CarbonData 的实践与调优
Action & Tuning in CarbonData

徐传印
Huawei SmartCare产品大数据平台
2018.12.01

关于我

Huawei SmartCare 产品大数据平台系统工程师

2013年进入华为公司，长期从事大数据技术在电信领域的研究、优化和应用。

2017年5月接触CarbonData，开始调研和优化CarbonData在详单查询业务中的使用，重点优化了CarbonData的入库性能。

2018年5月成为Apache CarbonData Committer。

1. 业务介绍与技术选择

2. 对CarbonData的优化

    2.1 入库优化

    2.2 查询优化

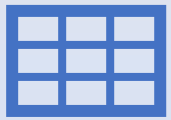
业务介绍

场景介绍

> 100TB/天

30%/年

5 min



百亿记录
300+ 字段

SQL 支持

交互式查询

大型分析

多租户支持

我们的选择(1/2)

Impala + Parquet的问题

1. 入库慢，现网实测 60M/s/Node，达到天花板

2. 使用两个动态分区，入库时数据shuffle加剧磁盘I/O，性能更差。no-shuffle时小文件太多。

3. 查询用不上分区时，查询性能比较差。

4. MPP架构，百节点以上的扩容存在问题。

5. 未集成Yarn资源管理，需要单独的集群。多用户时，没有资源隔离。和其他组件共用HDFS时，数据倾斜。

我们的选择(2/2)

什么是CarbonData

一种Hadoop Native的列存文件格式

Parquet ?

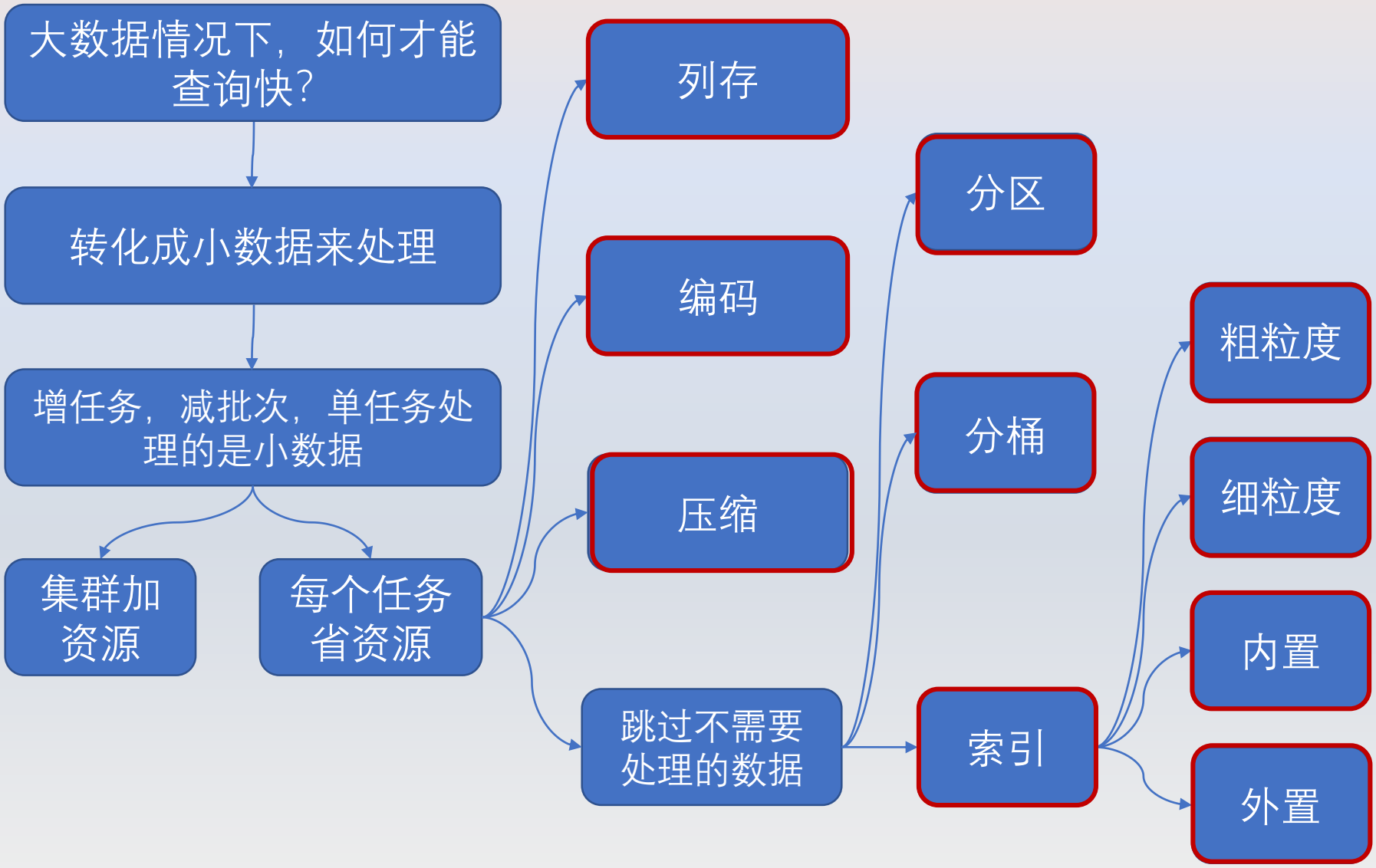
一种带索引的文件格式

Parquet 2.0 ?

一种带索引的文件格式 + Spark/Presto 计算引擎深度集成和优化的解决方案

拓展了 Spark SQL 的语法，提供了数据管理的功能，拥有丰富的索引，及针对计算引擎对查询和计算进行了深度优化

为什么选择CarbonData

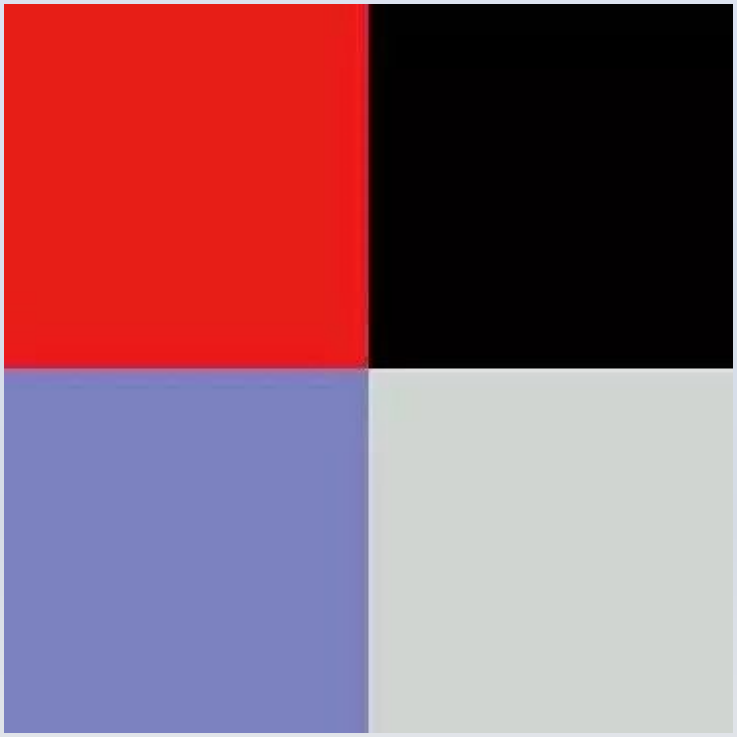


业务诉求

查询快

入库快

计算快



膨胀低

易扩容

易集成

易演化

够开放

总体优化效果

查询快	查询性能提升一倍以上
入库快	入库性能提升2倍， 35MB/s/Node -> 101MB/s/Node
	端到端I/O减少40+%
	数据倾斜
	小数据大集群的问题
膨胀低	引入 Zstd 压缩

入库优化

一些概念(1/3)

Segment：一次数据入库就是一个Segment。

Block：一个CarbonData数据文件，大小由TABLE_BLOCKSIZE控制。

Blocklet：CarbonData组织数据时，将数据水平分组，每组就是一个Blocklet，类似与Paruqet的RowGroup。Blocklet的大小可配置。一个Block中可包含多个Blocklet。

Chunk：每个列的列存数据。个数和列数相关。

Page：一个列的最小编码和压缩单元，一般包含32000个值。一个列的所有Page都放在一起，组成该列的Chunk。

一些概念(2/3)

Sort_Columns：数据按照这些字段进行排序。

为什么排序?

- ✓ 改变数据分布，相同的记录聚集在一起，便于扫描;
- ✓ 索引起来更高效;
- ✓ 编码和压缩效果更好。

一些概念(3/3)

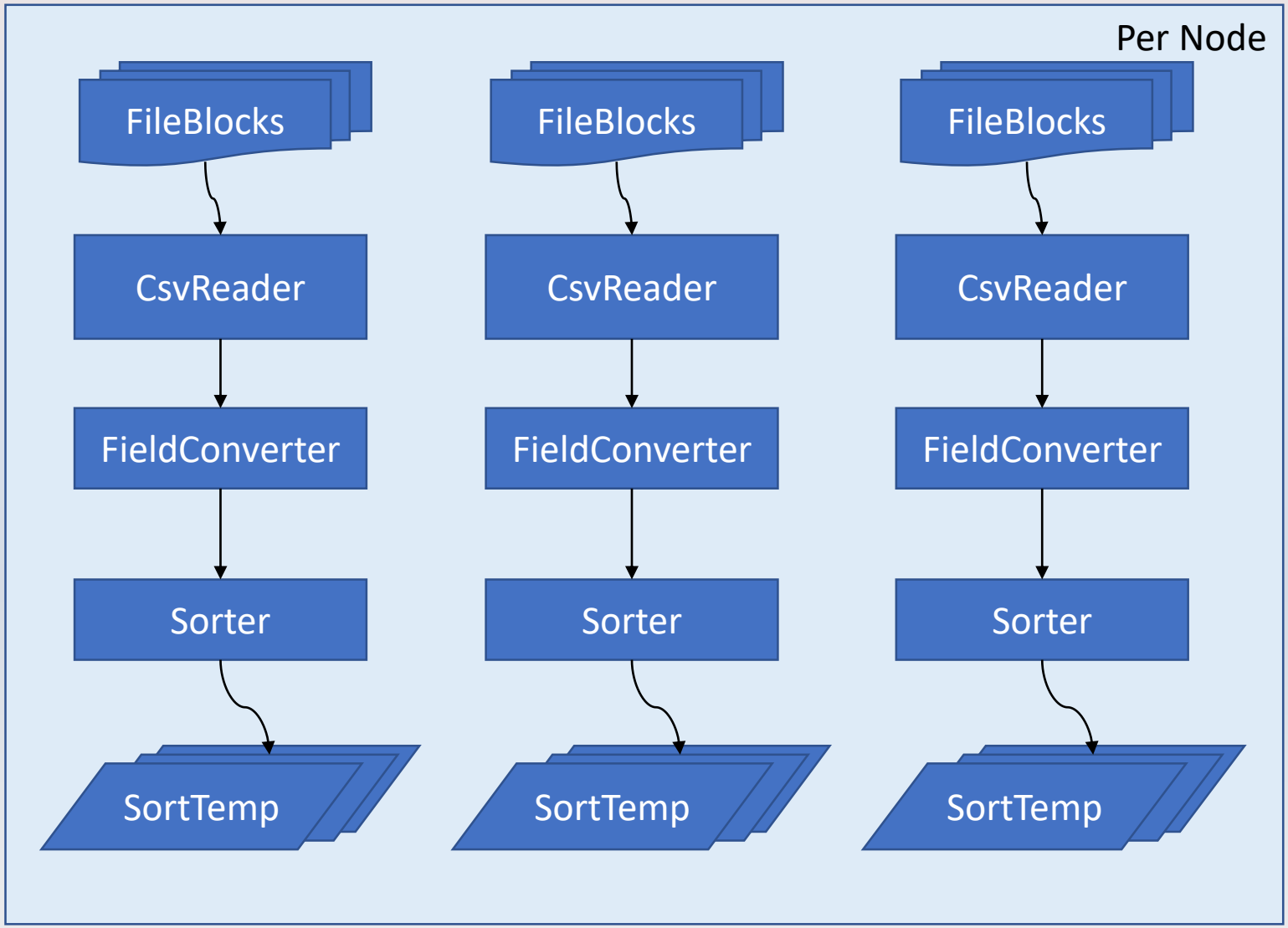
Sort_Scope：排序的级别。

排序级别	含义
Global_Sort	整个Segment内的记录是有序的。
Local_Sort	输入数据按照节点分成N份，每份内的记录是有序的。（默认选项）
Batch_Sort	根据内存的量，每读取一批数据就进行一次排序。
No_Sort	完全不排序。

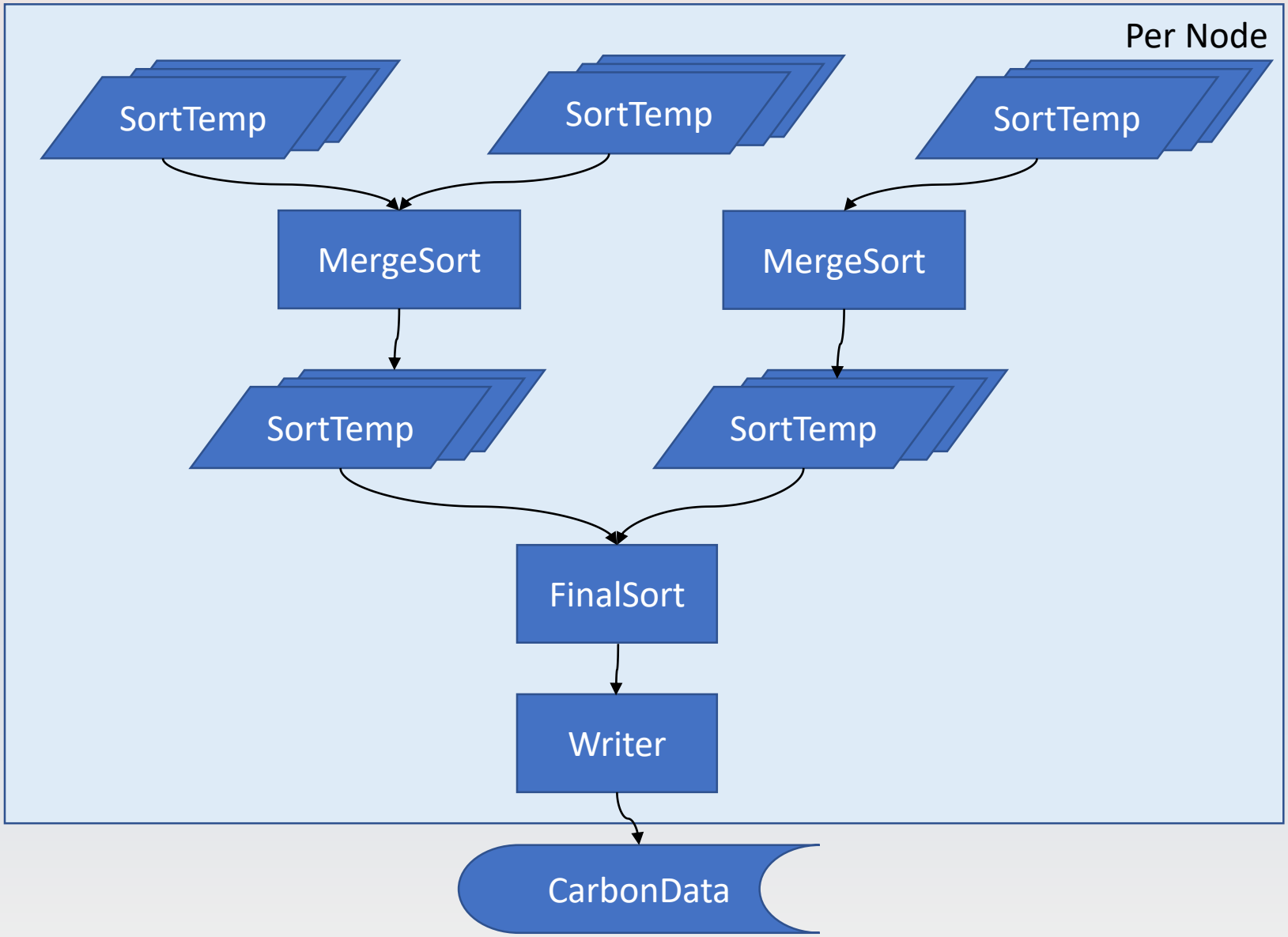
查询性能：Global_Sort > Local_Sort > Batch_Sort > No_Sort
入库性能：Global_Sort < Local_Sort < Batch_Sort < No_Sort

我们当前选择的是 Local_Sort

入库流程(1/2)

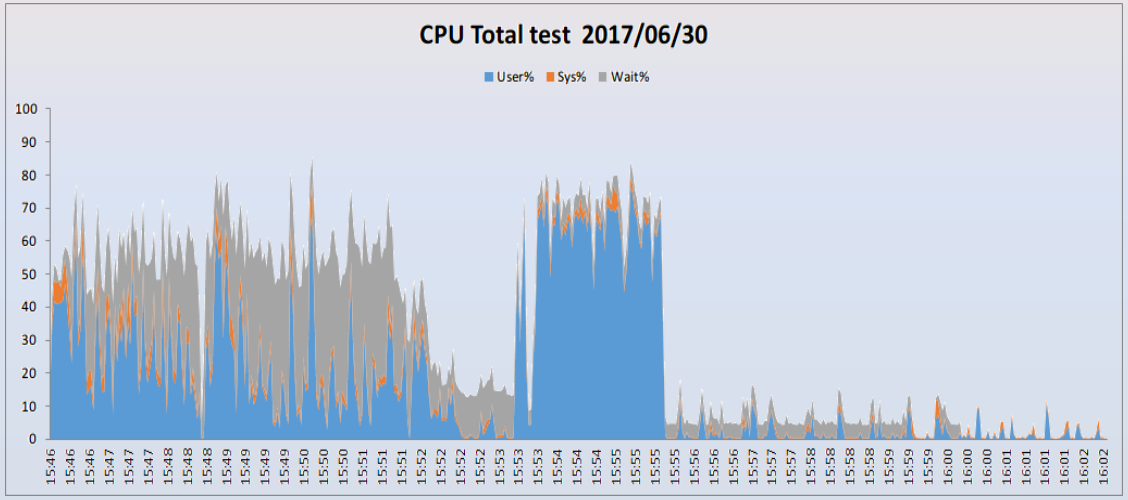

Per Node
FileBlocks
CsvReader
FieldConverter
Sorter
SortTemp
FileBlocks
CsvReader
FieldConverter
Sorter
SortTemp
FileBlocks
CsvReader
FieldConverter
Sorter
SortTemp

入库流程(2/2)

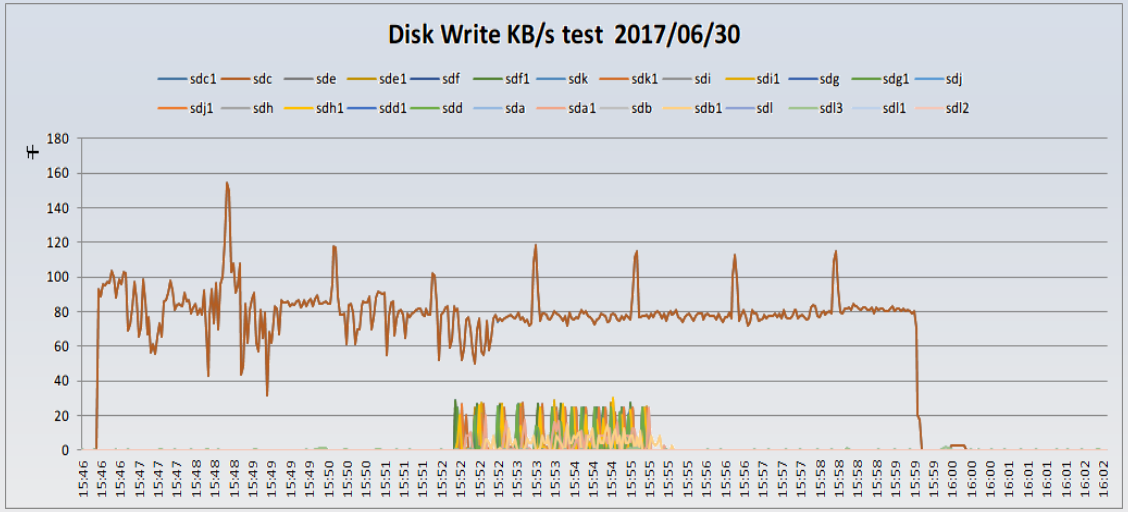

Per Node
SortTemp
SortTemp
SortTemp
MergeSort
MergeSort
SortTemp
SortTemp
FinalSort
Writer
CarbonData

入库优化#1-问题引入

35
MB/s/Node

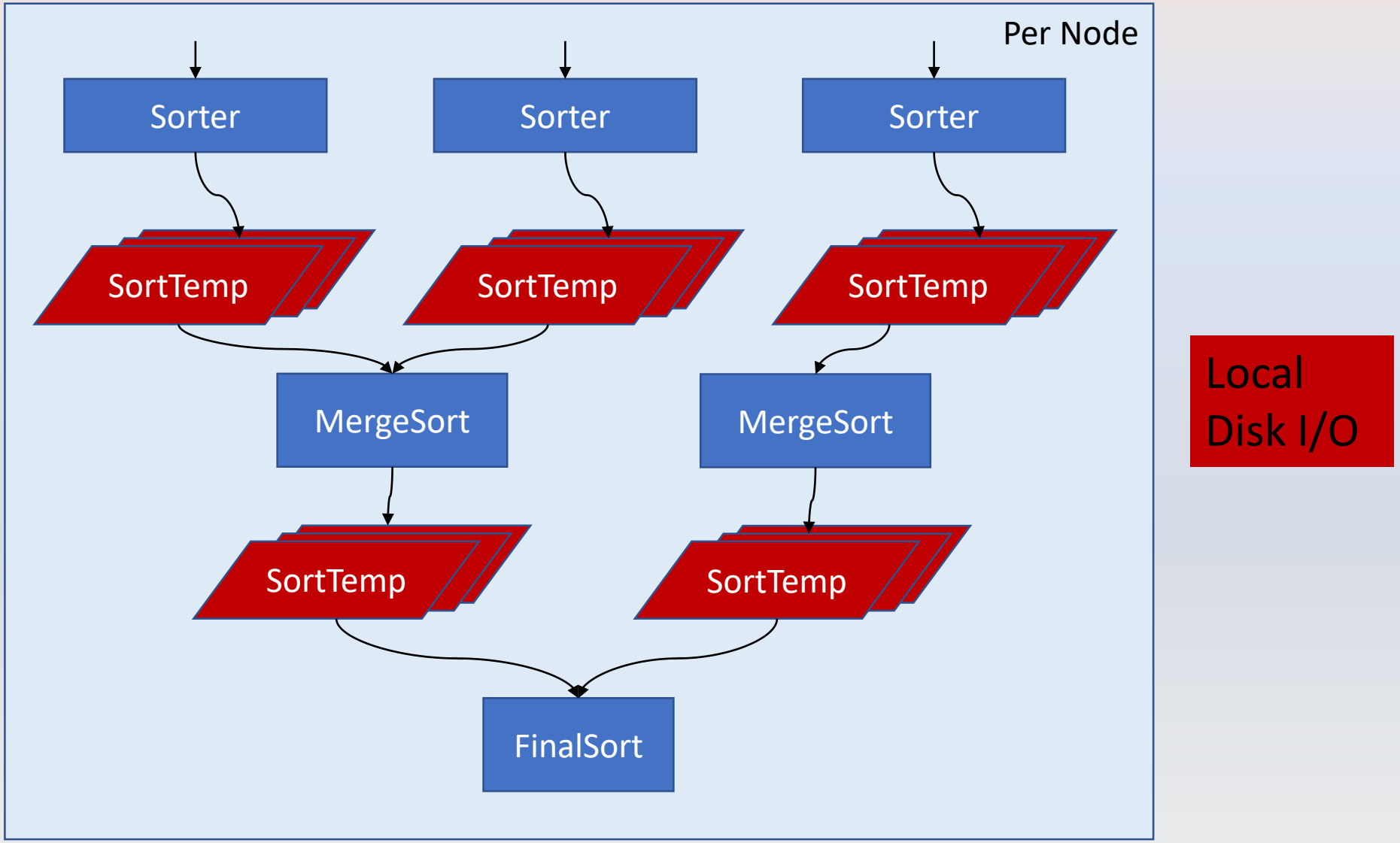


CPU存在大量Wait



Disk Write 存在热点

入库优化#1-原理与根因


Per Node
Sorter
Sorter
Sorter
SortTemp
SortTemp
SortTemp
MergeSort
MergeSort
SortTemp
SortTemp
FinalSort
Local
Disk I/O

入库优化#1-优化实施

Multiple Temp Dir

根因：

✓ SortTemp文件写到同一个磁盘了。

解决方法：

✓ 在每次写SortTemp文件时，从配置的数据目录中随机挑选一个，使Disk I/O均衡。

✓ 数据目录：yarn.nodemanager.local-dirs.

使用方式：

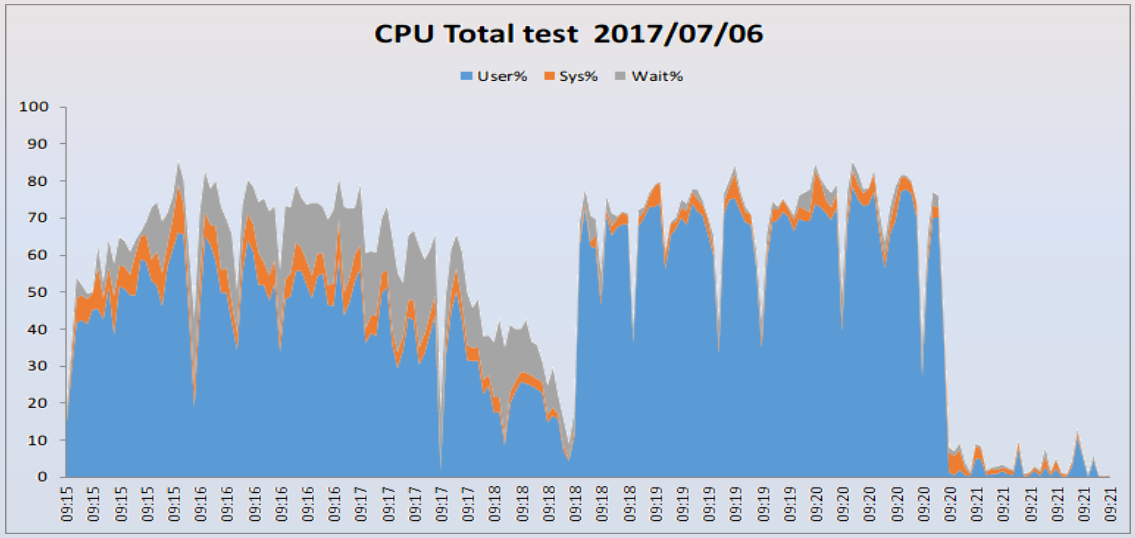
✓ 在1.3.0中引入该特性，默认关闭。使用时需同时打开 carbon.use.local.dir 和 carbon.use.multiple.temp.dir 。

✓ 从1.5.1开始，该特性默认开启，由参数 carbon.use.local.dir 控制

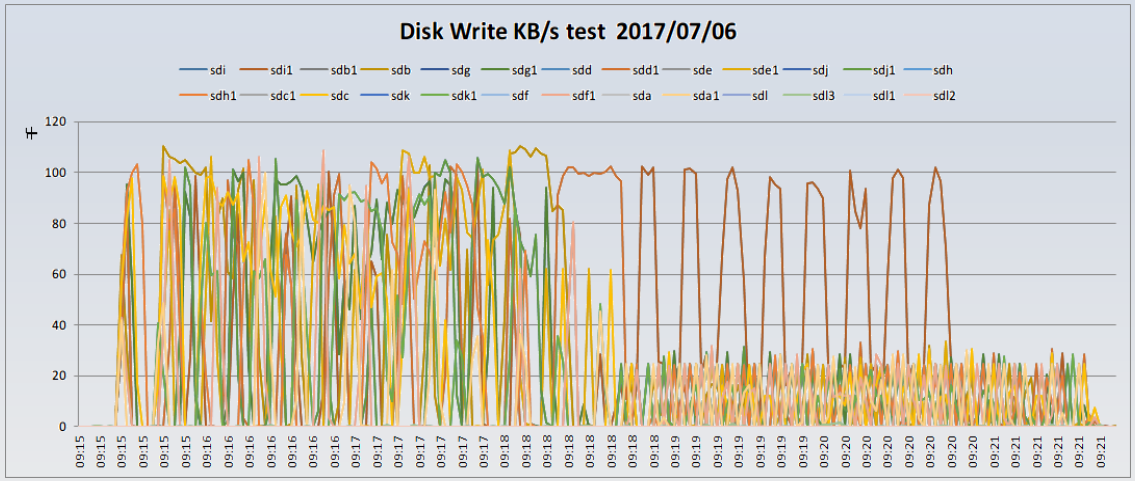
入库优化#1-优化效果

68 MB/s/Node

35 MB/s/Node



CPU Wait减少很多

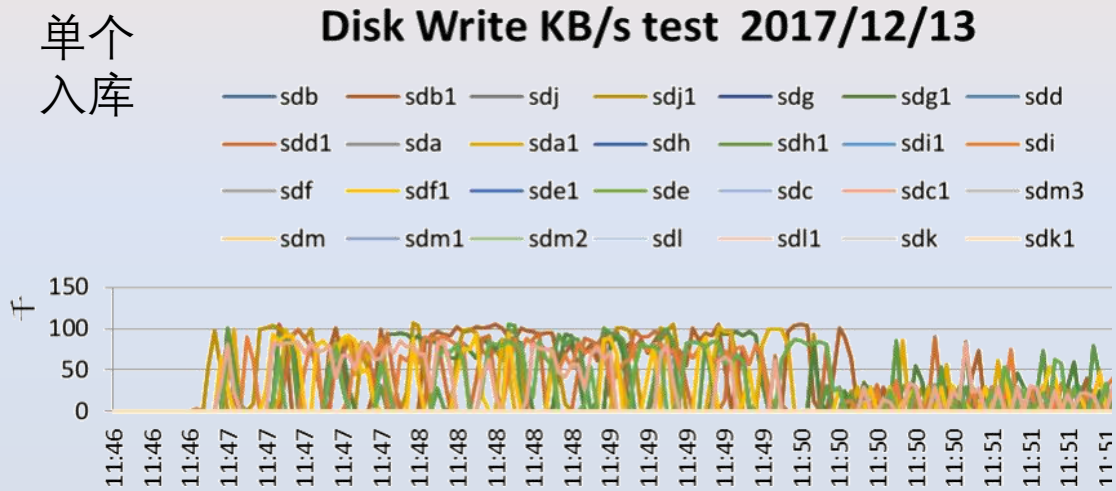


Disk Write热点消失

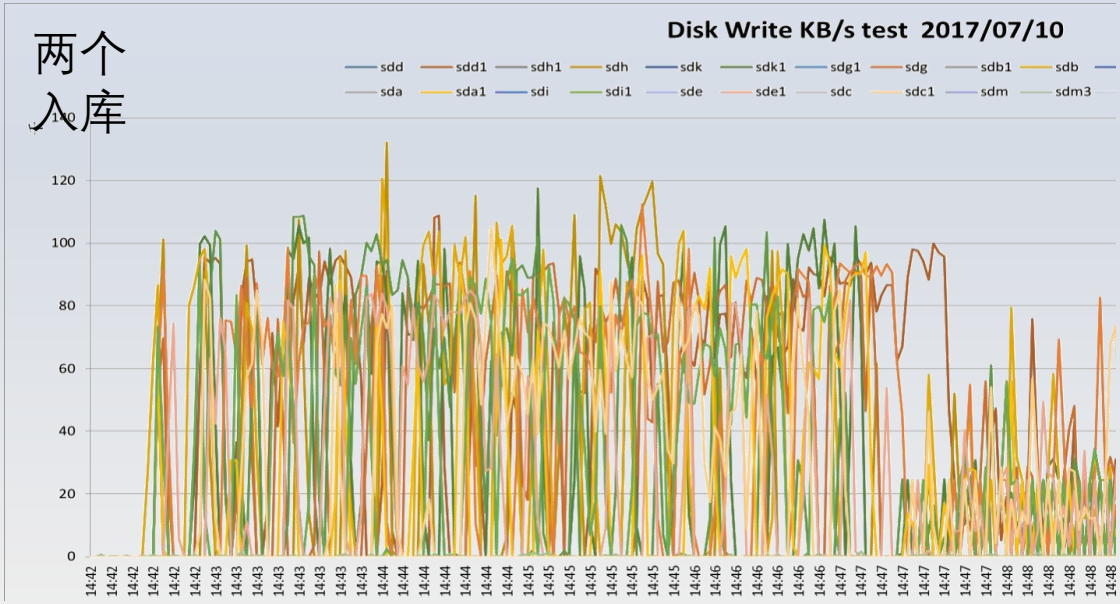
** 入库后半段的热点是因为最终数据写磁盘时也有热点，后续已优化。*

入库优化#2 – 问题引入

单个入库



两个入库



数据入库时，磁盘I/O很高；

排序阶段 Disk Write 几乎达到瓶颈

入库优化#2 – 环境变更

74 MB/s/Node

68 MB/s/Node

	配置项	旧集群	新集群	变化点
Huawei RH2288 * 3	CPU	E5-2667@2.9GHz * 24	E5-2680@2.5GHz * 48	单核CPU频率变低
	Mem	256GB	256GB	
	Disk	SAS * 10, 100MB/s (write per 64MB)	SATA * 12, 170MB/s (write per 64MB)	SAS换成SATA后，实际磁盘IO性能**更强**
Executor * 3	Cores	20	20	
	Mem	128GB	100GB	使用内存减少

** 测试数据: csv 72.1GB, 88million records, 330 fields*

** 旧集群偶尔出现磁盘导致系统宕机*

入库优化#2 – 优化实施

Compress Sort Temp

根因：

✓ SortTemp写盘太多。

解决方法：

✓ 对落盘的SortTemp进行压缩。

使用方式：

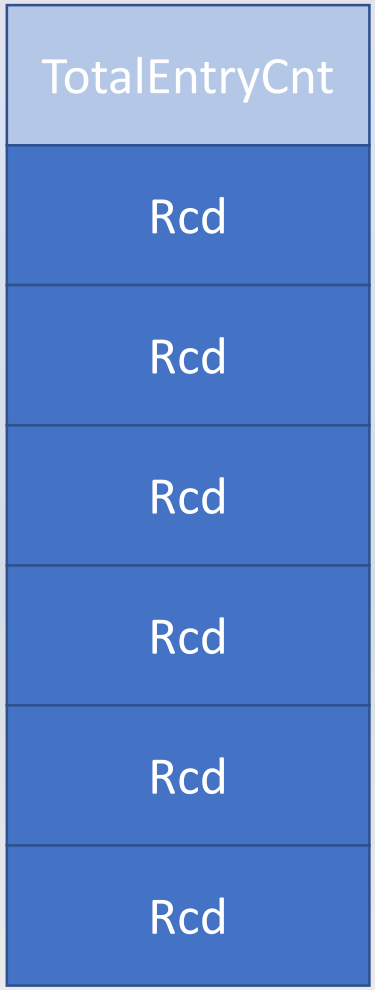
✓ 在1.3.0中引入该特性，由参数 carbon.sort.temp.compressor 控制，支持不压缩、Snappy、Gzip、BZip2、LZ4压缩。默认不压缩。

✓ 在1.4.1中引入Zstd压缩的支持。

✓ 从1.5.1开始，该默认启用Snappy压缩。

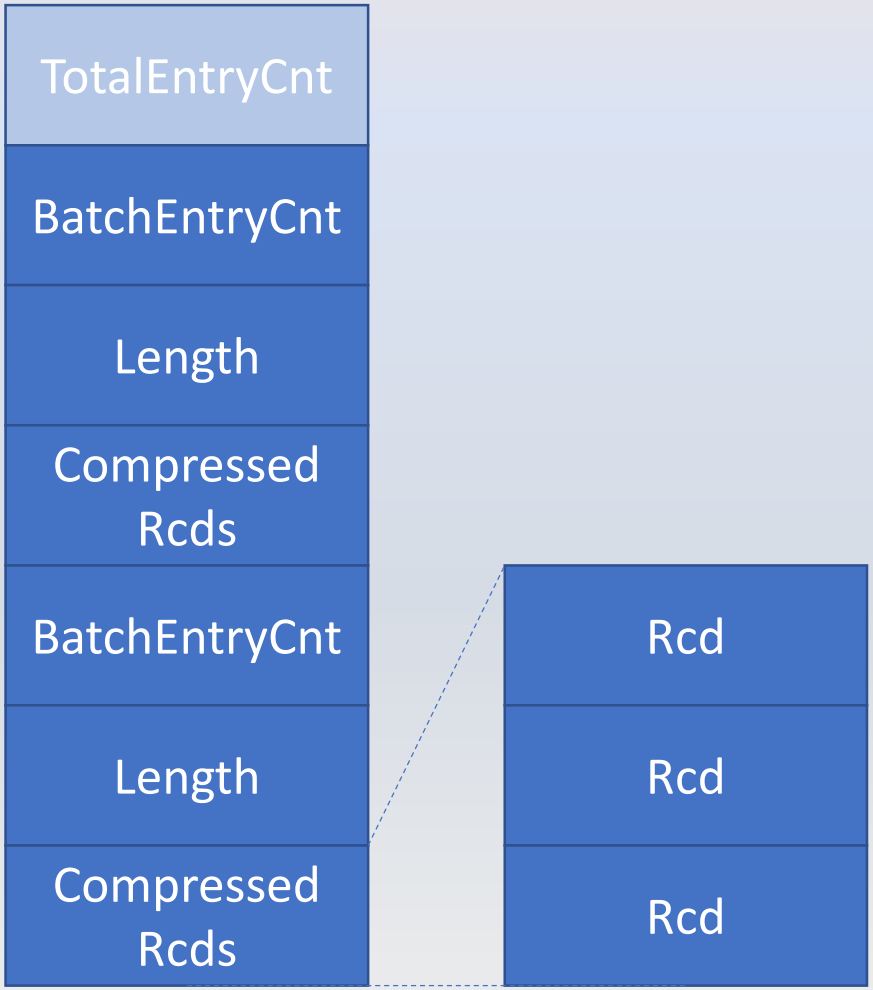
入库优化#2 – 实施过程(1/4)

Uncompressed



Compressed*

** Existed Implementation with bug*



入库优化#2 – 实施过程(2/4)

修改:

Bug Fix

效果:

入库性能几乎下降了一半

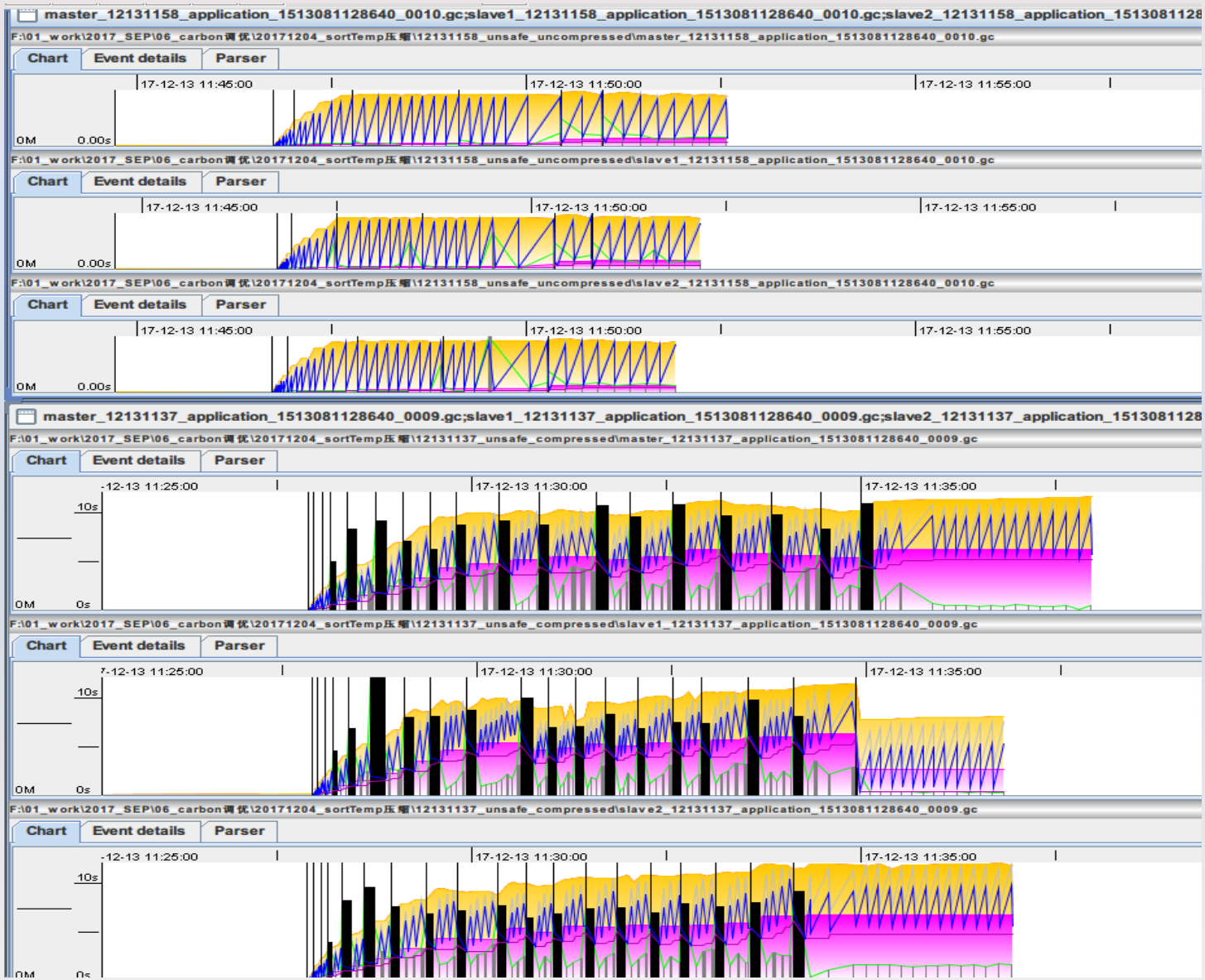
Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Re Se
0	0	0	SUCCESS	NODE_LOCAL	2 / slave2 stdout stderr	2017/12/13 11:46:42	5.4 min	0.2 s	1 s	11 s	2 m
1	2	0	SUCCESS	NODE_LOCAL	1 / slave1 stdout stderr	2017/12/13 11:46:42	5.6 min	91 ms	1 s	14 s	2 m
2	1	0	SUCCESS	NODE_LOCAL	3 / master stdout	2017/12/13 11:46:42	5.9 min	0.3 s	2 s	16 s	2 m

修改前

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Resu Seria
0	1	0	SUCCESS	NODE_LOCAL	2 / slave2 stdout stderr	2017/12/13 11:27:51	9.2 min	0.1 s	1 s	4.2 min	2 ms
1	2	0	SUCCESS	NODE_LOCAL	1 / slave1 stdout stderr	2017/12/13 11:27:51	9.0 min	0.2 s	1 s	4.1 min	2 ms
2	0	0	SUCCESS	NODE_LOCAL	3 / master stdout	2017/12/13 11:27:51	10 min	0.1 s	1 s	4.4 min	2 ms

修改后

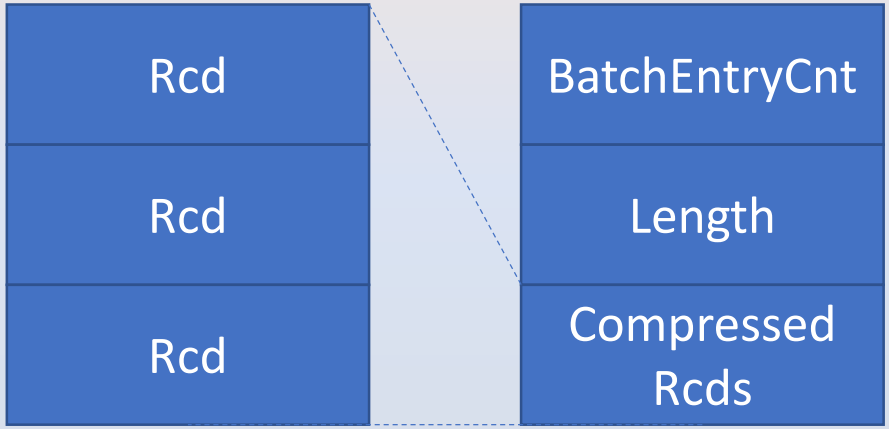
入库优化#2 – 实施过程(3/4)



修改前

修改后

入库优化#2 – 实施过程(4/4)



Batch 大小：1000
每条记录预分配空间：2MB*

每Batch需一次分配较大的空间，该空间可能会直接进入到老年代，从而容易引发频繁的FULL GC。多Batch同时处理时更加重该问题。

思路：
减少大空间的分配，尽量复用空间。

方案：
1.采用文件级别压缩，使用compressed file stream接口。
2.仅需分配2MB大小空间，并且一直复用该空间。
3. 文件结构和实现代码和Uncompressed场景统一。

** 在1.5.1中，该空间已经可以动态增长，以便于支持超过2MB的记录。*

入库优化#2 – 优化效果

71
MB/s/Node

74
MB/s/Node

从nmon上看，整个过程中磁盘写入量降低了近1/3。

使用Snappy压缩时，入库性能却下降了：74M/s → 71M/s

分析：

在老集群上，磁盘是瓶颈，在新集群上CPU是瓶颈。

Disk write: 100MB/s v.s. 170MB/s;

CPU: 20*2.9GHz v.s. 20*2.5GHz

原本加载时对CPU消耗已经很高，再加上压缩后，CPU更是瓶颈。

入库优化#3 – 问题、原理与优化

Pack No-Sort Fields

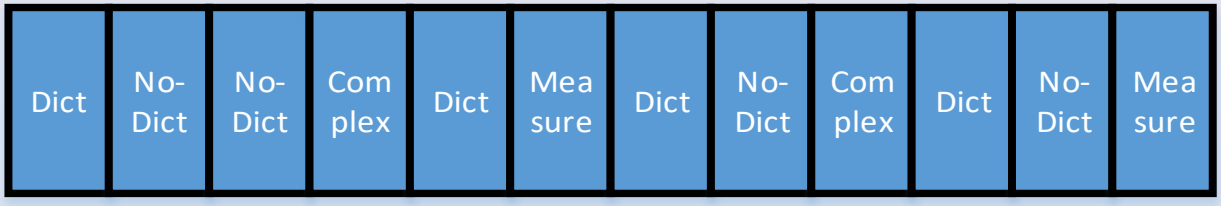
背景：

排序时，每次读写SortTemp文件，都需要对记录进行序列化/反序列化。

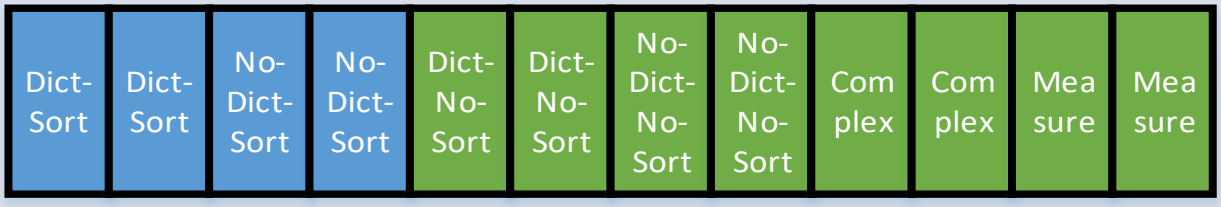
优化：

序列化和反序列化时，对不参与排序的字段进行整体打包，不去解析其内容，以减少CPU的消耗。

Raw: 原始输入的一行，dict/no-dict/complex/measure位置不确定



Rearrange：将参与排序的字段放前面，不参与排序的字段放后面



Pack：对no-sort的所有列打包成字节数组，序列化和反序列化时不解析

*在1.4.0引入，默认启用，无参数控制

入库优化#3 – 优化效果

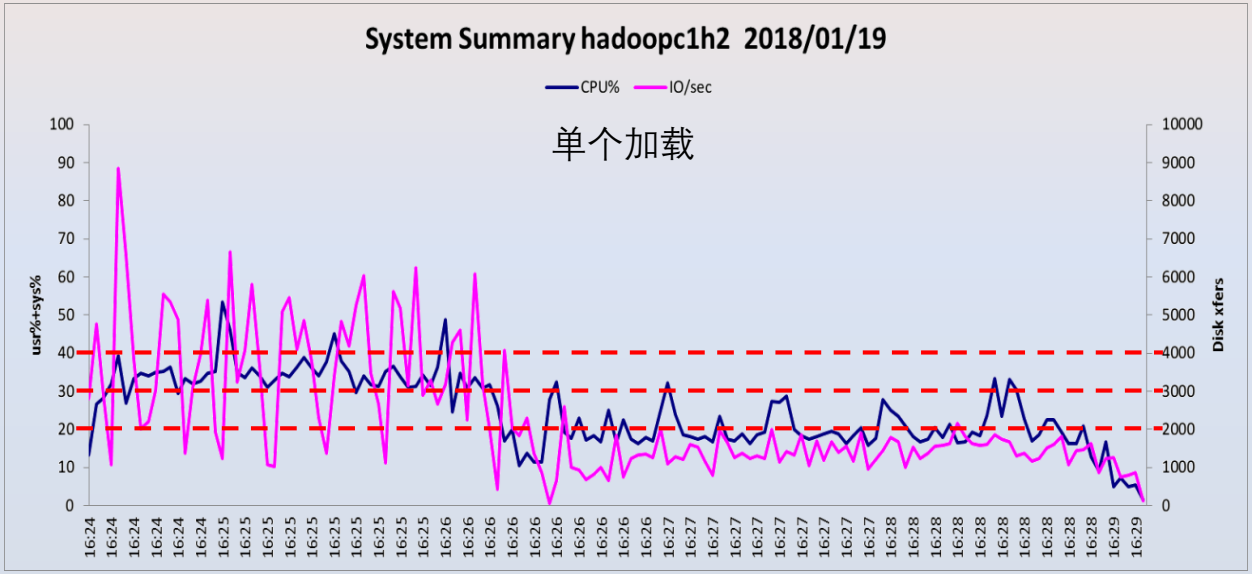
80
MB/s/Node

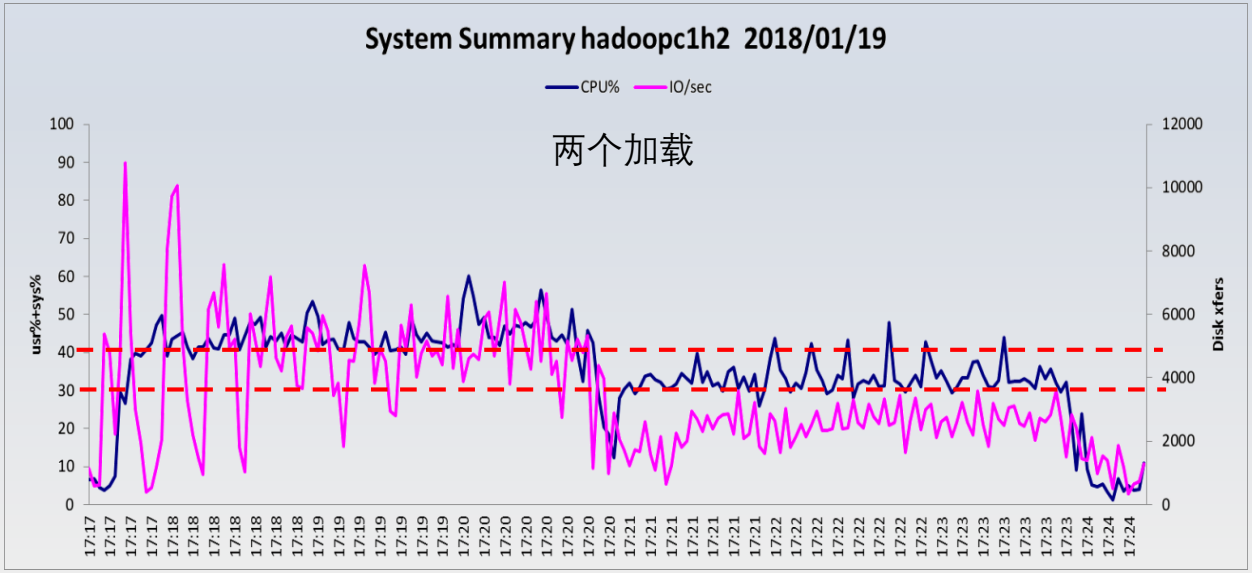
71
MB/s/Node

测试类别	单个加载	两个加载并行	性能提升
Uncompressed	74.3MB/s	112.6MB/s	基准
Snappy	71.1MB/s	101.5MB/s	- 5%
Uncompressed & Pack-fields	80MB/s	124MB/s	+ 8%
Snappy & Pack-fields	80.8MB/s	118.9MB/s	+ 8%

1. 当前Carbon入库时，在排序阶段，CPU已经接近瓶颈。
2. 启用压缩后，CPU更是瓶颈，但此时加载过程中数据写入量降低约1/3；
3. 使用Pack-nosort-fields后，由于去掉了中间的序列化和反序列化过程，CPU稍微有些减轻。
4. 总地来说，Snappy & Pack-fields表现最好。

入库优化#4 – 问题引入





现象：

当前加载的CPU资源约束为41.7%(20核)。

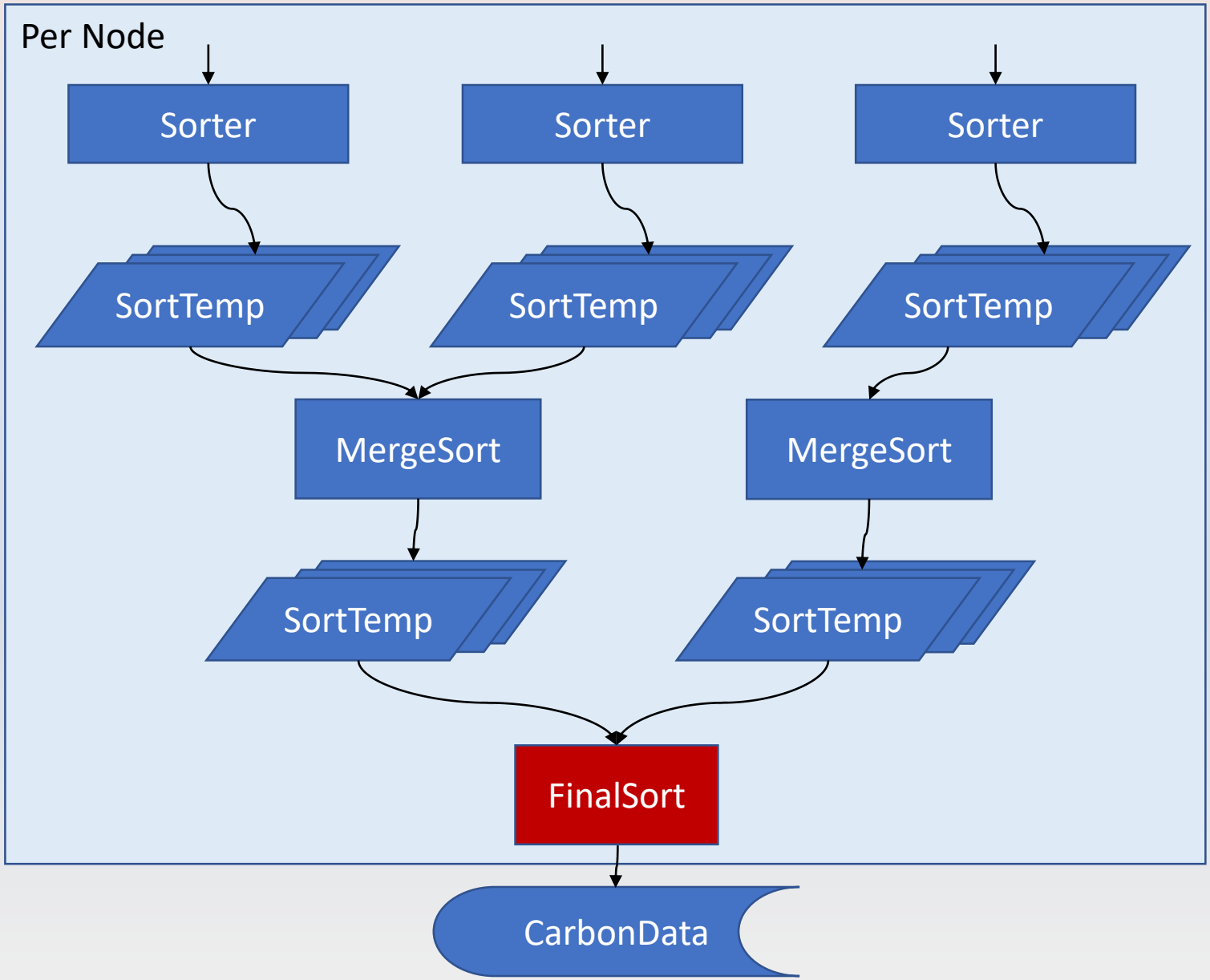
单个加载时，后半段的CPU利用率不到20%；

两个加载同时进行时，提高了后半段的利用率，所以加载性能得到了提升。但此时CPU仍未完全利用。

问题：

入库时，后半段的CPU利用率不高

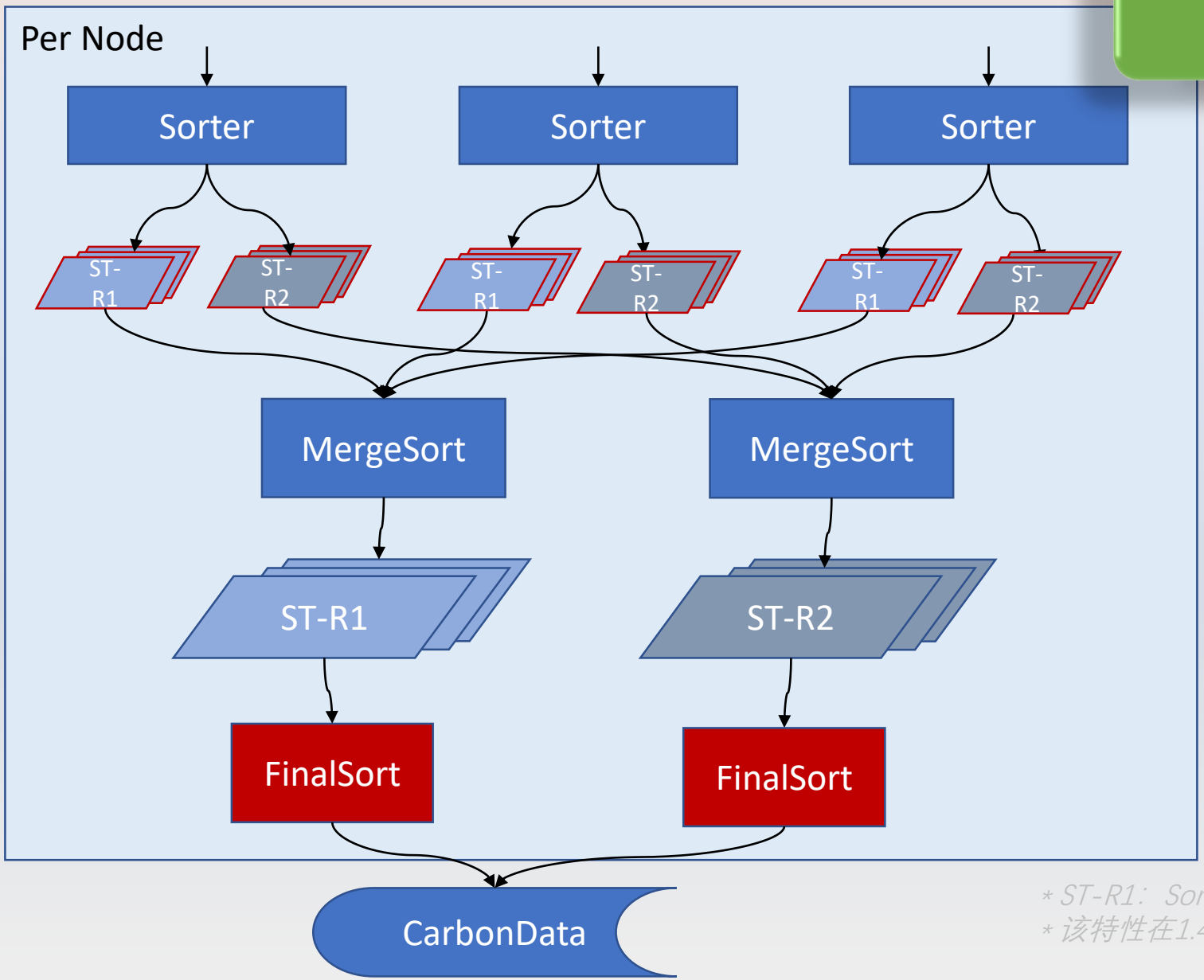
入库优化#4 – 原理与根因



Single Thread

入库优化#4 – 优化实施(1/2)

Sort Column Bounds



划分区间

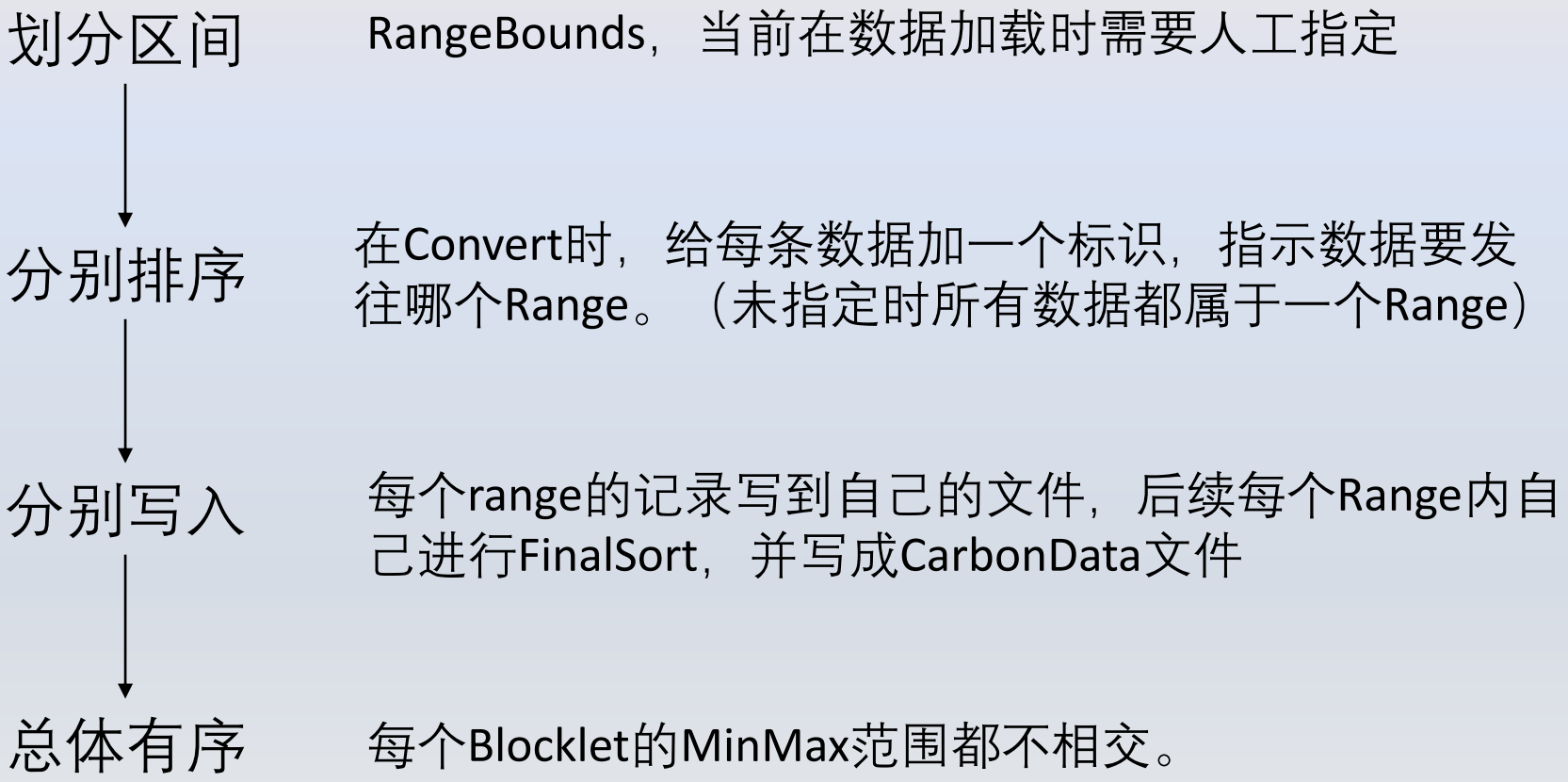
分别排序

分别写入

总体有序

* ST-R1：SortTemp in Range 1
* 该特性在1.4.0版本引入

入库优化#4 – 优化实施(2/2)



入库优化#4 – 使用示例

Step1. 建表时指定sort_columns

```
CREATE TABLE details_other_72g (XX xx, begin_time bigint, msisdn string, XX xx)
STORE BY 'carbondata'
 TBLPROPERTIES ('sort_columns'='msisdn,begin_time');
```

Step2. Load数据时指定sort_columns相应的bounds

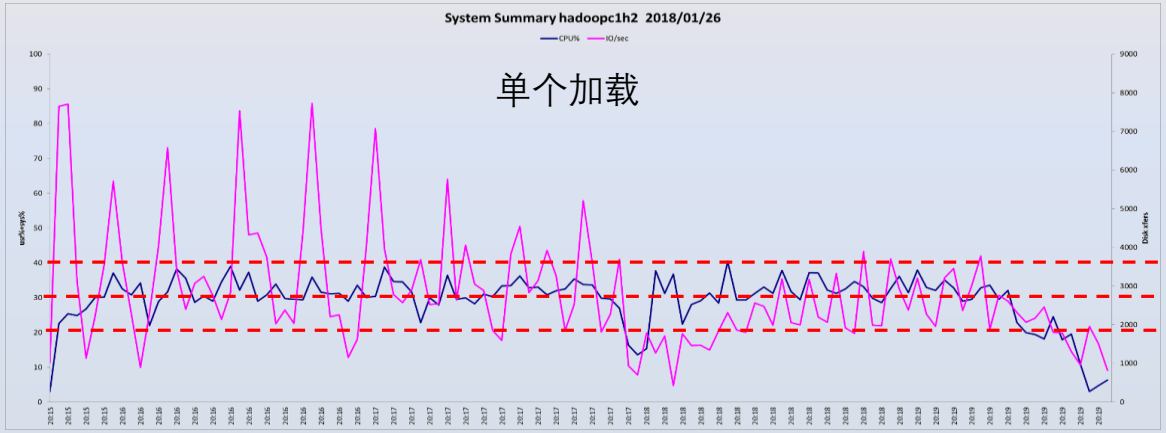
多个bounds以“;”分隔，每个bound内部的column value之间以“,”分隔，按照sort_columns指定的次序排列。最终的range个数是#bounds + 1。

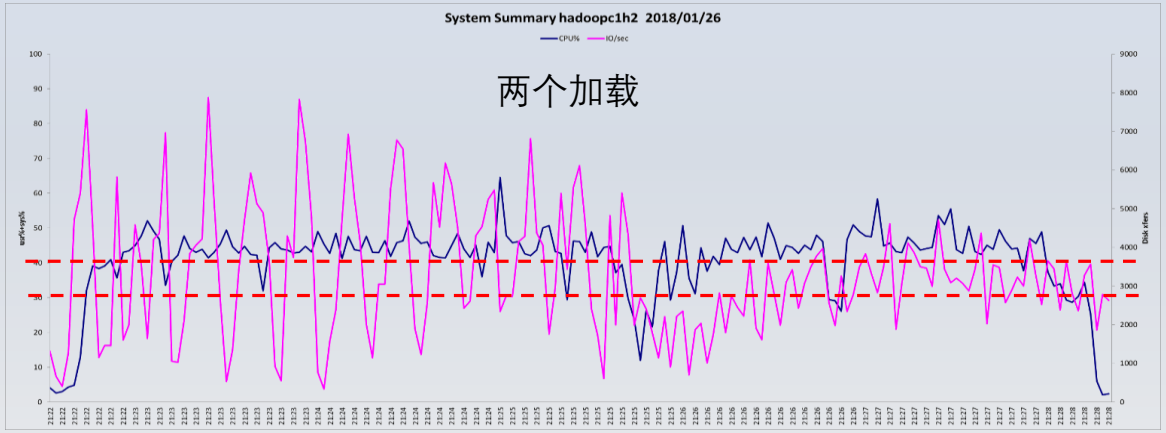
```
LOAD DATA INPATH 'path'  INTO TABLE details_other_72g
   OPTIONS ('sort_column_bounds'='14,00');

LOAD DATA INPATH 'path'  INTO TABLE details_other_72g
   OPTIONS ('sort_column_bounds'='1385,00;1586,00');
```

入库优化#4 – 优化效果

101 MB/s/Node

80 MB/s/Node





类别	单个加载	两个加载	提升
优化前	80.8MB/s	118.9MB/s	基准
优化后	101.6MB/s	140.3MB/s	+25%, +18%

原先三个加载才能将系统资源用满（134MB/s）；现在两个加载就用满了资源（140MB/s）。

注：当前测试时，只指定了一个bounds（2个range）。再增加一个bounds可能还会提高单个加载的性能。

入库优化#4 – 使用场景说明(1/3)

Q sort_column_bounds参数怎么得到?

如果数据分布特征变化不大，则可以人工统计先前的数据分布，得到相应的bounds。

以当前的测试为例，取msisdn（字符串类型）的值进行排序，找到min/max，然后进行试探，最终确定以'14'为界，可以将数据大约均分为两部分(4.0kw v.s. 4.8kw)。

如果对自己的数据分布比较了解，例如sort_column是号码的逆序，则可以直接用'5'作为分界将数据均分为两部分。

A

Q 如果手动指定的sort_column_bounds把数据分布的不均衡时，会怎样?

即使数据分布不均衡，和当前相比，数据加载性能不会比当前的差。

假设一种极端情况：我们指定的bounds将数据都分到一个分区里面了，另一个分区是空的，此时和不指定sort_column_bounds的情况是一样的——也就是说，数据加载的性能不会变差，和不指定该参数时的加载性能保持一致。

如果指定了多个bounds，每个bounds间数据分布不均衡，效果也是类似的：有很少数据的range可以认为该range不存在，所以对加性能不影响，至少不会变差。

A

入库优化#4 – 使用场景说明(2/3)

Q 该设置多少个sort_column_bounds为好，是不是越多越好?

设置bounds后会提高数据加载的并行度，建议设置1~3个bounds最好（对应把数据分到2~4个分区中）。

Bounds不能设置太多，否则会造成CPU资源的竞争，效果可能会变差。

原因是在Carbon中，为了应对数据分布不均的情况，每个分区内都使用的是全部的CPU（loading cores）资源，当分区太多时，CPU争抢严重会导致最终性能变差。 **A**

Q 能不能让carbon为我生成好sort_column_bounds?
这样就不用我自己去指定这个参数了。

当前Carbon中还没准备去做这个事情。

如果要让carbon生成sort_column_bounds，则需要对数据进行采样。如果加上数据采样的耗时，从数据加载的端到端来看，该sort_column_bounds的特性对加载的提升就很小了。做了之后得不偿失。

另外，如果数据分布特征变化不大，就没必要每次浪费时间去做这个采样了。前面也说过，数据在各bounds间分布不均时，和不开启该特性相比，最终效果依然有提升。 **A**

入库优化#4 – 使用场景说明(3/3)

Q 该特性有什么副作用？有没有什么使用约束？

建议在处理大数据量时使用该特性，小数据量时可以不使用。

使用bounds后，会把数据分区，然后并行处理，如果每个分区中的数据量很小，则会导致最终生成的文件数扩大N倍。既给HDFS增加负担，又影响Carbon查询的性能。但对大数据量来说，文件数就基本没有变化。

简单来说，不开启该特性时，假设生成的文件个数为NF，节点数为NN，bounds个数为NB，当NF/NN/(NB + 1) > 1时，开启该特性才不会使文件个数变多。

最后，该特性主要提升单个加载的CPU利用率以提升加载性能。如果队列当中同时有多个（>3）加载并行执行，并且CPU利用率已经很高了，则可以不使用该特性。

A

Q Anything else?

关于该特性，还有些场景需要考虑和完善。
1. 字典列做sort_column的情况。（支持，但由于字典的字面值和实际值次序不一定一致，所以可能存在数据倾斜的情况。）
2. 存在BucketColumns列的情况，以及Bucket列做sort_column的情况。（很可能不支持）
3. 存在分区Partition By的情况。（可以参考该特性的实现，优化带分区的数据加载）

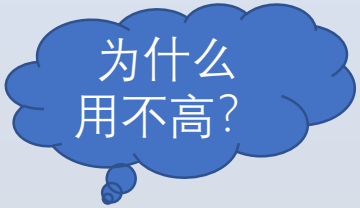
还有些特性不应该生效的场景：
NoSort/BatchSort/GlobalSort的场景（仅支持LocalSort）
和BloomFilter DataMap共用的场景，还未验证过。

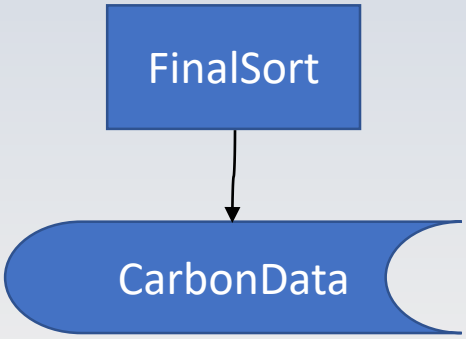
A

入库优化#5 – 问题引入

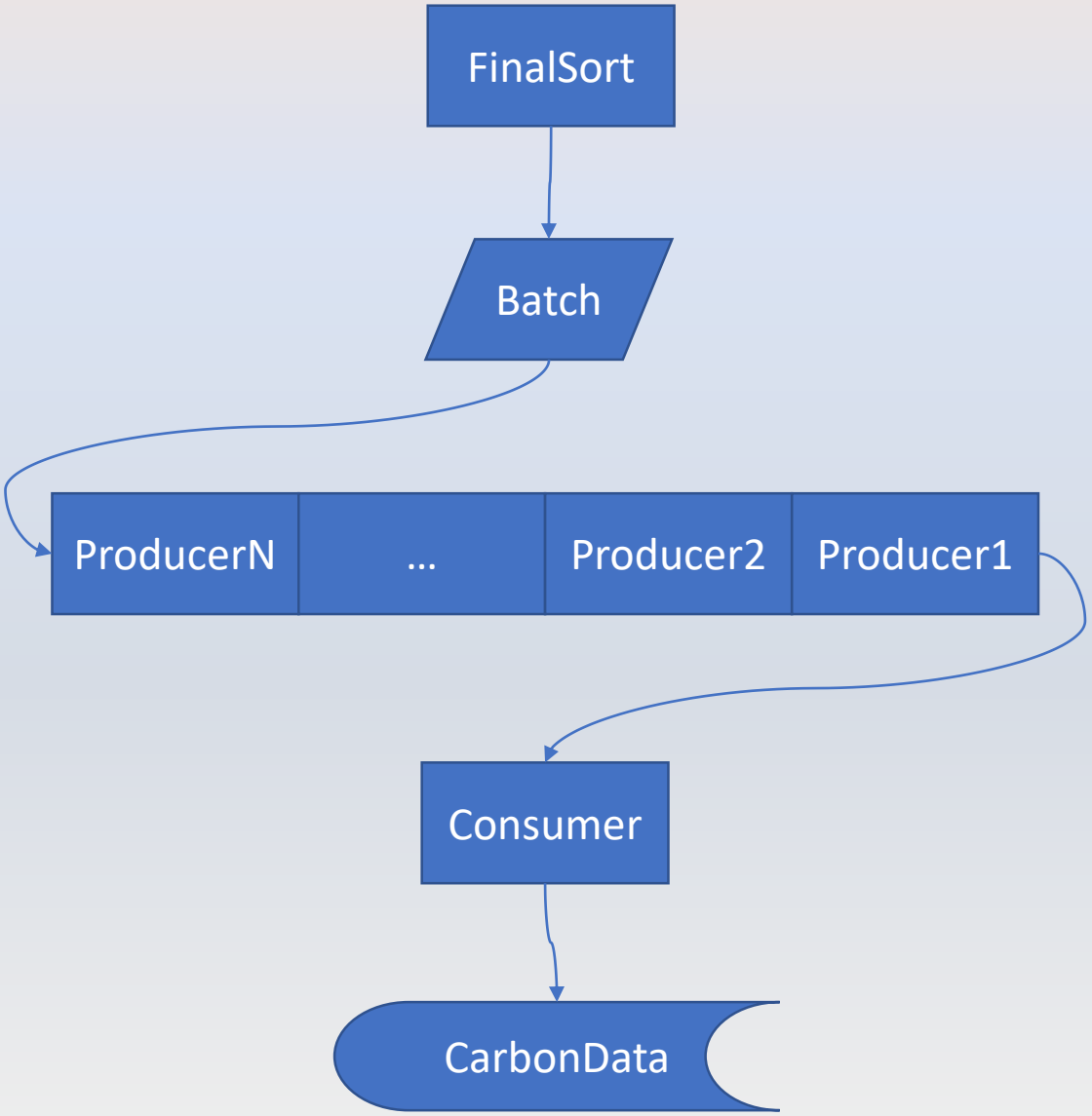
入库后半段CPU使用率不高

Sort_column_bounds特性：使final_sort并行化，提高了CPU的利用率。





入库优化#5 – 原理与根因(1/2)



用Pull From Heap的方式进行排序，后端不断取当前最小的记录

一般32000行凑足一个Batch

一个Producer异步处理一个Batch，进行列式存储、编码、压缩，生成一个Page，然后放到一个环形队列。

不断从队列中取Page，然后写Blocklet，再落地成文件。

** N = #loadingCores*

入库优化#5 –原理与根因(2/2)

1. Time wait to start a new Batch: 30ms

2. Time wait to acquire a Position for Producer: 0ms

3. Time consumed to execute Producer (produce a TablePage): 110ms

4. Time wait to retrieve a TablePage: 30ms

5. Time consumed to consume a TablePage: 10~30ms

入库优化#5 – 优化实施

Final Sort Prefetch

根因：

✓ 全排序时，后端采用pull方式获取记录太慢

解决方法：

✓ 后端pull记录时使用prefetch

Step1：构造两个Buffer，一个为working buffer，一个为backup buffer；
Step2：当使用working buffer时，数据异步填充到backup buffer；
Step3：当working buffer耗尽时，交换两者角色。继续Step2。

使用方式：

✓ 针对入库，该特性尚未开发。

✓ 在1.5.1引入，针对数据紧缩Compaction已开发该特性。当前由参数carbon.compaction.prefetch.enable 控制，默认不开启。每次Prefetch的量由参数carbon.detail.batch.size控制，默认为100，仅compaction时，推荐配置32000，以实测为准。

入库优化#5 – 优化效果

Code Branch	Prefetch	Batch Size	Load1 (s)	Load2 (s)	Load3 (s)	Compact 3 Loads (s)	Perf Enhanced
master	NA	100	447.4	445.9	450.1	661.3	Base Line
master	NA	32000	441.5	454.4	456.8	641.2	+ 3.10%
PR2906	enable	100	445.3	450.2	445.3	411.8	+ 60.60%
PR2906	enable	32000	438.7	446.8	441.8	333.1	+ 98.50%
PR2906	disable	100	458.1	459.4	450.9	659.5	+ 0.30%
PR2906	disable	32000	472.0	446.8	457.1	654.5	+ 1.00%

3 Huawei ECS instances as workers each has 16 cores and 32GB.
3 Spark executor each uses 12 cores and 24GB.
Using 74GB LineItem in 100GB TPCH as test data.

入库优化#5 – 优化效果

Code Branch	Prefetch	Batch Size	Load1 (s)	Load2 (s)	Load3 (s)	Compact 3 Loads (s)	Perf Enhanced
master	NA	100	147.4	142.3	144.6	201.4	Base Line
master	NA	32000	140.8	138.7	141.6	196.2	+ 2.70%
PR2906	enable	100	143.9	142.5	146.2	99.9	+ 101.60%
PR2906	enable	32000	142.1	139.3	136.9	98.3	+ 104.90%
PR2906	disable	100	146.7	137.4	139.6	200.6	+ 0.40%
PR2906	disable	32000	145.2	145	139.7	195.7	+ 2.90%

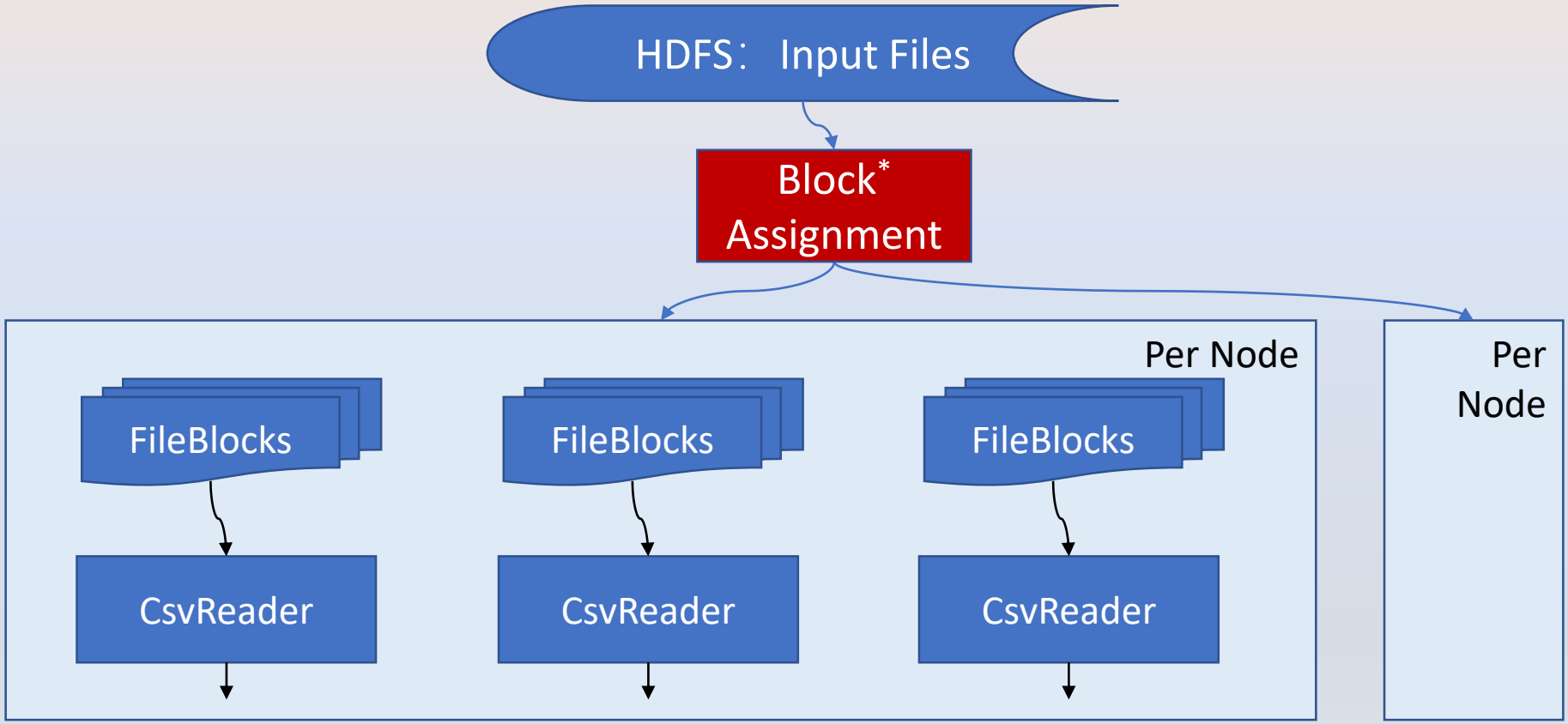
1 Huawei RH2288 with 32 cores and 128GB.
1 Spark executor using 30 cores and 90GB.
Using 7.3GB LineItem in 10GB TPCH as test data.

入库优化#6 – 问题引入

1. **G产品产生的数据文件大小差异很大**
   - 以某典型场景中为例，输入数据146GB，文件大小1KB~5GB，中位数不到1MB。
   - 数据入库时，某个节点处理67.8GB的数据，而另一个节点处理12.1GB数据，数据量差距近5倍。
   - Carbon单节点入库性能近41MB/s

2. **S产品部分表单次入库<1GB数据，而集群规模上百台**
   - 入库数据相对于集群规模太小，单节点处理数据较少。
   - 每个节点上产生一个小文件。
   - LocalSort范围太小，排序效果不明显，导致查询性能也比较差。

入库优化#6 – 原理与根因



当前仅让每个节点处理相近数量的FileSplit，而每个split对应的Size差别很大。-- NumBasedAllocation

* 这里的Block指的是一个FileSplit

入库优化#6 – 优化实施

Size Based Block Assignment

根因：

- ✓ NumBasedAllocation 在数据倾斜的场景下存在问题。

解决方法：

- ✓ 增加 SizeBasedAllocation，使得每个节点处理的数据总量接近，解决输入数据倾斜的问题。该策略同时遵循数据本地性。
- ✓ 增加 MinSizeBasedAllocation，限制单个节点处理数据的最小量。解决‘小数据大集群’的问题。
- ✓ 对于 MinSizeBasedAllocation，Carbon在给各个节点分配 Split 时，*会忽略数据本地性*，否则某些节点一直会很忙。

使用方式：

- ✓ SizebasedAllocation 在1.4.0中引入该特性，默认关闭
- ✓ MinSizeBasedAllocation 在1.5.1中引入该特性，默认关闭

入库优化#6 – 使用示例

1. 启用 SizeBasedAllocation

- ✓ 默认不启用该特性
- ✓ 将系统参数 carbon.load.skewedDataOptimization.enabled 置为true以启用特性。

2. 启用 MinSizeBasedAllocation

- ✓ 建表时指定TBLProperties：LOAD_MIN_SIZE_INMB=1024，或
- ✓ LOAD数据时指定Options： LOAD_MIN_SIZE_INMB=1024
- ✓ 默认值0表示不启用该特性。
- ✓ 当指定的值太小时，内部会走 SizeBasedAllocation 的策略

入库优化#6 – 优化效果

启用 SizeBasedAllocation 后，

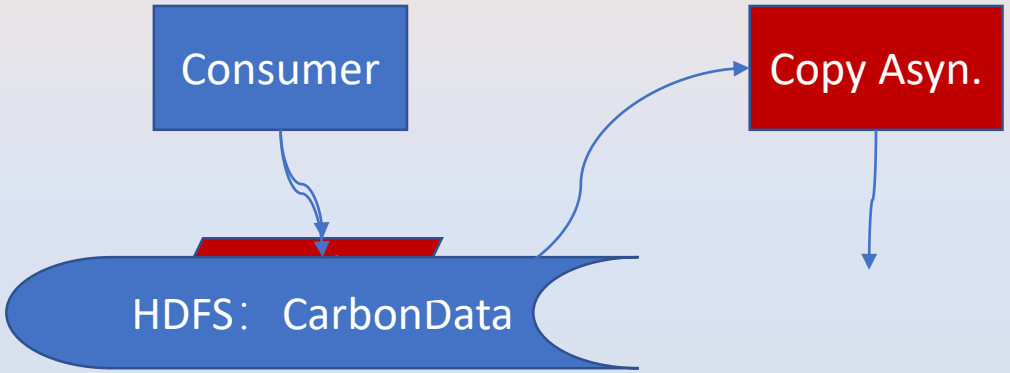
✓ G业务数据入库性能40MB/s → 60MB/s

✓ 对于无数据倾斜的S业务，入库性能基本没有影响

启用 MinSizeBasedAllocation 后，

✓ 尚未有实测数据

入库优化#7 – 问题与优化



Direct Write HDFS

在生产环境中，IO 资源非常宝贵

问题：

数据先写到本地盘，然后再异步拷贝到HDFS上，多了I/O开销。

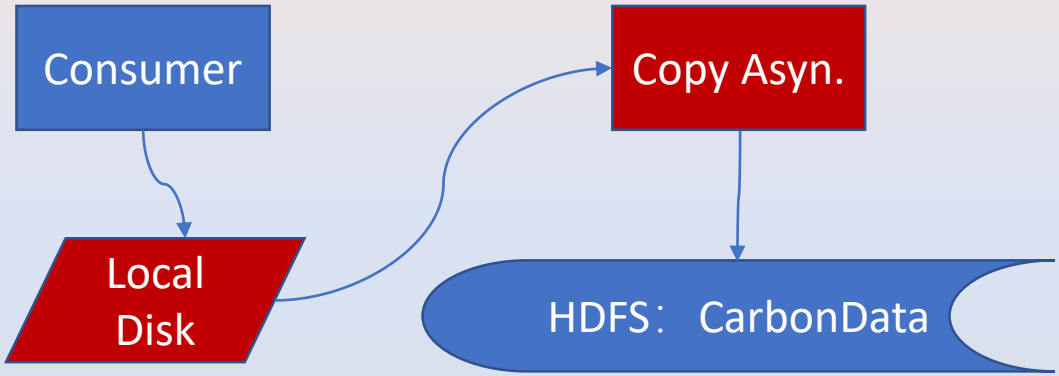
解决方法：

数据直接写到HDFS上

使用方式：

- ✓ 1.4.0中引入该特性，由参数 carbon.load.directWriteHdfs.enabled 控制，默认关闭
- ✓ 在1.5.0中为支持S3，将参数名修改为 carbon.load.directWriteStorePath.enabled，默认关闭

入库优化#7 – 问题与优化



Direct Write HDFS

在生产环境中，Disk资源非常宝贵

问题：

数据先写到本地盘，然后再异步拷贝到HDFS上，多了I/O开销。

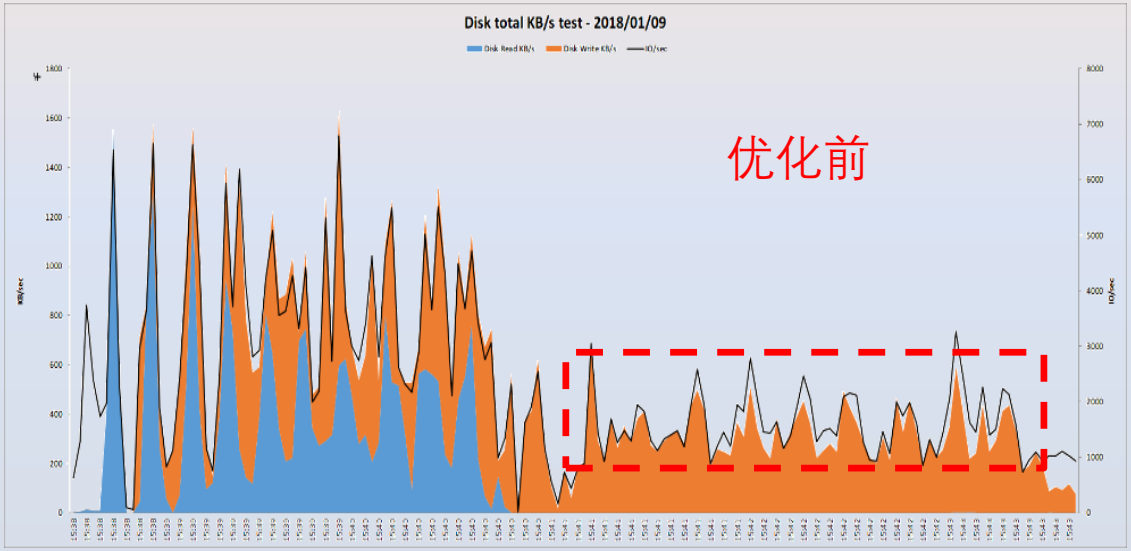
解决方法：

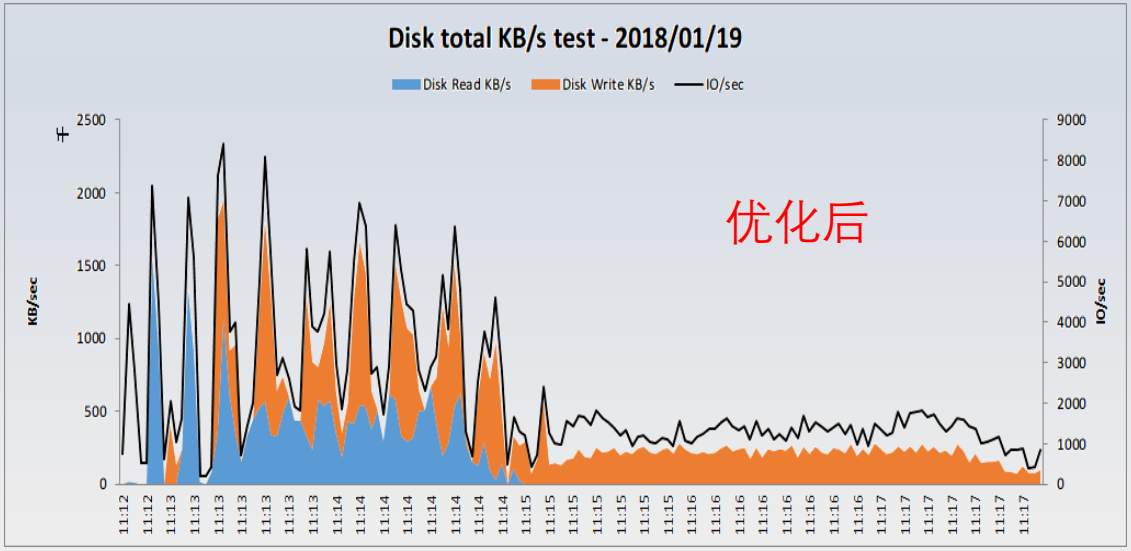
数据直接写到HDFS上

使用方式：

- ✓ 1.4.0中引入该特性，由参数 carbon.load.directWriteHdfs.enabled 控制，默认关闭
- ✓ 在1.5.0中为支持S3，将参数名修改为 carbon.load.directWriteStorePath.enabled，默认关闭

入库优化#7 – 优化效果

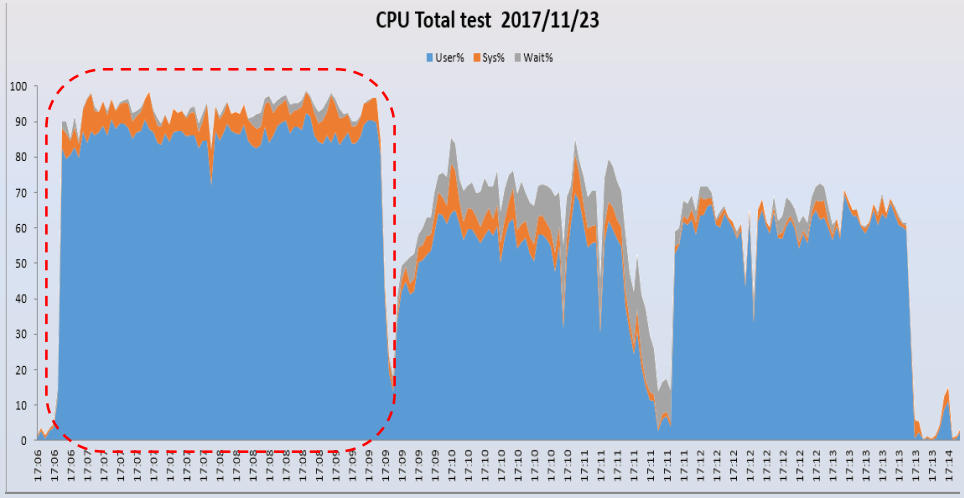


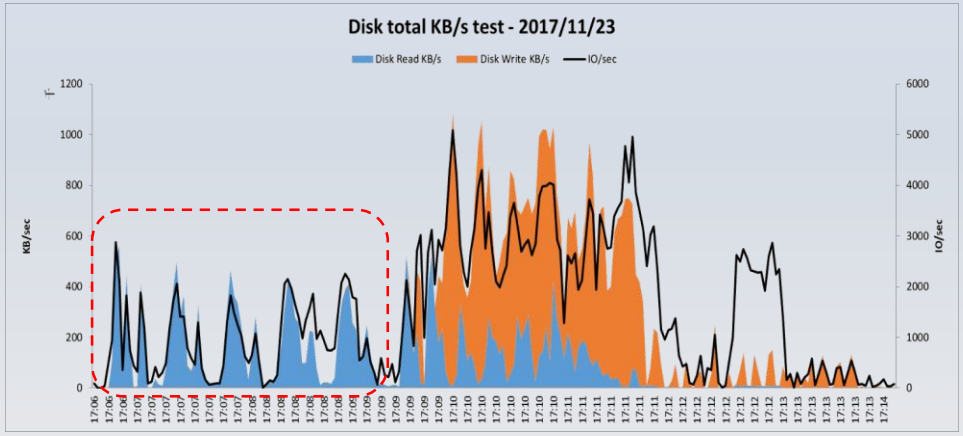


端到端Disk Write减少 11.4%

入库性能基本没有影响

入库优化#8 – 问题引入





全局字典生成的太慢:

生成全局字典需要2.9min;

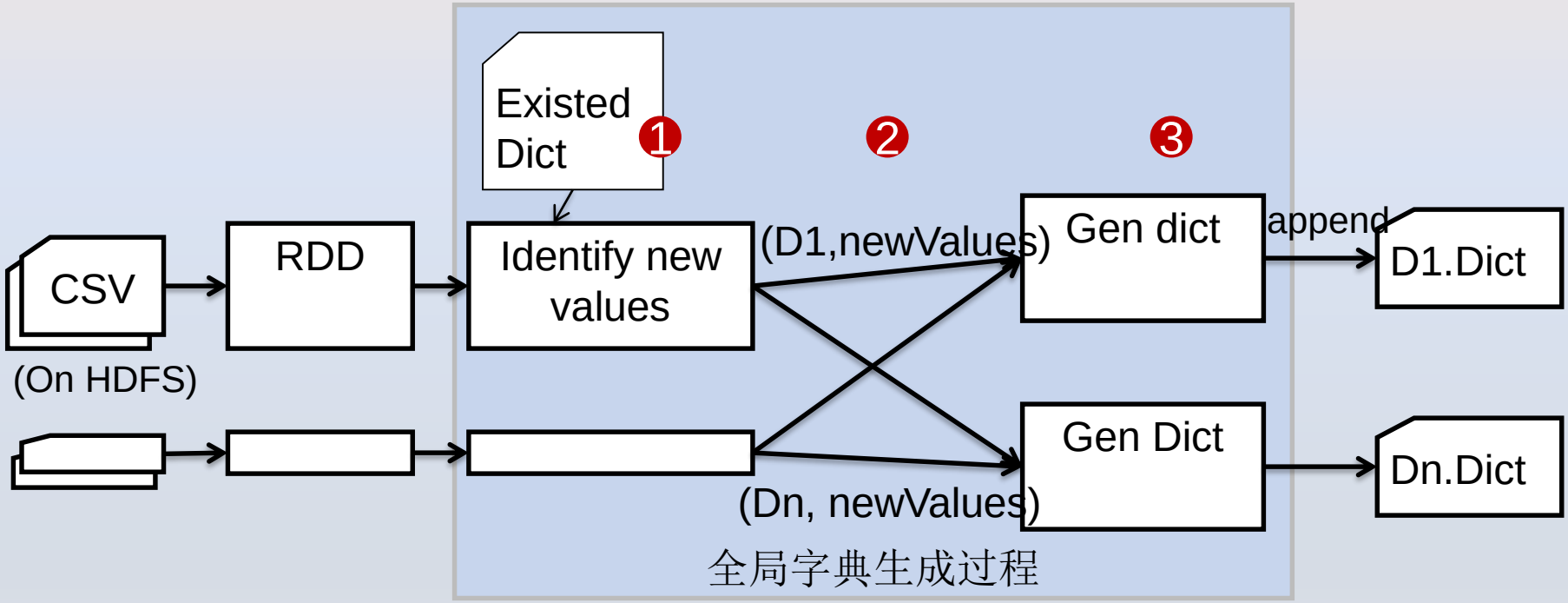
生成最终数据需要4.6min

测试数据:

CSV 35.8GB,130 million records, 200 fields;

3 dictionary column with distinct value 4, 9, 19213;

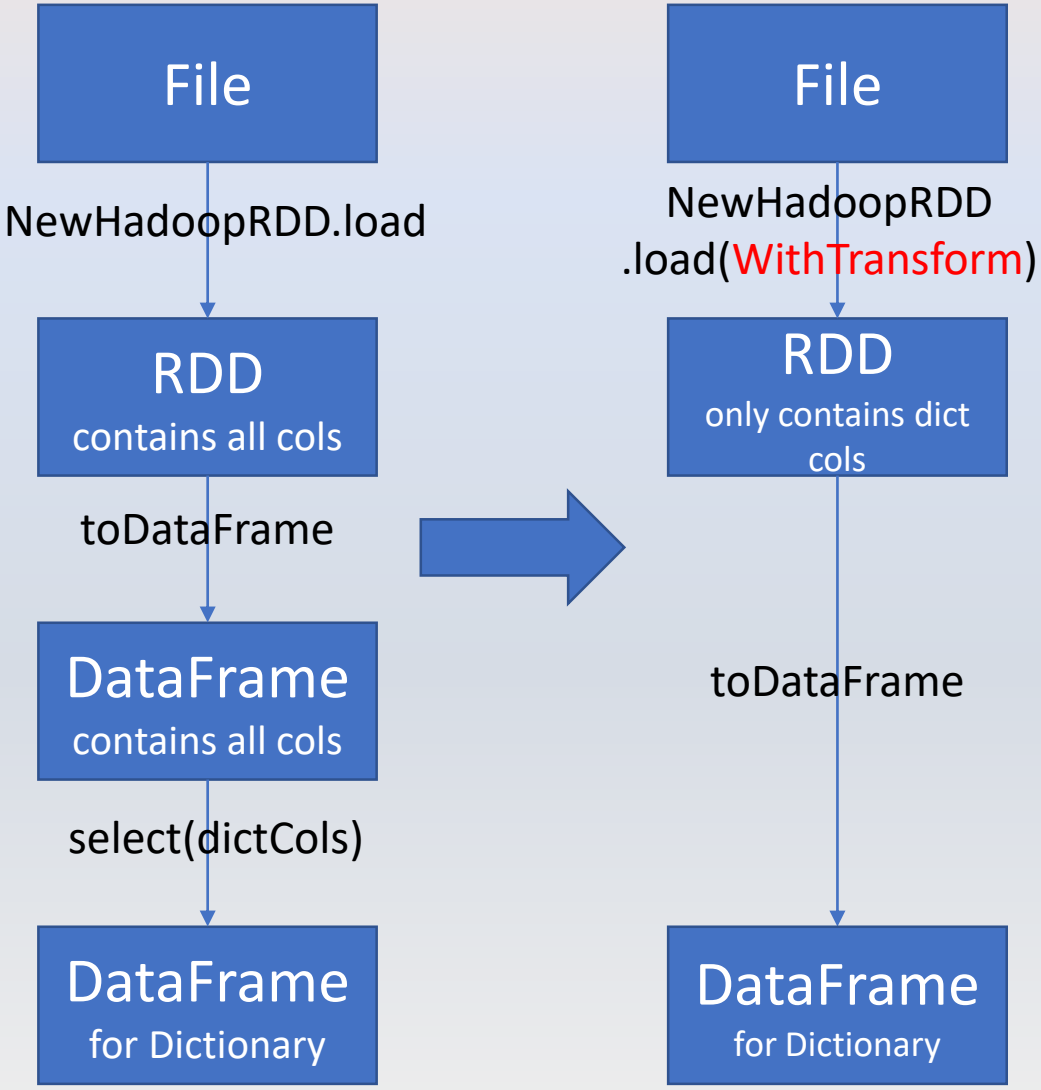
入库优化#8 – 原理



1. 从输入数据中读取要做字典的列，然后装载先前加载时生成的字典文件，通过比较，得到每个字典列新增的distinct值；
2. 所有字典列的distinct值按照字典列进行shuffle，以便后续每个task仅处理一个列的字典；
3. 每个task对新增的distinct值进行字典编码，并将新值写入到字典文件中。

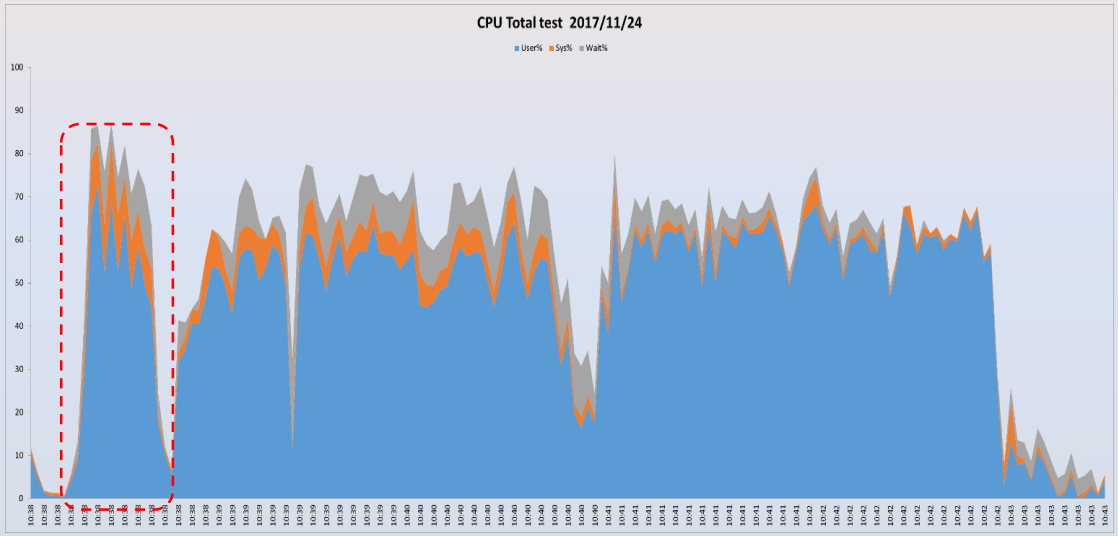
入库优化#8 – 根因与修改

Dictionary Optimization

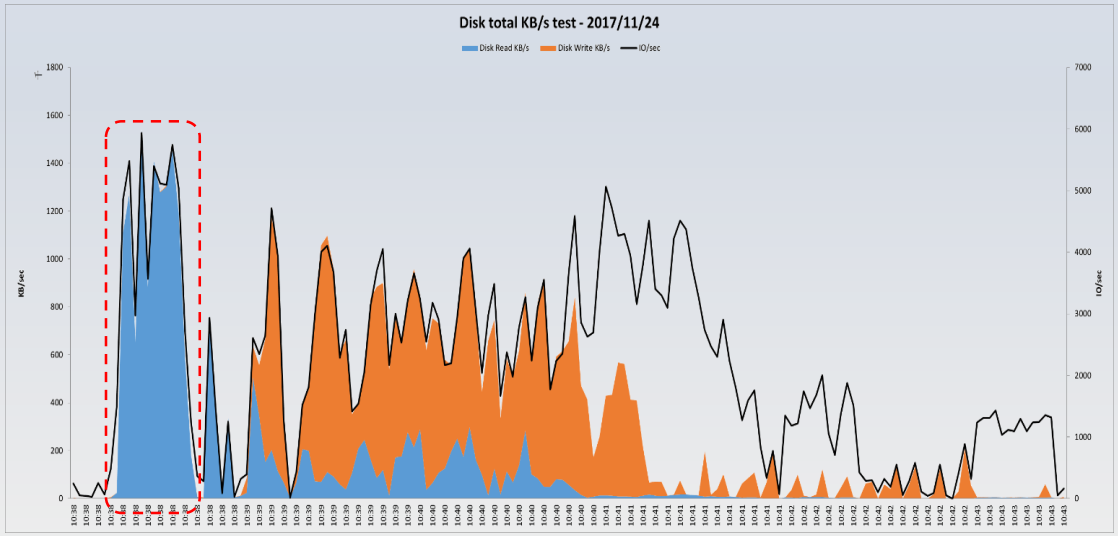


在加载数据时就进行“列裁剪”

入库优化#8 – 优化效果



	生成字典	全流程
优化前	174s	454.9s
优化后	29s	311s



入库优化#9 – 背景

Column Compressor

Local Dictionary	Compressor	Parquet (MB)	CarbonData (MB)
Yes	Uncompressed	501	×
	GZip	202.1	×
	Snappy	280.5	335
	Zstd	190	207
No	Uncompressed	-	×
	GZip	232	×
	Snappy	406	375
	Zstd	220	225

** 使用 1.2GB 原始CSV数据，使用 carbon 1.4.1版本，使用parquet 1.x版本验证*
** X：暂不支持*

入库优化#9 – 使用示例

系统级别设置

- ✓ 控制参数为 carbon.column.compressor, 支持 snappy 和 Zstd, 默认为 snappy。
- ✓ 可支持不同入库批次使用不同的compressor。

表级别设置

- ✓ TBLPROPERTIES(‘carbon.column.compressor’=‘snappy’)，支持 snappy和Zstd，默认为snappy。
- ✓ 如果表级别未设置，则默认使用系统级别的设置。

** 在 carbon 1.5.0版本中引入对Zstd支持*

入库优化#9 – 使用效果

数据压缩效果：压缩率更高

Compressor	1 Load (GB)	Compact 6 Loads (GB)
Origin	72.1	72.1 * 6
Snappy	18.3	47.6
Zstd	11.2	21.3

Compaction影响：没有影响

Compressor	Compact 6 Loads (s)
Snappy	1964.3
Zstd	1888.2

数据入库影响：没有影响，降低IO

SortTemp Compressor	Compressor	1 Load (s)
Uncompressed	Snappy	194.0
Uncompressed	Zstd	195.5
Snappy	Snappy	192.0
Snappy	Zstd	197.1
Zstd	Snappy	186.7
Zstd	Zstd	187.8

查询影响：

使用Zstd后，查询性能有所降低。尚未进行严格测试，和具体查询扫描的数据量相关。

推荐使用：

优先使用Zstd+Snappy；对于长期存储的数据，推荐使用Zstd+Zstd。

入库优化#9 – 自定义Compressor

```
// TestLoadDataWithCompression.scala
class CustomizeCompressor extends Compressor {
  override def getName: String =
"org.apache.carbondata.integration.spark.testsuite.dataload.Customize
Compressor"
  override def compressByte(unCompInput: Array[Byte]): Array[Byte] =
unCompInput
  override def compressByte(unCompInput: Array[Byte], byteSize: Int):
Array[Byte] = unCompInput
  override def unCompressByte(compInput: Array[Byte]): Array[Byte] =
compInput
  override def unCompressByte(compInput: Array[Byte], offset: Int,
length: Int): Array[Byte] = compInput
  // other codes
}
```

使用方式:

- ✓ Carbon 1.5.1 中引入该功能。
- ✓ 直接使用，不用注册。
- ✓ 使用类名作为 compressor 名，carbon.column.compressor= org.apache.carbondata.integration.spark.testsuite.dataload.CustomizeCompressor

入库优化 – 优化总结

Multiple Temp Dir

Compress Sort Temp

Pack No-Sort Fields

Sort Column Bounds

Final Sort Prefetch

Customized Block Assignment

Direct Write HDFS

Dictionary Optimization

Column Compressor

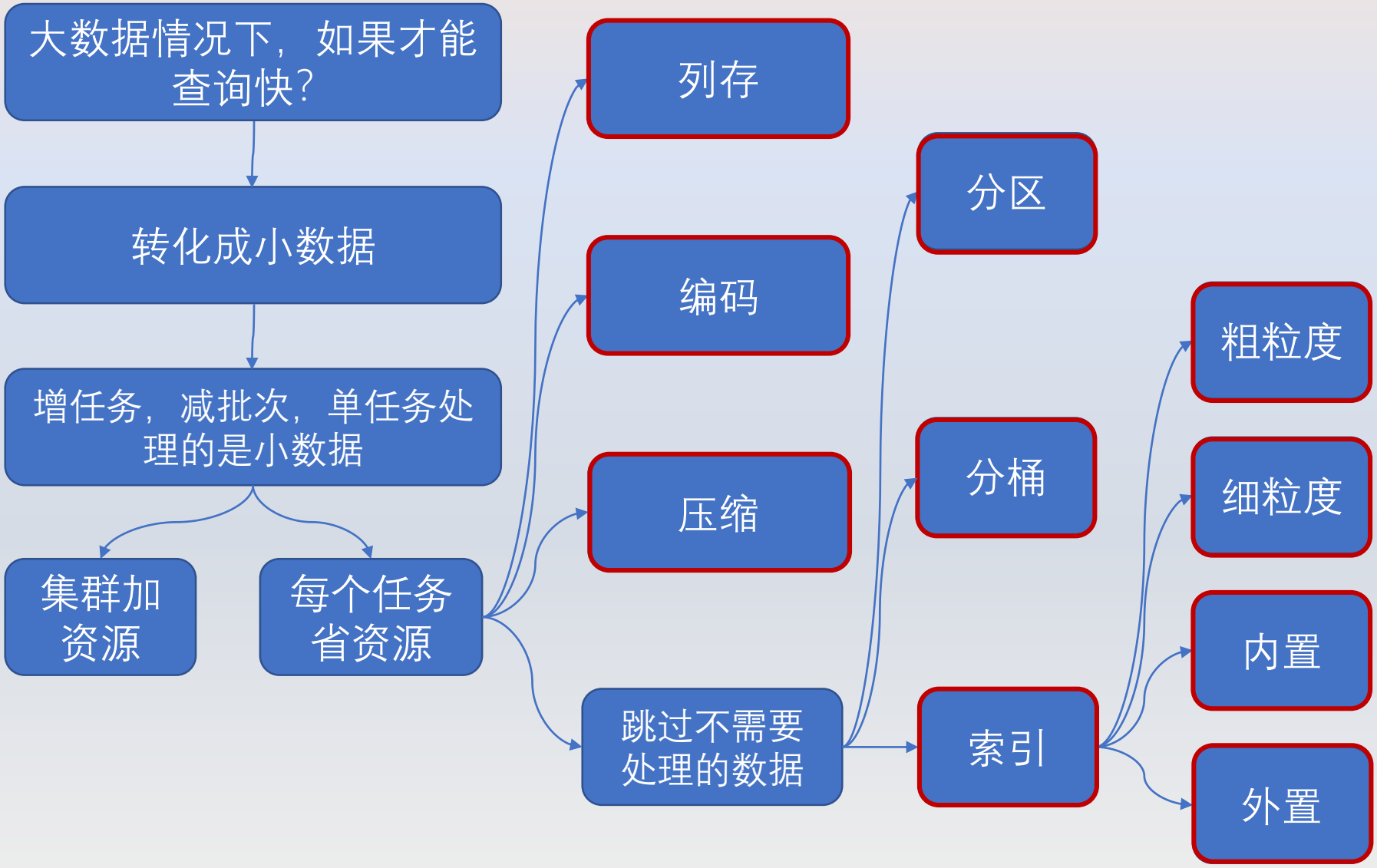
由参数控制，需根据业务场景调整

入库优化 – 其他影响项

Sort Scope	不同的scope会走不同的入库流程
Unsafe Memory	减少GC。不足时可能会出错。涉及多个参数配置。 1.5.1中引入offHeap自动回退到onHeap的机制。
In-memory Merge Sort	减少mergeSort文件落地，涉及多个参数配置。
Local Dictionary	本地字典会消耗更多资源，如果超过阈值会fallback成不用字典，影响入库性能。

查询优化

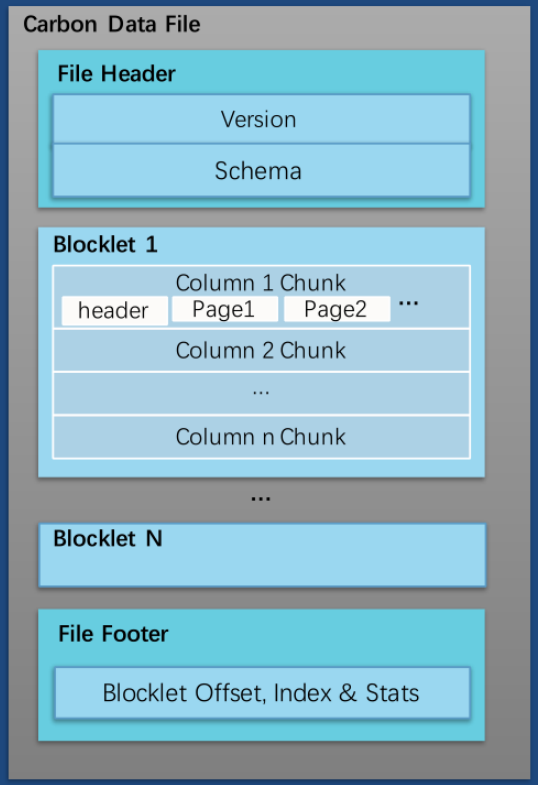
查询优化 – 回顾


大数据情况下，如果才能查询快?
转化成小数据
增任务，减批次，单任务处理的是小数据
集群加资源
每个任务省资源
列存
编码
压缩
跳过不需要处理的数据
分区
分桶
索引
粗粒度
细粒度
内置
外置

概念回顾

CarbonData File文件格式

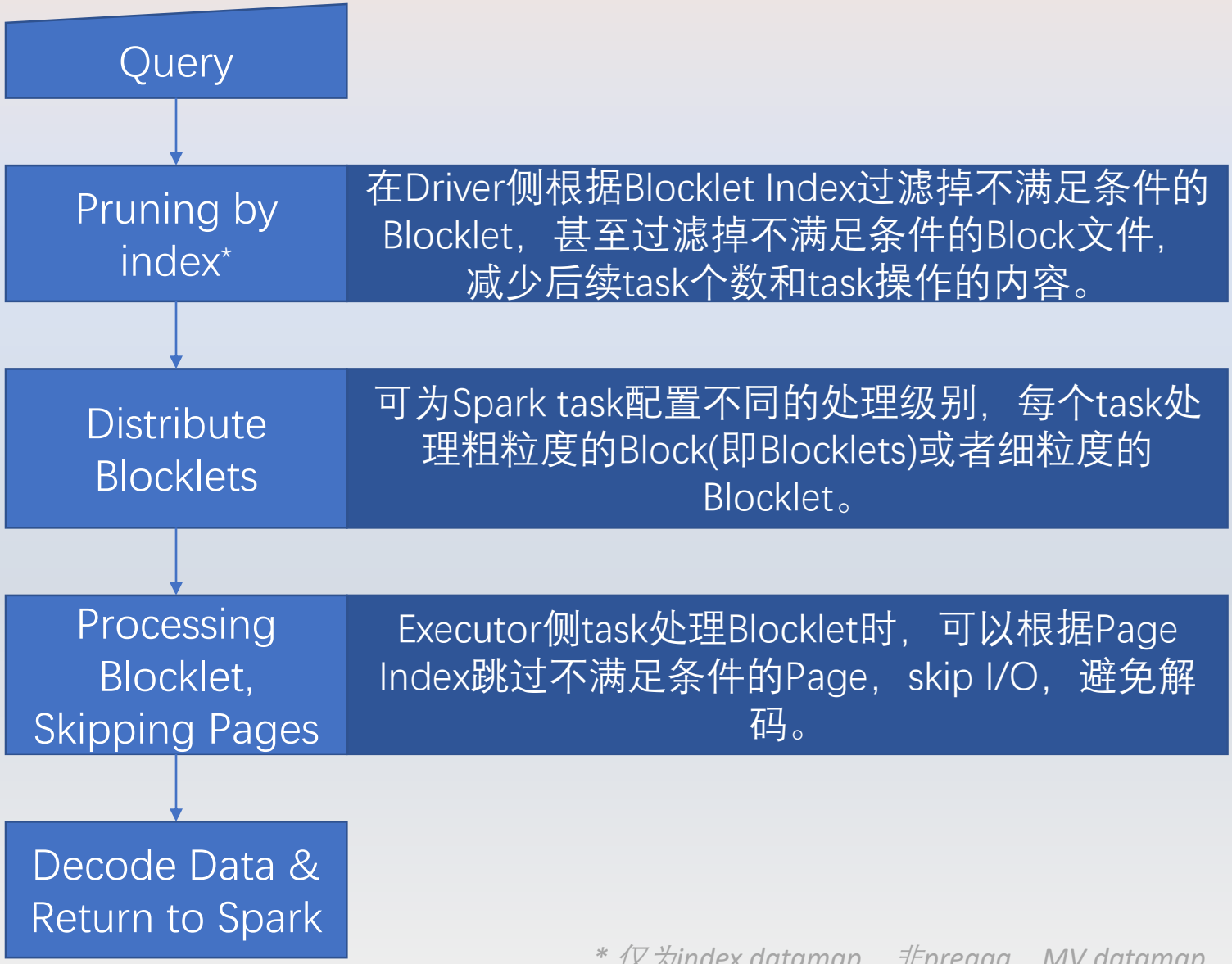
- 数据布局
  - Block：一个HDFS文件
  - Blocklet：文件内的列存数据块，是最小的IO读取单元
    - Column chunk: Blocklet内的列数据
    - Page：Column chunk内的数据页，是最小的解码单元
- 元数据信息
  - Header：Version，Schema
  - Footer： Blocklet Offset, Index & 文件级统计信息
- 内置索引和统计信息
  - Blocklet索引：B Tree start key, end key
  - Blocklet级和Page级统计信息：min，max等



** 来源于'CarbonData技术原理及使用介绍-蔡强'-2018.09.08 北京meetup*

** 在1.5.1版本中已经去掉了Btree，只有StartKey和endKey。*

查询流程



** 仅为index datamap，非preagg、MV datamap*

Index – Main/Default DataMap

含每个 Blocklet 每个列的 MinMax 信息	内置&外置的粗粒度索引，在 Driver 侧过滤掉 Blocklet 或者 Block。
含 Blocklet 中每个 Column Page 的 MinMax 信息	内置的细粒度索引，在 Executor 侧过滤掉 Blocklet 中的Page，skip I/O & decode
支持 LRU Cache	加快多次查询的性能
支持 Distributed DataMap Pruning	将 Driver 侧的过滤(Pruning)行为发往各Executor执行，然后再收集执行结果，减轻Driver压力。

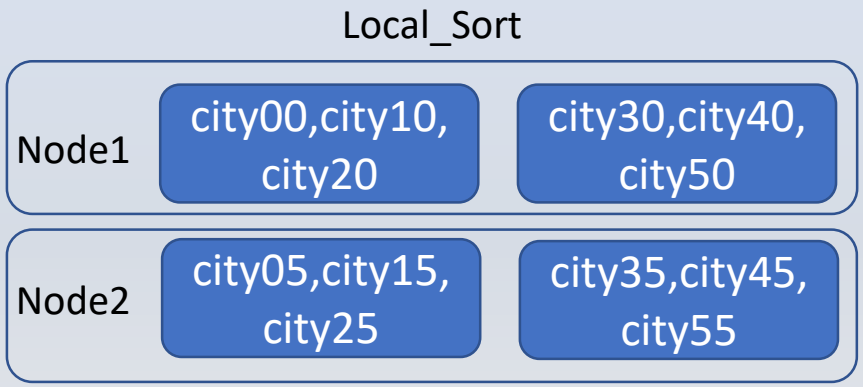
查询优化 – 问题引入

仅少数列能从SortColumns中受益

越靠前的列作用越明显，当几个列之间的数据分布没有相关性时，后面的排序列几乎没有用作。

SortScope影响查询的性能

Local_Sort可能导致同一个排序区间在每个节点上都存在。当SortColumns生效时，可能每个节点上都要起task进行查询。



很多Segment时也影响查询性能

每个segment的SortColumns范围差不多，导致每个segment都要起task进行查询。

查询优化 – BloomFilter索引

BloomFilter

- ✓ 用较少的空间来判断某个元素是否可能在某个集合中;
- ✓ 构造*k*个哈希函数，对某个元素，在长度为*m*的bits中标识*k*个位置。查询时，用同样的方式找*k*个位置，如果该*k*个位置被标识过，则认为该元素可能(*p*)存在，否则一定不存在。

BloomFilter DataMap

- ✓ 对某个索引列，每个Blocklet分别对应一个 BloomFilter，用来指示某个元素是否在该 Blocklet 中。
- ✓ 作用在 Driver 侧，是粗粒度的、外置索引。

BloomFilter DataMap 应用场景

- ✓ 高基数列上的精确查询
  - 低基数列会导致太多Blocklet命中，效果不明显
  - 等值查询：col_id = ‘a’
  - IN 查询：col_id IN (‘a’, ‘b’, ‘c’)

查询优化 – BloomFilter DataMap使用示例

创建

CREATE DATAMAP [IF NOT EXISTS] *datamap_name* ON TABLE *base_table*
USING '**bloomfilter**'
DMPROPERTIES ('**INDEX_COLUMNS**'='*id, name*', '**BLOOM_SIZE**'='*640000*',
'**BLOOM_FPP**'='*0.00001*', '**BLOOM_COMPRESS**'='*true*')

查询时效果

> EXPLAIN SELECT * FROM *base_table WHERE id='5432' AND name='David'*
|== CarbonData Profiler ==
Table Scan on base_table
 - total blocklets: 120
 - filter: (((name <> null and id <> null) and id = '5432') and name = 'David')
 - pruned by Main DataMap
    - skipped blocklets: 99
 - pruned by CG DataMap
    - name: datamap_name
    - provider: bloomfilter
    - **skipped blocklets: 15**
|
|== Physical Plan ==
…

查询优化 – BloomFilter DataMap使用示例

查看

SHOW DATAMAP ON TABLE *base_table*;

临时禁用

SET carbon.datamap.visible.dbName.*tableName*.*dataMapName* = false

删除

DROP DATAMAP [IF EXISTS] *datamap_name* ON TABLE *base_table*;

1. 在1.4.1版本中引入作为正式特性，建议在1.3.0及之后的表上使用。
2. 可以在建表时创建DataMap，也可以在几次数据入库后再根据查询需求创建DataMap。

查询优化 – BloomFilter DataMap效果

213 (e2cc7f23-32b0-4915-90f1-5814feb3f608)	SELECT * FROM CUSTOMER_BLOOM5 where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:02:09	0.2 s	1/1	1/1
212 (c0df006e-6c91-4263-95fc-97eb0706d309)	SELECT * FROM CUSTOMER_BLOOM5 where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:02:00	0.2 s	1/1	1/1
211 (0b419cbf-149f-4fb6-a2ab-5af03431106c)	SELECT * FROM CUSTOMER_BLOOM5 where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:01:52	0.2 s	1/1	1/1
210 (9325be59-1ae9-4db5-9347-c0593d63456e)	SELECT * FROM CUSTOMER_BLOOM5 where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:01:44	0.3 s	1/1	1/1
209 (06d6218b-5263-4b80-b2d9-e5fd0dbc5277)	SELECT * FROM CUSTOMER where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:01:23	1 s	1/1	300/300
208 (17562a0a-c005-4c58-8009-f4ee76e5dfc7)	SELECT * FROM CUSTOMER where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:01:18	1 s	1/1	300/300
207 (4d899ff8-25a5-44b0-aef1-fc5ff1ece89a)	SELECT * FROM CUSTOMER where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:01:15	1 s	1/1	300/300
206 (4c8af567-ef3a-473a-a433-e878e6fba367)	SELECT * FROM CUSTOMER where C_PHONE ='29-899-517-8165' run at AccessController.java:0	2018/05/04 15:01:11	1 s	1/1	300/300

48GB CSV for TPCH CUSTOMER with 100 segments containing 600 million records.
20 concurrent queries.

查询优化 – 自定义索引

Index DataMap 框架，支持自定义索引

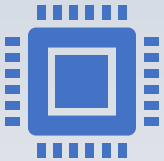
- ✓ BloomFilter DataMap，Lucene DataMap，MinMax DataMap (Example)均基于该框架完成;
- ✓ 索引数据生成：主要实现onPageAdded，onBlockletStart/End， onBlockStart/End接口，可以得到从row级别到文件级别各种不同粒度的索引信息;
- ✓ 索引的使用：主要实现prune接口，返回命中的Blocklet;
- ✓ Index DataMap 框架完成调用。

Q&A

欢迎加入 Huawei SmartCare 产品大数据平台!

大数据

AI



GPU加速