Exploiting SIMD for Complex Numerical Predicates

Dongxiao Song, Shimin Chen*

State Key Laboratory of Computer Architecture
Institute of Computing Technology
Chinese Academy of Sciences
{songdongxiao,chensm}@ict.ac.cn

***Abstract*—We study the use of SIMD instructions to support complex conjunctive numerical predicates. Compared to previous studies, we aim to model more realistic use scenarios, where different data types, different comparison operations, and different predicate types can be mixed in a single filtering clause. Moreover, the evaluation of the predicates on a set of columns can take advantage of multiple processor cores for maximum speed. We find that the diversity of the predicates and the introduction of multiple threads pose significant challenges in modeling and optimizing complex predicates. We propose a code framework based on three alternative SIMD algorithms for conjunctive predicates. Then, we investigate cost models for both single-threaded and multi-threaded evaluation of filtering predicates. Our experimental results on synthetic data show that an optimal SIMD plan can achieve up to 10.4x speedup over the best no SIMD plan, and up to 6.8x speedup over sub-optimal SIMD plans.**

I. Introduction

Filtering is a common operation in both database systems and big data systems. It removes irrelevant data from consideration, and often significantly reduces the amount of data to be processed. As the main memory capacity increases exponentially, main memory database systems/engines are moving to the main stream. All the major relational database vendors now have their own main memory database solutions [5], [1], [4], [7]. Scan-based filtering operations are widely employed to support analytical queries. Unlike disk-oriented database systems, I/Os are no longer the bottleneck for scans in main memory. Instead, it is important to examine and optimize the filtering algorithm in order to more efficiently support scan-based operations [9], [10], [11], [6], [8], [3].

Wider and wider SIMD instructions are becoming available in the mainstream processors. The current Intel Haswell architecture supports 256-bit AVX2 SIMD instructions. Every 256-bit instruction can process four 64-bit types, eight 32-bit types, 16 16-bit types, or 32 8-bit types at a time, thereby significantly improving the performance of the corresponding computation. The next generation mainstream processor architecture from Intel, Skylake, will be equipped with 512-bit SIMD instructions (a.k.a. AVX-512).

In this paper, we are interested in exploiting SIMD instructions for evaluating filtering predicates in main memory. In related work, Ross optimized the implementation of conjunctive predicates with the C programming language, choosing either the logical AND (&) or the branching AND (&&) C operators to connect predicates [9]. Sitaridi and Ross studied the evaluation of conjunctive predicates on GPUs [10]. Recently, Polychroniou et al. studied SIMD optimizations for relational operations [8]. The work exploits SIMD memory gather and scatter operations mainly on the Xeon Phi MIC processor. However, it considers only a particular type of predicates in the selective scan study. Moreover, several recent studies on scanning compressed columnar data consider the use of SIMD for decompression and/or predicate evaluation [11], [6], [3]. However, since their focus is on the structure of the compressed data, they do not provide systematic studies of various predicate conditions.

Compared to previous work, we would like to model and support realistic predicate use cases that are more complex. We study three representative types of predicates, six common comparison operators, and six numerical data types. We consider the combination of three different SIMD algorithms. We propose an algorithm framework to capture the essence of the three algorithms, where a plan consists of one or multiple steps, a step consists of one or multiple functions, and a function evaluates one or multiple predicates. This framework allows a large number of alternative plans for a given conjunctive clause. Moreover, we propose a linear cost model for the single-threaded execution of a plan. Based on the linear model, we consider the effect of memory bandwidth saturation and propose a cost model for multi-threaded executions. Finally, we describe two algorithms that search the plan space for the optimal plan.

The rest of the paper is organized as follows. Section II describes the algorithm framework and the code generation process. Then, Section III presents our cost models and optimizes conjunctive predicates. Preliminary experimental results are presented inline in this section. Finally, Section IV concludes the paper and discusses future work.

II. Supporting Complex Numerical Predicates

In this paper, we focus on conjunctive numerical predicates[1]. That is, the where clause in a SQL select statement consists of a conjunction of multiple predicates as follows:

$$\begin{array}{lcl} clause & = & pred \mid pred \text{ AND } clause \\ pred & = & col\ op\ val \mid col\ op\ col \mid col \text{ IN } [valuelist] \\ op & = & < \mid \leq \mid = \mid <> \mid \geq \mid > \end{array}$$

We consider three typical types of predicates: (i) a comparison between a column and a constant value, (ii) a comparison between two columns, and (iii) a test to see if a column is in a

*Corresponding author

[1]Note that filtering clauses that contain disjunctions can be canonically written as disjunctions of conjunctive sub-clauses. Our solution can be applied to compute each conjunctive sub-clause. The full result is a union of the results from each sub-clause.

given list of constant values. There are six types of comparison operations: $<$, $\leq$, $=$, $<>$, $\geq$, and $>$. The columns and the values are 8-bit, 16-bit, 32-bit, 64-bit integers, single precision or double precision floating point numbers.

Compared to previous studies, there are several distinct features of our study. First, we model more realistic use scenarios where different predicate types, different comparison types, and different data types can be mixed in a single clause. Every predicate type requires its own SIMD implementation. Different data types are often of different sizes with different memory access and comparison costs. Second, we consider the SIMD algorithms for multiple conjunctive predicates. We propose an algorithm framework that combines three alternative algorithms into a set of potential execution plans. We perform template based code generation and cost-based optimization to choose the best plan. Finally, we consider and model multi-threaded implementation of the SIMD based computation.

In the following, we first motivate and describe three alternative SIMD based algorithms to compute multiple predicates in Section II-A. Then, we combine the three algorithms into an algorithm framework and employ template based code generation in Section II-B. Finally, we discuss the multi-threaded implementation in Section II-C.

A. Algorithms for Computing Multiple Predicates

We assume a column oriented main memory database environment, similar to MonetDB [2] and SAP HANA [7]. A numerical attribute is stored as a column in memory, i.e. an array of numerical values. All the attributes of a table are stored in the same order. That is, the array index corresponds to the record ID (a.k.a. RID) of the record, and the attributes of the same record can be retrieved from the associated columns at the same index. The result of the evaluation of a clause is a list of RIDs of the records that satisfy the conjunctive predicates in the clause.

We implement and evaluate our solution on the mainstream Intel Haswell architecture supporting 256-bit SIMD instructions. In the following, we present our algorithms based on the Haswell AVX2 instruction set.

Supporting a Single Predicate. We show the example SIMD code for the three types of predicates in Figure 1. Since their code structures are similar, we show the full code in Figure 1(a) for `col > val`, and show only the comparison portion of the codes for `col > col2` and `col IN [valuelist]` in Figure 1(b) and (c), respectively. `__m256i` is the C data type for the 256-bit SIMD register. We use a special prefix, `s_`, to denote variables that are 256-bit SIMD values. All the examples in Figure 1 use 32-bit integer columns and values. A 256-bit AVX2 instruction can process eight 32-bit integer values at a time.

In Figure 1(a), the constant `val` is loaded into all the eight 32-bit integer locations in a SIMD register (`s_val`). Then, the main loop loads and compares eight records from the input `col[]` in every loop iteration. The comparison result is in a 256-bit bit mask, where every 32 bits are either 0xFFFF if the comparison outcome is true, or 0x0000 if false. The next portion of the code converts the SIMD bit mask into a normal bit mask, uses bit manipulation instructions to find the position

```
int nresult= 0;
__m256i s_val= _mm256_set1_epi32(val);
for (int i=0; i+8<=nrecords; i+=8) {
 // compare 256 bit at a time and set the mask
 __m256i s_col=_mm256_load_si256(&col[i]);
 __m256i s_msk=_mm256_cmpgt_epi32(s_col,s_val);
 // convert mask and put rids into result[]
 unsigned int mask=_mm256_movemask_ps(s_msk);
 for (int m= _mm_popcnt_u32(mask); m>0; m--) {
  int which= _bit_scan_forward(mask);
  result[nresult++]= i + which;
  mask= _blsr_u32(mask);
 }
}
// process the rest of the input
```

(a) `col > val`

```
__m256i s_col=_mm256_load_si256(&col[i]);
__m256i s_col2=_mm256_load_si256(&col2[i]);
__m256i s_msk=_mm256_cmpgt_epi32(s_col,s_col2);
```

(b) SIMD comparison code for `col > col2`

```
__m256i s_col=_mm256_load_si256(&col[i]);
__m256i s_val= _mm256_set1_epi32(inlist[0]);
__m256i s_msk=_mm256_cmpeq_epi32(s_col,s_val);
__m256i s_msk2;
for (k=1; k<in_num; k++) {
  s_val= _mm256_set1_epi32(inlist[k]);
  s_msk2=_mm256_cmpeq_epi32(s_col,s_val);
  s_msk = _mm256_or_si256(s_msk, s_msk2);
}
```

(c) SIMD comparison code for `col IN [value list]`

Fig. 1. Supporting a single predicate with SIMD. (The inputs are 32bit integer columns and values. There are `nrecords` records. The result RIDs are in `result[0..nresult]`.)

of every 1 in the mask, and compute the result RID by adding the 1's position and the starting array index of the iteration.

Note that we have chosen the above implementation of the mask conversion after testing a number of alternative implementations. For example, one alternative implementation is to test every bit in the mask with an if branch. However, this alternative implementation incurs significant testing cost when the selectivity is low and many branch mispredictions when selectivity is medium.

Figure 1(b) and (c) omit the main loop and the mask conversion code, which are the same as in Figure 1(a). To compute `col > col2`, the code in Figure 1(b) loads eight values from both columns and compares them using a SIMD instruction. The code in Figure 1(c) can handle any number of values in the value list. It achieves this by using a loop[2], where `s_col` is compared to every value in the list. The generated bit masks are OR-ed together, since a record satisfies the condition if it matches any value in the value list.

To support a different data type and/or a different comparison type, we modify the above example codes. The code structure remains the same. The main change is to use the SIMD instructions of the specific data type and/or the specific comparison type. The number of records to process per

[2]Note that our code generation process unrolls the loop into sequential code as `in_num` is known given the value list, as explained in Section II-B.

```
for (int i=0; i+num<=nrecords; i+=num) {
 unsigned int mask= 0xFFFF;
 // process predicate 1
 unsigned int msk1= 0;
 for (int j=0; j<num; j+=n1) {
  load data and compute s_msk;
  msk1=(msk1<<n1)|_mm256_movemask_ps(s_msk);
 }
 mask &= msk1;
 // process more predicates
 convert mask and put rids into result[];
}
```

(a) Algorithm 1: evaluating all predicates together

```
1. Compute Predicate 1 on all records,
   put RIDs into result1[];
2. Compute Predicate 2 on records given by
   result1[], put RIDs into result2[];
......
k. Compute Predicate k on records given by
   resultk_1[], put RIDs into resultk[];
```

(b) Algorithm 2: evaluating predicates one by one

```
1. Compute Predicate 1 on all records,
   store the results in a bitmap;
2. Compute Predicate 2 on all records,
   store the results in a tempbmp;
   bitmap = bitmap bit-wise-AND tempbmp;
......
k. Compute Predicate k on all records,
   store the results in a bitmapk;
   bitmap = bitmap bit-wise-AND tempbmp;
k+1. convert bitmap to RIDs;
```

(c) Algorithm 3: evaluating predicates then combining bitmaps

Fig. 2. Algorithms for multiple conjunctive predicates.

iteration is adjusted as:

$$nrecPerIter = \frac{256}{8 \cdot sizeof(type)} \quad (1)$$

Next, we describe three baseline algorithms to process multiple conjunctive predicates. Algorithm 1 scans through all the relevant columns at the same time, evaluating all the predicates. In comparison, Algorithm 2 and Algorithm 3 scan columns one by one. They differ in two aspects: (i) a predicate is evaluated either on all the records or on only the records that satisfy previous predicates, (ii) intermediate results are stored either as RID lists or as bitmaps. The following provides more details of the three algorithms.

Algorithm 1: Evaluating All Predicates Together. Our first algorithm to evaluate multiple conjunctive predicates is to extend the comparison code in Figure 1 to compute all the predicates. In the loop body, the predicates are evaluated one by one to generate a bit mask per predicate. Then all the bit masks are AND-ed together to obtain the conjunctive result. Finally, the resulting bit mask is converted to RIDs in a similar fashion as in Figure 1(a).

However, this simple solution does not work if columns with different data sizes are present in the conjunctive predicates. Suppose $n1$, $n2$, ... are the number of records per iteration (or loop stride) as computed by Equation 1 for every predicate, respectively. The problem arises if $n1$, $n2$, ... are not the same. Our solution is illustrated in Figure 2(a). We set $num = LCM(n1, n2, ...)$, which is the least common multiple of the individual loop strides. Then, for a predicate (e.g., predicate 1), an inner loop is introduced to compute and concatenate $\frac{num}{n1}$ bit masks before AND-ing the concatenated mask to the global mask. Note that the smallest data size is one byte, and thus num is at most $\frac{256}{8} = 32$. Therefore, 32-bit `unsigned int` is large enough for the concatenated mask.

Algorithm 2: Evaluating Predicates One by One. The second algorithm evaluates a predicate entirely before evaluating the next predicate, as shown in Figure 2(b). Step 1 in the algorithm is exactly the same as in Figure 1. The rest of the steps are also very similar. The main difference is that they do not directly load contiguous data from the underlying columns because the records that satisfy all previous predicates usually scatter across the column. Therefore, we employ a SIMD gather operation to retrieve the data values with the RIDs (i.e. the array indices) given by the previous `result[]`.

Algorithm 3: Evaluating Predicates then Combining Bitmaps. The third algorithm evaluates every predicate separately. As shown in Figure 2(c), every predicate is evaluated on all the records. The evaluation is similar to Figure 1 except that the mask conversion is omitted. Instead, the bit mask is copied to an intermediate buffer that will hold the entire bitmap of the comparison result. All the bitmaps are bit-wise AND-ed together. In the end, the result bitmap is converted into RIDs.

Both Algorithm 1 and Algorithm 3 compute the predicates on all the records regardless of selectivity, while Algorithm 2 avoids predicate computation for records that do not satisfy previous predicates. Therefore, Algorithm 2 may perform better when selectivity is low. On the other hand, when selectivity is high, Algorithm 2 pays higher overhead for the saving and reading of the intermediate results and the SIMD gather operations. Comparing Algorithm 1 and 3, we see that Algorithm 1 will typically be faster because it consumes the bit masks on the fly without storing the intermediate bitmaps. However, Algorithm 1 requires generating and compiling code for a set of predicates. In contrast, the high-level logic of Algorithm 3 in Figure 2(c) is generic. It is also feasible to pre-generate the code for each predicate type, comparison type, and data type combination, which is 3*6*6= 108 combinations here. Therefore, this solution is interesting for ad-hoc queries where code generation cost may be significant.

B. Template Based of Code Generation

Code Framework. We propose a code framework to combine the three algorithms into an execution plan for evaluating multiple conjunctive predicates as follows:

- *plan*: an execution plan consists of one or multiple steps, using the high-level structure of Algorithm 2 in Figure 2(b). Every step returns an RID list containing records that satisfy the predicates up to this step. If there are multiple steps, then Step k evaluates only the records in the RID list generated by Step k-1, (k$\geq$2);
- *step*: a step consists of one or multiple functions. If it has only one function, then the function computes

an RID list as in Algorithm 1. If it has multiple functions, then every function computes a bitmap, and the step follows the high-level structure of Algorithm 3 to merge all the bitmaps and generate an RID list.

- *function*: a function evaluates one or multiple predicates. If it is the only function in a step, then the function performs Algorithm 1. Otherwise, the function omits the mask conversion portion of the code, and computes a bitmap as the result.

We write a plan as a sequence of predicates. We use $\rightarrow$ to separate steps, parentheses to separate functions, and commas to separate predicates in a function.

For example, given a clause with four predicates

$$p1\ AND\ p2\ AND\ p3\ AND\ p4$$

the plan for Algorithm 1 is $(p1, p2, p3, p4)$. It has a single step with a single function. The function evaluates all four predicates. A plan for Algorithm 2 is $(p1) \rightarrow (p2) \rightarrow (p3) \rightarrow (p4)$. It has four steps, each having a single function. And the plan for Algorithm 3 is $(p1)(p2)(p3)(p4)$. It contains a single step. The step combines the bitmaps from four functions.

In addition to the above three algorithms, the code framework allows execution plans that combine features from the three algorithms. For example, $(p1, p2) \rightarrow (p3, p4)$ is a plan that performs two subsequent steps. The first step computes both $p1$ and $p2$ together in a function. The second step examines only the records satisfying the first step and computes $p3$ and $p4$ in a function.

Code Generation. Given a plan, we take a template based approach to generate code in C/C++. Then, we compile the code into a dynamic library with g++ with `-mavx2`, load the dynamic library with `dlopen`, and run the code.

We take advantage of C preprocessing macros in composing the code templates. In addition, we design a pattern parser that supports the patterns defined in Table I to facilitate code generation. For example, the SIMD comparison can be implemented in the template as follows:

```
__mm256i s_mask= ${simd_cmp}(s_col, s_val);
```

Our code generation module specifies `simd_cmp` based on the column data type and the comparison operation in a specific predicate. In this way, the same code template can be effectively reused.

The 'if' pattern is used to select a snippet of code from multiple alternative code snippets. For example, we put the code snippets of the three predicate types into the same template, and use a key to control which code snippet should be compiled.

The 'for' pattern repeats a portion of the code template. This is useful for supporting multiple values in an IN value list, multiple predicates in a function, multiple functions in a step, and multiple steps in a plan. The code is unrolled into sequential code to enable better optimizations by C/C++ compilers.

TABLE I. CODE TEMPLATE PATTERNS.

`${key}` Substitute the occurrence with `symbol[key]`
`${key[id]}` Substitute the occurrence with `symbol[key[id]]`. If `symbol[id]` exists, then first substitute `id` with `symbol[id]`.

`#if ${key} == value`
`  STATEMENTS1`
`#else`
`  STATEMENTS2`
`#endif`
If `symbol[key]` is value, then produce STATEMENTS1, otherwise produce STATEMENTS2

`${for (key=start; key < end; key+=step)}`
`  STATEMENTS`
`${endfor}`
`start`, `end`, `step` must be integers or they are defined in the symbol table as integers. The for loop has `(end - start)/step` iterations. In each iteration, `key` takes a different integer value, and STATEMENTS are produced with `key` substituted by the integer value.

C. Multi-threaded Implementation

We support multi-threaded evaluation of conjunctive predicates. There are two concerns. First, we would like to balance the loads across the threads. Second, we would like the data to be evaluated in every thread to start at 32-byte address boundaries so that efficient aligned SIMD loads can be used.

To address the concerns, we allocate all the columns starting at 32-byte boundaries. Given m threads, we divide the total number (N) of records into m conceptual chunks so that Chunk 0 to Chunk $m-2$ contain the same number of records, while the last Chunk $m-1$ may contain slightly fewer records. Chunk k is assigned to thread k to process. Therefore, the threads have similar amount of loads.

Moreover, we make sure that the number of records in Chunk 0 to Chunk $m-2$ is a multiple of 32. That is, the number of records in Chunk 0 is $32 \cdot \lceil \frac{N}{32m} \rceil$. In this way, it is easy to see that all the columns in all the chunks start at 32-byte boundaries regardless of column data types.

III. MODELING AND OPTIMIZING PLANS

We propose a linear cost model for a single-threaded plan in Section III-A. Then, based on this model, we consider the memory bandwidth saturation problem and propose a cost model for multi-threaded evaluation in Section III-B. Finally, we describe how to perform cost-based optimization in Section III-C.

A. Modeling Single-threaded Plans

The cost of a plan is a linear combination of all the steps:

$$C_{plan} = \sum_{i=1}^{n_{step}} C_{stepi}$$

The cost of a step depends on whether or not the step is the first step and whether the step consists of one or multiple functions:

$$C_{stepi} = \begin{cases} Ca_{1,1} & \text{if } i = 1 \text{ and } n_{fun1} = 1 \\ \sum_{j=1}^{n_{fun1}} Cb_{1,j} + C_{merge} & \text{if } i = 1 \text{ and } n_{fun1} > 1 \\ Sel_{i-1} Cc_{i,1} & \text{if } i > 1 \text{ and } n_{funi} = 1 \\ Sel_{i-1}(\sum_{j=1}^{n_{funi}} Cd_{i,j} + C_{merge}) & \text{if } i > 1 \text{ and } n_{funi} > 1 \end{cases}$$

The first step reads the original records, while the subsequent steps all read a subset of records. Sel_{i-1} is the combined selectivity of the first $i-1$ steps. If there is a single function, the step simply calls the function. In contrast, If there are multiple functions, the cost of the step consists of the cost of every function and the cost to merge the bitmaps.

The cost of a function is a linear combination of the cost of evaluating every predicate, the cost of converting masks into RIDs (in Ca and Cc) or copying bit masks into intermediate bitmaps (in Cb and Cd), and certain fixed cost to set up and run the function.

$$Ca_{1,1} = C_{fixed} + N_{iter}(\sum_{k=1}^{n_{pred1,1}} C_{pred1,1,k} + Sel_1 C_{maskconv})$$
$$Cb_{1,j} = C_{fixed} + N_{iter}(C_{pred1,j,1} + Sel_1 C_{maskcopy})$$
$$Cc_{i,1} = C_{fixed} + N_{iter}(\sum_{k=1}^{n_{predi,1}} C'_{predi,1,k} + Sel_i C_{maskconv})$$
$$Cd_{i,j} = C_{fixed} + N_{iter}(C'_{predi,j,1} + Sel_i C_{maskcopy})$$

The subscript x, y, z means Step x Function y Predicate z. Ca and Cc evaluates multiple predicates in a function and generates an RID list, while Cb and Cd evaluates a single predicate and computes an entire bitmap. Note that $Cpredi, j, k$ and $C'predi, j, k$ are the same except that the latter retrieves column values using the SIMD gather operation.

The cost of a predicate consists of the SIMD memory read cost and the SIMD comparison cost. The read cost and the comparison cost are dependent on the data type. We find that for a given numerical type, the SIMD comparison instruction of different comparison operation usually has the same latency and throughput. Therefore, the predicate cost can be computed as follows:

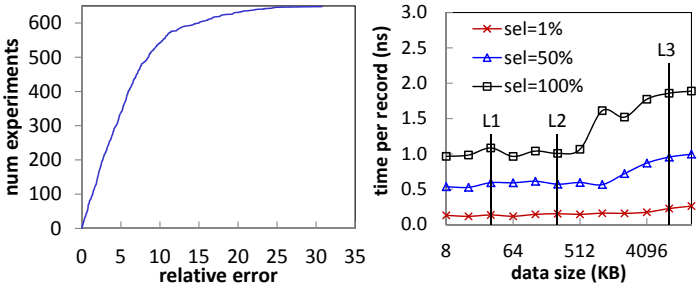
$$C_{pred} = \begin{cases} C_{read(type)} + C_{cmp(type)} & \text{if } col\ op\ val \\ 2C_{read(type)} + C_{cmp(type)} & \text{if } col\ op\ col2 \\ C_{read(type)} + C_{cmp(type)} & \text{if } col\ IN\ [valuelist] \\ \quad +(n_{in} - 1)C_{cmp2(type)} & \end{cases}$$

For $Cpredi, j, k$ and $C'predi, j, k$, we use two sets of read parameters

Parameter Estimation and Model Precision. We generate synthetic columns of data, synthetic predicates, and various plans. We measure the execution time of evaluating the plans. Using the linear model, we obtain a set of linear equations as follows:

$$\begin{bmatrix} c_1 \\ c_2 \\ \dots \\ c_u \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1v} \\ w_{21} & w_{22} & \dots & w_{2v} \\ \dots & & & \\ w_{u1} & w_{u2} & \dots & w_{uv} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \dots \\ p_v \end{bmatrix}$$

There are u measurements and v unknown parameters. ws are the parameters that can be computed from either the plan or the data, such as the numbers of iterations, the numbers of records and the selectivities. In this case, there are $v = 57$ unknown parameters. We compute the unknown parameters by using non-negative least squares since the parameters should be non-negative.



(a) Error distribution (b) Impact of memory hierarchy

Fig. 3. Modeling single-threaded plans.

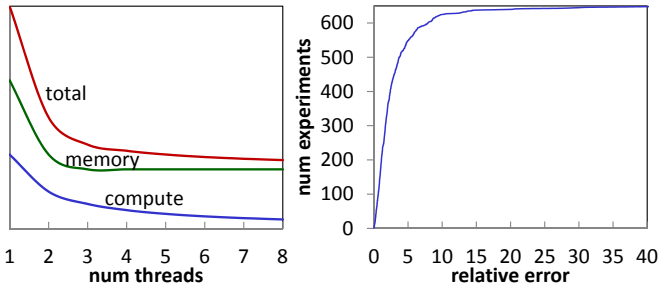
All the experiments are run on a ThinkCentre M8500t workstation equipped with an Intel Haswell I7-4770 CPU (4 cores, 8 threads, 32KB L1D, 256KB L2, and 8MB L3 cache) and 16GB DRAM, running Ubuntu 14.04 LTS. The Linux kernel version is 3.13.0-24-generic and the GCC version is 4.8.2. We generate synthetic data that are randomly uniformly distributed. We generate the six types of data and the three types of predicates. We vary the number of columns and the constructed plans. Altogether, we generate 648 different configurations to run experiments.

Figure 3(a) shows the precision of our single threaded cost model. We see that 52% of the experiments have less than 5% of errors, 84% of the experiments see less than 10% of errors, and the average error is 5.9%.

An interesting question to ask is whether the CPU cache sizes play a significant role in determining the performance. Figure 3(b) shows the micro-benchmark result of evaluating a single predicate on a 32-bit integer column while varying the selectivity and the underlying data size. When selectivity is low, the average run time of a record has only minor changes when the data size changes. On the other hand, there is a clear upward trend when selectivity is 50% and 100%. This means that the SIMD loading performance is not very sensitive when the data does not fit into the cache, while the writing of the RIDs is quite sensitive. Our cost model does not yet consider the cache effect, which will be interesting future work.

B. Modeling Multi-threaded Plans

We consider cost models for multi-threaded evaluation of plans. Figure 4 shows the idea of our solution. We divide the linear components of the single-threaded model into two categories: compute related components and memory related components. If a component is compute related, then its contribution to the overall run time is considered fully parallelizable because the computation can be performed on different CPU cores without significantly interfering with each other. On the other hand, if a component is memory related, we observe that the memory bandwidth is often saturated with only a small number of threads (e.g., 3). Therefore, the contribution of the component must reflect this saturation effect as shown in Figure 4. The total cost is the sum of the two parts.



(a) Illustration of model (b) Error distribution

Fig. 4. Modeling multi-threaded plans.

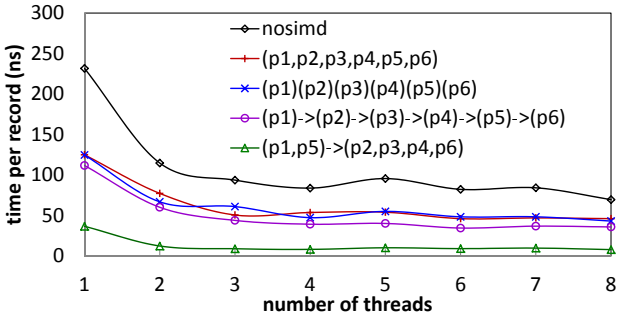


Fig. 5. Experimental result for a 6-predicate clause on synthetic data.

We rewrite the single-threaded cost model:

$$C\_plan = C\_compute + C\_memory$$

Then, the cost model for m threads is as follows:

$$C\_plan(m) = \frac{C\_compute}{m} + max(\frac{C\_memory}{m}, C\_saturate)$$

Here, $C\_saturate$ has the same form as C_memory except that all the parameters are replaced with new (unknown) parameters. We call these new parameters in the saturated case, saturated parameters.

We can estimate the model parameters in a similar fashion as the single-threaded case. Here, given a configuration with a plan and a set of columns, we can compute the single-threaded C_plan, $C\_compute$ and C_memory using the single-threaded model. Then we can measure the multi-threaded execution time, $C\_plan(m)$. In this way, we can find the saturated cases and use them to formulate a linear squares problem for the saturated parameters.

Figure 4 shows the error distribution for the multi-threaded cost model. We see that 85% of the experiments see less than 5% errors, 96% of the experiments see less than 10% errors, and the average errors are 3.0%. This means that the saturation effect helps improve the model precision compared to the single-threaded case.

C. Plan Optimization

We implemented two algorithms for selecting the optimal plans. The first algorithm enumerates all the possible plans and finds the best plan with the lowest cost. Ross [9] pointed out that the number of plans grows factorially for evaluating conjunctive predicates without SIMD. The code structure involves more flexibility than the no SIMD cases. Therefore, the number of SIMD plans grows at least factorially. Consequently, the enumeration algorithm will soon become intractable when the number of predicates becomes large.

We also implemented a greedy approximation algorithm to find a good plan. The algorithm works as follows. It first puts all the predicates into a single function. Then, it tries to move every predicate to the second step and computes the benefit. From all the predicates, it greedily moves the best predicate with a positive benefit (if any) to the second step. After that, it repeats to see if any more predicates should be moved to the second step. When there are no more predicates to be moved to the second step, it goes on to examine if any predicates should be moved from the second to the third step, and so on. In most of the cases that we see, two steps are good enough.

Figure 5 is the experimental result for evaluating a clause with 6 conjunctive predicates:

$$Col_{int8} < 30 \ AND \ Col_{int16} < 80$$
$$AND \ Col_{int32} < 100 \ AND \ Col_{int64} < 50$$
$$AND \ Col_{float} < 10.0 \ AND \ Col_{double} < 90.0$$

The data type is marked as the subscript of the Col. We randomly generate 1024000 records. Each data value in every column is uniformly distributed in $[0, 99]$. Therefore, the selectivity of a predicate (e.g., $Col_{int8} < 30$) can be easily seen (e.g., 30%).

In Figure 5, we vary the number of threads on the X-axis. The Y-axis reports the average time for processing every record. The lower the better. We have evaluated five plans. We implement and optimize no SIMD plans by following the description in the previous work [9]. The optimal no SIMD plan performs the worst among all the plans. This clearly shows the benefit of exploiting SIMD for predicate evaluation. The three middle plans correspond to the three Algorithms in Section II-A. The optimal plan consists of two steps the first predicate and the fifth predicate are evaluated in Step 1. The combined selectivity is 3%. Then the rest of the predicates are evaluated in Step 2. Our approximation algorithm achieves the same optimal plan as the enumeration algorithm in this case.

Overall, the optimal plan achieves 6.3–10.4x speedups over the best no SIMD plan, and 3.0–6.8x speedups over the other SIMD plans. Interestingly, the best performance is reached with three threads. This means that there is plenty of CPU power left unused because of the memory bandwidth bottleneck.

IV. Conclusion and Future Work

In conclusion, SIMD can be effectively exploited to improve the performance of conjunctive numerical predicates. However, the actual algorithm employed may have a very significant impact on the achieved performance. We have proposed a SIMD code framework, and cost models for modeling both the single-threaded and multi-threaded execution of the generated code. Our experiments show that the errors of the cost models are lower than 10% in most cases. The optimal plans computed based on the cost models achieve significantly better performance than both the no SIMD plan and the other SIMD plans.

As the next step, we would like to perform more extensive evaluation of our solution. We would like to use TPCH data and real-world data in the evaluation. Moreover, we plan to integrate our implementation with an open-source main

memory database system, and measure the performance of using our optimizations in the real system. Furthermore, there are several use cases that are worth considering if time allows. The comparison operation may link two expressions instead of columns or values as considered in this paper. Numerical type conversion may be necessary if multiple columns in the expressions have different data types. Note that value types can be easily converted during query parsing if needed.

ACKNOWLEDGMENT

The second author is partially supported by the CAS Hundred Talents program, by NSFC Project No. 61572468, and by NSFC Innovation Research Group No. 61221062.

REFERENCES

[1] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-memory BLU acceleration in ibm's DB2 and dashdb: Optimized for modern workloads and hardware architectures. In *ICDE*, pages 1246–1252, 2015.

[2] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[3] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, pages 31–46, 2015.

[4] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, and el al. Oracle database in-memory: A dual format in-memory database. In *ICDE*, pages 1253–1258, 2015.

[5] P. Larson, E. N. Hanson, and M. Zwilling. Evolving the architecture of SQL server for modern hardware trends. In *ICDE*, pages 1239–1245, 2015.

[6] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[7] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, 7(13):1722–1729, 2014.

[8] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[9] K. A. Ross. Conjunctive selection conditions in main memory. In *PODS*, pages 109–120, 2002.

[10] E. A. Sitaridi and K. A. Ross. Ameliorating memory contention of OLAP operators on GPU processors. In *DaMoN*, pages 39–47, 2012.

[11] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.