Getting Swole: Generating Access-Aware Code with Predicate Pullups

Andrew Crotty
Brown University
crottyan@cs.brown.edu

Alex Galakatos
Brown University
agg@cs.brown.edu

Tim Kraska
MIT CSAIL
kraska@mit.edu

***Abstract*—Code generation for in-memory query processing is now commonplace. While existing approaches use a wide range of techniques (e.g., inline expansion, pipelining, SIMD vectorization, prefetching) to reduce processing effort, we argue that generating code with better data access patterns is often more important. Therefore, we propose SWOLE, the first *access-aware* code generation strategy. Contradictory to the conventional wisdom, SWOLE heavily leverages predicate pullups to produce code with better access patterns, which outweighs the overhead of performing wasted work. Our experiments show that SWOLE can outperform the state-of-the-art approach by over $2.6\times$.**

I. Introduction

Code generation has become a popular approach for accelerating in-memory query processing. The main benefit of code generation is the ability to eliminate high-overhead abstractions (e.g., Volcano-style iterators [1]) and apply low-level optimizations, including inline expansion [2], pipelining [3], SIMD vectorization [4], and prefetching [5].

However, we argue that improving data access patterns is often more important than simply reducing the amount of work performed by the CPU. While others have previously examined the impact of access patterns on in-memory OLAP queries [6], [7], [8], [9], we are the first to propose a code generation strategy that explicitly optimizes for access patterns rather than CPU work. In fact, as we will illustrate in Section II, several existing strategies all produce code that exhibits essentially the same poor access patterns.

Our novel *access-aware* code generation strategy, called SWOLE, makes extensive use of predicate pullups, which defer filtering until after other operations. Although performing potentially wasted work seems counterintuitive, this approach can significantly outperform traditional predicate pushdowns due to improved access patterns.

In summary, we make the following contributions:

- We propose SWOLE, the first access-aware code generation strategy that reasons about and optimizes for data access patterns rather than minimizing CPU work.
- We introduce several techniques based on the idea of predicate pullups to improve the access patterns of the most common OLAP query operators, as well as cost models to decide when these techniques are beneficial.
- We evaluate SWOLE using TPC-H [10] and a series of microbenchmarks, and our results show an improvement of more than $2.6\times$ over the state-of-the-art approach.

II. Background & Related Work

Using code generation for query processing is not a recent idea. In fact, System R [11] originally compiled SQL statements directly to machine code by stitching together code fragments from a fragment library [12]. While beneficial for avoiding the overhead associated with interpreted query execution, this approach was eventually abandoned due to issues with software maintenance and debuggability [13], as well as poor portability [5].

Over two decades later, Daytona [14] pioneered the translation of queries into C programs, which could then be compiled into an executable and linked with the necessary libraries. Compared to machine code, generating relatively high-level C code avoided many of the main issues encountered with System R.

Similarly, JAMDB [13] translated queries to Java code while applying operator type specialization and aggressive expression inlining. Unlike the statically compiled C programs produced by Daytona, though, JAMDB sought to leverage the capabilities of the Java Virtual Machine to dynamically apply low-level optimizations at runtime.

HIQUE [2] used template-based code generation to compile queries into C programs with explicit data staging. Unlike past attempts at code generation, HIQUE's approach removed the overheads associated with traditional iterator-based execution, which emerged as a key bottleneck for in-memory query processing.

Some recent systems that have utilized code generation include HyPer [15], [3], Tenzing [16], Hekaton [17], [18], LegoBase [19], Impala [20], [21], Tupleware [22], [4], MemSQL [23], and Peloton [24], [5].

On the other hand, a number of studies [25], [26], [27] have compared the performance of code generation to vectorized query processing, which operates on vectors rather than individual tuples. The idea originated as part of the MonetDB/X100 project [28], which would eventually be commercialized as Vectorwise [29] (currently known as Actian Vector [30]). These studies largely concluded that neither approach is dominant and advocate instead for a combination of the two in order to achieve the best performance.

In the remainder of this section, we discuss in detail three of the most prominent existing code generation strategies for executing OLAP queries on column-oriented storage formats and then explain the importance of data access patterns.

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00114

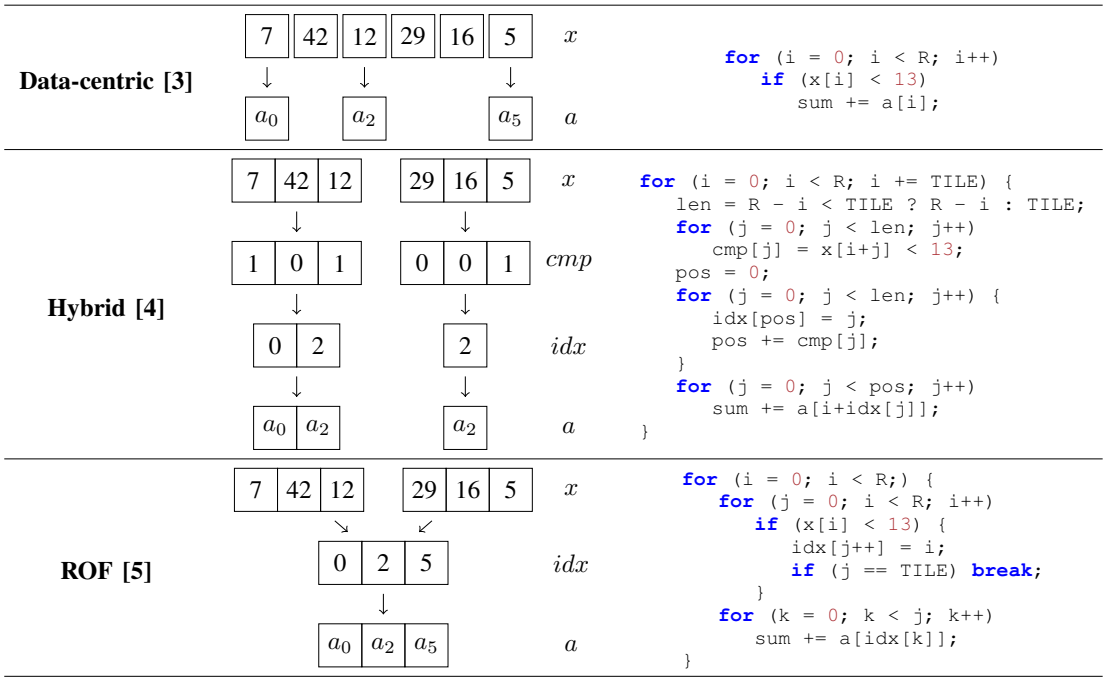


Fig. 1: Existing Code Generation Strategies

A. Existing Strategies

To illustrate each strategy, Figure 1 shows both a pictorial representation and the corresponding pseudocode fragment for the following simple example query:

```
select sum(a)
from R
where x < 13
```

1) Data-centric: Developed as the query execution framework for HyPer [15], the key idea of data-centric [3] code generation is to use a push-based, pipelined query processing model that attempts to maximize data locality, with data items remaining in CPU registers as long as possible. LLVM glue code is generated to stitch together precompiled operators, and then the query is JIT-compiled.

As shown in Figure 1, the data-centric strategy is the most straightforward approach that we consider, consisting of a single `for` loop over each tuple in `R` with a conditional `if` statement to evaluate the predicate. For tuples that pass the predicate, the corresponding value of the `a` attribute is added to `sum`.

This approach achieves excellent data locality but suffers from two main drawbacks: (1) the control dependency introduced by `if` statements typically precludes SIMD vectorization; and (2) `if` statements exhibit generally poor performance for predicates with intermediate selectivities due to frequent CPU branch mispredictions [31]. Later versions of HyPer incorporated Data Blocks [32] to enable SIMD processing on cold data, but the two code generation strategies we consider next provide different alternatives that mitigate the issues associated with the data-centric approach.

2) Hybrid: Tupleware [22] proposed a code generation strategy that is a hybrid between data-centric and vectorized processing [4]. This approach generates code that minimizes intermediate materialization while maximizing SIMD processing opportunities by fusing or splitting query operators.

The hybrid strategy naturally produces more complex code than the data-centric approach, as shown in Figure 1. First, note that the outer `for` loop over `R` is now broken up into three separate inner loops, each of which operates on `len` tuples at a time. In most cases, `len` is of size `TILE`, except for the remainder tuples when `R` is not divisible by `TILE`.

The first inner loop evaluates the predicate and stores the result (i.e., 0 or 1) in the `cmp` array, which is called the prepass technique [4]. Unlike the data-centric version, the hybrid strategy's prepass technique enables SIMD vectorization for predicate evaluation by removing the control dependency.

The second inner loop iterates over `cmp` to populate the `idx` array, which is a selection vector [28] containing the indexes of tuples that passed the predicate. In Figure 1, we show the no-branch [31] (i.e., predicated) version, which replaces the control dependency with a data dependency in order to avoid expensive branch mispredictions, although a branching implementation may be superior for some predicate selectivities. Finally, the third inner loop updates `sum` based on the tuple indexes stored in `idx`.

Again, by generating three separate inner loops, the hybrid strategy can leverage SIMD processing for predicate evaluation using the prepass technique while also benefiting from the ability to skip tuples that did not pass the predicate using the `idx` selection vector.

Section	Technique	Operators	Heuristics
III-A	Value Masking	All	Memory-Bound, Small Hash Tables
III-B	Key Masking	Group-By Aggregation, Join, Groupjoin	Complex Aggregation, Large Hash Tables
III-C	Access Merging	All	Always Better
III-D	Positional Bitmaps	Join, Semijoin	Always Better
III-E	Eager Aggregation	Join, Groupjoin	Low-Cardinality Group-By Keys

Fig. 2: Summary of SWOLE Techniques

3) ROF: Like Tupleware, the relaxed operator fusion [5] (ROF) strategy used by Peloton [24] blends data-centric code generation and vectorization through the materialization of intermediate results at staging points during query execution. Compared to Tupleware, ROF has three primary differences: (1) always operating on full intermediate result selection vectors; (2) using the Data Blocks [32] technique of a pre-computed lookup table to enable SIMD for selection operators; and (3) embedding explicit prefetch instructions to mask data request latencies, in particular for operators that need to access a hash table (e.g., joins, group-by aggregations).

The first and most important difference is depicted visually in Figure 1: whereas the hybrid strategy produces two partially full `idx` arrays, ROF fills a single `idx` array before moving on to the aggregation. The control flow to achieve this is now much more complicated than the comparatively straightforward tiling of the hybrid approach. Essentially, the first inner loop is responsible for filling up the `idx` array with the indexes of tuples that pass the predicate. Then, when `idx` is full (or no tuples remain), this loop breaks.

The second inner loop performs the aggregation and works almost identically to the previously explained hybrid version. However, since the predicate evaluation loop (almost always) fills the `idx` array, the ROF aggregation loop will (almost always) perform a fixed number of iterations. The hybrid aggregation loop, in contrast, will perform a variable number of iterations based on how many tuples pass the predicate.

For simplicity, we do not depict the SIMD implementation of the predicate evaluation in the pseudocode. Like the prepass technique, though, this optimization can provide substantial speedups over the data-centric strategy for predicates that are complex or select few tuples. Finally, since this simple aggregation query does not require a hash table, no explicit prefetching is necessary.

B. Access Patterns

As shown in Figure 1, the strategies we described have considerable variability in the structure of their generated code. Surprisingly, however, they all have one thing in common: the same data access patterns.

Specifically, all of these strategies produce code exhibiting a *Sequential Traversal with Conditional Reads* [7] (`s_trav_cr`) access pattern. In terms of the example query, the `s_trav_cr` pattern corresponds to sequential accesses of `x` for predicate evaluation followed by conditional accesses of `a` to update `sum` for selected tuples. The conditional access appears in the data-centric version as an `if` statement, whereas the hybrid and ROF versions utilize a selection vector (i.e., `idx`) to skip filtered tuples.

The `s_trav_cr` access pattern stems from the predicate pushdown, which has long existed as a fundamental query optimization heuristic. The goal of a predicate pushdown is to filter tuples as early as possible during query processing in order to minimize work performed by later operators.

However, we argue that a predicate pullup, which does not perform any early filtering, will actually produce better access patterns than predicate pushdowns at the cost of some potentially wasted work. As we demonstrate experimentally (Section IV), these improved access patterns outweigh the overhead of wasted work in many cases.

While predicate pullups have been previously explored for specific uses (e.g., delaying evaluation of expensive predicates [33], sharing work among continuous queries [34], leveraging specialized hardware [35]), we show how they can be utilized to improve the access patterns of generated code. The result is a set of techniques that we synthesize into the first access-aware code generation strategy, called SWOLE, which we discuss in the following section.

III. SWOLE

This section describes the techniques that comprise SWOLE and explains their applicability to some of the most common OLAP operators. A summary of these techniques appears in Figure 2, including basic heuristics to give a high-level intuition about when each is beneficial.

A. Value Masking

Consider again the example query from the previous section. As shown in Figure 1, the existing strategies all use some form of predicate pushdown (i.e., conditional branching or selection vectors) to filter tuples before performing the aggregation, thus resulting in the `s_trav_cr` access pattern.

Instead, we propose an alternative based on predicate pullups called *value masking*, which is shown in Figure 3. At a high level, we evaluate the predicate and use the result to mask non-qualifying values to 0 rather than immediately filtering them, similar to the idea behind SIMD implementations of certain aggregation functions [36]. Note that, although we primarily discuss the `sum` function in this paper, others (e.g., `min`, `max`) may require minor additional bookkeeping.

The pseudocode for this approach resembles the hybrid strategy, with the first inner loop storing the predicate evaluation result in `cmp`. However, instead of using a selection vector to perform early filtering, the value masking approach

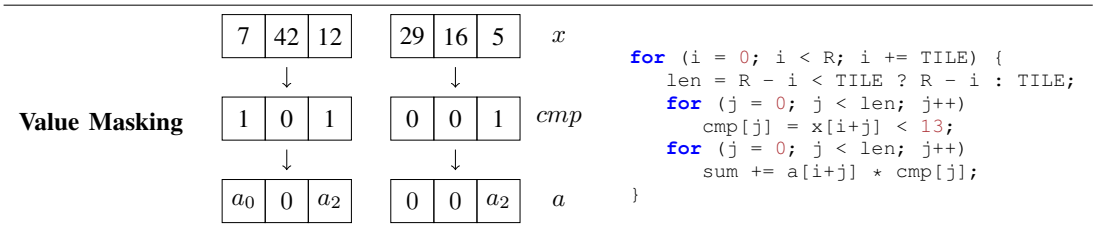


Fig. 3: Simple Aggregation

unconditionally accesses the values from `a` and then multiplies each by the corresponding predicate result stored in `cmp` (i.e., either 0 or 1) before finally updating `sum`.

While the implementation of value masking may be less intuitive than some of the other strategies we have discussed, the benefit is straightforward: we have replaced the conditional access of `a` with a sequential access. Our experimental evaluation (Section IV) demonstrates that this seemingly minor change can have a substantial impact on performance.

However, since the value masking technique will usually perform some degree of wasted work (i.e., processing tuples that are subsequently masked), predicate selectivity plays an important role in deciding if it is beneficial. For example, if a predicate selects very few tuples, the `cmp` array shown in Figure 3 will contain mostly 0 values, and the subsequent aggregation step will add primarily masked `a` values (i.e., 0) to `sum`. In many cases, the benefits of the improved access patterns will outweigh the costs of performing this wasted work, but some cases (e.g., expensive aggregations) may make value masking infeasible.

Therefore, in cases where value masking is not beneficial, we can simply fall back to generating code using the hybrid strategy. To choose between these two alternatives, we developed a cost model for each strategy. The cost of the hybrid version is:

$$Hybrid = R \cdot (\underbrace{read_{seq}}_{\text{Selection}} + \underbrace{\sigma_R \cdot \max(comp, read_{cond})}_{\text{Aggregation}})$$

where the number of tuples in R is multiplied by (1) the cost of the sequential read $read_{seq}$ needed to perform the selection plus (2) the cost of performing the aggregation $\max(comp, read_{cond})$ for every tuple that passes the predicate with selectivity σ_R. Note that if the aggregation is compute-bound, the model will use the cost $comp$ (in cycles) of that computation, which can be estimated through introspection [4]. Otherwise, if the aggregation is memory-bound, the model will use the cost of a conditional access [7].

On the other hand, the cost model for the value masking (VM) approach is:

$$VM = R \cdot (\underbrace{read_{seq}}_{\text{Selection}} + \underbrace{\max(comp, read_{seq})}_{\text{Aggregation}})$$

where the conditional read from the hybrid cost model has been replaced by the cost of a sequential read $read_{seq}$ and the selectivity term σ_R has been removed.

Essentially, if the aggregation is compute-bound, the hybrid approach is superior, since the selection vectors will perform early filtering. If the aggregation is memory-bound, though, the wasted work is masked by the data access, and predicate pullups are beneficial. In Section IV, we show cases where each of these strategies is preferable.

Finally, we note that the value masking technique can benefit a broad range of query operators. One interesting example is SQL's `CASE` statement, which would normally entail a series of branching `if-else` expressions. In many instances, though, we can unconditionally evaluate all cases and then mask the non-qualifying results. While this approach avoids the poor access patterns associated with conditional branching, unconditionally evaluating complex (or too many) cases can again become prohibitively expensive, and we must apply the cost model to see if this optimization is beneficial.

B. Key Masking

The example query from Section II produces a single aggregate value. On the other hand, a group-by aggregation query produces multiple aggregate values grouped by the specified key. Consider the following simple extension to the example:

```
select c, sum(a)
from R
where x < 13
group by c
```

The existing strategies would execute this query similarly to the previous one, first filtering tuples that do not satisfy the predicate before performing a lookup in a hash table to update the sum for the corresponding group-by key. Although straightforward, this approach still has the same poor access patterns (i.e., sequential reads of `x` followed by conditional reads of both `c` and `a`).

As an alternative, we can extend our value masking technique to work for group-by aggregation. Figure 4 (top) shows a value masking version of the example group-by query. After evaluating and storing the result of the predicate in the first inner loop, the second inner loop unconditionally accesses the group-by key `c` and performs a lookup in the hash table. As in the prior example, non-qualifying values are then masked using the `cmp` array before updating `sum`.

Similarly, the original cost model for value masking (VM) needs to be extended as follows:

$$VM = R \cdot (read_{seq} + \max(comp, read_{seq}, ht_{lookup}))$$

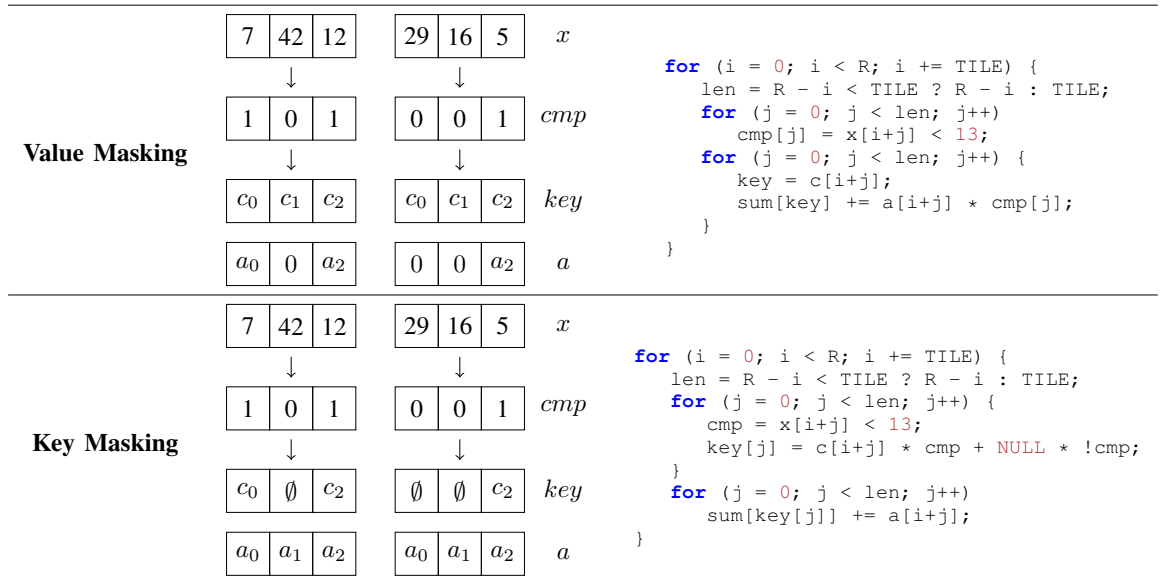


Fig. 4: Group-By Aggregation

where the cost [7] of performing a hash table lookup ht_{lookup} has been added to the max function, since it can be interleaved with the other parts of the aggregation.

As in the previous example, the value masking version replaces the conditional reads with a sequential access pattern for both `c` and `a`. However, with the addition of ht_{lookup}, unconditionally looking up a key in a large hash table can become expensive, which would dominate the cost and far outweigh the benefits of the improved access patterns. We must also perform an extra bookkeeping step by setting a flag during insertion to differentiate between masked entries and actual 0 values.

Another option is to mask the group-by key rather than the value, thereby mapping keys not selected by the predicate to a single throwaway entry in the hash table. This *key masking* technique mitigates the aforementioned problems with unconditional lookups in large hash tables because the throwaway entry will most likely be cached from frequent accesses in cases where the predicate selects relatively few tuples. Moreover, the extra bookkeeping step is no longer necessary, since all entries other than the throwaway are guaranteed to be valid.

Figure 4 (bottom) shows the key masking approach. In the first inner loop, the predicate result is now used to directly mask the values of `c` that we store in the `key` array. More specifically, if the predicate evaluates to true, then the value of `c` will be stored as the key at position `j`; otherwise, the `NULL` key will be stored at that position, which maps to the throwaway entry in the hash table. The second inner loop then performs the aggregation for every key, updating the appropriate entry for selected tuples and the throwaway for filtered ones. Again, note that the value of `a` is not masked in this version, since the masking is performed on the group-by key in the first inner loop.

The cost model for the key masking (KM) version is the same as value masking but with one additional term:

$$KM = R \cdot (read_{seq} + \sigma_R \cdot \max(comp, read_{seq}, ht_{lookup}) + \underbrace{(1-\sigma_R) \cdot \max(comp, read_{seq}, ht_{null})}_{\text{Masked Aggregation}})$$

where σ_R is the percentage of valid aggregations for tuples that pass the predicate and $(1 - \sigma_R)$ is the percentage of masked aggregations for tuples that do not. In particular, notice that the term for the masked aggregation includes the cost of accessing the hash table entry for the null key ht_{null}, which should be cheaper than a regular hash table lookup ht_{lookup} due to caching in cases where the predicate often evaluates to false. Intuitively, then, the key masking approach will be better than masking the value when the cost of an unconditional lookup is high, such as when the hash table is large. We experimentally validate this intuition in Section IV.

Finally, although we have focused on group-by aggregation, we can apply the key masking technique to any operator that uses a hash table, including joins. In this sense, key masking is a generalization of the predicated lookup technique proposed by Voodoo [37], which narrowly targeted queries containing an indexed join. Moreover, we show in Section IV that the key masking approach is not a dominant strategy, as suggested by Voodoo.

C. Access Merging

Often, queries will repeat the same expressions multiple times, such as in a complex aggregation (e.g., TPC-H Q1). Common subexpression elimination is a well-known technique for removing redundant computations in these situations by materializing intermediate values in temporary variables.

However, another type of redundancy can occur when the same attribute is referenced in several different expressions

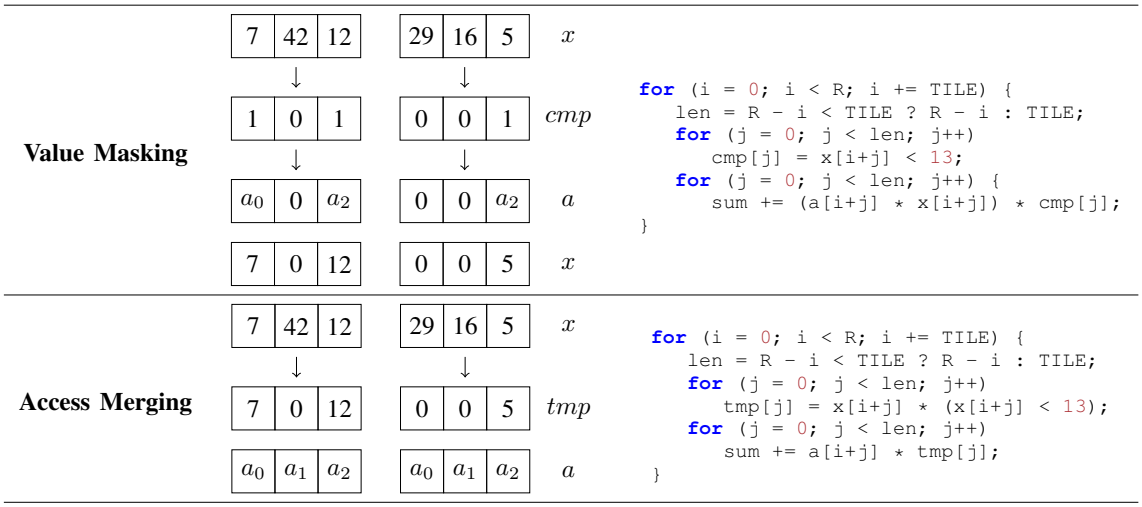


Fig. 5: Repeated References

throughout a query, such as in a predicate as well as an aggregation (e.g., TPC-H Q6). As a concrete example, consider the following query:

```
select sum(a * x)
from R
where x < 13
```

Like the previous examples, the existing approaches would execute this query by first evaluating the predicate, which requires a sequential access of `x`, followed by a conditional access of `a` and a second conditional access of `x` in order to update `sum`. As shown in Figure 5 (top), our value masking technique can improve the access patterns by replacing conditional accesses with sequential ones, but `x` is still accessed twice (i.e., once for the selection and then again for the aggregation).

We therefore propose an *access merging* technique that removes redundant data accesses by fusing multiple expressions that reference the same attribute into a single expression. The generated code for the access merging version of this query is shown in Figure 5 (bottom). Rather than storing the outcome of the predicate in the `cmp` array in the first inner loop, the predicate result is immediately multiplied with the actual value of `x`, which is stored in an intermediate `tmp` array. Then, in the second inner loop, the values in `tmp` are multiplied directly with `a`, thereby resulting in only a single access of `x`. This access merging approach can be seen as a form of operator fusion and is always beneficial if it can be applied, since it results in fewer total accesses.

D. Positional Bitmaps

A semijoin is a special case of an equijoin where attributes from the build side do not appear beyond the join, as in the following query:

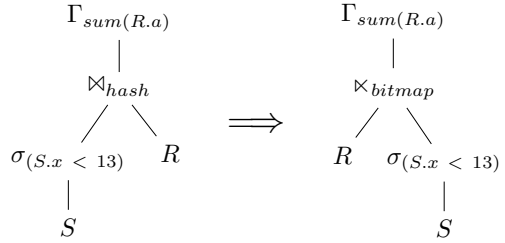
```
select sum(R.a)
from R, S
where R.fk = S.pk
  and S.x < 13
```

The existing strategies would execute this query as expected. First, in the build phase, a scan of `S` constructs a hash table containing `S.pk` for all tuples that pass the predicate. Then, in the probe phase, a scan of `R` performs a lookup in the hash table before updating `sum` if a match is found. The resulting access patterns include random accesses on both sides of the join for inserts and lookups.

Instead, we propose a data structure called a *positional bitmap*, which is based on an old technique [38] that has recently been rediscovered [39]. Positional bitmaps exploit the referential integrity constraint of foreign keys, which is typically enforced by building an index to check the corresponding primary key. Thus, since these indexes are necessary, our technique does not incur any additional overhead.

The algorithm to perform a semijoin using a positional bitmap works as follows. First, we perform a sequential scan over the build side of the join, evaluating the predicate to decide whether each tuple should be included in the result. Depending on the estimate of the value masking cost model, we either: (1) unconditionally set the corresponding bit at the tuple offset in the bitmap to the value of the predicate result; or (2) use a selection vector to set bits to 1 for only the tuples that pass the predicate. Therefore, we replace the random accesses required to insert keys that pass the predicate into a hash table with sequential accesses to the bitmap. On the probe side, we can then simply perform a positional lookup into the bitmap using the offset from the foreign key index to check for the inclusion of the corresponding primary key.

Applying this optimization to the example query yields:



The original version (left) matches our previous description of the existing strategies, which probe a hash table for every tuple in `R` to find a match from `S` before updating `sum`. On the other hand, the bitmap version (right) builds a bitmap for qualifying tuples from `S` and then checks the bit for each tuple in `R` using the foreign key index.

One concern might be that positional bitmaps will exhibit the same cache miss problems that would occur with a large hash table in a traditional join. Surprisingly, though, even for relatively large tables, the corresponding positional bitmap is still a very manageable size. For example, a table with 100M tuples requires only about 12.5MB of space, which fits easily within most caches. If size does become an issue, we can always compress the bitmap, either by replacing entire blocks of repeated values or through more advanced techniques [40], [41], but the benefits in size reduction would need to be weighed against the increased access overhead.

Finally, while we have focused on the semijoin operator, positional bitmaps can also be applied to regular joins using techniques similar to late materialization [42].

E. Eager Aggregation

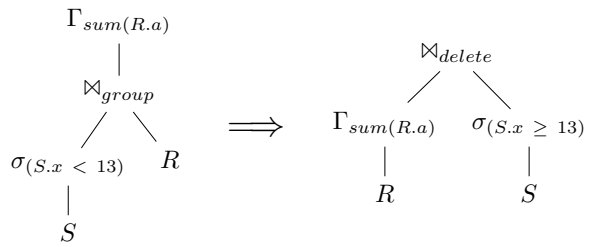
When a query uses the same attribute for both a join and group-by key (e.g., TPC-H Q3), the groupjoin [43] operator enables the reuse of the hash table constructed during the build side of the join to directly store the aggregate values. The following query exhibits this pattern:

```
select R.fk, sum(R.a)
from R, S
where R.fk = S.pk
  and S.x < 13
group by R.fk
```

While obviously more efficient than building two separate hash tables, the groupjoin does not actually improve the access patterns, since both the build and probe sides can have conditional reads followed by random lookups in the hash table.

We therefore propose to use an *eager aggregation* approach that unconditionally performs the aggregation before the join and then removes non-qualifying aggregates from the final result. While eager aggregation is a known technique [44], the main goal was to reduce the number of tuples that appear in subsequent joins. Instead, our version actually ends up performing unnecessary extra work (i.e., by computing aggregates for keys that are later discarded during the join) in order to achieve more favorable access patterns.

The eager aggregation optimization for the example query would look as follows:



As shown in the original version (left), we first build a hash table for all `S.pk` keys where `S.x < 13`. Then, on the probe side, we perform a lookup for every tuple in `R`. If a match is found, we update `sum` with the corresponding `R.a` value.

The eager aggregation version (right) reverses the traditional build and probe sides, performing an unconditional aggregation on `R` grouped by `R.fk`. The next step involves a sequential scan of `S` to select all non-qualifying `S.pk` keys and delete them from the hash table; in particular, note that the predicate has been inverted in the rewritten version to perform the deletion.

To understand when this optimization is beneficial, consider the following cost model for the groupjoin:

$$\begin{aligned} Groupjoin = S \cdot (&read_{seq} + \sigma_S \cdot (read_{cond} + ht_{insert})) \\ &+ R \cdot (read_{seq} + \sigma_R \cdot (read_{cond} + ht_{lookup}) \\ &+ \bowtie_{R,S} \cdot \max(comp, read_{cond})) \end{aligned}$$

where the first term is the cost of building the hash table for every tuple in S that passes predicate σ_S, and the second term is the cost of looking up the key for every tuple in R that passes predicate σ_R plus the cost of performing the final aggregation with probability $\bowtie_{R,S}$ that a join match is found.

On the other hand, the cost model for eager aggregation (EA) is:

$$\begin{aligned} EA = R \cdot (&read_{seq} + \sigma_R \cdot \min(Hybrid, VM, KM)) \\ &+ S \cdot (read_{seq} + (1 - \sigma_S) \cdot (read_{cond} + ht_{delete})) \end{aligned}$$

where the first term is the unconditional construction of the hash table for all tuples in R that pass predicate σ_R, and the second term is the subsequent deletion of keys filtered by the join. As shown in Section IV, this technique is most beneficial for queries with a smaller number of group-by keys or higher selectivities.

Finally, although we have introduced our eager aggregation approach in the context of groupjoins, the techniques can similarly be applied to equijoins with a few simple extensions. The basic idea is to again reorder the traditional build and probe sides of the join, performing a partial aggregation on the new build side grouped by the join key. Then, for all matches on the new probe side, we perform the final aggregation step with the actual group-by key.

IV. Evaluation

This section presents a detailed experimental evaluation of SWOLE using (1) the TPC-H [10] decision support benchmark and (2) a series of microbenchmarks that specifically isolate each proposed optimization. We ran all experiments on a server with an Intel E5-2660 v2 CPU (2.2GHz, 10 cores, 25MB cache) and 256GB RAM.

In all experiments, we compare SWOLE against both the data-centric and hybrid strategies. We did not include ROF because: (1) the reported relative runtimes [5] are the same or worse than we measured for hybrid; and (2) our evaluation hardware did not support the required AVX2 instructions.

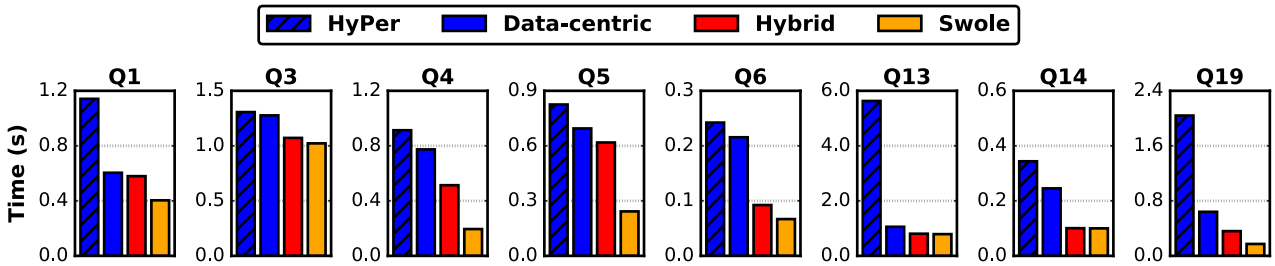


Fig. 6: TPC-H Results (SF 10)

Similar to a recent study [27], we hand coded each strategy in C to eliminate any overheads from tangential implementation differences or additional features (e.g., concurrency control, recovery logging) implemented in a full system. Moreover, we use the same library code (e.g., hash table implementations) and compile all versions using GCC-9.2, with experiments run single-threaded. This type of comparison fully isolates each code generation strategy, allowing us to evaluate them on a level playing field.

All of our implementations use well-known compression techniques, including: (1) dictionary encoding for low-cardinality string columns; (2) null suppression for low-cardinality integer columns; and (3) fixed-point storage, where decimals are multiplied by a power of 10 and stored as integers. Rather than performing explicit overflow checking, all aggregates are stored as 64-bit integers. For approaches that operate on tiles, we use a vector size of 1024, as suggested by other recent studies [5], [27]. Finally, we do not use any auxiliary data structures (e.g., indexes, materialized views) for query processing other than foreign key indexes built for checking referential integrity.

A. TPC-H

We selected the same eight TPC-H queries used in a recent code generation paper [5], which are a representative subset of the TPC-H benchmark [45], and conducted all experiments at scale factor (SF) 10. Figure 6 shows the results.

In addition to our hand-coded implementations, we also include HyPer v0.5-222, which is not intended as a direct comparison point but rather a sanity check to demonstrate that our implementations are reasonable. Our results for HyPer, which include only query processing time and not other parts of the execution (e.g., SQL parsing, planning, code generation), comport with published results from similar evaluations [46], [5], [27].

Overall, our hand-coded data-centric implementations outperform HyPer on all of the tested queries, ranging from a speedup of $1.02\times$ (Q3) to as much as $5.32\times$ (Q13). As such, we believe that they are a fair and accurate representation of the data-centric code generation strategy.

Relative to the data-centric baseline, our implementations of the hybrid approach exhibit speedups ranging from $1.04\times$ (Q1) up to $2.43\times$ (Q14). As expected, we observe the largest performance improvements in queries with more complex predicates that select relatively few tuples, where hybrid can leverage the prepass technique for SIMD vectorization of predicates and selection vectors for efficient filtering.

Finally, our SWOLE implementation outperforms the hybrid strategy by up to $2.63\times$ and data-centric by almost $4\times$. On almost every query we test, the application of SWOLE's techniques achieves significant speedups by inducing more favorable access patterns.

In the following, we provide a detailed discussion of the results for each query.

1) Q1: This query is a single-table scan of `lineitem` with a single predicate that selects almost all (i.e., roughly 98%) of the tuples. Q1 also contains the most compute-intensive aggregation of any TPC-H query.

Since the predicate is fairly simple (i.e., a single comparison on `l_shipdate`) and does not filter many tuples, the hybrid strategy provides almost no performance improvement over data-centric, with only a $1.04\times$ speedup. SWOLE, however, outperforms hybrid by an additional $1.43\times$.

In this case, SWOLE uses key masking to perform the aggregation rather than value masking. Our cost model determines that the complexity of the aggregation would require masking many individual aggregate values, which is significantly more expensive than masking the single group-by key. Moreover, the fact that the predicate selects nearly the entire `lineitem` table means that SWOLE performs very little wasted work.

2) Q3: This query involves a join between `customer` and `orders` followed by a groupjoin with `lineitem`. Every table has a predicate that filters at least half of the tuples, but each predicate contains only a single comparison.

The hybrid strategy achieves an improvement of $1.19\times$ compared to data-centric, due to the relative simplicity of each individual predicate. Similar to Q1, SWOLE again offers a $1.48\times$ speedup over hybrid by using a positional bitmap for the first join between `customer` and `orders`. Although we could replace the groupjoin with eager aggregation of the `lineitem` table, our cost model determines that too many keys are filtered by the join for this rewrite to be beneficial.

3) Q4: Since the predicate on `orders` has a selectivity of about 4%, the majority of the runtime in Q4 is spent constructing the hash table on `lineitem` for the semijoin. As such, the hybrid strategy has a moderate speedup of $1.5\times$ over the data-centric version, which comes from applying the prepass technique to the `orders` and `lineitem` predicates.

Unlike hybrid and data-centric, which again use a hash table to implement the semijoin, SWOLE builds a positional bitmap with a sequential scan of the `lineitem` table. Then, with a second sequential scan of `orders`, SWOLE probes the bitmap at the corresponding position to check for a match. This optimization results in a $2.63\times$ speedup over hybrid, which is the largest out of all TPC-H queries that we test.

4) Q5: With a total of six tables, Q5 is the query with the most complex plan that we evaluate. All of the joins are simple equijoins, with predicates on only `region` and `orders`. The largest table (i.e., `lineitem`) has no predicate, though, which means a hash table lookup is required for every tuple.

Since the runtime of Q5 is dominated by the join involving the unfiltered `lineitem` table, hybrid achieves only a $1.12\times$ improvement over data-centric by applying the prepass technique to the scan of `orders`, which is the second largest table in the query. SWOLE, on the other hand, shows a large $2.55\times$ speedup over hybrid, again through extensive use of bitmap semijoins. With only about 3% of tuples remaining after the last join, our cost model decides to replace all joins with bitmap semijoins and use the late materialization technique described in Section III-D before the final aggregation.

5) Q6: Like Q1, this query is also a single-table scan of `lineitem`. However, Q6 has several additional predicates that select only about 2% of the tuples.

Due to these additional predicates, which perform a total of five comparisons involving three different attributes, hybrid markedly outperforms the data-centric strategy by $2.33\times$. SWOLE achieves another $1.38\times$ improvement over hybrid through a combination of access merging for the `l_discount` attribute, which is used in the predicate as well as the aggregation, and value masking. Yet, while our cost model determines that value masking is still beneficial, the overall benefit is limited because of the high percentage of wasted work (i.e., approximately 98%).

6) Q13: Although appearing deceptively complex, Q13 requires only a simple groupjoin between `customer` and `orders` before a final aggregation step. The only predicate is a complex string-matching expression with three wildcard characters, which selects about 98% of the tuples.

Even though this string-matching predicate cannot benefit from SIMD vectorization, the hybrid approach still splits the predicate into a separate prepass loop, yielding a $1.31\times$ speedup relative to data-centric. For the aggregation, SWOLE utilizes the value masking technique, which incurs relatively little wasted work because nearly all tuples in the `orders` table pass the predicate. However, since the query runtime is dominated almost entirely by the string-matching predicate, SWOLE offers only a very slight additional benefit.

7) Q14: This query is a relatively straightforward index join between the `lineitem` and `part` tables. Since `p_type` is a low-cardinality attribute, the string-matching expression can be converted to a lookup in a small hash table computed on the fly during an initial scan of `part`.

The hybrid strategy achieves a $2.43\times$ speedup over the data-centric approach because the predicate, which performs two comparisons on the `l_shipdate` attribute, selects only about 1% of the `lineitem` table. Due to the small percentage of selected tuples and high overhead of the index join, SWOLE cannot further improve the performance.

8) Q19: Finally, Q19 requires a join between `part` and a filtered subset of `lineitem`. The join condition is a complex disjunctive predicate, and only a handful of tuples comprise the final aggregate.

By enabling SIMD vectorization for the independent predicates on `lineitem` (i.e., for `l_shipmode` and `l_shipinstruct`), hybrid gets a $1.78\times$ improvement over data-centric. However, the hybrid strategy cannot improve the evaluation of the join condition, which takes a considerable amount of processing effort.

On the other hand, SWOLE outperforms hybrid by $2.07\times$ using the positional bitmap technique described in Section III-D. In particular, SWOLE builds a total of three bitmaps in a purely sequential scan of the `part` table. The join then resolves to a union of semijoins, where we can use the bitmap that corresponds to each `lineitem` tuple.

B. Microbenchmarks

To further evaluate SWOLE, we created a microbenchmark that isolates each of the techniques described in Section III. Figure 7 shows the microbenchmark specification, including the schema (7a), five queries (7b), and a legend (7c).

The main table is `R`, which has 100M tuples. Each attribute has a data type (e.g., *int8*) and a cardinality (e.g., 100). Some attributes have multiple types and cardinalities listed because they are substituted in different query configurations to stress various dimensions. The other table is `S`, which has either 1K or 1M tuples in order to test joins of different sizes.

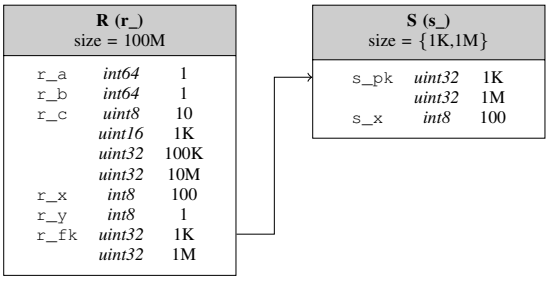
All values are assigned randomly with a uniform distribution, which represents the worst case for operations that use a hash table (e.g., group-by aggregation, join), since skew means some keys are more common than others and, therefore, more likely to be cached. In other words, a lookup in a large hash table with uniformly distributed values will almost certainly result in a cache miss. This lack of caching is important for stressing our techniques that perform unconditional lookups.

Similar to TPC-H, each of the queries also has substitution parameters that allow us to test different variations. The substitution parameters used for a particular configuration are displayed in the caption below the corresponding figure. For example, we replace the `OP` parameter of Q1 (Figure 8) with both the multiplication ($*$) and division ($/$) operators. The selectivity parameter `SEL` is typically varied from $0 - 100$ along the x-axis.

In the following, we describe our microbenchmark results for each query.

1) Q1: This query tests the value masking technique from Section III-A, and we show the results in Figure 8. As mentioned, we evaluate query performance for both multiplication (8a) and division (8b).

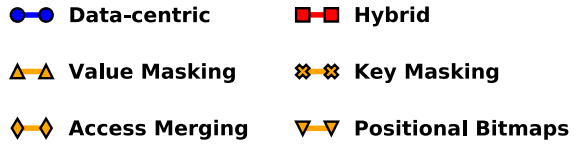
The results for the data-centric and hybrid strategies are unsurprising. For the multiplication query, data-centric exhibits



(a) Schema

Query	SQL	Parameters
Q1	`select sum(r_a [OP] r_b) from R where r_x < [SEL] and r_y = 1`	OP, SEL
Q2	`select r_c, sum(r_a * r_b) from R where r_x < [SEL] and r_y = 1 group by r_c`	|r_c|, SEL
Q3	`select sum(r_x * [COL]) from R where r_x < [SEL] and r_y = 1`	COL, SEL
Q4	`select sum(r_a * r_b) from R, S where r_fk = s_pk and r_x < [SEL1] and s_x < [SEL2]`	SEL1, SEL2
Q5	`select r_fk, sum(r_a * r_b) from R, S where r_fk = s_pk and s_x < [SEL] group by r_fk`	|S|, SEL

(b) Queries

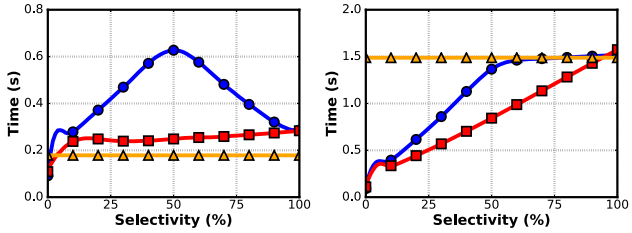


(c) Legend

Fig. 7: Microbenchmark Specification

the well-known curve [31] that results from CPU branch mispredictions. Initially, the hybrid runtime increases until leveling off at roughly 15% selectivity when the approach becomes memory-bound.

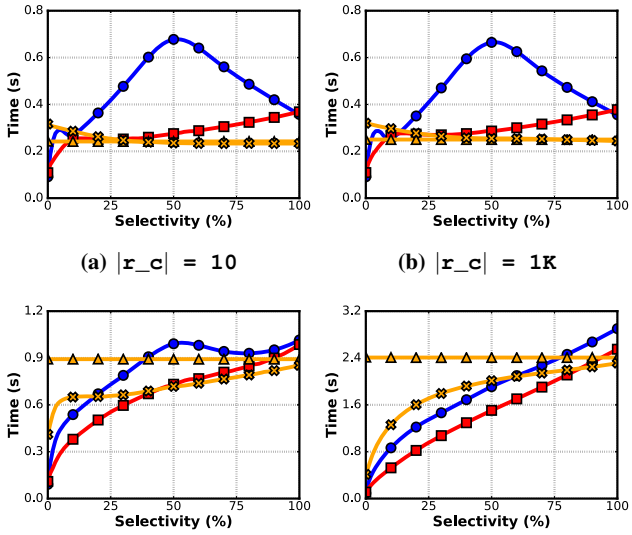
The division query, on the other hand, is much more compute-intensive. Consequently, the data-centric runtime does not decrease after peaking at 50% selectivity, as observed in the multiplication query. The runtime for the hybrid strategy also steadily increases along with selectivity.



(a) OP = *

(b) OP = /

Fig. 8: Microbenchmark Q1



(a) |r_c| = 10

(b) |r_c| = 1K

(c) |r_c| = 100K

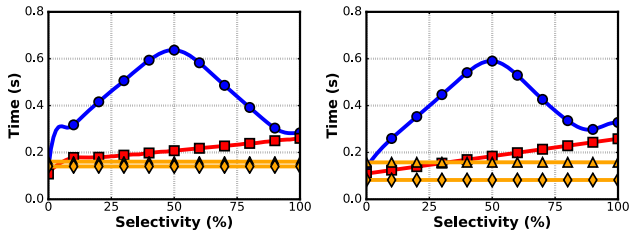
(d) |r_c| = 10M

Fig. 9: Microbenchmark Q2

For both query configurations, our value masking technique exhibits a constant runtime across the entire selectivity range. In fact, value masking outperforms the other two approaches for nearly all selectivities in the memory-bound multiplication query, but it only becomes beneficial in the compute-bound division query at around 95% selectivity. Importantly, this result validates the intuition that improving data access patterns is the most important consideration for memory-bound queries.

2) Q2: To test the key masking technique described in Section III-B, we extended the first configuration of Q1 to perform a group-by aggregation. Figure 9 shows the results, which include four different cardinalities for the group-by key `r_c` ranging from small (10) to very large (10M).

For all strategies, the performance difference between 10 (9a) and 1K (9b) keys is almost indistinguishable, as the hash tables are small and fit entirely in the cache. In fact, the absolute runtimes for these two cases are only slightly worse than those shown in Figure 8a, which has no group-by key. Moreover, key masking exhibits virtually equivalent performance to value masking, again due to the low overhead of accessing a small hash table.



(a) COL = r_b (b) COL = r_y

Fig. 10: Microbenchmark Q3

At 100K keys (9c), the query starts to become bottlenecked by the hash table lookups. Hence, value masking becomes markedly worse than key masking. The key masking technique performs comparatively poorly for low selectivities but begins overtaking hybrid at around 45% selectivity due to the better access patterns.

Finally, for the very large hash table with 10M keys (9d), the hybrid strategy outperforms all alternatives until roughly 85% selectivity, when key masking again becomes beneficial. As previously mentioned, these results contradict the findings of Voodoo [37], which posited that key masking (i.e., predicated lookups) is a dominant approach.

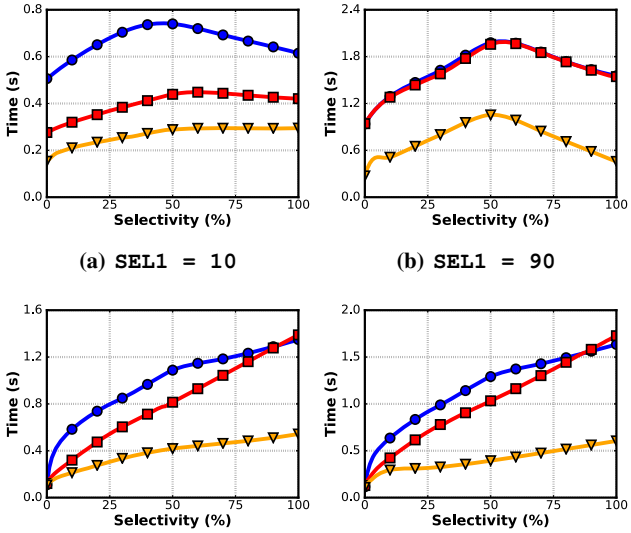
3) Q3: Again, this query resembles Q1 but reuses the predicate attributes in the aggregation to test our access merging technique from Section III-C. As shown in Figure 10, the data-centric and hybrid strategies exhibit similar performance to the results in Figure 8a, with slightly lower overall runtimes due to the decreased memory pressure from referencing fewer attributes. Compared to value masking, access merging achieves only about a 1.15× speedup in the configuration that reuses only one attribute (10a) but over a 1.9× speedup in the configuration that reuses both (10b).

4) Q4: Unlike the previous queries, Q4 contains a join between `R` and `S`. The two substitution parameters `SEL1` and `SEL2` control the selectivity on the probe and build sides, respectively. For this experiment, `S` has size 1M.

Figure 11 shows the results. First, we fix the selectivity of the probe side at 10% (11a) and 90% (11b) while varying the selectivity of the build side. Then, we vary the selectivity for the probe side with a fixed selectivity of 10% (11c) and 90% (11d) on the build side.

As shown, our positional bitmaps, which are described in Section III-D, significantly outperform the other two approaches in all configurations. Data-centric and hybrid perform comparably, since the main factor in overall query runtime is the hash table lookups. The only exception is the top left configuration, where the 10% selectivity on the probe side means that the predicate filters most of `R` and relatively few hash table lookups occur.

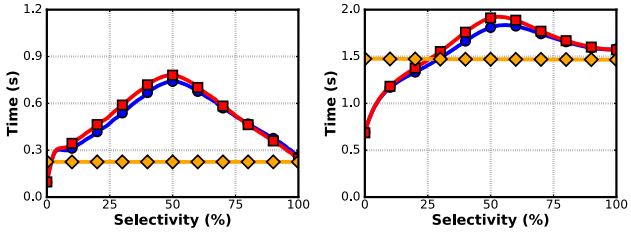
5) Q5: The main focus of Q5 is the groupjoin operator, which allows us to test our eager aggregation technique (Section III-E). The query contains a predicate on `S` but not on `R`, which represents the worst case for our approach; that



(a) SEL1 = 10 (b) SEL1 = 90

(c) SEL2 = 10 (d) SEL2 = 90

Fig. 11: Microbenchmark Q4



(a) |S| = 1K (b) |S| = 1M

Fig. 12: Microbenchmark Q5

is, we will need to unconditionally aggregate all tuples in `R`, many of which could be discarded depending on the selectivity of the `S` predicate. On the other hand, the traditional groupjoin only performs the aggregation for tuples with a match in the hash table.

Figure 12 shows the results for Q5 when `S` has both 1K (12a) and 1M (12b) tuples. The data-centric and hybrid approaches exhibit nearly identical performance for both join sizes, since the runtime is again dominated by the hash table lookups for every tuple in `R`. Due to branch mispredictions during the lookup (i.e., predicting whether a match exists), these strategies perform worst at around 50% selectivity for both the 1K and 1M cases.

Like our other techniques, the performance of eager aggregation remains relatively constant across the entire selectivity range, with a (very) slight improvement as selectivity approaches 100% because fewer non-qualifying aggregates need to be deleted. Eager aggregation is almost always superior for the 1K size but only becomes beneficial at around 30% selectivity for the 1M size, since lookups for the larger hash table are significantly more expensive.

V. Conclusion

This paper presented SWOLE, the first access-aware code generation strategy. Unlike existing approaches that apply predicate pushdowns to reduce processing effort, SWOLE counterintuitively leverages predicate pullups to achieve better access patterns, which often outweigh the cost of performing wasted work. Overall, our experiments demonstrated that SWOLE can outperform the state-of-the-art code generation strategy by over $2.6\times$.

Acknowledgments

This research was funded by NSF Career Award IIS-1453171 and supported by a Google PhD Fellowship.

References

[1] G. Graefe, "Volcano - an extensible and parallel query evaluation system," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 1, pp. 120–135, 1994.

[2] K. Krikellas, S. Viglas, and M. Cintra, "Generating code for holistic query evaluation," in *ICDE*, 2010, pp. 613–624.

[3] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *PVLDB*, vol. 4, no. 9, pp. 539–550, 2011.

[4] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *PVLDB*, vol. 8, no. 12, pp. 1466–1477, 2015.

[5] P. Menon, A. Pavlo, and T. C. Mowry, "Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last," *PVLDB*, vol. 11, no. 1, pp. 1–13, 2017.

[6] S. Manegold, P. A. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *VLDB*, 2002, pp. 191–202.

[7] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. L. Kersten, "CPU and cache efficient management of memory-resident databases," in *ICDE*, 2013, pp. 14–25.

[8] S. Zeuch and J. Freytag, "Selection on modern cpus," in *IMDM@VLDB*, 2015, pp. 5:1–5:8.

[9] S. Zeuch, H. Pirk, and J. Freytag, "Non-invasive progressive optimization for in-memory databases," *PVLDB*, vol. 9, no. 14, pp. 1659–1670, 2016.

[10] "Tpc-h," http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf, 2020.

[11] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, P. Tiberio, I. L. Traiger, B. W. Wade, and R. A. Yost, "System R: A relational data base management system," *IEEE Computer*, vol. 12, no. 5, pp. 42–48, 1979.

[12] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "A history and evaluation of system R," *Commun. ACM*, vol. 24, no. 10, pp. 632–646, 1981.

[13] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman, "Compiled query execution engine using JVM," in *ICDE*, 2006, p. 23.

[14] R. Greer, "Daytona and the fourth-generation language cymbal," in *SIGMOD*, 1999, pp. 525–526.

[15] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*, 2011, pp. 195–206.

[16] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong, "Tenzing A SQL implementation on the mapreduce framework," *PVLDB*, vol. 4, no. 12, pp. 1318–1327, 2011.

[17] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *SIGMOD*, 2013, pp. 1243–1254.

[18] C. Freedman, E. Ismert, and P. Larson, "Compilation in the microsoft SQL server hekaton engine," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 22–30, 2014.

[19] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, "Building efficient query engines in a high-level language," *PVLDB*, vol. 7, no. 10, pp. 853–864, 2014.

[20] S. Wanderman-Milne and N. Li, "Runtime code generation in cloudera impala," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 31–37, 2014.

[21] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder, "Impala: A modern, open-source SQL engine for hadoop," in *CIDR*, 2015.

[22] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik, "Tupleware: "big" data, big analytics, small clusters," in *CIDR*, 2015.

[23] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, "The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database," *PVLDB*, vol. 9, no. 13, pp. 1401–1412, 2016.

[24] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang, "Self-driving database management systems," in *CIDR*, 2017.

[25] J. Sompolski, M. Zukowski, and P. A. Boncz, "Vectorization vs. compilation in query execution," in *DaMoN*, 2011, pp. 33–40.

[26] T. Gubner and P. A. Boncz, "Exploring query compilation strategies for jit, vectorization and SIMD," in *ADMS@VLDB*, 2017, pp. 9–17.

[27] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *PVLDB*, vol. 11, no. 13, pp. 2209–2222, 2018.

[28] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *CIDR*, 2005, pp. 225–237.

[29] M. Zukowski, M. van de Wiel, and P. A. Boncz, "Vectorwise: A vectorized analytical DBMS," in *ICDE*, 2012, pp. 1349–1350.

[30] "Actian vector," https://www.actian.com/wp-content/uploads/2017/03/WP01-ActianVector-0424-1.pdf, 2020.

[31] K. A. Ross, "Conjunctive selection conditions in main memory," in *PODS*, 2002, pp. 109–120.

[32] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper, "Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation," in *SIGMOD*, 2016, pp. 311–326.

[33] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in *SIGMOD*, 1993, pp. 267–276.

[34] J. Chen, D. J. DeWitt, and J. F. Naughton, "Design and evaluation of alternative selection placement strategies in optimizing continuous queries," in *ICDE*, 2002, pp. 345–356.

[35] K. Dursun, C. Binnig, U. Çetintemel, G. Swart, and W. Gong, "A morsel-driven query execution engine for heterogeneous multi-cores," *PVLDB*, vol. 12, no. 12, pp. 2218–2229, 2019.

[36] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *SIGMOD*, 2002, pp. 145–156.

[37] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo - A vector algebra for portable database performance on modern hardware," *PVLDB*, vol. 9, no. 14, pp. 1707–1718, 2016.

[38] E. Babb, "Implementing a relational database by means of specialized hardware," *ACM Trans. Database Syst.*, vol. 4, no. 1, pp. 1–29, 1979.

[39] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh, "Quickstep: A data platform based on the scaling-up approach," *PVLDB*, vol. 11, no. 6, pp. 663–676, 2018.

[40] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *CoRR*, vol. abs/1402.6407, 2014.

[41] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. S. Y. Kai, "Roaring bitmaps: Implementation of an optimized software library," *CoRR*, vol. abs/1709.07821, 2017.

[42] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," in *ICDE*, 2007, pp. 466–475.

[43] G. Moerkotte and T. Neumann, "Accelerating queries with group-by and join by groupjoin," *PVLDB*, vol. 4, no. 11, pp. 843–851, 2011.

[44] W. P. Yan and P. Larson, "Eager aggregation and lazy aggregation," in *VLDB*, 1995, pp. 345–357.

[45] P. A. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *TPCTC*, vol. 8391, 2013, pp. 61–76.

[46] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, "Errata for "building efficient query engines in a high-level language" (PVLDB 7(10): 853-864)," *PVLDB*, vol. 7, no. 13, p. 1784, 2014.