Filter Representation in Vectorized Query Execution

Amadou Latyr Ngom

July

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andy Pavlo, Chair
Todd C. Mowry

Submitted in partial fulfillment of the requirements
for the Fifth Year Master's Program.

Keywords: Vectorization, Compilation, Filter, SIMD

Abstract

Advances in memory capacity have allowed Database Management Systems (DBMSs) to store large amounts of data in memory, thereby shifting the performance bottleneck of query execution from disk accesses to CPU efficiency (i.e., instruction count and cycles per instruction). One technique used to achieve such efficiency in analytical applications is batch-oriented processing or *vectorization*: it reduces interpretation overhead, improves cache locality, and allows for efficient loop optimizations (e.g., loop unrolling, SIMD vectorization). For each *vector* (i.e., a batch of tuples), the DBMS needs to track the set of valid tuples after a predicate application. To that end, systems employ one of two data structures, or *filter representations*: Selection Vectors (SelVecs) or Bitmaps. In this work, we analyze each approach's strengths and weaknesses to provide recommendations on how to implement vectorized operations. Through a wide range of micro-benchmarks, we determine that the optimal implementation strategy is a function of many factors: the cost of iterating through tuples, the cost of the operation itself, and how amenable it is to SIMD vectorization. Our analysis shows that Bitmaps perform better for operations that can be efficiently vectorized using SIMD instructions but that SelVecs perform better on all other operations due to a cheaper iteration logic.

Acknowledgments

I would like to thank my advisor Andy Pavlo for his guidance and help during my last two years at CMU. He has developed my interest in databases and computer science research in general and made my experience at CMU very educational and fun. I would also like to thank my amazing peers of the CMUDB group that helped me in this and other projects: Prashanth Menon, Matt Buttrovich, Wan Shen Lim, Lin Ma, Tanuj Nayak, Rohan Aggarwal, Gustavo Angulo, Tianyu Li, John Rollinson. Special shoutout to Prashanth Menon, whose help was invaluable in this project. Finally, I would like to thank my parents and siblings who guided and supported their son/little bro all throughout his life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

DBMSs typically handle two broad classes of applications: Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) applications. OLTP applications tend to be short-running, write-heavy queries that only touch a few entries in a database. OLAP applications are long-running, read-mostly queries that analyze large amounts of data. Due to the unique characteristics of each class, storage engines (i.e., the component of the DBMS stores the data), and query engines (i.e., the component that executes queries) must specialize storage formats and database algorithms to efficiently handle OLAP or OLTP applications [35].

OLTP applications aim for fast access and update on individual entries. Therefore, to improve cache performance, OLTP-specialized storage engines [1, 5, 7, 11], use a row-store (i.e., a storage format that stores all the attributes of an entry contiguously in memory and in cache). OLAP applications, on the other hand, optimize for fast reads and computing on attributes across multiple entries (e.g., to find the average value of an attribute across all tuples). OLAP-specialized storage engines [4, 8, 9, 10], thus, use a column-store (i.e., a storage format that stores similar attributes of different entries contiguously). Though most traditional DBMSs [5, 7] focused on OLTP applications and thus used row stores, column stores have resulted in significant performance gains for OLAP applications [12, 35].

Besides the storage engine, disk-accesses were, historically, the performance bottleneck of OLAP applications. For example, it takes around $1ns$ to perform random access in the L1 cache, $100ns$ for DRAM, and $17000ns$ for SSDs [6]. Disk accesses, thus, dominate the execution time of any query whose data is primarily disk-resident. Recent advances in DRAM manufacturing have exponentially decreased the price and increased the capacity of main-memory. The cost of DRAM per GB was around $1000 in 2000, $10 in 2010, and less than $1 in 2020 [3]. This dramatic decrease in cost, and the corresponding increase in capacity, have both obeyed Moore's Law [27]. It is now possible to store large databases entirely in DRAM. Rather than optimize for disk access, the query engines of in-memory DBMSs (i.e., DBMSs whose data fits in memory) must optimize for CPU efficiency by reducing instruction count and cycles per instruction when executing queries.

The traditional query execution technique, the Iterator Model [19], is ill-suited for in-memory DBMSs due to CPU efficiency reasons. It executes a query by interpreting the query plan. In this model, each database operator implements a virtual `next()` function that returns the next tuple it can generate; each expression (i.e. addition, multiplication) also provides an virtual `evaluate()`

```
SELECT * FROM A INNER JOIN B     # Hash Join
         ON A.id = B.id          # Join Predicate
         WHERE A.val < 10        # Scan Filter
```

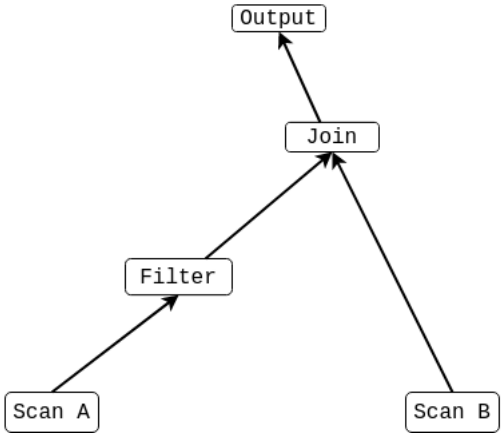


Figure 1.1: Sample SQL Query and Plan.

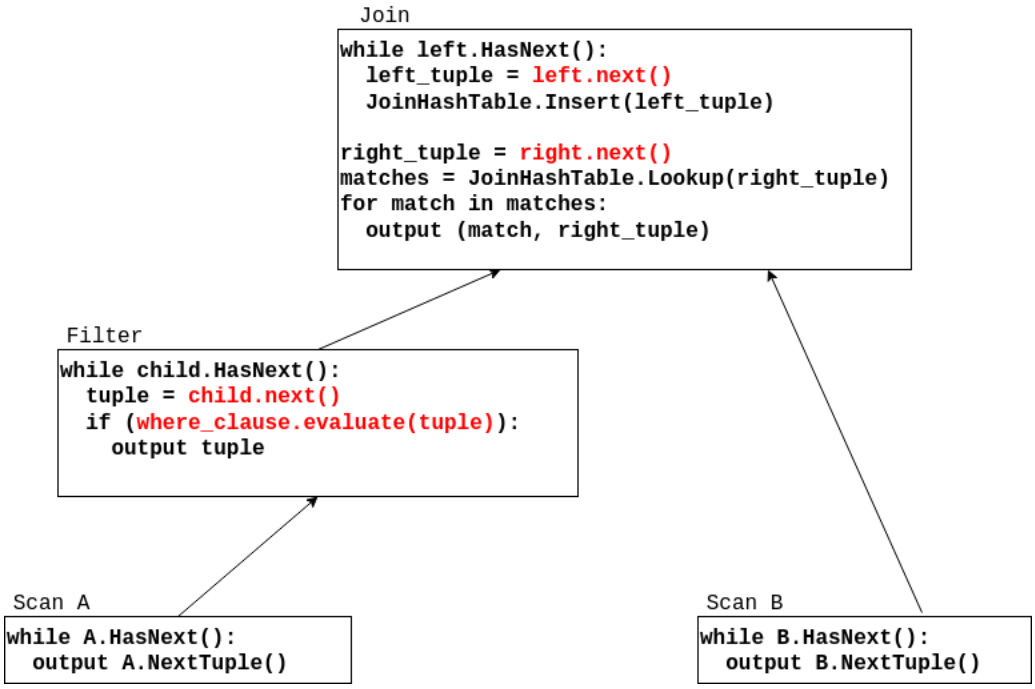


Figure 1.2: The Iterator Model on the sample SQL query – The code in red indicates virtual function calls.

function that takes in a tuple and returns a value. Figure 1.2 showcases the Iterator model with the sample query in Figure 1.1. The operators `Scan A` and `Scan B` iterate through their respective tables and output the tuples to their parent operators. The filter only outputs tuples that satisfy the `WHERE` clause. The join first builds a hash table, then probes it to find matches. For every

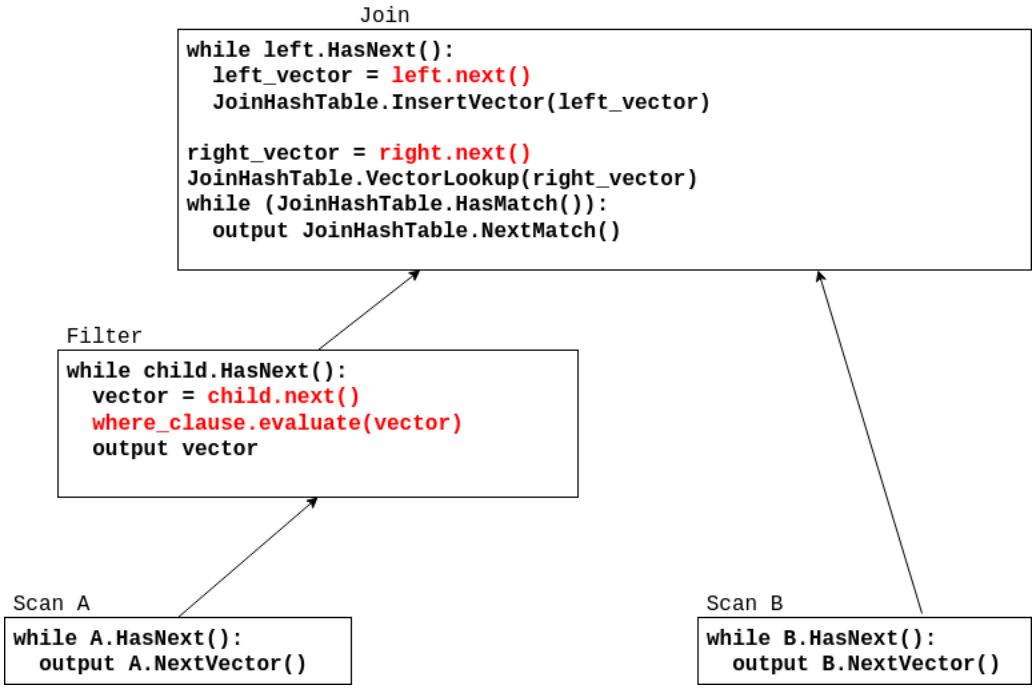


Figure 1.3: The Vectorization Model with the Sample Query – The code is nearly identical to that in Figure 1.2, except that it processes vectors.

tuple – of which there can be billions in an OLAP setting – the virtual function calls (marked in red in the figure) constitute a significant overhead, which decreases CPU efficiency. Since the 2000s, there has been much work dedicated to optimizing the CPU efficiency of in-memory column stores for high-performance OLAP applications.

Manegold et al. proposed an alternative to the Iterator Model: the Materialization Model [24]. In this model, rather than process and output one tuple at a time, each operator processes all its input tuples, then outputs all its results to the next operator. The authors also introduce specialized algorithms for each relational operator to exploit the columnar format of their system. Although the Materialization Model removes the Iterator Model's interpretation overhead, it can become memory-bound due to frequent materializations (e.g., copies) of large columns. To solve this issue, Boncz et al. proposed a middle ground between the Materialization and Iterator Model: the Vectorization Model [15]. The thesis explores strategies for speeding up vectorized queries (i.e., queries executed using the Vectorization Model). As such, we will dedicate the rest of this chapter to an overview of the Vectorization Model. We discuss alternative query engine designs and optimizations for in-memory analytical DBMSs in Chapter 5.

1.1 The Vectorization Model

The Vectorization Model is almost identical to the Iterator model except that it operates on batches of T tuples or *vectors* (e.g., 2048 tuples at a time). The `next()` function returns vectors and the `evaluate()` function operates on vectors too. Figure 1.3 showcases Vectorization

```
# Vector Multiplication by a constant.
MulVecByConst(Vec<double> in, double val, Vec<double> out):
  for (i = 0; i < in.size; i++):
     out[i] = val * in[i]
```

(a) Unoptimized version.

```
# Optimizion: SIMD vectorization and loop unrolling
MulVecByConst(Vec<double> in, double val, Vec<double> out):
  # Assume size is multiple of 16
  for (i = 0; i < in.size; i += 16):
     # First copy of the loop body
     in_vec = avx512_load(in + i)
     out_vec = avx512_mul(in_vec, val)
     avx512_store(out + i, out_vec)
     # Second copy of the loop body
     i += 8
     in_vec = avx512_load(in + i)
     out_vec = avx512_mul(in_vec, val)
     avx512_store(out + i, out_vec)
```

(b) Optimization version. There are two optimizations here: SIMD vectorization and loop unrolling with a factor of 2. The code in green indicates Intel's AVX512 instructions.

Figure 1.4: A Vectorized Multiplication Function – The programmer usually writes only the code on the left; the compiler can perform the optimizations on the right.

Model on the sample query in Figure 1.1; it is near identical to Figure 1.2. The difference is that processing occurs one vector at at time rather than one tuple at a time.

This batch-oriented processing provides several benefits. First, virtual function calls occur for every T tuples, which amortizes their overhead by a factor of T. Second, materializing T tuples at a time instead of all tuples eliminates the memory pressure of the Materialization Model. Third, functions that operate on vectors, also known as *vectorized primitives*, are usually amenable to loop optimizations (e.g., loop unrolling, SIMD vectorization) applied by either the compiler or the programmer. Consider, for a example, the primitive shown in Figure 1.4a. It multiplies the elements in a vector by a constant number, and stores the result in a new vector. The loop of this primitive has independent iterations. Furthermore, the body of the loop only contains a load, a multiplication, and a store, which can all operate on SIMD vectors. As a result, the compiler or programmer can unroll and SIMD vectorize the loop. Figure 1.4b shows one way to optimize the loop. The body now contains AVX512 instructions [22] that perform that load, the multiplication, and the store, each operating on eight elements at a time. Besides, the loop unrolling (by a factor of two) optimization reduces the cost of loop maintenance.

Despite these benefits, the Vectorization Model introduces a new challenge. Individual tuples within a vector may be *filtered out*, meaning that subsequent operators should not process them. For example, in Figure 1.3, the `Join` operator should not insert tuples that do no satisfy the `WHERE` clause in the `JoinHashTable`. It should only insert *selected* tuples – those not filtered out. Next, we discuss the methods used to identify selected tuples within a vector.

1.2 Filter Representation

A *filter representation* is the data structure used to identify selected tuples with a vector. Figure 1.5 displays the two common representations: Selection Vectors (SelVecs) and Bitmaps. A SelVec stores a list of indices into the selected tuples. Tuples at an index stored within the SelVec are selected; tuples without an index are implicitly filtered out. A Bitmap, on the other hand, stores a list of bits. Its set bits indicate the selected tuples; its unset bits indicate filtered out ones. In the rest of this thesis, we will use the noun *filter* to denote a particular instance of

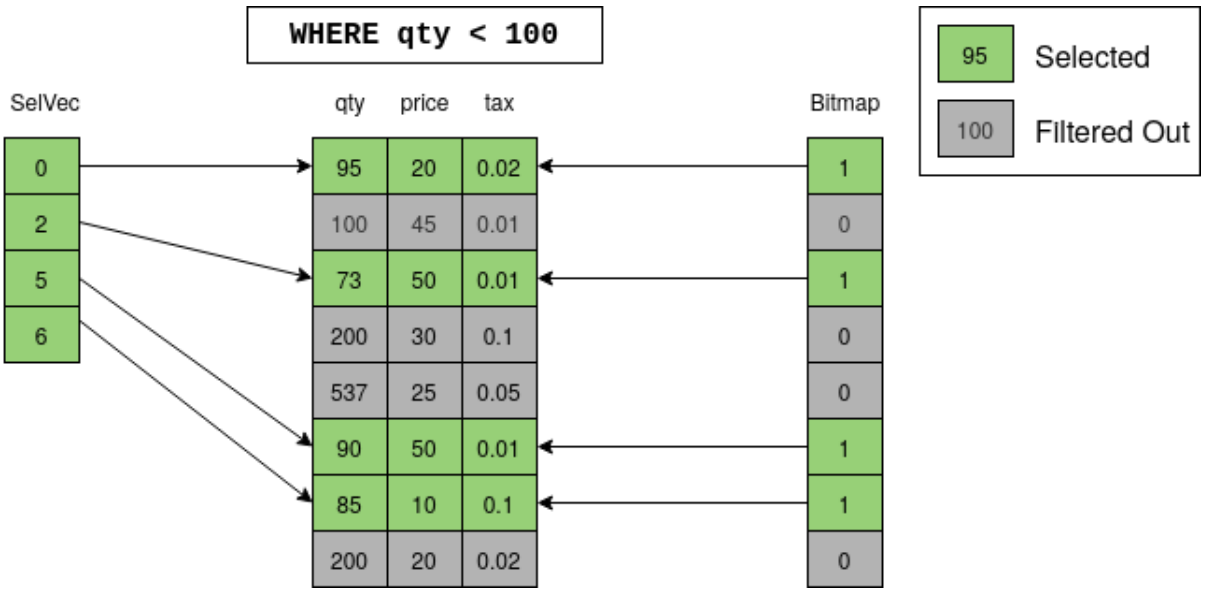


Figure 1.5: Filter Representation – Selection Vectors (left) index into selected tuples. Bitmaps (right) are only set on selected rows.

either data structure.

1.2.1 Contribution

Previous works on vectorized execution have chosen a filter representation strategy without providing an experimental comparison of the two approaches. Vectorwise [40], and works derived from it [23, 33, 34], rely on SelVecs. IBM DB2's BLU [31] and the more recent VIP [29] rely on Bitmaps for the intermediary results of a table scan's `WHERE` clause filters, and SelVecs for other relational operators. Nonetheless, we find that supporting both representations, and dynamically choosing between them results in better performance than static implementations. Depending on the specific primitive and on the selectivity (i.e., the ratio of selected tuples) of its input vector, SelVecs can outperform Bitmaps and vice-versa. This work's main contribution is a methodology on how to optimize the performance of arbitrary vectorized primitives by taking into account filter representation, selectivity, and loop optimizations. We also provide recommendations for developers working on vectorized query engines on how to best implement primitives.

The remainder of this thesis is structured as follows. In Chapter 2, we analyze the factors that influence the performance of vectorized primitive. We then derive insights for developers implementing vectorized query processing in Chapter 3. Next, we use these insights on the OLAP queries of the TPCH [37] benchmark and show the performance gains we obtain in Chapter 4. Finally, we discuss related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Computing on Filtered Vectors

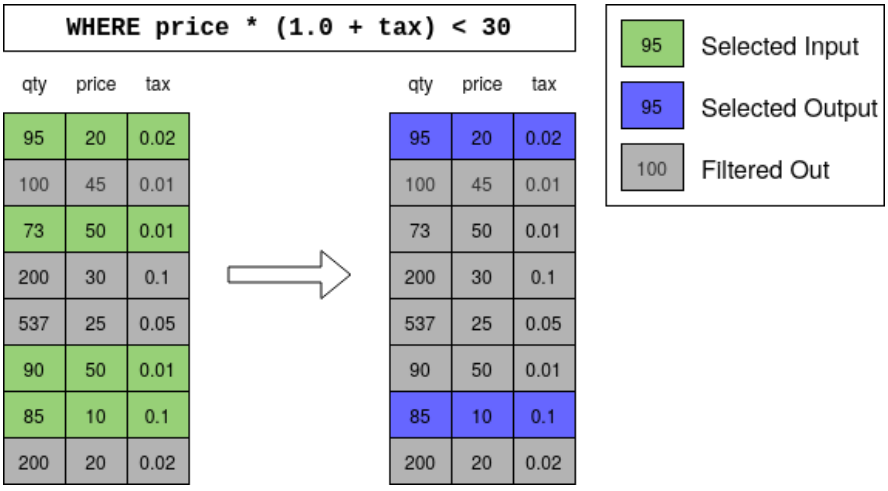
The purpose of this chapter is to discuss the factors that influence the performance of vectorized primitives. We first categorize the kinds of operations that primitives perform and the possible ways to implement them. Afterward, we will present and explain the equation that quantifies the performance of each implementation. Finally, we raise the main questions whose answers, provided in Chapter 3, will determine how to implement primitives optimally.

2.1 Operations on Filtered Vectors

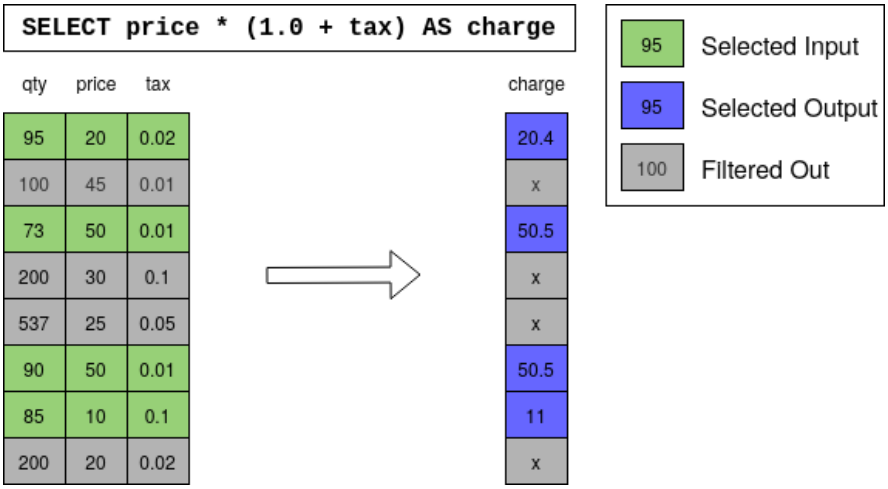
Figure 2.1 shows the two kinds of primitives we seek to optimize: Updates and Maps. Updates apply a predicate function to the vector and modify the set of selected tuples accordingly. A scan filter (i.e., `WHERE` clause) is an example of an update operation. On the other hand, Maps keep the selected set but compute a new vector using a mapping function. A projection (i.e., `SELECT` clause) is an example of a map operation. There are other primitives, but these often have side-effects (e.g., insertion in a hash table for joins or an array for sorting) and are, therefore, not amenable to our optimizations for correctness reasons (e.g., Full Compute introduced in Section 2.3). As such, we focus on side-effect-free primitives.

All relational operators are decomposable into a set of consecutive primitives. As mentioned above, simple relational operators like scan filters or projections consist of one individual primitive. Complex operators, like joins and aggregations, consist of multiple primitives. As a proof of concept, we showcase, in detail, the example of join probes in Figure 2.2. The first primitive computes the hashes of the input tuples; the result is a vector of hash values. It is, therefore, a Map primitive. The second primitive performs a hash table lookup using the hashes to obtain a vector of hash table entries. It is also a Map primitive. Due to hashing collisions, the entries may not match the input, so the algorithm compares the join keys to filter out inexact matches. For example, key 90's hit in the hash table is an exact match, whereas key 73's is not. The third primitive performs this comparison; it is an Update primitive. The fourth primitive advances the hash table chains to obtain the next entries (e.g., 85 follows 37); it is a Map primitive. The probe then loops back to the comparison primitive to find exact matches (e.g., 85 now has an exact match). The set of exact matches constitutes the output of the join probe.

Other relational operators (e.g., aggregations) are similarly decomposable.



(a) Update operation.



(b) Map operation.

Figure 2.1: Operations on Filtered Vectors – Updates change the set of selected tuple. Maps apply a function to the set of selected tuples without updating the set. Selected input tuples are in green. Selected output tuples are in blue.

2.2 Transitions

In between each primitive, operators may need to perform a *transition* (i.e., modifications to the filter passed into the next primitive). We now discuss transitions and their associated cost. Most of the time, this cost is non-existent because the same filter is passed on to the next primitive: the transition is *implicit*. For example, the hashing primitive and the lookup primitive in Figure 2.2 have the same filter; the transition performs no modifications. On the other hand, the Chain Following primitive operates on entries that do not satisfy the comparison: the transition involves a set difference. Set unions are another type of transition. Decomposed disjunctions [30] (shown in Figure 2.3), for example, involve both set differences and unions. They decompose disjunctive

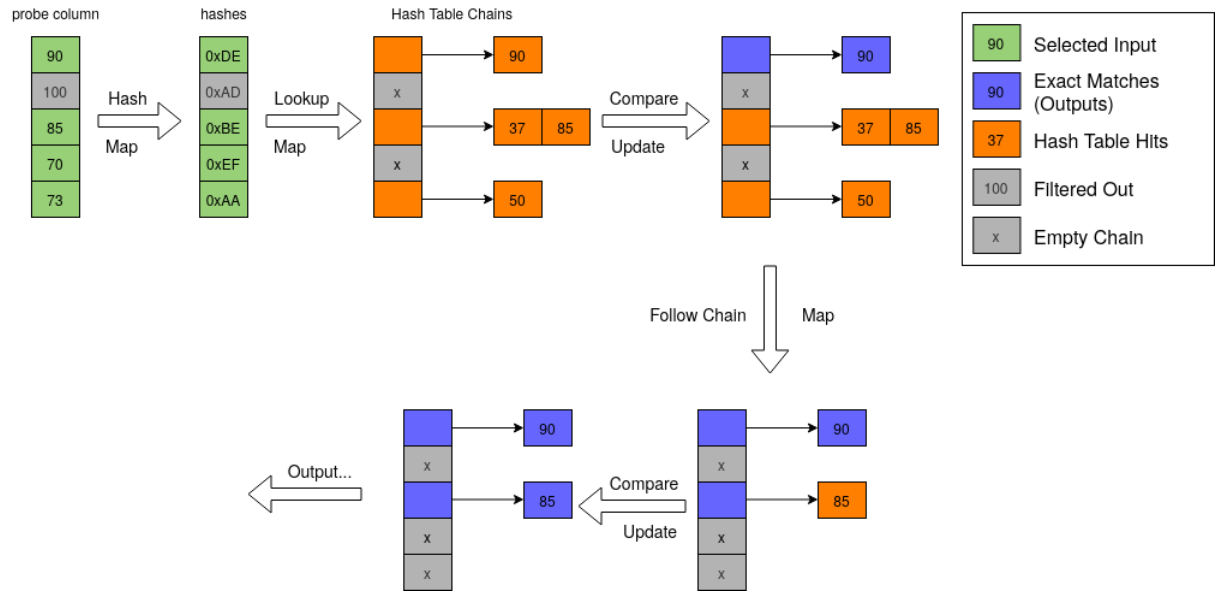


Figure 2.2: Multi-Step Join Probe – The color green indicates selected items in the input vector. Orange indicates hash table lookup hits. Blue indicates exact matches after key comparison; these represent the output of the probe. Greyed out squares are filtered out tuples, or empty hash table chains.

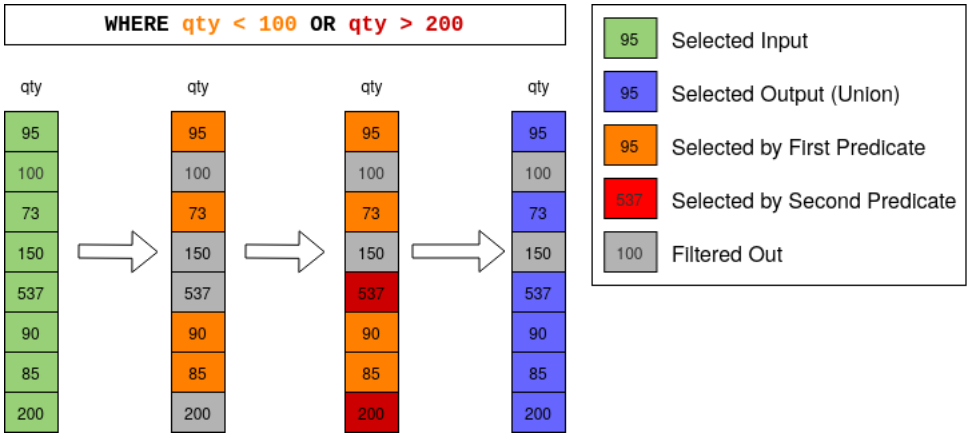


Figure 2.3: Decomposed Disjunctions – We apply the first clause (in orange) on all input tuples. In the second step, however, we only apply the second clause (in red) on those tuples that do not pass the first clause. In the third step, we take the union between the two outputs. The output tuples are in blue.

`WHERE` clauses into a sequence of primitives – one each for each conditional term. Each primitive operates on only tuples that have not satisfied the previous conditional terms, hence the set differences. The final filter consists of tuples that satisfy at least one primitive, hence the unions.

To the best of our knowledge, transitions are either implicit or take the form of set differences (for elements that do not pass the previous predicate) and unions (to obtain the final result). These operations are efficient to implement with Bitmaps by leveraging single-cycle bitwise AVX512

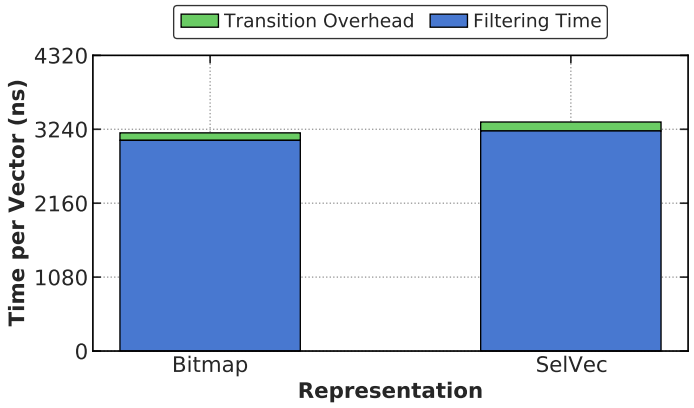


Figure 2.4: Transition Overheads – The overheads are 3.55% and 4.06% for Bitmaps and SelVecs respectively.

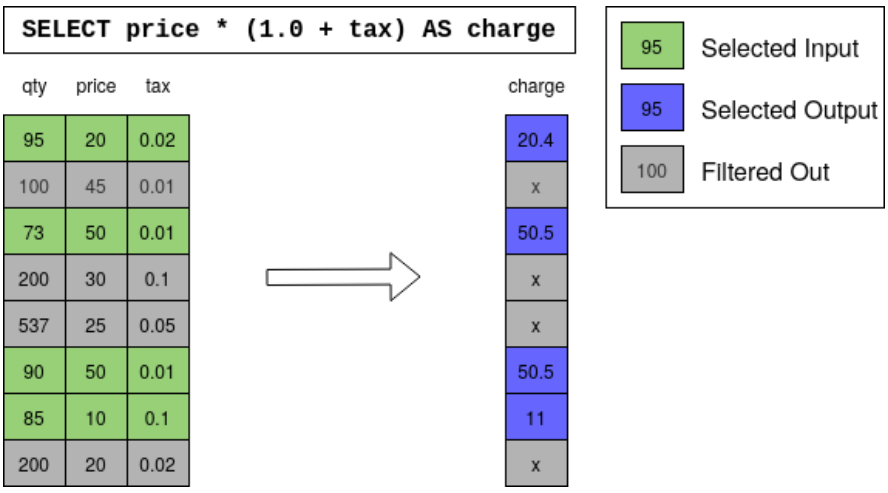
instructions [22] that operate on 512 bits at a time. Efficiently implementing set operations on SelVecs is more complicated. Whenever necessary, Updates can write to one additional SelVec for tuples that do not pass the filter; this second SelVec implicitly corresponds to the result of a set difference. Note that it does not overlap with the first SelVec, so a union of any of their subsets is just an array append, a fast operation that can leverage SIMD instructions.

To show that the overhead of these set operations is similar and negligible for both Bitmaps and SelVecs, we run an experiment that performs the decomposed disjunction shown in Figure 2.3. We synthetically generate the data so that each tuple has a 50% chance of satisfying either conditional terms, the average case scenario. The data contains around 2000000 tuples split into 1000 vectors. The results are shown in Figure 2.4. The overheads are negligible; they are also nearly equal (3.55% and 4.06% for Bitmaps and SelVecs, respectively). The representation, therefore, matters little when it comes to transitions. Given this, we will focus on optimizing primitives and will ignore transition costs.

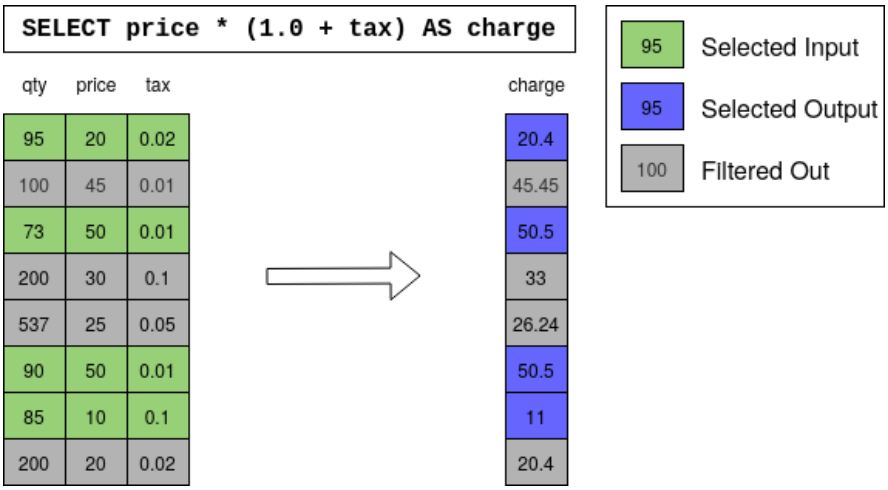
2.3 Compute Strategies

A compute strategy determines the implementation of Map and Update primitives. We consider three compute strategies: (1) Selective, (2) Full, and (3) Mixed Compute. Selective Compute only applies the given functions on selected tuples, whereas Full Compute applies them on all tuples. Figure 2.5 shows each approach's effect. The Selective Compute strategy (Figure 2.5a) leaves undefined values because it does not operate on filtered out entries, whereas the Full Compute strategy (Figure 2.5b) operates of every tuple and thus produces a result for each. As we show in Chapter 3, although Full Compute usually performs more work, it benefits from SIMD vectorization, simple loop structure (for loop unrolling and interleaving), and easy branch prediction. To exploit this trade-off, the third strategy, Mixed Compute, switches from Full Compute to Selective Compute when the selectivity (i.e., the ratio of selected tuples) goes below a threshold. The next chapter will show how to derive this threshold experimentally.

On Update primitives, Full Compute first finds the set of all tuples in the vector that pass



(a) Selective Compute. An x indicates an uninitialized value.



(b) Full Compute. There are no uninitialized values

Figure 2.5: Compute Strategies

the predicate, regardless of the input filter. The final filter is the intersection of this intermediary filter and the input filter. This process is shown in Figure 2.6. Notice how the intermediary filter contains elements filtered out of the input vector (e.g., 100 and 200). The intersection is, thus, necessary to remove these elements from the final output. Bitmap intersection is efficient thanks to the AVX512 and instruction that intersects 512 bits in one cycle [22], but SelVecs require a slow intersection of sorted sets [21]. Thus, Full Compute is not efficiently compatible with SelVecs on Update primitives. This incompatibility will become important when choosing a strategy to implement.

Furthermore, for Updates with SelVecs, there is another implementation decision: Conditional versus Unconditional Store. Figure 2.7 shows these options. Both primitives attempt to select the values less than a given input constant. They differ only in how they update the SelVec (code in green). The Conditional Store (Figure 2.7a) updates the SelVec when the condition is

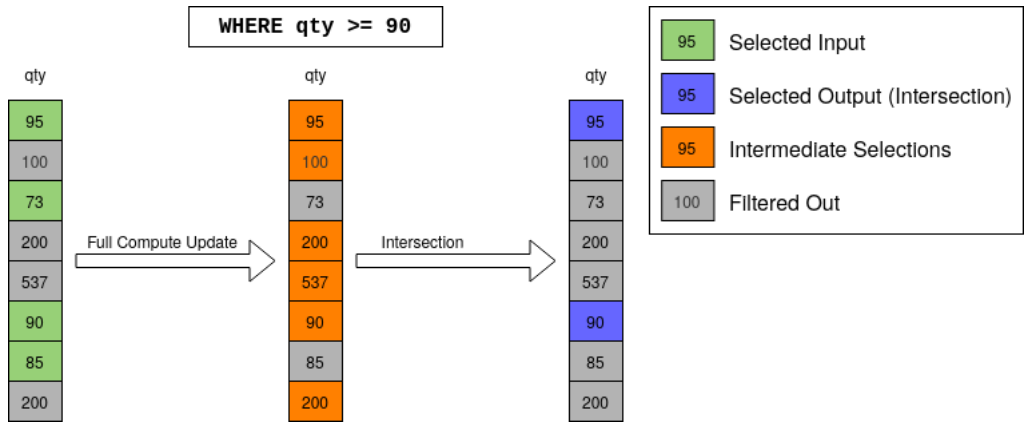


Figure 2.6: Update with Full Compute – The leftmost vector is the input vector; its selected elements are shown in green. The middle vector is the intermediary output of the full compute; its selected elements are shown in orange. The last vector has the intersection of the input filter and the intermediary filter; its selected elements are shown in blue.

true. The Unconditional Store (Figure 2.7b) always the updates the SelVecs, but only increments the write index when the condition is true. We do not focus on this decision, for it has been studied extensively before [33, 34]. Previous work identified branch mispredictions costs as the main factor influencing this decision. The Conditional Store implementation is competitive when the same branch is taken around 90% of the time or more, meaning that less than 10%, or more than 90% of tuples satisfy the condition. All the experiments in this thesis use the optimal strategy on Update primitives that use SelVecs.

Given a compute strategy and a filter representation, the next section provides a framework for analyzing the performance of any vectorized primitive.

2.4 Primitive Performance

This section analyzes the performance of primitives according to their implementations. Our primitive implementations follow one of the patterns shown in Figure 2.8. This listing shows the strategies to multiply a vector by a constant (a Map primitive) along with their optimized versions using SIMD instructions: Full Compute (which does not depend on representation), Selective Compute with SelVecs, and Selective Compute with Bitmaps. In each instance, the code is a loop divided in two parts: the *iteration logic* (code in red), and the *core operation* (code in green). The core operation (e.g., multiplication by a constant) is independent of the strategy. On the other hand, the iteration logic is strategy dependent; it determines how to iterate through tuples – whether selected ones in the case of Selective Compute or all tuples in the case of Full Compute – to perform the core operation and store its result.

The equation in Figure 2.9 shows the formula to compute the running time of a primitive. The first equality indicates that the total running time is equal to the number of tuples processed multiplied by the time spent per tuple. The second equality splits the time spent per tuple in two: the time dedicated to the iteration logic, and the time dedicated to the core operation. This

```
# Conditional
VecLt(Vec<double> in, double val, SelVec sel):
  w_idx = 0 # Where to write indices that pass the predicate
  for (i = 0; i < sel.size; i++):
     idx = sel[i]
     cmp = in[idx] < val
     # Conditional Store
     if (cmp):
       sel[w_idx] = idx
       w_idx++
```

(a) Conditional Store. We only update the SelVec when the condition is satisfied (code in green).

```
# Unconditional
VecLt(Vec<double> in, double val, SelVec sel):
  w_idx = 0 # Where to write indices that pass the predicate
  for (i = 0; i < sel.size; i++):
     idx = sel[i]
     cmp = in[idx] < val
     # Unconditional Store
     sel[w_idx] = idx
     w_idx += cmp
```

(b) Unconditional Store. We always update the SelVec regardless of the condition (code in green). The condition determines whether to increment the write index or not.

Figure 2.7: Conditional versus Unconditional Store – Both listings select the elements within the input vector (first parameter) that are less than the second parameter. Initially, the SelVec (third parameter) contains selected indices of the input vector. In the end, it only contains those that pass the predicate.

formula provides a framework to analyze any vectorized primitive. Let us consider how each of the three factors in the second equality relates to implementation strategies.

Number of Tuples Processed: Selective Compute processes only selected tuples; Full Compute processes every tuple. Thus, the lower the selectivity (i.e., the ratio of selected tuples), the more wasteful Full Compute is.

Iteration Logic Time per Tuple: Full Compute, while processing more tuples, has the simplest iteration logic because it does not aim to identify the selected ones. For Selective Compute, iterating over a Bitmap (e.g., Figure 2.8b) is more expansive than iterating over a SelVec (e.g., Figure 2.8c) because of the bit operations required by Bitmaps. Optimizations like loop unrolling and SIMD vectorization can further decrease the iteration logic time per tuple, but they are only available for Full Compute and SelVecs. The Bitmap's selective iteration logic is too complex to perform such optimizations.

Core Operation Time per Tuple: The core operation is the same for all strategies. The only way to reduce its contribution to the running time is through data-parallel SIMD instructions, which are only available with Full Compute and Selective Compute with SelVecs. There are,

```
# Without SIMD
MulVecByConst(Vec<double> in, double val, Vec<double> out):
  for (i = 0; i < in.size; i++):
     result = val * in[idx]
     out[idx] = result

# With SIMD
MulVecByConst(Vec<double> in, double val, Vec<double> out):
  # Assume size is multiple of 8
  for (i = 0; i < sel.size; i += 8):
     in_vec = avx512_load(in + i)
     result = avx512_mul(in_vec, val)
     avx512_store(out + i, out_vec)
```

(a) Full Compute. This function iterates through all elements, including filtered out ones.

```
MulVecByConst(Vec<double> in, double val, Vec<double> out, Bitmap bitmap):
  for (word in bitmap):
    while (word != 0):
      t = word & -word
      r = __builtin_ctzl(word)
      idx = k * 64 + r
      result = in[idx] * val
      out[idx] = result
      bitset ^= t
```

(b) Selective Compute with a Bitmap. The iteration logic is adapted from [18]. The logic is not amenable to SIMD vectorization.

```
# Without SIMD
MulVecByConst(Vec<double> in, double val, Vec<double> out, SelVec sel):
  for (i = 0; i < sel.size; i++):
     idx = sel[i]
     result = val * in[idx]
     out[idx] = result

# With SIMD
MulVecByConst(Vec<double> in, double val, Vec<double> out, SelVec sel):
  # Assume size is multiple of 8
  for (i = 0; i < sel.size; i += 8):
     idxs = avx512_load(sel + i)
     in_vec = avx512_gather(in, idxs)
     result = avx512_mul(in_vec, val)
     avx512_store(out + i, result) # Or avx512_scatter(out, result, idxs)
```

(c) Selective Compute with a SelVec. Reading an element involves an indirection to first obtain its index. In the SIMD code, the `gather` instruction handles this indirection. The last `scatter` is only necessary to maintain consistent indices between the input and output vector during multi-step Map operations. It should otherwise be avoided due to its slowness.

Figure 2.8: Implementing Vector Primitives – The code in green indicates the core operation (multiplication). The code in red indicates iteration logic.

however, situations where SIMD vectorization provides little gain or even hurts performance. Consider the example in Figure 2.10. This function selects an element from one of two vectors depending on the value in a third boolean vector. It corresponds to a relational `CASE` statement. For each loop iteration, the SISD code (Figure 2.10a) only executes one branch. The SIMD code (Figure 2.10b) executes all branches and relies on SIMD masking to set the correct vector

```
Let:
R = Running time of the primitive
N = Number of tuples processed.
T = Time per tuple
I = Iteration logic time per tuple
O = Core operation time per tuple

Performance Equation:
R = N × T = N × (I + O)
```

Figure 2.9: The Primitive Performance Equation

```
CastStmt(Vec<bool> condition, Vec<double> vec1, Vec<double> vec2, Vec<double> out):
  for (i = 0; i < vec1.size; i++):
    if (condition[i]):
      # Load vec1 when the condition is true
      result = vec1[i]
    else:
      # Load vec2 when the condition is false
      result = vec2[i]
    out[i] = result
```

(a) SISD code.

```
CastStmt(Vec<bool> condition, Vec<double> vec1, Vec<double> vec2, Vec<double> out):
  # Assume size is a multiple of 8
  for (i = 0; i < vec1.size; i += 8):
    # Load vec1 when the condition is true.
    mask = condition[i:i+8]
    result = avx512_masked_load(vec1 + i, mask)
    # Load vec2 when the condition is false.
    mask = avx512_negate(mask)
    result = avx512_masked_load(vec2 + i, mask)
    avx512_store(out + i, result)
```

(b) SIMD code.

Figure 2.10: SISD Branching Code versus SIMD – Both listings select elements from one of `vec1`, or `vec2` depending on the `condition`. In the SISD code, only one branch is executed every iteration. In the SIMD code, we rely on masking to select which vector to load from, but all instructions are executed at every iteration.

lanes; it ends up executing more instructions that the SISD code, which reduces the benefits data-parallelism. In general, core operations benefit most from SIMD vectorization when they only contain straight-line (i.e., no branches), arithmetic, and bitwise instructions.

The explanation above raises three questions that can allow us to determine the optimal strategy for any primitive: What is the input vector's selectivity? Can we use data-parallelism to perform the core operations on multiple elements at a time? Does the core operation contain instructions that make data-parallelism inefficient (e.g., branching)? The next chapter experimentally derives the best strategies according to the answers to these questions.

Strategy Name	Representation	Compute	SIMD	Compatibility
SelVecPartial	Selection Vectors	Selective	None	Update and Map
SelVecManual	Selection Vectors	Selective	Manually Written	Update and Map
BitmapPartial	Bitmaps	Selective	None	Update and Map
BitmapFull	Bitmaps	Full	Auto Vectorization	Update and Map
BitmapFullManual	Bitmaps	Full	Manually Written	Update and Map
Full	Either	Full	Auto Vectorization	Map
FullManual	Either	Full	Manully Written	Map

Table 3.1: Implemented Strategies – For the SIMD columns, 'Auto Vectorization' indicates that we rely on compiler auto vectorization, whereas 'Manually written' indicates that we write SIMD code ourselves using intrinsics. On the compatibility column, Update operations cannot use both Full Compute and Selection Vectors, which explains the last two entries.

Chapter 3

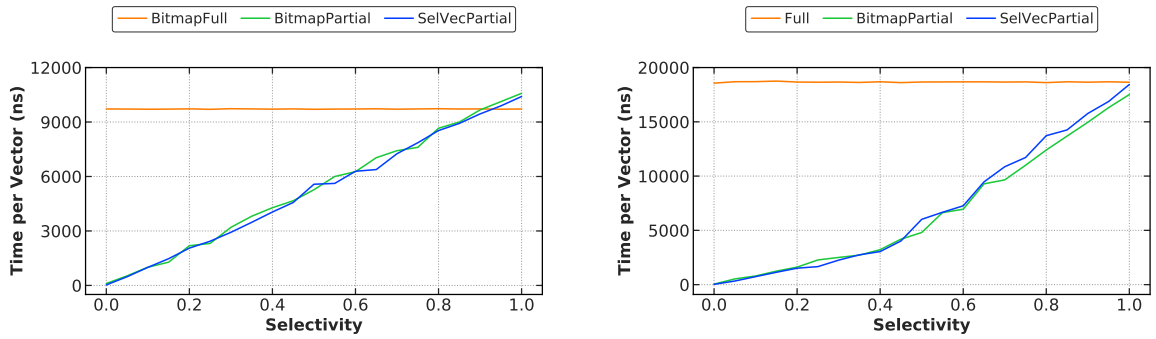
Optimal Strategies

In this chapter, we aim to derive the optimal execution strategy for any given primitive experimentally. In doing so, we will provide a decision tree that developers can use to guide the implementation of vectorized primitives.

The chapter is divided into a series of experiments based on the questions raised in Section 2.4 to determine the optimal strategy for a given primitive. Table 3.1 summarizes all the strategies we implemented. As noted before, there is no Full Compute strategy for Updates that works with SelVecs. Update experiments will thus always use a Bitmap for Full Compute; For Map experiments, the representation is irrelevant because Full Compute neither reads nor updates the filter. In addition to relying on compiler auto-vectorization, we also experimented with manual vectorization (e.g., with SelVecManual) for straight-line arithmetic and bitwise operations (as in Section 3.3).

We run the experiments on an Intel Xeon Platinum 8124M CPU @ 3.00GHz with AVX512 support. We use the NoisePage DBMS [17], with a read-only column store as its storage engine, compiled with clang version 9 [2].

In each experiment, we synthetically generate the data and manually vary the selectivity of each primitive's input vector from 0.0 to 1.0 in increments of 0.05 to show the impact selectivity



(a) Update operation corresponding to `WHERE country < 'Japan'`

(b) Map operation corresponding to `SELECT POSITION('a' in country)`.

Figure 3.1: Variable Length Data – Country names are all short strings that highlight the fact that even cheap variable-length operations should avoid full compute.

has on what the optimal strategy is. For example, we set a random 25% of bits in a Bitmap to obtain filter with a selectivity of 0.25. Each experiment executes its primitive on enough vectors to obtain a stable average running time. For example, fast primitives are executed in the order of 10^6 times, whereas the slower ones are executed in the order of 10^4 or 10^5 times. Map primitives materialize their results on a vector in-memory, but Update primitives only modify the input filter without materializing a new vector.

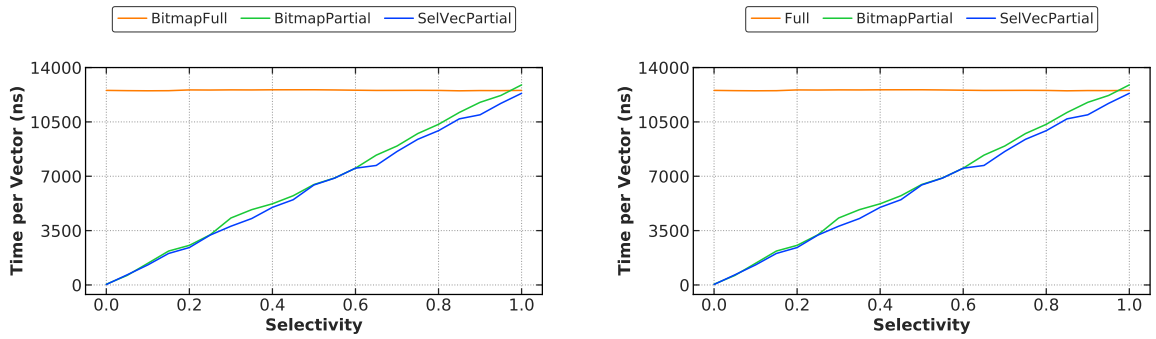
3.1 Non Data-Parallel Core Operations

First, we consider core operations that are not data-parallel. These are the operations where it is impossible to use SIMD instructions to process multiple tuples at a time. We experiment on two kinds of such operations: those that operate on variable-length data (e.g., string operations) and those that use instructions without a SIMD counterpart (e.g., integer division, modulo).

3.1.1 Variable Length Data

We first analyze the performance of operations on variable-length data, as is often the case for string operations (e.g., string comparison, sub-string). Let us consider the factors in Figure 2.9. Here, the core operation executes a variable number of instructions depending, for example, on the length of the input strings. As we show in the next experiment, it dominates the iteration logic time, even for short strings. Because data parallelism is not available, the core operation time per tuple is the same across all strategies. The principal factor that differentiates strategies is, therefore, the number of tuples processed, which favors Selective Compute strategies over Full Compute ones.

To confirm this intuition, we run a micro-benchmark using simple string operations on short strings. The Update micro-benchmark performs string comparison, and the Map micro-benchmark performs character location. The strings are all short country names (e.g., USA, Senegal, China).



(a) Update operation corresponding to `WHERE col1 % col2 < val`.

(b) Map operation corresponding to `SELECT col1 / col2`.

Figure 3.2: Integer Division.

The purpose is to show that the number of tuples processed is the principal performance factor, even for relatively cheap variable-length operations.

The results are shown in Figure 3.1. We see that Full Compute (with BitmapFull or Full) consistently performs worst because it processes more tuples, except at full selectivity. At full selectivity, Full Compute and Selective Compute process the same number of tuples, but the former slightly benefits from a simpler iteration logic. We also see that the performance of SelVecPartial is similar to that of BitmapPartial, despite the former's simpler iteration logic, meaning that the core operation is the dominating cost.

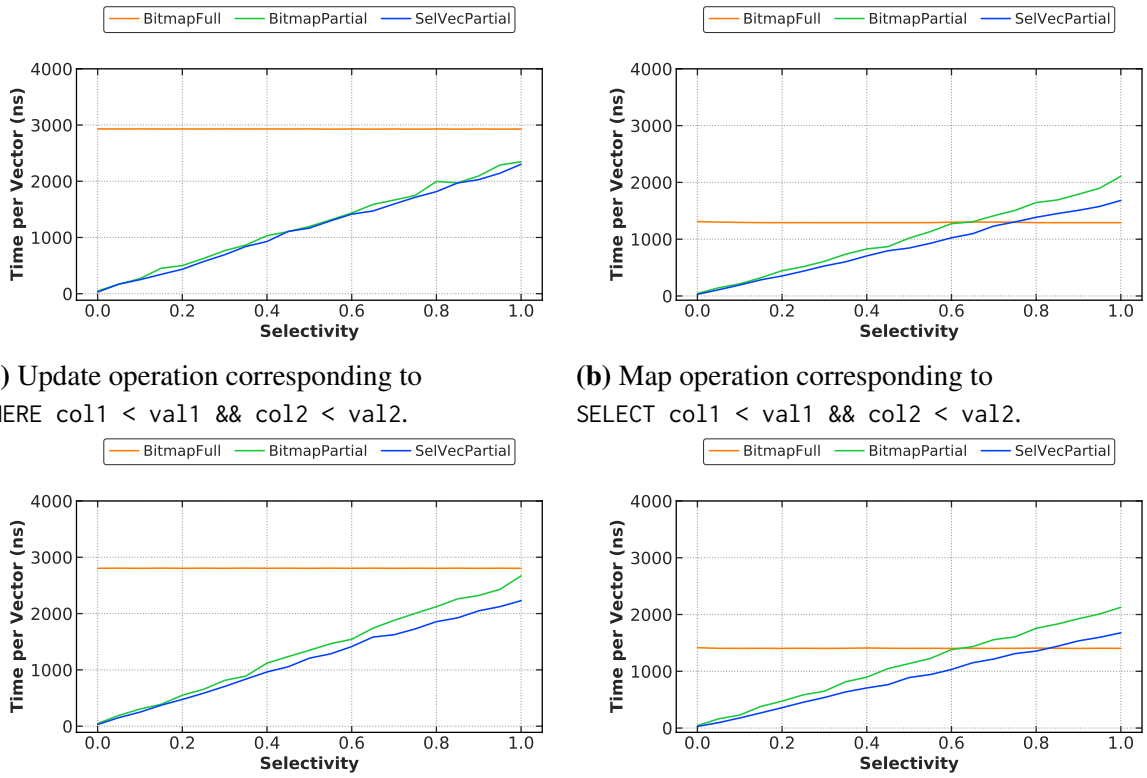
The optimal strategy in this scenario is SelVecPartial. It processes fewer tuples than Full Compute strategies and has a slightly cheaper iteration logic than BitmapPartial (though the performance difference is small). Full Compute is only marginally competitive at full selectivity; its benefits are mostly negligible; the more expansive the core operation, the worse Full Compute becomes.

3.1.2 Integer Division

Second, we analyze the integer division instruction as it does not have a SIMD counterpart. Let us consider the factors in Figure 2.9. Integer division is so expansive that the iteration logic time is negligible compared to the core operation time. Because data parallelism is not available, its cost per tuple is the same across all strategies. Once again, the main factor that differentiates strategies is the number of tuples processed, which favors Selective Compute.

The micro-benchmarks of this section evaluate an Update and a Map primitive that both contain an integer division. The results are shown in Figure 3.2. The explanation of its results is similar to Figure 3.1's: the cost of the integer division and the unavailability of data-parallelism make Full Compute inefficient. SelVecPartial is, once again, the optimal strategy, but only has a slight edge over BitmapPartial due to the former's simpler iteration logic.

We conclude that the number of tuples processed in the dominant performance factor for non-data-parallel primitives, making Full Compute impractical. Because of the small impact of iteration logic time, SelVecPartial has a small performance edge over BitmapPartial, but developers can choose either representation without much affecting performance.



(a) Update operation corresponding to `WHERE col1 < val1 && col2 < val2.`

(b) Map operation corresponding to `SELECT col1 < val1 && col2 < val2.`

(c) Update operation corresponding to `WHERE col1 < val1 || col2 < val2.`

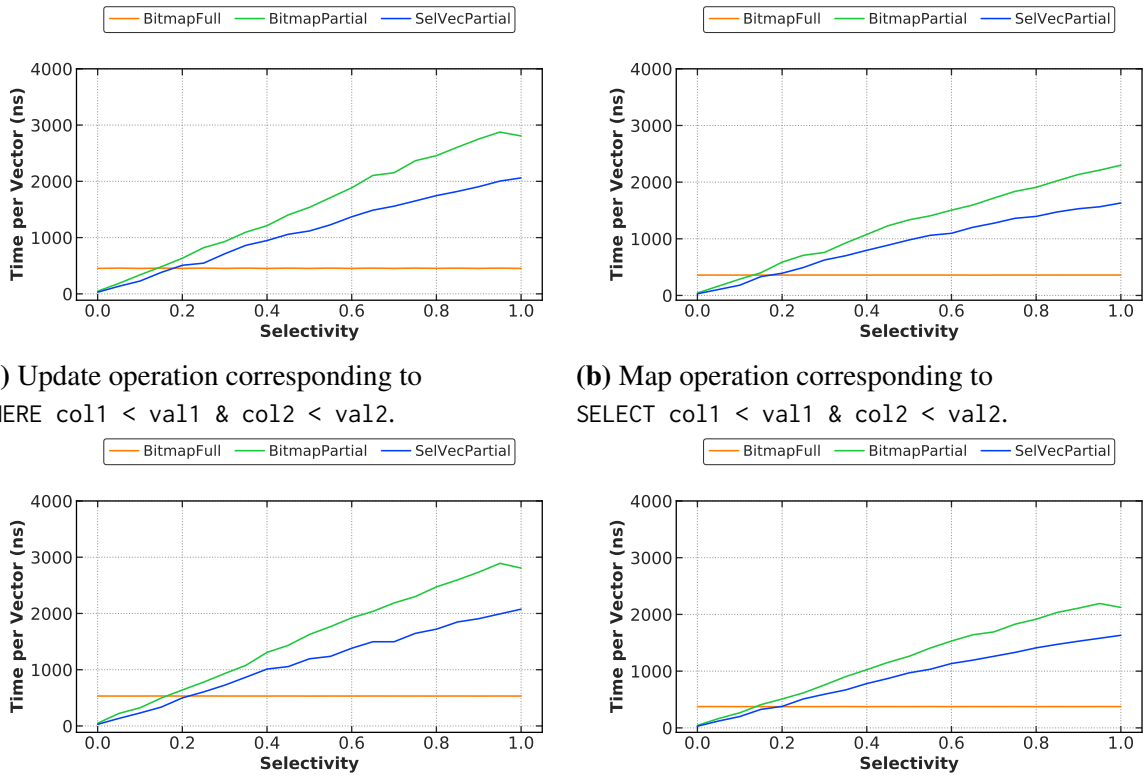
(d) Map operation corresponding to `SELECT col1 < val1 || col2 < val2.`

Figure 3.3: Branching Operations – Each operation contains one branch because of boolean short-circuiting.

3.2 Inefficient Data Parallelism

To study the effect of inefficient use of SIMD instructions, we now consider core operations with branching code. Chapter 4 will discuss primitives that contain pointer manipulation and several memory accesses. SIMD strategies (e.g., BitmapFull) benefit from processing multiple tuples at a time, but suffer from executing more code than SISD strategies (e.g., SelVecPartial). Their core operation time per tuple may even decrease depending on the branching structure (e.g., having to execute all branches of a switch statement). Thus, if Full Compute can outperform Selective Compute at all, it will do so at the highest selectivities because the small reduction in time spent per tuple cannot compensate for the higher number of tuples processed. Among SISD strategies, SelVecPartial will outperform BitmapPartial due to its simple iteration logic.

To determine the impact of a single branch within the core operation, we implemented primitives that perform logical and/or operations. Due to boolean short-circuiting, these primitives will contain exactly one branch. We have confirmed that the compiler manages to auto-vectorize Full Compute strategies. The results are shown in Figure 3.3. We can see that Full Compute is either always the worst strategy (e.g., in Figure 3.3c), or only competitive at high selectivities ($>= 0.85$ in Figure 3.3d). To confirm that branching is indeed responsible for the poor perfor-



(a) Update operation corresponding to `WHERE col1 < val1 & col2 < val2.`

(b) Map operation corresponding to `SELECT col1 < val1 & col2 < val2.`

(c) Update operation corresponding to `WHERE col1 < val1 | col2 < val2.`

(d) Map operation corresponding to `SELECT col1 < val1 | col2 < val2.`

Figure 3.4: Bitwise Operations – The increase in performance compared to Figure 3.3 is due to the lack of branching.

mance of Full Compute, we implemented branch-less primitives with bitwise and/or operations rather than logical ones. The results are in Figure 3.4. Full Compute now outperforms Selective Compute for selectivities $<= 0.2$, confirming that branching code significantly hurts SIMD strategies' performance.

In conclusion, though Full Compute can sometimes be competitive at the highest selectivities, we recommend, for the sake of simplicity, the SelVecPartial strategy.

3.3 Straight-Line and Data-Parallel Core Operations

We now consider core operations with Straight-line and data-parallel (SLDP) code, i.e., non-branching code that can efficiently leverage SIMD instructions. Primitives that only perform arithmetic (without integer division) and bitwise instructions fall under this category. Let us consider the equation in Section 2.4 to analyze the performance of each strategy in this scenario. Full Compute strategies (e.g., BitmapFull, Full) can leverage SIMD instructions to significantly reduce the iteration logic and core operation time per tuple. AVX512 [22] integer addition instructions, for example, performs 8 parallel 64-bit additions in a single cycle. We, therefore,

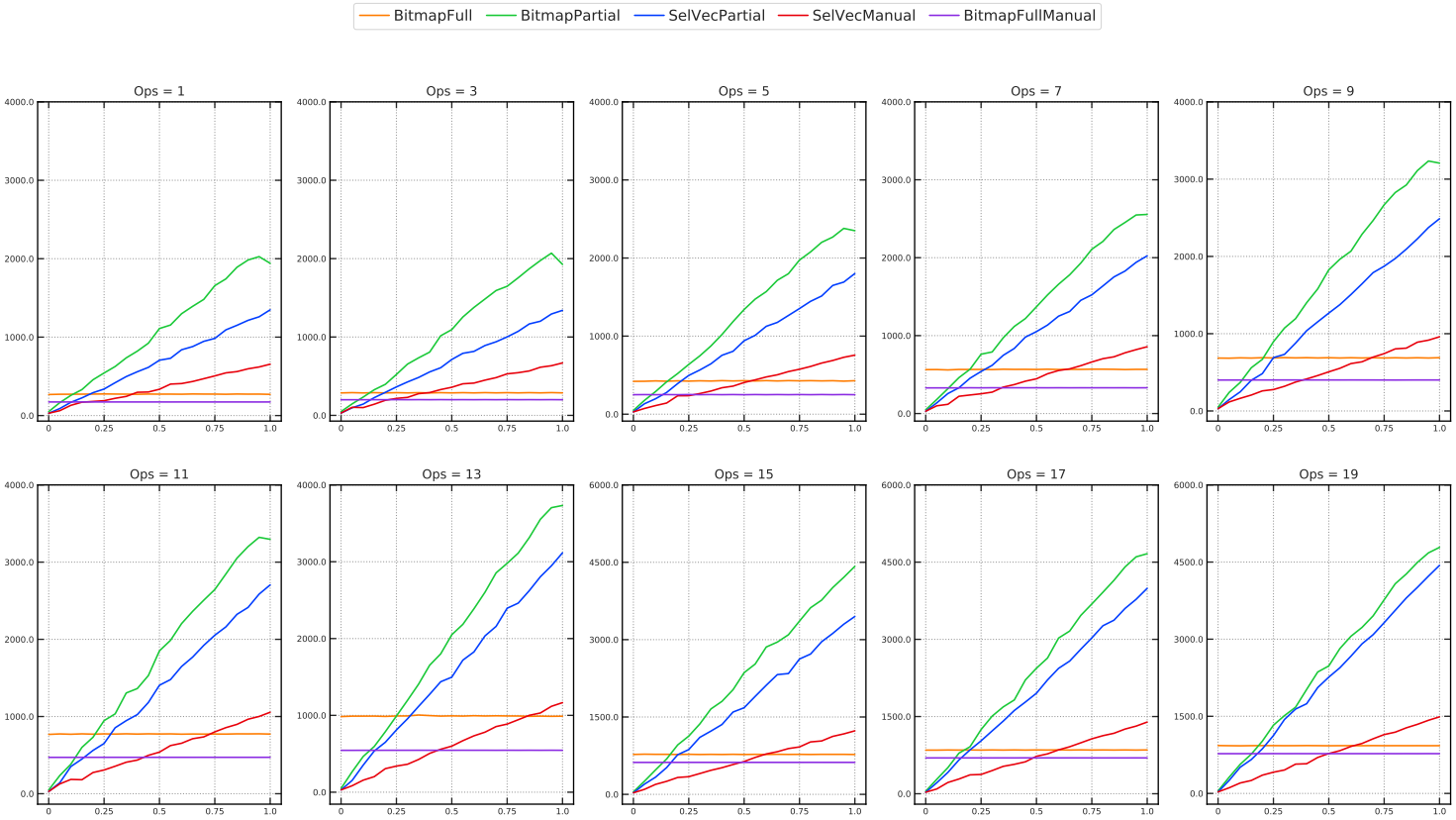


Figure 3.5: SNBS Update Operations – For Updates, SelVecManual relies on the `compress_store` SIMD instruction rather than on `scatter`, which reduces its overhead.

expect Selective Compute strategies to only be competitive when the selectivity is very low, meaning that Full Compute processes far more tuples. SelVecManual is the exception: it is a Selective Compute strategy that uses SIMD instructions. Its gain in iteration logic time is, however, not as high as that of Full Compute strategies because it uses a `gather` instruction [22] to collect the elements at the selected indices (shown in Figure 2.8c). The higher the core operations time per tuple, the less important the `gather` overhead because iteration logic time becomes more and more insignificant. Thus, we expect SelVecManual to be competitive with Full Compute at medium or below selectivities depending on the core operation. The next experiments will quantify the differences between each strategy.

We implemented Update and Map primitives in which the core operation contains single-cycle arithmetic and bitwise instructions. We progressively increase the number of such instructions to increase the core operations time per tuple and quantify the selectivity thresholds above which Full Compute strategies become faster than Selective Compute ones. The results are shown in Figures 3.5 and 3.6 (for Updates and Maps). The selectivity thresholds for each graph are respectively in Figures 3.7a and 3.7b. Let us analyze the results in detail:

- In all experiments, SelVecPartial performs better than BitmapPartial. As explained in the previous chapter, this is only due to a simpler iteration logic.
- SelVecManual is always the best Selective Compute strategy because it uses data-parallelism

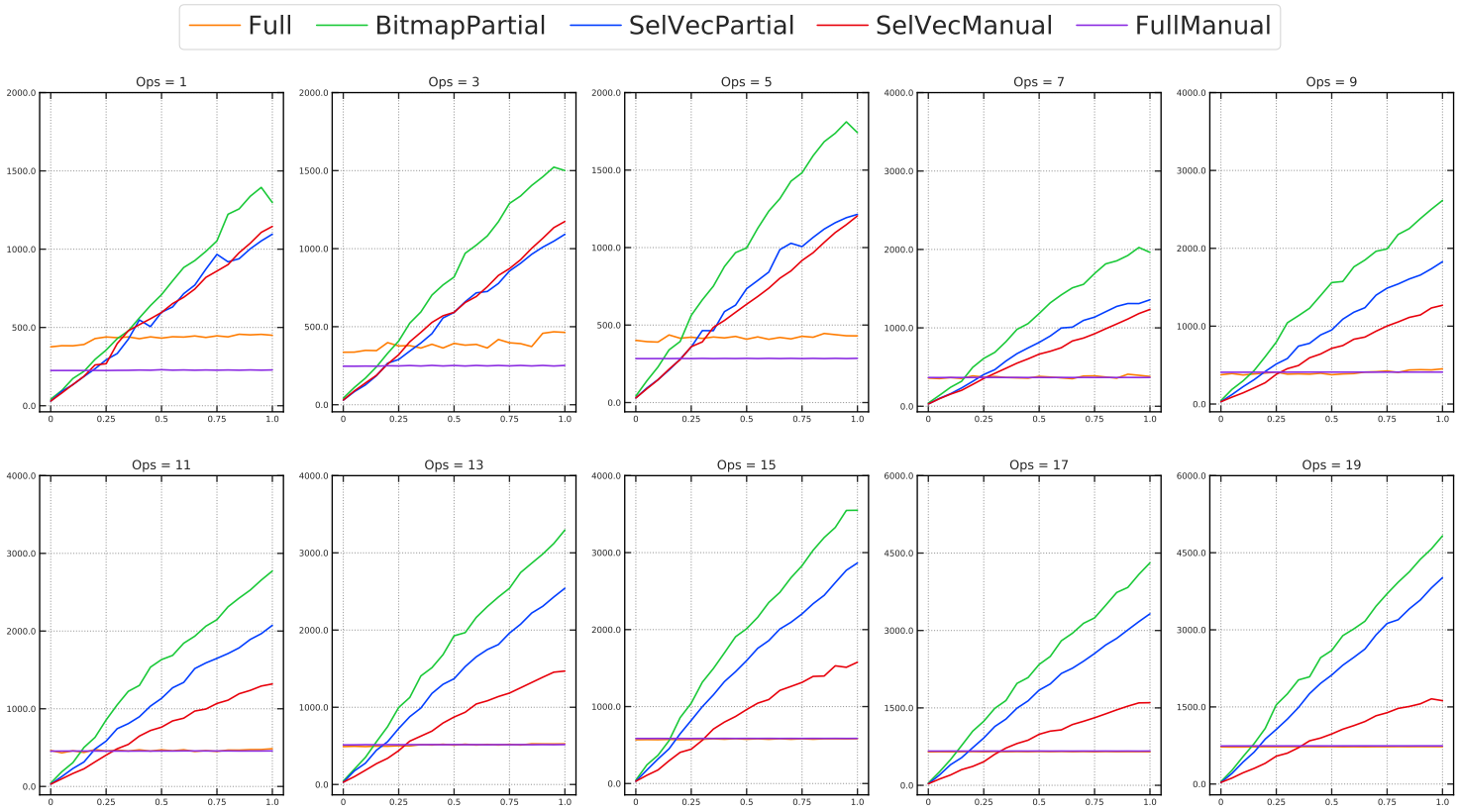
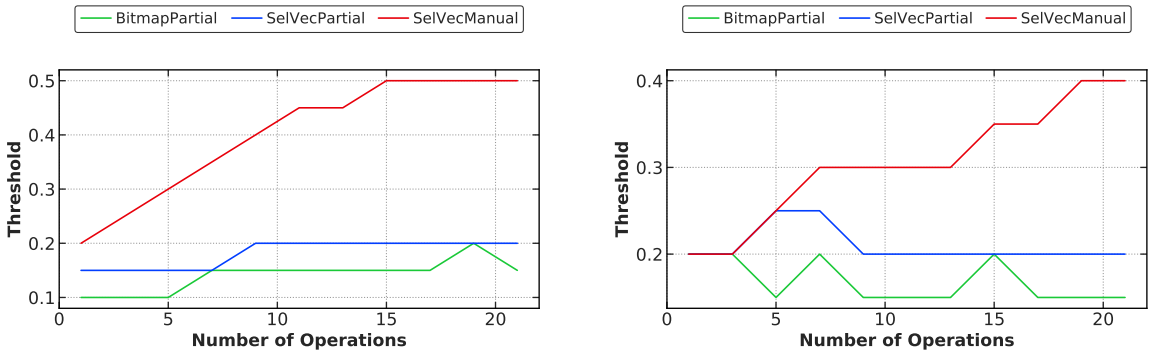


Figure 3.6: SNBS Map Operations – For Maps, SelVecManual relies on the `scatter` SIMD instruction, which explains why it initially does not perform better than SelVecPartial.



(a) Update thresholds between BitmapFullManual and Selective Compute strategies.

(b) Map thresholds between FullManual and Selective Compute strategies.

.

Figure 3.7: Mixed Compute Thresholds – The Full Compute strategies are BitmapFullManual and FullManual because compiler auto-vectorization does not consistently use AVX512 registers.

to reduce the time spent per tuple.

- Manual Vectorization (with FullManual and BitmapFullManual) sometimes performs much

better than Auto-Vectorization (with Full and BitmapFull). For example, in Figure 3.5's graph for `NumOps=13`, BitmapFullManual is 1.8 times faster than BitmapFull. With further investigation of the generated code, we discovered that the compiler is overly conservative when it comes to using AVX512. AVX512 registers result in decreased CPU frequency [26], so the compiler is careful not always to use them. It often uses AVX2 registers instead. We, on the other hand, always use AVX512.

- The thresholds for SelVecPartial and BitmapPartial, shown in Figures 3.7a and 3.7b, prove that these strategies are only competitive with Full Compute at low selectivities ($<= 0.2$) because, as explained above, they do not use SIMD instructions.
- The threshold for SelVecManual, on the other hand, increases with the number of operations because its iteration logic overhead becomes more and more insignificant as the core operation time increases. SelVecManual thus benefits from data-parallelism while processing fewer tuples than Full Compute strategies. The iteration logic difference is never entirely negligible, though; the threshold is medium at best (0.5 at 15 operations in Figure 3.7a).

This analysis shows the importance of Mixed Compute: we should use Selective Compute below the selectivity thresholds, and Full Compute otherwise. The next section details our implementation of Mixed Compute.

3.4 Implementing Mixed Compute

The previous section showed that for a given SLDP primitive, there is a threshold below which SelVecManual is optimal and above which BitmapFullManual and FullManual are optimal for Updates and Maps respectively. To find this threshold, we can run a micro-benchmark similar to the one in Figure 3.7. The lowest selectivity at which the runtime of Full Compute strategies exceeds that of Selective Compute strategies represents the Mixed Compute threshold for a given primitive. In summary, the optimal SLDP strategy is as follows:

- For Maps: Full or FullManual when the selectivity is above the threshold, and SelVecManual when the selectivity is below the threshold.
- For Updates: BitmapFull or BitmapFullManual when the selectivity is above the threshold, and SelVecManual when the selectivity is below the threshold. Note that the BitmapFull and SelVecManual have different filter representations. There is a small cost associated with the conversion from Bitmap to SelVec, but, in practice, other operations always follow an update (e.g., there is a projection after a scan filter). The subsequent operations often amortize the conversion cost. For small queries, in which the amortization does not neutralize the conversion cost, BitmapPartial can slightly outperform SelVecManual because it maintains the Bitmap representation. Nonetheless, as we show in Chapter 4, its gains are mostly negligible.

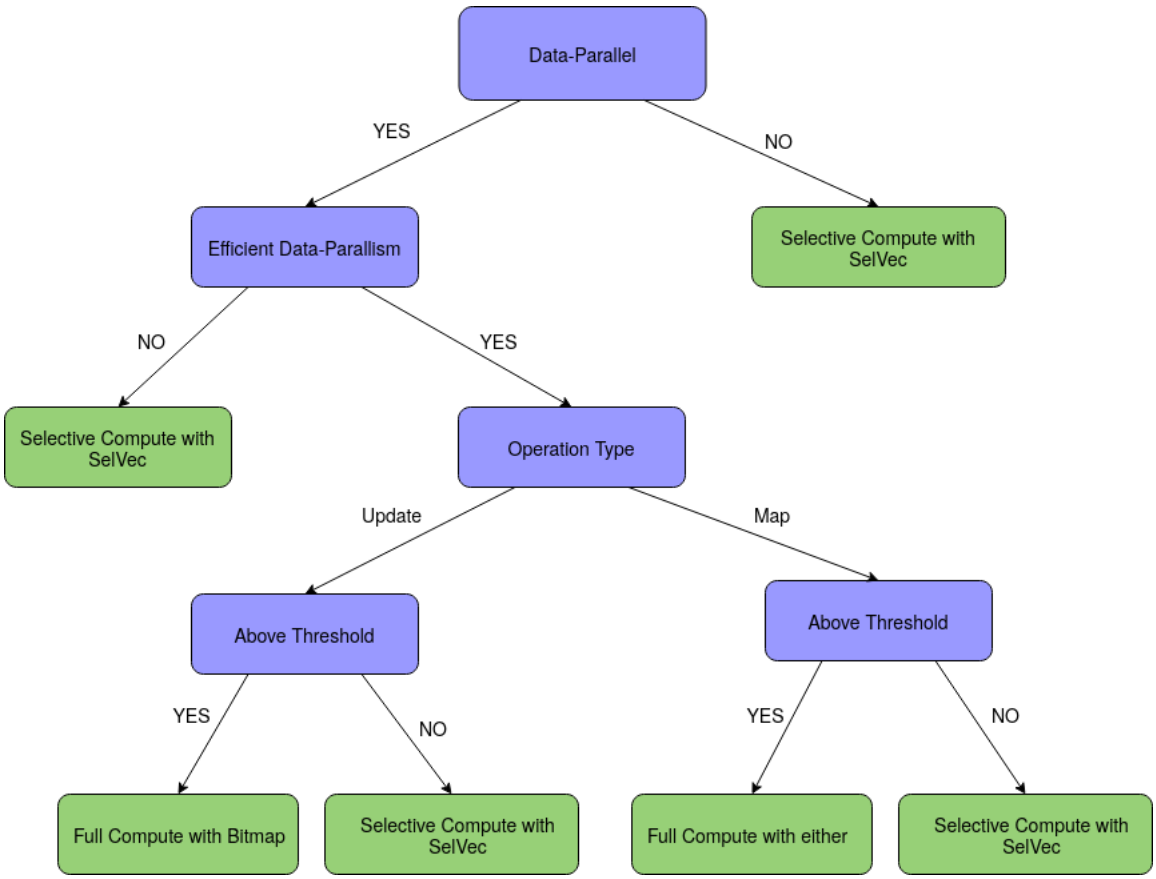


Figure 3.8: The Summary Decision Tree – It summarizes the results in this chapter.

3.5 Vectorization Decision Tree

The decision tree in Figure 3.8 summarizes the results of this chapter. It uses the experiments above to derive the optimal implementation strategy depending on the core operation and the input selectivity of a given primitive. The **Data-Parallel** node, the **Effecient Data-Parallelism** respectively use the results in Section 3.1 and Section 3.2 that designated SelVecPartial as the optimal strategy. The **Operation Type** and **Above Threshold** nodes use the results in Section 3.3 and Section 3.4 that designed a Mixed Compute strategies for each primitive type (Update or Map).

The tree assumes that the system developer implements all representations and compute strategies. The total number of variants of each primitive will be extensive, which constitutes a costly engineering endeavor. To remedy this cost, we note that the BitmapPartial strategy does not perform significantly worse than SelVecPartial or SelVecManual at most selectivities. Developers can, thus, focus on focus on the Bitmap representation without losing too much performance. Switching between Full and Selective Compute is critical, though; the performance gain is too significant.

Chapter 4

Experimental Evaluation

We now evaluate the benefits of the decision tree in Chapter 3. For this evaluation, we implemented the strategies for a subset of queries from the TPCH benchmark, a decision support system workload that simulates an OLAP environment [37]. We use a scale factor of 10 ($\sim$10 GB). The hardware setup is unchanged from the previous chapter. We use the columnar storage engine of the NoisePage DBMS [17]. The string columns are all dictionary compressed for faster processing [14]. Whenever the compiler (clang [2]) failed to SIMD vectorize a primitive that should be data-parallel or used AVX2 instead of AVX512, we hand-wrote SIMD code using AVX512 intrinsics [22].

Each query contains primitive with side-effects (e.g., hash table insertion), and primitives without side-effects (e.g., multiplication of two columns). Given that this thesis focuses on side-effect-free primitives, we report two metrics for each query: the running time of the whole query and the running time of the side-effect free primitives within the query. The output of each query is materialized in memory and printed to the standard output; no network transfer occurs.

4.1 Q1: High Selectivity SLDP primitives

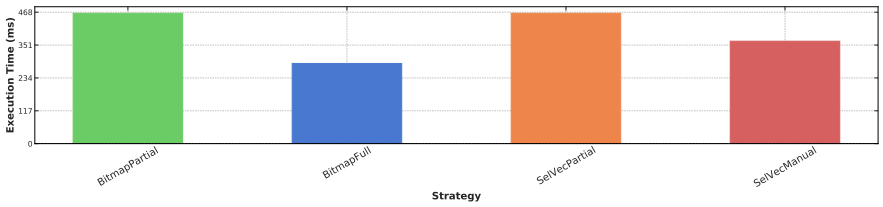
Figure 4.1 shows the SQL code of the Q1 query. It contains a single SLDP filter in its `WHERE` clause; its selectivity is consistently above 0.95 for the vectors from the `lineitem` table. Its `SELECT` clause contains multiple arithmetic SLDP operations followed by side-effect-full aggregation and order-by operators that dominate the running time. Before the aggregation, there is an SLDP primitive that hashes the `GROUP BY` keys. Because the selectivity is high and the query is SLDP-heavy, our decision tree favors SIMD vectorized Full Compute strategies. Besides, switching between strategies is not required.

The results for the strategies we implemented are shown in Figure 4.2. We omit the Mixed strategies' measurements because no switching ever occurs in the query plan; they have the same running times as BitmapFull. Figure 4.2a shows the running time of the primitives we optimized. As predicted by our decision tree, the SIMD Full Compute strategy, BitmapFull, best optimizes these primitives. It outperforms BitmapPartial and SelVecPartial (which have SISD primitives) by $1.6\times$, and SelVecManual (which has manually written SIMD primitives) by $1.3\times$.

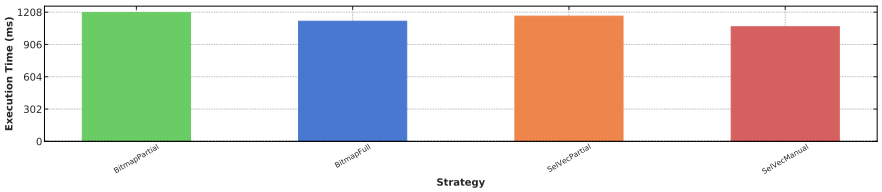
The total running times (Figure 4.2b) are mostly the same for all queries for two reasons.

```
SELECT
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
FROM
        lineitem
WHERE
        l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY
        l_returnflag,
        l_linestatus
ORDER BY
        l_returnflag,
        l_linestatus
```

Figure 4.1: The TPCH Q1 query.



(a) Running time of side-effect free primitives.



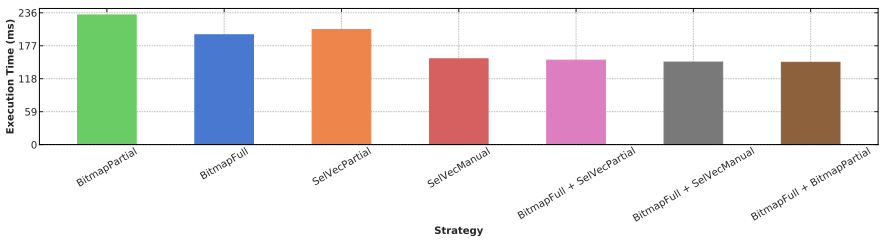
(b) Total running time.

Figure 4.2: Q1 Performance

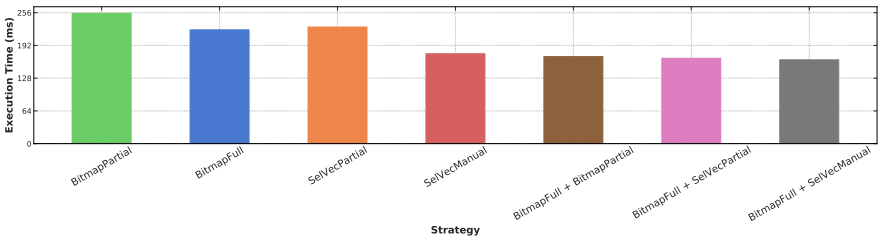
First, the side-effect-full aggregation, which we do not try to optimize, is the query's dominant component. Second, Intel's implementation of AVX512 instructions decreased CPU speed. This effect explains why clang limits their usage to avoid performance degradations [26]. Compare, for example, the BitmapPartial and BitmapFull strategies. They both have the same code for the side-effect-full primitives, but BitmapFull performs them in 838ms, whereas BitmapPartial performs them in 739ms. Naively avoiding AVX512 instructions is not the solution; AVX2 uses vectors that have half the width of AVX512's, which slows down the execution of SLDP primitives. Careful analysis is thus required to balance the benefits and disadvantages of AVX512 instructions. This analysis is left as future work.

```
SELECT
        SUM(l_extendedprice * l_discount) AS revenue
FROM
        lineitem
WHERE
        l_shipdate >= date '1994-01-01'
        and l_shipdate < date '1995-01-01'
        and l_discount between 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
```

Figure 4.3: The TPCH Q6 query.



(a) Running time of side-effect free primitives.



(b) Total running time.

Figure 4.4: Q6 Performance. The + notation indicates mixed strategies.

4.2 Q6: Mixed Selectivity SLDP Primitives

Figure 4.3 shows the SQL code of the Q6 query. Its `WHERE` contains five SLDP filters, and its `SELECT` one arithmetic SLDP projection followed by an aggregation. For all vectors, the second filter leads to a selectivity below 0.15, triggering a threshold-based switch. Thus, our decision tree favors a mixed strategy: BitmapFull at high selectivities, and SelVecManual at low selectivities.

The results are shown in Figure 4.4. Figure 4.4a shows the running time of the primitives we optimized. This graph confirms the necessity of the caveat we raised in Section 3.4: the cost of converting from Bitmaps to SelVecs makes BitmapFull+SelVecManual slightly more expansive than BitmapFull+BitmapPartial by 0.3%, even though SelVecManual performs better than BitmapPartial on the individual primitives in this query. This difference is insignificant, so our decision tree does not take it into account. The BitmapFull+SelVecPartial strategy also suffers from the conversion overhead, but the difference only amounts to 2%. All mixed strategies perform better than non-mixed ones. BitmapFull+BitmapPartial and BitmapFull+SelVecManual

```
SELECT
        o_orderpriority,
        count(*) as order_count
FROM
        orders
WHERE
        o_orderdate >= date '1993-07-01'
        AND o_orderdate < date '1993-10-01'
        AND exists (
                SELECT
                        *
                FROM
                        lineitem
                WHERE
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
GROUP BY
        o_orderpriority
ORDER BY
        o_orderpriority
```

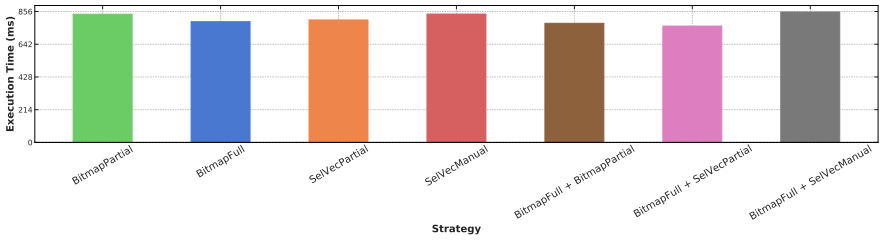
Figure 4.5: The TPCH Q4 query.

are faster than BitmapFull by $1.3\times$ because Full Compute becomes wasteful after the selectivity drops below 0.15. The SIMD SelVecManual strategy is slower by $1.04\times$ because it is only suboptimal in the first two filters of the `WHERE` clause. The SISD Selective Compute strategies BitmapPartial and SelVec Partial are slower by $1.6\times$ and $1.4\times$ because they do not take advantage SIMD instructions for SLDP operations.

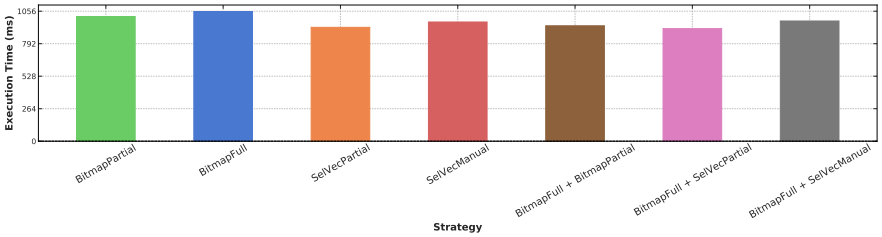
Figure 4.4b shows the total running time that includes the side-effect-full aggregation primitive. This primitive only accounts for a small portion of the running time, so we do not observe the slowdown caused by AVX512 registers. In SLDP-dominated queries like Q6, the benefits of AVX512 outweigh its disadvantages.

4.3 Q4: Low Selectivity, Inefficient Data Parallelism

Figure 4.5 shows the SQL code of the Q4 query. It is a join of two tables (`lineitem` and `orders`) followed an aggregation and an order-by operator. The join's left side (from the `orders` table) has two SLDP filters that reduce the selectivity to 0.3 and 0.1, respectively, and an SLDP hashing primitive before insertion into the join hash table. Its right side (from the `lineitem` table) has a filter with a selectivity > 0.6 and a join probe that contains the multiple primitives shown in Figure 2.2. Although the probe's hashing primitive is an SLDP operation, its other primitives contain multiple indirect lookups (e.g., accessing hash table entries or the keys within those entries for exact comparison). Because these operations are called in a loop to find all potential matches during the join, they constitute the bulk of the running time. Unfortunately, their data-parallel versions are inefficient. For example, the comparison primitive, which we manually implemented for the SelVecManual strategy, contains three `gather` instructions: one to collect the keys from the right side of the join, one to collect the hash table entries, and one to collect the keys within the hash table entries for comparison. SIMD instructions can cause performance



(a) Running time of side-effect free primitives.



(b) Total running time.

Figure 4.6: Q4 Performance

degradations in these primitives. Thus, we predict that we should use BitmapFull to perform the SLDP filters and hashing, then switch to using SelVecPartial, not the SIMD SelVecManual, for the complex primitives within the join probe.

The results are shown in Figure 4.6. Figure 4.6a shows the running time of the primitives we optimized. Our predicted strategy, BitmapFull+SelVecPartial, is indeed the optimal one, closely followed by BitmapFull+BitmapPartial, which also switches from SIMD to SISD code for complex primitives. Notice the performance degradation caused by SIMD vectorization, as mentioned in Section 3.2. Unlike the previous queries' results, BitmapFull+SelVecPartial performs better than the BitmapFull+SelVecManual by $1.1\times$ because the latter relies on SIMD vectorization for complex primitives. BitmapFull is also slightly worse (by $1.04\times$). Its performance degradation is attenuated by the fact that it requires fewer `gather` instructions than SelVecManual (e.g., one instead of three in the comparison primitive), relying instead on faster `vector_load` instructions to load all elements in a vector, whether they are selected or not. The SISD Selective Compute strategies BitmapPartial and SelVecPartial are, respectively, $1.1\times$ and $1.05\times$ worse because they do not optimize SLDP primitives.

The results in Figure 4.6a mostly reflect those in Figure 4.2a. AVX512 registers, once again, cause a slowdown in SIMD implementations, causing, for example, BitmapFull to become slower than BitmapPartial even though they share the same code for side-effect-full primitives. Here, the disadvantages of AVX512 outweigh its benefits on the full query.

4.4 Summary

Our analysis shows that our results in Chapter 3 provide us with the best implementation for each query. The downside is that it requires a wide variety of variants for each primitive because any

strategy can be optimal under specific scenarios. We see, however, that Bitmap only strategies (e.g., BitmapFull, or BitmapFull+BitmapPartial) perform nearly as well as the optimal ones in the queries above. Developers can thus discard the SelVec representation to save engineering effort without losing too much performance.

Chapter 5

Related Works

5.1 Optimizing Analytical Queries for DBMSs

Besides the Vectorization and Materialization model introduced in Chapter 1, much work has been dedicated to optimizing analytical queries on column store DBMSs. Abadi et al. studied the effect of materialization strategies (i.e. late or early materialization) on query performance [13]. All experiments in this paper use late materialization and rely on filters to keep track of valid tuples. Nonetheless, Abadi et al. found that the optimal materialization strategy depends on selectivity and on the exact operations performed on tuples [13]. Stonebraker et al. studied compression as way to optimize both data storage and query execution in column stores [36]. Abadi et al. performed an extensive evaluation of various compressions schemes [14]. The experiments in Chapter 4 rely only on dictionary compression.

Researchers have also used SIMD vectorization to optimize relational algorithms. Zhou and Ross optimized scans and nested loop joins using techniques most similar to our BitmapFull-Manual implementation. Ross further applied SIMD vectorization hash tables probes [32]. Inoue et al. and Chhugani et al. proposed sorting algorithms using SIMD instructions [16, 20]. The relational order-by operators, sort-merge joins, and sorting aggregations can all use these efficient sorting algorithms.

Raman et al. and Willhalm et al. combined this body of work and built full commercial DBMSs using column stores with compression and SIMD vectorization techniques to optimize table scans [31, 38]. Polychroniou and Ross further built a system using SIMD vectorization on all relational operators [29]. These systems make a static filter representation decision: Bitmaps for table scans, and SelVecs for other operations.

Along with the Vectorization Model, the current state-of-the-art execution technique is the Data-Centric Query Compilation technique pioneered by Neumann [28]. This technique splits a given query plan into several pipelines and compiles them individually. A pipeline is a sequence of relational operators that do not materialize their result set followed by a pipeline-breaker – an operator that does materialize its result set (e.g., to build a hash table or sort tuples). This approach's benefits are: (1) eliminate interpretation overhead by compiling specialized code and (2) maximize cache locality by keeping tuples in the registers or the L1 cache for entire pipelines. Our system uses the relaxed operator fusion technique [25], which combines data-centric code

generation with the Vectorization Model. It generates code that operates on vectors of tuples, thereby eliminating the interpretation overhead and optimizing cache locality while maintaining all the Vectorization Model's benefits.

5.2 Primitive Optimization

There has been previous work dedicated to optimizing primitives in the Vectorization Model. They mostly differ from this thesis in that they do not consider the impact of filter representation in their performance. We now discuss this work and how they relate to our proposed method.

Sompolski et al. compared the cost of vectorization to data-centric query compilation [34] and combined the two approaches to generate new vectorized primitive at runtime, which is the method we use in Section 3.3. Using only the SelVec representation, the authors compare vectorized primitives' performance under various compute strategies (e.g., Full versus Selective Compute, Conditional versus Unconditional Store). They, however, do not provide a way to switch between these strategies.

Kersten et al. performs a similar analysis for whole queries in addition to individual primitives [23]. They developed a SIMD implementation of primitives using Selection Vectors, just like our SelVecManual strategy. Like ours, their implementation was effective on compute-heavy queries (e.g., Q6 in Section 4.2) and ineffective on memory access-heavy queries (e.g., Q4 in Section 4.3).

Răducanu et al. recognized the importance of switching compute strategy at run-time [33]. For each operation, the authors implement several *flavors* (i.e., different ways to perform the same operations). They then use reinforcement learning (RL) to switch between the best *flavors* dynamically at runtime. This approach is more general than our micro-benchmarks because it takes into account changes in system load. SelVec is the only representation considered, limiting the range of adaptivity on Update tasks. Besides, our decision tree provides more explainability than RL techniques. As future work, we could employ a similarly dynamic strategy to determine our thresholds.

Chapter 6

Conclusion

This work analyzed the impact of filter representation (i.e., Bitmap vs. SelVec) and compute strategy (i.e., Full vs. Selective Compute) on the performance of the vectorized primitives in an in-memory analytical DBMS. We identified the factors that influence performance: number of tuples processed, iteration logic, and core operation time per tuple. We explained how each combination of representation and compute strategy balances between these three factors. Full Compute has the cheapest iteration logic, processes all tuples, but spends less time on each tuple when SIMD vectorization is possible. Full Compute is, however, only available with Bitmaps on Update primitives. Selective Compute with SelVecs has a cheaper iteration logic than Selective Compute with Bitmaps, and is more amenable to SIMD vectorization. We confirmed these observations with several micro-benchmarks, and created a decision tree to help systems developers find the optimal implementation of arbitrary side-effect-free primitives. Finally, we showcased the benefits of our analysis on OLAP queries with multiple primitives and consistently achieved the best performance. Our performance gains over the best techniques that do not adapt filter representation and compute strategy can be up to $1.3\times$.

Bibliography

[1] Amazon Aurora - Relational Database Built for the Cloud. https://aws.amazon.com/rds/aurora/, 2020. 1

[2] Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2020. 3, 4

[3] Historical Memory Prices 1957+. https://jcmit.net/memoryprice.htm, 2020. 1

[4] DuckDB. https://duckdb.org/, 2020. 1

[5] MySQL. https://www.mysql.com/, 2020. 1

[6] Number Every Programmer Should Know by Year. https://colin-scott.github.io/personal_website/research/interactive_latency.html, 2020. 1

[7] PostgreSQL. https://www.postgresql.org/, 2020. 1

[8] Amazon Redshift - Cloud Data Warehouse. https://aws.amazon.com/redshift/, 2020. 1

[9] SAP HANA - In-Memory Database. https://www.sap.com/products/hana.html, 2020. 1

[10] Vertica - Big Data Analytics On-Premises, in the Cloud, or on Hadoop. https://www.vertica.com/, 2020. 1

[11] VoltDB. https://www.voltdb.com/, 2020. 1

[12] Daniel Abadi. Query execution in column-oriented database systems. November 2008. 1

[13] Daniel Abadi, Daniel Myers, David DeWitt, and Samuel Madden. Materialization strategies in a column-oriented dbms. pages 466–475, 05 2007. ISBN 1-4244-0803-2. doi: 10.1109/ICDE.2007.367892. 5.1

[14] Daniel J. Abadi, Samuel R. Madden, and Miguel C. Ferreira. Integrating compression and execution in column-oriented database systems. In *In SIGMOD*, pages 671–682, 2006. 4, 5.1

[15] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005. 1

[16] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 1(2): 1313–1324, August 2008. ISSN 2150-8097. doi: 10.14778/1454159.1454171. URL https://doi.org/10.14778/1454159.1454171. 5.1

[17] CMUDB Group. The noisepage system. https://noise.page/, June 2020. 3, 4

[18] Daniel Lemire. Iterating over set bits quickly. https://lemire.me/blog/2018/02/21/iterating-over-set-bits-quickly/, February 2018. 2.8b

[19] G. Graefe. Volcano an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994. ISSN 1041-4347. doi: 10.1109/69.273032. URL https://doi.org/10.1109/69.273032. 1

[20] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. pages 189–198, 10 2007. ISBN 978-0-7695-2944-8. doi: 10.1109/PACT.2007.4336211. 5.1

[21] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, November 2014. ISSN 2150-8097. doi: 10.14778/2735508.2735518. URL https://doi.org/10.14778/2735508.2735518. 2.3

[22] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Octobber 2019. 1.1, 2.2, 2.3, 3.3, 4

[23] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, September 2018. ISSN 2150-8097. doi: 10.14778/3275366.3284966. URL https://doi.org/10.14778/3275366.3284966. 1.2.1, 5.2

[24] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, December 2000. ISSN 1066-8888. doi: 10.1007/s007780000031. URL https://doi.org/10.1007/s007780000031. 1

[25] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, September 2017. ISSN 2150-8097. doi: 10.14778/3151113.3151114. URL https://doi.org/10.14778/3151113.3151114. 5.1

[26] Michael Larabel. Intel tightens up its avx-512 behavior for the llvm clang 10 compiler. https://www.phoronix.com/scan.php?page=news_item&px=LLVM-Clang-10-AVX512-Change, September 2018. 3.3, 4.1

[27] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. 1

[28] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011. ISSN 2150-8097. doi: 10.14778/2002938.2002940. URL https://doi.org/10.14778/2002938.2002940. 5.1

[29] Orestis Polychroniou and Kenneth A. Ross. Towards practical vectorized analytical query engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machin-

ery. ISBN 9781450368018. doi: 10.1145/3329785.3329928. URL https://doi.org/10.1145/3329785.3329928. 1.2.1, 5.1

[30] Prashanth Menon, Amadou Ngom, Todd Mowry, Andy Pavlo. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. April 2020. 2.2

[31] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, August 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536233. URL https://doi.org/10.14778/2536222.2536233. 1.2.1, 5.1

[32] Kenneth A. Ross. Efficient hash probes on modern processors. *2007 IEEE 23rd International Conference on Data Engineering*, pages 1297–1301, 2007. 5.1

[33] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1231–1242, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465292. URL https://doi.org/10.1145/2463676.2465292. 1.2.1, 2.3, 5.2

[34] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, page 33–40, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306584. doi: 10.1145/1995441.1995446. URL https://doi.org/10.1145/1995441.1995446. 1.2.1, 2.3, 5.2

[35] Michael Stonebraker and Ugur Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, page 2–11, USA, 2005. IEEE Computer Society. ISBN 0769522858. doi: 10.1109/ICDE.2005.1. URL https://doi.org/10.1109/ICDE.2005.1. 1

[36] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 553–564. VLDB Endowment, 2005. ISBN 1595931546. 5.1

[37] The Transaction Processing Council. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/, June 2013. 1.2.1, 4

[38] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, August 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687671. URL https://doi.org/10.14778/1687627.1687671. 5.1

[39] Jingren Zhou and Kenneth Ross. Implementing database operations using simd instructions. pages 145–156, 01 2002. doi: 10.1145/564691.564709. 5.1

[40] Miroslaw Zukowski and Peter Boncz. Vectorwise: Beyond column stores. *Journal of Theoretical Biology - J THEOR BIOL*, 35:21–27, 01 2012. 1.2.1