Apache Arrow #ArrowTokyo

須藤功平
株式会社クリアコード

Apache Arrow東京ミートアップ2018
2018-12-08

自己紹介：名前

- ✓ 須藤功平（すとうこうへい）
  「と」はにごらない！
- ✓ よく使うアカウント名：kou
  KOUhei
  - ✓ ↑を使えないときのアカウント名：ktou
    Kouhei suTOU

自己紹介：プログラミング

- ✓ Rubyが好き
  - ✓ 2004-01からコミッター
  - ✓ 130くらいのライブラリーをメンテナンス
    詳細はRubyKaigi 2018でのキーノートを参照
  - ✓ Rubyでプレゼンツールも作っている（このツール）
- ✓ C/C++を書いている時間も結構ある

自己紹介：C/C++を書く理由

- ✓ RubyでC/C++のライブラリーを使うため！
  Rubyを書くためにC/C++を書く
- ✓ Apache ArrowをC++で開発しているのもそう
  - ✓ テストはRubyで書いている

自己紹介：Apache Arrowの開発

- ✓ 2016-12-21に最初のコミット
- ✓ 2017-03-16にGLibバインディングを寄贈
- ✓ 2017-05-10にコミッター
- ✓ 2017-09-15にPMCメンバー
- ✓ 2018-12-06現在コミット数3位（224人中）

自己紹介：仕事

- ✓ 株式会社クリアコードの代表取締役
  - ✓ 自由なソフトウェアでビジネスをする会社
    自由なソフトウェア：OSSが参考にしたやつ
- ✓ 私の最近の業務内容
  - ✓ 自由なソフトウェアの推進
    - ✓ 例：SpeeeさんがOSSを開発することをサポート
  - ✓ データ処理ツールの開発事業立ち上げ（New!）

データ処理ツールの開発事業

- ✓ データ分析をする事業じゃない
- ✓ データ分析をする人たちが使うツールを開発する事業
  - ✓ Apache Arrowはそのために有用なツール
    Apache Arrowの開発に参加し始めたのはRubyで使いたかったから
    事業立ち上げのためにApache Arrowの開発に参加し始めたわけではない
    Apache Arrowの開発に参加していたら面白そうと思えてきた
  - ✓ 募集：開発して欲しい・開発したい（転職したい）

Apache Arrow

各種言語で使えるインメモリーデータ処理プラットフォーム

実現すること

データ処理の効率化
（大量データが対象）

効率化のポイント

- ✓ 速度
  - ✓ 速いほど効率的
- ✓ 実装コスト
  - ✓ 低いほど効率的

速度向上方法

- ✓ 遅い部分を速く
- ✓ 高速化できる部分を最適化

遅い部分

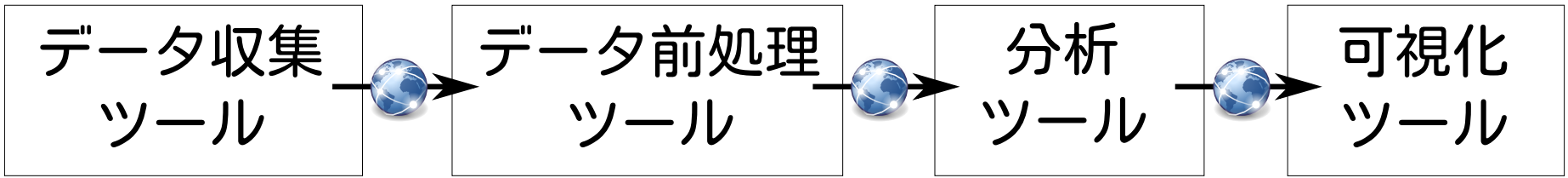
データ交換

データ交換

- ✓ データ処理ツール間で必要
- ✓ データ処理システム
  - ✓ 複数ツールを組み合わせて実現
  - ✓ データ処理システムではデータ交換が必須

データ処理システム例

- データ交換



データ交換処理

1. シリアライズ
   データをバイト列へ変換
2. 転送
   バイト列を別ツールに渡す
3. デシリアライズ
   バイト列からデータを復元

データ交換処理：必要なリソース

1. シリアライズ：CPU
2. 転送：I/O（ネットワーク・ストレージ・メモリー）
3. デシリアライズ：CPU

Ruby+JSONでデータ交換

```ruby
# 1000要素の数値配列
n = 1000
numbers = n.times.collect {rand}
# シリアライズ
JSON.dump(numbers, output)
# デシリアライズ
JSON.load(input)
```

Ruby+JSONの速度の傾向

n	シリアライズ	デシリアライズ
1000	0.011秒	0.004秒
10000	0.093秒	0.037秒
100000	0.798秒	0.369秒

注：ストレージI/Oなしで計測

データ量の増加と同じくらいの比率で遅くなる

データ交換の高速化

- ✓ データ量が増加すると
  シリアライズ・デシリアライズ速度が劣化
- ✓ 速度劣化を抑えられれば高速化可能

Apache Arrowのアプローチ

- ✓ データフォーマットを定義
  - ✓ シリアライズ・デシリアライズが速い
  - ✓ データ量増加に影響を受けにくい
- ✓ このフォーマットの普及
  - ✓ 各種言語で読み書き処理を実装

Ruby+Apache Arrowでデータ交換

```ruby
# 1000要素の数値配列
n = 1000
numbers = Arrow::Int32Array.new(n.times.collect {rand})
table = Arrow::Table.new("number" => numbers)
# シリアライズ
table.save(output)
# デシリアライズ
Arrow::Table.load(input)
```

Ruby+Apache Arrowの速度の傾向

n	シリアライズ	デシリアライズ
1000	0.0003秒	0.0004秒
10000	0.0004秒	0.0004秒
100000	0.0015秒	0.0004秒

注：ストレージI/Oなしで計測

全体的に速い+デシリアライズ速度が一定

Apache Sparkでの高速化事例

- Spark⇔PySpark間でデータ交換
  - 従来：pickleでシリアライズ
    pickle：Python標準のシリアライズ方法
  - Apache Arrowを使うことで数十倍レベルの高速化
  - Spark⇔sparklyrも同様
- 詳細：上新さんの話

Apache Arrowフォーマットの特徴

- メモリー上でのフォーマットを変換しない
  - JSONは「数値」を「数字」に変換
  - 例：29（1バイト整数）→"29"（2バイト文字列）
- シリアライズ時：変換不要
- デシリアライズ時：パース不要

メモリーマップの活用

- メモリーマップ機能
  - ファイルの内容をメモリー上のデータのようにアクセスできる機能
  - readせずにデータを使える（データコピー不要）
- パース不要+メモリーマップ
  - デシリアライズ時にメモリー確保不要
  - 「転送」コスト削減

遅い部分の高速化まとめ

- ✓ 遅い部分を速く
  - ✓ **データ交換**を速く
- ✓ 高速化できる部分を最適化

高速化できる部分

大量データの計算

大量データの計算の高速化

- ✓ 各データの計算を高速化
- ✓ まとまったデータの計算を高速化

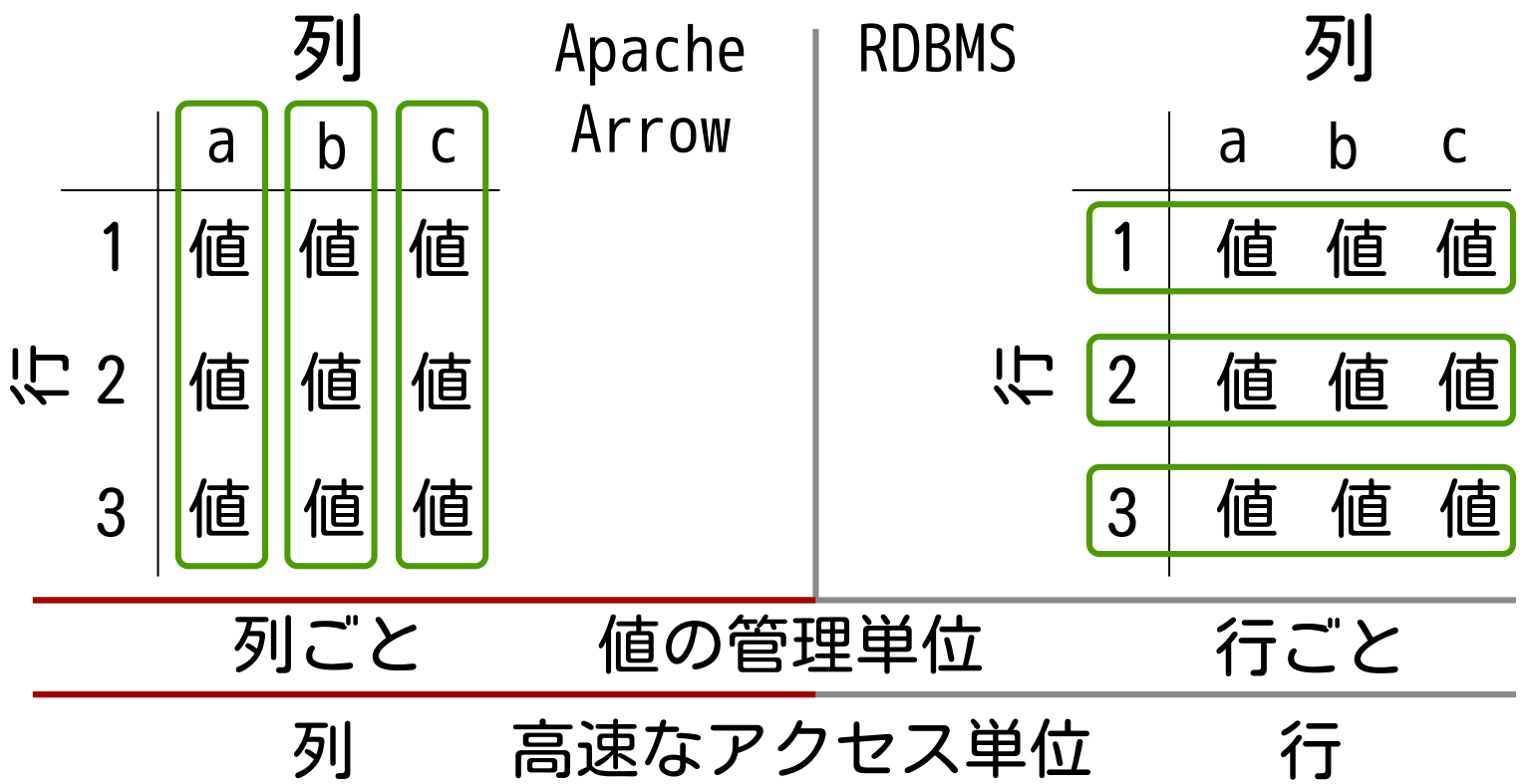
各データの計算の高速化

- データを局所化
  - CPUのキャッシュメモリーを活用できる
- 局所化に必要な知識
  - データの使われ方
  - 局所化：一緒に使うデータを近くに置く

想定ユースケース

- OLAP (OnLine Analytical Processing)
  - データから探索的に知見を探し出すような処理
- 列単位の処理が多い
  - 集計処理・グループ化・ソート…

OLAP向きのデータの持ち方



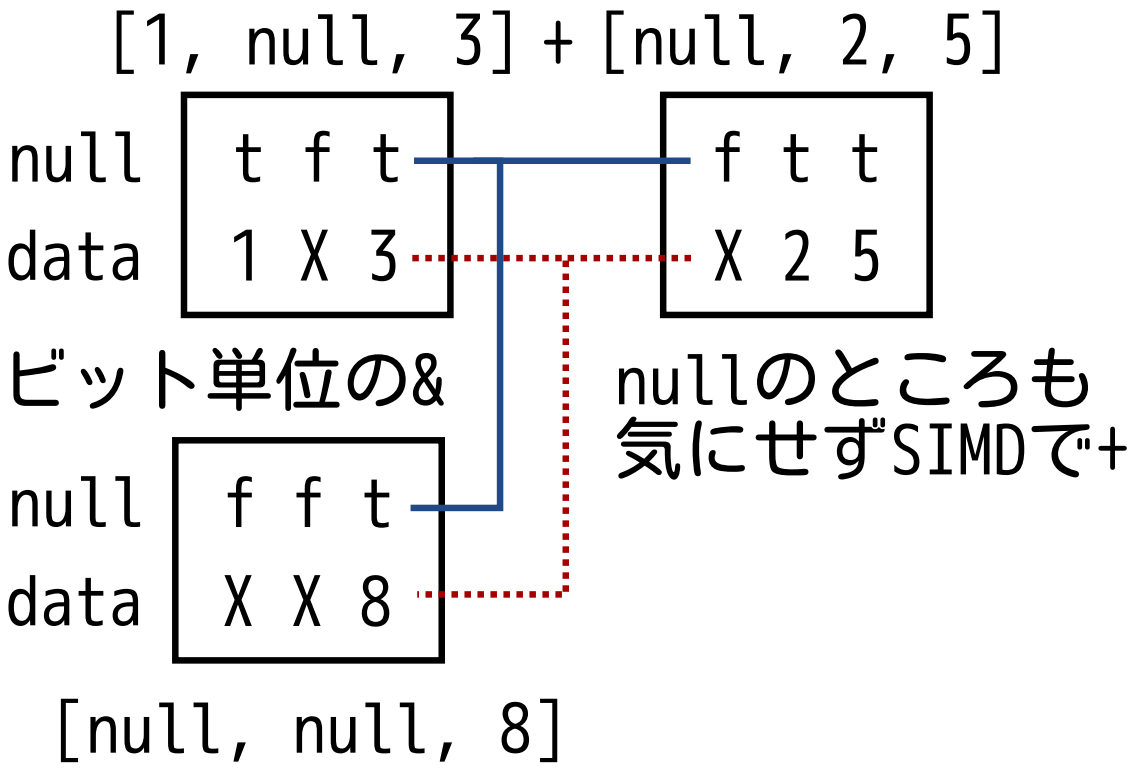
まとまったデータの計算を高速化

- ✓ SIMDを活用
  Single Instruction Multiple Data
- ✓ スレッドを活用

SIMDを活用

- CPU：データをまとめてアライン
  アライン：データの境界を64の倍数とかに揃える
- GPUの活用
- 条件分岐をなくす
  - null/NA用の値は用意せずビットマップで表現
    Is it time to stop using sentinel values for null / NA values?

条件分岐とnull


[1, null, 3] + [null, 2, 5]
null
t f t
data
1 X 3
f t t
X 2 5
ビット単位の&
nullのところも
気にせずSIMDで+
null
f f t
data
X X 8
[null, null, 8]

スレッド活用時のポイント

- 競合リソースを作らない
  - リソースロックのオーバーヘッドで遅くなりがち
- アプローチ
  - リソースを参照するだけ
  - 各スレッドにコピー

Apache Arrowとスレッド

- ✓ データはリードオンリー
  - ✓ スレッド間でオーバーヘッドなしで共有可能
- ✓ データコピーは極力避けたい
  - ✓ データ交換時もスレッド活用時も

高速化のまとめ

- ✓ 速度
  - ✓ 遅い処理（**データ交換処理**）を高速化
  - ✓ 速くできる処理（**大量データの計算**）を最適化
- ✓ 実装コスト
  - ✓ 低いほど効率的

実装コストを下げる

- ✓ 共通で使いそうな機能をライブラリー化
  - ✓ メリットを受ける人たちみんなで協力して開発
  - ✓ 最適化もがんばる
- ✓ Apache Arrowの実装コストは下がらない
  - ✓ Apache Arrowを使うツールの実装コストが下がる
    実装コストが下がる：ツール開発者のメリット

共通で使いそうな機能

- ✓ 高速なデータフォーマット
- ✓ フォーマット変換機能
- ✓ 効率的なデータ交換処理
- ✓ 高速なデータ処理ロジック

実装コストのまとめ

- ✓ 速度
- ✓ 実装コスト
  - ✓ 共通で使いそうな機能を**ライブラリー化**
  - ✓ **みんなで協力**して開発

Apache Arrowが扱えるデータ

- ✓ 表・データフレーム
- ✓ 多次元配列

表・データフレーム

- 行と列で管理する2次元データ
  RDBMSでいう表
  - すべての行は同じ列を持つ
  - 各列は異なる型にできる
  - すべての型でnullをサポート
- 2次元配列との違い
  - 2次元配列はすべての値が同じ型

扱える型：真偽値・数値

- ✓ 真偽値（1ビット）
- ✓ 整数（{8, 16, 32, 64}ビット{非負, }整数）
- ✓ 浮動小数点数（{16, 32, 64}ビット）
- ✓ 128ビット小数

リトルエンディアンのみ対応

ビッグエンディアンに対応させようという動きもある：ARROW-3476

真偽値・数値：データの配置

固定長データなので連続して配置

32ビット整数：[1, 2, 3]

```
0x01 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x03 ...
```

扱える型：文字列・バイト列

- ✓ UTF-8文字列
- ✓ バイナリーデータ（{可変, 固定}長）

文字列・バイト列：データの配置

実データバイト列+長さ配列に配置

UTF-8文字列：["Hello", "", "!"]
実データバイト列："Hello!"
長さ配列：[0, 5, 5, 6]
i番目の長さ：長さ配列[i+1] - 長さ配列[i]
i番目のデータ：
　実データバイト列[長さ配列[i]..長さ配列[i+1]]

扱える型：日付・タイムスタンプ

- 日付
  - UNIXエポックからの経過日数（32bit）
  - UNIXエポックからの経過ミリ秒数（64bit）
- タイムスタンプ（64ビット整数）
  - UNIXエポックからの経過{,ミリ}秒数
  - UNIXエポックからの経過{マイクロ,ナノ}秒数

扱える型：時間

- 時間
  - 深夜0時からの経過{, ミリ}秒数
    （32bit整数）
  - 深夜0時からの経過{マイクロ, ナノ}秒数
    （64bit整数）

扱える型：リスト

- ✓ 0個以上の同じ型の値を持つ
- ✓ 例：32ビット整数のリスト
  - ✓ 👍0要素：[]
  - ✓ 👍2要素：[2, 3]
  - ✓ 👎型が違う：[1, "X"]

リスト：データの配置

実データの配列+長さ配列に配置

32bit整数：[[1, 2], null, [3]]
実データの配列：[1, 2, 3]
長さ配列：[0, 2, 2, 3]
i番目の長さ：長さ配列[i+1] - 長さ配列[i]
i番目のデータ：
　実データの配列[長さ配列[i]..長さ配列[i+1]]

扱える型：構造体

- ✓ 1個以上のフィールドを持つ
- ✓ 各フィールドは別の型にできる
- ✓ 例：aは32ビット整数、bはUTF-8文字列
  - ✓ 👍全部ある：`{a: 1, b: "X"}`
  - ✓ 👍nullもOK：`{a: 1, b: null}`
  - ✓ 👎bがない：`{a: 1}`

構造体：データの配置

フィールド分のデータを個別に持つ

```
[{a : 1, b: "X"}, {a: 2, b: null}]
a : [1, 2]
b : ["X", null]
```

扱える型：{疎, 密}共用体

- ✓ 1個以上のフィールドを持つ
- ✓ 各フィールドは別の型にできる
- ✓ どれかひとつのフィールドの値のみ設定
- ✓ 例：aは32ビット整数、bはUTF-8文字列
  - ✓ 👍aだけある：`{a: 1}`
  - ✓ 👎2つある：`{a: 1, b: "X"}`

疎共用体：データの配置

フィールドと整数を対応付ける配列を持つ
どのフィールドを使うか配列を持つ
フィールド分のデータを個別に持つ
各データはすべての要素を持つ

```
[{a : 1}, {b: "X"}, {b: "Y"}]
フィールドID配列：[2, 9] aは2、bは9
使用フィールド配列：[2, 9, 9]
a：[1, null, null]
b：[null, "X", "Y"]
```

密共用体：データの配置

フィールドと整数を対応付ける配列を持つ
どのフィールドを使うか配列を持つ
フィールド分のデータを個別に持つ
各データは必要な要素だけを持つ
各データの何番目の要素を使うか配列を持つ

```
[{a : 1}, {b: "X"}, {b: "Y"}]
フィールドID配列：[2, 9] aは2、bは9
使用フィールド配列：[2, 9, 9]
要素オフセット配列：[0, 0, 1] a[0], b[0], b[1]
a：[1]
b：["X", "Y"]
```

扱える型：辞書

- 名義尺度なカテゴリーデータ
  （統計っぽい説明）
- 各値にIDを割り当て、そのIDで値を表現
  （実装よりの説明）
- 例：["X","X","Y"]を辞書型の列にした場合
  - 値：[0, 0, 1]
  - IDの割り当て：{"X": 0, "Y": 1}

辞書：データの配置

データの配列と何番目のデータを使うかの配列を持つ

```
["X", "X", "Y"]
データ配列：["X", "Y"]
インデックス配列：[0, 0, 1]
```

表・データフレームのまとめ

- ✓ 表・データフレーム
  - ✓ 2次元データ
  - ✓ 各列は異なる型にできる
  - ✓ 型はたくさんある
- ✓ 多次元配列

多次元配列

- ✓ n次元のデータを扱う
- ✓ 要素はすべて同じ型
- ✓ 型は整数・浮動小数点数のみ
- ✓ 密・疎ともに対応
  疎は対応中：ARROW-854 [Format] Support sparse tensor
- ✓ 詳細：村田さんの話

多次元配列のまとめ

- ✓ データフレーム
- ✓ 多次元配列
  - ✓ n次元データ
  - ✓ すべての要素は**同じ型**の値
  - ✓ 型は**整数・浮動小数点数**のみ
  - ✓ **疎・密**両方サポート

Apache Arrowが提供する機能

- ✓ 高速なデータフォーマット
- ✓ フォーマット変換機能
- ✓ 効率的なデータ交換処理
- ✓ 高速なデータ処理ロジック

フォーマット変換機能

- Apache Arrowフォーマット
  - インメモリー用のフォーマット
  - 永続化向きではない
- 永続化されたデータを使う場合
  - 永続化に適したフォーマットで保存
  - 読み込み時にApache Arrowに変換して
    インメモリーでの処理にApache Arrowを使う

対応フォーマット：CSV

- よく使われているフォーマット
- 亜種が多くてパースが大変

対応フォーマット：Apache Parquet

- ✓ 永続化用フォーマット
  - ✓ 列単位でデータ保存：Apache Arrowと相性がよい
- ✓ 小さい
  - ✓ 列単位の圧縮をサポート
- ✓ 速い
  - ✓ 必要な部分のみ読み込める（I/Oが減る）

対応フォーマット：Apache ORC

- 永続化用フォーマット
  - 列単位でデータ保存：Apache Arrowと相性がよい
  - Apache Parquetに似ている
- Apache Hive用に開発
  - 今はHadoopやSparkでも使える

対応フォーマット：Feather

- 永続化用フォーマット
  - 列単位でデータ保存：Apache Arrowと相性がよい
  - データフレームを保存
- PythonとR間のデータ交換用
- **今は非推奨！**
  - Apache Arrowを使ってね

対応中フォーマット：JSON

- ✓ よく使われているフォーマット

対応中フォーマット：Apache Avro

- RPCフレームワーク
  - データフォーマットも提供
- 永続化にも使えるフォーマット
- ARROW-1209
  - ちょっと止まっている

非公式対応フォーマット：MDS

- Multiple-Dimension-Spread
  - 紹介スライド
  - スキーマレスなカラムナーフォーマット
  - Yahoo! Japan開発
- 詳細：井島さんの話

フォーマット変換機能まとめ

- ✓ 高速なデータフォーマット
- ✓ フォーマット変換機能
  - ✓ **永続化データ**を処理するために必要
- ✓ 効率的なデータ交換処理
- ✓ 高速なデータ処理ロジック

効率的なデータ交換処理

- Plasma
  - 同一ホスト上のプロセス間でのデータ共有
- Apache Arrow Flight
  - Apache ArrowベースのRPC
- DB連携
  - 各種DBのレスポンスをApache Arrowに変換

Plasma

- 同一ホストでのデータ共有システム
  - サーバー：データ管理
  - クライアント：データ登録・参照
- Apache Arrowフォーマットは使っていない
  - 任意のバイト列を共有
- ユースケース：マルチプロセス連携
  - もともとRay（分散計算システム）用に開発

Apache Arrow Flight

- Apache ArrowベースのRPCフレームワーク
  - クライアント：リクエストデータフレームを送信
  - サーバー：レスポンスデータフレームを返信
- gRPCベース

DB連携

- DBのレスポンスをApache Arrowに変換
- 対応済み
  - Apache Hive, Apache Impala
- 対応予定
  - MySQL/MariaDB, PostgreSQL, SQLite
    MySQLは畑中さんの話の中にPoCが！
  - SQL Server, ClickHouse

効率的なデータ交換処理のまとめ

- ✓ 高速なデータフォーマット
- ✓ フォーマット変換機能
- ✓ 効率的なデータ交換処理
  - ✓ **同一ホスト内**での高速なデータ交換
  - ✓ **異なるホスト間**での高速なデータフレーム交換
  - ✓ **DBのレスポンス**をApache Arrowに変換
- ✓ 高速なデータ処理ロジック

高速なデータ処理ロジック

- ✓ 高速なデータフレーム処理
  - ✓ CPU：コンパイラーの最適化でベクトル化
  - ✓ GPU：RAPIDSが[cuDF]を開発
- ✓ 高速なクエリー処理エンジン
  - ✓ Gandiva

Gandiva

- SQLレベルのクエリーの実行エンジン
  - 四則演算だけではない
  - 集計・フィルター・結合などもカバー
    GROUP BY, WHERE, JOIN
- 実行時に最適化

Gandiva：実行時に最適化

- クエリーを解析して最適化
  - 不要な処理を削除・処理順番を入れ替え
    Gandivaでやるようになるんじゃないかな
- クエリーをJITコンパイルして実行
  - インタプリターが実行、ではない
  - 最適化済みネイティブコードにして実行
    実行環境のCPUに合わせてベクトル化とか

高速なデータ処理ロジックのまとめ

- ✓ 高速なデータフォーマット
- ✓ フォーマット変換機能
- ✓ 効率的なデータ交換処理
- ✓ 高速なデータ処理ロジック
  - ✓ CPUでもGPUでも**最適化**
  - ✓ **クエリーレベル**の高速な実行エンジン

対応言語

- ✓ C, C#, C++, Go, Java, JavaScript, Lua
- ✓ MATLAB, Python, R, Ruby, Rust

非公式実装：

- ✓ Julia (Arrow.jl)

実装方法

- ネイティブ実装
  - C#, C++, Go, Java, JavaScript, Julia, Rust
  - その言語になじむ
- C++バインディング
  - C, Lua, MATLAB, Python, R, Ruby
  - ホスト言語が遅めでも速い実装になる

C#の実装状況

- ✓ 未対応の型
  - ✓ 16bit浮動小数点数・小数・構造体・共用体・辞書
- ✓ 多次元配列未対応
- ✓ 計算未対応
- ✓ Plasma・Flight未対応

C++の実装状況

- ✓ すべて実装済み
  - ✓ 一番実装が進む
  - ✓ C++実装のバインディングとして開発する言語があるため開発者が多い
- ✓ Apache Parquet実装も取り込んだ

Goの実装状況

- ✓ 未対応の型
  - ✓ 16ビット浮動小数点数・小数・共用体
- ✓ 計算対応（Gandivaは未対応）
- ✓ フォーマット変換はCSVのみ対応
- ✓ Plasma・Flight未対応

Javaの実装状況

- ✓ 未対応の型
  - ✓ 16ビット浮動小数点数
- ✓ 多次元配列未対応
- ✓ 計算対応（Gandivaも対応）
- ✓ フォーマット変換未対応
- ✓ JDBCを使ったDB連携対応

JavaScriptの実装状況

- ✓ TypeScript実装
  - ✓ WebブラウザーでもNode.js上でも動く
- ✓ 多次元配列未対応
- ✓ 計算対応（Gandivaは未対応）
- ✓ フォーマット変換未対応
- ✓ Plasma・Flight未対応

Juliaの実装状況

- ✓ 未対応の型
  - ✓ 16bit浮動小数点数・小数・構造体・共用体
- ✓ 多次元配列未対応
- ✓ 計算未対応
- ✓ フォーマット変換未対応
- ✓ Plasma・Flight未対応

Rustの実装状況

- ✓ 未対応の型
  - ✓ 小数・バイナリーデータ・共用体・辞書
- ✓ 計算対応（Gandivaは未対応）
- ✓ フォーマット変換はCSV・Parquet対応
- ✓ Plasma・Flight未対応

C・Lua・Rubyの実装状況

- ✓ C++バインディング
- ✓ 計算対応（Gandivaも対応）
- ✓ Plasma対応・Flight未対応
- ✓ Ruby関連は畑中さん・橋立さんの話で！

MATLABの実装状況

- ✓ C++バインディング
- ✓ Featherの読み込みのみ対応

Pythonの実装状況

- ✓ C++バインディング
- ✓ pandas・NumPy相互変換対応
- ✓ 計算対応（Gandivaも対応）
- ✓ Plasma対応・Flight未対応
- ✓ Python関連は堀越さんの話で！

Rの実装状況

- ✓ C++バインディング
- ✓ 未対応の型：小数・共用体
- ✓ 計算対応（Gandivaは未対応）
- ✓ フォーマット変換はFeatherのみ対応
- ✓ Plasma・Flight未対応
- ✓ R関連は湯谷さんの話で！

対応言語まとめ

- ✓ 基本的な型はすべての言語で対応済み
  - ✓ すでに**データ交換用途に使える**
- ✓ データの高速な計算は一部言語で対応
- ✓ フォーマット変換も各言語の対応は様々

Apache Arrowの今後案

Thoughts about 2019 Arrow development focus areas

- ✓ クラウドストレージ対応
- ✓ データセット対応
- ✓ プッシュダウン対応
- ✓ 演算グラフ対応強化
- ✓ 計算結果のキャッシュ用途対応

クラウドストレージ対応

- クラウドストレージ
  - Amazon S3、Google Cloud Storageとか
  - HDFS（Hadoop File System）は対応済み
- ストレージ上のデータの読み書きを簡単に

データセット対応

- ✓ データセット
  - ✓ 複数ファイルに分かれたデータ群
- ✓ データセットの読み書きを簡単に

データセット例

```
a_1/0.parquet # a=1のデータのみ
a_1/1.parquet # a=1のデータのみ
a_2/2.parquet # a=2のデータのみ
a_3/3.parquet # a=3のデータのみ
a_3/4.parquet # a=3のデータのみ
```

プッシュダウン対応

- 下位レイヤーに情報を渡して
  不要な処理をごっそりすっ飛ばす
  - 例：必要な部分のみファイルから読む
- データベースはよくやっている
  - MySQL/PostgreSQLもやっている

プッシュダウン例

```
WHERE a = 1のとき
a_1/0.parquet # 読む
a_1/1.parquet # 読む
a_2/2.parquet # 読まない
a_3/3.parquet # 読まない
a_3/4.parquet # 読まない
```

演算グラフ対応強化

- 演算グラフ
  - SQLのSELECTでできるような処理をグラフで表現
  - Gandivaがすでに作っている
- Ibisやdplyrぐらいリッチにする
- 演算グラフのシリアライズ対応
- 実行をもっと速く

まとめ

- ✓ Apache Arrowが言語を超えて実現すること
  - ✓ データ交換・データ処理の高速化
  - ✓ 実装の共有（共同開発・ライブラリー化）
- ✓ Apache Arrowを使いたくなった人は
  Apache Arrowの開発にも参加しよう！