DEIM2020 H1-4(day1 p50)

# ステージング型クエリコンパイラにおけるSIMD命令活用方式の提案

根本　潤†　川島　英之††　遠山　元道†††

† 慶應義塾大学大学院理工学研究科　〒223–8522 神奈川県横浜市港北区日吉 3-14-1
†† 慶應義塾大学環境情報学部　〒252–0882 神奈川県藤沢市遠藤 5322
††† 慶應義塾大学理工学部　〒223–8522 神奈川県横浜市港北区日吉 3-14-1
E-mail: †nemoto@keio.jp, ††river@sfc.keio.ac.jp, †††toyama@ics.keio.ac.jp

**あらまし**　RDBMS における先進的なクエリエンジンの多くは，クエリ実行プランを逐次解釈するインタープリタではなく，クエリ実行プランから効率的な機械語を Just-In-Time（JIT）に生成・実行するクエリコンパイラとして実装されている．このようなクエリコンパイラの最新研究として，ステージング型のクエリコンパイラがある．ステージング型のクエリコンパイラでは，部分評価を用いた最適化により，高級言語でインタープリタを実装するのと同等の開発効率で，高速なクエリ処理を実現する．本論文では，第一に，インタープリタに対するステージング型クエリコンパイラの性能優位性について定量的な評価を行った．評価の結果，ステージング型クエリコンパイラはインタープリタに比べて 8〜20 倍程度高速であることが観察され，その性能差の理由は (1) タプルの NULL チェック，(2) 仮想関数呼出し，(3) 述語評価時の構造体メンバ参照であることが判明した．本論文では，第二に，クエリコンパイラにおける SIMD 命令活用法を明らかにするため，SIMD 命令を生成するクエリコンパイラを設計，実装し，評価を行った．評価の結果，選択オペレータにおける SIMD 命令活用による性能向上は限定的であり，最大でも 3%に留まることが観察された．

**キーワード**　JIT クエリコンパイラ，SIMD，部分評価

## 1 はじめに

従来，RDMBS のクエリエンジンの多くは，クエリ実行プランを逐次解釈し，反復的に関係オペレータを評価するインタープリタとして実装されてきた [1]．ディスク I/O が主たる性能のボトルネックだった頃において，直感的で理解しやすいインタープリタの処理モデルは，クエリエンジンの標準となっていたが，近年，メインメモリ大容量化や NVM のようなストレージの普及，計算インテンシブな分析処理ニーズの増大にともなって，クエリ処理が CPU バウンドになってきたため，クエリ実行プランから効率的な機械語を Just-In-Time（JIT）に生成・実行するクエリコンパイラが主流になりつつある [2] [3] [4]．

クエリコンパイラの最新研究として，ステージング型のクエリコンパイラがある [5] [6] [7]．ステージング型のクエリコンパイラでは，部分評価を用いた最適化により，高級言語でインタープリタを実装するのと同等の開発効率で，高速なクエリ処理を実現する．例えば N 段の関係結合演算を行う場合を例に取り，その効果を述べる．広く使われている Volcano 方式のインタープリタ [1] [8] では，多数回の関数呼出し，ならびにそれにともなう中間タプルの生成，受渡し処理が必要となる．一方，本研究で扱うステージング型クエリコンパイラ LB2 [5] を用いる場合，N 段の for ループが形成されるために関数呼出し回数は高々1 回となり，中間タプルの生成も不要となる．さらに，近年の研究においては，CPU の命令レベルでのデータ並列化のため，クエリコンパイラにおいて SIMD 命令を活用する研究も行われている [9]．

このようなクエリコンパイラ研究において，本論文では 2 つの課題に取り組んだ．第一の課題は定量的評価である．クエリコンパイラ研究において，その性能改善の理由は巨視的・定性的には知られているが，微視的・定量的には筆者が知る限り調査が行われていない．例えば，CPU 命令レベルで挙動を分析した場合，ステージング型クエリコンパイラが Volcano 方式のインタープリタに対して如何なる理由で優位であるかは研究的な疑問である．第二の課題は SIMD 命令の活用法である．SIMD 命令を用いる場合にはハードウェアに特化した複雑なコード生成が求められるため，一般的に効果的な生成手段は，現状では未知である．文献 [9] [10] では SIMD による問合せ処理の高性能化が報告されているが，それが発現する条件については整理がされていない．

本論文では上記 2 つの課題に取り組んだ結果を報告する．第一の課題を解決すべく，多段の結合に関して Volcano 方式インタープリタとステージング型クエリコンパイラ LB2 をスクラッチ実装して比較評価した．具体的には，処理時間，CPU 統計情報，命令数を計測し，両者を比較した．その結果，LB2 は Volcano 方式インタープリタに比べて 8〜20 倍程度の性能向上が観察された．その性能差の理由は (1) タプルの NULL チェック，(2) 仮想関数呼出し，(3) 述語評価時の構造体メンバ参照であることが判明した．第二の課題を解決すべく，SIMD 命令を生成するクエリコンパイラを設計・実装し，TPC-H ベンチマーク Q1, Q14 の処理時間を測定した．その結果，選択オペレータにおける SIMD 命令活用の効果は限定的であり，最大でも 3%

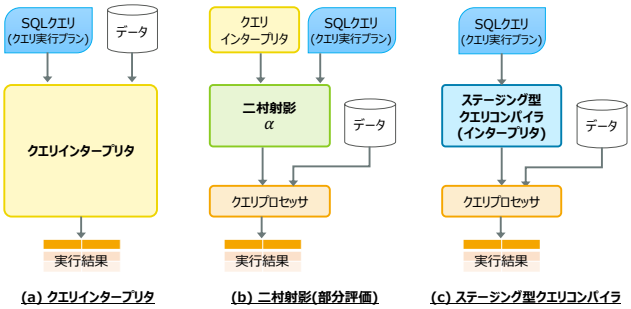


図 1　ステージング型クエリコンパイラ

に留まることが観察された．このような限定的な性能向上は従来研究 [9] と非一貫的なものであり，SIMD により性能向上発現条件の明確化には，さらなる研究が必要なことが判明した．

本論文の構成は以下の通りである．まず，2 章で関連研究について述べる．次に，3 章で提案するクエリコンパイラのシステムアーキテクチャと SIMD 命令を活用した選択オペレータの処理方式について述べる．4 章で実装の詳細を述べた後，5 章で評価を行い，最後に 6 章でまとめを述べる．

## 2　関 連 研 究

### 2.1　ステージング型クエリコンパイラ

文献 [5] は，二村射影の考え方を参考に，ステージング型のクエリコンパイラ LB2 を提案している．ステージング型クエリコンパイラの構成を図 1 に示す．

二村射影は，インタープリタとソースプログラムを入力として，同等の結果を出力するより効率的なコンパイル済みプログラムに変換する（特化する）ことを言う．図 1(b) のように，クエリインタープリタとクエリ実行プランに対して同様のことを行えば，特化の過程でインタープリタのオーバヘッドが除去され，効率化が期待できる．ただ，クエリインタープリタとクエリ実行プランを入力に，完全自動で変換を行うプログラムを得ることは容易ではない．

そのため，LB2 では，図 1(c) のように，クエリ実行プランをインタープリタ的に解釈する過程 (第 1 ステージ) で部分評価を行い，クエリ処理 (第 2 ステージ) を行う C コードを生成することで，二村射影に相当する変換を実現している．部分評価とコード生成に際しては，内部的に，ライブラリベースの生成的プログラミングフレームワークである Lightweight Modular Staging (LMS) [11] を利用している．第 2 ステージで処理する式を LMS が提供する型コンストラクタ Rep[T] を用いて明示することでコード生成を行う．

LMS で生成された C コードは，特化により効率化されており，高速なクエリ処理 (第 2 ステージ) が可能であるものの，SIMD 命令のような最新ハードウェアを積極的に活用するような高速化手法については議論されていない．

### 2.2　SIMD 命令活用

文献 [9] では，Relax Operator Fusion(ROF) と呼ばれるクエリ処理モデルを導入することで，クエリコンパイラにおいて SIMD 命令を活用する方法が提案されている．昨今のクエリコンパイラは，通常，Data-Centric(もしくはプッシュ型) と呼ばれるモデルを採用しており，効率化のため，クエリの各オペレータ間では極力タプルのマテリアライズ (バッファへの書き出し) を行わないようにするのが一般的だが，ROF では，クエリ実行プランの中にあえてマテリアライズを行うポイントを設けることで，SIMD 命令を活用可能にする．ただ，文献 [10] において言及されているように，Data-Centric なモデルにおける SIMD 活用コードの生成は複雑であり，その容易化は引き続き重要な研究課題である．また，文献 [10] では，SIMD 命令による性能向上が報告されているが，VectorWise [12] のような Block-at-a-time なベクトル指向処理モデルにおける評価に留まっており，Hyper [2] のような Tuple-at-a-time な処理モデルにおける SIMD 命令活用については議論されていない．

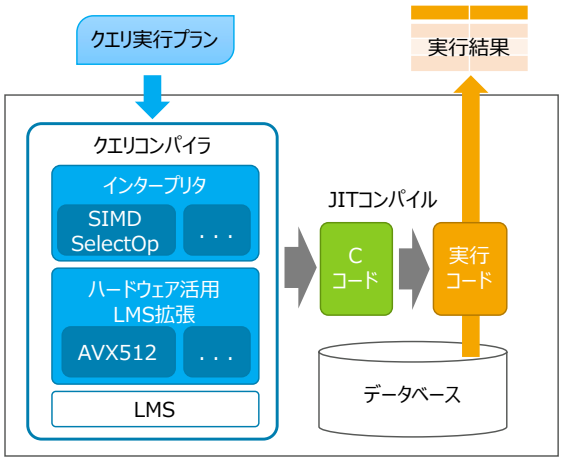


図 2　システムアーキテクチャ

## 3　提 案 方 式

### 3.1　システムアーキテクチャ

提案する SIMD 命令を活用したステージング型クエリコンパイラのシステムアーキテクチャを図 2 に示す．提案システムでは，クエリ実行プランを入力として，それを逐次的に解釈し，C コードを生成する．そして，生成した C コードを JIT コンパイルし，実行することで問合せ結果を応答する．

クエリコンパイラを構成する主要なコンポーネントは，インタープリタ，LMS，ハードウェア活用 LMS 拡張の 3 つである．クエリ実行プランを解釈しながらコードを生成するインタープリタ部分では，選択，集約，結合といった一般的な関係オペレータに加え，SIMD 対応の選択オペレータなど，最新ハードウェアを活用するオペレータを導入する．コード生成には先行研究である LB2 と同様に LMS を使用するが，ハードウェアに特化した機能はないため，ハードウェア活用 LMS 拡張部分で，SIMD 拡張命令セットである AVX-512 に対応したコード生成を可能にする．LMS は，拡張可能なライブラリとなっており，AVX-512 における _m512i や __mmask16 のようなデータ型の

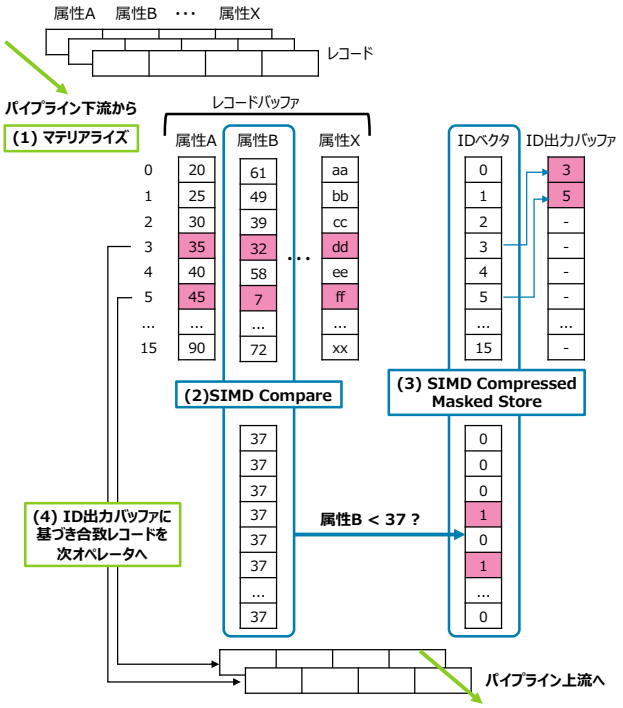


図 3　SIMD 対応選択オペレータの処理フロー

追加や，_mm512_add_epi32 といった組込み関数を利用した処理を独自に定義することができる．

### 3.2　SIMD 対応選択オペレータ

提案システムにおいて追加する SIMD 対応選択オペレータの処理フローを図 3 に示す．所謂 Data-Centric な処理モデルにおける通常の選択オペレータは，クエリ処理パイプラインの下流から渡されてくるタプルを逐次（tuple-at-a-time で）評価するが，SIMD 命令を活用して複数のタプルを一括評価するため，所定の数のタプルを一旦バッファに格納する（図 3(1)）．このとき，バッファは述語評価に必要なカラム単位で管理しておき，バッファが一杯になったところで SIMD レジスタにロードし，比較演算を行う（図 3(2)）．例えば，図 3 のように，「属性 B < 37」という条件で選択処理を行う場合，属性 B のベクタと 37 のみで構成されるベクタで比較演算を行い，条件に合致する 3 番目と 5 番目のビットだけが 1 であるビットマスクが得られる．そして，得られたビットマスクと，バッファのオフセットが格納されたインデックスベクタ（ID ベクタ）を用いて，合致したタプルのオフセットの配列を得る（図 3(3)）．最後に，当該配列の内容に基づいてバッファから合致したタプルを取り出して，パイプラインの上流へ渡していく（図 3(4)）．

## 4　実　　装

本章では，試作した提案システムの実装について述べる．提案システムにおけるクエリコンパイラは，LMS を利用するため，Scala を用いて実装した．インタープリタ部分に関しては，通常のスカラーデータのまま処理する以下のオペレータを実装した．これらのオペレータの評価モデルとしては，LB2 と同様

```
1  class SelectVectorOp(child: Op,
2                       predicates: Seq[Predicate]){
3    def exec = {
4      val buffer = new RecordBuffer(16)
5      val indexBuf = new Buffer[Int](16)
6      val indexVec = Vec16(15, 14, ..., 1, 0)
7      val foreachRecord = child.exec
8      (callback: RecordCallback) => {
9        foreachRecord { record =>
10         buffer.add(record)
11         if (buffer.len == 16) {
12           var mask: Mask16 = Mask16FromInt(0xffff)
13           predicates.foreach { pred =>
14             mask = mask
15               && evalVectorPredicate(pred, buffer)
16           }
17           indexVec.toBufferPacked(indexBuf, mask)
18           for (i <- 0 until mask.popcount) {
19             callback(Record(buffer(indexBuf(i))))
20           }
21           buffer.clean
22         }
23       }
24       // Scalar operations for the rest are omitted
25     } // Retruns (RecordCallback => Unit) function
26 } }
```

図 4　SIMD 対応選択オペレータの疑似コード

にコールバックを用いた Data Centric なモデルを用いた．

- SelectOp: 選択
- ProjectOp: 射影
- CalculateOp: 式評価
- NestedLoopJoinOp: ネストループ結合
- HashJoinOp: ハッシュ結合
- AggregateOp: 集約
- CaseWhenOp: 条件分岐
- SortOp: ソート

上記に加えて，SIMD 命令を用いてフィルタリングを行う選択オペレータ (以下，SelectVectorOp) を実装した．図 4 に示す疑似コードを用いて，SelectVectorOp について詳細に説明する．

SelectVectorOp は，子オペレータと，述語のシーケンスをパラメータに取り，exec メソッドを有するクラスとして実装する．exec メソッドは，タプルに対して選択操作などの各種処理を適用するコールバック関数 (type RecordCallback = Record => Unit) を入力に，当該オペレータの処理を行う関数を返すメソッドである (8〜25 行目)．SelectVectorOp では，子オペレータから受け取った反復処理 (7 行目) の中で，一旦マテリアライズを行ったのち (10 行目)，evalVectorPredicate() において各述語を SIMD 命令で一括評価していく (15 行目)．そして，得られたビットマスクとインデックスベクタを用いて，真となったタプルのインデックスをバッファに戻し (17 行目)，そのインデックスに該当するタプルを次のオペレータへと引き渡していく (19 行目)．

ここで，Vec16 や Mask16 は，拡張した LMS で定義した Rep[__m512i] 型や，Rep[__mmask16] 型の演算をカプセル化したクラスである．Vec16 の疑似コードを図 5 に示す．例えば，3 行目に定義するように，Vec16 同士の加算は，コード生成時に AVX-512 向けの組込み関数_mm512_add_epi32 による加算に置き換わる．同様に，7〜8 行目に定義するように，Vec16 同士の

```
abstract class Vec
case class Vec16(reg: Rep[__m512i]) extends Vec {
  def add(x: Vec16) = Vec16(_mm512_add_epi32(reg, x.reg))
  def sub(x: Vec16) = Vec16(_mm512_sub_epi32(reg, x.reg))
  def mul(x: Vec16) = Vec16(_mm512_mullo_epi32(reg, x.reg))
  def div(x: Vec16) = Vec16(_mm512_div_epi32(reg, x.reg))
  def isEq(x: Vec16)
    = Mask16(_mm512_cmpeq_epi32_mask(reg, x.reg))
  def isGe(x: Vec16)
    = Mask16(_mm512_cmpge_epi32_mask(reg, x.reg))
  def isGt(x: Vec16)
    = Mask16(_mm512_cmpgt_epi32_mask(reg, x.reg))
  def isLe(x: Vec16)
    = Mask16(_mm512_cmple_epi32_mask(reg, x.reg))
  def isLt(x: Vec16)
    = Mask16(_mm512_cmplt_epi32_mask(reg, x.reg))
  def toBufferPacked(buffer: Rep[Array[Int]], mask: Mask16)
    = _mm512_mask_compressstoreu_epi32(buffer, mask.reg, reg)
}
```

図 5　Vec16 クラスの疑似コード

等価比較演算は，_mm512_cmpeq_epi32_mask を用いた等価比較演算に置き換わる．また，17〜18 行目の Vec16 の toBuffer-Packed メソッドは，_mm512_mask_compressstoreu_epi32 を用いて指定したバッファにビットマスクに該当する要素のみを詰めて書き出す処理を行う．このような抽象化が可能なのがステージング型クエリコンパイラの特徴の 1 つである．

## 5　評　　価

### 5.1　評 価 環 境

試作したクエリコンパイラを用いて 2 つの評価を行った．1 つは，単純なネストループ結合実行時間に関するマイクロベンチマークである．クエリコンパイラの性能優位性を定量的に確認するため，本評価用に実装した Volcano 方式のインタープリタと比較評価を行った．もう 1 つは，TPC-H クエリのサブセットを用いた評価であり，SIMD 命令活用の効果を確認するために実施した．

評価環境は，次の通りである．

- CPU: Intel Xeon Platinum 8176 2.10GHz
- L3 キャッシュ: 38.5 MB
- メモリ: 1TB
- OS: Ubuntu 18.04 (Linux Kernel 4.15.0)
- コンパイラ: GCC 7.4.0, Clang 6.0.0 (最適化: -O3)

なお，スレッド数，TPC-H のスケールファクタはそれぞれ 1 とした．

### 5.2　インタープリタ型との比較評価

はじめに，単純なネストループ結合の実行時間に関する評価を行った．使用するデータベースは，それぞれ 10 万件のタプルが格納された 2〜10 個のテーブルで構成される．各テーブルは，整数型の Key カラムと Value カラムの 2 つのカラムを有する．Key カラムには，0〜99,999 までの整数が順に格納されており，Value カラムには，0〜99,999 の値のうちいずれかが一様ランダムに格納されている．

上記のテーブルを 2〜10 個と結合数を変えながらネストループ結合した際の実行時間を図 6 に示す．評価の結果，試作した

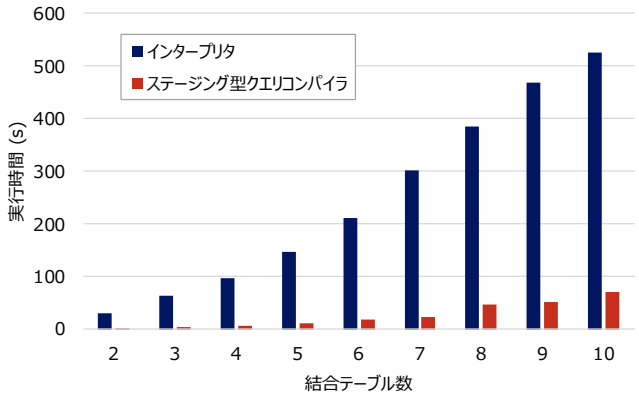


図 6　ネストループ結合評価結果

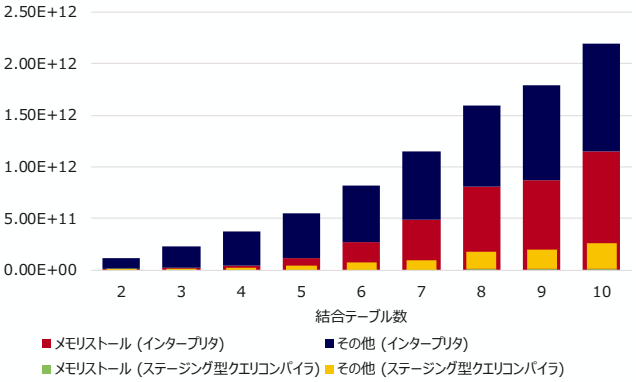


図 7　ネストループ結合実行時のサイクル数

クエリコンパイラが Volcano 方式のインタープリタに比べて，約 8〜20 倍高速であった．

この性能差について，実行命令数とサイクル数の観点から詳細に分析する．perf コマンドを用いて CPU 統計情報を採取したところ，Volcano 方式のインタープリタは，結合テーブル数が 2 の場合でクエリコンパイラの約 13 倍の命令を，結合テーブル数が 10 の場合で約 4 倍の命令を実行していた．結合テーブル数が 2 の場合における，Volcano 方式インタープリタの命令内訳は 1 の通りである．Volcano 方式のインタープリタは，(1) タプルの NULL チェック，(2) 仮想関数呼出し，(3) 述語評価時の構造体メンバ参照の 3 つがオーバヘッドとなっていた．

(1) は，所謂プル型のクエリ処理モデルに起因するもので，試作したクエリコンパイラではプッシュ型のクエリ処理を採用しているため，そうしたチェックを避けられる．また，(2) と (3) はインタープリタとコンパイラの違いに起因するものである．試作したクエリコンパイラでは，LMS を用いたコード生成の過程で関数呼出しや構造体のメンバ参照が部分評価され，取り除かれるため，より少ない命令数で実行が可能である．加えて，クエリコンパイラにおいては，コンパイラによる自動ベクトル化が行われ，最終的に約 13 倍という命令数の差になった．

次に，結合テーブル数毎のサイクル数について，メモリストールサイクルとその他のサイクルに分けて表示したグラフを図 7 に示す．図のように，結合テーブル数が増えるほどメモリ

表 1　Volcano 方式インタープリタの命令内訳 (結合テーブル数=2)

| | 命令数 | 割合 |
|---|---|---|
| NULL チェック | $3.0 \times 10^{10}$ | 13.6% |
| 仮想関数呼出し | $6.0 \times 10^{10}$ | 27.1% |
| 述語評価 | $8.0 \times 10^{10}$ | 36.2% |
| テーブルスキャン | $5.0 \times 10^{10}$ | 22.6% |
| その他 | $0.1 \times 10^{10}$ | 0.4% |
| 合計 | $22.1 \times 10^{10}$ | 100% |

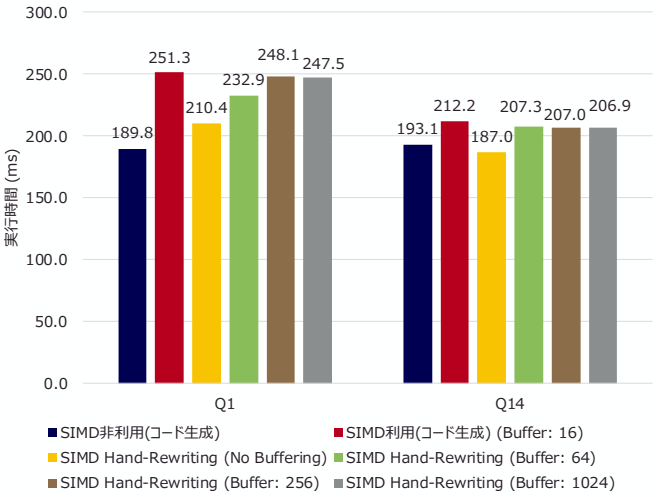


図 8　TPC-H 評価結果 (SF=1)

ストールが増大しており，最大で約 50%がメモリストールに終わっている．結果として，IPC(Instructions Per Cycle) の観点でも，Volcano 方式インタープリタが 0.9〜1.9 に対し，クエリコンパイラの IPC は 2.0〜4.5 であり，2 倍以上の差が開いた．

### 5.3　SIMD 命令活用の評価

SIMD 命令活用の効果を確認するため，TPC-H クエリを用いて評価を行う．使用するクエリは文献 [9] において，SIMD 化の効果が低かった Q1 と効果が高かった Q14 を用いて評価する．Q1 は，選択率が約 98%と高く，結合のともなわない集約クエリで，Q14 は，選択率が約 2%と低く，結合をともなう集約クエリである．

なお，データセットは，クエリ実行前にパースした上でメモリ上にプリロードする．プリロード時の格納形式は，LB2 におけるカラム指向のレコードバッファと同様であり，テーブルの各属性単位で列方向にメモリ領域を確保し，テーブルをスキャンする際は，先頭からタプル単位でアクセスしていく．

SIMD 利用しない場合と利用した場合のクエリ実行時間について，図 8 の SIMD 非利用 (濃青色) と SIMD 利用 (橙色) にそれぞれ示す．図 8 のように，SIMD 化により Q1 において約 32%，Q14 において約 10%性能が低下するという結果となった．

そこで，perf コマンドにより CPU 統計情報を採取した．その結果を，表 2 と表 3 に示す．Q14 では，分岐ミスは約半分に減少したものの，命令数が約 9%増加しており，バッファリングのペナルティが SIMD 化の効果を上回ってしまったことが原因であると推測できる．

表 2　CPU 統計情報 (TPC-H Q1)

| | SIMD 非利用 | SIMD 利用 |
|---|---|---|
| IPC | 1.84 | 1.86 |
| instructions | 2,785,684,557 | 2,933,436,777 |
| cycles | 1,513,071,944 | 1,579,272,780 |
| cycle_activity.stalls_mem_any | 381,347,705 | 295,388,040 |
| cache-misses | 19,398,410 | 19,379,645 |
| branch | 179,971,263 | 183,132,856 |
| branch-misses | 374,780 | 485,776 |
| L1-dcache-load-misses | 61,829,810 | 62,498,595 |
| LLC-load-misses | 692,110 | 556,278 |

表 3　CPU 統計情報 (TPC-H Q14)

| | SIMD 非利用 | SIMD 利用 |
|---|---|---|
| IPC | 1.27 | 1.38 |
| instructions | 1,435,151,001 | 1,565,911,667 |
| cycles | 1,128,176,211 | 1,136,294,807 |
| cycle_activity.stalls_mem_any | 269,245,919 | 247,053,673 |
| cache-misses | 3,544,330 | 5,050,291 |
| branch | 84,346,251 | 65,521,974 |
| branch-misses | 705,945 | 363,814 |
| L1-dcache-load-misses | 17,632,213 | 19,657,928 |
| LLC-load-misses | 655,471 | 512,490 |

上記の考察を踏まえ，生成された C コードに対して手書きで修正を加えることで，バッファリングのオーバヘッドを削減する 2 つの改善案についても評価を行った．1 つは，プリロード時に格納したバッファからスキャンしてくる際に，複数の要素を直接 SIMD レジスタにロードし，その場で述語評価を行う方法である．もう 1 つは，SIMD 対応選択オペレータがタプルを一時的にバッファする際のサイズを拡張し，複数の SIMD ベクタ分のタプルが蓄積した段階でまとめて述語評価を行う方法である．これらの方法を用いた場合のクエリ実行時間を，同じく図 8 に示す．直接ロードする場合が SIMD Hand-Rewriting (No Buffering)，バッファサイズを拡張した場合が SIMD Hand-Rewriting (Buffer 64〜1024) が対応する．

SIMD レジスタに直接ロードした場合，バッファリングのオーバヘッドが削減されることにより，選択率の低い Q14 では，SIMD 非利用時よりも約 3%と僅かだが性能が改善した．一方，選択率が高い Q1 では，合致したタプルの ID の書き出しオーバヘッドが大きく，SIMD 非利用時よりも依然として低い性能となった．また，バッファサイズは SIMD ベクタサイズ分 (=16) よりも大きくすることで，性能が改善されることを確認できたが，依然として SIMD 非利用時よりもオーバヘッドの方が大きいという結果となった．

## 6　おわりに

本論文では，第一に，インタープリタに対するステージング型クエリコンパイラの性能優位性について定量的な評価を行った．評価の結果，ステージング型クエリコンパイラは Volcano 方式インタープリタに比べて 8〜20 倍程度高速であることが観察され，その性能差の理由は（1）タプルの NULL チェック，

（2）仮想関数呼出し，（3）述語評価時の構造体メンバ参照であることが判明した．

本論文では，第二に，クエリコンパイラにおける SIMD 命令活用法を明らかにするため，SIMD 命令を生成するクエリコンパイラを設計，実装し，TPC-H ベンチマーク Q1, Q14 の処理時間で評価を行った．その結果，ステージング型クエリコンパイラにおいて，SIMD 命令による性能向上は限定的であり，最大でも 3%に留まることが観察された．このような限定的な性能向上は文献 [9] の従来研究と非一貫的なものであり，SIMD により性能向上発現条件の明確化には，さらなる研究が必要なことが判明した．今後は，SIMD 化の適用範囲，適用方法の再検討や，より大規模なデータセットでの評価を行う予定である．

## 謝　　辞

本研究の一部は，NEDO の「実社会の事象をリアルタイム処理可能な次世代データ処理基盤技術の研究開発」の委託により実施したものである．

## 文　　献

[1] G. Graefe and W. J. McKenna. The volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, April 1993.

[2] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[3] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1243–1254, 2013.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, 2015.

[5] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 307–322, 2018.

[6] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1907–1922, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, June 2014.

[8] PostgreSQL. https://www.postgresql.org/.

[9] 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.

[10] 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.

[11] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, June 2012.

[12] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005.