Apache Flink 极客挑战赛

Flink TPC-DS性能优化

参赛队 ：至尊宝

Apache Flink　Alibaba Cloud　intel

Intro

- **金明剑 (Jin Mingjian)**
- **Manager of Data Department at Tigerjoys, now**
- **Pioneer of Chinese Java High-Performance Engineering**
  - **Landz** (high performance Java 8 foundation) *, 2014
    - 30% faster than Netty in TechemPower Benchmark in that time
  - Data engineering, 2014 - now
- **Flink Contributor, 2017**
  - Two trivial PRs merged (Flink-5692, Flink-4422)
  - Why I participate
  - Final battle (挂靴之战)

** https://jinmingjian.xyz/archives/landz/home.html*

Principle

- Solve (or track) the bottleneck in Flink's foundation
  - Contributed as highest engineering standard as possible
- Non goal: make the score ranking No.1
  - violation: merge pipeline breakers into one node (skip blocking mode)
    - e.g. pushdown join op into scan op
  - violation: immediate cache and indexing (skip blocking mode)
    - e.g. cache date_dim for no scanning
    - scans beat btrees when selectivity > 1% (Kester et al., 2017)
  - pure re-ordering in plan opt
    - Calcite works primarily good
    - *"Join reordering is not enabled (by default)... Reordering joins without somewhat accurate es timates is basically gambling"* - Fabian Hueske (Co-Founder DataArtisans)

TPC-DS Benchmark

** TeraData: Can We Trust Hadoop Benchmarks?(*https://www.teradata.com/Blogs/Can-We-Trust-Hadoop-Benchmarks*)

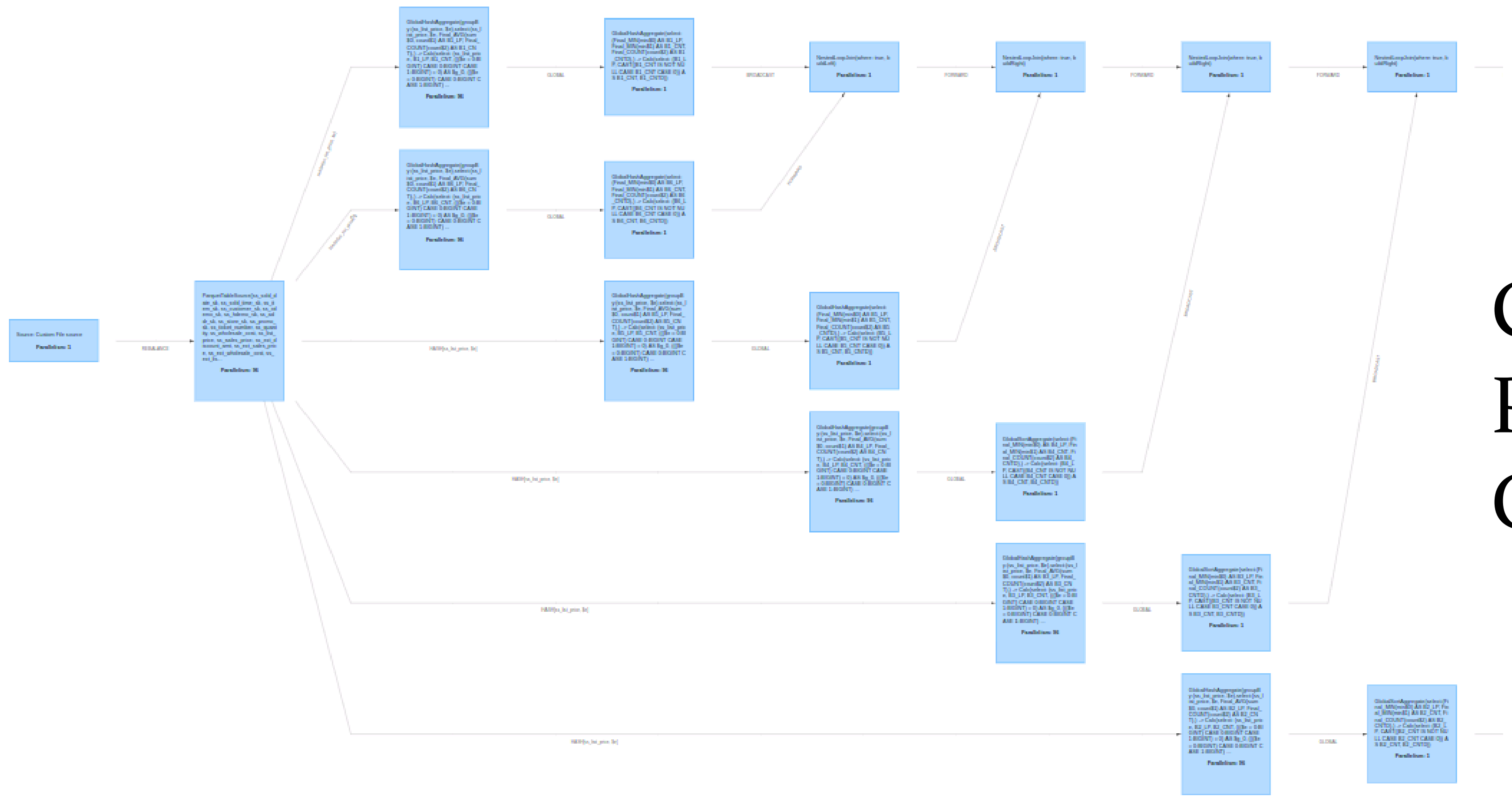
- **Far away from true practices for real world bigdata**
  - Denormalization is common knowledge
- **Criticisms****
  - "Manual optimizer"
    - Traverse plan spaces of TPCDS, then adjust "order" of plan to best score
  - "Amnesia cache" (mentioned above)
  - "Cheater tweaks"
    - generally-wrong special assumption
    - e.g. sorted/unqiue/primary key(there is no sorted/unqiue/primary key idea in parquet)
- **Shame to have this in our competition**

Query93

```
select  ss_customer_sk
            ,sum(act_sales) sumsales
      from (select ss_item_sk
                  ,ss_ticket_number
                  ,ss_customer_sk
                  ,case when sr_return_quantity is not null then (ss_quantity-sr_return_quantity)*ss_sales_price
                                                            else (ss_quantity*ss_sales_price) end act_sales
            from store_sales left outer join store_returns on (sr_item_sk = ss_item_sk
                                                               and sr_ticket_number = ss_ticket_number)
                ,reason
            where sr_reason_sk = r_reason_sk
              and r_reason_desc = 'Package was damaged') t
      group by ss_customer_sk
      order by sumsales, ss_customer_sk
limit 100
```

No clever plan

Query28



Current Plan are Good

Name	Records Sent	Parallelism	Start Time	Duration
ParquetTableSource(ss_sold_date_sk, ss_sold_time_sk, ss_item_s…	66,906,358	96	2019-11-07 19:18:08	1m 41s
GlobalHashAggregate(groupBy:(ss_list_price, $e),select:(ss_list_pri…	96	96	2019-11-07 19:19:49	10s

Optimization Points

- Basic Parameters Tweaks
- Compressed Transport
- Fastest Native IO
- Resilient Operator Memory Management
- More Fair Task Scheduling
- Other Random Fixs

Basic Parameters Tweaks

```
##------ 复赛默认配置，请勿修改，修改后算作无效成绩！-------##
jobmanager.heap.size: 8g
taskmanager.heap.size: 225g
taskmanager.numberOfTaskSlots: 225
##------ 复赛默认配置，请勿修改，修改后算作无效成绩！-------##

taskmanager.memory.off-heap: true
taskmanager.memory.preallocate: false
taskmanager.memory.fraction: 0.94
taskmanager.network.numberOfBuffers: 65536
env.java.opts.jobmanager: "-XX:+UseParallelGC"
env.java.opts.taskmanager: "-XX:+UseParallelGC"
parallelism.default: 104
table.optimizer.reuse-source-enabled: true (default true, enabled in source)
```

KEY:
taskmanager.heap.size
 != the JVM heap size
of taskmanager.
It counts for off-heap
memory usage.

Local Benchmark Configuration

- Xeon Platinum 8260 24c/48ht/1socket (performance mode + turbo boo st disabled)
- DDR4-2400 2*32G as Cache + DCPMM 2*128G (DRAM:PM 1:4)
- Intel 900P SSD for stable 2.1GB/s read IO, and Samsung consumer-level SSD pm981a for write(2GB+/s in-cache, 800MB/s out-cache)
- TPC-DS SF500 data + officially provided benchmark tool
- Linux(kinds of kernel, version is not much important here)
- My own local setup ~=

  0.45 * official online setup
  - Only NUMA can not

  be reproduced

```
—➤ sudo ipmctl show -topology

DimmID | MemoryType                  | Capacity    | PhysicalID| DeviceLocator
==============================================================================
0x0001 | Logical Non-Volatile Device | 126.375 GiB | 0x0011    | CPU1_DIMM_A2
0x0101 | Logical Non-Volatile Device | 126.375 GiB | 0x0015    | CPU1_DIMM_D2
N/A    | DDR4                        | 32.000 GiB  | 0x0010    | CPU1_DIMM_A1
N/A    | DDR4                        | 32.000 GiB  | 0x0014    | CPU1_DIMM_D1
```

DCPMM

- **Intel Optane DC Persistent Memory Module**
  - Memory mode (PMM Memory Mode-ware programming)
- **Observation**
  - local dithering is around 50sec (total ~1500sec)
  - online dithering is around 100sec (total ~1500sec)
- **Why larger dithering than that of local**
  - double larger size of working dataset
  - soft effect of automatic NUMA balancing(RHEL/Centos)
- **Why so large dithering?**

DCPMM

* Intel® 64 and IA-32 Architectures Optimization Reference Manual

- **Problem of PMM Memory Mode**
  - DRAM memory is used as directly-mapped cache for PMMs*
  - Buffering and combining at 256B does not work for multithreads
- **PMM Memory Mode-ware Programming**
  - Working dataset size SHOULD be smaller than the size of dram cache
  - Do NOT do random access < 256B
- **Conclusion**
  - Large dithering mainly stems from DCPMM
  - DCPMM is surprisingly a nice replacement of the DRAM for underutilized software even in memory mode

Compressed Transport

- **Observation**
  - IO-intensive in blocking mode
  - Disk-IO max write bandwidth: 1GB/s ...
- **Compression is natural optimization for (slow) disk-IO**

Query	largest size of scan op generated data
Query70	105GB
Query93	76.4GB
Query98	54.8GB

Compressed Transport

- **BoundedBlockingSubpartitionType**
  - FILE_MMAP, FILE, MMAP, AUTO(default)
  - Only two actually used: FILE_MMAP(64bit), FILE(other)
- **FILE_MMAP**
  - file write, mmap read
- **No zero-copy if general compression algorithm enabled**
  - FILE
  - How about unused MMAP type(mmap read, mmap write)?
    - Answer this question later

Compressed Transport

- **Which compression algorithm?**
  - CPU power is VERY VERY redundant, so compression ratio is preferred
- **Measurement**
  - Local standalone test, for sampling 32KB-128KB chunks
  - zstd compression ratio: 1/4 to 1/6
  - lz4 compression ratio: ~ 1/3
  - zstd compression bandwidth: 1.2GB+/s per core

Compressed Transport

- **Measurement (cont.)**
  - Online benchmark
  - Decrease totally average ~25% online run time
  - The overhead of blocking disk IO has been significantly mitigated

Query	largest size of scan op generated data	online benchmark time change
Query70	105GB	221s -> 121s
Query93	76.4GB	326s -> 275s
Query98	54.8GB	258s -> 142s

Compressed Transport

- **How about unused MMAP type?**
  - generally, mmap is sweet in fact
  - Yes, mmap write still need to dump to disk
    - It is async before hitting threshold ("lazy" called in API docs)
    - */proc/sys/vm/dirty_ratio* = 30 (default of RH/CENTOS)
    - Zstd gives awesome compression: 78GB->18GB for Query93 ss
    - Default implementation for MMAP has big problem(so I guess this is the reason it is abandoned), but can be fixed
    - Local standalone measurement: 2x faster than blocking file IO
    - But two reasons here

Fastest Native IO

- **Observation**
  - ParquetTableScan operator takes **RIDICULOUS** runtime for most cases even after compressed transport improvement
  - Disk-IO is definitely **NOT** the culprit
- **Measurement**
  - Local, typical, before op improvement

Query	run time of scan phase (seconds)	run time of whole query (seconds)
Query1	9	30
Query28	121	140
Query93	51	258

Fastest Native IO

- **Native IO for Parquet scan**
  - arrow_parquet_xx project
    - Column based low level C++ API on top of Apache Arrow Parquet
  - NativeParquetReader
    - Row based Flink-compatible high level Java Reader/API
  - Self-balancing splitting/partitioning for better scan subtask scheduling
- **Speedup**
  - C++ side APIs 10x than Prestosql's *
  - Java side APIs 5x than Prestosql's *
  - Java side APIs 2x than Flink's (ParquetVectorizedColumnRowReader) **

* https://github.com/jinmingjian/presto-parquet

** test and benchmark codes have been provided

Fastest Native IO

** https://databricks.com/glossary/what-is-databricks-runtime*
*** DBR recently rebranded to "Delta Platform" but still not open sourced as my understanding*

- **Native IO to push scan op into its limitation**
  - Flink/Blink claims 2x-3x faster than open-source Spark (Flink forward, 2018)
  - DBR(DataBricks Runtime) as commercially enhanced version of Spark 3x-8x faster than open-source Spark **
  - DBR just have a native IO layer(written in C++)**
  - **Fastest Parquet Reader in Java world (in highest standard)**
    - Passed local TPC-DS SF500 dedicated testcases and online SF1000 check
    - I am confident current impl can beat close-sourced DBR's
  - Bundles of extensions can be unlocked in future
- **No shame to have C++ written components for performance**

Fastest Native IO

- **Measurement**
  - Local, standalone test
  - SF500 store_sales table, columns reading same to Query93
  - Scan 1.4B records -> 48GB mmap writting with mmap-problem fixed (note: 78GB subpartition out for Flink's serialization schema)
  - 8 workers(threads)
  - 5 seconds for scan only (no dumping)
  - **16 seconds** for fixed mmap writing(why faster)
  - 26 seconds for mmap preallocated
  - VS ~**50 seconds**, Query93 store_sales scan time (only compressed 18GB subpartitions disk dump) (note: both use same hardware setup)

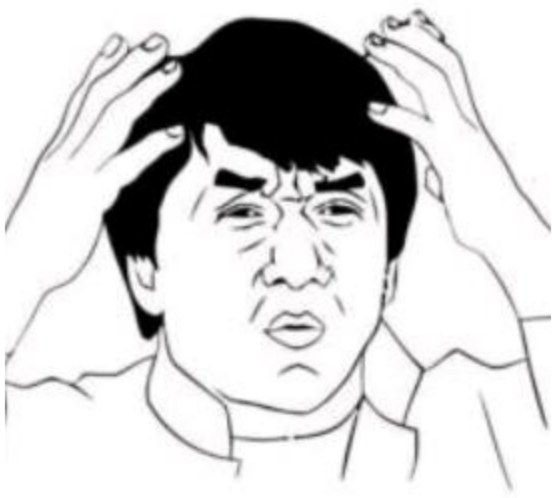
Fastest Native IO

- **Measurement (cont.)**
  - Local, the shortest run time measure in five runs

Query	run time of scan phase before (seconds)	run time of scan phase after (seconds)
Query1	9	7
Query2	20	15
Query28	118	101
Query30	3	3
Query44	21	18
Query70	60	53
Query93	50	42
Query98	35	29

Fastest Native IO

- **Measurement (cont.)**
  - online evaluation (included dithering)
    - estimated ~5% average improvement for online benchmark



Fastest Native IO

- Analysis

```
write:166, DataOutputSerializer (org.apache.flink.core.memory)
serializeWithoutLength:147, BinaryRowSerializer (org.apache.flink.table.runt
serialize:88, BinaryRowSerializer (org.apache.flink.table.runtime.typeutils)
serialize:91, BaseRowSerializer (org.apache.flink.table.runtime.typeutils)
serialize:50, BaseRowSerializer (org.apache.flink.table.runtime.typeutils)
serialize:175, StreamElementSerializer (org.apache.flink.streaming.runtime.s
serialize:46, StreamElementSerializer (org.apache.flink.streaming.runtime.str
write:54, SerializationDelegate (org.apache.flink.runtime.plugable)
serializeRecord:78, SpanningRecordSerializer (org.apache.flink.runtime.io.ne
emit:152, RecordWriter (org.apache.flink.runtime.io.network.api.writer)
emit:120, RecordWriter (org.apache.flink.runtime.io.network.api.writer)
pushToRecordWriter:107, RecordWriterOutput (org.apache.flink.streaming.r
collect:89, RecordWriterOutput (org.apache.flink.streaming.runtime.io)
collect:45, RecordWriterOutput (org.apache.flink.streaming.runtime.io)
collect:727, AbstractStreamOperator$CountingOutput (org.apache.flink.stre
collect:705, AbstractStreamOperator$CountingOutput (org.apache.flink.stre
collect:104, StreamSourceContexts$NonTimestampContext (org.apache.flink
run:331, ContinuousFileReaderOperator$SplitReader (org.apache.flink.strea
```

Fastest Native IO

- Analysis

```
try {
    OT nextElement = serializer.createInstance();  nextEl
    while (!format.reachedEnd()) {
        synchronized (checkpointLock) {  checkpointLock:
            nextElement = format.nextRecord(nextElement);
            if (nextElement != null) {
                readerContext.collect(nextElement);  read
            } else {
                break;
            }
        }
    }
    completedSplitsCounter.inc();

} finally {
```

Fastest Native IO

- Analysis

```
public void serializeRecord(T record) throws IOException {
    if (CHECKED) {
        if (dataBuffer.hasRemaining()) {  dataBuffer: "java
            throw new IllegalStateException("Pending seria
        }
    }

    serializationBuffer.clear();
    lengthBuffer.clear();  lengthBuffer: "java.nio.HeapByte

    // write data and length
    record.write(serializationBuffer);  record: Serializat

    int len = serializationBuffer.length();
    lengthBuffer.putInt( index: 0, len);

    dataBuffer = serializationBuffer.wrapAsByteBuffer();
}
```

Fastest Native IO

- Analysis

```
public void processElement(org.apache.flink.streaming.runtime.streamrecord.StreamRecord element) throws Exception {
  org.apache.flink.table.dataformat.BaseRow in1 = (org.apache.flink.table.dataformat.BaseRow) element.getValue();

  org.apache.flink.table.dataformat.BinaryString field$81;
  boolean isNull$81;
  boolean isNull$85;
  boolean result$86;



  isNull$81 = in1.isNullAt(1);
  field$81 = org.apache.flink.table.dataformat.BinaryString.EMPTY_UTF8;
  if (!isNull$81) {
    field$81 = in1.getString(1);
  }
  org.apache.flink.table.dataformat.BinaryString field$82 = field$81;
  if (!isNull$81) {
    field$82 = (org.apache.flink.table.dataformat.BinaryString) (typeSerializer$83.copy(field$82));
  }



  isNull$85 = isNull$81 || false;
  result$86 = false;
  if (!isNull$85) {

    result$86 = field$82.equals(((org.apache.flink.table.dataformat.BinaryString) str$84));

  }
```

Generated BatchCalc Op

Fastest Native IO

- **Measurement**
  - Tracing Query93 reader next-to-next latency (HdrHistogram*)

** https://github.com/HdrHistogram/HdrHistogram*

```
       Value     Percentile TotalCount 1/(1-Percentile)

     377.000 0.000000000000          1           1.00
     695.000 0.100000000000    1539884           1.11
     738.000 0.200000000000    3040272           1.25
...
256770047.000 0.999999928474   15143624    13981013.34
340525055.000 0.999999940395   15143625    16777216.00
340525055.000 1.000000000000   15143625
#[Mean    =       3177.550, StdDeviation   =    243864.199]
#[Max     = 340525055.000, Total count    =      15143625]
#[Buckets =             21, SubBuckets     =          2048]
```

<- Histogram dump from one partition reader (Unit: nanoseconds)

Let us do a simple math:
*3177e-9*15.14e6 = 48s !!!*
(total time elapse 52s in this run, subpartition dumping is async and overlapped)

Resilient Operator Memory Management

- **Observation**
  - Many disk-IO and spilling logging for long run hash join op
  - System memory still has
  - Tweaking option "table.exec.resource.hash-join.memory"
    - Small: all happy except spilling for big hash join and unused system memory
    - Middle: short-run joins start uprising, but still spilling for big hash join
    - Large: single runs of big hash joins seem great improved (in that all in memory), but all 20-case benchmark can not be completed for out-of-memory
- **Analysis**
  - Suspected memory leak: mem usage just increase and not decrease
    - Too many locations to take and "free"
  - Naïve allocation algorithm: preallocated when hashjoin op opened

Resilient Operator Memory Management

- **Native Memory Manager**
  - A new kind off-heap memory manager introduced
- **API Design (for hashjoin)**
  - Strict memory ownership boundary (mechanism guarantees no memory leaking)
    - Clean and converge all-around take/free points into one take method and two free variants in BaseHybridHashTable
    - Only root table op is responsible for allocate/free
    - Children data structure ask table for "take" mem segments and do not care "free"
    - Except when eagerly free wanted, e.g. rehash, they can ask table for immediate "free"
  - Resilient memory usage
    - Request from 0 to unallocable , and return to system from unallocable to 0

Resilient Operator Memory Management

- **Native Memory Manager (cont.)**
  - JEMalloc based
  - **4x faster** than UNSAFE.allocateMemory/freeMemory (for 32KB chunk, tested)
    - Same behind java.nio.ByteBuffer#allocateDirect
    - and behind MemorySegmentFactory#allocateUnpooledOffHeapMemory…
  - **Strict "Contract"** guarantees memory-leak-free coding
    - All memory segments that Children "take"/"free"-ed must be allocated by ROOT
    - ROOT will and only "free" all its allocated memory segments
    - VS that direct ByteBuffer only released when Full GC
  - **Eagerly return memory to system**
    - VS that glibc's malloc (behind direct ByteBuffer) has problem to return its memory to system
  - **Advanced APIs** can help to boost the performance of general memory usages

Resilient Operator Memory Management

- **Measurement**
  - Online benchmark

Query	run time of query before (seconds)	run time of query after (seconds)
query25	347	244
query26	49	39
query38	106	106
query41	4	4
query93	275	180
query98	142	134

Resilient Operator Memory Management

- **Measurement (cont.)**
  - local

dsk/nvme2n1		--total-cpu-usage--					------memory-usage-----				io/nvme2n1p	
read	writ	usr	sys	idl	wai	stl	used	free	buff	cach	read	writ
0	0	48	3	49	0	0	79.3G	79.4G	36.2M	88.0G	0	0
0	500M	48	3	48	0	0	79.3G	79.1G	36.2M	88.2G	0	442
0	0	49	3	48	0	0	79.3G	78.8G	36.2M	88.5G	0	0
0	0	48	3	49	0	0	79.3G	78.6G	36.2M	88.7G	0	0
0	22k	49	3	48	0	0	79.3G	78.3G	36.2M	89.0G	0	1.67
0	0	48	3	49	0	0	79.3G	78.1G	36.2M	89.3G	0	0
0	351M	49	3	48	0	0	79.3G	77.8G	36.2M	89.5G	0	308
0	164M	48	3	49	0	0	79.3G	77.5G	36.2M	89.8G	0	147
0	0	49	3	49	0	0	79.3G	77.2G	36.2M	90.1G	0	0
0	18k	49	3	49	0	0	79.3G	77.0G	36.2M	90.3G	0	0.67
0	0	59	9	33	0	0	75.1G	81.7G	36.2M	89.9G	0	0
0	1475k	91	6	3	0	0	75.1G	83.9G	36.2M	87.6G	0	46.3
0	309M	92	6	2	0	0	75.1G	85.5G	36.2M	86.0G	0	297
0	503k	92	6	2	0	0	75.1G	87.6G	36.2M	84.0G	0	16.0
0	4640k	90	7	4	0	0	75.1G	89.6G	36.2M	82.0G	0	155

Resilient Operator Memory Management

- **Measurement (cont.)**
  - local

```
  0      0 | 99   1   0   0   0|51.3G  147G 35.2M 49.0G|   0     0
  0      0 | 99   1   0   0   0|51.5G  146G 35.2M 49.3G|   0     0
  0  5120k| 99   1   0   0   0|51.5G  146G 35.2M 49.5G|   0   160
  0      0 | 99   1   0   0   0|51.6G  146G 35.2M 49.7G|   0     0
  0    11M| 99   1   0   0   0|51.6G  145G 35.2M 49.9G|   0   256
sk/nvme2n1 --total-cpu-usage-- ------memory-usage----- io/nvme2n1p
read  writ|usr sys idl wai stl| used  free  buff  cach| read  writ
  0      0 | 99   1   0   0   0|51.6G  145G 35.2M 50.1G|   0     0
  0      0 | 99   1   0   0   0|51.6G  145G 35.2M 50.3G|   0     0
  0      0 | 99   1   0   0   0|51.6G  145G 35.2M 50.4G|   0     0
  0   408M| 97   3   0   0   0|51.8G  145G 35.2M 50.6G|   0   945
  0   153M| 98   2   0   0   0|51.8G  144G 35.2M 50.7G|   0   485
  0  1618k| 99   1   0   0   0|51.8G  144G 35.2M 50.7G|   0  29.3
  0      0 | 98   2   0   0   0|52.0G  144G 35.2M 50.7G|   0     0
  0      0 | 96   3   0   0   0|52.3G  144G 35.2M 50.8G|   0     0
  0   992k| 96   3   0   0   0|52.6G  143G 35.2M 50.8G|   0  31.3
  0   167M| 97   3   0   0   0|52.7G  143G 35.3M 50.9G|   0   409
  0    19M| 96   4   0   0   0|52.7G  143G 35.3M 50.9G|   0   441
```

Resilient Operator Memory Management

- **This is best engineering practice for large chunk** (1KB/4KB/32KB+)
  - **Strict** allocate/free contract and no memory leak
  - **Variant** sizes supported
  - **Return** to system
  - **No GC** pressure
  - **Negligible** JNI overhead
- Small chunk allocation(free)
  - (my) Landz's pure on-heap ZMalloc beats natively JEMalloc
- **Proposed as an universal memory manager**
  - Can be trivially extended to all operators
  - Orthogonal and extensible to kinds of high-level cleaner schema
  - Secrete weapon more powerful than "Project Tungsten"

More Fair Task Scheduling

- **Status**
  - Current scheduling algorithm is **not fair enough** between Task Managers
    - Random like
  - Especially for **NUMA and benchmark**
    - TM usually pinned to some node(socket)
    - If not pinned, OS scheduling between nodes could be a little more expensive(so NUMA-ware)
    - True parallelism of this high task parallelism scenario is limited by hardware in fact
- **Observation**
  - Let Flink/OS do scheduling freely, there is **10%-15% unbalanced slot assignment** and very large dithering.

More Fair Task Scheduling

- **Improvement**
  - **Pin** two Task Managers to different nodes
  - **Round-robin** between all Task Managers
  - Simple but **efficient**

Random Stuff

- **NUMA-ware start script Fix**
  - Option “taskmanager.compute.numa”
    - Bind Task Managers to different NUMA Nodes
  - But it does NOT work when missing numactl tool
    - Fix start-cluster.sh to use taskset to do numa binding
- **High performance Java Util library**
  - Stripped from Landz project
  - Include many pearls
    - Unsafe tools(address <-> Buffer <-> NIO Direct Buffer)
    - Unsafe thread local data structures (faster than java.lang.ThreadLocal#ThreadLocal and Flink’s its usages)
    - Faster common expensive object constructors …

Next

- **At hand**
  - Merge intra-pipeline predicate operator into reader
    - BatchCalc op is very expensive as we seen
    - Further 2x speedup expected
  - Further optimization for Native reader
    - skip_to_row
  - BuildWriteBuffer is expensive
    - possible 0-copy even with compression
- **Next's Next**
  - Redesign whole ser/deser/late-materialization schema
  - Unlimited gameplay on the top of my contribution: Apache Arrow based

Summary

- All optimizations are **fundamental** and **benefit all cases**
  - **PMM Memory Mode-ware Programming**, carefully design opts to match the hardware
  - approaches current **architecture limit** when queries can not be further more "clever" planned
- All optimizations are **firstly originally created** in the competition and in the whole Java big data ecosystem
- All optimizations are trusted as **best of world** and almost in **highest engineering standard**
- **Suggestions** are provided for further performance breakthrough based on **scientific measurements**

Thanks