



<HDG>

# CarbonData介绍

在Hadoop上实现SQL交互式分析





# 为什么需要CarbonData



报表分析



OLAP、BI



批处理、ETL



实时分析



机器学习



data



# 挑战

- Data Size 数据规模
  - Single Table > 10 B 单表大于100亿行
  - Fast growing 快速增长
  - Nested data structure for complex object 数据结构复杂
- Multi-dimensional 数据维度多
  - Every record > 100 dimensions 分析的维度超过100
  - Add new dimension occasionally 维度不断增长
  - Billion level high cardinality 不同值范围在亿级别

# 应用场景分类

<HDG>

较固定的查询条件，BI拖拽分析  
Multi-dimensional OLAP Query



Ad-hoc，快速全量扫描，大表汇聚/Join  
Big Scan Query

Ad-hoc, 快速过滤，详单提取  
Small Scan Query



# How to choose storage 当前各种大数据方案分析

## 1. NoSQL Database

- 只支持单列key value查询 <5ms
- 不支持标准SQL

## 2. MPP relational Database

- Shared-nothing架构
- 不支持大集群 <100节点, 没有容错

## 3. Cube Data

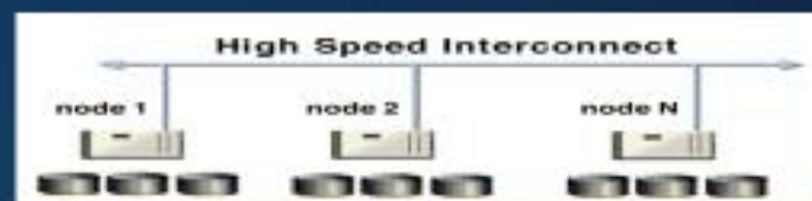
- 预聚合, 查询快
- 但数据膨胀大, 不支持查明细数据

## 4. Search Engine

- 通过索引快速找到数据
- 数据膨胀大2-4倍, 不支持SQL

## 5. SQL on Hadoop

- 聚焦计算引擎的分布式扫描
- 存储效率不高



# 架构师的苦恼： 不同应用不同数据存储，如果让数据存储统一？

Choice 1: 只支持部分应用



Choice 2: 复制数据





# CarbonData : 实现一份数据同时满足多种业务需求，与大数据生态无缝集成

Multi-dimensional OLAP Query



Full Scan Query

Small Scan Query



CarbonData: Unified Storage



**Apache CarbonData**实现一份数据同时满足多种业务需求，与**Spark**引擎融合后形成一站式大数据分析方案。

# Carbon大数据生态集成

<HDG>

- 内置支持Hadoop和Spark
  - Hadoop: > 2.2
  - Spark 1.5, 1.6, 2.1
- 接口
  - SQL
  - DataFrame API
- 支持操作:
  - Load, Query (with optimization)
  - Update, Delete, Compaction, etc
  - DataFrame
  - Machine Learning





# Apache CarbonData 源码阅读

源码: <https://github.com/apache/incubator-carbondata>

[david.caiq@gmail.com](mailto:david.caiq@gmail.com)

2017/3/13

目录

- 1. 源码目录..... 11
- 2. format 模块..... 12
  - 2.1 文件目录结构 .....12
  - 2.2 Schema 文件格式.....13
  - 2.3 carbondata 文件格式 .....14
  - 2.4 carbonindex 文件格式.....15
  - 2.5 dictionary 文件格式.....16
  - 2.6 tablestatus 文件格式 .....16
- 3. integration 模块 ..... 17
  - 3.1 spark 集成.....17
  - 3.2 spark2 集成.....18
- 4. core 模块..... 19
  - 4.1 Scan（查询） .....19
  - 4.2 Filter Expression.....20
  - 4.3 LRU Cache .....21
  - 4.4 BTree Index .....22
- 5. processing 模块.....25
  - 5.1 Global Dictionary.....25
  - 5.2 DataLoading（数据加载） .....25
  - 5.3 Compression Encoding .....27



1. 源码目录

|             |   |
|-------------|---|
| 文件夹         | 说明  |
| assembly    | 构建单一的 Jar 包， mvn clean -DskipTests package -Pdist 或者 mvn clean -DskipTests -Pinclude-all package -Pdist   |
| bin         | 提供了两个 Linux shell 脚本 carbon-spark-shell 和 carbon-spark-sql， 能够在 local mode 下快捷体验  |
| common      | 公共模块， 目前仅包含日志部分   |
| conf        | 配置文件 carbon.properties.template, dataload.properties.template   |
| core        | 核心模块, 包含了查询模块代码与一些基础的模型类型和工具类<br>1. core: 表、字典、索引等逻辑结构模型， 以及字典缓存、MDK 生成、数据解压缩、文件读写等<br>2. Scan: 实现查询功能， 包含查询数据扫描、表达式计算和数据过滤、详单查询行结果收集、查询执行工具类等  |
| dev         | 开发者工具, Java/scala Code style， findbugs 配置文件   |
| docs        | 文档维护  |
| examples    | 可运行的功能演示例子， 包括： flink, spark, spark2  |
| format      | CarbonData 文件格式定义， 使用 Apache Thrift 定义  |
| hadoop      | Hadoop 接口实现， 例如： CarbonInputFormat, CarbonRecordReader 等  |
| integration | 集成模块<br>1. Spark-common: spark 与 spark2 能重用代码的模块<br>2. Spark: CarbonContext,sqlparser, optimizer, sparkplan<br>3. Spark2: CarbonSession, CarbonEnv, CarbonScan, CarbonSource<br>4. Spark-common-test: spark 与 spark2 能重用的测试用例 |
| processing  | 数据加载模块<br>InputProcessorStep: 数据输入接收                      DataConverteProcessorStep: 数据转换(类型转换， 字典编码)<br>SortProcessorStep: 节点内排序                      DataWriterProcessorStep: 生成 Carbondata， Carbonindex 文件                 |

2. format 模块

2.1文件目录结构

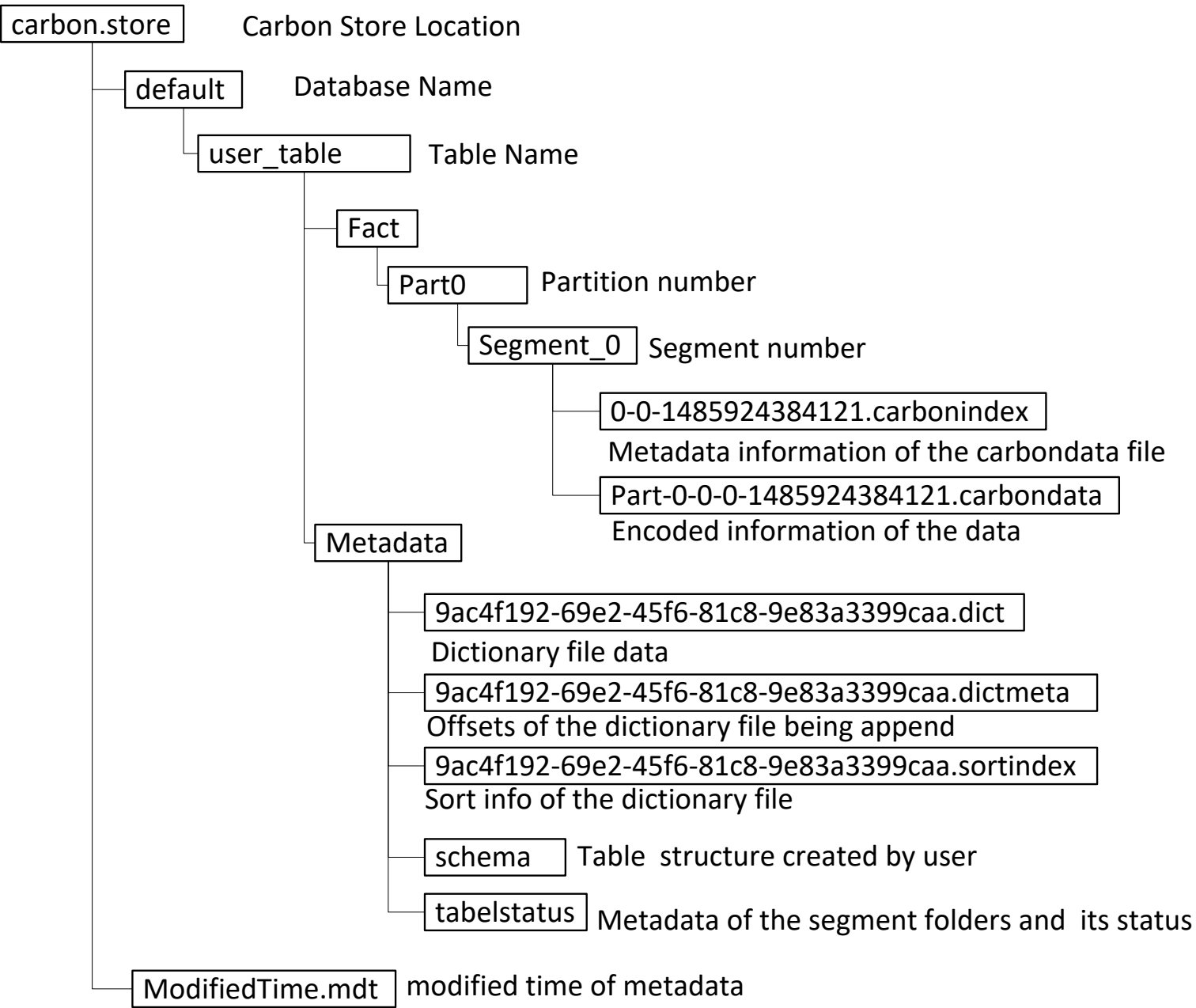
CarbonData 数据存储在 carbon.storelocation 配置项指定的位置（在 carbon.properties 中配置；若未配置则默认为../carbon.store）。

文件目录结构如图所示：

1.carbon.store 目录下有 ModifiedTime.mdt 文件和 database 目录（默认 database 名：default）。

2.default 目录下有属于该 database 的用户表目录（例如：user\_table）。

3.user\_table 目录下有 Metadata 目录和 Fact 目录；创建表时，在 Metadata 目录下生成 schema 文件，记录了表结构；其他文件均在 dataloading 时生成。其中每个字典编码的列对应生成 dict, dictmeta 和 sortindex 三个文件；每一次批量 loading 生成一个新的 segment 目录，其中每个 task 生成一个 carbonindx 和多个 carbondata 文件。





2.2Schema 文件格式

schema 文件内容如图中 TableInfo 类所示；

1. TableSchema 类

表名由 schema 文件所在的表目录决定。

tableProperties 用来记录 table 相关的属性，

例如：table\_blocksize。

2. ColumnSchema 类

encoders 用来记录 column 存储时采用的编码

columnProperties 用来记录 column 相关的属性。

3. BucketingInfo 类

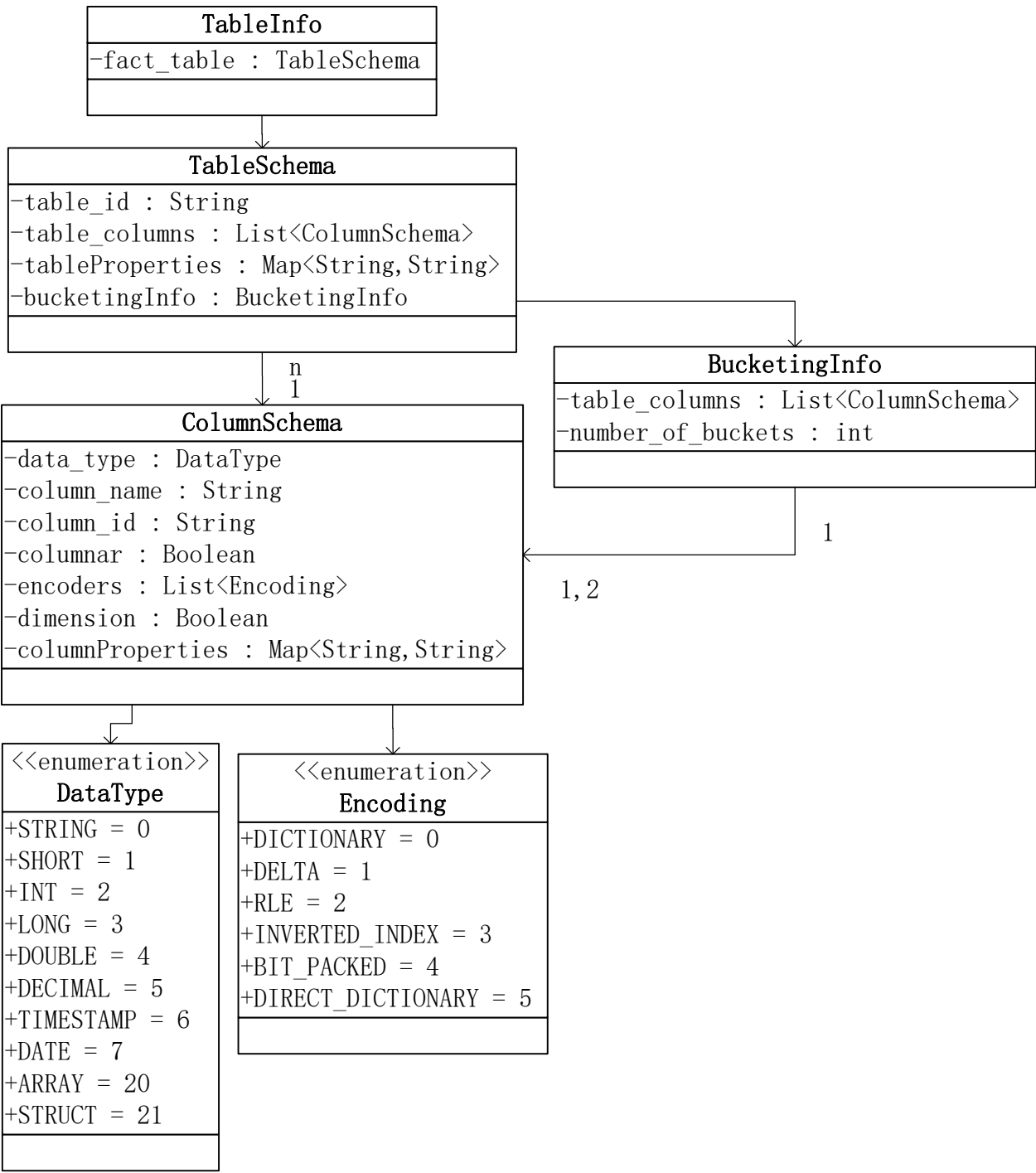
创建 bucket 表时可以指定表的 bucket 数量以及 column 来 split buckets。

4. DataType 类

描述了 CarbonData 支持的数据类型。

5. Encoding 类

CarbonData 文件可能用到的几种编码。



2.3carbodata 文件格式

2.3.1 blocklet 部分

carbodata file 由多个 blocklet 和一个 footer 部分组成。

blocklet 是 carbodata 文件内部的数据集（默认配置是 32 万行）。

carbodata 文件目前有如下 3 个版本：

V1: blocklet 由所有 column 的 data page, RLE page, rowID page 组成。

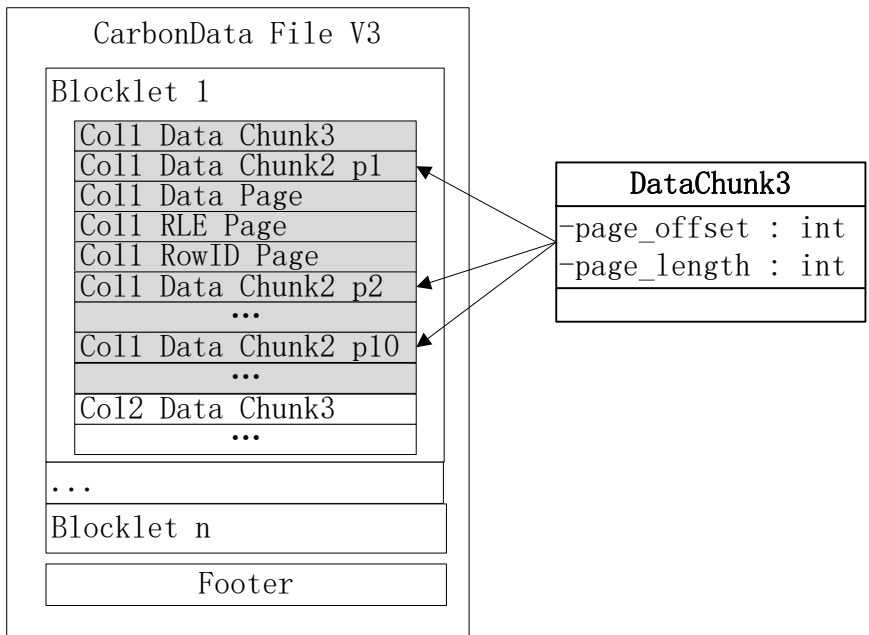
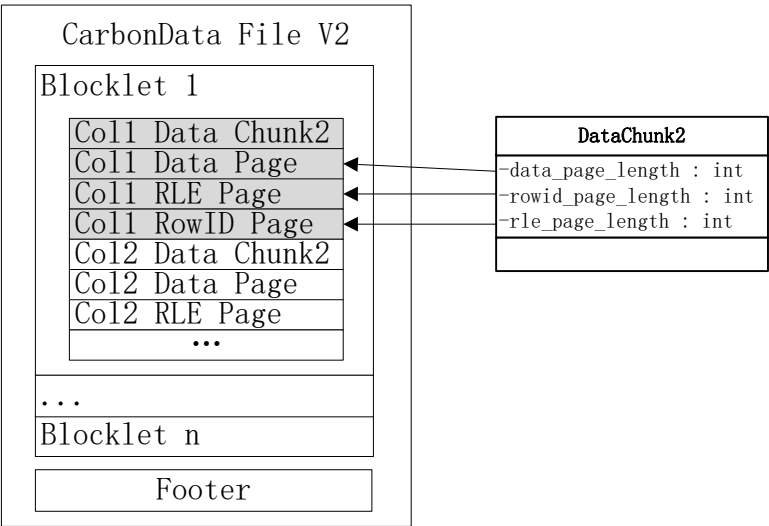
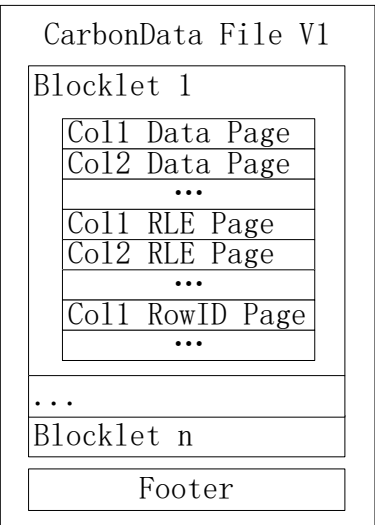
每个 column 的这 3 部分数据是在 blocklet 内分散存储的，在 footer 部分需要记录全部 page 的 offset 和 length 信息。

V2: blocklet 由 ColumnChunk2 组成。

ColumnChunk2 将 column 的 3 部分数据聚集在了一起，可以使用更少的 reader 去完成 column 数据读取。由于 DataChunk2 记录 page 的长度信息，因此，footer 只需要记录 ColumnChunk 的 offset 和 length，减小了 footer 数据量，有利于缩短元数据加载时间，提升 first query 性能。

V3: blocklet 由 ColumnChunk3 组成。

ColumnChunk3 默认由 10 个 ColumnChunk2 组成。在 blocklet 内 ColumnChunk3 包含了每个 column 的全部 page。ColumnChunk2 新增加了 BlockletMinMaxIndex，同时 footer 只需记录 ColumnChunk3 级别 BlockletMinMaxIndex。进一步缩小 footer 而且减小了 page 的数据行数，使索引更精确的命中 page。





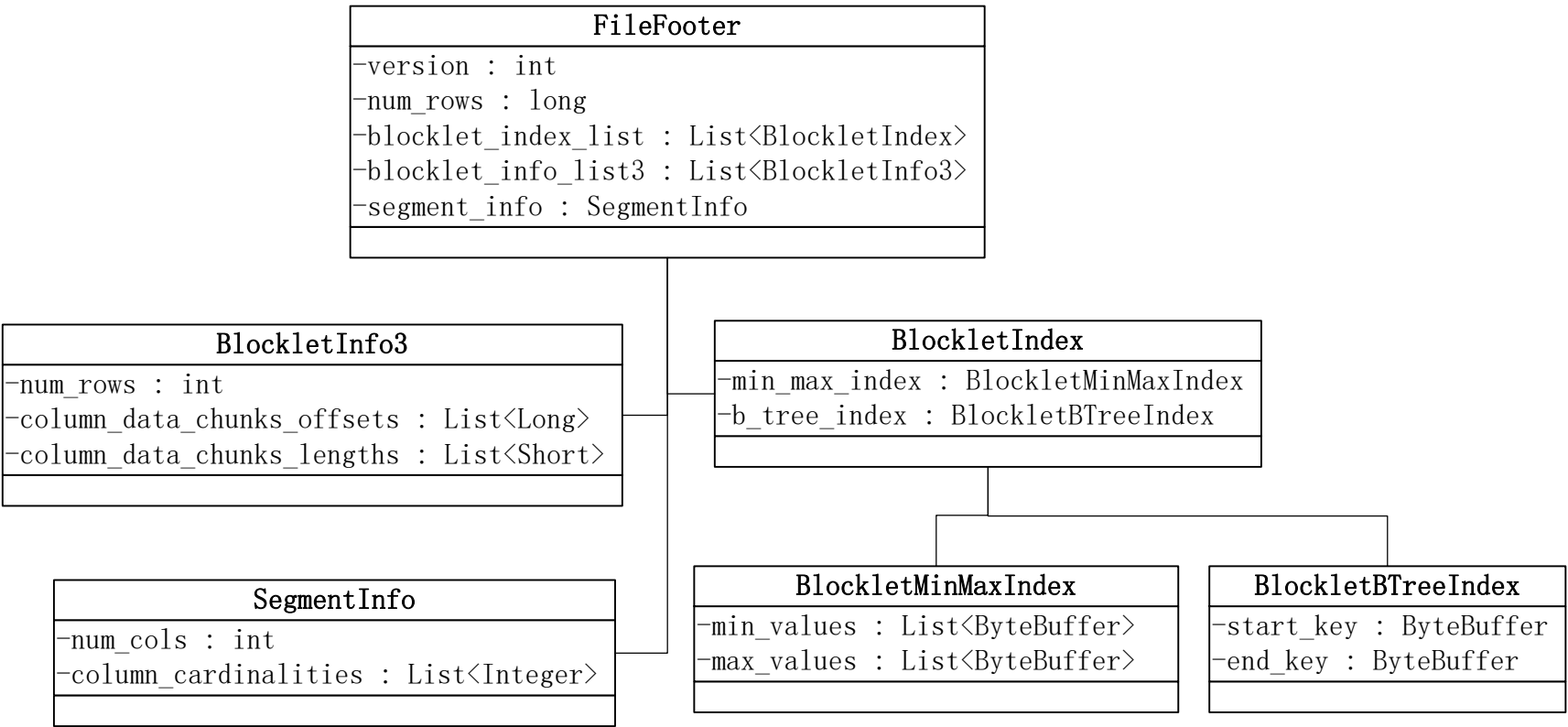
2.3.2 footer 部分

footer 记录每个 carbondata file 内部全部 blocklet 数据分布信息以及统计相关的元数据信息(minmax, startkey/endkey)。

1.BlockletInfo3 用来记录全部 ColumnChunk3 的 offset 和 length。

2.SegmentInfo 用来记录 column 数量以及每个 column 的 cardinality。

3.BlockletIndex 包括 BlockletMinMaxIndex 和 BlockletBTreeIndex。BlockletBTreeIndex 用来记录 block 内所有 blocklet 的 startkey/endkey; 查询时通过过滤条件结合 mdkey 生成查询的 startkey/endkey，借助 BlocketBtreeIndex 可以圈定每个 block 内满足条件的 blocklet 范围。BlockletMinMaxIndex 用来记录 blocklet 内所有列的 min/max 值; 查询时通过对过滤条件使用 min/max 检查,可以跳过不满足条件的 block/blocklet。



2.4carbonindex 文件格式

抽取出“2.3 footer 部分”中 BlockletIndex 部分生成 carbonindex 文件。Dataloading 时，一个 node 启一个 task，每个 task 生成一个 carbonindex 文件。 carbonindex 文件记录 segment 内每个 task 生成的所有 carbondata 文件内部所有 blocklet 的 index 信息。BlockIndex 记录 carbondata filename， footer offset 和 BlockletIndex， 数据量要比 footer 少些， 查询时直接使用该文件构建 driver 侧索引， 而无需跳扫数据量更大的 footer 部分。

2.5dictionary 文件格式

- 1.dict file 记录某列的 distinct value list。首次 dataloading 时，使用所有的 distinct value 生成该文件; 后续采用追加的方式。在 DataConvertStep，字典编码列会使用 key 替换 origin value。
- 2.dictmeta 记录每次 dataloading 的新增 distinct value 的元数据描述,字典缓存使用该信息增量刷新缓存。
- 3.sortindex 记录字典编码按照 origin value 排序后的 key 的结果集。 基于字典列的过滤查询，需要将 value 的过滤条等价转换为 key 的过滤条件，使用 sortindex 文件可以快速构建有序的 value 序列，以便快速查找 value 对应的 key 值。dataLoading 时，如果有新增的字典值，会重新生成 sortindex 文件。

2.6tablestatus 文件格式

tablestatus 记录每次加载和合并的 segment 相关的信息（采用 gson 格式），包括加载时间,加载状态，segment 名称，是否被删除以及合并入的 segment 名称等等。每次加载或合并完成后，重新生成 tablestatus file。

| BlockIndex                   |
|------------------------------|
| -num_rows : long             |
| -file_name : String          |
| -offset : long               |
| -block_index : BlockletIndex |
|                              |

| ColumnDictionaryChunk      |
|----------------------------|
| -values : List<ByteBuffer> |
|                            |

| ColumnDcitionaryChunkMeta |
|---------------------------|
| -min_surrogate_key : int  |
| -max_surrogate_key : int  |
| -start_offset : long      |
| -end_offset : long        |
| -chunk_count : int        |
| -segment_id : long        |
|                           |

| ColumnSortInfo                       |
|--------------------------------------|
| -sort_index : List<Integer>          |
| -sort_index_inverted : List<Integer> |
|                                      |

| LoadMetadataDetails      |
|--------------------------|
| -timestamp : String      |
| -loadStatus : String     |
| -loadName : String       |
| -partitionCount : String |
| -isDeleted : String      |
| -mergedLoadName : String |
|                          |

### 3. integration 模块

#### 3.1spark 集成

在 Spark 1.x 中，使用 CarbonContext 作为 spark sql 接口的入口。CarbonContext 扩展了 HiveContext，并定制了 CarbonSqlParser，部分 LogicalPlan, CarbonOptimizer 和 CarbonStrategy 等。

1.CarbonSqlParser用来解析 CarbonData 的 create table , load data 等 sql 语句，具体调用流程如图所示，CarbonSQLDialect 优先使用 CarbonSqlParser 去解析 sql(主要包括 create table, load table 等 sql 语句)，若无法解析，继续使用 HiveQL 去解析 sql（主要是 select 查询语句），并生成 Carbon LogicalPlan。

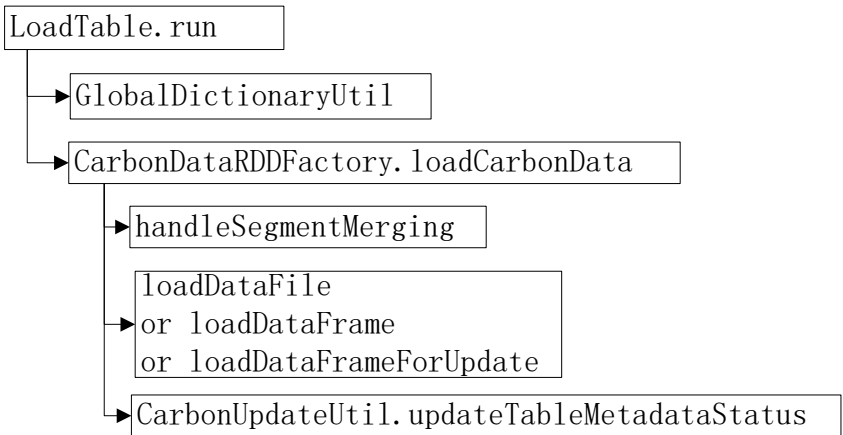
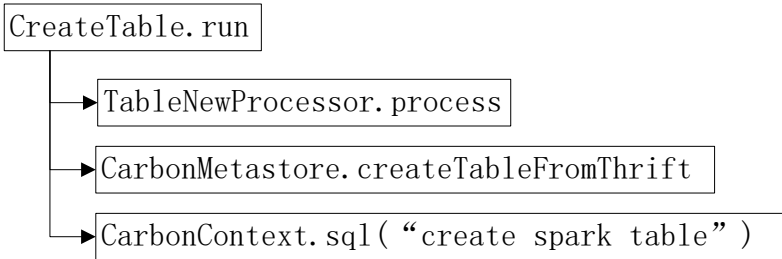
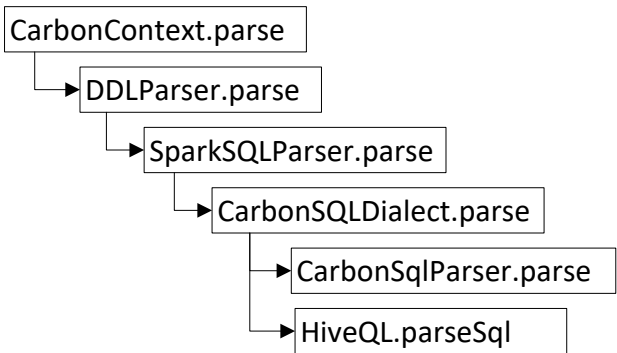
2.CarbonData LogicalPlan 主要有 CreateTable（创建表）和 LoadTable（加载数据）。

CreateTable: TableNewProcessor 用于生成 carbondata table schema 文件（ thrift 格式）；CarbonMetastore.createTableFromThrift 加载 schema 文件到元数据缓存中；最后创建 Spark table。

LoadTable: 主要分为 generateGlobalDictionary 和 loadCarbonData 两部分。

generateGlobalDictionary 用于生成表级字典编码。

loadCarbonData 用于生成 carbondata 和 carbonindex 文件。首先，会依据 load, insert 和 update 操作分别进入 loadDataFile,loadDataFrame 和 loadDataFrameForUpdate 加载过程生成 carbondata 和 carbonindex 文件；其次，CarbonUpdateUtil.updateTableMetadataStatus 记录数据加载的 tablestatus 信息。如果配置了 carbon.enable.auto.load.merge=true, 经过多次加载后，在加载前会触发 handleSegmentMerging 按配置的规则循环合并多个较小 segment 为一个较大的 segment，合并可以有效的防止小文件问题（也可以使用 alter table compact 命令触发合并）。



3. CarbonOptimizer 调整 LogicalPlan 在需要做字典解码的 LogicalPlan 前面插入 CarbonDictionaryCatalystDecoder LogicalPlan, 尽可能的延迟解码或者不解码。

4. CarbonStrategy (CarbonTableScan) 适配 CarbonData 相关的 LogicalPlan 生成 SparkPlan, 包括: 字典延迟解码 CarbonDictionaryDecoder, ExecutedCommand(LoadTableByInsert) 和表扫描 TableScan 等。TableScan 用于生成 CarbonScanRDD, 读取 CarbonData 表数据。

5. CarbonSource 是 carbondata datasource api, shortname 为 carbondata; Carbon table relation 为 CarbonDataSourceRelation。

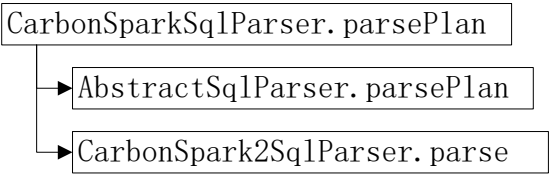
6. CarbonEnv: 主要作用是初始化加载 CarbonData 的元数据信息。

3.2 spark2 集成

在 Spark2 中, 使用 CarbonSession 作为 spark Sql 接口的入口。CarbonSession 扩展了 SparkSession, 包括 CarbonSessionState, CarbonSparkSqlParser, CarbonLateDecodeRule 和 CarbonLateDecodeStartegy, DDLStartegy 等。

与 spark 1.x 相对比, 不同点有:

CarbonSparkSqlParser 调用流程发生了变化, 优先调用 spark 的 AbstractSqlParser 解析 sql, 若无法解析, 继续使用 CarbonSpark2SqlParser 来解析, 并生成 Carbondata 的 LogicalPlan。CarbonSpark2SqlParser 主要处理 Carbondata 表上的 create table ,load table 等 sql 解析; 目前,Carbondata 对 select 语句未做扩展, 仍然由 spark AbstractSqlParser 来解析, 这样的顺序调整有利于减少查询语句的解析时间。



InsertInto 实现方式发生了变化。Spark 通过添加 Analyzer(CarbonPreInsertionCasts), 并在 DDLStrategy 中适配为 LoadTableByInsert。通过 CarbonDatasourceHadoopRelation 扩展 InserttableRelation 来实现的。

Carbon table relation 改为 CarbonDatasourceHadoopRelation, 改用 buildScan 方法来生成 CarbonScanRDD。

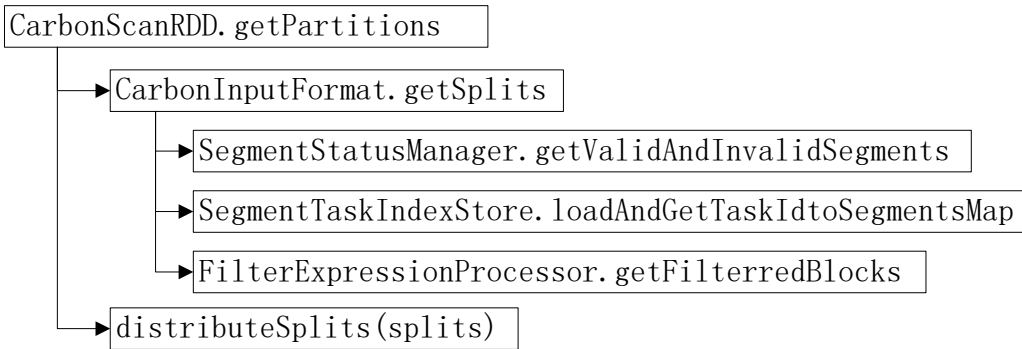


4. core 模块

4.1Scan（查询）

CarbonScanRDD 是 Carbon 表数据集，用于读取 Carbon 表数据；关键的 method 是 getPartitions（获取过滤条件命中的部分 blocklet 的 block 分块）和 compute（扫描 blocklet 过滤并拼接出数据行的结果集）。

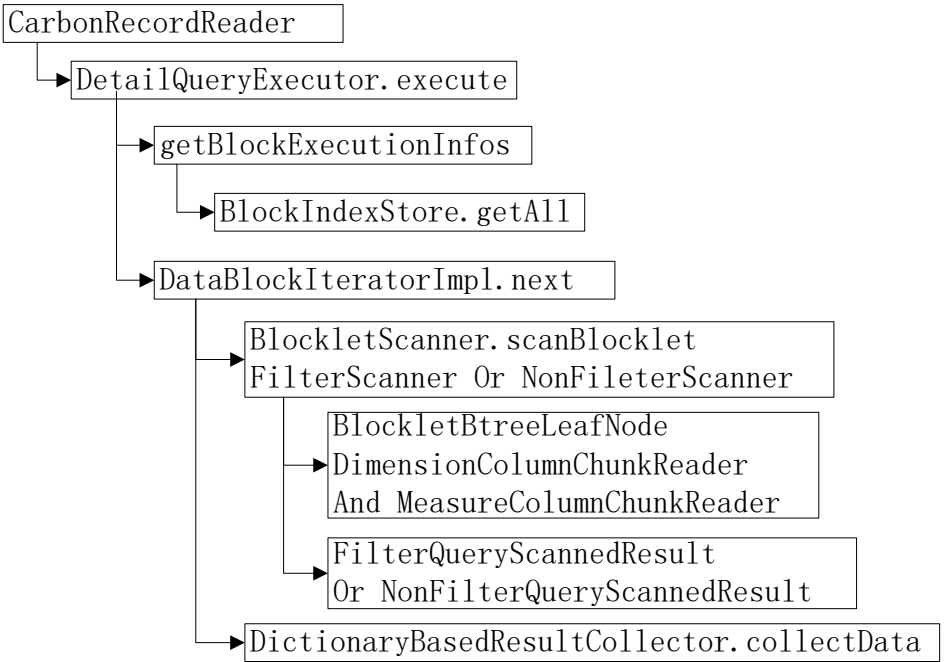
1.getPartition 实现流程图所示。CarbonInputFormat.getSplits 方法用来获取查询命中的 block 分块，其中 SegmentStatusManager.getValidAndInvalidSegments 方法可以获取 carbon 表中有效的所有 Segment，通过 SegmentTaskIndexStore.loadAndGetTaskIdtoSegmentsMap 加载 segment 的 carbonindex 索引文件，然后 FilterExpressionProcessor.getFilterredBlocks 从索引信息中查找出被命中的所有 block。distributeSplits 方法用于将 splits 根据数据本地化计算原则均匀的按集群节点分组，然后按照分组的结果为所有 splits 生成 Spark Partition。



2.Compute 使用实现流程如图所示。CarbonRecordReader 内部封装了 DetailQueryExecutor 来实现 carbon 表的 detail query。

getBlockExecutionInfos 获取各个 block 上查询执行需要的信息，包括 blockindex, startkey, endkey, startBlockletIndex, numberOfBlockletToScan, filterExecuter 等。Executor 侧 BlockIndex 由 BlockIndexStore.getAll 方法加载构建。

DataBlockIteratorImpl.next 扫描 block 获取 batchsize 行数据。BlockletScanner 根据有无过滤条件分为：FilterScanner 和 NonFilterScanner, BlockletScanner.scanBlocklet 使用已缓存的 BlockletBTreeLeafNode 元数据信息加载命中的 blocklet（driver 侧传递过来的 start blocklet 和 blocklet 数量，以及 executer 侧使用 blocklet 级别的 minmax 检查）内所需列的数据来填充 ScannedResult，对于 FilterScanner，还会生成过滤条件命中的数据行号索引， 返回 FilterQueryScannedResult 或者 NonFilterQueryScannedResult。DictionaryBasedResultCollector 调用 collectData 方法从上一步的 result 中，将需要的列数据拼接为最后需要的行数据。



## 4.2 Filter Expression

在 `executer` 侧内部实现了一套 `blocklet` 内列存储数据按表达式过滤的方法。

首先，被 `pushdown` 到 `CarbonScanRDD` 里的 `filter expression`，会被转换为如右图所示的 `carbodata` 的 `Expression`。实现代码：`CarbonFilters.createCarbonFilter`。

其次，基于上一步的 `Expression tree` 生成 `FilterResolverIntf tree`，子类如下图所示。转换过程代码为：`CarbonInputFormatUtil.resolveFilter`，执行 `Expression` 表达式计算，根据表达式类型将结果封装在不同的 `FilterResolverIntf` 中。

|   |   |          |
|---|---|----------|
| Choose Implementation of FilterResolverIntf (7 found) |   |          |
| ConditionalFilterResolverImpl                         | (org.apache.carbondata.core.scan.filter.resolver) | carbonda |
| LogicalFilterResolverImpl                             | (org.apache.carbondata.core.scan.filter.resolver) | carbonda |
| RowLevelFilterResolverImpl                            | (org.apache.carbondata.core.scan.filter.resolver) | carbonda |
| RowLevelRangeFilterResolverImpl                       | (org.apache.carbondata.core.scan.filter.resolver) | carbonda |

|                                |  |                  |
|--------------------------------|--|------------------|
| AndExpression                  | (org.apache.carbondata.core.scan.expression.logical)     | carbodata-core   |
| BinaryConditionalExpression    | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| BinaryExpression               | (org.apache.carbondata.core.scan.expression)             | carbodata-core   |
| BinaryLogicalExpression        | (org.apache.carbondata.core.scan.expression.logical)     | carbodata-core   |
| ColumnExpression               | (org.apache.carbondata.core.scan.expression)             | carbodata-core   |
| EqualToExpression              | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| FalseExpression                | (org.apache.carbondata.core.scan.expression.logical)     | carbodata-core   |
| GreaterThanOrEqualToExpression | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| GreaterThanExpression          | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| InExpression                   | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| LeafExpression                 | (org.apache.carbondata.core.scan.expression)             | carbodata-core   |
| LessThanOrEqualToExpression    | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| LessThanExpression             | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| ListExpression                 | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| LiteralExpression              | (org.apache.carbondata.core.scan.expression)             | carbodata-core   |
| NotEqualsExpression            | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| NotInExpression                | (org.apache.carbondata.core.scan.expression.conditional) | carbodata-core   |
| OrExpression                   | (org.apache.carbondata.core.scan.expression.logical)     | carbodata-core   |
| SparkUnknownExpression         | (org.apache.spark.sql)                                   | carbodata-spark2 |
| UnknownExpression              | (org.apache.carbondata.core.scan.expression)             | carbodata-core   |

接着，基于上一步的 `FilterResolverIntf tree` 生成 `FilterExecuter tree`，子类如下图所示。转换过程代码为：`FilterUtil.getFilterExecuterTree`，根据 `FilterResolverIntf` 中 `executer` 类型生成对应的 `FilterExecuter`。

|   |   |                |
|---|---|----------------|
| AndFilterExecuterImpl                         | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| ExcludeColGroupFilterExecuterImpl             | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| ExcludeFilterExecuterImpl                     | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| IncludeColGroupFilterExecuterImpl             | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| IncludeFilterExecuterImpl                     | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| OrFilterExecuterImpl                          | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| RestructureFilterExecuterImpl                 | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| RowLevelFilterExecuterImpl                    | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| RowLevelRangeGrtrThanFiterExecuterImpl        | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| RowLevelRangeGrtrThanEquaToFilterExecuterImpl | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| RowLevelRangeLessThanEqualFilterExecuterImpl  | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |
| RowLevelRangeLessThanFiterExecuterImpl        | (org.apache.carbondata.core.scan.filter.executer) | carbodata-core |

最后，`FilterScanner.fillScannedResult` 调用 `FilterExecuter.applyFilter` 计算出过滤条件命中的 `Blocklet` 内部数据的行号 `BitSet`，利用 `RowId index` 转换成行号索引。

4.3LRU Cache

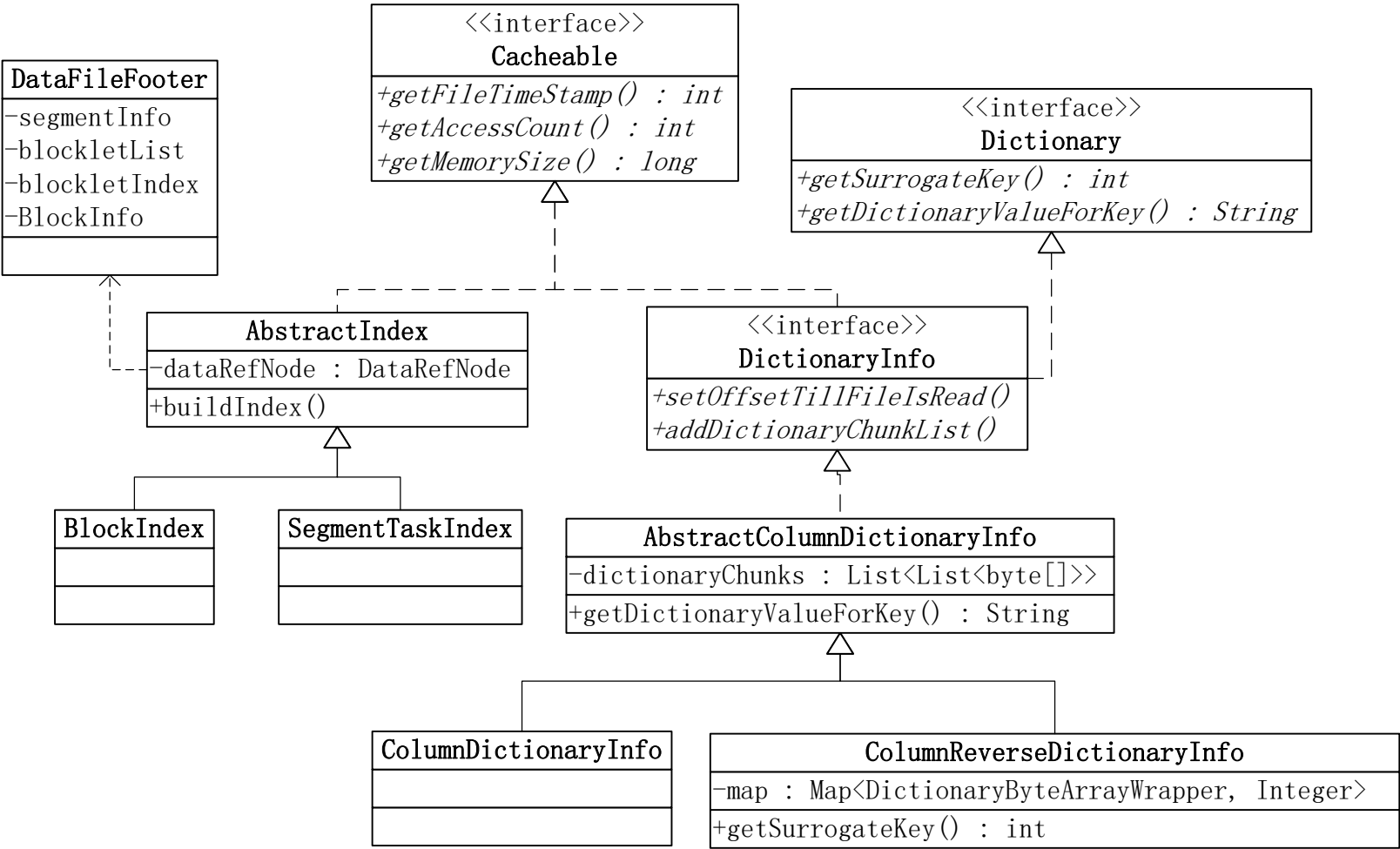
CarbonLRUCache 根据用途有 4 种类型，结合下图所示，分别是：

Key->Value 的 ForwardDictionaryCache: 内容为 ColumnDcitionaryInfo

Value->Key 的 ReverseDictionaryCache: 内容为 ColumnReverseDcitionaryInfo

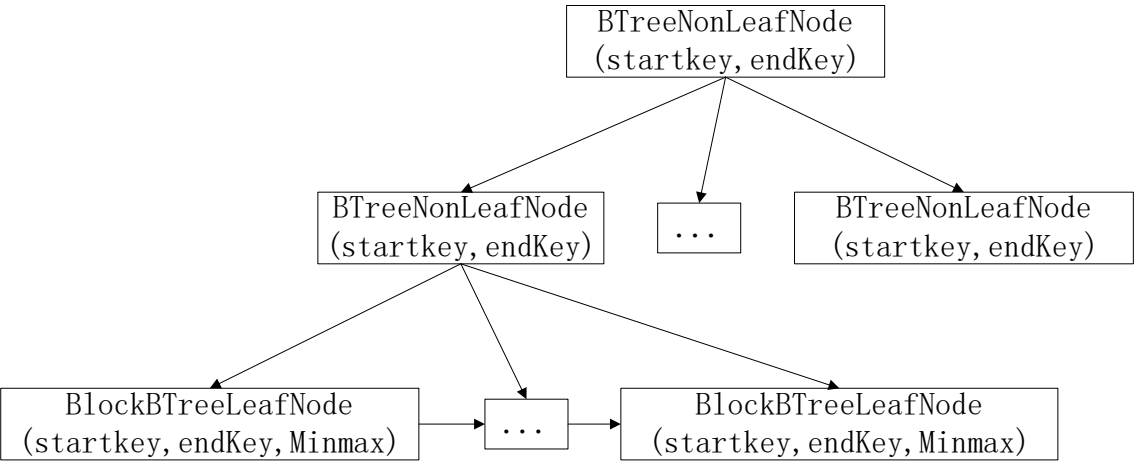
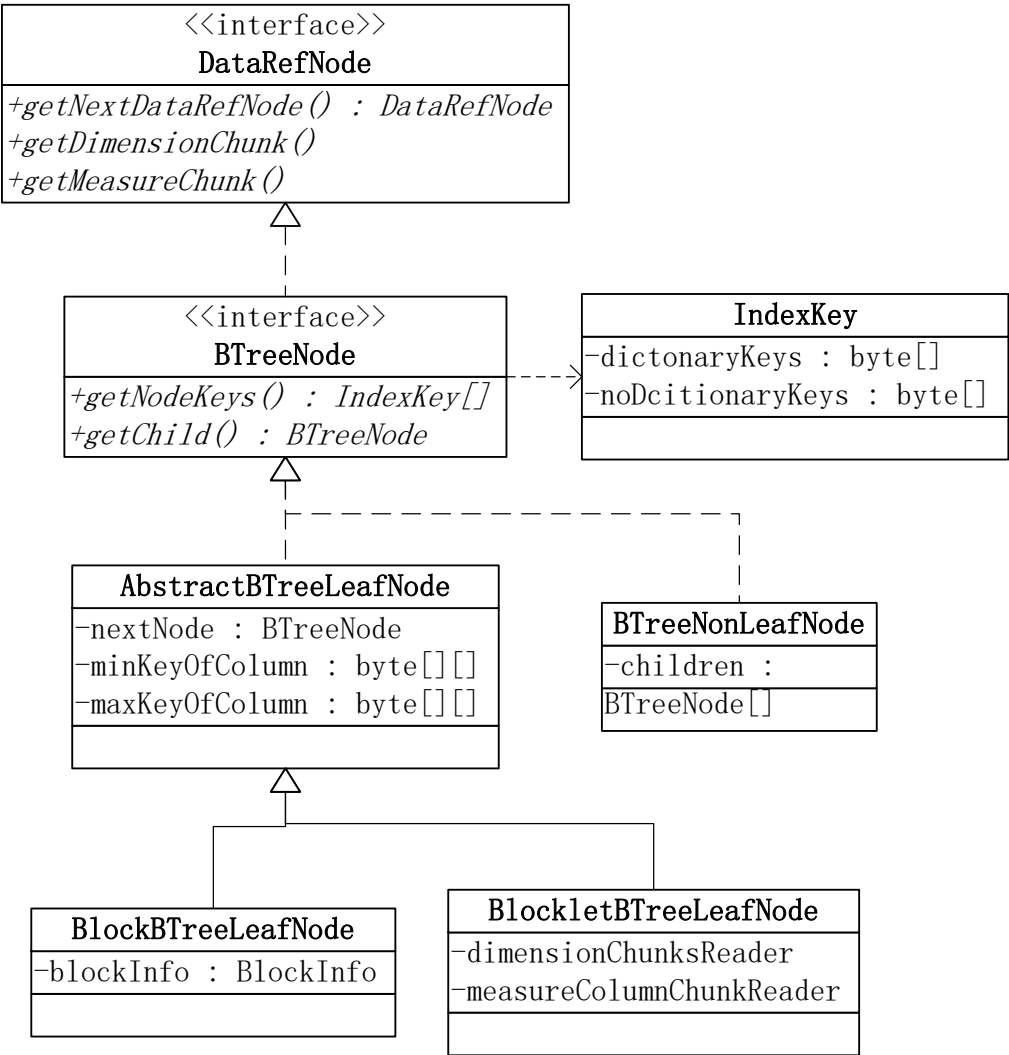
Driver 侧 BTree 索引缓存 SegmentTaskIndexStoree: 内容为 SegmentTaskIndex

Executor 侧 Btree 索引缓存 BlockIndexStore: 内容为 BlockIndex



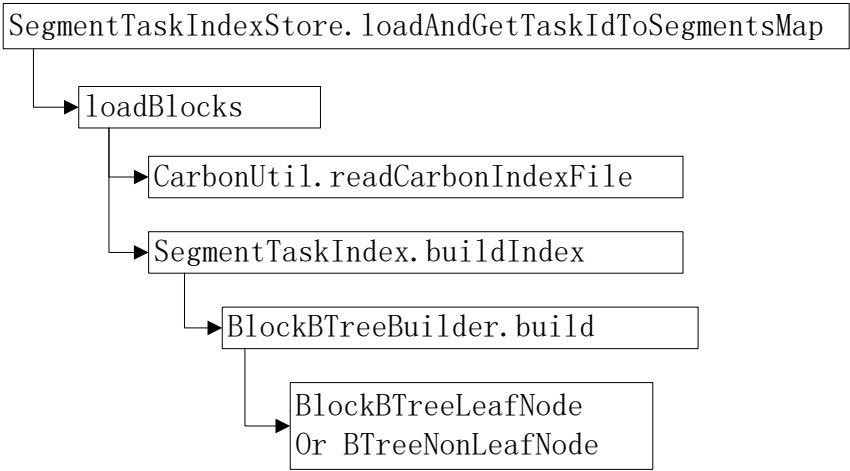
4.4BTree Index

借助 cache 的索引，来快速命中需要的 block 和 blocklet。索引模型如下图所示，IndexKey 包含 dicitoanry 和 nodictionary 列，这也是当前 carbondata dataloading 时用到的排序列。BTreeNode 中的 IndexKey 为相关的 Block/Blocklet 的 startKey,endKey。AbstractBTreeLeaf 包含了 NodeBlock/Blocklet 中各列的 min/max 值。BlockBTreeLeafNode 用于 driver 侧缓存 carbonindex 信息， BlockletBTreeLeafNode 用于 executor 侧缓存 footer 信息，并封装了从 hdfs 读取 column data chunk 的方法。



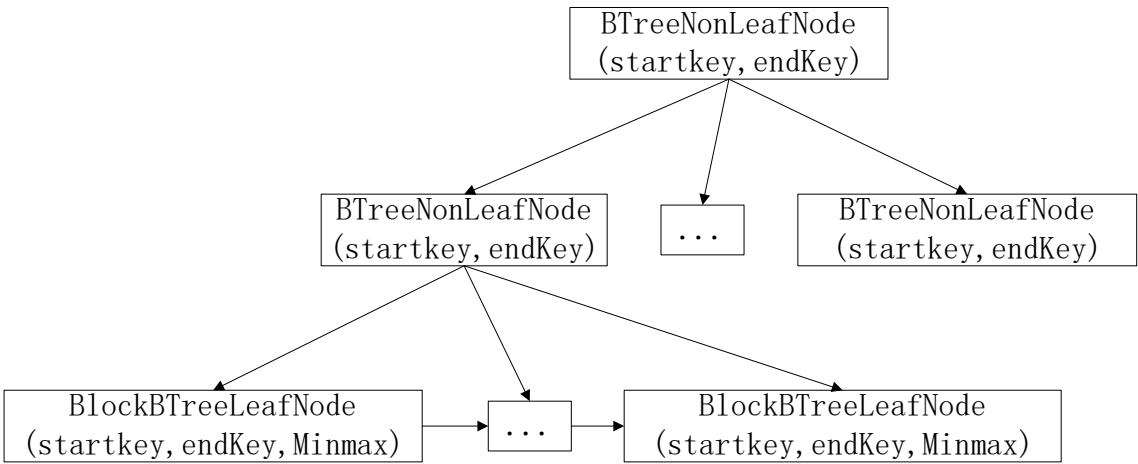


1.Driver 侧 index cache 由 SegmentTaskIndexStore 来管理, 索引加载构建过程如图所示, CarbonUtil.readCarbonIndexFile 读取 carobnindex 文件, SegmentTaskIndex.buildIndex 为每个 segment 的每个 task 构建一棵索引树。

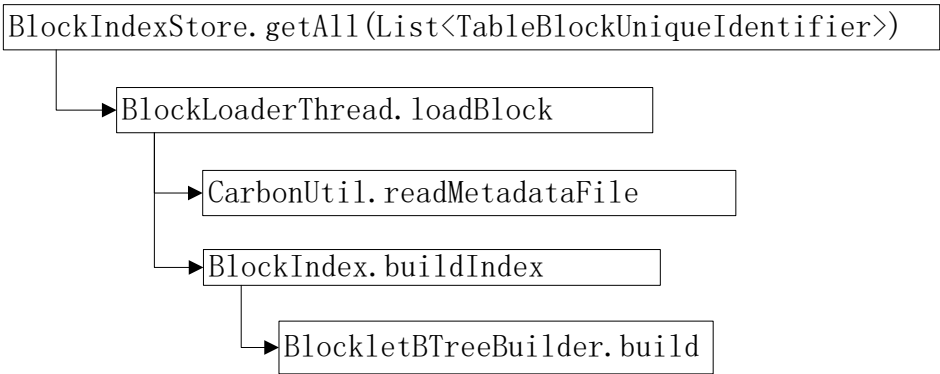


索引树结构如下图所示, 每个 BTreeNonLeafNode 有不多于 32 个子节点, 其 startkey 和 endkey 分别是最左侧的子节点的 startkey 和最右侧的子节点的 endkey。

查询时, 首先通过过滤条件计算出 startkey,endkey 以及 minmax 值, 然后, 使用 startkey,endkey 从树中搜索出符合条件的 leafNode 节点范围, 最后, 对于 leafNode(对应一个 block)使用 minmax 检查剔除部分 block.



2.Executer 侧索引 cache 由 BlockIndexStore 来管理, 索引加载构建过程如下图所示, CarbonUtil.readMetadataFile 读取 data file footer metadata 信息, BlockletBtreeBuilder.build 为每个 block 构建一棵索引树。索引树结构与上图类似, leaf node 为 BlockletBtreeLeafNode.



查询时, 首先使用 driver 侧传递过来的 start blocklet 和 blocklet 数量, 结合 executer 侧 BlockletBtree 确定需要 scan 的 blocklets, 并使用 blocklet 级别的 minmax 检查跳过不符合条件的 blocklet。

<HDG>

## Spark Driver

## 1. File pruning

## Spark Executor

## 2. Blocklet pruning

HDFS

File

Blockdet

Blocklet

Blocklet

Footer

File

## Blocklet

Blocklet

Blocklet

Footer

File

## Blocklet

Blocklet

Blocklet

Footer

File

Blocklet

Blocklet

Blocklet

Footer

100

## Blocklet

C1

C2

C3

CA


• •

Cn

### 3. Read and decompress filter column

4. Binary search using inverted index, skip to next blocklet if no matching

## 5. Decompress projection column

6. Return decoded data to spark  HUAWEI

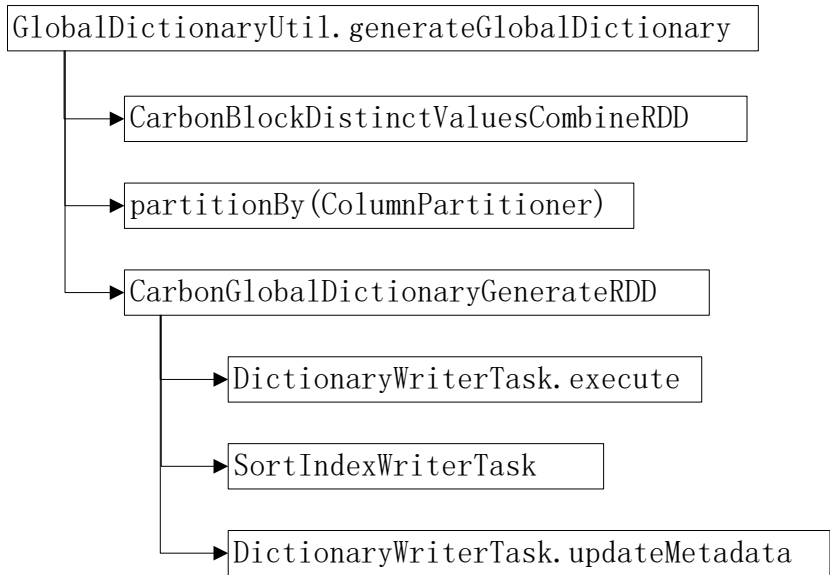


**HUAWEI**

5. processing 模块

5.1Global Dictionary

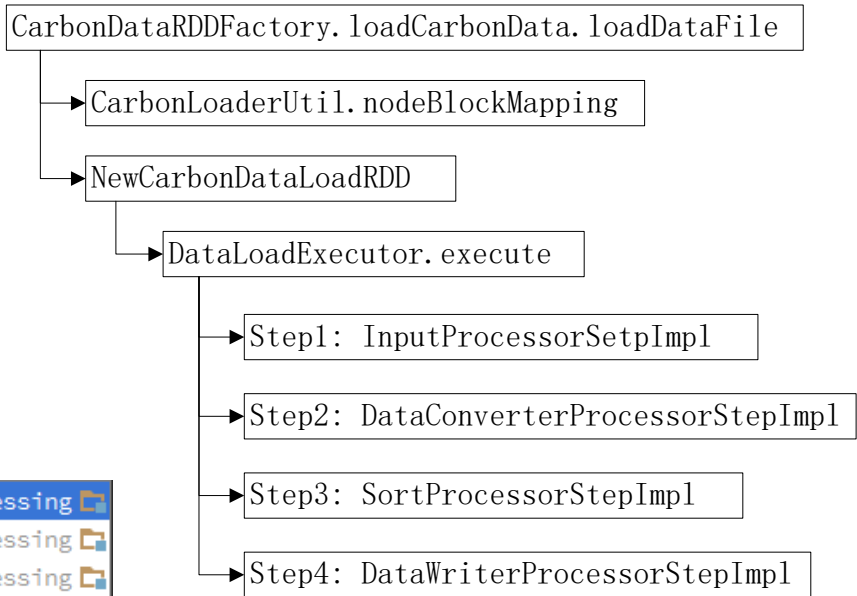
全局字典编码（目前为表级字典编码）使用统一的整数来代理真实数据，可以起到数据压缩作用，同时基于字典编码分组聚合的计算也更高效。全局字典编码生成过程如图所示，CarbonBlockDistinctValuesCombineRDD 计算每个 split 内字典列的 distinct value 列表，然后按照 ColumnPartitioner 做 shuffle 分区，每列一个分区。CarbonGlobalDictionaryGenerateRDD 每个 task 处理一个字典列，生成字典文件（或者追加新字典值），刷新 sortindex, 更新 dictmeta。



5.2DataLoading（数据加载）

CSV 文件数据加载主流程，如下图所示。首先，CarbonLoaderUtil.nodeBlockMapping 将数据块按节点分区。NewCarbonDataLoadRDD 采用该节点分区，每个节点启动一个 task 来处理数据块的加载。DataLoadExecutor.execute 执行数据加载，主流程包括图中所示的 4 个步骤。

- Step1: InputProcessorStepImpl 使用 CSVInputFormat.CSVRecordReader 读取并解析 CSV 文件。
- Step2: DataConverterProcessStepImpl 用来转换数据，FieldConverter 主要有以下实现类，包括字典列，非字典列，直接字典，复杂类型列和度量列转换。



|  |                      |
|--|----------------------|
| AbstractDictionaryFieldConverterImpl (org.apache.carbondata.processing.newflow.converter.impl) | carbodata-processing |
| ComplexFieldConverterImpl (org.apache.carbondata.processing.newflow.converter.impl)            | carbodata-processing |
| DictionaryFieldConverterImpl (org.apache.carbondata.processing.newflow.converter.impl)         | carbodata-processing |
| DirectDictionaryFieldConverterImpl (org.apache.carbondata.processing.newflow.converter.impl)   | carbodata-processing |
| MeasureFieldConverterImpl (org.apache.carbondata.processing.newflow.converter.impl)            | carbodata-processing |
| NonDictionaryFieldConverterImpl (org.apache.carbondata.processing.newflow.converter.impl)      | carbodata-processing |

Step3: SortProcessorStepImpl 将数据按照 dimension sort, 默认的 Sorter 实现类是 ParallelReadMergeSorterImpl, Sorter 主流程如右图所示。SortDataRows.addRowBatch 方法缓存数据, 当数据记录数达到 sort buffer size (默认 100000), 调用 DataSorterAndWriter 排序并生成 tmp file 到 local disk; 当 tmpfile 数量达到配置的阈值 (默认 20) 调用 SortIntermediateFileMerger.startMerge 将这些 tmpfile 归并排序生成 big tmp file. 在 Step1 和 Step2 的输入数据都完成排序并生成文件 (一些 big tmpfile 和不到 20 个的 tmpfile) 到 tmp 目录后, SingleThreadFinalSortFilesMerger.startFinalMerge 启动 final merge, 流式归并排序所有的 tmpfile, 目的是使本节点本次 loading 的数据有序, 并为后续 Step4 提供数据的流式输入。

Step4: DataWriterProcessorStepImpl 用于生成 carbondata 和 carbonindex 文件。主流程如下图所示。MultiDimKeyVarLengthGenerator.generateKey 为每一行的字典编码 dimesion 生成 MDK。CarbonFactDataHandlerColumnar.addDataToStore 缓存 MDK 编码等数据, 记录数达到 blockletsize 大小后, 调用 Producer 生成 Blocklet 对象(NodeHolder)。

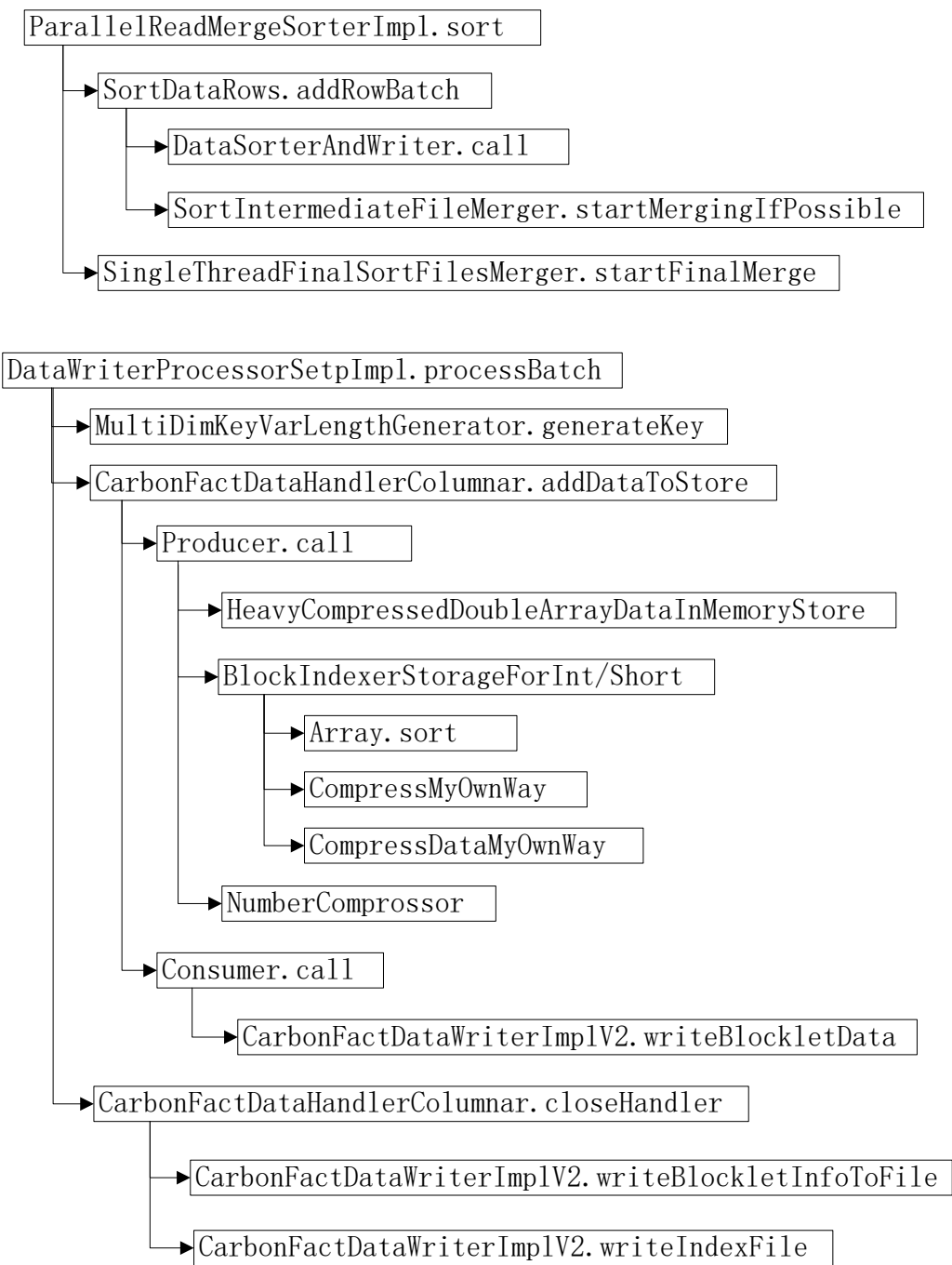
BlockIndexerStorageForInt/Short 处理 blocklet 内 dimension 列数据的排序(Array.sort)、生成 RowId index(compressMyOwnWay), 采用 RLE 压缩(compressDataMyOwnWay)

HeavyCompressedDoubleArrayDataInMemoryStore 处理 bloclet 内 meause 类数据的压缩(使用 snappy)。

CarbonFactDataWriterImplV2.writerBlockletData 将已有的一个 blocklet 数据写入本地数据文件。如果 blocklet 累计大小已经达到了 table\_blocksize 大小, 新建 carbondata 来写入数据。

在 carbondata file 的 blocklet 写入结束后, 调用 writeBlockletInfoToFile 完成 footer 部分写入。

在本节点 task 结束后, 调用 writeIndexFile 生成 carbonindex 文件。





### **5.3 Compression Encoding**

- 1.Snappy Compressor
- 2.Number Compressor
- 3.Delta Compressor
- 4.Dictionary Encoding
- 5.Direct-Ditionary Encoding
- 6.RLE(Running Length Encoding)

# 索引和编码介绍

<HDG>

- 数据和索引合一存储，数据即索引
- 多维索引 (Multi-Dimensional Key)
- 全局字典编码

| Years | Quarters | Months | Territory | Country   | Quantity | Sales  |
|-------|----------|--------|-----------|-----------|----------|--------|
| 2003  | QTR1     | Jan    | EMEA      | Germany   | 142      | 11,432 |
| 2003  | QTR1     | Jan    | APAC      | China     | 541      | 54,702 |
| 2003  | QTR1     | Jan    | EMEA      | Spain     | 443      | 44,622 |
| 2003  | QTR1     | Feb    | EMEA      | Denmark   | 545      | 58,871 |
| 2003  | QTR1     | Feb    | EMEA      | Italy     | 675      | 56,181 |
| 2003  | QTR1     | Mar    | APAC      | India     | 52       | 9,749  |
| 2003  | QTR1     | Mar    | EMEA      | UK        | 570      | 51,018 |
| 2003  | QTR1     | Mar    | Japan     | Japan     | 561      | 55,245 |
| 2003  | QTR2     | Apr    | APAC      | Australia | 525      | 50,398 |
| 2003  | QTR2     | Apr    | EMEA      | Germany   | 144      | 11,532 |

Blocklet Logical View

| C1 | C2 | C3 | C4 | C5 | C6  | C7    |
|----|----|----|----|----|-----|-------|
| 1  | 1  | 1  | 1  | 1  | 142 | 11432 |
| 1  | 1  | 1  | 1  | 3  | 443 | 44622 |
| 1  | 1  | 1  | 3  | 2  | 541 | 54702 |
| 1  | 1  | 2  | 1  | 4  | 545 | 58871 |
| 1  | 1  | 2  | 1  | 5  | 675 | 56181 |
| 1  | 1  | 3  | 1  | 7  | 570 | 51018 |
| 1  | 1  | 3  | 2  | 8  | 561 | 55245 |
| 1  | 1  | 3  | 3  | 6  | 52  | 9749  |
| 1  | 2  | 4  | 1  | 1  | 144 | 11532 |
| 1  | 2  | 4  | 3  | 9  | 525 | 50398 |

Sorted MDK Index

[1,1,1,1,1] : [142,11432]  
[1,1,1,1,3] : [443,44622]  
[1,1,1,3,2] : [541,54702]  
[1,1,2,1,4] : [545,58871]  
[1,1,2,1,5] : [675,56181]  
[1,1,3,1,7] : [570,51018]  
[1,1,3,2,8] : [561,55245]  
[1,1,3,3,6] : [52,9749]  
[1,2,4,1,1] : [144,11532]  
[1,2,4,3,9] : [525,50398]

Encoding

[1,1,1,1,1] : [142,11432]  
[1,1,1,3,2] : [541,54702]  
[1,1,1,1,3] : [443,44622]  
[1,1,2,1,4] : [545,58871]  
[1,1,2,1,5] : [675,56181]  
[1,1,3,3,6] : [52,9749]  
[1,1,3,1,7] : [570,51018]  
[1,1,3,2,8] : [561,55245]  
[1,2,4,3,9] : [525,50398]  
[1,2,4,1,1] : [144,11532]

Sort  
(MDK Index)