Vector H:
Taking SQL-on-Hadoop to the Next Level

Xiaotong Niu, Xiaoyan Ge

Authors

Andrei Costea[‡] Adrian Ionescu[‡] Bogdan Răducanu[‡] Michał Świtakowski[‡] Cristian Bârcă[‡]

Juliusz Sompolski[‡] Alicja Łuszczak[‡] Michał Szafrański[‡]

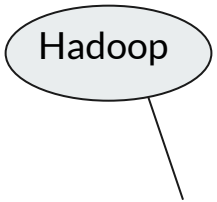
Giel de Nijs[‡]
Actian Corp.[‡]

Peter Boncz[§]
CWI[§]

Agenda

- Introduction
  - Hadoop
  - SQL on Hadoop
  - Vectorwise
- VectorH: a new **SQL-on-Hadoop** system on top of the fast **Vectorwise** analytical database system
  - Features
  - Integration
  - Transactions in Hadoop
  - Connectivity with Spark
  - Evaluation
- Related Work
- Conclusion & Future Works

Introduction



SQL-on-Hadoop system on top of the fast **Vectorwise** analytical database system

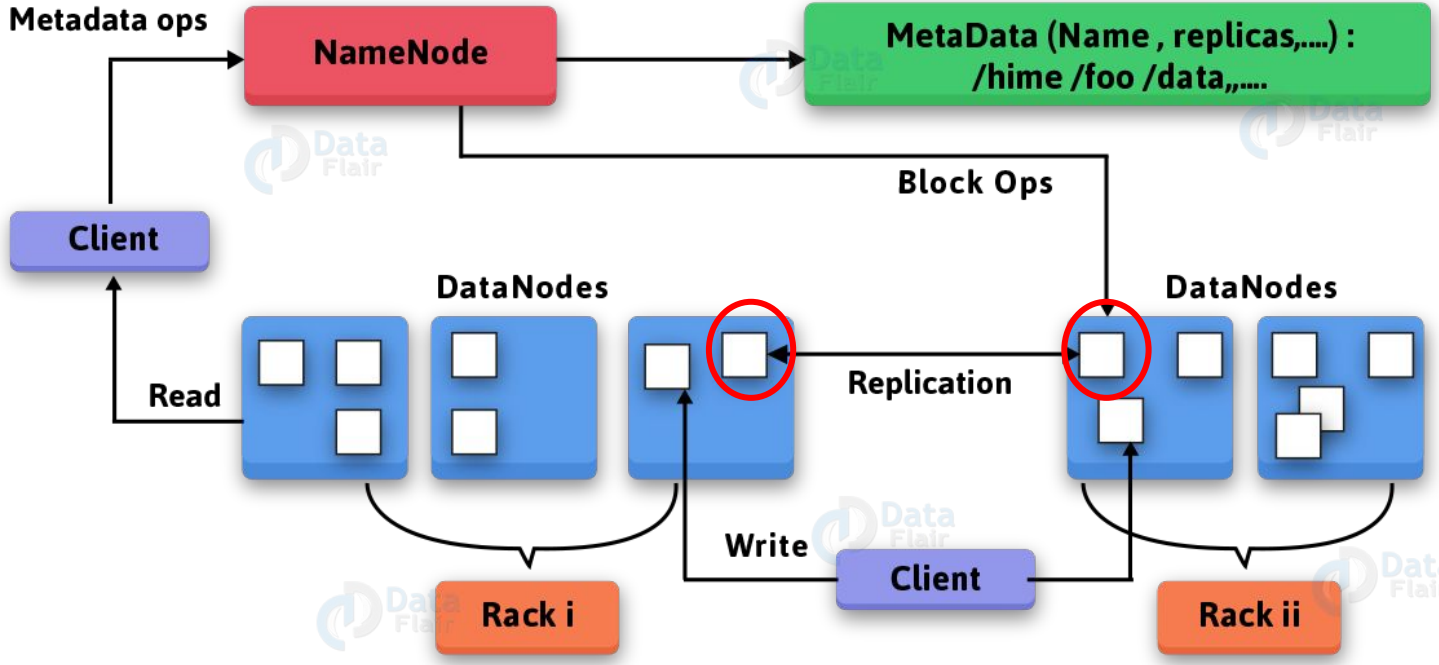
What is Hadoop?

- Collect data ----> Analyze Data
- Mainly for Big Data Clusters
- MapReduce, YN, and HDFS

Hadoop - HDFS

- Provide Rapid Data Access across the node
- Fault tolerance
- An Append-only file System

HDFS Architecture

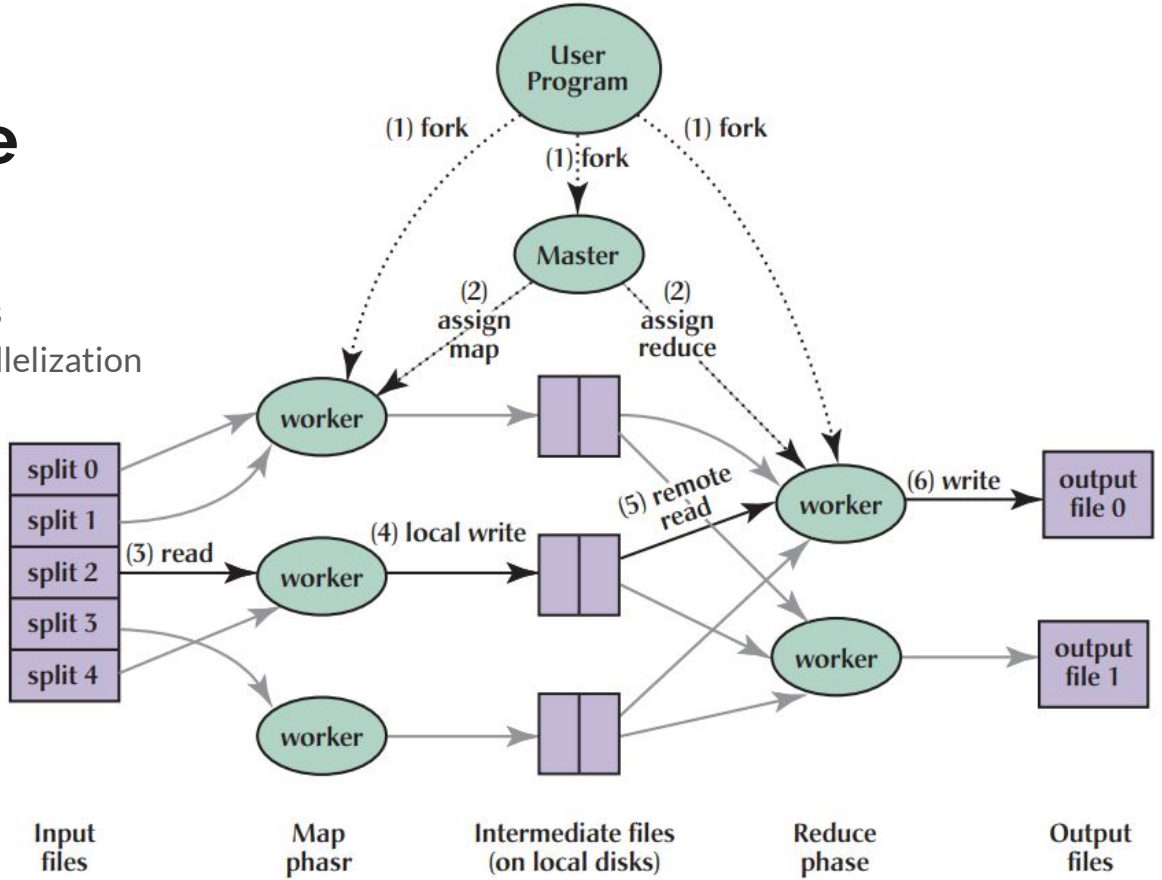


Hadoop - YARN

- Yet Another Resource Negotiator
- Resource Manager

Hadoop Map/Reduce

- On large cluster of PCs
- Enable automatic parallelization
- Map and Reduce



MapReduce Example - Word Count

"Aaron likes to drink, sleep and play games.

Aaron does not like to do presentation. "

File A

Map Worker 1:"Aaron, 1" , "likes, 1" , "to ,1" … etc.

Map Worker 2: "Aaron, 1" , "does, 1", "like, 1", "to, 1"

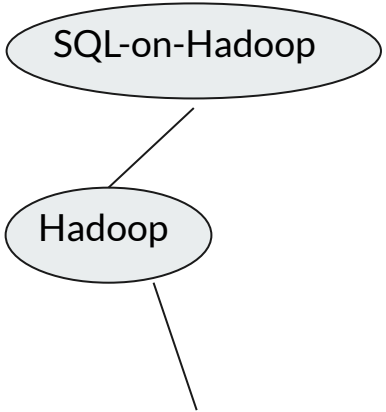
Reduce Worker 1: "Aaron,2", "to, 2", "does, 1" etc

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Question to Tackle

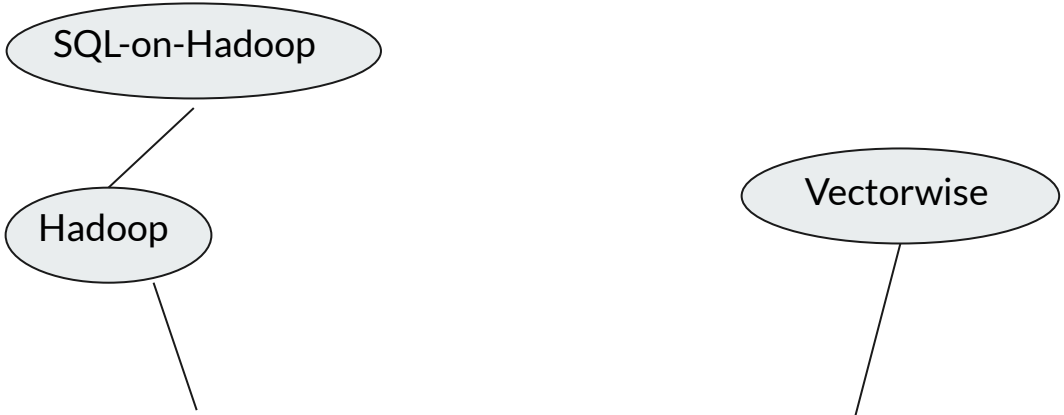
What if I want to use SQL on the Hadoop?



SQL-on-Hadoop system on top of the fast **Vectorwise** analytical database system

SQL on Hadoop

- Win-Win cooperation!
- Examples: Hadoop Hive, Impala



SQL-on-Hadoop system on top of the fast **Vectorwise** analytical database system

Vectorwise

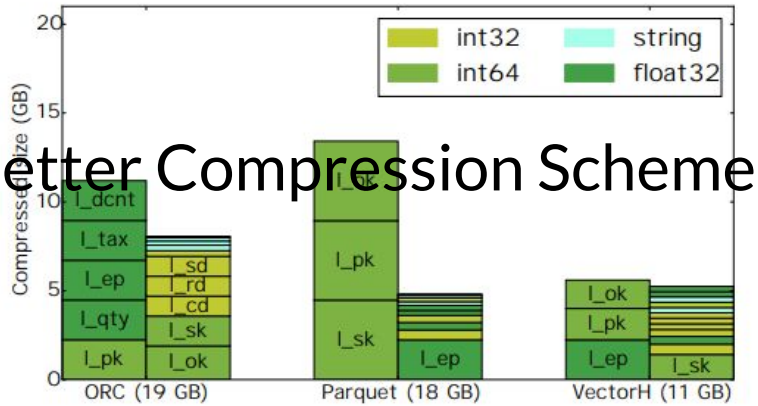
- Query Engine that operates on vectors of values, rather than tuple
- Advantages:
  - Reducing query interpretation overhead
  - Increase Locality
  - Allow SIMD instructions
  - Compression method: PDICT, PFOR, and PFOR-delta
  - **MinMax indexes**

Vectorh performs better!

SELECT max(l_linenumber)
FROM lineitem WHERE l_shipdate<X

Vectorh finished faster!

Vectorh touched less data!



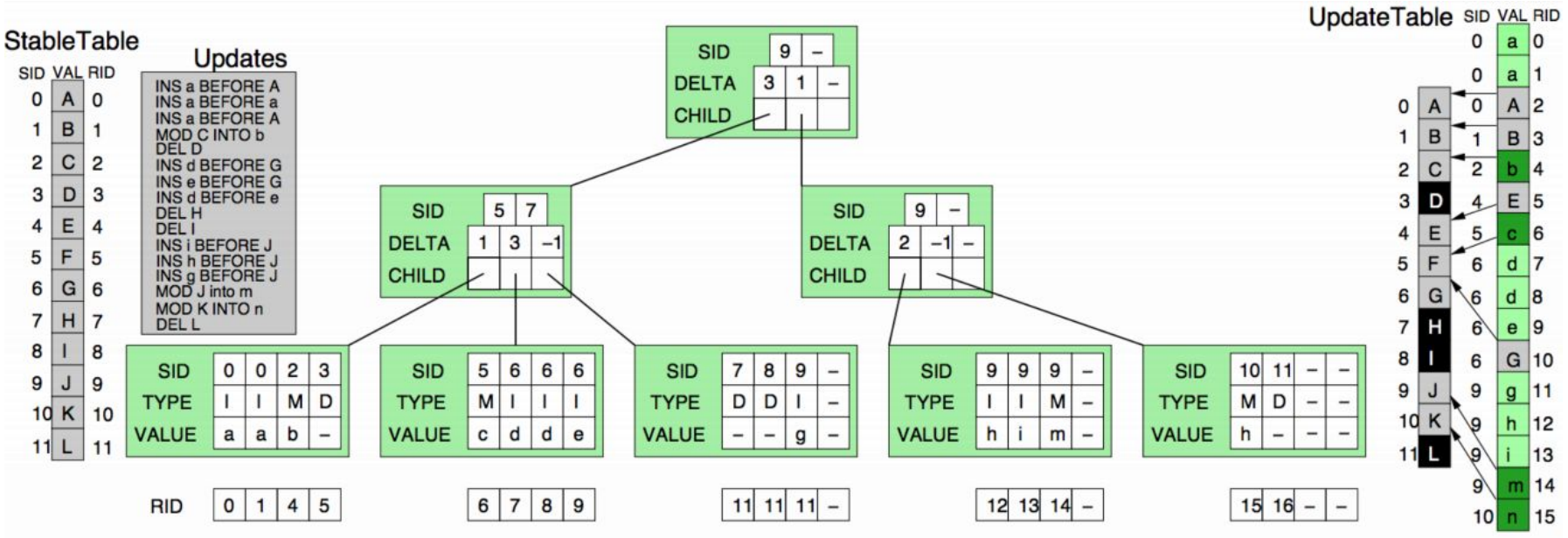
Better Compression Scheme!

Vectorwise Physical Design Option

- Stored unordered or called clustered index
- Benefit when scan for range queries on index keys
- **Drawback**: Co-ordered Table Layout -- **difficult to insert/delete**
    - Resolve by Positional Delta Trees

Positional Delta Trees

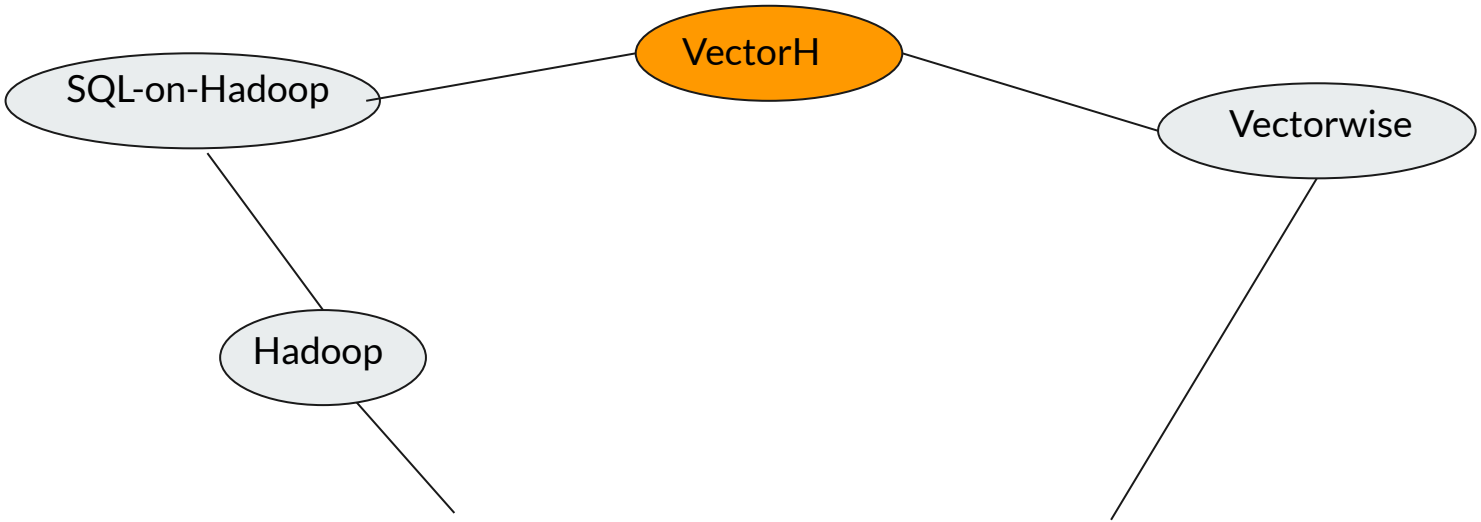
- Static table(old) and SID, B+ tree, Update Table(new) and RID
- Identifies tuples by position, rather than primary key
- Store old position - SID and current position - RID
- Goal - Fasting merging by differential Update
- In logarithmic complexity


StableTable
SID VAL RID
0 A 0
1 B 1
2 C 2
3 D 3
4 E 4
5 F 5
6 G 6
7 H 7
8 I 8
9 J 9
10 K 10
11 L 11
Updates
INS a BEFORE A
INS a BEFORE a
INS a BEFORE A
MOD C INTO b
DEL D
INS d BEFORE G
INS e BEFORE G
INS d BEFORE e
DEL H
DEL I
INS i BEFORE J
INS h BEFORE J
INS g BEFORE J
MOD J into m
MOD K INTO n
DEL L
SID 9 –
DELTA 3 1 –
CHILD
SID 5 7
DELTA 1 3 –1
CHILD
SID 9 –
DELTA 2 –1 –
CHILD
SID 0 0 2 3
TYPE I I M D
VALUE a a b –
SID 5 6 6 6
TYPE M I I I
VALUE c d d e
SID 7 8 9 –
TYPE D D I –
VALUE – – g –
SID 9 9 9 –
TYPE I I M –
VALUE h i m –
SID 10 11 – –
TYPE M D – –
VALUE h – – –
RID 0 1 4 5
6 7 8 9
11 11 11 –
12 13 14 –
15 16 – –
UpdateTable
SID VAL RID
0 a 0
0 a 1
0 A 2
1 B 3
2 b 4
4 E 5
5 c 6
6 d 7
6 d 8
6 e 9
6 G 10
9 g 11
9 h 12
9 i 13
9 m 14
10 n 15

Question to Tackle Cont.

Its good and all, but I want it to be faster!

What should I do?



SQL-on-Hadoop system on top of the fast **Vectorwise**

From Vectorwise to VectorH

- Took all the beautiful features that was mentioned before
- Take it to next level - YARN-based cluster Vectorwise with HDFS as storage(fault tolerance)
- **VectorH Integrating HDFS and YARN**

VectorH

What is VectorH?

- Short for **Actian Vector in Hadoop**
- **SQL-on-Hadoop** system on top of **Vectorwise**
  - Query execution
  - ELASTICITY -- YARN
  - Local I/O -- HDFS
  - Updatability -PDT
  - Spark integration

STORAGE AFFINITY WITH HDFS

Original Approach: Store in a fixed compressed size and write in consecutive blocks

One file stores: 1 column, of 1 partition, of 1 copy

Problem:

1. **Unable to reuse block**
2. **Opening too many files**

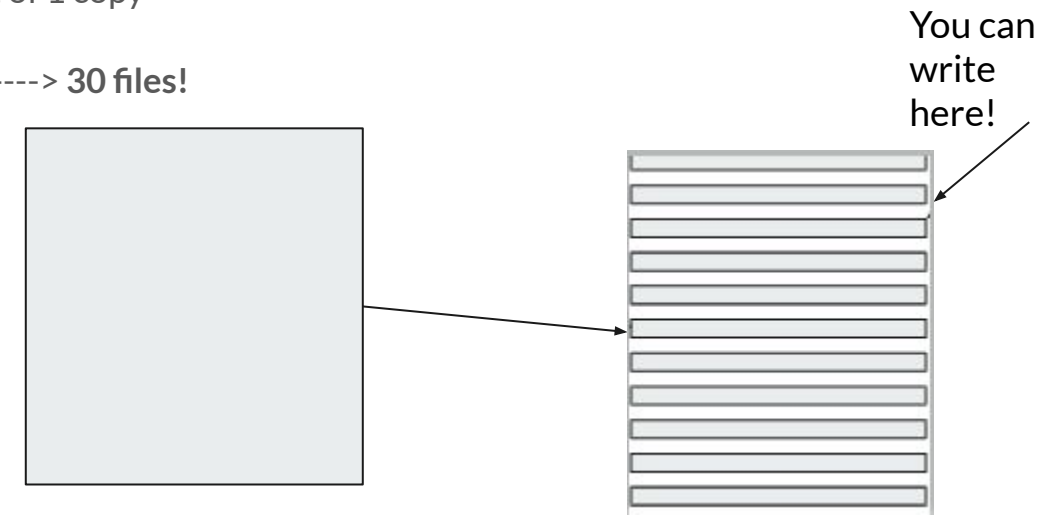
3 X 100 X 10 = 3000 files!

File-per-Partition Layout

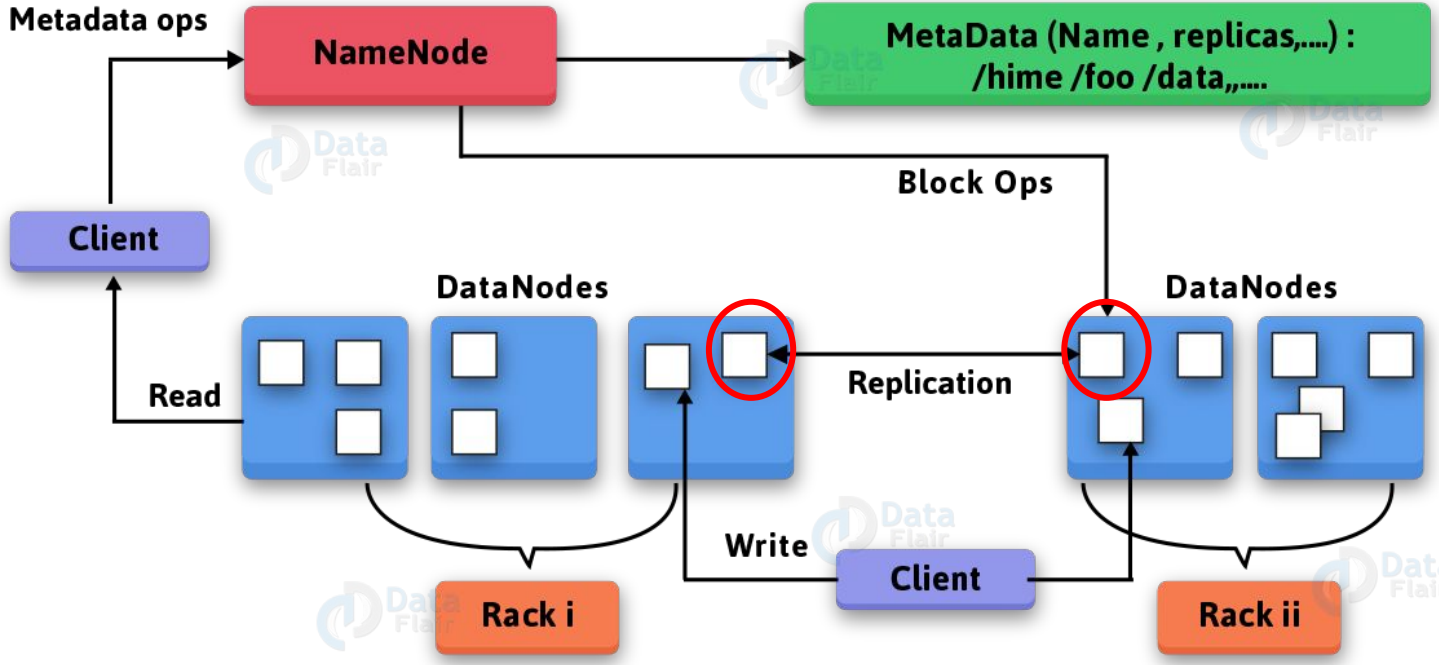
One file stores: all the column, of 1 partition, of 1 copy

From 3000 ----> **30 files!**

Split data files horizontally



HDFS Architecture



Instrumenting HDFS Replication

When - When client initiates a file append, and when you need to re-balance/re-replicate copies

Where: Force it to be local

How: **BlockPlacementPolicy** Class and **chooseTarget()** method

A plus one: Able to join tables with same keys and same amount of partition

STORAGE AFFINITY WITH HDFS

node1			node2			node3			node4		
R01	**R02**	**R03**	**R04**	**R05**	**R06**	**R07**	**R08**	**R09**	**R10**	**R11**	**R12**
S01	**S02**	**S03**	**S04**	**S05**	**S06**	**S07**	**S08**	**S09**	**S10**	**S11**	**S12**
R10a	R11a	R12a	R01a	R02a	R03a	R04a	R05a	R06a	R07a	R08a	R09a
S10a	S11a	S12a	S01a	S02a	S03a	S04a	S05a	S06a	S07a	S08a	S09a
R07b	R08b	R09b	R10b	R11b	R12b	R01b	R02b	R03b	R04b	R05b	R06b
S07b	S08b	S09b	S10b	S11b	S12b	S01b	S02b	S03b	S04b	S05b	S06b

after node4 failure:

Figure 2: Partition Affinity Mapping for the 12 partitions of table R,S before (top) & after (bottom) node4 failure. Responsible partitions in bold; a/b are the second/third copy (R=3).

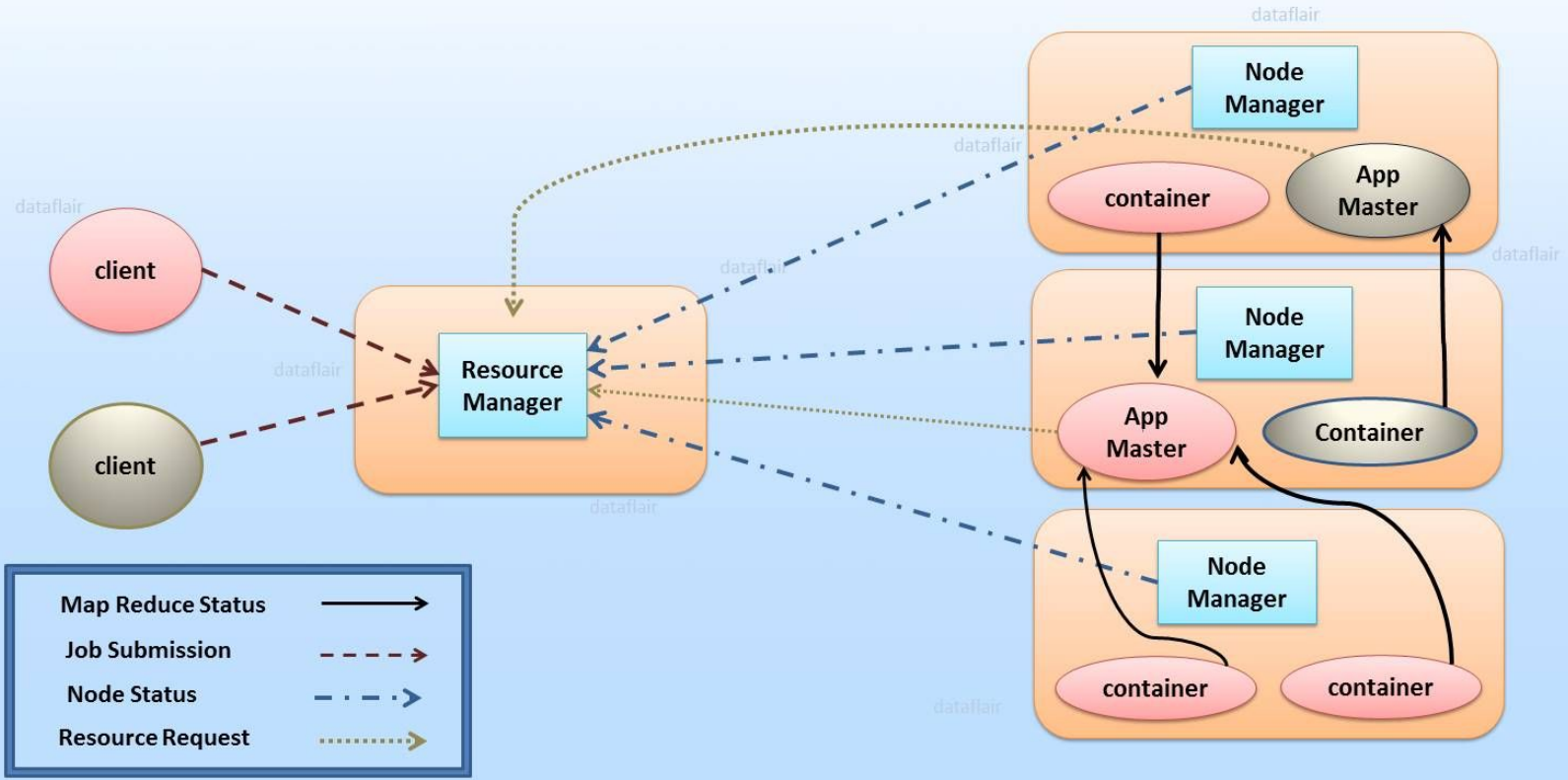
YARN

- Workers and Master
- Out-of-band YARN
- Min-cost Flow Network Algorithms

Workers and Masters

- Worker: Nodes in clusters that run the Vectorwise process
- Master: One of the Worker, can be interchanged, responsible for parallel query optimization

Apache Hadoop YARN-Architecture



YARN - Out of Band

Why? - cannot run in the container, and also we are not able to modify container's resource, Do not want to Stop and restart every time.

How?- out of band, separate the process from container

Min-cost Flow Network Algorithms

When? When it first start, or node failed and need recover

Why? Ensure most resources and most locality

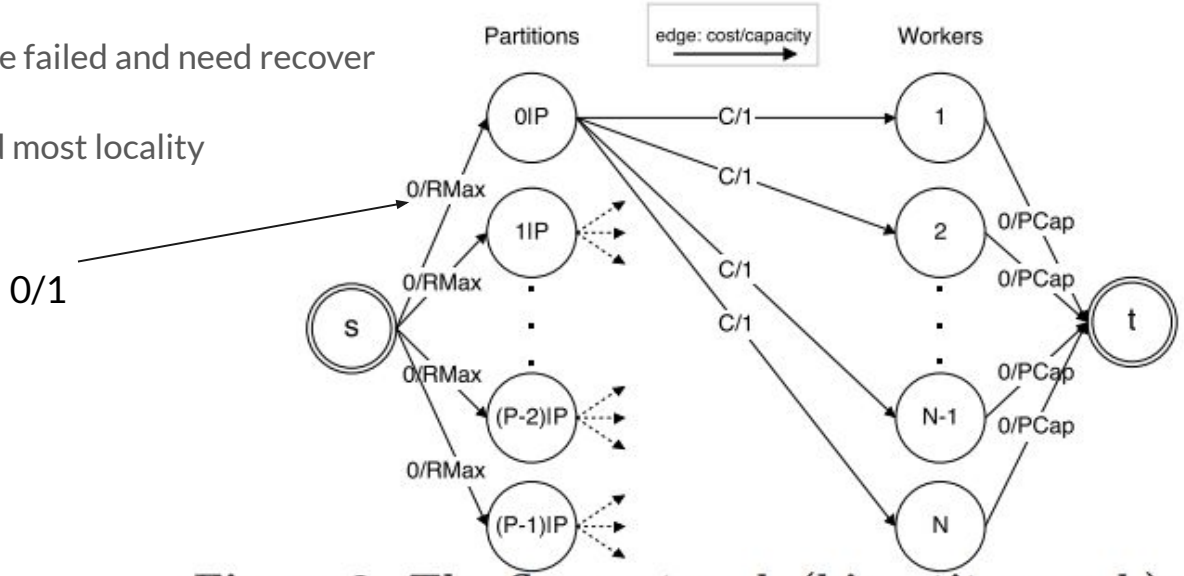


Figure 3: The flow network (bipartite graph) model used to determine the responsibility assignment.

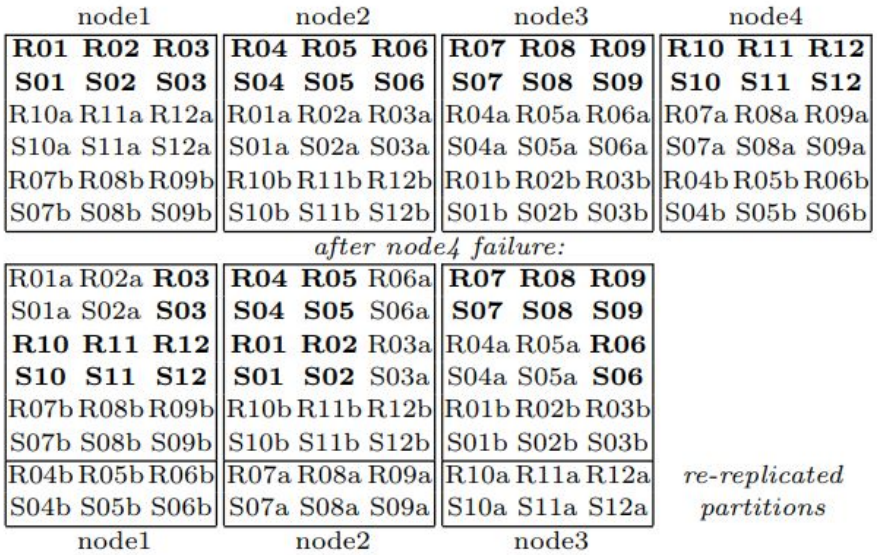


Figure 2: Partition Affinity Mapping for the 12 partitions of table R,S before (top) & after (bottom) node4 failure. Responsible partitions in bold; a/b are the second/third copy (R=3).

PARALLELISM WITH MPI

- Implementation of Xchg operators - achieve parallelism
  - Only redistributing streams, not altering
  - Encapsulate parallelism
- Serve as Synchronization point as between producer and consumer.
- Examples: **XchgHashSplit, XChgUnion**

Distributed Exchange

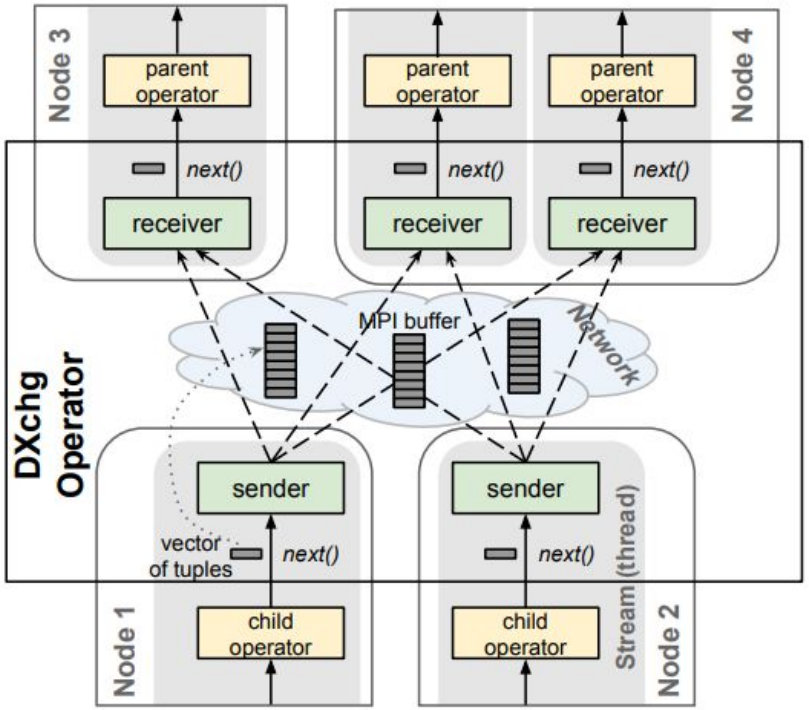


Figure 4: DXchg Operator Implementation

- Use the MPI(Message Passing Interface) for high performance and well-defined point to point communication
- Producers is sending data to all consumers with **double buffering**
- Originally using **thread-to-thread,** but it cost too much.
- Now using **thread-to-node, with a column specify which receiver**

#fanout: num_nodes * num_cores
#memory: 2* num_nodes*num_cores^2

```
SELECT FIRST 10 s_suppkey, s_name, count(*) as l_count
FROM   lineitem, orders, supplier
WHERE l_orderkey=o_orderkey AND l_suppkey=s_suppkey AND
      l_discount>0.03 AND
      o_orderdate BETWEEN '1995-03-05' AND '1997-03-05'
GROUP BY s_suppkey, s_name
ORDER BY l_count
```

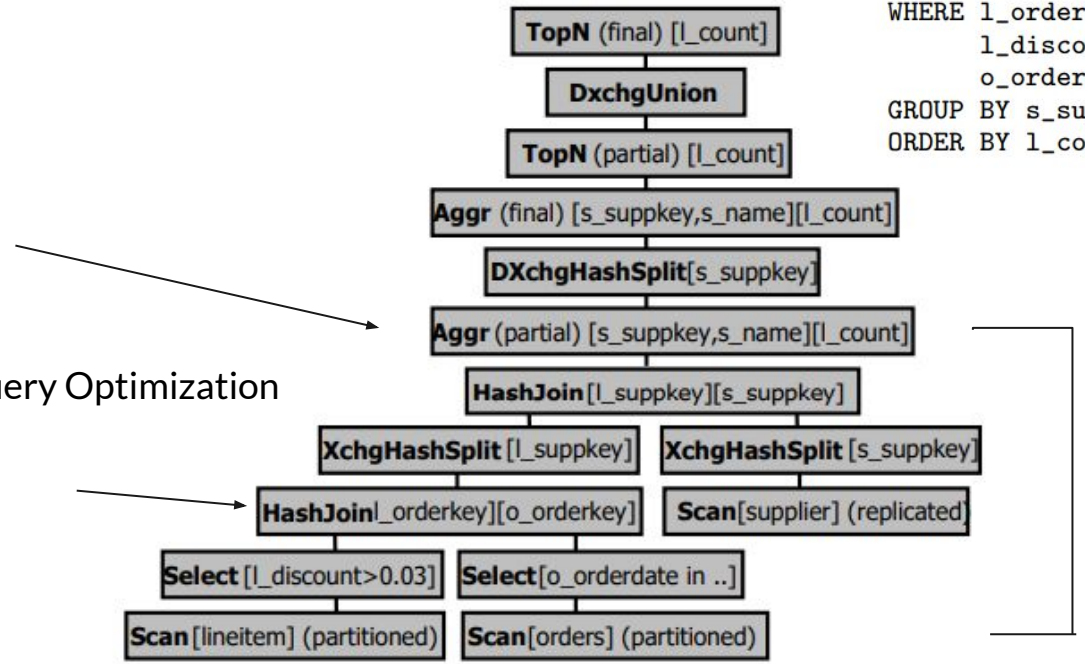


Figure 5: Example Distributed Query Plan

Query Optimization

5.02s -> 26.14s

Transactions in Hadoop

- Vectorwise transaction management
  - Differences stored in Positional Delta Trees(PDTs)
  - Stackable
    - Trans-PDT -> Read-PDT -> Write-PDT
  - Snapshot Isolation
  - Serialized Trans-PDT
    - Written into Write Ahead Log(WAL)
    - for persistence

Transactions in Hadoop (cont.)

- Distributed Transactions in VectorH
  - Table partition-specific WALs
    - Update table partition at responsible nodes
    - Modified PDT / HDFS
  - 2 Phase Commit(2PC)
    - ACID

Transactions in Hadoop (cont.)

- Log Shipping
  - Broadcast changes
  - ✅ reuse

Transactions in Hadoop (cont.)

- Update Propagation
  - Flushing PDTs to the compressed column store
  - PDT sizes, tuple fractions
    - Better performance
    - tail inserts vs. other updates

Transactions in Hadoop (cont.)

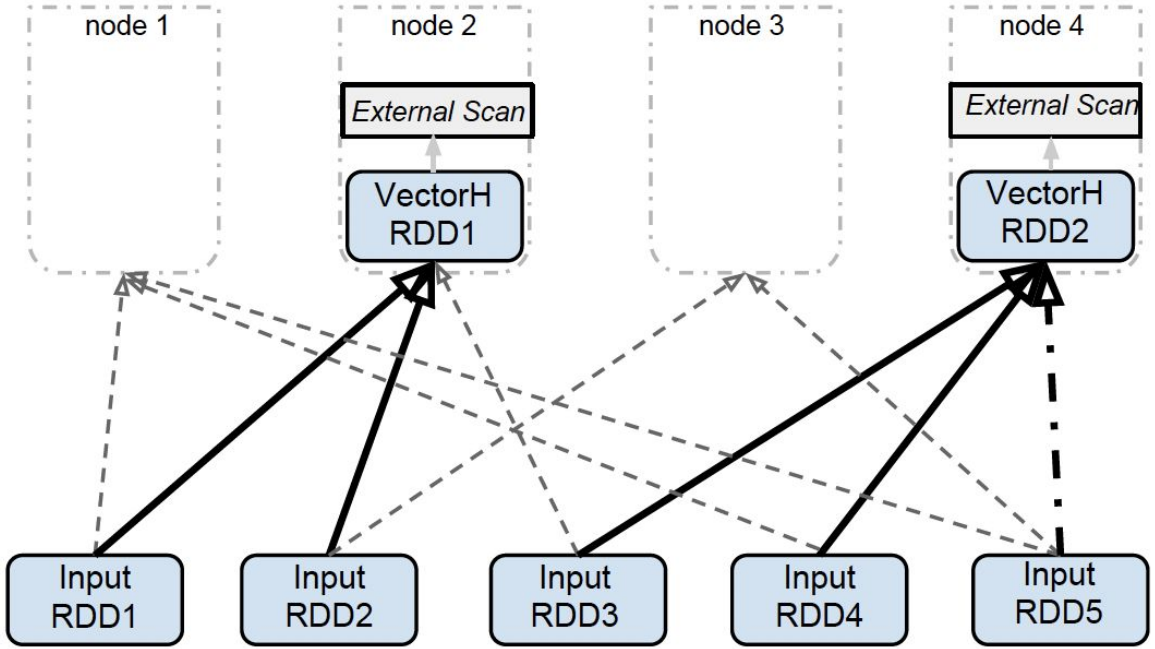
- Referential Integrity
  - Key uniqueness
  - Node-local verification

Transactions in Hadoop (cont.)

- MinMax Indexes
  - Divide tables -> keep Min Max
  - Stored in WAL
  - Prevent data accesses

Connectivity with Spark

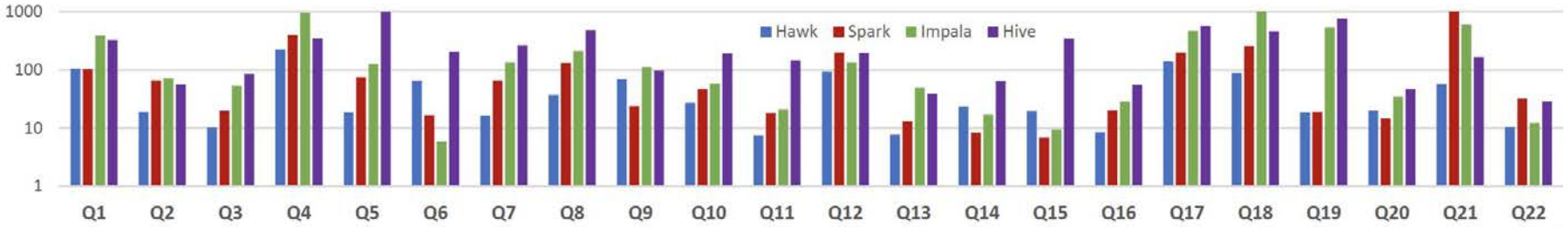
- vwload
  - Load data from HDFS
- Spark-VectorH Connector
  - SparkSQL
  - ExternalScan, External Dump
  - VectorH RDD extends Spark's RDD
    - getPreferredLocations()
  - NarrowDependency


node 1
node 2
node 3
node 4
External Scan
External Scan
VectorH
RDD1
VectorH
RDD2
Input
RDD1
Input
RDD2
Input
RDD3
Input
RDD4
Input
RDD5

Evaluation in TPC-H SF1000

- Setup
  - 10 nodes with Hadoop 2.6.0
    - 1 node runs Hadoop namenode
    - 9 nodes for SQL-on-Hadoop experiments
- VectorH vs. Impala
  Apache Hive
  HAWQ
  SparkSQL

HAWQ



VectorH	**1.5**	**1.14**	**3.16**	**0.17**	**1.94**	**0.31**	**2.75**	**1.31**	**11.11**	**1.21**	**1.69**	**0.34**	**3.66**	**0.83**	**1.63**	**1.68**	**1.24**	**0.99**	**1.32**	**2.15**	**1.48**	**2.84**
HAWQ	158.2	21.46	32.06	38.21	36.38	20.19	44.74	48.38	766.4	32.97	12.48	31.75	27.97	19.47	31.58	14.17	173.2	87.08	24.82	42.84	84.7	29.44
SparkSQL	155.4	74.98	62.38	68.27	146.5	5.1	180.2	174.6	264.0	56.62	30.28	66.97	47.65	6.92	11.16	33.81	244.9	254.7	24.89	31.56	1614	91.18
Impala	585.4	81.81	167.7	163.18	242.5	1.81	369.0	276.2	1242.9	69.97	35.04	45.67	180.8	13.95	15.19	47.52	581.53	1234	714.7	74.25	880.8	34.81
Hive	490.1	63.57	266.6	59.08	DNF	63.63	721.8	625.6	1077	230.5	246.1	65.78	140.7	53.23	556.5	92.51	711.7	454.5	1010	100.5	247.7	81.11
after executing updates:																						
VectorH	1.68	0.94	3.21	0.23	1.9	0.27	2.74	1.4	11.62	1.21	1.44	0.37	3.9	0.81	1.57	1.64	1.27	0.95	1.5	2.25	1.78	2.82
Hive	608.4	80.8	335.7	205.4	DNF	128.0	690.7	719.8	1150	334.4	218.7	170.5	143.8	130.7	596.7	101.4	891.2	594.6	1167	153.3	275.6	67.85

Hive: RF1=34s RF2=112s **GeoDiff=138.2%** - VectorH: RF1=17.8s RF2=8.4s **GeoDiff=102.8%**

Figure 7: Table: TPC-H SF1000 results (seconds). Chart: How many times faster is VectorH?

Related Work

- Previous Works
  - High-performance vectorized query engine
  - Parallel query optimization using Xchg operators
  - VectorwiseMPP project
  - Creating elastic DBMS in YARN
- ORC and Parquet
- HAWQ
- Impala
- Previous Version of VectorH

Conclusion & Future Works

Conclusion

- Mature SQL support
- 1-3 orders of magnitude faster

References

Slide 6: https://data-flair.training/blogs/how-hadoop-works-internally/
Slide 8: https://data-flair.training/blogs/hadoop-hdfs-architecture/
Slide 10: “MapReduce: Simplified Data Processing on Large Clusters” by Jeffrey Dean and Sanjay Ghemawat
Slide 17, 35, 36,38, 39, 47, 49: “VectorH: Taking SQL-on-Hadoop to the Next Level” by Andrei Costea
Slide 20: “Positional Delta Trees to reconcile updates with read-optimized data storage”by S´andor H´eman, Niels Nes
Slide 33: https://data-flair.training/blogs/hadoop-yarn-tutorial/

Future Works

- Integration
  - Spark-VectorH connector