Apache ArrowのRubyバインディングをGObject Introspectionで

須藤功平

株式会社クリアコード

名古屋Ruby会議03

2017-02-11

内容

1. まくら
2. 本題
3. オチ

やりたいこと

Rubyで
データ分析

データ分析？

いろいろある

やりたいデータ分析

全文検索関連

- ✓ 同義語・関連語の自動抽出
  例：整数とIntegerは同じ
- ✓ 最近アツいキーワードの抽出
- ✓ ...

課題

- 道具が足りない
  - ライブラリーがない・
    メンテナンスされていない
- 道具が使いにくい

解決方法

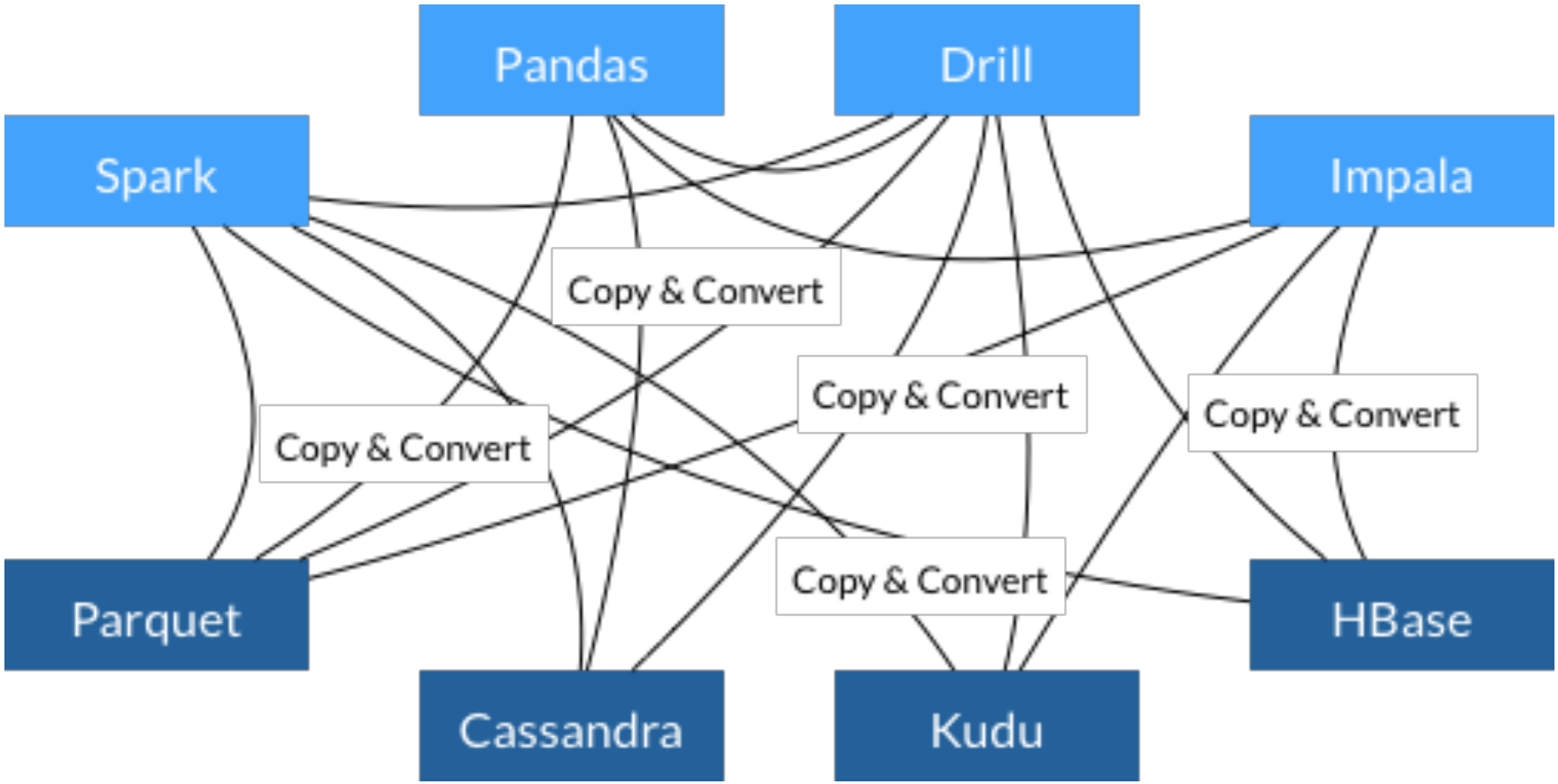
やる

（道具を整備する）

世間の様子

- ✓ 主にJavaとPythonを使う
  - ✓ 道具が揃っている
- ✓ 組み合わせてデータ分析

組み合わせた様子



https://arrow.apache.org/より（Apache License 2.0）

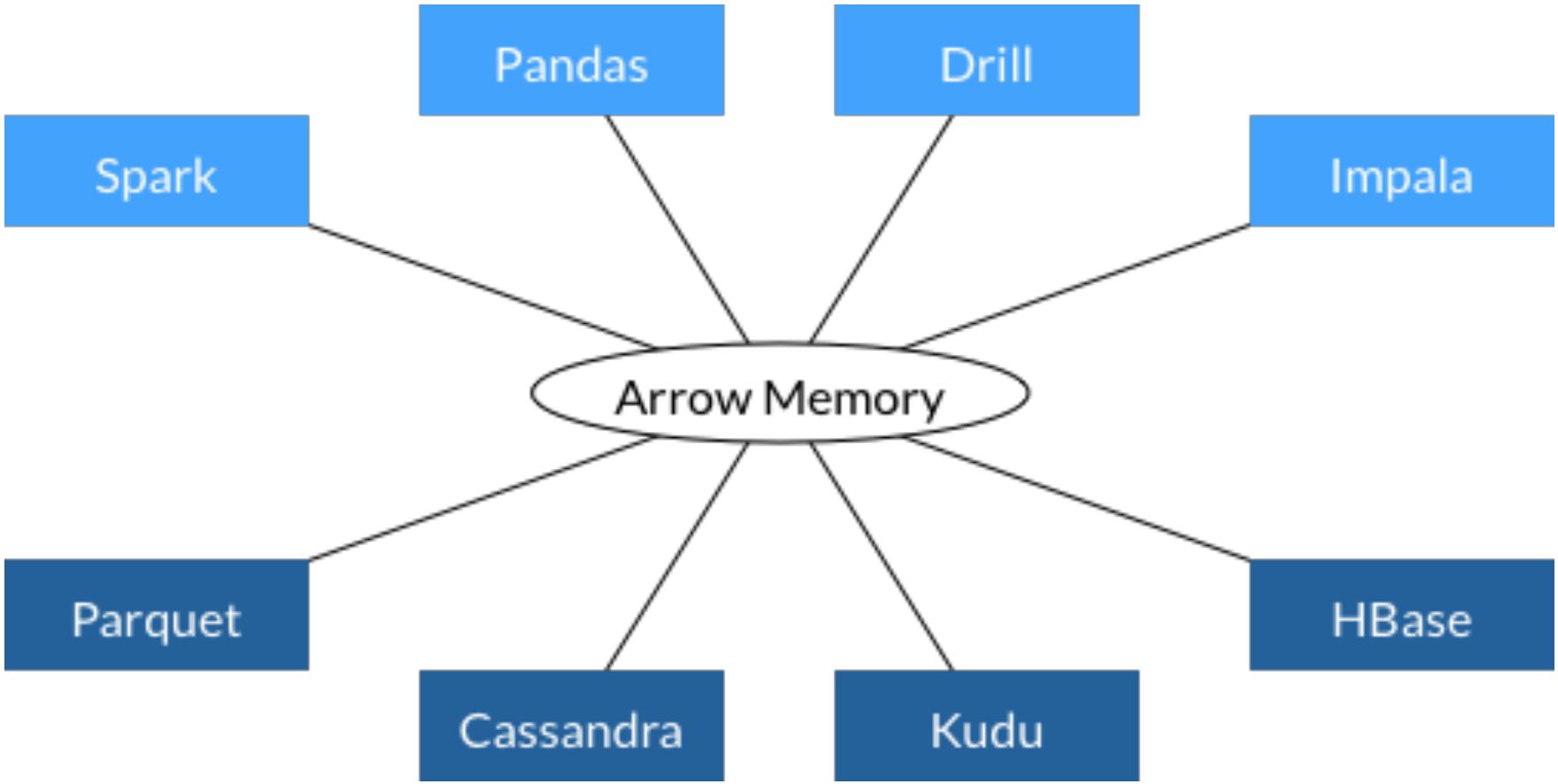
組み合わせの課題

- データ交換コストが高い
  - データの直列化でCPUの8割を使用
    by https://arrow.apache.org/

解決方法

データの
フォーマットを
統一しよう！

統一した様子

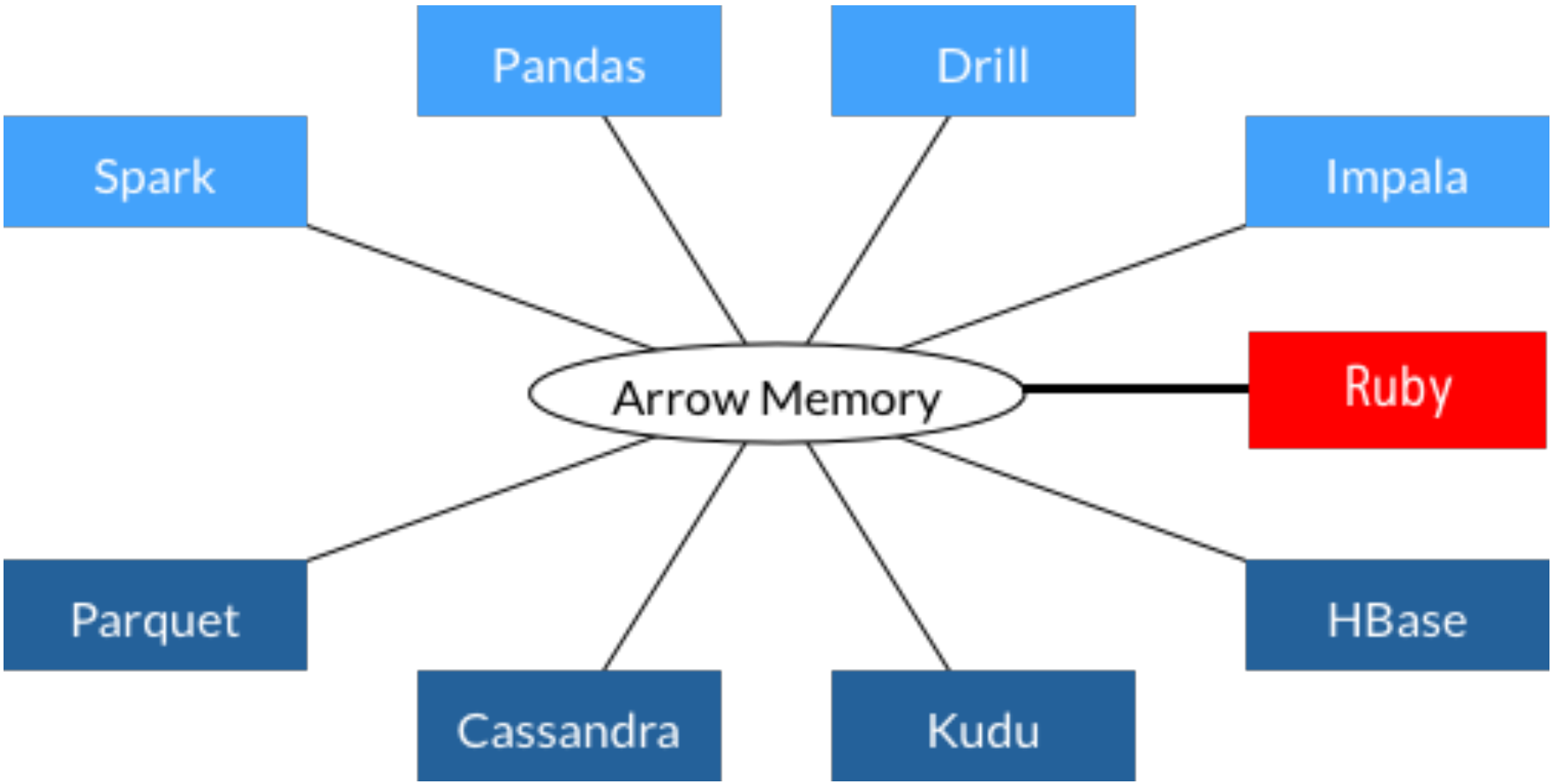


https://arrow.apache.org/より（Apache License 2.0）

これはチャンス！

- RubyがArrow対応すると…
  - 既存のシステムとやりとりできる
  - 「この部分だけはRubyを使う」をできる

一部でRuby

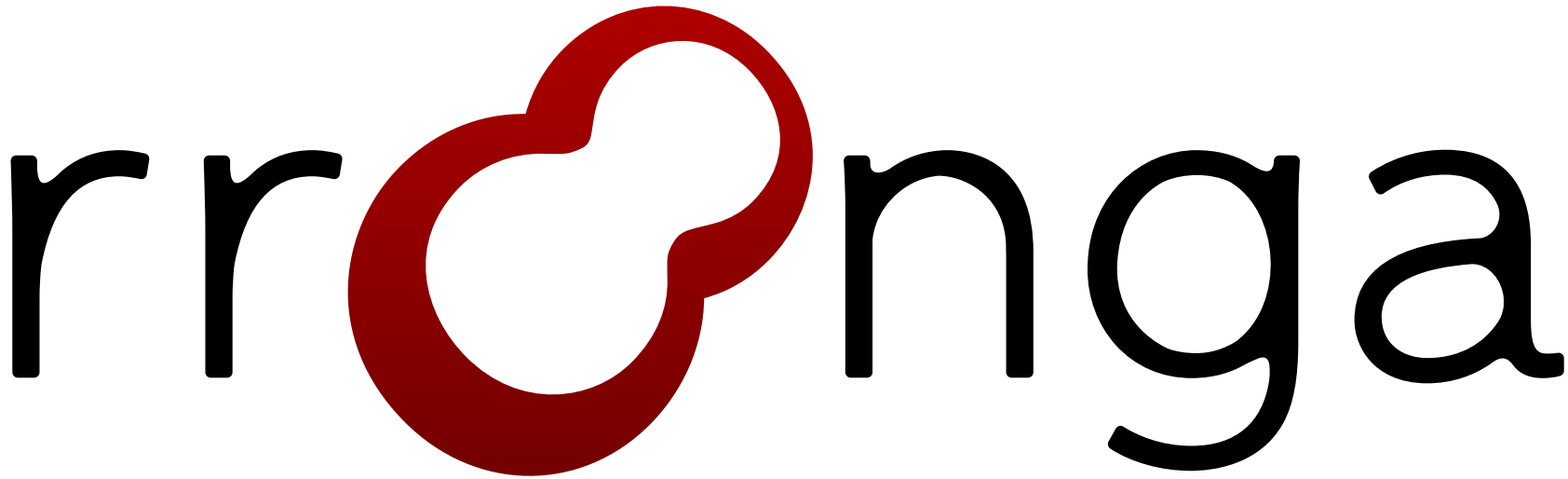


前述の図を変更（Apache License 2.0）

一部でRubyを…使える？

道具が
ないじゃん！

道具



Groongプロジェクト（CC BY 3.0）

Rroonga

- 全文検索エンジン**ライブラリー**
  - SQLで操作とかじゃなく
    オブジェクトとして触れる

オブジェクト例

転置索引
オブジェクト

転置索引

雑な説明

- Hashみたいなもの
- キー：
  - トークン（単語みたいなもの）
- 値：
  - キー（トークン）を含む文書の一覧

転置索引オブジェクト（1）

```
index = Groonga["Words.index"]
token = "オブジェクト"
p index.estimate_size(token)
# => 19048
# 「オブジェクト」を含む文書は
# 約19048個ありそう
```

転置索引オブジェクト（2）

```
index.open_cursor(token) do |cursor|
  # 最初の出現情報
  posting = cursor.next
  # 「オブジェクト」を含む最初の文書のID
  p posting.record.id # => 17
  # この文書が何個「オブジェクト」を含むか
  p posting.term_frequency # => 1
  # 文書の内容
  puts posting.record.document
end
```

分析に使ってみよう

るりまを分析

↓

関連語を抽出

関連語の抽出方法例

1. 文書を前処理
2. 全文書をトピックに分類

   ✓どんなトピックがあるかは学習

3. 同じトピックで使われやすい単語を抽出
   →関連語！

文書の前処理

単語の出現頻度
(Bag of Wordsという)
に変換

単語の出現頻度

```
"名古屋マジ名古屋"
#  ↓
{
   "名古屋" => 2, # 2回
   "マジ"   => 1, # 1回
}
```

単語は数値に変換

```
# "名古屋" => 10
# "マジ"   => 20
{
  10 => 2, # 「名古屋」2回
  20 => 1, # 「マジ」1回
}
```

なんで数値にするの？

計算しやすい

Rroongaで前処理できる？

- ✓ 出現回数は求められる？
  - ✓ 👍：posting.term_frequency
- ✓ 単語を数値に変換できる？
  - ✓ 👍：全文検索エンジンの必須機能

Rroongaで前処理（1）

```
bow = {} # 「名古屋」の分だけ
index.open_cursor("名古屋") do |cursor|
  cursor.each do |posting|
    record_id      = posting.record_id
    term_id        = posting.term_id # "名古屋"のID
    term_frequency = posting.term_frequency # 出現回数
    bow[record_id] ||= {}
    bow[record_id][term_id] = term_frequency
    # bow: { # ↓実際は文字列じゃなくてID
    #   2 => {"名古屋" => 9}  # 文書2では9回出現
    #   5 => {"名古屋" => 19} # 文書5では19回出現
    # }
  end
end
```

Rroongaで前処理（2）

```
bow = {}
index.table.each do |token| # 全トークンを処理
  index.open_cursor(token) do |cursor|
    cursor.each do |posting|
      # ...同じ...
    end
  end
end
# bow: { # 完成！
#   2 => {"名古屋" => 9,  "マジ" => 2}
#   5 => {"名古屋" => 19, "寄席" => 1},
#   ...
# }
```

前処理終わり

- ✓ 次はトピックに分類
  - ✓ どんなトピックがあるかは学習

トピックの学習方法

LDA

（Latent Dirichlet Allocation/潜在的ディリクレ配分法）

他にも色々ある
参考：http://www.ism.ac.jp/~daichi/lectures/H24-TopicModel/ISM-2012-TopicModels-daichi.pdf

RubyでLDA

lda-ruby gem

RubyでLDAをしない！

なぜなら！
Apache Arrowとか
GObject Introspectionの
話をする
機会がなくなるからだ！

PythonでLDA

scikit-learn

（gensimの方がよく使われているかも）

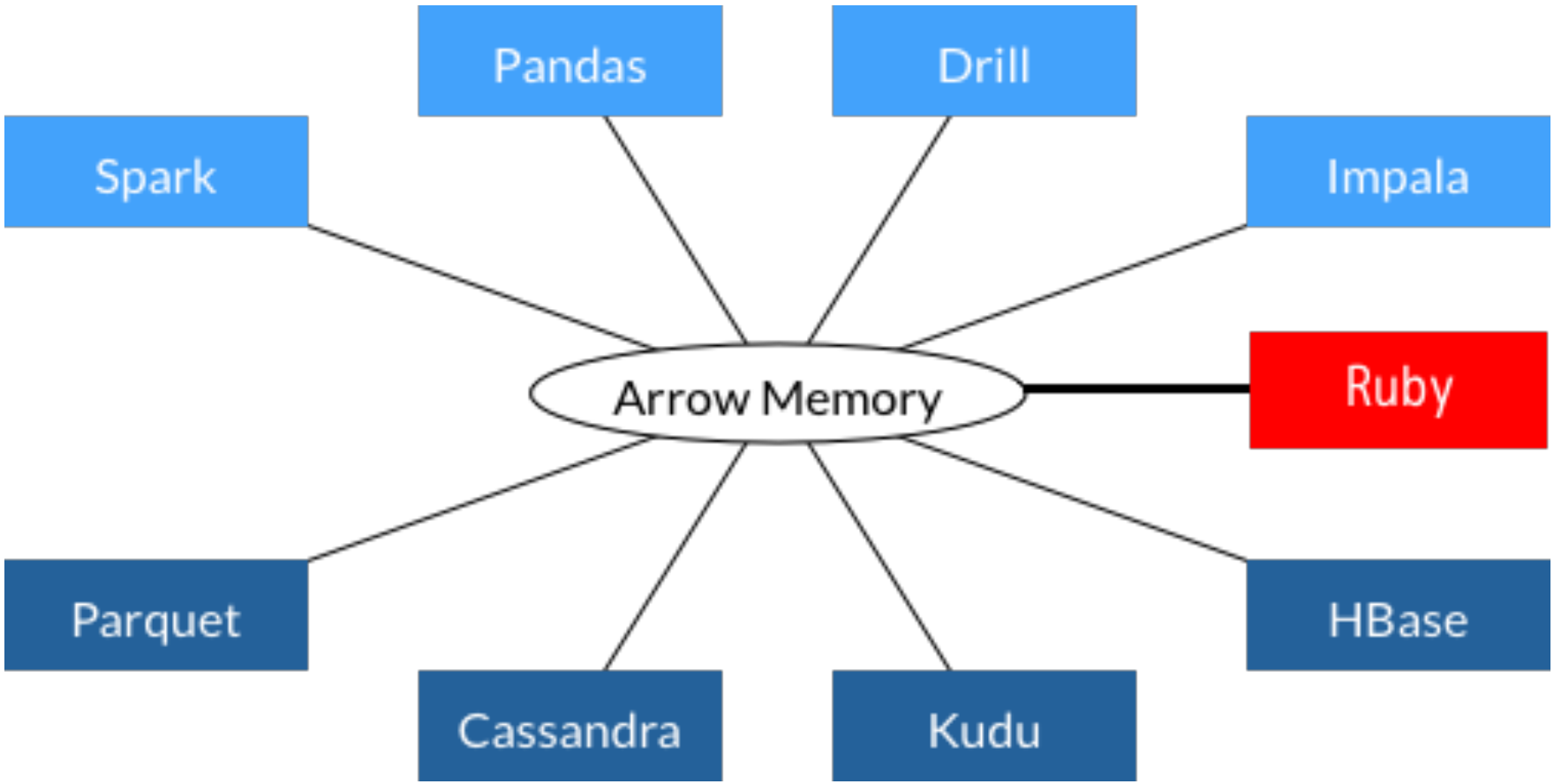
scikit-learnでLDA

```
import sklearn.decomposition
LDA = sklearn.decomposition.LatentDirichletAllocation
model = LDA(n_topics=100, # 100トピックに分類
            learning_method="online",
            total_samples=len(bag_of_words)) # 文書数
for words in bag_of_words: # 前処理結果
  model.partial_fit(words) # 要フォーマット変換
model.components_ # ここに学習したトピックがある
```

前処理結果がない！

- 前処理はRubyでやった
  - Python側にはない
  - 前処理結果がないと
    scikit-learnで分析できない！
- どうしよう！
  - そうだ！Apache Arrowだ！

Arrowでつながる世界



(Apache License 2.0)

RubyでArrow

- ArrowはC++のライブラリー
  - Rubyからは使えない
- どうしよう！
  - やる（バインディングを作る）

Arrowのバインディング

- arrow-glib
  - github.com/kou/arrow-glib
  - Arrowのラッパー・C APIを提供
- GObject Introspection対応
  - バインディングを実行時に生成

使い方：準備

```
require "gi"
Arrow = GI.load("Arrow")
```

使い方：配列を作る

```ruby
# Arrowは高速に1次元・2次元配列を
# 扱うAPIを提供するライブラリー
builder = Arrow::UInt32ArrayBuilder.new
builder.append(29) # 要素追加
builder.append(9)  # 要素追加
term_ids = builder.finish # 配列作成
p term_ids.length         # => 2
p term_ids.get_value(0) # => 29
p term_ids.get_value(1) # => 9
```

少し…使いにくい！

- GObject Introspection
  - だいたいいい感じになる！（すごい！）
  - が！Ruby特有のところは一手間必要

使いやすさ検討

```
builder = Arrow::UInt32ArrayBuilder.new
builder.append(29)
builder.append(9)
term_ids = builder.finish
# ↓
term_ids =
  Arrow::UInt32ArrayBuilder.build([29, 9])
```

一手間

```
class Arrow::ArrayBuilder
  class << self
    def build(values)
      builder = new
      values.each do |value|
        builder.append(value)
      end
      builder.finish
    end
  end
end
```

もう一手間

```
class Arrow::UInt32Array
  class << self
    def new(values)
      UInt32ArrayBuilder.build(values)
    end
  end
end
```

一手間後

```
builder = Arrow::UInt32ArrayBuilder.new
builder.append(29)
builder.append(9)
term_ids = builder.finish
# ↓
term_ids = Arrow::UInt32Array.new([29, 9])
```

さらに使いやすさ検討

```
p term_ids.get_value(0) # => 29
p term_ids.get_value(1) # => 9
# ↓
p term_ids[0] # => 29
p term_ids[1] # => 9
```

二手間

```
class Arrow::Array
  def [](i)
    get_value(i)
  end
end
```

もう一手間

```
class Arrow::Array
  include Enumerable
  def each
    length.times do |i|
      yield(self[i])
    end
  end
end
```

二手間後

```
p term_ids.get_value(0) # => 29
p term_ids.get_value(1) # => 9
# ↓
p term_ids.to_a # => [29, 9]
```

一手間はどこに書くの？

```
# こう？
require "gi"
Arrow = GI.load("Arrow")
module Arrow
  class Array
    # ...
  end
end
```

違う！

- GI.loadはデモ用のAPI
  - ちゃんと作るときは使わない
- GI::Loaderを継承
  - #post_loadフック時に一手間

GI::Loader#post_load

```
class Arrow::Loader < GI::Loader
  private
  def post_load(*args)
    require "arrow/array"
    require "arrow/array-builder"
  end
end
```

arrow/array.rb

```
class Arrow::Array
  include Enumerable
  def each
    length.times do |i|
      yield(self[i])
    end
  end
end
```

arrow.rb

```
require "arrow/loader"
module Arrow
  Loader.load("Arrow", self)
  # ↑の中で#post_loadが呼ばれる
end
```

使い方

```
require "arrow"

term_ids = Arrow::UInt32Array.new([29, 9])
p term_ids.to_a # => [29, 9]
```

すごい！

Rubyっぽい！

実装

- RArrow
  - github.com/kou/rarrow
  - RのArrowバインディングみたいでアレかもしれない
- 他の一手間例もアリ
  - 例：.open {|io| ...}で自動close

前処理結果を渡す

- 忘れていたでしょ？
  - Arrowの話をしていたのはこのため
- 手順
  a. Rubyで書き出し
  b. Pythonで読み込み

Rubyで書き出し

```
FOS = Arrow::IO::FileOutputStream # 長いから
SW = Arrow::IPC::StreamWriter     # 長いから
FOS.open("/tmp/bow", false) do |output_stream|
  # schema：カラム名と型の定義（省略）
  SW.open(output_stream, schema) do |writer|
    bow.each do |record_id, words|
      # record_batch（省略）：
      # テーブルからN行だけ抜き出したもの
      writer.write_record_batch(record_batch)
    end
  end
end
```

Pythonで読み出し

```
from scipy.sparse import csr_matrix
import pandas as pd
import pyarrow as A
with A.io.MemoryMappedFile("/tmp/bow", "rb") as source:
    reader = A.ipc.StreamReader(source)
    for record_batch in reader: # ストリームで順に処理
        # Pandasのデータフレームに変換（ゼロコピー！）
        df = record_batch.to_pandas()
        # 疎な行列に変換
        corpus = csr_matrix((df["score"].values,
                             df["term_id"].values,
                             [0, df["term_id"].size]),
                            shape=(1, n_features))
        model.partial_fit(corpus) # 学習
```

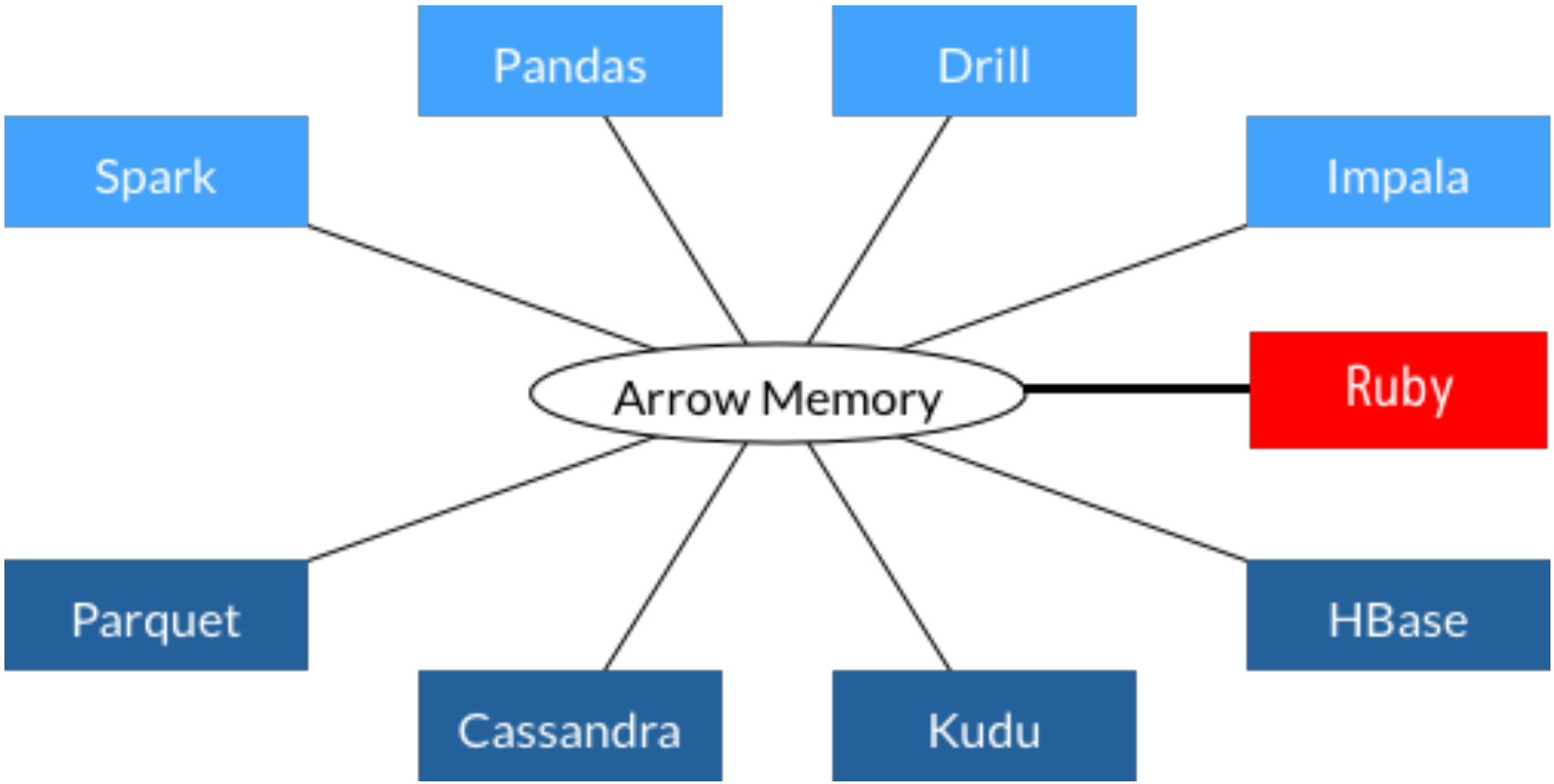
トピックを確認

```
for topic in model.components_:
    n_top_terms = 10
    # このトピックに関連している
    # 上位10単語を計算
    top_term_indexes = # ↓[::-1]でreverse
        topic.argsort()[:-n_top_terms-1:-1]
    for i in top_term_indexes:
        term_id = i       # 単語ID
        score = topic[i] # 関連度
        print("%s:%s" % (term_id, score))
```

単語IDじゃわからん！

- ✓ 単語はIDにして計算した
  - ✓ けど、確認は単語でしたい！
- ✓ 単語とIDの管理は…Rroonga！
  - ✓ トピックをRubyに持っていかないと
  - ✓ そうだ！Apache Arrowだ！

Arrowでつながる世界



(Apache License 2.0)

Pythonで書き出し

```
with open("/tmp/topics", "wb") as sink:
    # schema：batch.schemaで取得できる（省略）
    writer = A.ipc.StreamWriter(sink, schema)
    def topic_to_df(topic):
        # 前述のトピックを確認した処理と同じ
        values = [[i, topic[i]]
                  for i in topic.argsort()[:-10-1:-1]]
        return pd.DataFrame(values,
                            columns=["term_id", "score"])
    for topic in model.components_:
        df = topic_to_df(topic)
        batch = A.RecordBatch.from_pandas(df)
        writer.write_batch(batch)
    writer.close()
```

Rubyで読み込み

```
MMF = Arrow::IO::MemoryMappedFile # 長いから
SR = Arrow::IPC::StreamReader     # 長いから
MMF.open("/tmp/topics", :read) do |input|
  SR.open(input) do |reader|
    reader.each do |record_batch|
      record_batch.each do |record|
        # 単語IDを単語に変換
        term = index.table[record["term_id"]]
        p [term, record["score"]]
      end
    end
  end
end
```

実際の結果

```
1: uzqpuaglzig, あきらめ, ご覧
2: useloopback, タスク
3: プラットホーム, mydog
4: delegateclass, barbar
...
```

微妙…

大事なこと

Garbage in,
Garbage out

前処理をがんばる

- ✓ いらない文書を無視
- ✓ いらないトークンを無視

いらない文書を無視

```
entries = Groonga["Entries"]
# 全文検索エンジンで検索するので高速！すごい！
target_entries = entries.select do |record|
  (record.version == "2.4.0") -
    (record.document =~ "@todo")
    # ↑@todoな文書を対象外
end
# ... do |posting|
  # 存在チェックも全文検索エンジンがやるので高速！
  next if target_entries.key?(posting.record_id)
# ... end
```

いらないトークンを無視

```
n_entries = target_entries.size
# ほとんどの文書に含まれるなら重要じゃない
too_many_threshold = n_entries * 0.9
# ごく一部の文書にしか含まれないなら重要じゃない
too_less_threshold = n_entries * 0.01
# ... do |term|
  n_match_documents = index.estimate_size(term)
  next if n_match_documents >= too_much_threshold
  next if n_match_documents <= too_less_threshold
# ... end
```

実際の結果

```
1: self, ブロック
2: each, enumerator
3: integer, 整数, 表示
4: ruby, object
...
```

それっぽくなってきた！

まとめ（1）

- ✓ Rubyでデータ分析
  - ✓ 全部はムリ
  - ✓ でも一部ならいける

まとめ（2）

- 一部ならいけるのは…
  - Apache Arrowの登場
    データ交換しやすくなる
  - Rroongaの存在
    Rubyで高速に自然言語処理できる

まとめ（3）

- ✓ RubyでApache Arrow
  - ✓ バインディングが必要
  - ✓ GObject Introspection ベースで作った（arrow-glib）

まとめ（4）

- ✓ GObject Introspection
  - ✓ だいたいいい感じ
  - ✓ さらに一手間でグッとよくなる

まとめ（5）

- ✓ 実際に分析してみた
  - ✓ Rubyで前処理
    ↓Arrow
  - ✓ Pythonで分析
    ↓Arrow
  - ✓ Rubyで確認
    （本当は全文検索用DBに入れて活用する）

まとめ（6）

- ✓ Rubyでデータ分析いけそう！
  - ✓ Arrowでデータの受け渡しが容易に
    →分析処理へのRubyの参加が容易に
  - ✓ RroongaでRubyでも高速に
    自然言語処理をできる

おしらせ（1）

- ✓ Rubyでデータ分析したい人は クリアコードに相談してね！
  - ✓ 道具の用意・活用を手伝える

おしらせ（2）

- Groonga Meatup名古屋2017
  - 時間：明日の午前（10:00-）
  - 場所：Misoca

別案

- RroongaのPython版を作る
  - RubyもArrowも必要なくなる…
  - けど、より高速に前処理できる！
  - Cythonで作ろうかと思っている…