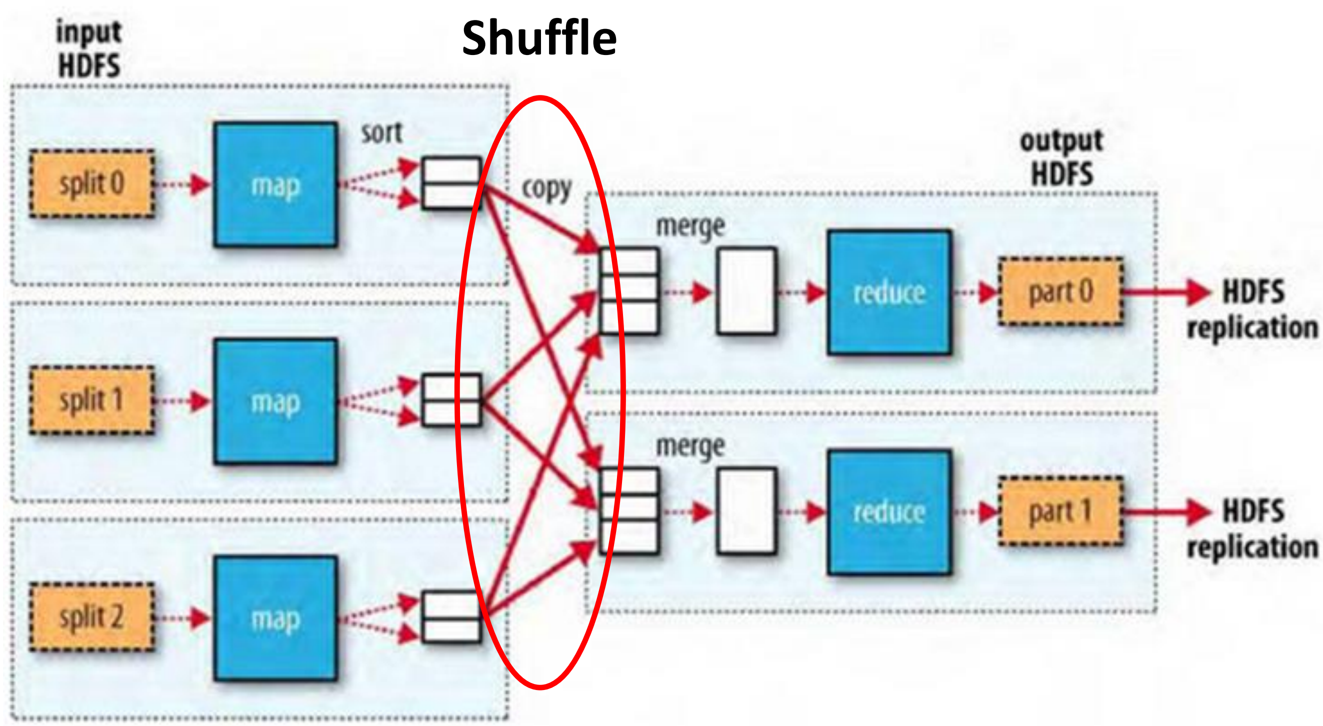
Anti-Combining for MapReduce

Alper Okcan Mirek Riedewald
Northeastern University, Boston, USA
SIGMOD 2014

MapReduce Overview



Shuffle is always the bottleneck of a MR job execution

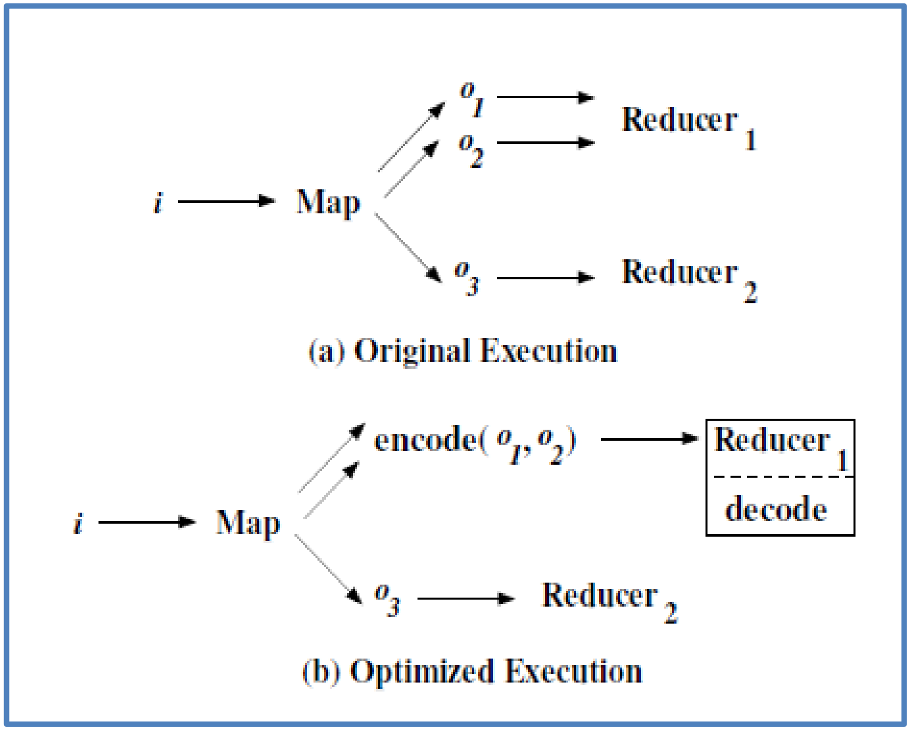
- large amounts of data are grouped, sorted, and moved across the network
- data transfer is inherent to enable parallel execution
- Network links and switches are in fact limited

Reducing network load is essential for increasing throughput in highly utilized environments

The methods of reducing network load

Methods	Advantages	Disadvantages
Combiner	User-defined	Limitation to applications; Efficiency;
Compression	The reduction of data	Overheads for compression and decompression; Same pattern;
Active Storage	Make good use of devices' computation	complex

Motivation

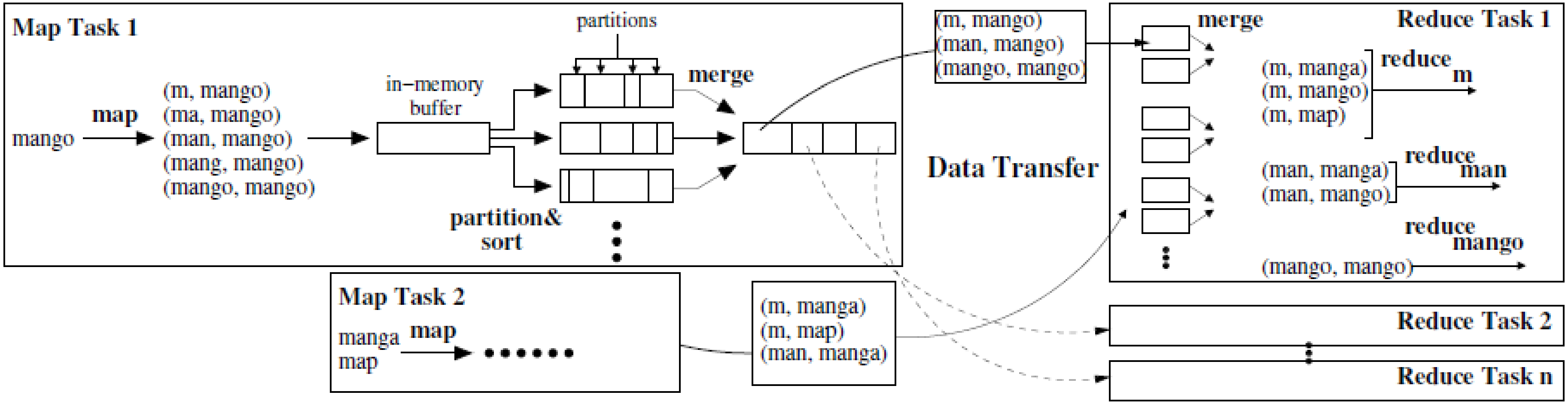


- Two design goals
  - Simple encoding/decoding functions
    - CPU cost
    - buffer space
  - Fine-grained adaptive optimization
    - the choice of encode
      - driven by the data

MapReduce Overview with Query Suggestion Example

The Query-Suggestion problem

We are given a log of search queries. For any string P that occurred as a prefix of some query in the log, pre-compute the five most frequent queries in the log starting with prefix P



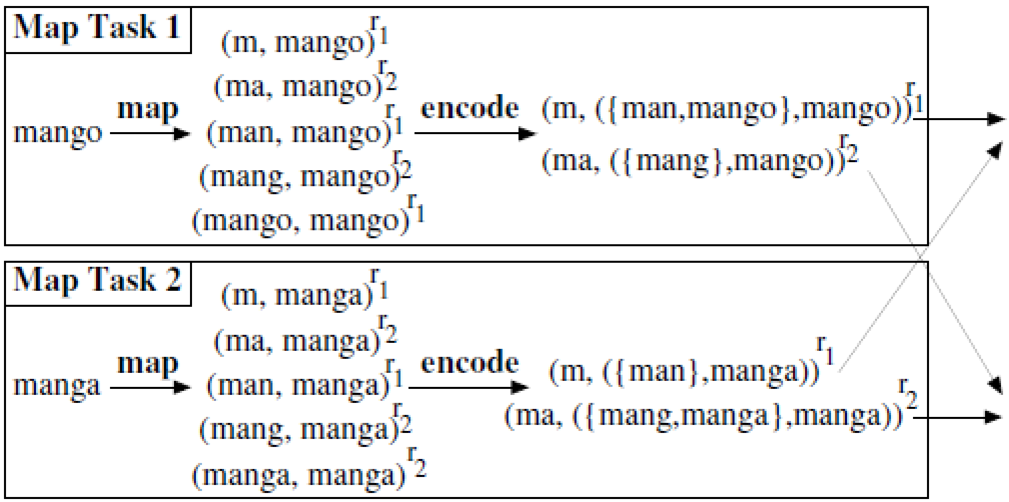
The Problem of current Query Suggestion execution

- For a search query of length n
  - the Map function will generate n output records
  - each Map function call's output is quadratic in its input size
    - each output record contains the query itself

- Combiner
  - The large number of distinct query strings
  - a Combiner might not be applicable at all

Eager Sharing Strategy

- ## EagerSH Map Phase



first executes the original Map on the given input record

Then groups the original Map's output by value and partition number

For each group, a single record is emitted

Algorithm 1 : *EagerSH*'s Map Function

Input: input tuple I

```
MapOutput = O-map(I) /* Original map */
result = SELECT MIN(O.key) AS key,
(setOfOtherKeysInGroup(), O.value) AS value
FROM MapOutput O
GROUP BY getPartition(O.key), O.value
for all r ∈ result do
    Emit(r.key, r.value)
```

Eager Sharing Strategy

- ## EagerSH Reduce Phase

EagerSH 's Reduce only receives three encoded records, in this case all those with key “m“, EagerSH 's Reduce scans through all records with that key and inserts into Shared the corresponding key/value combinations for all keys encoded in the value component.

Standard Reduce

Reduce Input

Key	Value
m	manga mango map
man	manga mango
mango	mango

EagerSH Reduce

Reduce Input

Key	Encoded Keys	Value
m	{man} {man,mango} {}	manga mango map

Shared:

(a) Initial state of reduce tasks.

Standard Reduce

Reduce Input

Key	Value
m	manga mango map
man	manga mango
mango	mango

EagerSH Reduce

Reduce Input

Key	Encoded Keys	Value
m	{man} {man,mango} {}	manga mango map

Shared:

man	manga mango
mango	mango

(b) State after first reduce call execution on values with key m.

reduce task 1 receives all the records with the keys assigned to it by the Partitioner, in key order. It then calls Reduce three times, first for key “m", followed by”man", and finally “mango" in the example.

EagerSH 's Reduce Function

Algorithm 2 : *EagerSH*'s Reduce Function

Input: $<$ key_k, KVAL = listOf(key set K, value)$>$

repeat
    altKey = Shared.peekMinKey()
    if altKey $<$ key_k **then**
        O-reduce(altKey, Shared.popMinKeyValues()) /* Original reduce on values with smaller key altKey */
until altKey $\geq$ key_k
for all (K, value) in KVAL **do**
    for all key in K **do**
        Shared.add(key, value) /* Store for later reduce calls */
Values = KVAL.getValues()
if altKey = key_k **then**
    Values = Values $\cup$ Shared.popMinKeyValues() /* Append values with same key in Shared */
O-reduce(key_k, Values) /* Original reduce */

Anti-Combining Design — Lazy Sharing Strategy

- To decrease mapper output size
  - a Combiner
    - requires records with the same key
  - EagerSH
    - requires records with the same value
  - traditional compression techniques
    - require some form of redundancy among the keys and values

When all keys and values in the output of a Map call are unique, how to decrease mapper output size

Anti-Combining Design — Lazy Sharing Strategy

EagerSH

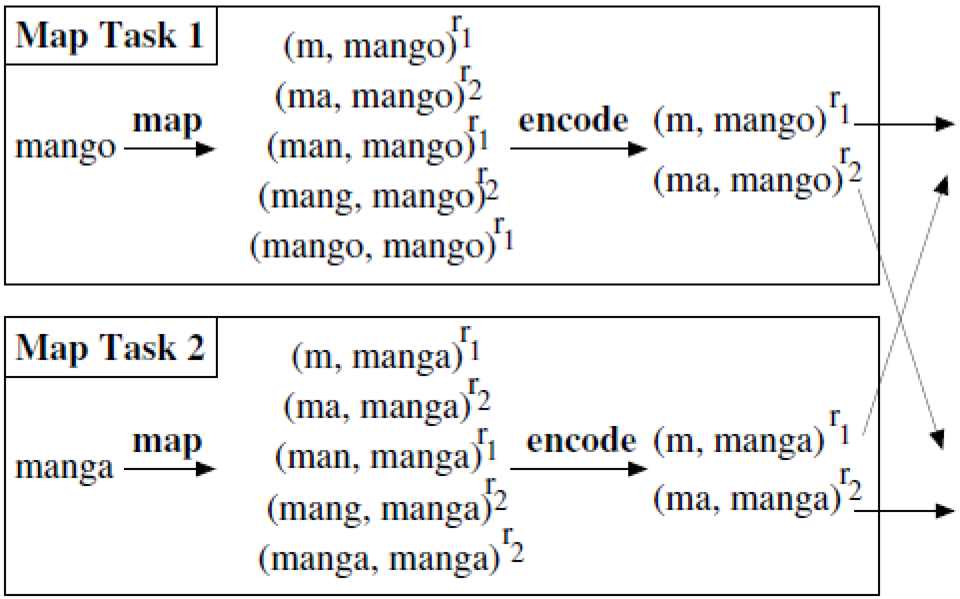
$$(k_{in}, v_{in}) \xrightarrow{map} \begin{array}{l} (k_1, v_1)^{r_1} \\ (k_2, v_1)^{r_2} \\ (k_3, v_2)^{r_2} \\ (k_4, v_2)^{r_2} \\ (k_5, v_2)^{r_2} \end{array} \xrightarrow{encode} \begin{array}{l} (k_1, (\{\}, v_1))^{r_1} \\ (k_2, (\{\}, v_1))^{r_2} \\ \\ (k_3, (\{k_4, k_5\}, v_2))^{r_2} \end{array}$$

LazySH

$$(k_{in}, v_{in}) \xrightarrow{map} \begin{array}{l} (k_1, v_1)^{r_1} \\ (k_2, v_2)^{r_1} \\ (k_3, v_3)^{r_2} \\ (k_4, v_4)^{r_2} \\ (k_5, v_5)^{r_2} \end{array} \xrightarrow{encode} \begin{array}{l} (k_1, (k_{in}, v_{in}))^{r_1} \\ (k_3, (k_{in}, v_{in}))^{r_2} \end{array}$$

LazySH: transfers the Map input record to all reduce tasks

- LazySH Map Phase



Algorithm 3 : *LazySH*'s Map Function

Input: input tuple I

```
MapOutput = O-map(I) /* Original map */
result = SELECT MIN(O.key) AS key
FROM MapOutput O
GROUP BY getPartition(O.key)
for all r ∈ result do
    Emit(r.key, I)
```

First computes the output of the original Map call for input record

then finds the minimal key for each reduce task, i.e., partition

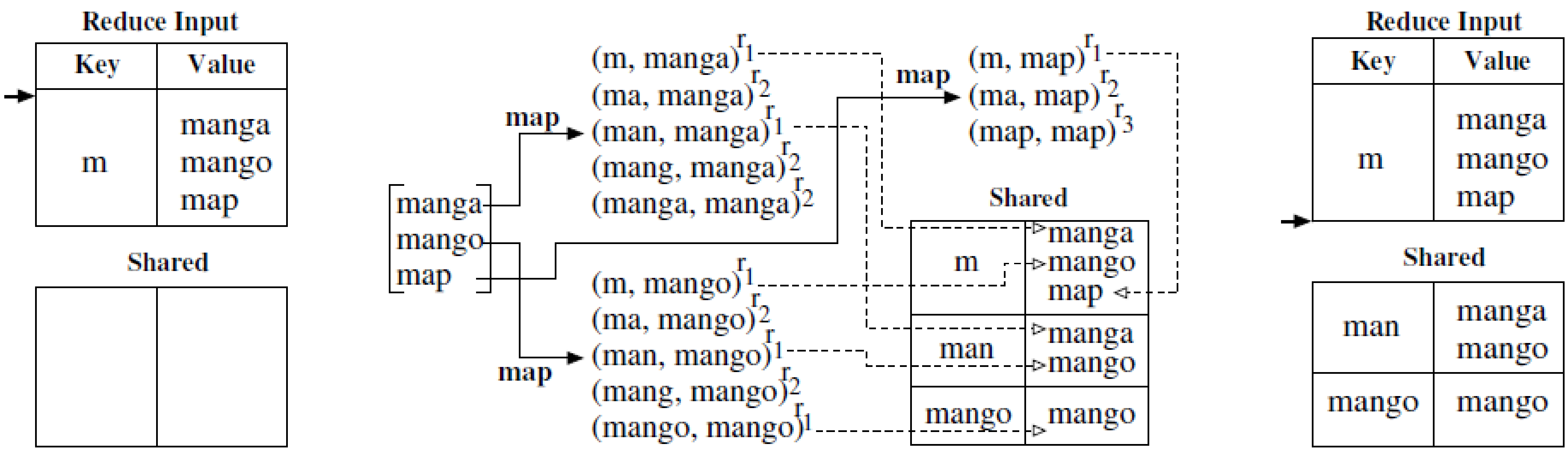
record I is emitted for each of these minimal keys

Analysis of an example

- A real query, and its length is n
  - If the Partitioner assigns all its prefixes to the same reduce task
    - Original program
      - $1+n + 2+n + \ldots.. + n+n = n(1+n)/2 + n*n$
    - EagerSH
      - $1 + (2 + 3 + \ldots.. + n) + n = n(1+n)/2 + n$
    - LazySH
      - $1+n$

Lazy Sharing Strategy

- LazySH Reduce Phase



(a) Initial state of reduce task.

(b) Decoding key/value pairs into Shared during reduce call execution on values with key m.

(c) State after first reduce call.

The reduce tasks of LazySH receive Map input, not output, therefore decoding in the reducer requires re-execution of the original Map function

Anti-Combining Design — ENABLING ADAPTIVE RUNTIME OPTIMIZATION

```
// Original Mapper class
Class Mapper {
   map(K, V, context) {...}
   setup(...) {...}
   cleanup(...) {...}
}
```

```
// Extended Context class
Class AntiContext {
   mapOutput

   write(K key, V val) {
      mapOutput.insert(key, val)
   }

   getOutput() {
      return mapOutput
   }
}
```

```
// Adaptive Mapper for Anti–Combining
Class AntiMapper {
   Mapper o_mapper, AntiContext a_context

   setup(...) { call o_mapper.setup() }
   cleanup(...) { call o_mapper.cleanup() }

   map(K key, V val, context) {
      o_mapper.map(key, val, a_context) //Call original map, measure cost
      mapOutput = a_context.getOutput()
      mapOutput.partition(Partitioner)     //Call Partitioner, measure cost
      if ((cost of map + cost of partition call) * number of partitions > T)
         for all partitions P in mapOutput do context.write(EagerSH–encoded P)
      else
         for all partitions P in mapOutput
            if (size of EagerSH–encoded P < size of input record (key, val))
               use EagerSH to encode P
            else
               use LazySH to encode P
            context.write(encoded partition P)
   }
```

first executes the original program's Map function on input record (key, val) (through the o_mapper object)

then partitions the output

compute the total cost of re-executing o mapper.map and getPartition to determine EagerSH or LazySH

Anti-Combining Design — ENABLING ADAPTIVE RUNTIME OPTIMIZATION

```
// Original Reducer class
Class Reducer {
   reduce(K, iter<V>, context) {...}
   setup(...) {...}
   cleanup(...) {...}
}
```

```
Class Shared<K, V> {
   minHeap<K>
   hashMap<K, V>
   keyComparator
   groupingComparator

   add(K, V) {...}
   popMinKeyValues() {...}
}
```

```
// Adaptive Reducer for Anti–Combining
Class AntiReducer {
   Shared
   Reducer o_reducer; Mapper o_mapper

   setup(...) {  initialize Shared; call o_reducer.setup() }
   cleanup(...) { cleanup Shared; call o_reducer.cleanup() }

   reduce(K key, iter<adaptiveV> values, context) {
      repeat {
         altKey = Shared.peekMinKey()
         if (altKey < key)
            o_reducer.reduce(altKey, Shared.popMinKeyValues(), context)
      } until altKey >= key
      for all val in values {
         determine encoding used for val
         decode val and insert key–value pairs into Shared
         // Decoding for LazySH calls o_mapper.map and the
         // original Partitioner, which is available through the context
      }
      o_reducer.reduce(key, Shared.popMinKeyValues(), context) }
   }
}
```

Evaluation

- Platform
  - a 12-machine cluster running Hadoop 1.0.3
  - Each machine
    - a single quad-core Xeon 2.4GHz processor
    - 8MB cache
    - 8GB RAM
    - two 250 GB 7.2K RPM SATA hard disks
- Data sets

QLog	140 million real queries	4.3GB
ClueWeb09	a real data set containing the first English segment of a web crawl	7GB
Cloud	a real data set containing extended cloud reports from ships and land stations	28.8GB
RandomText	a synthetic data set containing randomly generated text records	360GB

Query Suggestion I

- the Query-Suggestion problem on QLog and explore the eect of the choice of Partitioner by comparing three alternatives

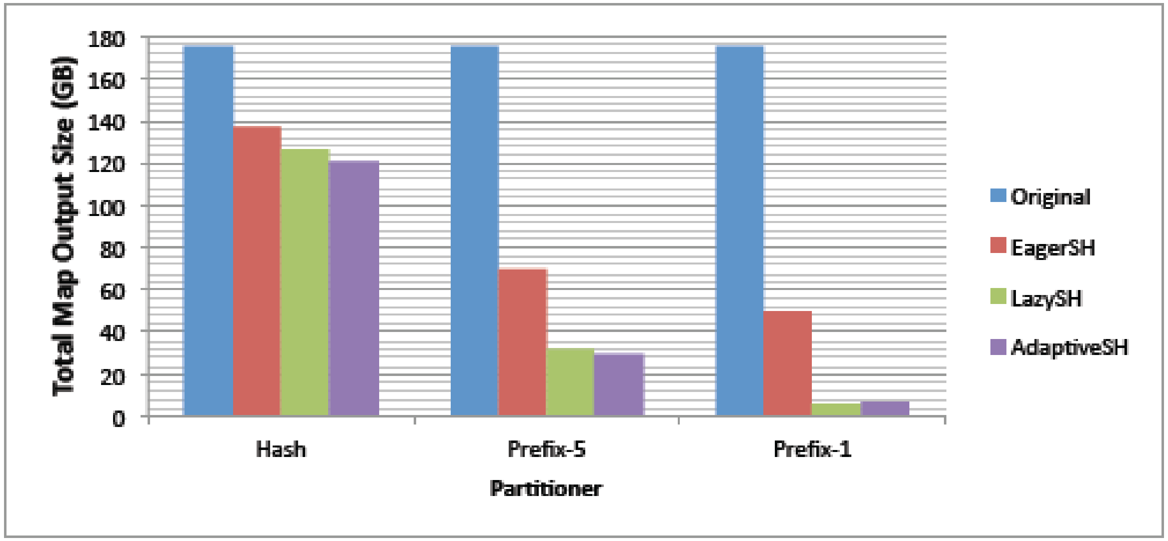


Figure 9: Total Map Output Size for Query-Suggestion

Query Suggestion II

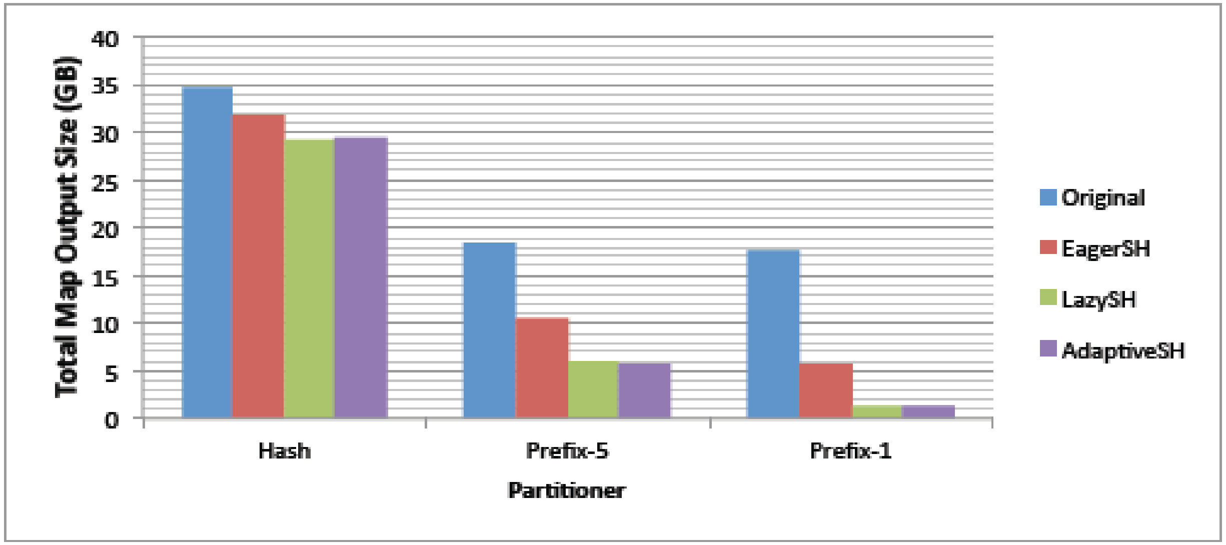


Figure 10: Total Map Output Size for Query-Suggestion using Combiner and Compression

Table 1: Total Cost Breakdown for Prefix-5, using different Compression Techniques

	Deflate	Gzip	Bzip2	Snappy	*AdaptiveSH* with Gzip
Total Disk Read(GB)	65	65	56	105	15
Total Disk Write(GB)	82	82	70	133	21
Total Map Output Size(GB)	18	18	15	30	6
Total CPU Time(1000 sec)	126.9	125.2	332.4	77.4	27.9

Effect on Disk I/O and CPU

- The cost breakdown for total disk read/write and total CPU time of the Query Suggestion
  - "-CB" and "-CP" means Combiner and compression, respectively.

Table 2: Total Cost Breakdown of Query-Suggestion

Algorithm	Total CPU Time (1000 sec)	Disk Read (GB)	Disk Write (GB)
Original	168.8	566.1	741.5
Original-CB	172.9	510.4	664.6
Original-CP	125.2	64.5	82.3
AdaptiveSH	30.8	150.8	179.9
AdaptiveSH-CB	20.8	61.9	84.9
AdaptiveSH-CP	27.9	15	20.6

CPU Intensive Workloads

- the performance of Anti-Combining for CPU intensive workloads by adding extra CPU intensive calls to the Map function

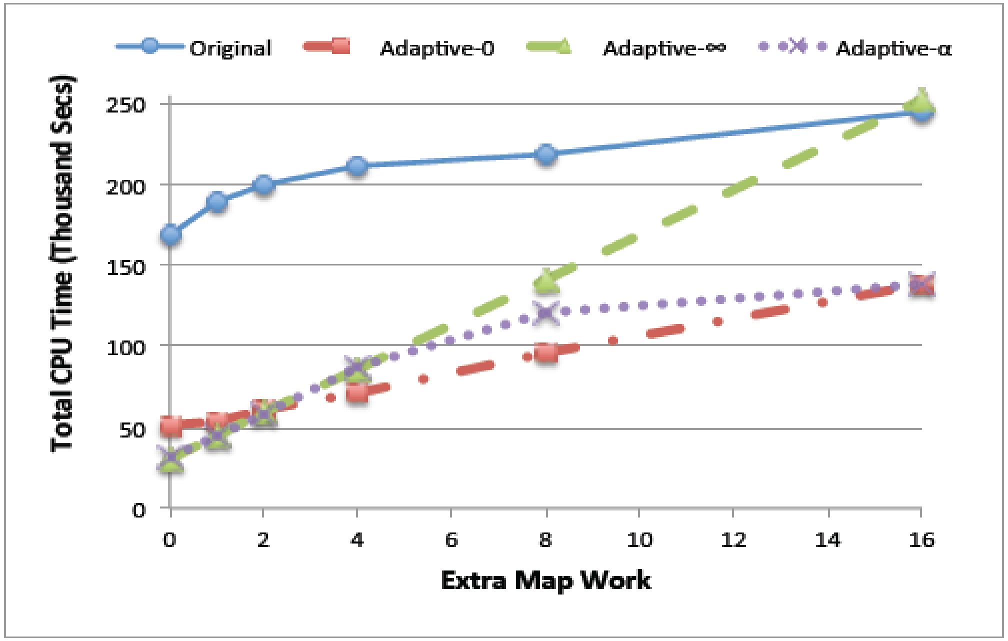


Figure 11: Total CPU Time using Runtime Cost-Based Optimization

Conclusions

- Anti-Combining
  – A novel approach for reducing the amount of data transferred between mappers and reducers
  – Shifts mapper-side processing to the reducers
  – Adaptive runtime optimization

- Applicable的问题