Project Tungsten
Bringing Spark Closer to Bare Metal

Reynold Xin @rxin

Spark Conference Japan

Feb 8, 2016

Hardware Trends

Storage

Network

CPU

Hardware Trends

	2010
Storage	50+MB/s (HDD)
Network	1Gbps
CPU	~3GHz

Hardware Trends

	2010	2016
Storage	50+MB/s (HDD)	500+MB/s (SSD)
Network	1Gbps	10Gbps
CPU	~3GHz	~3GHz

Hardware Trends

	2010	2016	
Storage	50+MB/s (HDD)	500+MB/s (SSD)	10X
Network	1Gbps	10Gbps	10X
CPU	~3GHz	~3GHz	☹

On the flip side その一方で

Spark IO has been optimized Spark のIOは最適化された

- Reduce IO by pruning input data that is not needed
  不要な入力データを排除することによる IO の削減
- New shuffle and network implementations (2014 sort record)
  新しいシャッフルとネットワークの実装 (2014年ソート記録)

Data formats have improved データフォーマットが改善された

- E.g. Parquet is a “dense” columnar format
  Parquet は「密」なカラムナフォーマット

Goals of Project Tungsten　Project Tungsten のゴール

Substantially improve the **memory and CPU** efficiency of Spark backend execution and push performance closer to the limits of modern hardware.

Spark の「バックエンドの実行」のメモリとCPU効率を実質的に改善することと、性能をモダンなハードウェアの限界に近づけること

Note the focus on "execution" not "optimizer": very easy to pick broadcast join that is 1000X faster than Cartesian join, but hard to optimize broadcast join to be an order of magnitude faster.

「最適化」ではなく「実行」に注目していることに注意：broadcast join を Cartesian joinより 1000 倍早くするのはとても簡単だが、broadcast join をケタ違いに早くなるよう最適化するのは難しい

Phase 1

Foundation

立ち上がり

Memory Management — **メモリ管理**

Code Generation — **コード生成**

Cache-aware Algorithms — **(ハードウェア)キャッシュを意識したアルゴリズム**

Phase 2

Order-of-magnitude Faster

ケタ違いの高速化

Whole-stage Codegen — **全ステージコード生成**

Vectorization — **ベクトル化**

Phase 1
Laying The Foundation

Optimized data representations

最適化されたデータ表現

Java objects have two downsides:

- Space overheads
- Garbage collection overheads

Java オブジェクトには2つのマイナス面がある
・空間のオーバーヘッド
・ガベージコレクションのオーバーヘッド

Tungsten sidesteps these problems by performing its own manual memory management.

Tungsten はこれらの問題を、独自のメモリ管理を行うことで回避する

The overheads of Java objects

Java オブジェクトのオーバーヘッド

“abcd”

Native: 4 bytes with UTF-8 encoding

Java: 48 bytes

ネイティブ：UTF-8エンコーディングで4バイト

Java: 48バイト

```
java.lang.String object internals:
OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
     0     4        (object header)                           ...
     4     4        (object header)                           ...
     8     4        (object header)                           ...
    12     4 char[] String.value                              []
    16     4    int String.hash                               0
    20     4    int String.hash32                             0
Instance size: 24 bytes (reported by Instrumentation API)
```

インスタンスサイズ: 24バイト (Instrumentation API による)

12 byte object header

12バイト：オブジェクトヘッダ

20 bytes of overhead + 8 bytes for chars

20バイトのオーバーヘッド + 8バイトの文字

8 byte hashcode

8バイト：ハッシュコード

sun.misc.Unsafe

安全性のチェックを行わずに直接的にメモリを操作する JVM 内部 API (すなわち “unsafe”)

on- と off-heap メモリの両方でデータ構造をビルドする際にこのAPIを利用している

JVM internal API for directly manipulating memory without safety checks (hence “unsafe”)

We use this API to build data structures in both on- and off-heap memory

フラットなデータ構造

ポインタを保持したデータ構造

複雑な例

Code generation

表現ロジックの汎用的な評価は JVM 上では非常にコストが高い

Generic evaluation of expression logic is very expensive on the JVM

- Virtual function calls
- Branches based on expression type
- Object creation due to primitive boxing
- Memory consumption by boxed primitive objects

仮想関数呼び出し

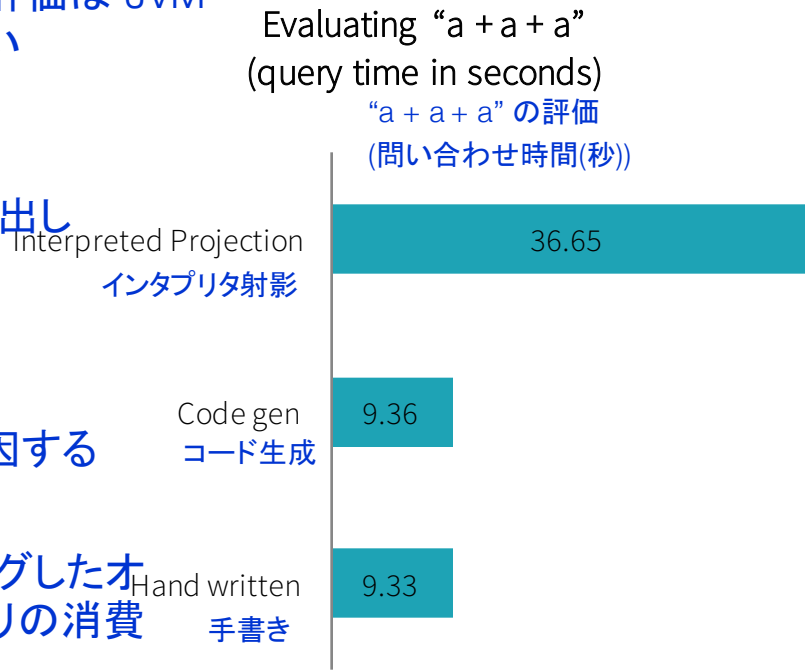
型表現に基づくブランチ

プリミティブのボクシングに起因するオブジェクトの生成

プリミティブをボクシングしたオブジェクトによるメモリの消費

Generating custom bytecode can eliminate these overheads

カスタムバイトコードの生成によりこれらのオーバーヘッドを排除できる



Code generation

Tungsten uses the Janino compiler to reduce code generation time

Tungsten はコードの生成時間を減らすために Janino コンパイラを利用する

Spark 1.5 added ~100 UDF's with code gen:

Spark 1.5 は code gen で ~100 の UDF を追加した:

```
AddMonths
ArrayContains
Ascii
Base64
Bin
CheckOverflow
CombineSets
Contains
CountSet
Crc32
DateAdd
DateDiff
DateFormatClass
DateSub
DayOfMonth
DayOfYear
Decode
Encode
EndsWith
Explode
Factorial
FindInSet
FormatNumber
FromUTCTimestamp
FromUnixTime
GetArrayItem
GetJsonObject
GetMapValue
Hex
InSet
InitCap
IsNaN
IsNotNull
IsNull
LastDay
Length
Levenshtein
Like
Lower
MakeDecimal
Md5
Month
MonthsBetween
NaNvl
NextDay
Not
PromotePrecision
Quarter
RLike
Round
Second
Sha1
Sha2
ShiftLeft
ShiftRight
ShiftRightUnsigned
SortArray
SoundEx
StartsWith
StringInstr
StringRepeat
StringReverse
StringSpace
StringSplit
StringTrim
StringTrimLeft
StringTrimRight
TimeAdd
TimeSub
ToDate
ToUTCTimestamp
TruncDate
UnBase64
Unhex
UnixTimestamp
```

AlphaSort Cache-friendly Sorting

AlphaSort キャッシュフレンドリーなソート

Idea: minimize random memory accesses

アイデア：ランダムメモリアクセスの最小化

Naïve layout

単純なレイアウト

Cache friendly layout

キャッシュフレンドリーなレイアウト

- Store prefixes of sort keys inside the sort pointer array
- Short-circuit by comparing prefixes first
- Swap only pointers and prefixes

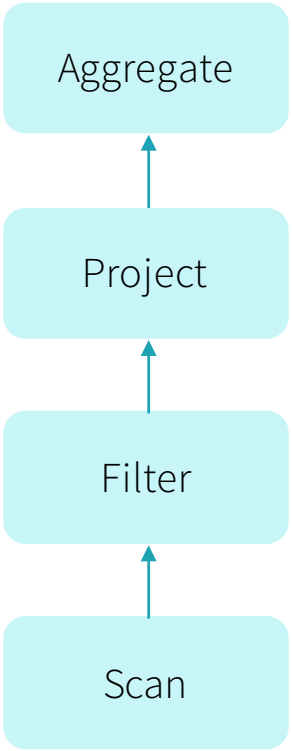
・ソートのポインタ配列の中にソートキーのプレフィックスを格納

・先にプレフィックスを比較することによる短絡評価(ショートサーキット)

・ポインタとプレフィックスのみのスワップ

Phase 2
Order-of-magnitude Faster

```
select count(*) from store_sales
where ss_item_sk = 1000
```



Volcano Iterator Model

Standard for 30 years: almost all databases do it

30年間標準

ほぼすべてのデータベースが行っている

Each operator is an “iterator” that consumes records from its input operator

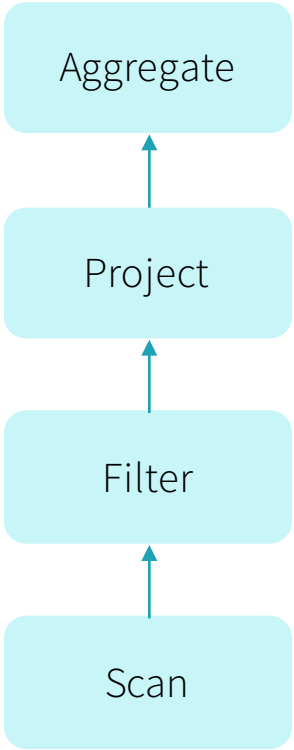
各オペレータは入力オペレータからのレコードを消費する「イテレータ」

```
class Filter {
  …
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

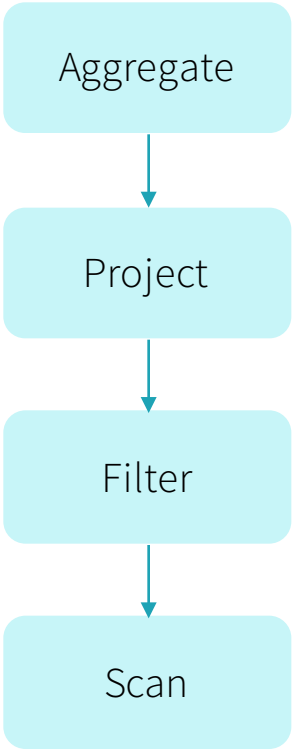
data flow

データフロー



call stack to process a row

一行を処理するためのコールスタック



Downside of the Volcano Model

ボルケーノモデルの欠点

1. Too many virtual function calls
   - at least 3 calls for each row in Aggregate

仮想関数呼び出しが多すぎる

Aggregateで少なくとも3回の呼び出し

2. Extensive memory access
   - “row” is a small segment in memory (or in L1/L2/L3 cache)

メモリアクセスが広範

「行」はメモリ(またはL1/L2/L3キャッシュ)中の小さな断片

3. Can’t take advantage modern CPU features
   - SIMD, pipelining, prefetching …

モダンなCPUの機能の恩恵が受けられない

SIMD、パイプライン、プリフェッチ、・・・

What if we hire a college freshman to implement this query in Java in 10 mins?

もし10分間で Java でこのクエリを実装するために、大学1年生を雇ったら？

```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

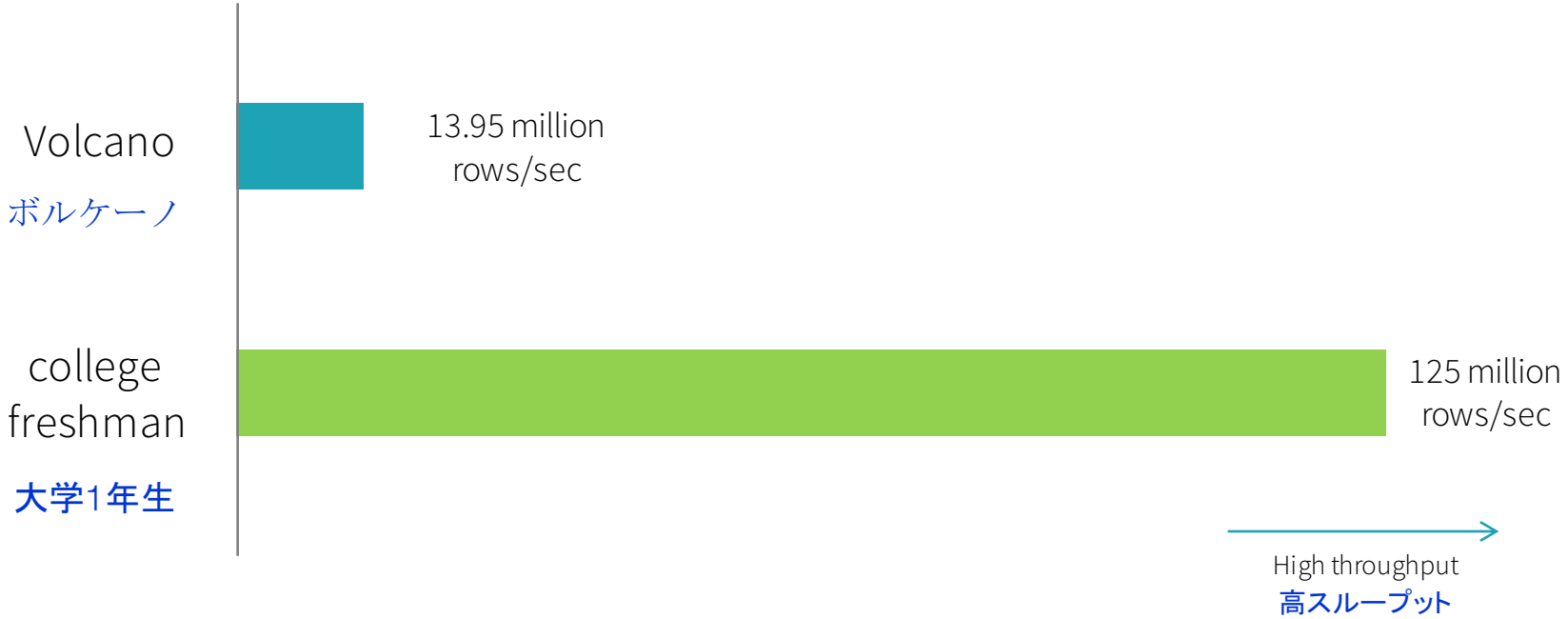
Volcano model
30+ years of database research

ボルケーノモデル
30年以上のデータベースの研究

vs

college freshman
hand-written code in 10 mins

大学1年生
10分間の手書きのコード



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

How does a student beat 30 years of research?

どうして学生が30年の研究に勝ったのか？

Volcano

1. Many virtual function calls

   大量の仮想関数呼び出し

2. Data in memory (or cache)

   データをメモリに(もしくはキャッシュに)

3. No loop unrolling, SIMD, pipelining

   ループ展開、SIMD、パイプラインが無い

hand-written code

1. No virtual function calls

   仮想関数呼び出しがない

2. Data in CPU registers

   データを CPU のレジスタに

3. Compiler loop unrolling, SIMD, pipelining

   コンパイラによるループ展開、SIMD、パイプライン

Whole-stage Codegen

全ステージ コード生成

Fusing operators together so the generated code looks like hand optimized code:

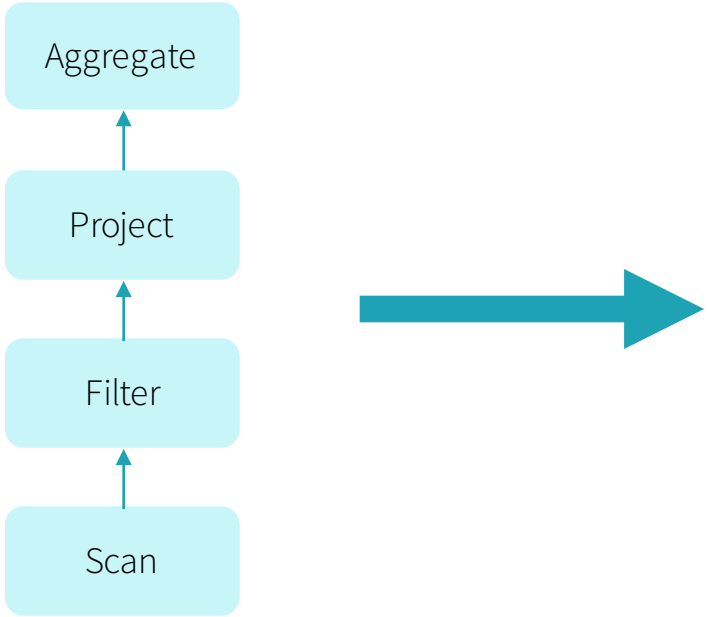
複数のオペレータを一括で扱い結合することで、生成されたコードが手動で最適化されたコードのようにみえる

- Data stays in registers, rather than cache/memory
- Minimizes virtual function calls
- Compilers can unroll loops and use SIMD instructions

- データはキャッシュ / メモリよりもレジスタの中に残る
- 仮想関数呼び出しを最小化する
- コンパイラがループを展開することができ、SIMD命令を使う

Whole-stage Codegen: Spark as a "Compiler"

全ステージ コード生成：「コンパイラ」としての Spark




```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

But there are things we can't fuse

しかし結合させることができないものがある

Complicated I/O

- CSV, Parquet, ORC, …

複雑な I/O

- CSV, Parquet, ORC, …

External integrations

- Python, R, scikit-learn, TensorFlow, etc

外部とのインテグレーション

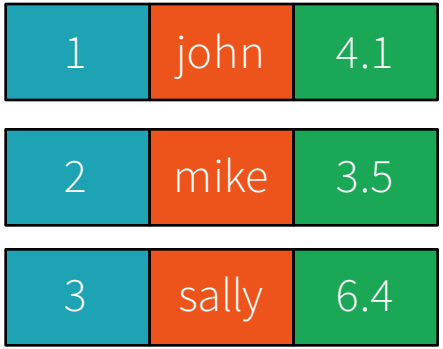
- Python, R, scikit-learn, TensorFlow, など

Don't want to compile a CSV reader inline for every query!

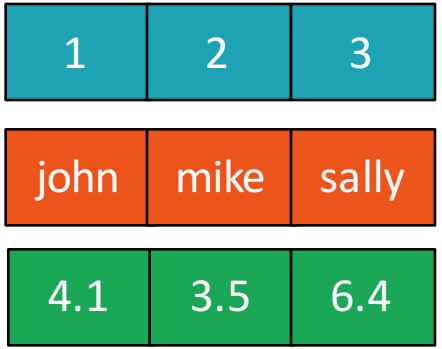
クエリごとに CSV reader をコンパイルしたくない！

Vectorization
ベクトル化

In-memory
Row Format
メモリ内の行フォーマット



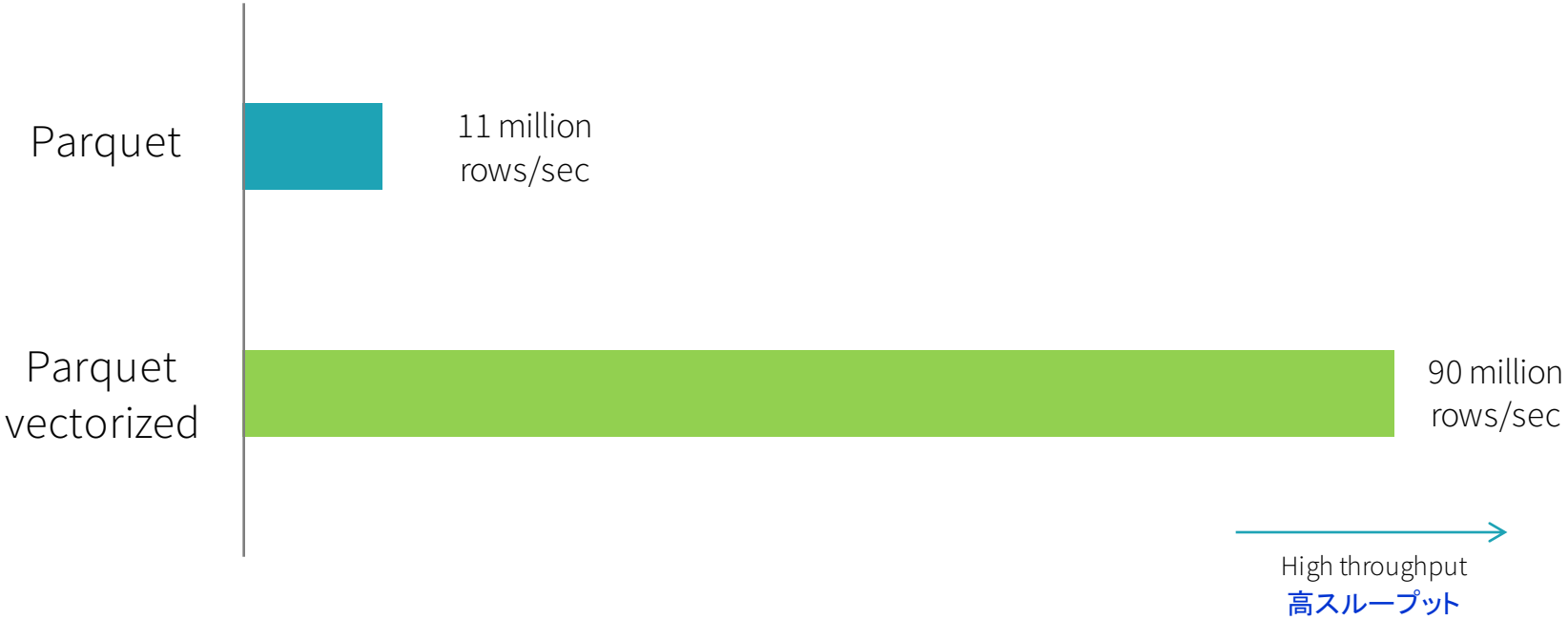
In-memory
Column Format
メモリ内の列フォーマット



Why Vectorization?

なぜベクトル化？

1. Modern CPUs are better at doing one thing over and over again (rather than N different things over and over again)
   モダンな CPU はひとつのことを何度も何度も行うのが得意 (N 個の異なることを何度も何度も行うよりも)
2. Most high-performance external systems are already columnar (numpy, TensorFlow, Parquet) so it is easier to integrate with
   ハイパフォーマンスな外部システムの多くは、既にカラムナ (numpy, TensorFlow, Parquet) であるため、インテグレートするのがより簡単



databricks

Note: End-to-end, single thread, single column, and data originated in Parquet on disk

Phase 1
Spark 1.4 - 1.6

Memory Management
Code Generation
Cache-aware Algorithms

メモリ管理
コード生成
キャッシュを意識したアルゴリズム

Phase 2
Spark 2.0+

Whole-stage Codegen
Vectorization

全ステージ コード生成
ベクトル化

ありがとうございました

@rxin