



ТЕХНОСФЕРА

Методы распределенной обработки больших объемов данных в Hadoop

Лекция 5: MapReduce в Hadoop, алгоритмы

Алгоритмы и паттерны MapReduce



WordCount

- **Описание проблемы**
 - Есть коллекция документов
 - Каждый документ – это набор термов (слов)
 - Необходимо подсчитать кол-во вхождений каждого терма во всех документах
- **Дополнительно**
 - Функция может быть произвольной
 - Напр., файл лога содержит время ответа. Необходимо подсчитать среднее время.

WordCount, baseline

```
class Mapper
```

```
  method Map (docid id, doc d)
```

```
    for all term t in doc d do
```

```
      Emit(term t, count 1)
```

```
class Reducer
```

```
  method Reduce (term t, counts [c1, c2,...])
```

```
    sum = 0
```

```
    for all count c in [c1, c2,...] do
```

```
      sum = sum + c
```

```
    Emit(term t, count sum)
```

WordCount, “In-mapper combining”, v.1

- Минусы baseline
 - Много лишних счетчиков от *Mapper*
 - Агрегируем их для каждого документа

```
class Mapper  
  method Map (docid id, doc d)  
    H = new AssociativeArray  
    for all term t in doc d do  
      H{t} = H{t} + 1  
    for all term t in H do  
      Emit(term t, count H{t})
```

WordCount, Combiner

- Для всех документов *Mapper* используем *Combiner*

```
class Mapper
  method Map (docid id, doc d)
    for all term t in doc d do
      Emit(term t, count 1)

class Combiner
  method Combine (term t, [c1, c2,...])
    sum = 0
    for all count c in [c1, c2,...] do
      sum = sum + c
    Emit(term t, count sum)

class Reducer
  method Reduce (term t, counts [c1, c2,...])
    sum = 0
    for all count c in [c1, c2,...] do
      sum = sum + c
    Emit(term t, count sum)
```

WordCount, “In-mapper combining”, v.2

```
class Mapper  
  method Initialize  
     $H = \text{new } \textbf{AssociativeArray}$   
  
  method Map (docid id, doc d)  
    for all term t in doc d do  
       $H\{t\} = H\{t\} + 1$   
  
  method Close  
    for all term t in H do  
      Emit(term t, count  $H\{t\}$ )
```

WordCount, “In-mapper combining”, v.2

- “In-mapper combining”
 - “Заворачиваем” функционал комбайнера в *mapper* путем сохранения состояния между вызовами функции *map()*
- Плюсы
 - Скорость
 - Почему это быстрее, чем стандартный *Combiner*?
- Минусы
 - Требуется “ручное” управление памятью
 - Потенциальная возможность для багов связанных с сортировкой порядка элементов

Среднее значение, v.1

```
class Mapper  
  method Map(string t, integer r)  
    Emit(string t, integer r)  
  
class Reducer  
  method Reduce(string t, integers [r1, r2, ...])  
    sum = 0  
    cnt = 0  
    for all integers r in [r1, r2, ...] do  
      sum = sum + r  
      cnt = cnt + 1  
    avg = sum / cnt  
    Emit(string t, integer avg)
```

Можно ли использовать Reducer в качестве Combiner?

Среднее значение, v.2

```
class Mapper
```

```
  method Map(string t, integer r)
```

```
    Emit(string t, integer r)
```

```
class Combiner
```

```
  method Combine(string t, integers [r1, r2, ...])
```

```
    sum = 0
```

```
    cnt = 0
```

```
    for all integers r in [r1, r2, ...] do
```

```
      sum = sum + r
```

```
      cnt = cnt + 1
```

```
    Emit(string t, pair(sum, cnt))
```

```
class Reducer
```

```
  method Reduce(string t, pairs[(s1,c1),(s2,c2) ...])
```

```
    sum = 0
```

```
    cnt = 0
```

```
    for all pairs p in [(s1,c1),(s2,c2) ...] do
```

```
      sum = sum + p.s
```

```
      cnt = cnt + p.c
```

```
    avg = sum / cnt
```

```
    Emit(string t, integer avg)
```

Почему это не работает?

Среднее значение, v.3

*class **Mapper***

*method **Map**(string t, integer r)
Emit(string t, pair (r,1))*

*class **Combiner***

*method **Combine**(string t pairs[(s1,c1),(s2,c2) ...]))
sum = 0
cnt = 0
for all pairs p in [(s1,c1),(s2,c2) ...] do
sum = sum + p.s
cnt = cnt + p.c
Emit(string t, pair(sum, cnt))*

*class **Reducer***

*method **Reduce**(string t, pairs[(s1,c1),(s2,c2) ...])
sum = 0
cnt = 0
for all pairs p in [(s1,c1),(s2,c2) ...] do
sum = sum + p.s
cnt = cnt + p.c
avg = sum / cnt
Emit(string t, pair (avg, cnt))*

А так будет работать?

Среднее значение, v.4

```
class Mapper  
  method Initialize  
     $S = \text{new } \textbf{AssociativeArray}$   
     $C = \text{new } \textbf{AssociativeArray}$   
  
  method Map (string  $t$ , integer  $r$ )  
     $S\{t\} = S\{t\} + r$   
     $C\{t\} = C\{t\} + 1$   
  
  method Close  
    for all term  $t$  in  $S$  do  
       $\text{Emit}(\text{term } t, \text{pair}(S\{t\}, C\{t\}))$ 
```

Distinct Values (Unique Items Counting)

- Описание проблемы
 - Есть множество записей
 - Каждая запись содержит поле F и производное число признаков категорий $G = \{G1, G2, \dots\}$.
- Задача
 - Подсчитать общее число уникальных значений поля F для каждого подмножества записей для каждого значения в каждой категории

```
Record 1: F=1, G={a, b}  
Record 2: F=2, G={a, d, e}  
Record 3: F=1, G={b}  
Record 4: F=3, G={a, b}
```

Result:

```
a -> 3 // F=1, F=2, F=3  
b -> 2 // F=1, F=3  
d -> 1 // F=2  
e -> 1 // F=2
```

Distinct Values, v.1

- Решение в две фазы (две задачи MapReduce)
- Первая фаза
 - *Mapper* пишет все уникальные пары $[G, F]$
 - *Reducer* подсчитывает общее кол-во вхождений такой пары
 - Основная цель этой фазы – гарантировать уникальность значений F
- Вторая фаза
 - Пары $[G, F]$ группируются по G и затем считается общее кол-во элементов в каждой группе

Distinct Values, v.1

```
// phase 1
class Mapper
  method Map(null, record [value f, categories [g1, g2,...]])
    for all category g in [g1, g2,...]
      Emit(record [g, f], count 1)

class Reducer
  method Reduce(record [g, f], counts [n1, n2, ...])
    Emit(record [g, f], null )
```

```
// phase 2
class Mapper
  method Map(record [f, g], null)
    Emit(value g, count 1)

class Reducer
  method Reduce(value g, counts [n1, n2,...])
    Emit(value g, sum( [n1, n2,...] ) )
```

Distinct Values, v.2

- Требуется только одна фаза MapReduce
 - *Mapper*
 - Пишет значение и категории
 - *Reducer*
 - Исключает дубликаты из списка категорий для каждого значения
 - Увеличивает счетчик для каждой категории
 - В конце *Reducer* пишет общее кол-во для каждой категории
- Первая фаза
 - Данный подход не очень хорошо масштабируется
 - Подходит для небольшого числа категорий
 - Напр. парсинг действий пользователей из web-логов
 - *Combiners* позволяют уменьшить кол-во дубликатов перед фазой *Reduce*

Distinct Values, v.2

*class **Mapper***

*method **Map**(null, record [value f, categories [g1, g2,...])*
for all category g in [g1, g2,...]
Emit(value f, category g)

*class **Reducer***

*method **Initialize***

H = new AssociativeArray : category -> count

*method **Reduce**(value f, categories [g1, g2,...])*
[g1', g2',...] = ExcludeDuplicates([g1, g2,...])
for all category g in [g1', g2',...]
H{g} = H{g} + 1

*method **Close***

for all category g in H do
Emit(category g, count H{g})

Cross-Correlation

- Описание проблемы
 - Есть множество кортежей объектов
 - Для каждой возможной пары объектов посчитать число кортежей, где они встречаются вместе
 - Если число объектов N , то $N*N$ объектов будет обработано
- Применение
 - Анализ текстов
 - Кортежи – предложения, объекты – слова
 - Маркетинг
 - Покупатели, кто покупает одни товары, обычно покупают и другие товары
- Если $N*N$ небольшое и можно построить матрицу в памяти, то реализация довольно проста

Cross-Correlation: Pairs

- Каждый *Mapper* принимает на вход кортеж
 - Генерит все пары соседних объектов
 - Для всех пар выполняет $emit(a, b) \rightarrow count$
- Reducer суммирует все count для всех пар
 - *Combiners*?

```
class Mapper
```

```
    method Map(null, items [i1, i2,...] )
```

```
        for all item i in [i1, i2,...]
```

```
            for all item j in [i1, i2,...]
```

```
                Emit(pair [i j], count 1)
```

```
class Reducer
```

```
    method Reduce(pair [i j], counts [c1, c2,...])
```

```
        s = sum([c1, c2,...])
```

```
        Emit(pair[i j], count s)
```

Cross-Correlation: Pairs

- Плюсы
 - Нет затрат по памяти
 - Простая реализация
- Минусы
 - Множество пар надо отсортировать и распределить по редьюсерам (sort & shuffle)
 - *Combiner* вряд ли поможет (почему?)

Cross-Correlation: Stripes

- Основная идея:
 - Группировать пары вместе в ассоциативный массив
 - Каждый *Mapper* принимает на вход последовательность
 - Генерит все пары рядом расположенных объектов
 - Для каждого объекта выполняет $emit\ a \rightarrow \{ b: count_b, c: count_c, d: count_d \dots \}$
 - Reducer*'ы выполняют поэлементное суммирование ассоциативных массивов

 $(a, b) \rightarrow 1$ $(a, c) \rightarrow 2$ $(a, d) \rightarrow 5$ $(a, e) \rightarrow 3$ $(a, f) \rightarrow 2$

$$\begin{array}{r} a \rightarrow \{ b: 1, \quad d: 5, e: 3 \} \\ + \quad a \rightarrow \{ b: 1, c: 2, d: 2, \quad f: 2 \} \\ \hline a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \} \end{array}$$

Cross-Correlation: Stripes

*class **Mapper***

*method **Map**(null, items [i1, i2,...])*

for all item i in [i1, i2,...]

H = new AssociativeArray : item -> counter

for all item j in [i1, i2,...]

H{j} = H{j} + 1

Emit(item i, stripe H)

*class **Reducer***

*method **Reduce**(item i, stripes [H1, H2,...])*

H = new AssociativeArray : item -> counter

H = merge-sum([H1, H2,...])

for all item j in H.keys()

Emit(pair [i j], H{j})

Cross-Correlation: Stripes

- Плюсы
 - Намного меньше операций сортировки и shuffle
 - Возможно, более эффективное использование *Combiner*
- Минусы
 - Более сложная реализация
 - Более “тяжелые” объекты для передаче данных
 - Ограничения на размеры используемой памяти для ассоциативных массивов
- Pairs vs Stripes
 - Обычно, подход со *stripes* быстрее, чем с *pairs*

Реляционные паттерны MapReduce



Selection

```
class Mapper  
  method Map(rowkey key, value t)  
    if t satisfies the predicate  
      Emit(value t, null)
```

Projection

```
class Mapper
```

```
method Map(rowkey key, value t)
```

```
value g = project(t) // выбрать необходимые поля в g
```

```
Emit(tuple g, null)
```

```
// используем Reducer для устранения дубликатов
```

```
class Reducer
```

```
method Reduce(value t, array n) // n - массив из nulls
```

```
Emit(value t, null)
```

Union

// на вход подаются элементы из двух множеств A и B

*class **Mapper***

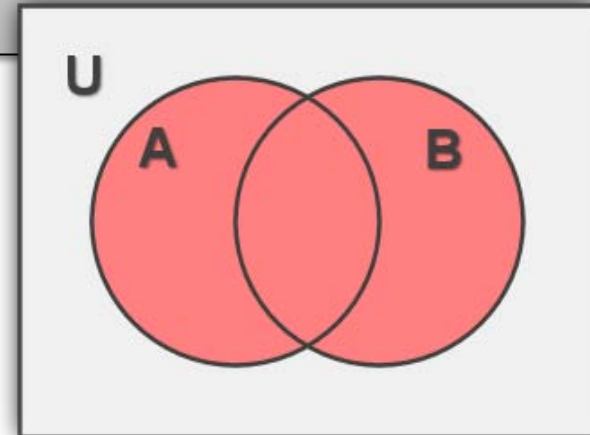
*method **Map**(rowkey key, value t)*

Emit(value t, null)

*class **Reducer***

*method **Reduce**(value t, array n) // n - массив из nulls*

Emit(value t, null)



Intersection

// на вход подаются элементы из двух множеств A и B

*class **Mapper***

*method **Map**(rowkey key, value t)*

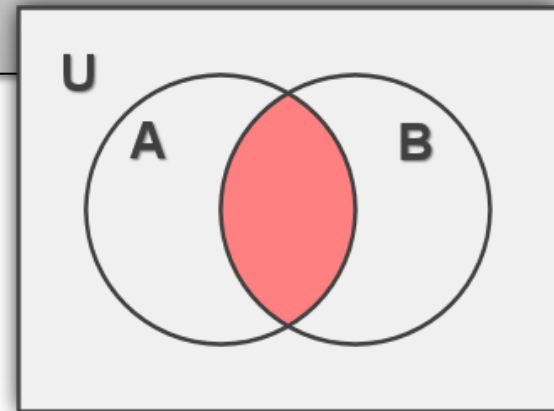
Emit(value t, null)

*class **Reducer***

*method **Reduce**(value t, array n) // n - массив из nulls*

if n.size() = 2

Emit(value t, null)



Difference

// на вход подаются элементы из двух множеств A и B

*class **Mapper***

*method **Map**(rowkey key, value t)*

Emit(value t, string t.SetName) // t.SetName либо 'A' либо 'B'

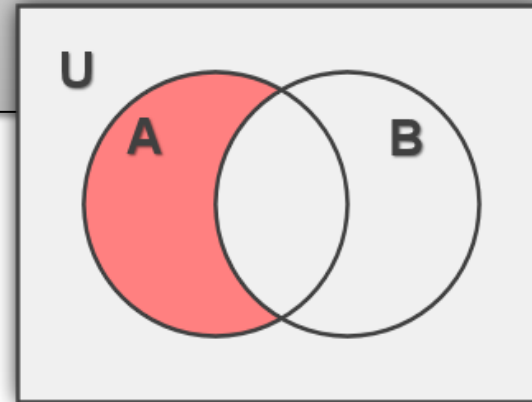
*class **Reducer***

// массив n может быть ['A'], ['B'], ['A' 'B'] или ['A', 'B']

*method **Reduce**(value t, array n)*

if n.size() = 1 and n[1] = 'A'

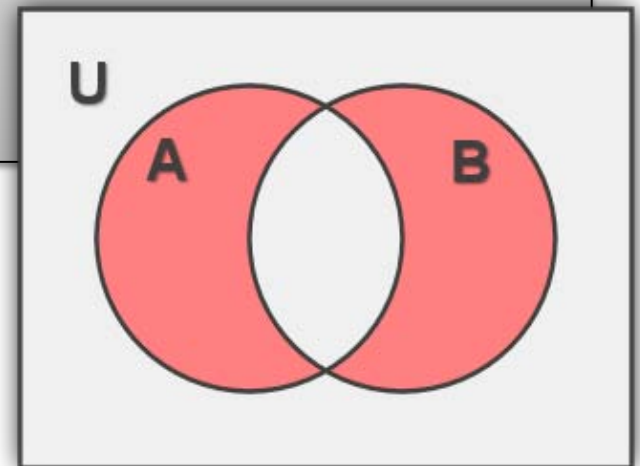
Emit(value t, null)



Symmetric Difference

```
// на вход подаются элементы из двух множеств A и B
class Mapper
    method Map(rowkey key, value t)
        Emit(value t, string t.SetName) // t.SetName либо 'A' либо 'B'

class Reducer
    // массив n может быть ['A'], ['B'], ['A' 'B'] или ['A', 'B']
    method Reduce(value t, array n)
        if n.size() = 1 and (n[1] = 'A' or n[1] = 'B')
            Emit(value t, null)
```



GroupBy и Aggregation

*class **Mapper***

*method **Map**(null, tuple [value GroupBy, value AggregateBy, value ...])*

Emit(value GroupBy, value AggregateBy)

*class **Reducer***

*method **Reduce**(value GroupBy, [v1, v2,...])*

// aggregate() : sum(), max(),...

Emit(value GroupBy, aggregate([v1, v2,...]))

Repartition Join

- *Reduce Join, Sort-Merge Join*
- Описание задачи
 - Объединить два множества A и B по ключу k
- Решение
 - *Mapper* проходит по всем значениям каждого множества
 - Выбирает ключ k и маркирует каждое значение тегом, определяющим множество, откуда пришло значение
 - *Reducer* получает все значения, объединенные по одному ключу и размещает их по двум корзинам, соответствующим каждому множеству
 - После этого проходит по обеим корзинам и генерирует значения из двух множеств с общим ключом

Repartition Join

```
class Mapper
```

```
method Map(null, tuple [join_key k, value v1, value v2,...])
```

```
Emit(join_key k, tagged_tuple [set_name tag, values [v1, v2, ...] ] )
```

```
class Reducer
```

```
method Reduce(join_key k, tagged_tuples [t1, t2,...])
```

```
H = new AssociativeArray : set_name -> values
```

```
for all tagged_tuple t in [t1, t2,...] // separate values into 2 arrays
```

```
H{t.tag}.add(t.values)
```

```
for all values a in H{'A'} // produce a cross-join of the two arrays
```

```
for all values b in H{'B'}
```

```
Emit(null, [k a b] )
```

Repartition Join

- Минусы
 - *Mapper* отправляет в output все данные, даже для тех ключей, которые есть только в одном множестве
 - *Reducer* должен хранить все значения для одного ключа в памяти
 - *Нужно самостоятельно управлять памятью в случае, если данные в нее не помещаются*

Replicated Join

- *Map Join, Hash Join*
- Часто требуется объединять два множества разных размеров – маленькое и большое
 - Напр. список пользователей с логом их активности
- Для этого можно использовать хеш-таблицу, куда загружать все элементы маленького множества, сгруппированных по ключу k
- Затем, идти по элементам большого множества в *Mapper* и выполнять lookip-запрос к этой хеш-таблице

Replicated Join

```
class Mapper  
  method Initialize  
     $H = \text{new AssociativeArray} : \text{join\_key} \rightarrow \text{tuple from } A$   
     $A = \text{load}()$   
    for all  $[\text{join\_key } k, \text{tuple } [a1, a2, \dots]]$  in  $A$   
       $H\{k\} = H\{k\}.\text{append}([a1, a2, \dots])$   
  
  method Map( $\text{join\_key } k, \text{tuple } B$ )  
    for all  $\text{tuple } a$  in  $H\{k\}$   
       $\text{Emit}(\text{null}, \text{tuple } [k \text{ } a \text{ } B])$ 
```

TF-IDF на MapReduce



TF-IDF

- Term Frequency – Inverse Document Frequency
 - Используется при работе с текстом
 - В Information Retrieval

TF-IDF

- **TF** (*term frequency* — частота слова) — отношение числа вхождения некоторого слова к общему количеству слов документа.
 - Таким образом, оценивается важность слова в пределах отдельного документа.

$$\text{tf}(t, d) = \frac{n_i}{\sum_k n_k}$$

где n_i есть число вхождений слова в документ, а в знаменателе — общее число слов в данном документе.

- **IDF** (*inverse document frequency* — обратная частота документа) — инверсия частоты, с которой некоторое слово встречается в документах коллекции.

$$\text{idf}(t, D) = \log \frac{|D|}{|(d_i \supset t_i)|}$$

Где:

- $|D|$ — количество документов в корпусе;
- $|(d_i \supset t_i)|$ — кол-во документов, в которых встречается t_i (когда $n_i \neq 0$).

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D)$$

TF-IDF

- Что нужно будет вычислить
 - Сколько раз слово T встречается в данном документе (tf)
 - Сколько документов, в котором встречается данное слово T (n)
 - Общее число документов (N)

TF-IDF

- **Job 1:** Частота слова в документе
- *Mapper*
 - Input: (*docname*, *contents*)
 - Для каждого слова в документе надо сгенерить пару (*word*, *docname*)
 - Output: ((*word*, *docname*), 1)
- *Reducer*
 - Суммирует число слов в документе
 - Outputs: ((*word*, *docname*), *tf*)
- *Combiner* такой же как и *Reducer*

TF-IDF

- **Job 2:** Кол-во документов для слова
- *Mapper*
 - Input: $((word, docname), tf)$
 - Output: $(word, (docname, tf, 1))$
- *Reducer*
 - Суммирует единицы чтобы посчитать n
 - Output: $((word, docname), (tf, n))$

TF-IDF

- **Job 3:** Расчет TF-IDF
- *Mapper*
 - Input: $((word, docname), (tf, n))$
 - Подразумевается, что N известно (его легко подсчитать)
 - Output: $((word, docname), (TF * IDF))$
- *Reducer*
 - Не требуется

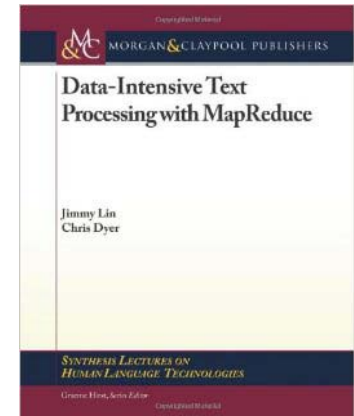
TF-IDF, масштабируемость

- Несколько MapReduce задач позволяют реализовать сложные алгоритмы и улучшить масштабируемость
 - Думая в стиле MapReduce часто означает разделение комплексных задач на более мелкие
- Стоит следить за тем, сколько используется ОЗУ, при работе с большим объемом данных
 - Каждый раз, когда необходимо хранить данные в памяти, это может стать потенциальной проблемой масштабируемости

Ресурсы

Data-Intensive Text Processing with MapReduce
Jimmy Lin and Chris Dyer (Authors) (April, 2010)

Chapter3: MapReduce Algorithm Design



<http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/>

Вопросы?

