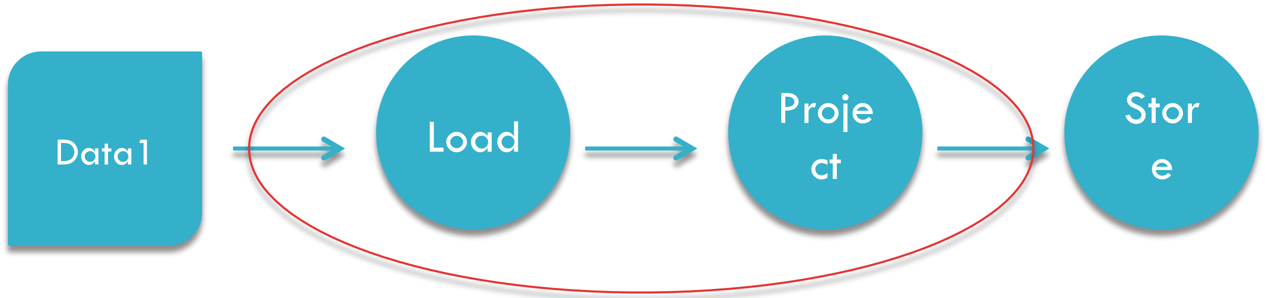
RESTORE: REUSING RESULTS OF MAPREDUCE JOBS

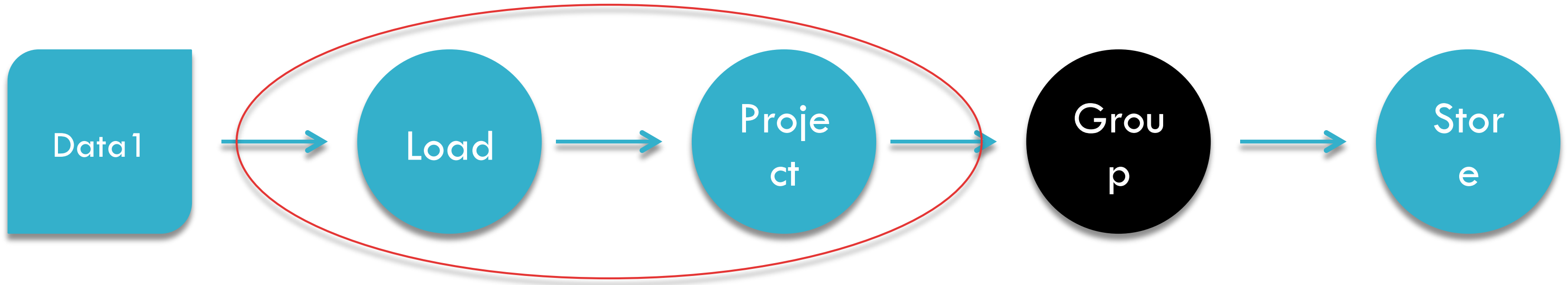
Junjie Hu

Introduction

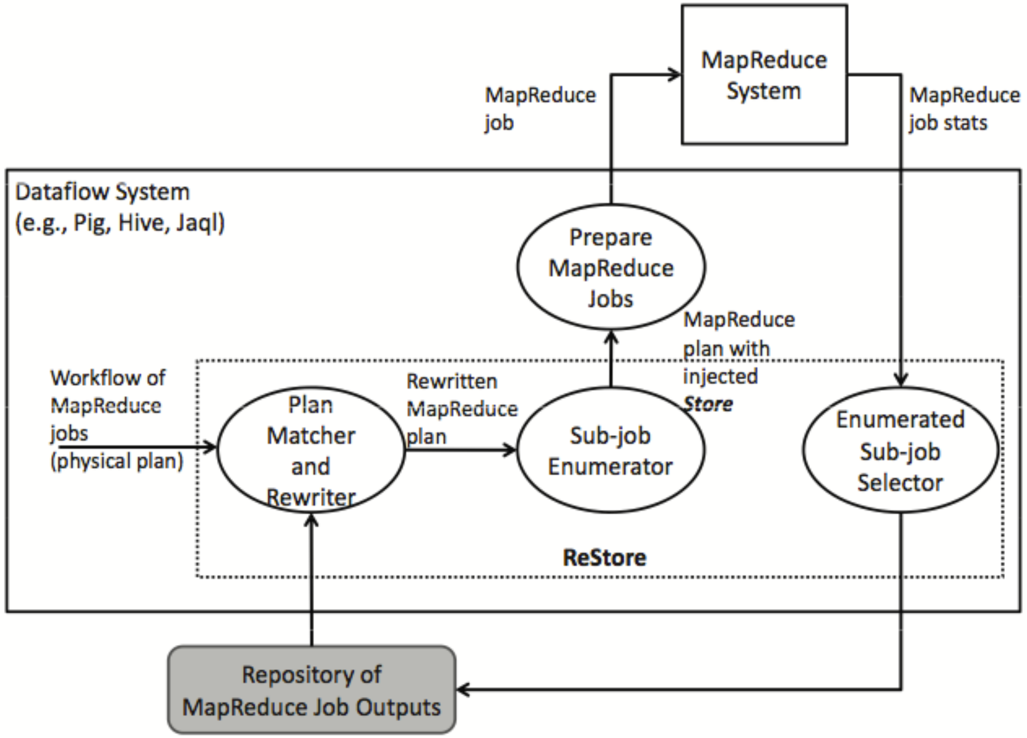
- Current practice deletes intermediate results of MapReduce jobs
- These results are not useless
- A system that reuses the output of MapReduce jobs / sub-jobs -- ReStore

Example





Restore system architecture


MapReduce System
MapReduce job
MapReduce job stats
Dataflow System (e.g., Pig, Hive, Jaql)
Prepare MapReduce Jobs
MapReduce plan with injected *Store*
Workflow of MapReduce jobs (physical plan)
Plan Matcher and Rewriter
Rewritten MapReduce plan
Sub-job Enumerator
Enumerated Sub-job Selector
ReStore
Repository of MapReduce Job Outputs

Plan Matcher and Rewriter

- Before a job J is matched, all other jobs J depends on have to be matched and rewritten to use the job stored in the repository
- A physical plan in the repository is considered matched if it is contained within the input MapReduce job

Example

- A = load ‘page_review’ as (user, timestamp, page_info);
- Store A into ‘out1’;

- A = load ‘page_review’ as (user, timestamp, page_info);
- B = foreach A generate user, page_info
- Store B into ‘out2’;

Match Algorithm

- Use DFS
- ReStore uses the first match(greedy)
- Rules to order the physical plans:
  1) A is preferred to B if all the operators in B have equivalent operators in A(A subsumes B)
  2) Based on the ratio between I/O size, execution time

Two types of reuse

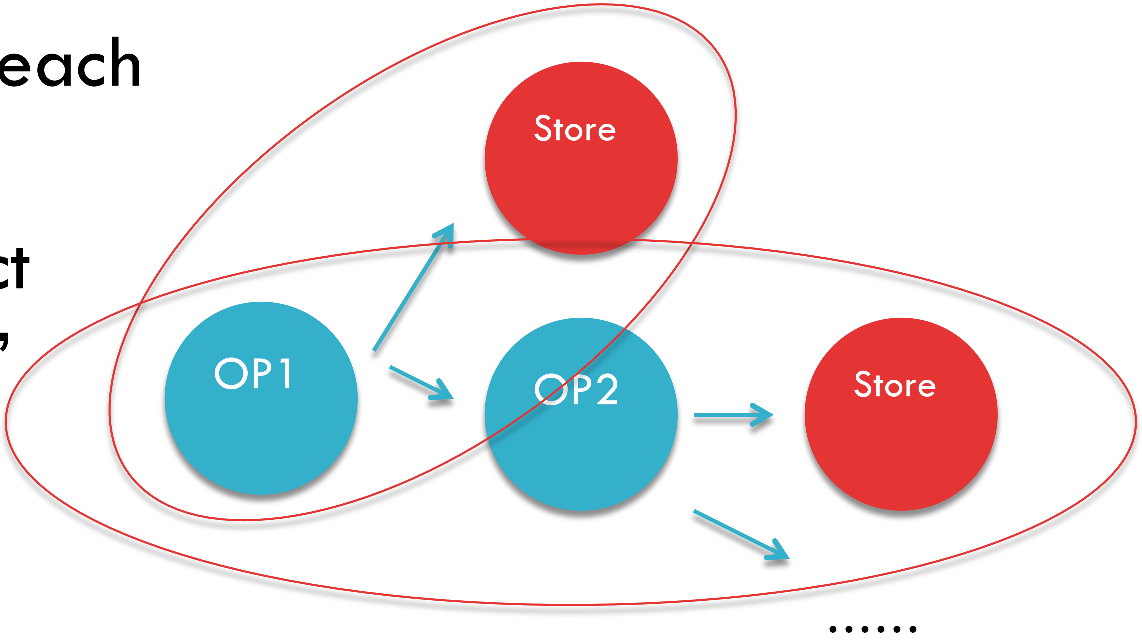
- Job
  pros: 1) easy to reuse 2) already stored
  cons: 1) not always reusable
- Sub-jobs (how to generate)
  pros: 1) more opportunities to be reused
  cons: discuss later

Discussion

- Why not always reuses jobs?
- The challenge in reusing sub-jobs?
- The disadvantages in reusing sub-jobs?

How to generate sub-jobs

- Inject ‘store’ after each operators
- Use heuristics, inject ‘store’ after ‘good’ candidate



Heuristics for choosing sub-jobs

- Conservative Heuristic the operator that reduces the input-size. E.g.: project, filter.

- Aggressive Heuristic the operator that reduces input size and outputs of operators are known to be expensive. E.g.: join, group, project,filter

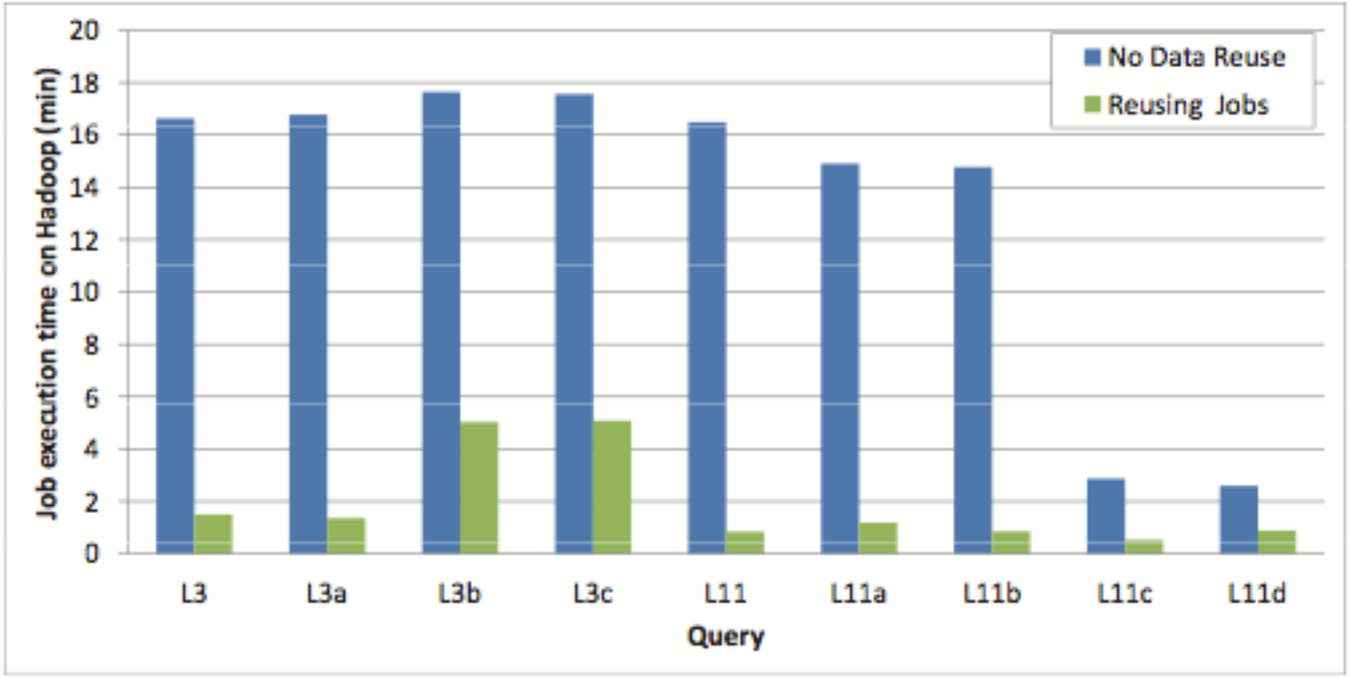
The property of the job should be kept in the ReStore Repository

- Property 1: can reduce the execution time of a workflow that contains this job/sub-job
- Property 2: can be reused in future workflows
- Check these properties based on statistics of MapReduce system

Experiment

- Use PigMix: a set of queries used to test Pig performance. E.g.: L3 join, L11 distinct + union
- Two instances to test: 15GB and 150GB(more details on paper)
- Speedup: improved execution time / original execution time
- Overhead: executing time in addition to injecting store operators / original execution time

Overall: effect of reusing jobs

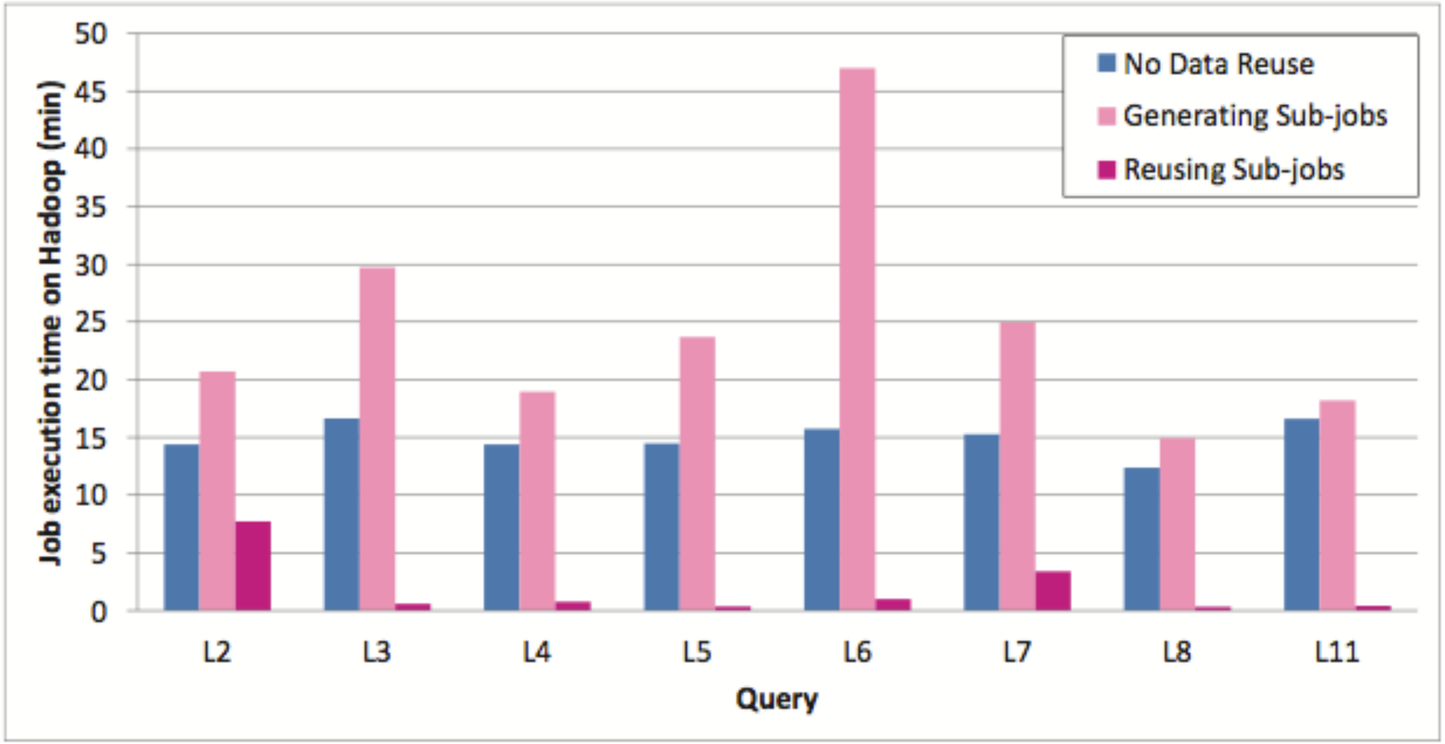


Speedup: 9.8

L3:Group and aggregate

L11:union two data sets

The effect of reusing sub-jobs outputs for data size 150GB

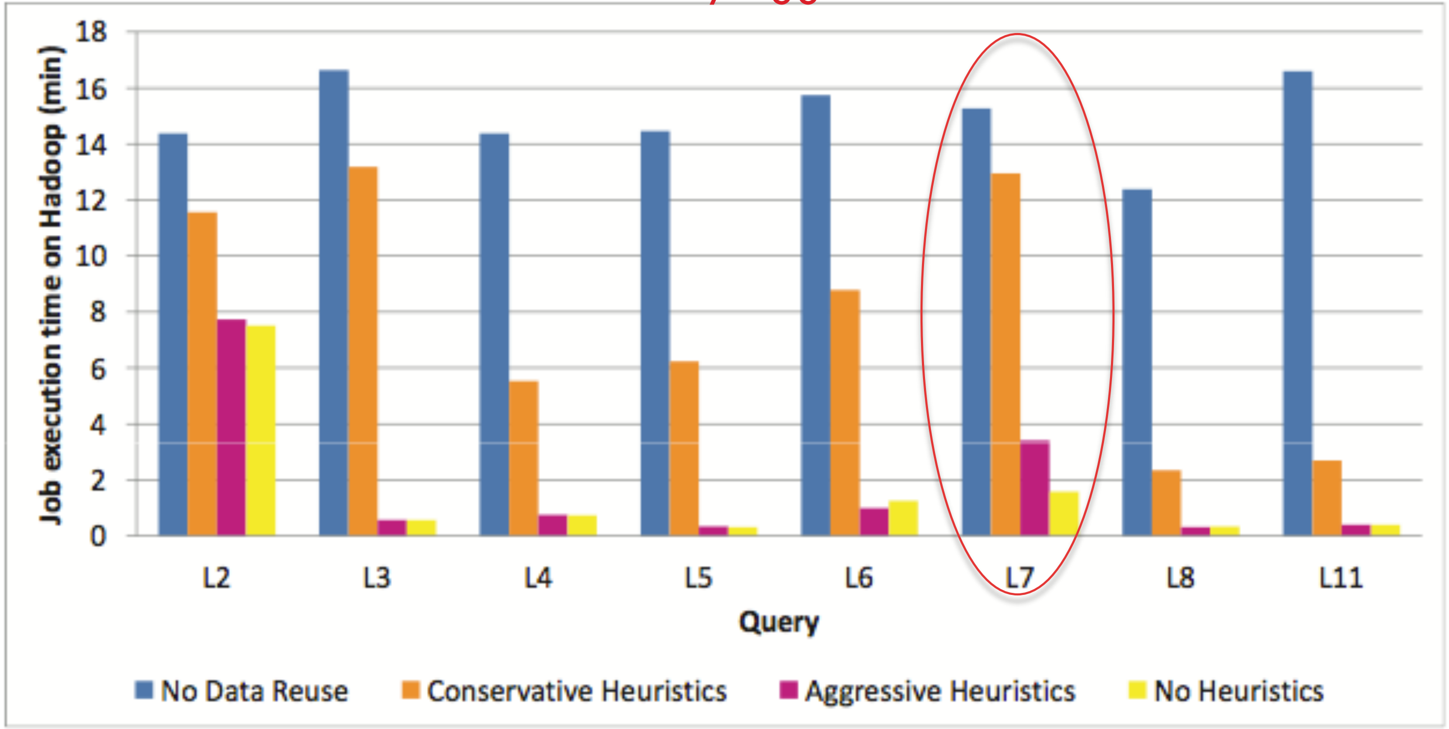


Speedup: 24.4

Overhead: 1.6

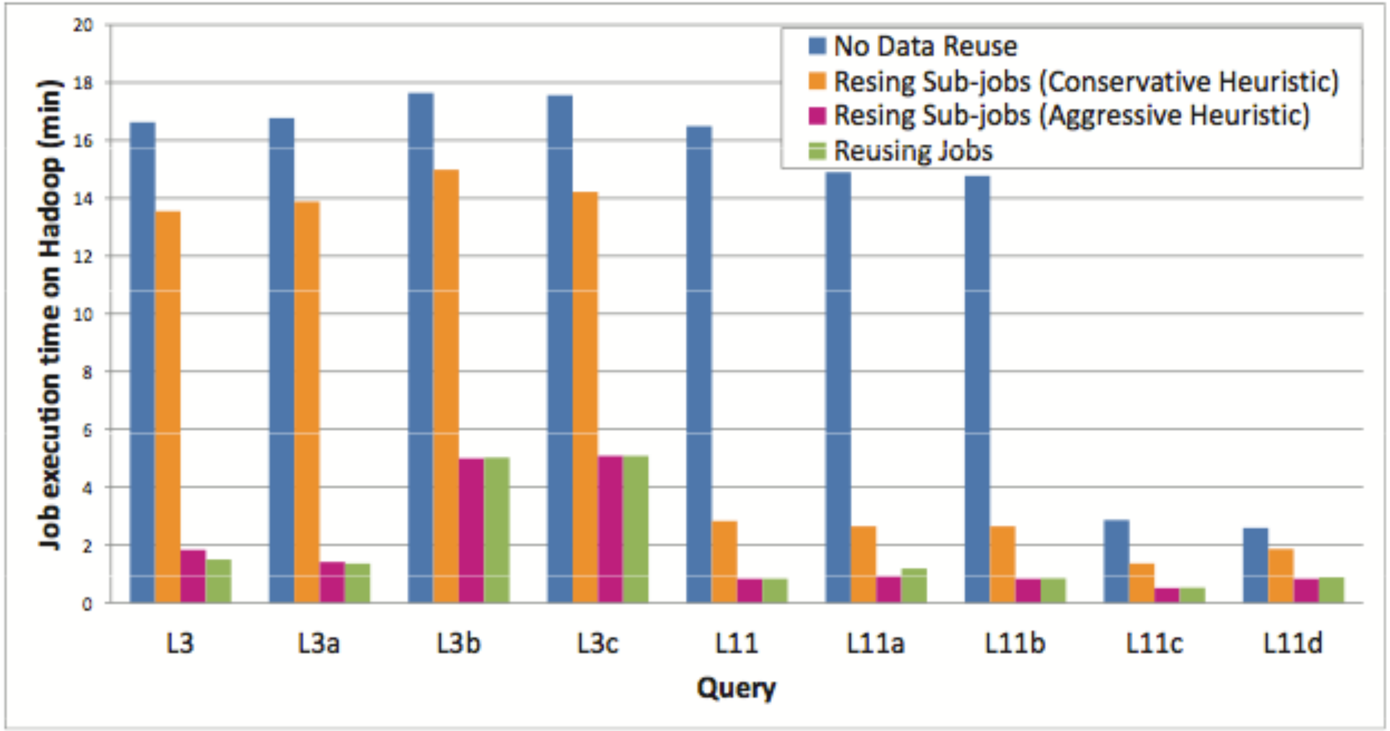
Execution time when reusing sub-jobs chosen by different heuristics

Why aggressive is much worse than no-h?



L7: nested split

Overall: Reusing whole jobs and sub-jobs



Performance on 15GB and 150GB

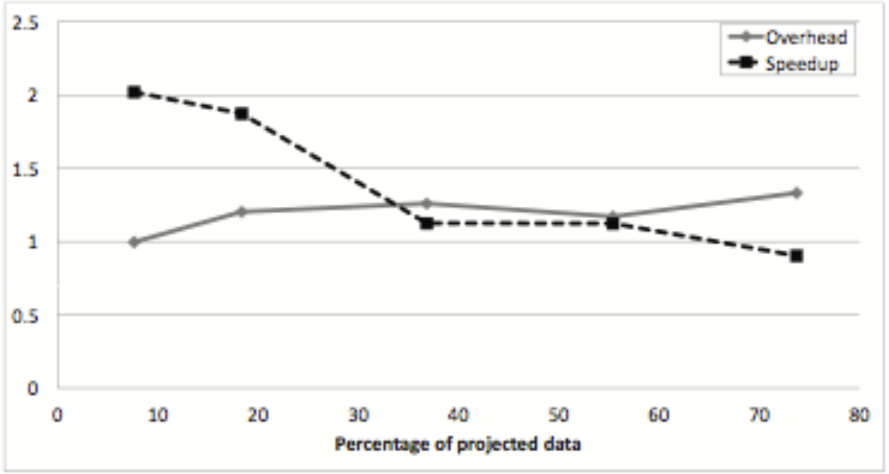
- Data size:150GB
  Speedup: 24.4
  Overhead:1.6

- Data size: 15GB
  Speedup: 3
  Over head:2.4

Win!

Effect of Data Reduction

- As the amount of data eliminated by the Filter of Projector operator increases, overhead decreases and speedup increases.



Conclusion

- Jobs of MapReduce can be reused
- Intermediate results of MapReduce jobs can be useful
- Trade-off between increased workload by injecting extra store operators and decreased workload by reusing results
- The type of command

ONLY AGGRESSIVE ELEPHANTS ARE FAST ELEPHANTS

Xueman Mou

Background

- Hadoop + HDFS
  - Each different filter conditions trigger a new MapRedece Job
  - “going shopping without a shopping list”
  - “Let’s see what I am going to encounter on the way”

What is HAIL…

- Hadoop Aggressive Indexing Library
- HAIL:
  - Keeps existing replicas in different sort orders and with different clustered indexes
  - Faster to find a suitable index
  - Longer runtime for a workload

Why HAIL

- Each MapReduce job requires to scan the whole disk
  - slow query time
- Trojan index
  - expensive index creation
  - How to use General attributes for other tasks
- HDFS keeps replicas which all have the same physical data layouts

HAIL

- Client analyzes input data for each HDFS block
- Converts each HDFS block to binary PAX
- Sort data in parallel in different sorting orders

- Datanode creates clustered index
- MapReduce job exploits the indexes

- Failover: Standard Hadoop scanning

What is PAX?

- *Partition Attributes Across*
- A data organization model
- *Significantly improves cache performance by grouping together all values of each attribute within each page. Because PAX only affects layout inside the pages, it incurs no storage penalty and does not affect I/O behavior.*

http://www.pdl.cmu.edu/ftp/Database/pax.pdf

Use case

- Bob: representative analyst
- A large web log has three fields, which may serve as different filter conditions:
  - visitDate
  - adRevenue
  - sourceIP

Upload Process

Reuse as much HDFS existing pipeline as possible

1: parse into rows based on end of line

2: parse each row by the schema specified

3: HDFS gets list of datanodes for block

4: PAX data is cut into packets

8: DN1, DN2 immediately forward pckt

9: DN3 verify checksums

10: DN3 acknowledge pckt back to DN2

6: assemble block in main memory

7: Sorts data, create indexs, form HAIL block

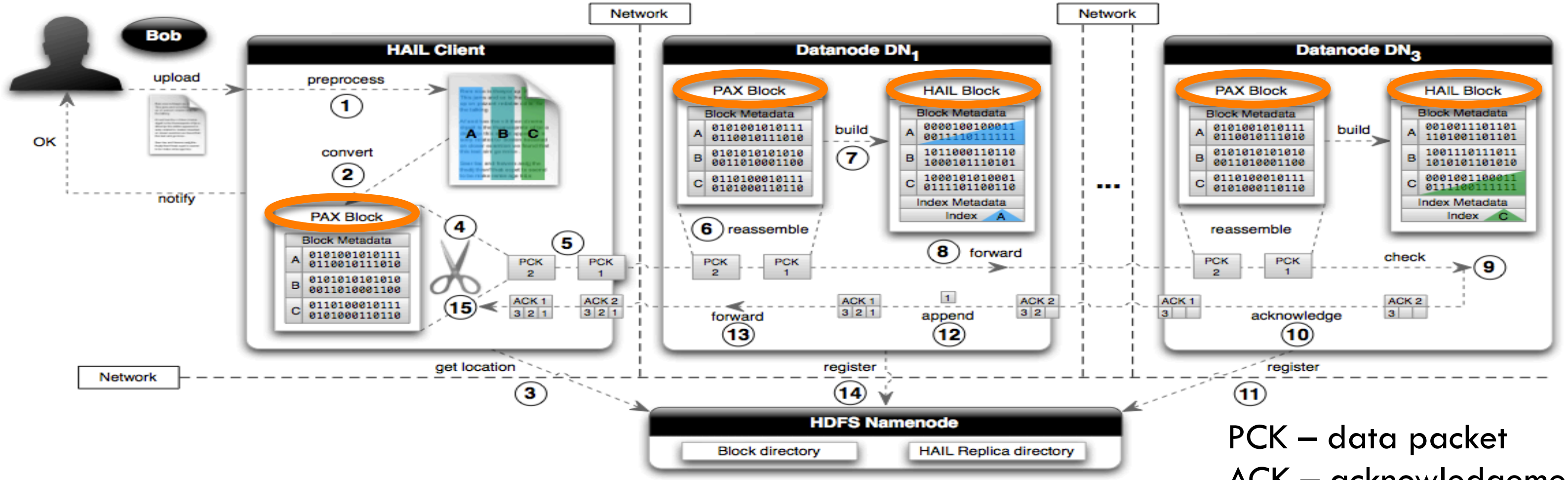


Figure 1: The HAIL upload pipeline

PCK – data packet

ACK – acknowledgement number

HDFS Namenode Extension

- It keeps track of different sort orders

- HAIL needs to schedule map tasks close to replicas having suitable indexes
- Central namenode keeps Dir_Block mapping:

  blockID → Set Of DataNodes.

  and Dir_rep mapping:

  (blockID, datanode) → HAILBlockReplicaInfo.

Indexing Pipeline

- Why clustered indexing?
  - Cheap to create in main memory
  - Cheap to write to disk
  - Cheap to query from disk

- Divides data of attribute sourceIP into partitions
  Consisting of 1024 values
- Child pointers to start offset
- Only the first child pointer is explicit
  - all leaves are contiguous on disk
  - can be reached by simply multiplying the leaf size with the leaf ID.

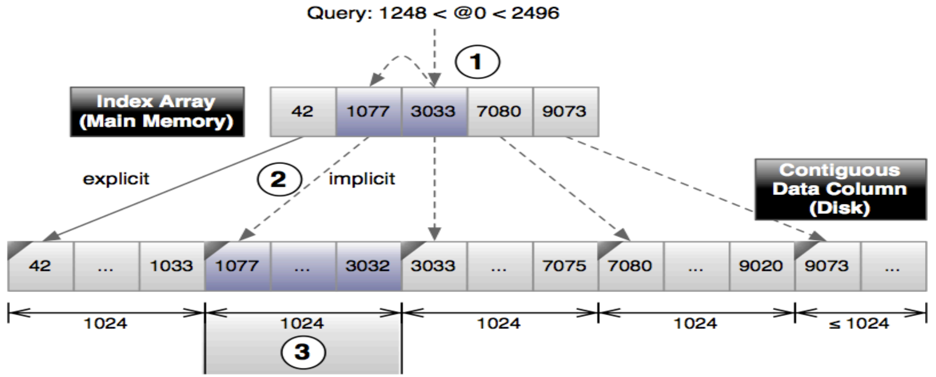


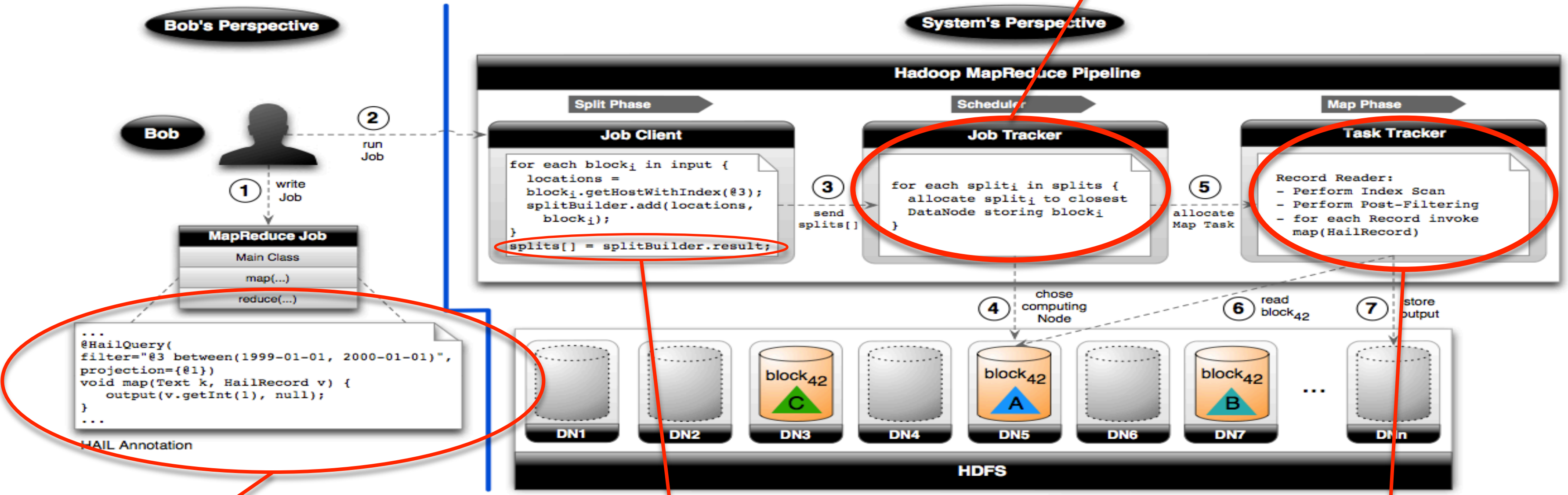
Figure 2: HAIL data column index

Query

- SELECT sourceIP
  FROM UserVisits WHERE visitDate
  BETWEEN ‘1999-01-01’ AND ‘2000-01-01’;

Query Pipeline

For each map task, the JobTracker decides on which computing node to schedule the map task, using the split locations.



```
...
@HailQuery(
filter="@3 between(1999-01-01, 2000-01-01)",
projection={@1})
void map(Text k, HailRecord v) {
   output(v.getInt(1), null);
}
...
```

```
for each block_i in input {
  locations =
  block_i.getHostWithIndex(@3);
  splitBuilder.add(locations,
    block_i);
}
splits[] = splitBuilder.result;
```

```
for each split_i in splits {
  allocate split_i to closest
  DataNode storing block_i
}
```

```
Record Reader:
- Perform Index Scan
- Perform Post-Filtering
- for each Record invoke
  map(HailRecord)
```

Figure 3: The HAIL query pipeline (with default Hadoop scheduling)

Annotates his map function to specify the selection predicate and the projected attributes required by his MapReduce job.

JobClient logically breaks the *input* into smaller pieces called *input splits*. An input split defines the input data of a map task.

The map task uses a *RecordReader* UDF in order to read its input data blocki from the closest datanode.

Query Pipeline – System Perspective

- It is crucial to be non-intrusive to the standard Hadoop execution pipeline so that users run MapReduce jobs exactly as before.
- HailInputFormat
  - a more elaborate splitting policy, called HailSplitting.
- HailRecordReader
  - responsible for retrieving the records that satisfy the selection predicate of MapReduce jobs.

Experiment

- Six different clusters
  - One physical cluster with 10 nodes
  - Three EC2 clusters using different data types each with 10 nodes
  - Two EC2 clusters: one with 50 nodes, the other 100 nodes
- Two datasets:
  - UserVisits table – 20GB data per node
  - Synthetic dataset – 13GB data per node
    - consisting of 19 integer attributes in order to understand the effects of selectivity.

Queries

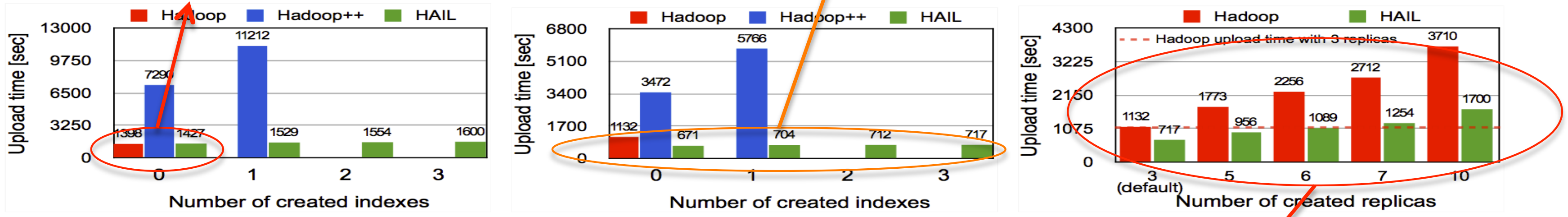
- Bob-Q1 (selectivity: 3.1 x 10—2)
  SELECT sourceIP FROM UserVisits WHERE visitDate
- BETWEEN ‘1999-01-01’ AND ‘2000-01-01’; Bob-Q2 (selectivity: 3.2 x 10—8)
  SELECT searchWord, duration, adRevenue
- FROM UserVisits WHERE sourceIP=‘172.101.11.46’; Bob-Q3 (selectivity: 6 x 10—9)
- SELECT searchWord, duration, adRevenue FROM UserVisits WHERE sourceIP=‘172.101.11.46’ AND visitDate=‘1992-12-22’; Bob-Q4 (selectivity: 1.7 x 10—2)
- SELECT searchWord, duration, adRevenue FROM UserVisits WHERE adRevenue>=1 AND adRevenue<=10; Additionally, we use a variation of query Bob-Q4 to see how well HAIL performs on queries with low selectivities:
- Bob-Q5 (selectivity: 2.04 x 10—1)
- SELECT searchWord, duration, adRevenue FROM UserVisits WHERE adRevenue>=1 AND adRevenue<=100;

Experiment Result (1)

HAIL has a negligible upload overhead of ~2% over standard Hadoop.

HAIL outperforms Hadoop by a factor of 1.6 even when creating three indexes.

------ marks the time Hadoop takes to upload with the default repli- cation factor of three.



(a) Upload time for UserVisits (b) Upload time for Synthetic (c) Varying replication for Synthetic

Figure 4: Upload times when varying the number of created indexes (a)&(b) and the number of data block replicas (c)

When HAIL creates one index per replica the overhead still remains very low (at most ~14%).

HAIL significantly outperforms Hadoop for any replication factor.

Experiment Result (2)

- HAIL achieves roughly the same upload times for the Synthetic dataset.
- HAIL improves its upload times for larger clusters for UserVisits dataset.
- More interesting, we observe that HAIL does not suffer from
  high performance variability.

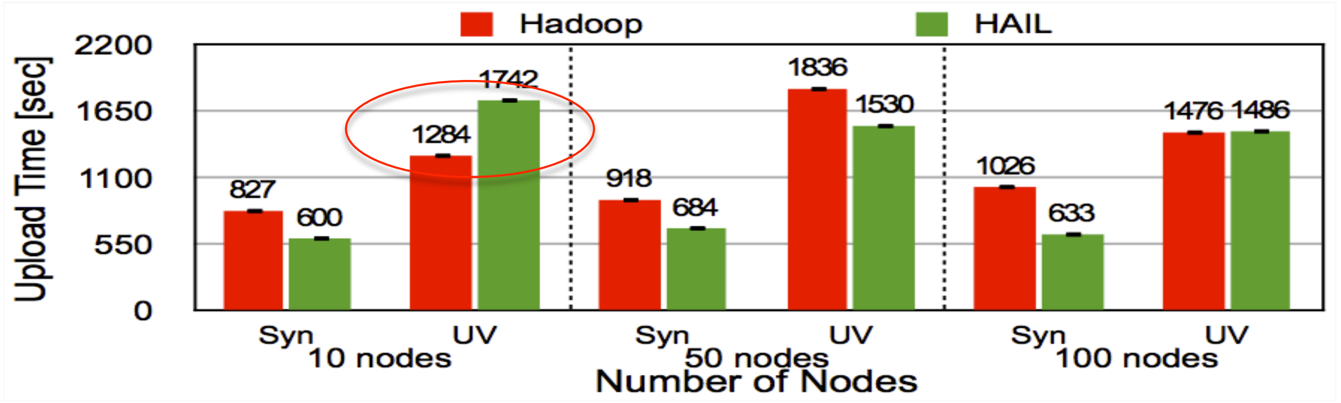
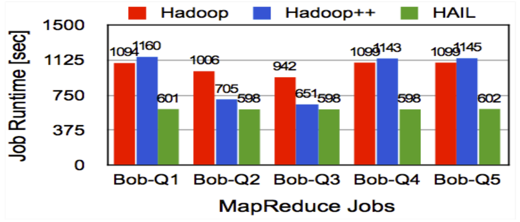


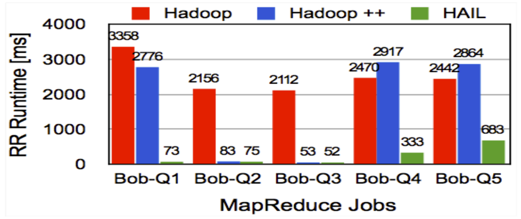
Figure 5: Scale-out results

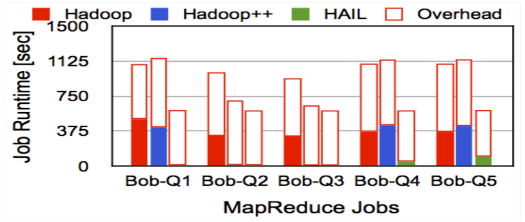
UV: Upload times for UserVisits when scaling-up

Syn: Upload times for Synthetic when scaling-up [sec]

Experiment Result (3)

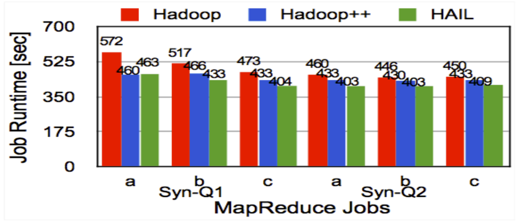


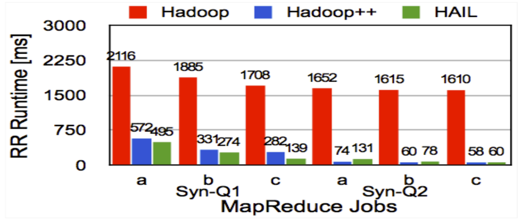


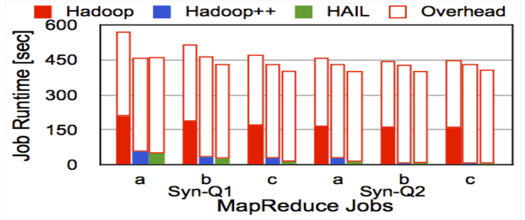


(a) End-to-end job runtimes (b) Average record reader runtimes (c) Hadoop framework overhead

Figure 6: Job runtimes, record reader times, and Hadoop MapReduce framework overhead for Bob's query workload filtering on multiple attributes







(a) End-to-end job runtimes (b) Average record reader runtimes (c) Hadoop framework overhead

Figure 7: Job runtimes, record reader times, and Hadoop MapReduce framework overhead for `Synthetic` query workload filtering on a single attribute

Fault tolerence

- HAIL preserves the failover property of Hadoop by having almost the same slowdown.
- When HAIL creates the same index on all replicas (HAIL-1Idx), HAIL has a lower slowdown since failed map tasks can still perform an index scan even after failure.

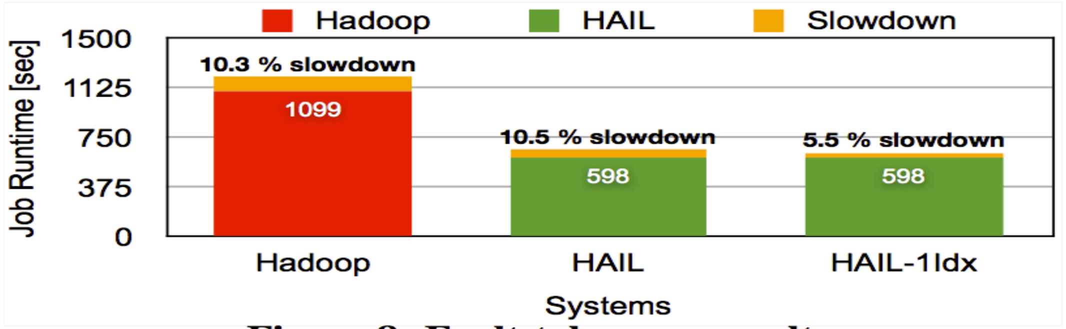


Figure 8: Fault-tolerance results

HAIL: create indexes on three different attributes, one for each replica.
HAIL-1Idx: create an index on the same attribute for all three replicas.

Wrap Up

- HAIL can:
  - Improve both upload and query times
  - Keep failover properties of Hadoop
  - works with existing MapReduce jobs incurring only minimal changes to them

Questions to Ponder

- Why does HAIL have different performance on different data types?
- Would it be possible to use HAIL for the data that is already stored in the cluster?
- Is HAIL open source?
- Can HAIL integrate with other systems, such as Pig or Hive? – API
- Why do/don't the authors use Hadoop++?
- What is it like when using HAIL on other cases?
- How useful it will be for needs of different users/queries?
- ….

Backup Slides

- HAIL vs. Hadoop++
  - HAIL create Trojan indexes per physical replica instead of logical HDFS replica
  - HAIL create indexes less expensive
- Twitter full text indexing
  - Only suitable for highly selective queries
- CoHadoop
  - Did not improve indexing features of Hadoop++

BUILDING WAVELET HISTOGRAMS ON LARGE DATA IN MAPREDUCE

Junjie Hu

Introduction

- Histograms are important for summarizing data
- Wavelet histogram is one of the most widely used
- Straightforward adaption for building wavelet histograms to MapReduce is inefficient
- Require new algorithm

How to build wavelet histograms

- Suppose dataset has a key drawn from domain $u = \{1, 2, \ldots, u\}$
- Frequency vector as $v = (v(1), v(2) \ldots, v(u))$ where $v(x)$ is the number of occurrences of key x in the data sets
- Calculate wavelet basis vector ψ, same length as v. ψ is unrelated to v
- Coefficients are $w(i) = \langle v, \psi(i) \rangle$ (dot products), $i = 1, \ldots, u$
- Computer the best k-term wavelet representations using centralized algorithm

Baseline solution (Send-V)

- m mappers(node) and single reducer(coordinator)
- Mapper: each mapper emits $(x, v_{local}(x))$ for all x in the splits and its local frequency
- Reducer: aggregate (x, v(x)).
  Calculate $w(i) = <v, \psi(i)>$ then select best k-terms.

Alternative baseline (Send-Coefficient)

- Due to the Distributive Law, we can calculate each local coefficient on mapper, and let reducer aggregate those results

$$w_i = \left\langle \sum_{j=1}^{m} \mathbf{v}_j, \psi_i \right\rangle = \sum_{j=1}^{m} \langle \mathbf{v}_j, \psi_i \rangle,$$

Discussion?

- Drawback?
- Any improvement? (note that coefficient could be negative)
- How many intermediate files ? Suppose the splits number is m, domain size is u.

Hadoop Wavelet top-k(H-WTopk)

- For an item(key) x, $r(x)$ denotes its aggregated score(coefficient) and $r_j(x)$ is its score at node j.
- An lower bound $\tau(x)$ of item x score's magnitude. $\tau(x) \leq |r(x)|$
- A global threshold τ, kth largest $\tau(x)$.
- If an item's local score is always below τ/m, then it can be discarded.

Three rounds for H-WTopk: Round 1

- Each node emits k highest and lowest score. If coordinator receives x' score from a node, update its upper bound $\tau^+(x)$ and lower bound $\tau^-(x)$ with addition of $r_{local}(x)$. Otherwise, add the kth highest score of this node sends to $\tau^+(x)$, and the kth lowest to $\tau^-(x)$.
- Set $\tau(x) = 0$ if $\tau^+(x)$ and $\tau^-(x)$ have different signs. Otherwise, $\tau(x) = \min(|\tau^+(x)|, |\tau^-(x)|)$
- Pick the k-th largest $\tau(x)$, denotes as T_1. It's a threshold for the magnitude of the top-k items.

Three rounds for H-WTopk: Round 2

- For each node j, emits item x if $|r_j(x)| > T_1/m$.
- Define R as the set of items coordinator received. Refine $\tau^+(x)$ and $\tau^-(x)$ for every x from R. If a node has not been received from a node, use T_1/m and $-T_1/m$ to update upper/lower bound.
- Calculate a better threshold T_2. For any x from R, compute $\tau(x) = \max(|\tau^+(x)|, |\tau^-(x)|)$. Delete x from R if $\tau(x) < T_2$.

Three rounds for H-WTopk: Round 3

- Ask each node for the scores of all items in R.
- Computer the aggregated scores exactly for those items, from which we pick k items of largest magnitude.

H-WTop

- Use lower/upper bound to estimates the score the item, and calculate a threshold to prune items
- Cost on communication is better than baseline solution.
- Need 3 rounds.
- Drawback?

Drawback for H-WTopk

- Three rounds of MapReduce jobs incurs a lot of overhead
- On node j, every split needs to be fully scanned to compute local frequency vector v_j and compute local wavelet coefficient $w_{i,j}$, $i = 1, \ldots, u$
- Any solution?

Sampling

- Assume n is #records in dataset, if we want to approximate each frequency v(x) with a standard deviation of εn, a sample of size $\Theta(1/\varepsilon^2)$ is required. A sample probability $p = 1/(\varepsilon^2 n)$.
- If n is very large, we need a very small ε to keep accuracy. For $\varepsilon = 10^{-6}$, even with one-byte key, still needs to emit 1TB data.
- Cost of communication is $O(1/\varepsilon^2)$

Two-level sampling (TwoLevel-S)

- For each split j, extract a sample from input. Calculate $(x, s_j(x))$ from sample. $s_j(x)$ is the counts of x.
- Perform a second-level sample, for any item x with $s_j(x) \geq (1 / \varepsilon \sqrt{m})$, emit the $(x, s_j(x))$, otherwise, sample it with a probability proportional to $s_j(x)$, i.e. $(\varepsilon\sqrt{m} \times s_j(x))$, and emit the pair$(x, NULL)$.

TwoLevel-S

- Communication cost reduced to $O(\sqrt{m}/\varepsilon)$.
- It provides an unbiased estimation of coefficient w. (see proof in paper)
- Both H-WTopk and TwoLevel-S work well in practice(see experiment results in paper)

Wrap-up

- Provide two approaches to calculate coefficients used in wavelet histograms: one exact computation approach and one approximate computation approach.
- Design algorithms for MapReduce jobs, one should consider the cost of communication (i.e., number of intermediate results). It's one of the factors that influences the efficiency of algorithm. (For one who has taken cs425 last semester and worked on mp4 may have a good understanding for this)

- Thanks for watching.