Kylin基于MOLAP实现，查询的时候利用Calcite框架，从存储在Hbase的segment表（每一个segment对应着一个htable）获取数据，其实理论上就相当于使用Calcite支持SQL解析，数据从Hbase中读取，中间Kylin主要完成如何确定从Hbase中的哪些表读数据，如何读取数据，以及解析数据的格式。

场景设置

首先设想一种cube的场景：

维度：A（cardinality=10）、B（cardinality=20）、C（cardinality=30）、D（cardinality=40），其中A为mandatory维度，rowkey顺序为A、B、C、D，只有一个分组。

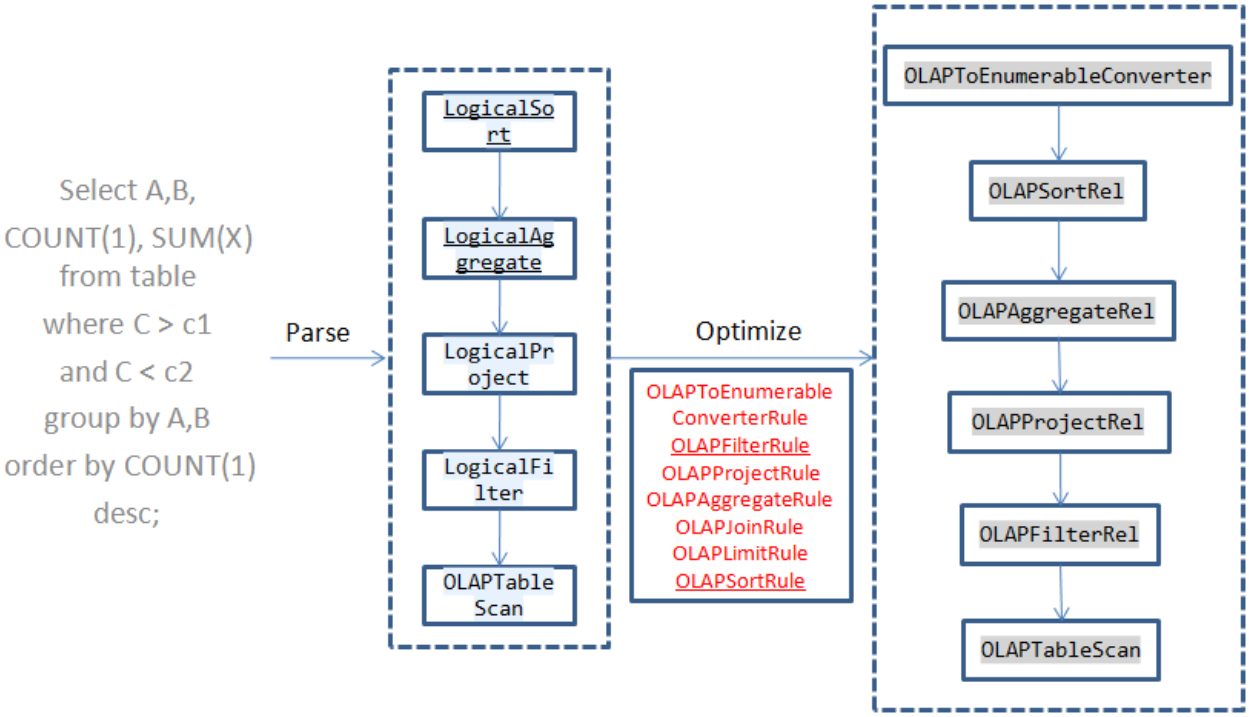
度量：COUNT(1), SUM(X)

在这种情况下，这个cube包含如下的cuboid：ABCD、ABC、ABD、ACD、AB、AC、AD、A。目前Kylin在执行查询的时候只能通过查找cube进行匹配，如果能够找到一个匹配的cube则读取通过扫描该cube的所有segment处理该请求，首先先看一下kylin是如何处理一个SQL查询的。

执行查询

Kylin提供了两种执行SQL查询的方式：jdbc访问和http api的访问，前者的实现实际上是在客户端封装了http api请求，然后获取结果再转换成ResultSet对象，在执行查询之前Kylin服务端会对查询的SQL做缓存，尤其是执行时间比较久的查询，缓存是基于SQL的内容作为key，结果作为value的，所以重复执行一个查询会很快返回的（这是因为Kylin假设数据是只读的，不会被修改）。如果缓存不命中则使用服务器内嵌的Calcite创建一个向Calcite的jdbc connection，然后使用jdbc的方式获取执行结果，在使用Calcite的时候用户只需要给Calcite提供数据，Calcite能够完成其他物理算子的优化和执行，但是对于Kylin来说，它深度定制了Calcite，增加了一些优化的策略，所以总的来说查询可以分成两部分：1、kylin是如何使用calcite完成SQL的解析，获取SQL的上下文；2、kylin如何从预计算的数据中获取数据并进行计算的。

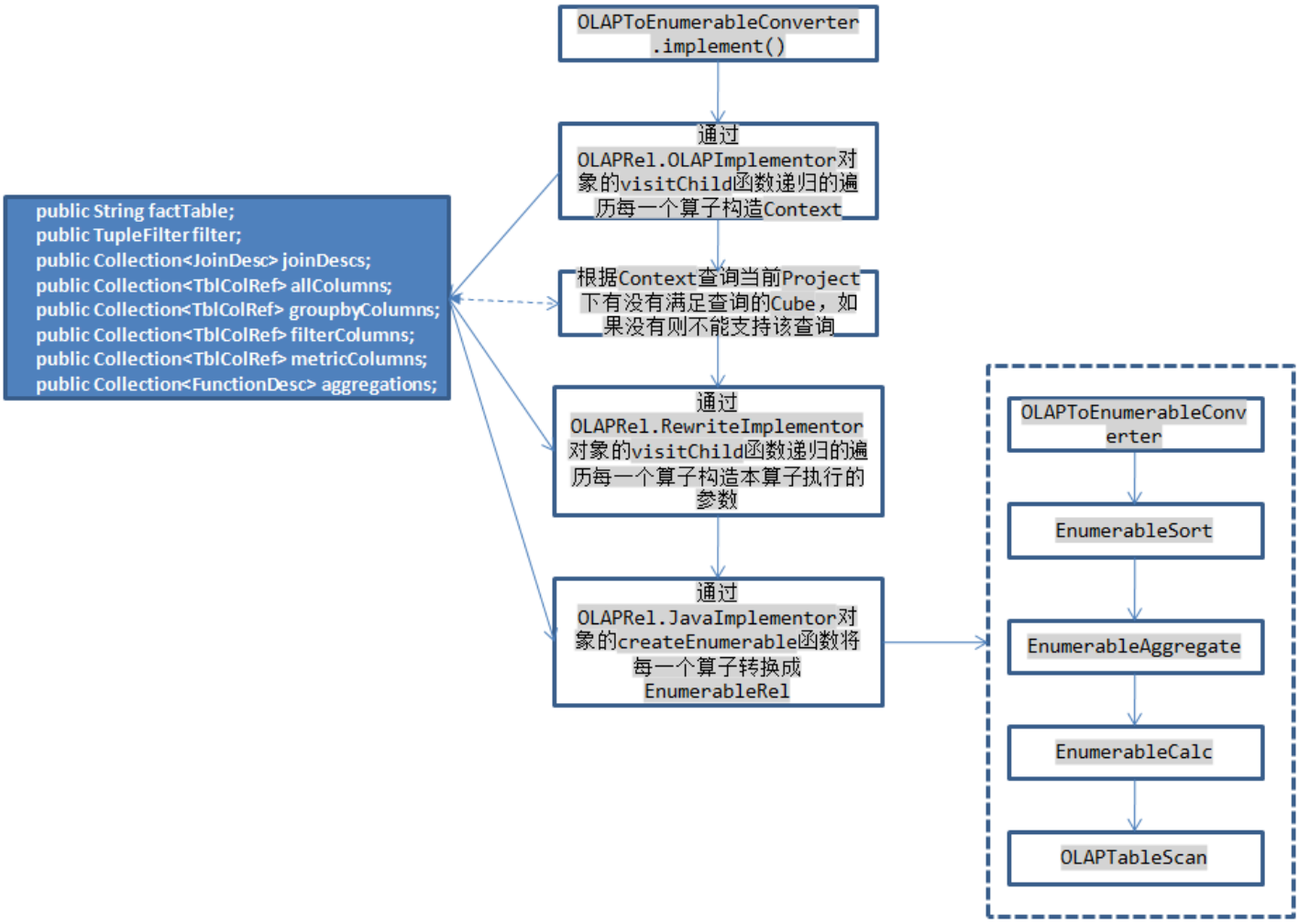
使用Calcite完成SQL解析，获取查询上下文



当在Calcite中执行一个SQL时，Calcite会解析得到AST树，然后再对逻辑执行计划进行优化，Calcite的优化规则是基于规则的，在Calcite中注册了一些新的Rule，在优化的过程中会根据这些规则对算子进行转换为对应的物理执行算子，接下来Calcite从上到下一次执行这些算子。这些算子都实现了EnumerableRel接口，在执行的时候调用implement函数：

```
public interface EnumerableRel
    extends RelNode {
  /**
   * Creates a plan for this expression according to a calling convention.
   *
```

```
   * @param implementor Implementor
   * @param pref Preferred representation for rows in result expression
   * @return Plan for this expression according to a calling convention
   */
  Result implement (EnumerableRelImplementor implementor , Prefer pref);
}
```



在所有Kylin优化之后的查询树中，根节点都是OLAPToEnumerableConverter，在它的implement函数中首先根据每一个算子中保持的信息构造本次查询的上下文OLAPContext，例如根据OLAPAggregateRel算子获取groupByColumns，根据OLAPFilterRel算子将每次查询的过滤条件转换成TupleFilter。然后根据本次查询中使用的维度列（出现在groupByColumns和filterColumns中）、度量信息（aggregations）查询是否有满足本次查询的Cube，如果有则将其保存在OLAPContext的realization中，获取数据时需要依赖于它。然后再rewrite回调函数中递归调用每一个算子的implementRewrite函数重新构造每一个算子的参数，最后再调用每一个算子的implementEnumerable函数将其转换成EnumerableRel对象，这一步相当于将上面生成的物理执行计划再次转换生成一个新的物理执行计划。

Calcite会根据这个执行计划动态生成执行代码，其中代码的生成根据每一个算子的implement函数构造，并且Calcite根据算子之间的依赖关系生成在新生成的类中构造bind函数，在bind函数中首先会执行TableScan获取数据，数据是通过一个Enumerable对象返回的，所以OLAPTableScan需要负责产生一个该对象获取原始数据，在执行moveNext获取下一条记录的时候通过filter中指定的条件对原始数据进行过滤，在current函数中执行映射返回select中指定的列数据，接着对这个Enumerable依次执行groupBy和orderBy函数，将结果返回。本次查询的statement会根据bind函数返回的Enumerable对象构造ResultSet对象。

上面大致上介绍了Kylin利用Calcite框架执行查询的流程，Kylin主要注册了几个优化规则，在每一个优化规则中将对应的物理算子转换成Kylin自己的OLAPxxxRel算子，然后再将每一个算子根据本次查询的参数生成Calcite自身的EnumerableXXX算子执行，比较特殊的是OLAPTableScan并不会转换成其他的算子，同样的还有OLAPJoinRel（当执行的sql有JOIN是会产生该算子），这OLAPTableScan算子的implement函数实现如下：

```
@Override
public Result implement(EnumerableRelImplementor implementor, Prefer pref ) {
    PhysType physType = PhysTypeImpl. of(implementor.getTypeFactory(), this.rowType , pref .preferArray());

    String execFunction = genExecFunc();
```

```
    MethodCallExpression exprCall = Expressions.call(table.getExpression(OLAPTable. class), execFunction , implementor.getRootExpr
    return implementor .result(physType , Blocks.toBlock( exprCall));
}

private String genExecFunc() {
    // if the table to scan is not the fact table of cube, then it's a lookup table
    if (context .hasJoin == false && tableName.equalsIgnoreCase(context .realization .getFactTable()) == false) {
        return "executeLookupTableQuery" ;
    } else {
        return "executeIndexQuery" ;
    }
}
```

可以看出它根据MethodCallExpression对象exprCall执行Blocks.toBlock生成对应的代码段（在bind函数中调用），例如本例中生成的代码段如下：

```
final org.apache.calcite.linq4j.Enumerable _inputEnumerable = ((org.apache.kylin.query.schema.OLAPTable)
     root.getRootSchema().getSubSchema("databaseName").getTable("tableName")).executeIndexQuery(root, 0);
```

返回的Enumerable是由executeIndexQuery函数返回的，在genExecFunc函数中会判断是根据之前生成的查询上下文OLAPContext，如果本次查询没有join并且查询的表不是当前使用的Cube的事实表，则使用executeLookupTableQuery函数，否则（有join或者查询事实表）则使用executeIndexQuery函数。

而在OLAPJoinRel的implement函数的实现则是直接使用executeIndexQuery函数。

```
@Override
public Result implement(EnumerableRelImplementor implementor, Prefer pref ) {
    PhysType physType = PhysTypeImpl. of(implementor.getTypeFactory(), getRowType(), pref.preferArray());
    RelOptTable factTable = context .firstTableScan .getTable();
    MethodCallExpression exprCall = Expressions.call(factTable.getExpression(OLAPTable. class), "executeIndexQuery" , implementor.
    return implementor .result(physType , Blocks.toBlock( exprCall));
}
```

为什么是这两个不同的函数呢？这是由于在Kylin中预计算了所有可能的组合值保存在hbase中，rowkey为值的组合，例如A="abc",B="xyz"就对应着一条记录，value为select count(1), sum(X) from table where A="abc" and B="xyz"的返回值，所以对于事实表中的数据都是需要进行计算的，保存在hbase中，只能通过访问hbase获取，而Kylin会保存所有维度表的信息，在内存中生成SnapshotTable，这样对维度表的查询则不需要扫描hbase了。

Kylin从Hbase中获取数据

上面吧Calcite解析和执行部分介绍完了，在bind函数中需要返回一个Enumerable对象给Calcite执行接下来的过滤、Project、groupBy、orderBy、limit等操作，这里不关注只对维度表的查询，而是看一下Kylin如何从Hbase中获取数据的。首先这个Enumerable对象时OLAPTable的executeIndexQuery函数返回的。

```
public Enumerable<Object[]> executeIndexQuery(DataContext optiqContext, int ctxSeq) {
    return new OLAPQuery(optiqContext, EnumeratorTypeEnum. INDEX, ctxSeq );
 }
```

它的enumerator函数如下：

```
 public Enumerator<Object[]> enumerator() {
    OLAPContext olapContext = OLAPContext.getThreadLocalContextById( contextId);
    switch (type ) {
    case INDEX :
        return new CubeEnumerator(olapContext, optiqContext);
    case LOOKUP_TABLE :
        return new LookupTableEnumerator(olapContext);
    case HIVE :
        return new HiveEnumerator(olapContext);
    default:
        throw new IllegalArgumentException("Wrong type " + type + "!");
```

```
    }
}
```

在CubeEnumerator中主要由current返回当前的数据，moveNext查看是否还有数据，它们完成了一个迭代器的功能：

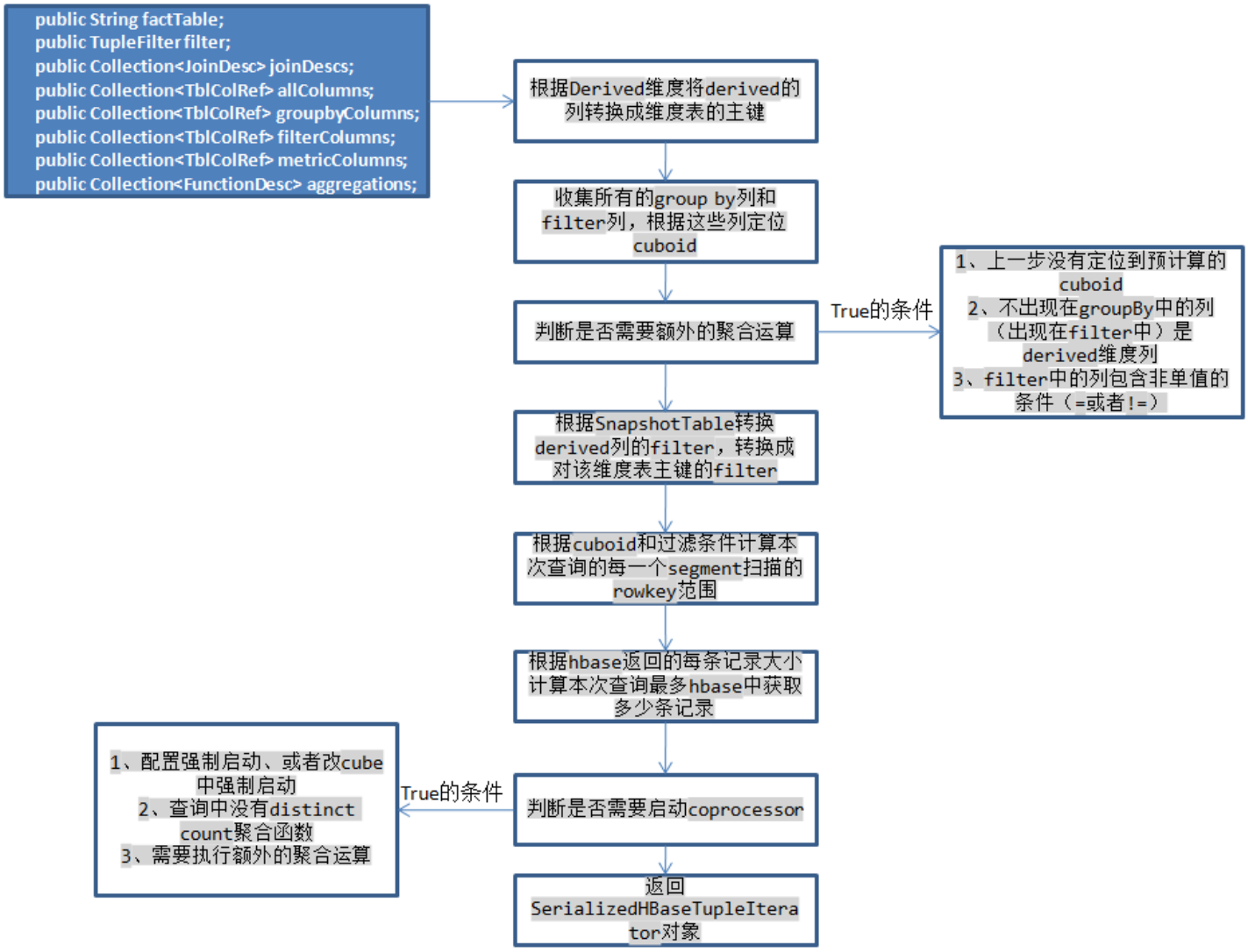
```
@Override
public Object[] current() {
    return current ;
}

@Override
public boolean moveNext() {
    if (cursor == null) {
        cursor = queryStorage();
    }

    if (!cursor .hasNext()) {
        return false ;
    }

    ITuple tuple = cursor.next();
    if (tuple == null) {
        return false ;
    }
    convertCurrentRow (tuple );
    return true ;
}
```

queryStorage函数返回一个迭代器，所有的数据都是通过这个迭代器获得，其中current变量是在covertCurrentRow函数中根据hbase中的数据解码之后的值，为什么需要解码呢？首先hbase中存储的都是二进制的数据，然后由于维度的成员的值可能会占用很大的空间，如果存储原始值的话会造成：1、hbase存储空间增大，2、相同cuboid的rowkey的长度不一样，所以Kylin在构建Cube的时候会将每一个维度下的成员进行编码，每一个维度中的每一个成员编码程一个从0开始的整数值，存储在hbase中的数据是这些编码值的二进制组合，因此读取到这些值之后需要解码获取原始的维度值。

querySorage函数主要执行逻辑：

```
    IStorageEngine storageEngine = StorageEngineFactory.getStorageEngine( olapContext.realization );
    ITupleIterator iterator = storageEngine.search(olapContext .storageContext , olapContext.getSQLDigest());
```

首先根据本次查询选中的Cube生成storageEngine对象，然后通过search方法返回一个迭代器，从其中获取全部数据。CubeStorageEngine是在Cube中获取数据使用的engine，它的search方法执行逻辑如下：



由于在线程局部变量中保存了本次查询的OLAPContext，可以根据它保存的信息获取根据哪些列group by和filter，以及对哪些度量进行计算，此时需要考虑derived维度，这种维度实际上会被它所在的维度表的主键代替，所以需要将这些列转换为主键列，并根据snapshotTable修改filter对象，然后判断本次查询是否需要启动hbase的coprocessor，Kylin对于每一个htable都设置了一个observer类型的coprocessor，当执行scan操作之前会回调这个类的doPostScannerObserver函数，执行对表中的原始记录执行一些过滤和聚合运算，这样可以减小每一个scan返回的记录数，例如执行select A，count(1) from table where B > 1 and C not in ('') group by A，这样的查询可以根据B>1计算出本次查询需要扫描的rowkey范围，而C not in ('')则需要在coprocessor对扫描获得的每一条记录执行判断，如果满足才可以从hbase中返回。例如上例中查询出现了A/B/C维度，但是这个cuboid并没有预计算，此时只能定位到A/B/C/D这个cuboid，在coprocessor中需要再根据D这一列执行聚合，进一步减小返回记录数。

关于内存

- 1、首先在coprocessor中，它是在hbase的regionServer中执行的，所以不能占用hbase太多的内存，Kylin在这里做的内存限制是500MB，因为需要执行额外的聚合运算，因此在coprocessor中维护了一个map保存每一个需要返回的记录并且持续的执行聚合运算，但是如果查询中带有distinct count的聚合运算，Kylin使用HLL实现的，每一个聚合值大概占用32KB大小（根据精确度），所以如果查询中有这样的聚合函数会很快消耗完这些内存，所以这种聚合的查询不会启动coprocessor。

- 2、对于返回的记录，只是原始的数据，需要再交给calcite完成下面的聚合、过滤和排序等操作，但是既然coprocessor中都已经把过滤和聚合做完了，为什么还要在coprocessor中做呢？filter的确是在Kylin中已经完成的了，再使用Calcite执行过滤是为了正确性的保证，但是这样也限制的Kylin不能支持全部的Calcite的过滤（这里可以扩展，Kylin只处理自己能处理的，剩余由Calcite处理），至于还需要聚合运算是因为一个Cube查询可能涉及到多个segment，因此这些segment可能返回相同的key，此时就需要Calcite执行聚合运算，运算函数是由Kylin指定的，但是需要将所有从hbase中返回的记录保存在内存中，Kylin为每一个查询设置了最大内存内存上线为3GB，根据每一个key-value的大小计算出hbase最多返回的记录数，如果超出这个数则根据配置是否接受部分结果，如果不接受则返回查询失败，如果接受则指根据已返回的记录进行Calcite的运算，可能出现错误。

获取数据

将filter转换为扁平式的使用AND连接filter，然后每一个childFilter可以根据不同的segment生成一个keyRange，这里成为ColumnKeyRange，每一个segment中有多个ColumnKeyRange，由于每一个segment对应着一个htable，所以首先会尝试merge每一个segment下的ColumnKeyRange（根据是否有重合的范围），生成多个HBaseKeyRange（merge之后的多个范围，直接对应着hbase中rowkey的范围），根据这些HBaseKeyRange生成SerializedHBaseTupleIterator。

在这个SerializedHBaseTupleIterator迭代器中按照每一个segment下的HbaseKeyRange创建一个map，segment为key，这个segment下需要扫面的HbaseKeyRange数组作为value，然后为每一个Segment创建一个CubeSegmentTupleIterator对象，它中保持了多个HbaseKeyRange，然后对每一个HbaseKeyRange创建Scan对象，接着使用该对象向Hbase发起一个scan请求，上面每一个迭代器都是对它包含的迭代器数组的封装。

总结

本文介绍了Kylin如何处理Sql的查询，充分利用了Calcite的sql解析和优化的功能，可以看到Calcite是一个非常强大的SQL引擎框架，Kylin较深入的定制了Calcite的功能，对于Calcite的初级使用可以参考：http://blog.csdn.net/yu616568/article/details/49915577，而Kylin提供从Hbase中读取数据返回前端又有点类似于phoenix的做法（它也是通过Calcite完成解析和优化的），但是后者更加通用一些。Kylin 2.0中把存储做成插件式的，理论上可以支持更多的存储组件（需要支持scan和类似coprocessor的功能啊），但是基本上查询流程是类似的。本文如果有什么错误，还请多多指正，谢谢~