA. Part of this overhead in Weld is caused by complex intermediate structures. Additionally, in Weld, it is impossible to express primary-foreign-key joins (joins that can only produce one or none match) which further exaggerates already existing inefficiencies. We noticed that, for Q1 and Q6, single-threaded Weld is substantially faster and performs roughly on par with VOILA. VOILA outperforms MonetDB, especially on simple queries by up to 28.8×. On complex join queries, MonetDB performs better, but VOILA is still 50% faster.

6.6 Engineering Aspects

We investigate the complexity of our code generation modules and compare development times to hand-writing queries.

Back-End Complexity. To understand the complexity of back-end modules, we measured the lines of code (LOC) excluding debug information, comments and empty lines.

Our compilation back-end modules are rather compact (600 – 1300 LOC). Direct back-ends tend to be simpler as they just concatenate strings (600 – 700 LOC). FUJI back-ends are more complex (600 – 1300 LOC) as (a) they include buffering logic for `BLEND` and (b) interact with other modules (FUJI front-end, FUJI generic back-end).

Compared to the code generator of Weld [33] which contains 20k LOC, our back-ends are 16 – 33× more compact. Compared to Hyper's code generator, which was 10k LOC in size [31], our data-centric back-end modules are 16.7× more compact. The smaller size of our back-end modules makes them rather easy to engineer.

Personal Experiences. In our personal experience, writing a direct back-end took roughly 5 days. Compared to 1-3 months expected to hand-written specific flavors for the queries, this is a 6 – 18× speedup! FUJI back-ends, without buffering, were roughly equivalent. Buffering, to enable `BLEND`ing, provided a small extra hurdle of roughly 1-2 extra days. Overall, most time was consumed by engineering a generic runtime framework.

7 RELATED WORK

First, we discuss related works that explore semantically equivalent programs. Afterwards, we discuss related languages and intermediate representations and, finally, relate to works on code generation, as well as query paradigms.

7.1 Exploring Equivalent Programs

Our approach of automatically generating semantically equivalent query implementations is related to super compilation [42]. Super compilation tries to explore *all* possibilities by modifying/mutating the program's instructions. Our approach is limited to the programs generatable from our domain-specific language and a specific back-end. Both, super compilation as well as our approach, have advantages and disadvantages. Super compilation will take a *extremely* long time to discover non-trivial structurally different programs, e.g. re-discovering vectorized execution [8] from a data-centric program [31]. Our approach goes into the opposite direction, we want to explore combinations of "good" points. This limits the search space and allows practical exploration of non-trivial programs.

Kersten et al.[22] studied the performance differences between the two state-of-the-art query execution paradigms: vectorized execution [8] and data-centric compilation [31]. As they did not have an elaborate synthesis framework, Kersten et al. had to engineer a basic framework and implement the required queries by hand.

Microbenchmark-based Approaches. Most works focus on exploring the very narrow design space of certain operations to (a) understand the space better and (b) find optima [3–6, 13, 23, 28, 30, 35, 36, 38, 39, 44]. These works contain substantial contributions. Arguably, the impact of certain optimizations on the whole query (or multiple queries) is significantly under-explored, a problem we address. Related to our synthesis framework is the Data Calculator by Idreos et al. [21], a tool able to predict the access cost of a data structure. The Data Calculator decomposes data structures, e.g. list, hash table, into specific primitives. During re-composition from these primitives, the Data Calculator can compute the performance of the final data structure. However, the effect of specific data structures on the holistic query performance is unclear, as it depends on context. For instance, vectorized hash lookups have better performance than data-centric lookups into large hash tables [22]. This interaction between execution flavor and data structure is ignored by the Data Calculator, whereas our framework focuses on exploring it.

7.2 Languages and Representations

We abstract implementation details using a domain-specific language. We classify related languages into categories: plans, comprehensions, vector models and imperative languages.

Plans are frequently used in data management systems. Most commonly, they either describe logical or physical execution strategies. Recent works using the concept of low-level plan operators (LOLEPOPs) [19, 27] break queries and operators into primitive operations and are conceptually very similar to VOILA. In a query engine, LOLEPOPs might not be so low-level (e.g. describe a hash join via `FindMatch` and `GatherPayload`), or require a more complex environment in order to function (program instead of directed acyclic graph). More high-level LOLEPOPs tend to lead to higher (re-)implementation effort as the operator needs to be implemented for every flavor. Very low-level LOLEPOPs would be similar to VOILA which requires state management/update. Needless to say,this also holds LOLEPOP-based representations such as Hawk [10].

Comprehensions describe enumerations as composition of scalar operations. Well-known classes are Monad [16] (e.g. Weld [33]) and Monoid [14] comprehensions. In general, comprehensions heavily rely on scalar operations and, therefore lose information about data-parallelism, e.g. branches are introduced. This requires re-discovery of data-parallelism , when e.g. SIMD or GPUs are supposed to be used. Like many high-level languages, Weld [33] allows the creation of temporary collections (arrays, lists, etc.) and, therefore, requires deforestation [43] to eliminate unnecessary intermediate data structures. Deforestation is a very hard optimization problem and not fully solvable in a reasonable time. VOILA avoids creating such intermediates through a more complex program.

Vector Models describe queries as the application of certain primitive functions onto vectors of data. Notable examples are MIL and VOODOO. MIL [9] (or now MAL) has been the foundation of query execution in MonetDB [20]. It defines operations in a column-at-a-time fashion. However, non-trivial plan operators, such as hash join or hash group-by, commonly translate to complex primitive expressions in MIL. For example, a join in MonetDB translates into a `JOIN` primitive. For design space exploration, this would require re-implementing many different joins. Instead, VOILA decomposes complex operators into sequences of statements and expressions, e.g. a hash join will end up as a sequence of hash table lookup, hash table insert, gather, etc. VOODOO [34] has no notation of hash tables (hash join, hash group-by) and "deliberately omits control-statements" [34], instead VOILA embraces both.

Low-level Imperative Languages typically break complex operations (e.g. hash join) into smaller very specific instructions. To allow fast execution (of generated) programs, their instructions tend to be close to the actual hardware. Due to their performance, low-level languages are frequently used as compilation target. Most

notably SystemR [11] generated assembly code, as well as HyPer [31] which generated LLVM IR [25]. Compared to LLVM IR [25], VOILA is much less low-level. For instance, VOILA supports multiple execution strategies (tuple-at-a-time, vector-at-a-time). LLVM's auto-vectorization could come somewhat "close". However, not all algorithms are vectorizable, e.g. a selection might introduce a branch and, hence, breaking possible auto-vectorization for the whole operator/pipeline. Similar low-level languages to LLVM IR are assembly or C. These require re-discovering data-parallelism via auto-vectorization to have VOILA-like functionality.

7.3 Code Generation/Compilation

Code generation and compilation are well studied topics. Typically, compilers shrink the search space by choosing specific implementations, usually cost-based optima. Our approach does the opposite, we expand the search space, hoping to explore many new points.

Shaikhha et al. [40] propose a stack of domain-specific languages, namely QPlan, ScaLite[Map,List], ScaLite[List], ScaLite and C. A query is described in QPlan and then lowered through the stack until C is reached. There is some similarity to our approach: from query to VOILA to CLite to C++, which exists mainly because stacking languages simplifies each language/compiler layer. ScaLite and VOILA might seem similar but they are *not*. The main difference lies in the semantics, ScaLite relies on sequential evaluation and, hence removes data-parallelism, whereas VOILA embraces data-parallelism. Compared to ScaLite, VOILA has certain advantages: (a) In its core VOILA is a simpler language as there is no need for complex statements such as `if/case`. (b) VOILA only operates on flat data structures (intermediate arrays or (hash) tables), as opposed to allowing nested data structures. Consequently, this minimizes the time spent on removing complex nested data structures at compile-time i.e. reduce deforestation overhead [43]. However, these advantages do not come for free. One disadvantage of VOILA is its reliance on very specialized primitive operations.

Tahboub et al. [41] argue that such a language stack can be flattened into one language using partial evaluation. . Our program synthesis from VOILA is similar: the direct back-ends directly generate C++ code from VOILA. FUJI back-ends, however, translate into our C-like language CLite and from there into C++. The translation from CLite to C++ is very lightweight and straight-forward. However, the extra step – translating into CLite – allows more flexibility, e.g. one could generate LLVM IR [25] instead of C code.

7.4 Query Paradigms

For analytical queries, many basic execution paradigms have been proposed. VOILA synthesizes the two most established state-of-the-art paradigms, data-centric [31] and vectorized [8]. Hence, we focus on the more elaborate paradigms.

Relaxed Operator Fusion (ROF) [29] aims at combining vectorized execution and data-centric compilation while improving the final result using prefetching. We, too, use buffering when mixing multiple flavors. ROF buffers on the operator-level whereas our approach allows much more fine-grained buffering on the language-level. Our approach is more generic, but comes with a higher overhead as the current context (alive variables) needs to be preserved instead of, in case of ROF, only buffering an operator's output.

Interleaved Multi-Vectorizing (IMV) [13] describes a method for efficient prefetching during particular operations (hash join, tree lookup etc.). In essence, IMV interleaves prefetching with other operations through state machines, one per "lightweight thread". In a hash join, these "other operations" are buffering, to keep all SIMD lanes busy, and key checking i.e. while prefetches are running (in the background), less memory-heavy operations can proceed. Further, IMV fully relies on AVX-512 intrinsics (`compress/expand`) to achieve fast buffering. Unfortunately, IMV[13] exists only as a single hand-coded hash join query; the authors deferred developing IMV for more complex queries (e.g. TPC-H Q9) and in a generic way, to future work. We showcase, with our VOILA-generated queries, an approach to broaden IMV to non-trivial queries. However, a complete and fast IMV implementation, would require hand-optimized kernels (using AVX-512) for buffering, as well as, the whole query.

8 CONCLUSIONS & FUTURE WORK

We think the design space of query execution is under-explored because it is (a) extremely large, (b) tedious to explore and (c) features a low chance of success. Our approach is a first step into the exploration by automatically synthesizing query engines. Automating this process is much faster than implementing prototypes by hand, as it reduces months of development time to seconds.

Our framework synthesizes engines from descriptions in our prudently designed domain-specific language VOILA. Programs in VOILA describe algorithmic details (of operators) while hiding physical implementation details. Many state-of-the-art languages and compilers do not achieve performance on par with "good" implementations of state-of-the-art paradigms. But, VOILA does! Our code generation back-ends are up to 33× smaller than comparable code generators, outperform competing analytical systems, and approach the performance of hand-written implementations.

An important contribution is our success in generating the state-of-the-art paradigms vectorized execution [8] and data-centric compilation [31] from a single algorithmic description. VOILA thus is flexible enough to capture two opposite ends in the design space.

With our novel component-based code generator (FUJI), we further open the way towards generating new and novel execution paradigms. We currently can generate thousands of different execution tactics by adding in SIMD, prefetching and *blending* different paradigms, where the execution tactic can change within a single pipeline. This way of generating many "new" paradigms can be criticized as rehashing existing ideas, so we hope that our work on automatic engine generation in general, and VOILA specifically, will encourage follow-on work by the community in creative new directions. This can include new query operators, new operator algorithms, data structures, but also the creation of back-ends that target non-CPU hardware (GPU, FPGA). Furthermore, VOILA can be also used as the foundation for highly flexible and efficient database engines. Such a next generation engine could operate as a virtual machine (VM), using VOILA as its instruction set. During query execution, the VM can start interpreting code fragments using highly efficient vectorized interpretation. Later, the VM can generate optimized code for expensive fragments, tailored specifically to the current workload and hardware [17].

REFERENCES

[1] [n.d.]. https://github.com/epfldata/dblab.
[2] 2017. What are _mm_prefetch() locality hints? https://stackoverflow.com/questions/46521694/what-are-mm-prefetch-locality-hints. Accessed: 2021-01-28.
[3] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *PVLDB* 5, 10 (2012), 1064–1075.
[4] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *PVLDB* 7, 1 (2013), 85–96.
[5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. 362–373.
[6] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam S. Lightstone, and David Sharpe. 2014. Memory-Efficient Hash Joins. *PVLDB* 8, 4 (2014), 353–364.
[7] Peter Boncz. 2002. *Monet: A next-generation DBMS kernel for query-intensive applications*. Universiteit van Amsterdam.
[8] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*. 225–237.
[9] Peter A. Boncz and Martin L. Kersten. 1999. MIL Primitives for Querying a Fragmented World. *VLDB Journal* 8, 2 (1999), 101–119.
[10] 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 Journal* 27, 6 (2018), 797–822.
[11] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, Jim N. Gray, William-Frank. King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A History and Evaluation of System R. *Commun. ACM* (1981), 632–646.
[12] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. [n.d.]. https://github.com/fzhedu/db-imv/blob/master/src/imv/engine.cpp, line 226. Accessed: 2021-01-28.
[13] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. 2019. Interleaved Multi-vectorizing. *PVLDB* 13, 3 (2019), 226–238.
[14] Leonidas Fegaras. 2016. An Algebra for Distributed Big Data Analytics. (2016).
[15] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. 1994. Fast, Randomized Join-Order Selection - Why Use Transformations?. In *VLDB'94*. 85–95.
[16] Torsten Grust. 2004. Monad comprehensions: a versatile representation for queries. In *The Functional Approach to Data Management*. Springer, 288–311.
[17] Tim Gubner. 2018. Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware. In *ICDE*.
[18] Tim Gubner and Peter Boncz. 2017. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. In *ADMS*.
[19] Laura M. Haas, Wendy Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. 1990. Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowl. and Data Eng.* (1990), 143–160.
[20] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* (2012).
[21] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *SIGMOD*. 535–550.
[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB* (2018), 2209–2222.
[23] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *PVLDB* 2, 2 (2009), 1378–1389.
[24] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *PVLDB* 9, 4 (2015), 252–263.
[25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. Palo Alto, California.
[26] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*. 743–754.
[27] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. *SIGMOD Rec.* (1988), 18–27.
[28] Stefan Manegold, Peter Boncz, and Martin Kersten. 2000. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *PVLDB*. 339–350.
[29] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *PVLDB* 11, 1 (2017), 1–13.
[30] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD*. 1123–1136.
[31] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* (2011), 539–550.
[32] Michal Nowakiewicz, Eric Boutin, Eric Hanson, Robert Walzer, and Akash Katipally. 2018. BIPie: Fast Selection and Aggregation on Encoded Data Using Operator Specialization. In *SIGMOD*. 1447–1459.
[33] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *CIDR '17*.
[34] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB* (2016), 1707–1718.
[35] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. 1493–1508.
[36] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB* 11, 2 (2017), 230–242.
[37] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*. 1981–1984.
[38] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *PVLDB* 9, 3 (2015), 96–107.
[39] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*. 1961–1976.
[40] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*. 1907–1922.
[41] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. 307–322.
[42] Valentin F. Turchin. 1986. The Concept of a Supercompiler. *ACM TPLS* 8, 3 (1986), 292–325.
[43] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP*. 231–248.
[44] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*. 145–156.