ORACLE

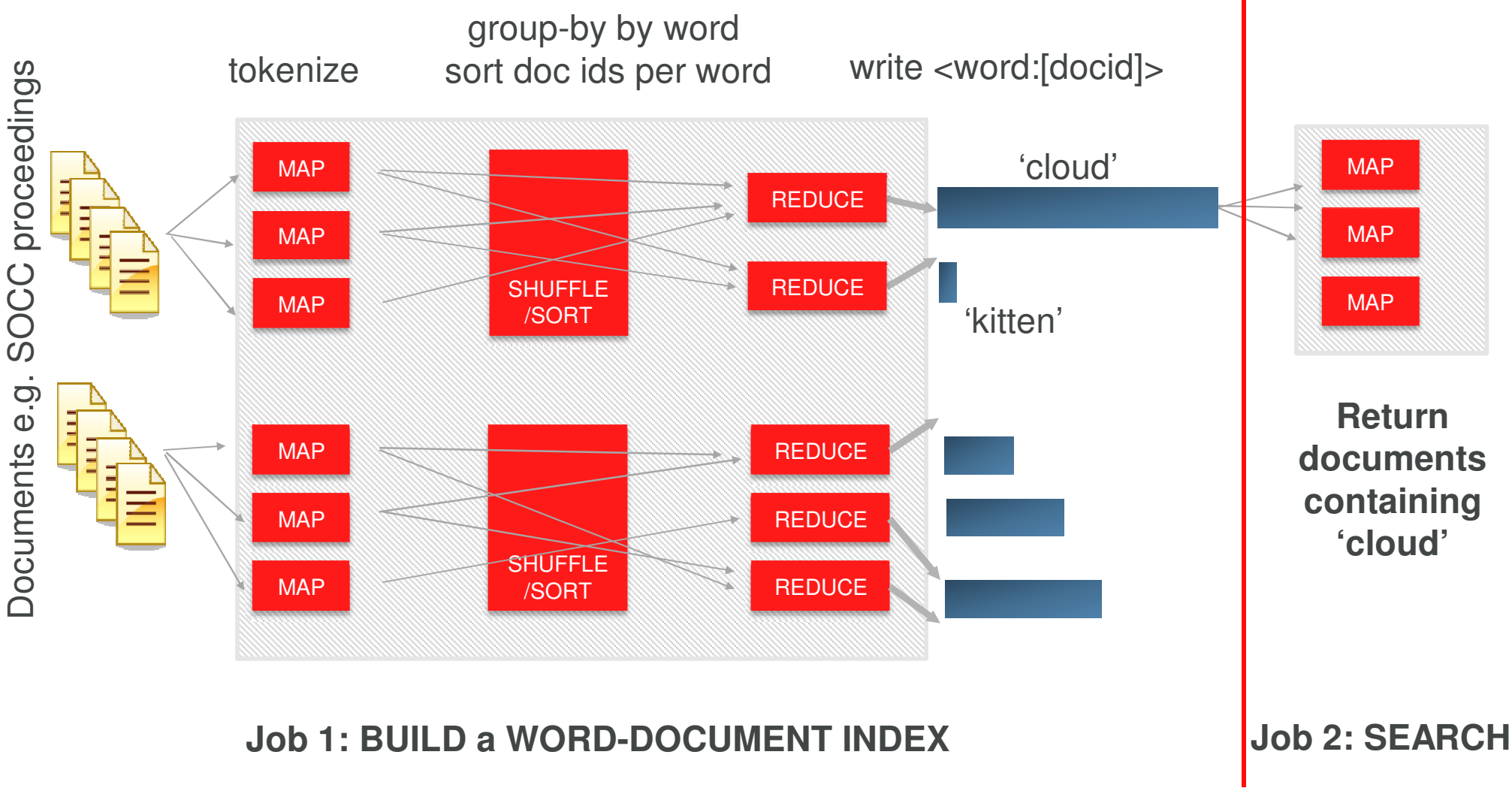
ACM Symposium on Cloud Computing
SoCC'12

Balancing Reducer Skew using Progressive Sampling

Smriti Ramakrishnan
Garret Swart
Aleksey Urmanov

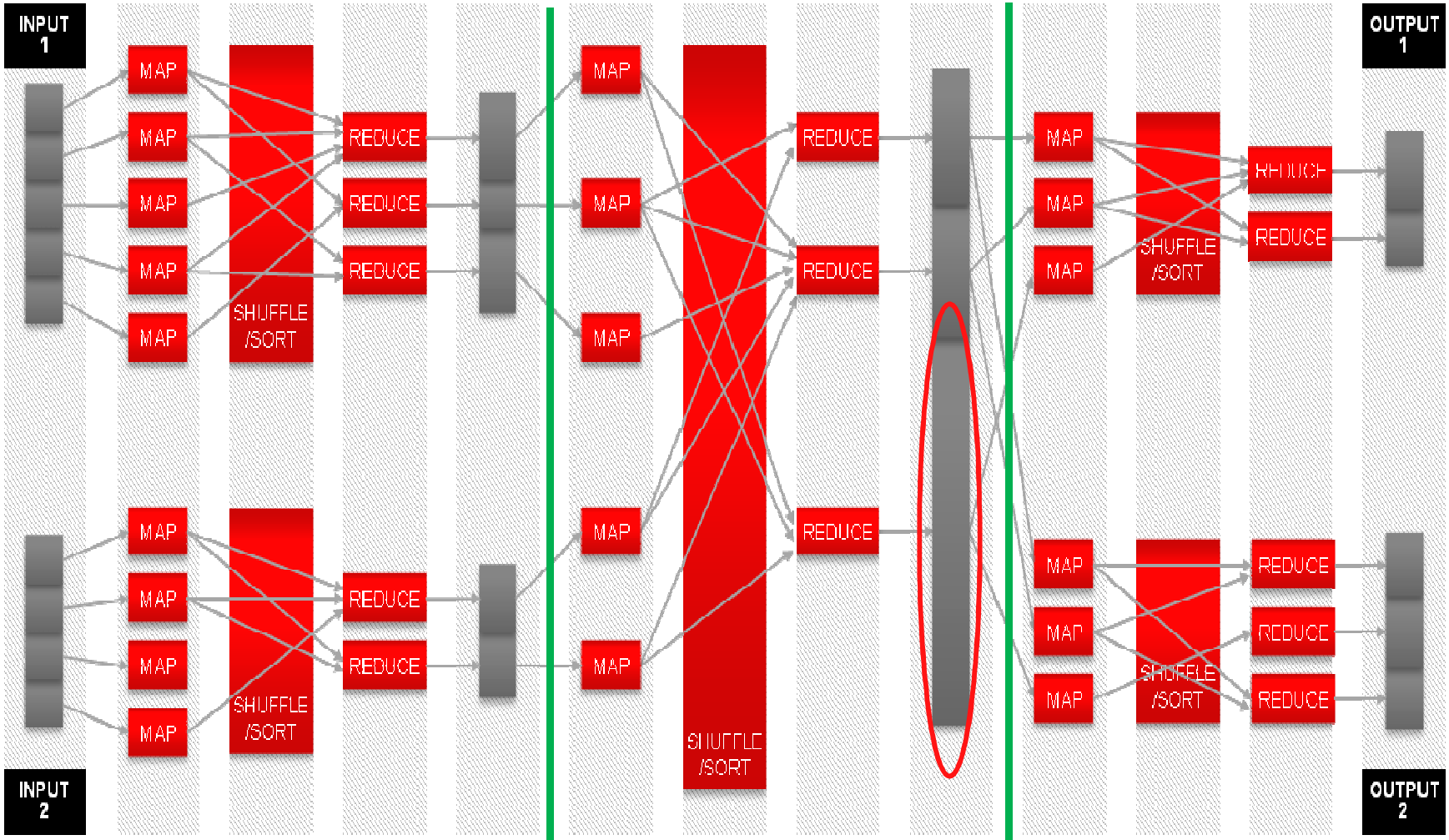
Typical MapReduce Job

Inverted Index Creation: data structure to speedup text searches



Straggler tasks delay jobs and pipelines

Elapsed time of a Hadoop M/R job is the end time of its slowest reducer



Data skew in mapper output affects reducer running times

- Task is susceptible to Data Skew if its running time depends on the size of the input
  - Other types: Computational Skew, Machine Skew
- Data skew affects all four stages of Hadoop's reduce task
  - Copy keys-values from map nodes, merge value by key, reduce(), write
- In addressing data skew, partitioning strategy is important

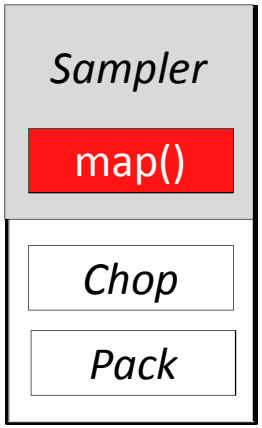
Hash partitioning may not address skew

Partitioning assigns map-output keys to reducers

- To address skew, partitioning scheme must be aware of
  - (a) key load
  - (b) reducer capacity
- Key load taxonomy
  - **Large**: key load > reducer's capacity
  - **Medium**: key load < reducer capacity
    - but large enough that hash collisions likely to overload a reducer
  - **Small**: key load << reducer capacity
    - hash collisions unlikely to overload a reducer

Solution: a load-aware partitioning strategy

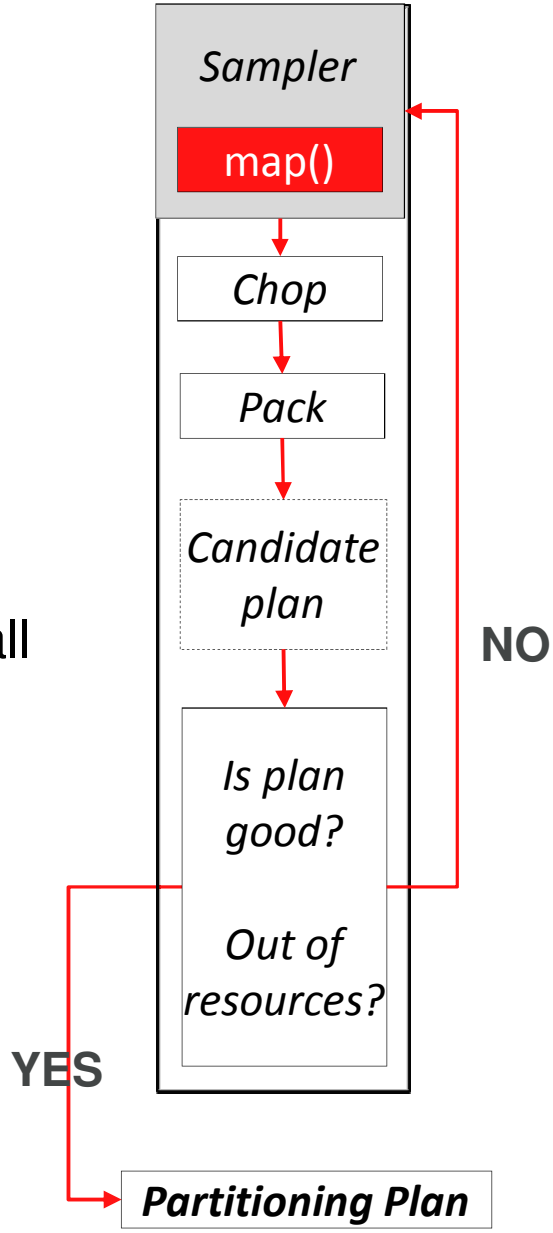
- Identify Large and Medium map-output keys
  - Sample: larger the key, more likely to be sampled (simple random sample)
    - Need not estimate full key histogram
- Two pronged partitioning attack
  1. Chop Large key payload into Medium keys
  2. Bin pack Medium keys instead of hashing them



- Reminiscent of database work in balancing parallel group-bys, joins

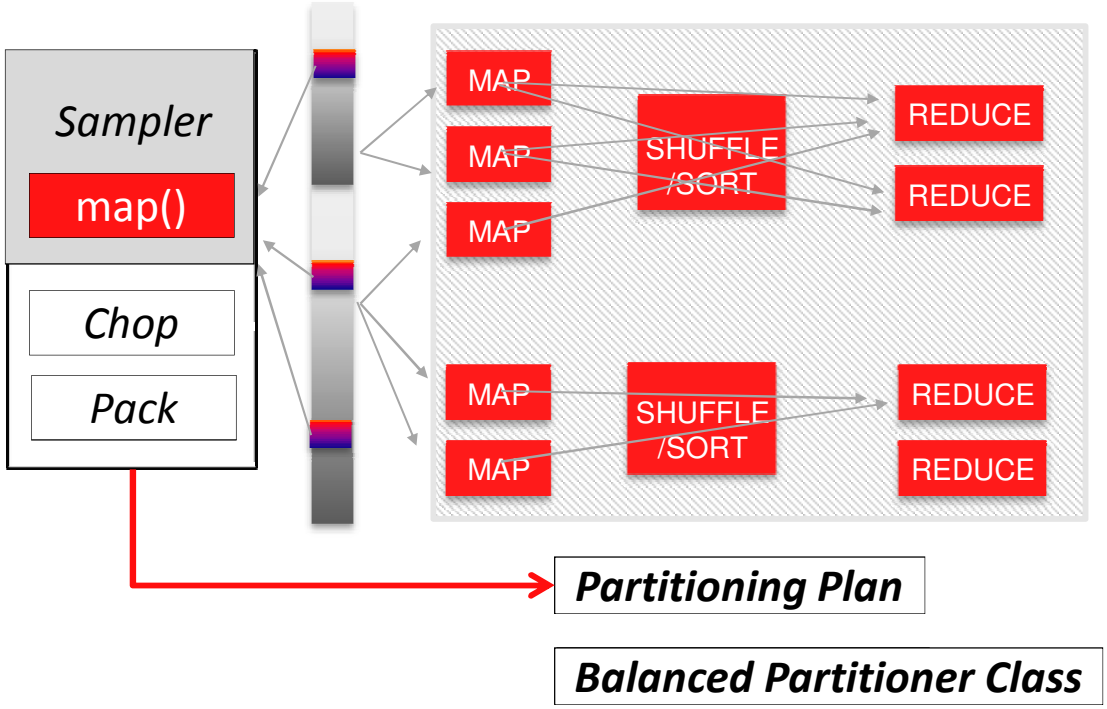
Algorithm Highlights

- Sample the output of **arbitrary map functions**
- Progressive: objective-driven sampling
  - Until error bars on predicted reducer loads are small
  - User need not specify sample size
  - Potential: sampler can decide execution strategy
    - distributed mode for large samples



Implemented as a static load balancer

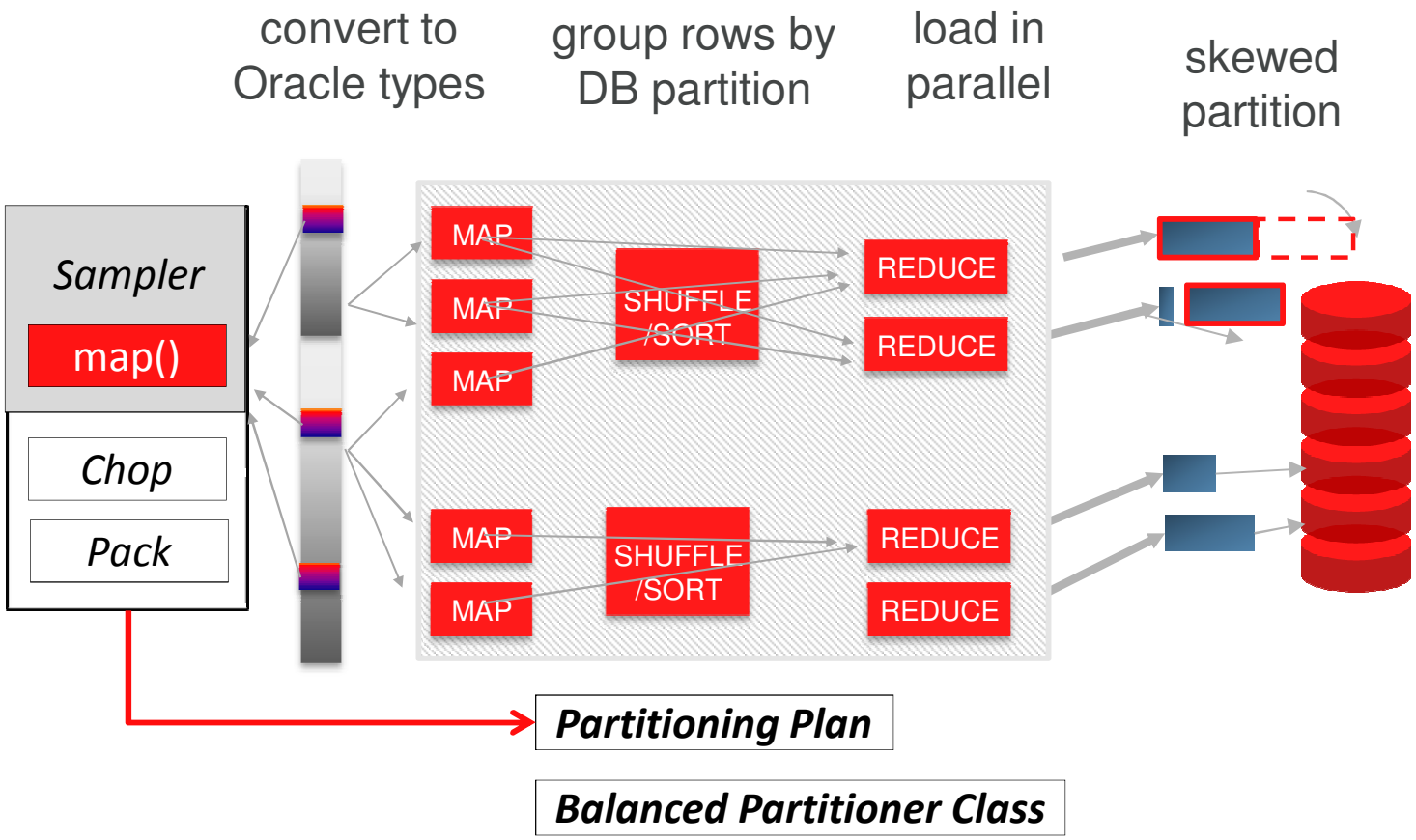
(1) Static: runs before job
(vs. dynamic →modify Hadoop to buffer map output, repartition [Vernica et al EDBT'12, Kwon et al SIGMOD'12]



(2) Block sampler: small, randomly distributed Hadoop Input Splits

Ships in the Oracle Loader for Hadoop

Efficient parallel data loads from Hadoop to partitioned Oracle tables



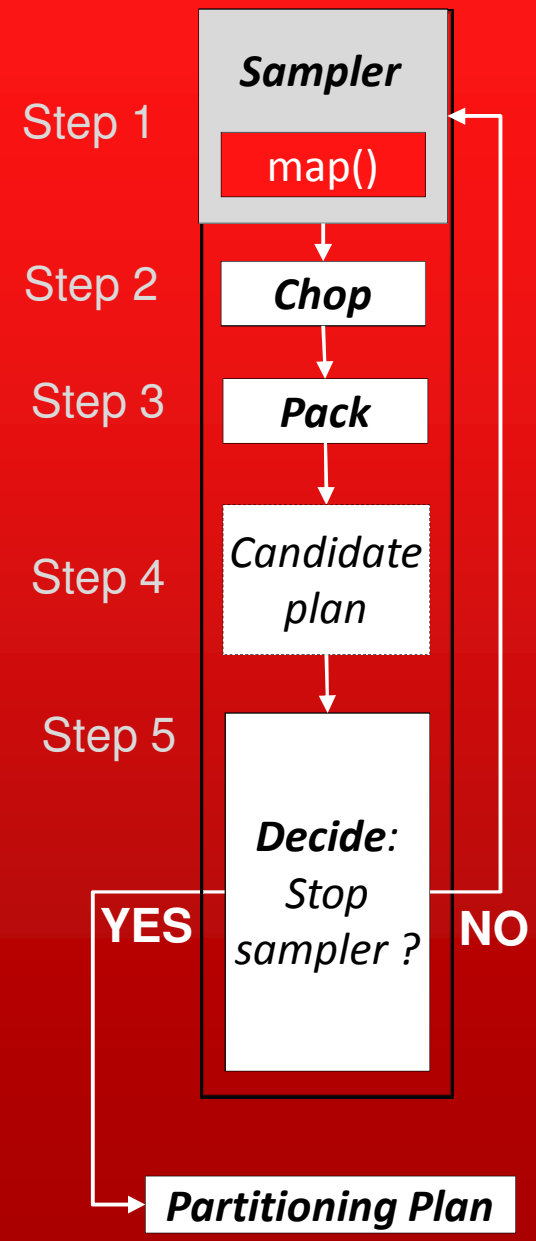
Collaborative work with the Oracle Loader for Hadoop team

Problem

Solution Overview

Details of Sample-Chop-Pack Algorithm

Results

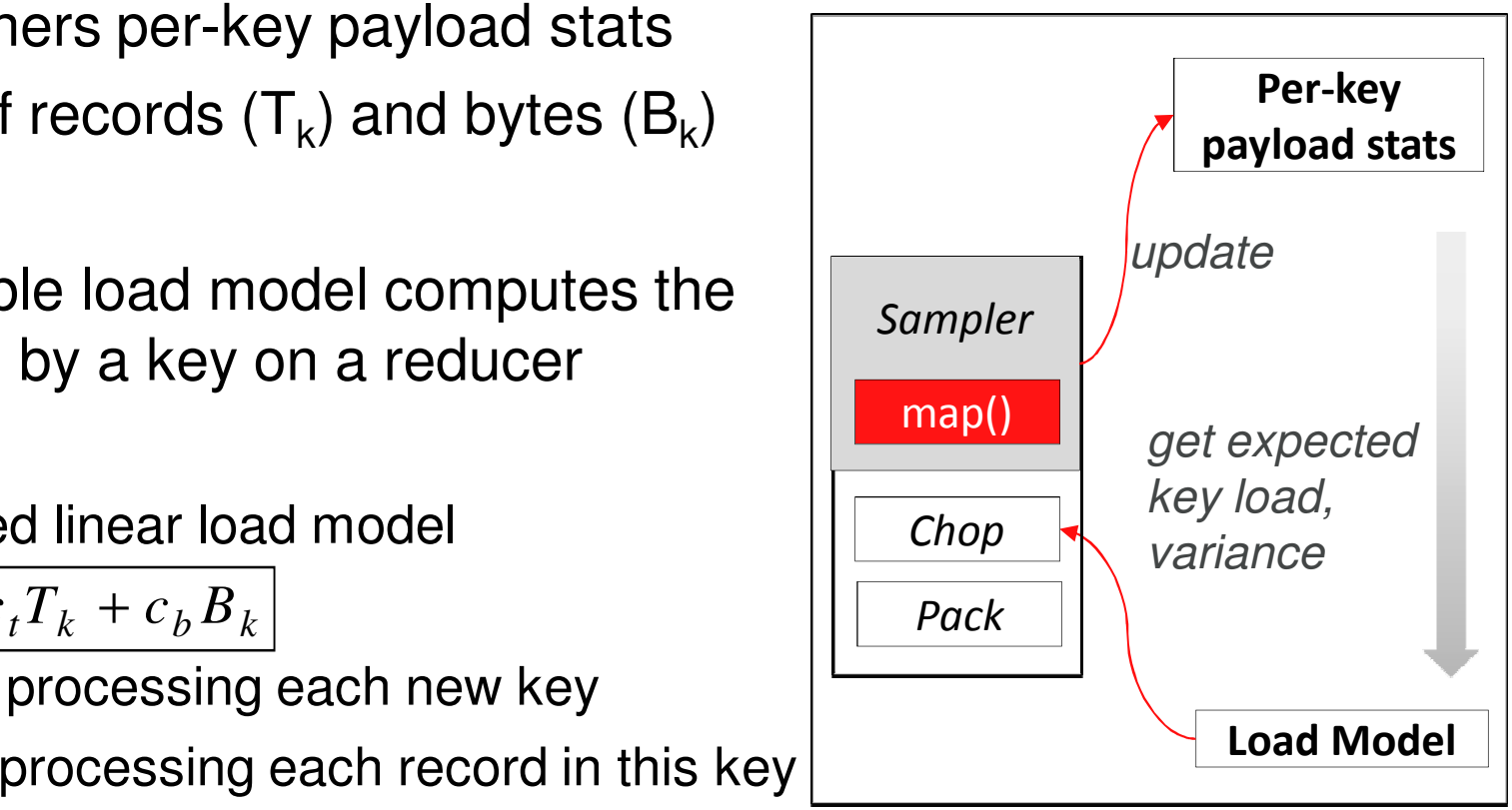


ORACLE

Step 1: SAMPLE and estimate key loads

To estimate reducer load, we need to estimate key loads

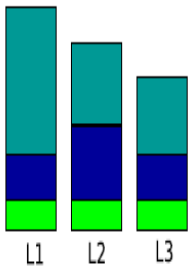
- Sampler gathers per-key payload stats
  - number of records (T_k) and bytes (B_k)
- A customizable load model computes the load induced by a key on a reducer



Default: weighted linear load model

$$L_k = c_k + c_t T_k + c_b B_k$$

- c_k: cost of processing each new key
- c_t: cost of processing each record in this key
- c_b: cost of processing bytes in the records

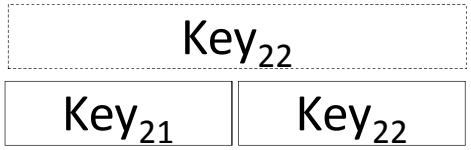


[Keys figure credit: Geng Li]

Step 2: CHOP Large keys into Medium keys

Split Large keys into Medium keys

- Partition the values associated with a Large key
  - Option 1: Range-partition the sampled values
  - Option 2: Hash the sampled values

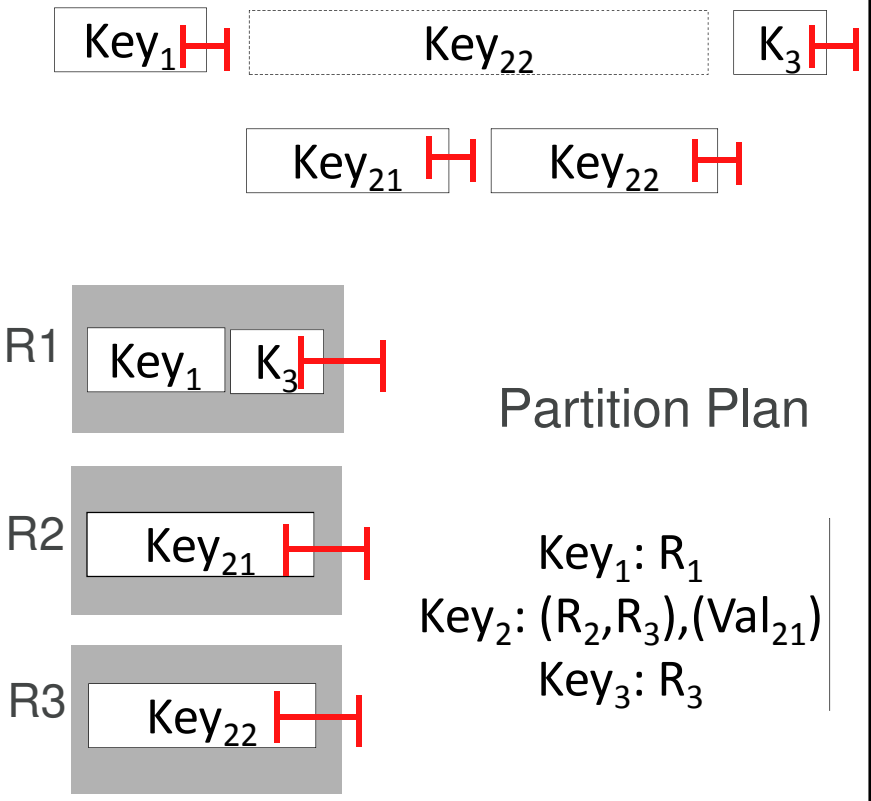


- Chopping a sampled key adds extra variance to its load
- Chopping only works for distributive reduce() → make it optional
  - Chopping is often OK in parallel M/R
    - joins handled as special case [e.g. fragment-replicate]

Step 3: PACK sampled keys to reducers

Generate a candidate partitioning of sampled keys to reducers

- Assign Medium keys to reducers using greedy bin-packing
- **Bin packing of random variables**
- Don't pack Small keys, hash them
  - Not stored in the partition plan
    - Small plan, low overhead



Step 4: UPDATE reducer loads

Based on the candidate partitioning plan

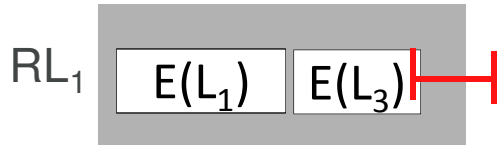
- RL: Reducer load is the sum of its keys' loads L_k

$$RL_r = \sum_{k \in K(r)} L_k$$

- E(RL): expectation is the sum of expected key loads
- Var(RL): variance is the sum of key load variance
  - Independence assumption for keys

$$\mathrm{E}(RL_r) = \sum_{k \in K(r)} \mathrm{E}(L_k)$$

$$\mathrm{Var}(RL_r) = \sum_{k \in K(r)} \mathrm{Var}(L_k)$$



- Use reducer variance to get a stopping condition

Step 5: DECIDE if the plan is good enough

Stop condition for progressive sampling

- Goal: reducer load must be within 5% (δ) of predicted load with 95% probability (α)

$$\forall r, \Pr(RL_r \leq (1+\delta)E(RL_r)) \geq \alpha^{1/R}$$

- Stop sampling when the reducer load variance Var(RL) is small enough
  - Hypothesis test on reducer load distribution

$$\forall r, \frac{Var(RL_r)}{E(RL_r)^2} \leq \left(\frac{\delta}{RL^{inv}_{(1-\alpha)}} \right)^2$$

STOPPING CONDITION

- Early stopping: user can specify max sampling time, memory

Problem

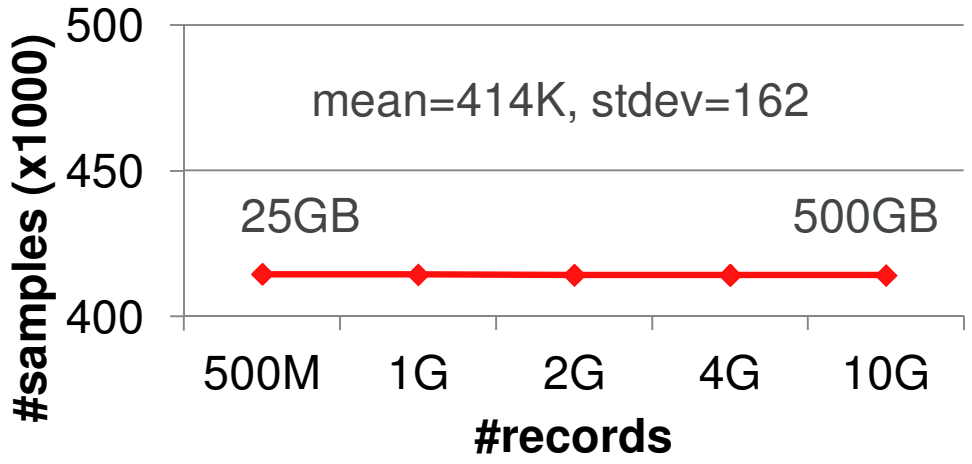
Solution Overview

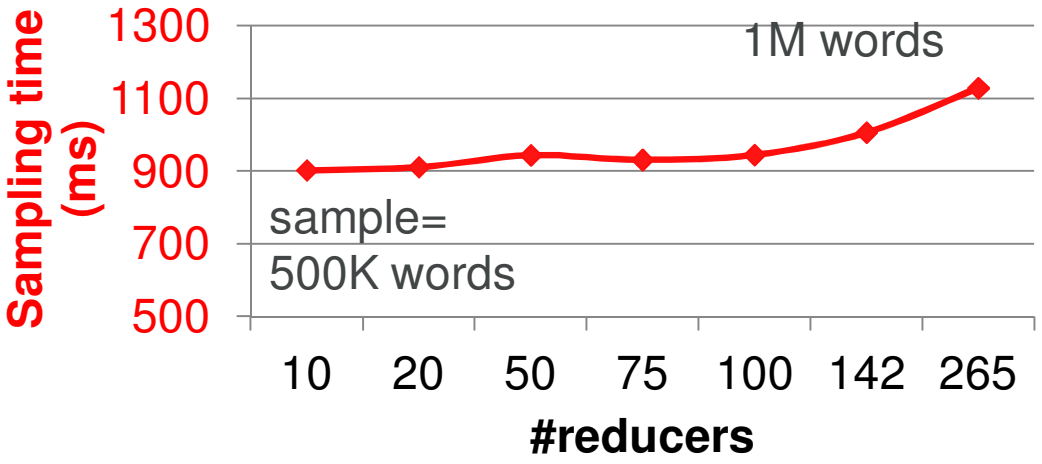
Details of Sample-Chop-Pack Algorithm

Results

Sample size depends on data skew, not size

Zipf-skewed key loads, keys uniformly distributed across records



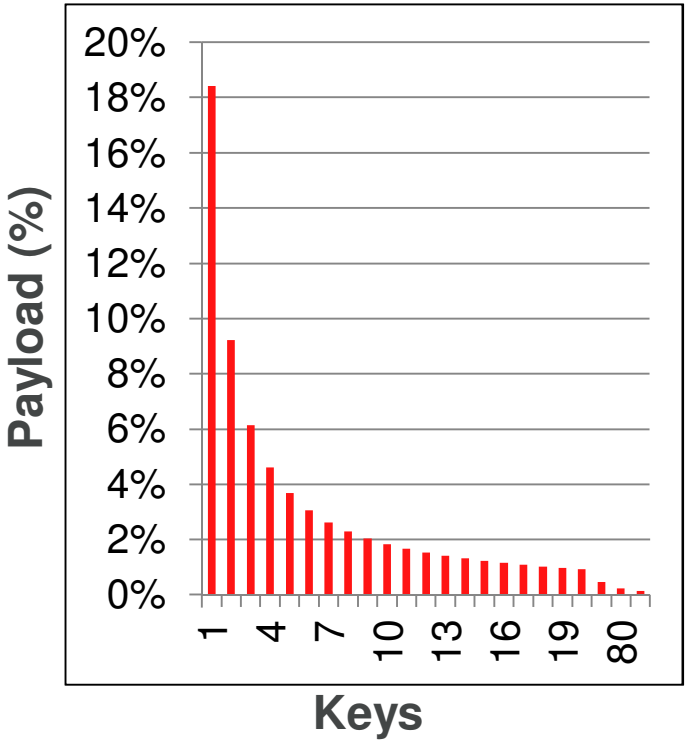


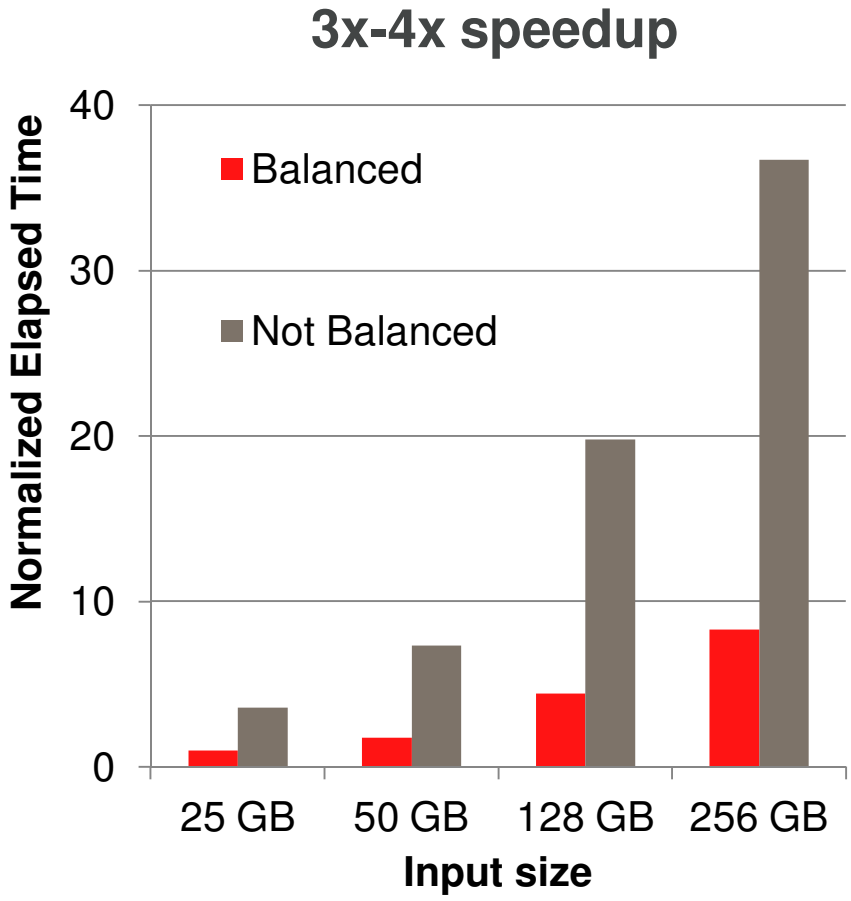
Scale out: for fixed skew, sample size and time were relatively constant with increasing data size

Load balancing time was a small fraction of total job time (<1%-3% for Zipf and Poisson skews)

Speedups depend on data skew, not size

Oracle Loader for Hadoop: 128 reducers, 128 keys (DB partitions) writing binary format files to HDFS

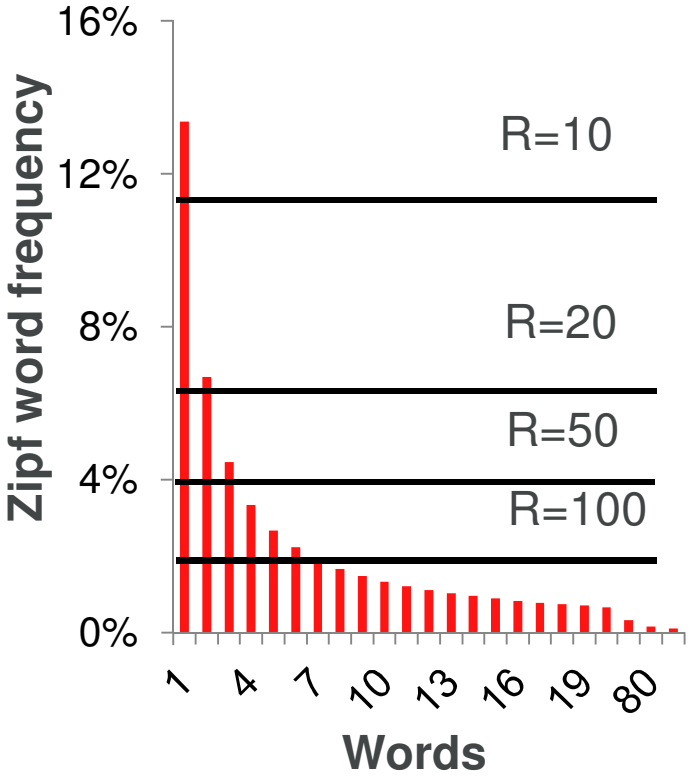


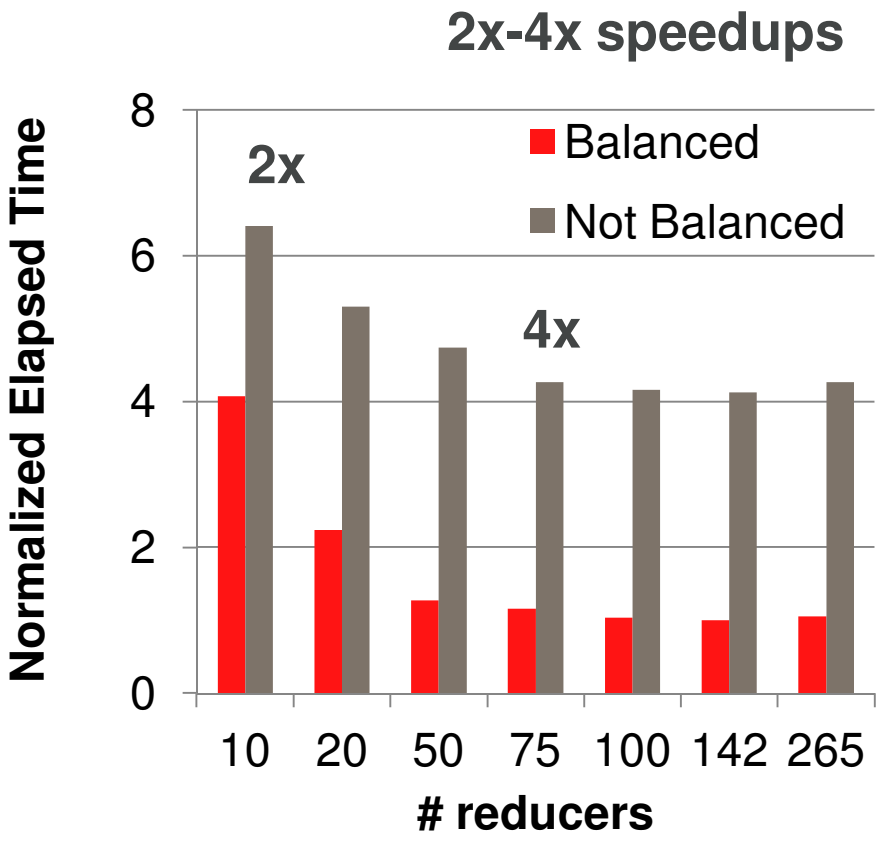


Input: Zipf-skewed key payloads

Speedups scale with number of reducers

Inverted Index, 1000 keys (words)





For a given key distribution: more reducers → more reduce data skew

Summary

Mitigate reducer data skew in MapReduce workloads

- Static balancer with a progressive sampler
  - Objective-driven sampling stop condition
- Effective for reduce-heavy MapReduce jobs
  - Speedups scale with increasing data size and #reducers
- Ships in the Oracle Loader for Hadoop

Thank you

Questions?