Execution Primitives for Scalable Joins and Aggregations in Map Reduce

Srinivas Vemuri, Maneesh Varshney, Krishna Puttaswamy, Rui Liu
LinkedIn
Mountain View, CA

ABSTRACT

Analytics on Big Data is critical to derive business insights and drive innovation in today's Internet companies. Such analytics involve complex computations on large datasets, and are typically performed on MapReduce based frameworks such as Hive and Pig. However, in our experience, these systems are still quite limited in performing at scale. In particular, calculations that involve complex joins and aggregations, e.g. statistical calculations, scale poorly on these systems.

In this paper we propose novel primitives for scaling such calculations. We propose a new data model for organizing datasets into *calculation data units* that are organized based on user-defined cost functions. We propose new operators that take advantage of these organized data units to significantly speed up joins and aggregations. Finally, we propose strategies for dividing the aggregation load uniformly across worker processes that are very effective in avoiding skews and reducing (or in some cases even removing) the associated overheads.

We have implemented all our proposed primitives in a framework called Rubix, which has been in production at LinkedIn for nearly a year. Rubix powers several applications and processes TBs of data each day. We have seen remarkable improvements in speed and cost of complex calculations due to these primitives.

1. INTRODUCTION

Analytics on large data sets to extract insights is critical in today's world to make key business decisions. Typical batch-computation use cases for big data processing on MapReduce-based platforms involve joining massive datasets and performing aggregations on them. The state-of-the-art computation platforms, such as Pig [6], Hive [12], Cascading, Tenzing [8] etc, translate such data transformations down to physical level operators. The two significant operators are Join and Group By, and in our experience, the performance of these operators in existing platforms is sub-optimal.

Consider two datasets, dataset A with the schema (salesman, product) and dataset B with the schema (salesman, location). Both the datasets are big and hence cannot be completely loaded into all the mappers/reducers to perform replicated join. Consider the execution of the following query on these two datasets:
SELECT SomeAggregate() FROM A INNER JOIN B ON A.salesman = B.salesman GROUP BY A.product, B.location

Common implementations to execute this query, in Hive and Pig, do the following: Mappers load the two dataset and shuffle the datasets on the join key (salesman). The reducers perform the joins on (parts of) the two datasets they receive on the join keys, one unique join key at a time. The problem with this approach is that in order to compute the final aggregate, the joined data must be written to disk first, loaded in the next MR job and shuffled on the group by keys (product, location) to compute the aggregate.

An optimization that Hive provides to improve this execution is to compute the partial aggregate on the reducers, in the first job, right after join. This works well if the cardinality of the group by fields is low, which means that the hash table needed for partial aggregates is small and fits in memory. However, in practice, we see that the cardinalities of the keys seen are quite large, and we also see many more fields to group by. Together this explodes the hash table (HT) size requirement, which means the HT is flushed very frequently. In practice, this leads to spilling of raw join results to disk (taking us back to the previous problem).

Broadly speaking, both the key operators, join and group by, have two flavors of execution: merge-style which requires the data sets to be both partitioned and sorted on relevant keys (join keys and group by keys, respectively); and hash-style, which make use of in-memory hash-table and only requires the data to be partitioned on the said keys.

Computations on big data in contemporary systems, using a combination of above operators, suffer from several, and not infrequently all of the following problems. The intuition behind the cause of these problems goes back to the query execution example we gave above.

- Joining massive datasets generates data that is too large to be materialized (that is, writing to filesystem at the end of reducer phase or shuffling at the end of map phase). The only viable strategy is to perform inline aggregations along with joining the data.

- The merge-style group by operator is inefficient in big data contexts, since the output of the join must be materialized and shuffled on the group by keys – recall

Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08.

that the merge-style operator requires (and produces) data is sorted order of join keys and not the sorted order of group by keys. Hash-based group by is a better candidate; however, it typically happens that the size of hash-table is too large that it must be either spilled to hard-disk or flushed prematurely. In either cases, the effectiveness of hash group by deteriorates to that of merge-style (spilling of raw join results).

- Typically, it is not a single group by aggregate that must be computed. Rather multiple aggregates - called grouping sets - are computed. One strategy is to compute each group by within the set as a separate work flow, however this needs loading the input data as many times, which is usually impractical due to large size of the datasets. Another strategy is using the CUBE operator, as implemented in Pig or Nandi et al [11]. This strategy is also impractical as the CUBE operator generates all "ancestors" of each input row, which is simply too large to be processed.

- The aggregation process typically generates partial aggregates in one phase of the program (say, in mapper), shuffles them on the grouping keys, and aggregates them in the next phase (say, in the reduce). However, for a large datasets with large number of grouping keys, the volume of the partial aggregates is occasionally too large to be materialized.

- Finally, existing merge-style and hash-style operators are prone to the skew problem: wherein a subset of mappers or reducers can see a set of high frequency keys. This leads to disproportional distribution of work among the mappers/reducers leading to longer overall job run times.

The aforementioned problems are inherent in any non-trivial computation over big datasets. For example, the A/B testing platform at LinkedIn processes large datasets having 100s of Billions of records each day. Such systems perform complex statistical calculations via joins and aggregations on these datasets. In our experience, such calculations are not even feasible to perform using existing technologies even after optimizations (details in §6).

In this paper, we present Rubix: a framework for complex calculations over big data. Rubix is built on novel model of data organization using *Data Units*, and a novel computation paradigm using new operators: *MeshJoin* and *Cube*. By organizing datasets in calculation units, and processing them using the operators built to work on these units, Rubix is capable of avoiding all the problems listed above. To our knowledge, Rubix is the only framework capable of doing so. We describe our proposed primitives to accomplish this in this paper.

The rest of the paper is organized as follows. We describe our data organization framework in Section 2, and the operators and calculation paradigm in Section 3. In Section 4, we describe primitives for further improving the efficiency of our operators for complex calculations. Section 5 describes the implementation of Rubix, and Section 6 describes a case study of an application of Rubix within LinkedIn. We present performance measurements of Rubix in Section 7, and finally discuss related work.

2. DATA MODEL

In this section we describe the basic unit of data in Rubix, the reasons for designing them and how to create them.

2.1 Data Unit

Rather than tuple-oriented processing, Rubix defines a notion of Data Unit (a.k.a Block), which is the primary organizational unit of data, and operators work on these units. Data unit (DU) is the distinguishing feature of Rubix compared to related data flow and relational paradigms (such as Pig, Hive). DUs have the following important properties.

- A data unit is a subset of rows from the original data set partitioned by one or more keys of the data set, called *partition keys*, with data unit boundaries generated according to a user specified criteria. The partitioning scheme produces data units that are complete sub-set of the original data set such that all tuples from the original relation with the same value of the partitioning key are guaranteed to be present in the same data unit.

- Data units are further sorted on one or more columns, called *sort keys*. Note that only the data within a data unit is sorted (rather than globally sorted across the data units). By default, the data units are sorted on the partition keys; however, it is possible to sort the data units on other columns as well.

- Data units are created such that they satisfy a user-defined cost-functions. For e.g., that the data units created be no more than a certain size threshold. Setting the threshold at less than or equal to a few hundred MB would ensure that the data unit is of a size suitable for in memory processing. More sophisticated criteria of data unit generation can be chosen based on the specialized needs of an application.

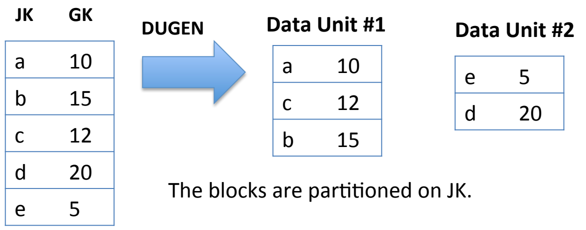
- Finally, data units are named entities, each unit has a data unit ID (DUID). They can be stored and loaded individually.

2.1.1 *Why Data Units?*

There are several reasons why data units are attractive. Data units bound the amount of work done by a worker, which balances the skew seen by the workers in a job. User-defined costs functions can be used to ensure that a data unit can be loaded in memory thus executing computations at memory speeds. Most importantly, it helps us in performing the calculation by loading the right data unit at the right place at the right time. All these properties lead to remarkable performance improvements, as we show in §7.

2.2 DUGen

The process of generating data units from input data source is called DUGen operation. Each generated data unit comprises of a range of partition keys and is uniquely identified by a data unit ID (DUID). A key difference between DUGen and file based partitioning is that the former prefers to co-locate data units into the same physical file as much as possible (so in theory, there is no direct relationship between the count of data units and the count of files). This is done by design in order that the file count is kept minimal, since it is known to have high meta-data overheads in file systems like the Hadoop Distributed File System (HDFS). The



The blocks are partitioned on JK.

Each block is internally sorted on a sort key GK.

Figure 1: DUGen Illustrated.

DUGen is implemented in Map-Reduce; the Mapper shuffles rows by the partition key. The sort key for the shuffle is set based on the DUGen configuration. On the reducer, we bundle rows into manageable (in-memory) segments based on the specified cost function, which are primarily sorted by the sort keys. When the data unit boundary criterion is satisfied (e.g. size exceeds the specified threshold), the data unit is written to disk on the reducer. The pseudo code for DUGen is described in Algorithm 1. We can rely on Hadoop's sorting during shuffle if the partition key is a prefix of the sort key; otherwise we create data units and sort the tuples within on the sort keys before giving them to the subsequent operator.

Data Unit Index. For every data set that is organized using the DUGen operation, there is an associated index that tracks data unit boundaries and the physical locations of data units on disk. When a data unit is written on the reducer, along with it, an index entry is updated which tracks the partition key range, as well as the unique DUID, corresponding to this particular data unit. The index entry also tracks the physical HDFS file name and beginning offset within this file that marks the start of the data unit. This index is very small in size as there is only one metadata entry in the index per data unit. Typically the index is in the order of KBs and hence can be cached in memory.

Cost Function. The DUGen operation also supports a pluggable cost function that determines where data unit boundaries are drawn. The framework provides built-in cost functions, for example, based on data unit size or row count that users can leverage without writing custom code. However, sophisticated criteria can be specified using the pluggable interface to perform custom calculations e.g. for graph calculations or cube computation. For instance, lets say we are interested in performing an aggregation such as a GROUP BY on a single large data set. A custom cost function can determine the data unit boundaries to be drawn when the number of distinct GROUP BY keys within a single data unit exceeds a memory threshold (say, 1MB) beyond which the results can no longer be kept/accumulated in memory.

The process of DUGen is illustrated in the figure 1. In this example, the data units are created by using the partition key to be the join key (JK), and the sort key to be the group by key (GK).

2.3 DUGen By Index

DUGen by index is the process of co-dependent organization of datasets using the Index based on the DUGen of a different dataset. Here, we have a first data set A that is partitioned on keys PK, along with some cost function. We also have IndexA, which is the index that demarcates data unit boundaries for this data set. Now, we can take a second data set B, which has the same partition keys PK and ask that data units be created on this data set using the same index, IndexA.

Algorithm 1 DUGen Pseudocode.

```
Input = dataset A;
Let SK be the sort key for data units of A;

function map ():
for row r of A do
   let PK = partition key of r
   emit (PK, r)
end for

function reduce (PK, list< $r$ >):
for each PK do
   add list< $r$ > to the current block;
   if current block reached cost function threshold then
      sort current block on SK;
      store current block;
      start a new block;
   end if
end for
```

The insight here is that the Index can be viewed as a function from arbitrary partition key to Data Unit ID. Note that list< r > need not be stored in memory on the reducer.

In order to perform DUGen by index, the Rubix framework loads the compact IndexA index into memory, assigns each tuple of data set B a data unit ID (DUID) column computed by looking up the value of partition key PK from the IndexA index, and shuffles tuples to the reducer based on the assigned DUID column. On the reducer, all tuples that map to the same DUID are accumulated and assembled into the same data unit. Note that the tuples corresponding to a data unit of B need not be kept in memory on the reducer and can be streamed to disk as the memory buffers keep filling up. In addition, we can use the sorting performed by the shuffle phase of MapReduce to keep the data sorted on sort keys, by setting the sort key to be the combination of (ID, SK). Note that we have no control over size of data units of dataset B, as indeed we did for the DUGen of the original data set A. The data unit sizes of the data units for dataset B are output parameters i.e. determined by the pre-existing data unit boundaries indicated by the IndexA index.

The DUGen by index Pseudocode is in Algorithm 2.

2.4 Pivoting Data Units

Given a data unit created on a set of sort keys, it is often necessary to obtain the rows that have identical values for the sort keys. For instance, to compute a group by on the sort keys. The process of accepting one data unit and a set of keys (called pivot keys) as input, and generating multiple output (sub-)data units where each output data unit consists of rows with identical pivot keys, is called *Pivoting the data units.* We will use this operator to write pseudocode of other operators in the paper. A key insight here is that the sub-data units output from the pivotBlock operation can be treated as a data unit by itself. Typically such sub data units can be fully contained in memory and thus leads to

Algorithm 2 DUGen by index Pseudocode.

```
Store IDX = Index of A in Dist Cache
Input = dataset B;
Let SK be the sort key for data units of B;

function map ():
for row r of B do
    let PK = partition key of r
    let ID = DUID of PK from IDX
    emit (ID, r)
end for

Shuffle on ID and sort on (ID, SK);

function reduce (ID, list < r >):
store list < r > as one block;
```

speed up in computations. The pseudocode is presented in Algorithm 3. The user of this API calls morePivotBlocks() as long as it returns true and calls pivotBlock() to obtain a pivoted sub-block within the loop.

Algorithm 3 pivotBlock Pseudocode.

```
Input = block1 of a dataset;
function pivotBlock(pivotCols):
let pivot be pivotCols of block1's currentTuple
initialize a new subblock
while block1.currentTuple.get(pivotCols)==pivot do
    add currentTuple to subblock;
    advance to next tuple
end while
return subblock

function morePivotBlocks():
return (currentTuple==null)
```

2.5 Common Calculation Patterns on Data Units

We will now describe how data units can be used to solve common problems seen in processing relational queries.

2.5.1 Joins On Two Data Sets

Consider two tables: TableA has member Id and the dimension values for the member such as the country, industry, education, etc (memberId, dim0, dim1, dim2, dim3). The second table, TableB, has details about the pages viewed by a member, including the memberId, pageId, and other page view attributes such as the date of view, device from which it was viewed, browser viewed, etc. (memberId, pageId, att0, att1, att2). Assume that these two tables are large and do not qualify for replicated joins (which loads one dataset completely on all the mappers/reducers) or its variants.

In this setup, we are interested in counting the number of page views by the members on a given day, but we want to filter out the records only based on certain dimensions or UDFs (user defined functions) on some dimensions. In this example, the partition, the group by, and the join key are all the same – memberId.

Using data units, this can be accomplished by performing a DUGen on TableA first to have, say 10M, members per data unit. Then we can do a DUGen by index on TableB. The sort and partition keys are both memberId in this case. Then we can simply load the corresponding data units (as in the data units that have the same set of members) and perform a merge-join followed by filter and aggregation on the mappers of the subsequent job.

A main advantage of creating data units is that the cost function bounds the amount of work done by each mapper for join and hence avoids skews during join. We will describe how the cost of DUGen can be reduced (or even removed) later in this section.

In Pig/Hive one could perform this by doing a shuffle join on the join key and aggregate on the reducer. But they are prone to skews.

2.5.2 Group Bys on a Single Fact Table

If we need to perform a group by computation on a single fact table, even then using DUGen has advantages over existing implementations. Pig, for e.g., shuffles the fact table on the group by key and then would compute the aggregate on the reducer. There are a couple of issues with this approach: One is skew, and the other problem is due to lack of sort order on the tuples. For a group of tuples in the same group by key, the tuples need not be in any particular order and hence aggregate computations can be inefficient. Count distinct computation, for instance, would require a hash table per group.

Rubix, on the other hand, would do a DUGen on the group by keys. Aggregates could be computed efficiently by setting the appropriate sort keys. And if there are multiple group bys to be computed, we could store the data units perform the aggregate computation on all the group bys in the follow on job (using the operators described next). Data unit cost function avoid skews in the work done by these worker nodes.

2.5.3 DUGen for Group Bys

Building on the previous example, suppose that the query is to count the number of page views, broken down by different combinations of dimensions. a) broken down by member dimensions dim0, and dim1. b) broken down by dim0 and att0. c) broken down by att0 and att1. In these cases, the partition and the join key is the memberId, but the group by keys are combinations of keys from the two tables and they are different from the join key.

Performing this aggregation on Pig/Hive would require materializing the join results and then shuffling them on the group by keys, which leads to the problems described in §1.

We can avoid such problems in Rubix by organizing the data into data units by having different sort keys and this can speed up aggregation (as described in the next section) and avoid shuffling join data. The organization of the data is as follows: first we perform a DUGen on TableA with memberId as the partition key. But we would have sort keys to be the keys that are contributed by this dataset into the group by keys. Then we extract the index of these data units and use it to perform a DUGen by index on TableB again using memberId as the partition key. But we set the sort keys for TableB to be the keys it contributes to the group by keys. We will show how aggregation is done next.

3. COMPUTATION MODEL

We introduce two new operators: Mesh join as an alternative to traditional join, and Cube as an alternative to

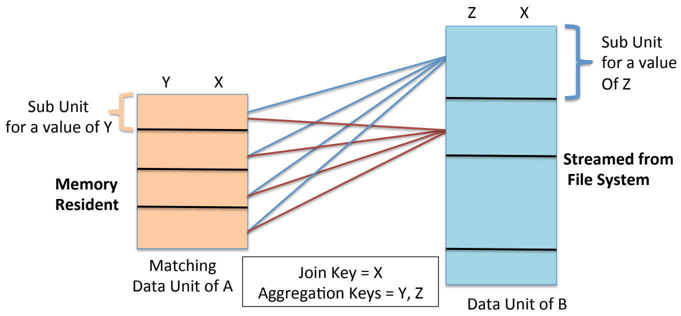


Figure 2: MeshJoin Illustrated.

traditional group bys on big data. These operators are designed to take advantage of the Data Unit organization and significantly reduce the overheads involved in aggregation.

3.1 Mesh Join (or Cyclic Join)

Recall the SQL query we introduced earlier where group by keys were coming from different data sets.
SELECT SomeAggregate FROM A INNER JOIN B ON A.j1 = B.j2 GROUP BY A.g1, B.g2.

MeshJoin Requirement. Traditional merge and hash-style operators are not efficient at processing such queries. Merge join requires the two data sets to be partitioned and sorted on join keys, and it retains the sort order on the join keys for the output. The hash join neither requires sort order on input datasets nor provides any sort order on the output, as a result, computing group bys on the results requires a (large) hash-table for aggregation or requires a shuffle and aggregation on the reducer.

The MeshJoin operator, on the other hand, is capable of computing the final aggregate without requiring a shuffle or hash-based aggregation. The MeshJoin operator requires data partitioned on join keys, but allows data to be primarily sorted on other keys, lets call them **prefix keys**, and secondary sorted on the join keys. The generated output is sorted on combined prefix and join keys. By setting the prefix keys to be that of the following group by keys, we can compute the results of the group by immediately after the join. This is merge-style operator, as in, it does not require in-memory hash table, it benefits from cache locality, and can compute group by results with O(1) memory.

Consider the Example:
Table A: [g1, j1], col3
Table B: [g2, j2], col4. (bracket indicates sort order)
Output: [g1, g2, j1], col3, col4

The prerequisite for applying MeshJoin operator is the one data is DUGen-ed and partitioned on join keys but sorted on prefix keys of this data (g1 in the e.g.). The size constraint on the blocks are set to ensure that the blocks of A (or one dataset) can reside completely in memory. The other dataset is DUGen BY INDEX on the join key and sorted on corresponding prefix keys (g2 in the e.g.). This requirement has been illustrated earlier in §2.5 with the examples where group by keys were combinations of the non-join keys from the two tables.

MeshJoin Operator Execution. Figure 2 illustrates the process in which MeshJoin operates with an example. The operator works by loading a block of A into memory on each mapper, then it streams the matching block of the other dataset, so each mapper is processing exactly one pair of matching blocks from A and B. We stream the block from B (as we do not have control on its size). For each set of records which correspond to one distinct value of the pivot column on B (i.e a distinct value of g2), we load all of the records into memory and perform the join with the in-memory records of A. Since the result is sorted on A.g1, B.g2, we need to join the pivoted sub-block of B with each batch of As records corresponding to a distinct value of A.g1. So the join is actually on sub-blocks generated from the main block by pivoting along the prefix keys. And this join has to be performed iteratively for every unique combination of (A.g1, B.g2).

If we need to compute a group by on (A.g1, B.g2), we can immediately aggregate the results of the join and generate a partial result for this combination with O(1) memory requirement. This is output to the reducers for further aggregation to produce the final result (from different mappers).

Then we move onto the next batch of As records. Once we reached the end of block for A, we advance the input stream for Bs block, to obtain a new value for B.g2, reset the block of A to the beginning (this can be done as it is in memory) and repeat the steps outlined above for the new batch. The Pseudocode for the MeshJoin operator is shown in Algorithm 4.

The above strategy is a nested loop join in which a single pass is made over the unbounded block from B, while multiple in-memory scans are done on the memory-bounded block of A. While the idea of partitioning data on the hash of the join key and joining the partitions are proposed in the past [10], MeshJoin is different due to the sort order of the tuples and the nested loop join on the sub-blocks.

Algorithm 4 MeshJoin Pseudocode.

```
Input = blocks of dataset B;
function map (block1 of datasetB):
blockA1 = LOAD-BLOCK FROM datasetA MATCHING block1.blockId
for block2 = block1.pivotBlock(B.g2) do
    for block3 = blockA1.pivotBlock(A.g1) do
        block5 = JOIN block3, block2 ON A.j1 = B.j2
        results = GROUP block5 BY A.g1, B.g2 agg()
        emit results
    end for
    blockA1.rewind()
end for
```

To further reduce the data loaded in memory, note that we can actually partition the sub-block of B (generated from DUGen by index) by the join key. As in, instead of loading all the tuples of a unique value of B.j2, we could load only a sub-set of these tuples at the boundary of B.j2 (this subset can be bound by the size of memory available). We just need to ensure that all the tuples for a value of B.j2 are all in this subset. This can be accomplished by extending the pivotBlock pseudocode to take a partition key (which is B's join key in MeshJoin).

3.2 Cube Operator

Definition. In this paper we use the term Cube to encompass the traditional OLAP cube operator, which computes the group bys on all the combinations of dimensions (cuboids) *as well as* the idea of grouping sets, where a subset of the cuboids are explicitly specified for computation. A grouping set is a list of group by key combinations. For instance, a grouping set with two group by entries or cuboids is specified as: ((g1), (g1, g2)). The output of the computation has three columns: g1, g2, aggregate. The values for g2 would be NULL for the group by results corresponding to the cuboid (g1). If the grouping set has a single grouping set entry, then it degenerates into a single group by computation. If the grouping set is empty, then the entire OLAP cube is computed.

Ancestor enumeration. Based on the definition above, cube operator is capable of generating results for the grouping sets, as in multiple group bys can be computed, in a single job. As a result, each tuple "contributes" data to a set of grouping key entries. Given a tuple, the "ancestors" for this tuple are all the parent cuboids of the dimensions in the tuple. In the case of full OLAP cube, each tuple contributes to all 2^D ancestors, where D is the number of dimensions. In the case of a grouping set, each tuple contributes to as many entries present in the grouping set.

For executing the Cube operators we need both the mapper and reducer tasks. The mapper does most of the work: The ancestors are maintained in a hash table keyed on the ancestor (dimension values for the ancestor cuboid), and the corresponding value stored in the HT is the aggregation results for that cuboid. For each tuple, this HT is updated based on which ancestors a tuple contributes to (we will describe why this hash-table size in Cube operator is not prohibitive in §4). The mapper then shuffles the partial aggregates with the ancestor as the key. The reducer simply computes the final aggregate using the partial aggregates from the different mappers for a given ancestor.

3.2.1 Additive Cube Operator

Aggregation functions. Additive aggregates include functions such as sum, min, max, count, etc. The notation of the operator is: *CUBE ADDITIVE dataset on* **measure** *BY GROUPING SET aggregate().*

Requirement. The additive cube operator does not require any partitioning or sorting on the data as additive operators are algebraic.

Operator execution. The operator execution is as follows. For each tuple in the input data unit, the operator enumerates the ancestors based on the input grouping set. The hash table maintained for the ancestor is checked for the presence of each ancestor. The partial result for the ancestors present in the hash table is updated and a new entry is created in the HT for the ancestors not present already. After processing a tuple, if the HT size exceeds the user-specified memory threshold, then the results computed so far are flushed and the HT is cleared before proceeding to the next tuple. The Pseudocode for the Additive Cube operator is shown in Algorithm 5.

3.2.2 Partitioned-Additive Cube Operator

Aggregation functions. Partitioned-additive aggregates are aggregation functions such as count distinct, and other complex aggregates (as described in detail in §6) typically needed for statistical calculations. When partitioned appropriately, these functions become additive across partial results and hence the name.

Algorithm 5 Additive Cube Operator Pseudocode.

```
function map (block of datasetB):
let H = hashtable grouping keys => (Aggregator)
for tuple r in block do
    let G = grouping keys in r
    let E = set of "ancestors" for G
    for each e in E: do
        invoke Aggregator
    end for
    flush and clear the HT if it exceeds the threshold
end for
```

We can model such aggregates using two functions: innerAggregate() (which is a per-member aggregate) and a outerAggregate() (which is a cross member function). For count distinct, for instance, the inner aggregate is countOnce() and outer aggregates is a sum(). For variance calculation on per-member sum (see §6), the inner aggregate is a sum, while the outer is a sum of the squares.

For partitioned-additive aggregates, we only focus on queries where measure (memberId, for e.g.) is the partition key and on which aggregates are computed. In most of the queries we see in LinkedIn, memberId is the typical measure on which aggregates are computed and it is the partition key. Rubix came out of the use cases of computing complex aggregates for memberId data. As a result, we made this decision. Extending our operators to work for measures that are not partition keys is left for future work.

The notation of the operator is: *CUBE dataset on* **measure** *BY GROUPING SET innerAggregate(), outerAggregate().*

Requirement. Our partitioned-additive Cube operator requires data to be partitioned and sorted on the measure (memberId). The input is the data units after DUGen on the measure. This is a basic approach without considering the type of dimensions. In the next section we will show how to optimize this further by considering the type of dimension into account.

Operator execution. The Cube operator works as follows. Each mapper in a job takes a data unit as input. Then the operator creates a pivoted block of the data unit on the measure (effectively gets a sub-data unit with rows having the same measure). For each tuple in the sub-data unit we enumerate the ancestors. In this operator, the hash table is keyed on the ancestor but the value stored is the results of the two aggregation functions. While processing each tuple in the sub-block, if an ancestor is found in the hash-table, then the (inner) aggregate values are updated otherwise a new key and value is inserted into the hash table. When a sub-data unit is done processing, the outer aggregate is updated for the ancestors in the hash table. After processing each sub-block of the measure the hash table size is checked, and it is emitted out of the mapper if the threshold is crossed. Then we move on to the next sub-block. After the mapper is done, the combiner aggregates the results flushed to the disk by the mapper (with the same key) and then the results are sent to the reducer. The Pseudocode for the Cube operator is shown in Algorithm 6.

We noted earlier in the introduction section, that a typical problem with big data is that the hash table size explodes and it needs to be spilled to disk or flushed too often. Even

Algorithm 6 Partitioned-Additive Cube Operator Pseudocode.

```
function map (block of datasetB):
let H = hashtable grouping keys => (Inner Aggregator,
    Outer Aggregator)
P = pivotedBlock of B on measure
while P.morePivotBlocks() == true: do
    Q = P.pivotBlock()
    for tuple r in Q do
        let G = grouping keys in r
        let E = set of “ancestors” for G
        for each e in E: do
            invoke InnerAggregator(r)
        end for
    end for
    invoke OuterAggregator
    flush and clear the HT if it exceeds the threshold
end while
```

though the Cube operator uses hash table, we will show in the next section how we avoid this problem, by creating the hash table only on a subset of the grouping key values.

4. SCALING MULTI-DIMENSIONAL CALCULATIONS INVOLVING COMPLEX JOINS AND AGGREGATIONS

In this section we describe our primitives for improving the efficiency of computing the aggregates by taking into account the type of dimensions involved in aggregation. We start with a a few definitions, followed by a model for calculating and comparing the cost of different algorithms and then propose our primitives for bringing down the cost.

Types of Dimensions We classify the dimensions into two groups: member dimensions and context dimensions. Member dimensions are the dimensions that describe member attributes. A member has a single value for each dimensions. For e.g., country of a user. Context dimensions, on the other hand, describe the context of events. For e.g., PageView. When a user views a page, there are dimension values such as the browser and the device type used to view the page, etc. A member can have many values for such dimensions. Context dimension is used in a general sense here: some datasets have grouping keys that are not dimensions in the traditional sense but can have multiple values for a member. We refer to them also as context dimensions.

	md0 (10)	md1 (10)	cd0 (10)	cd1 (10)
GS1	x	x	x	
GS2			x	x
GS3		x	x	
GS4	x		x	x

Table 1: An example grouping set with four entries.

4.1 Metrics

Given a set of (member and context) dimensions and the cardinality of these dimensions, they together determine the (a) total number of dimension keys generated in the hash table for aggregation, as well as (b) the total number of dimension keys shuffled across the mappers/reduces. These are the two main metrics that determine the overhead and performance of the operators. Our goal is to minimize both these values so that computations can be done efficiently.

This cost can be derived using a matrix of dimensions and grouping sets. Table 1 shows an example with 4 dimensions and 4 grouping sets. 2 of these dimensions (md0, md1) are member dimensions and 2 are context dimensions (cd0, cd1). We can use this matrix to gain an intuition on how dimension cardinality and grouping sets have an impact on the two metrics.

Given a grouping set entry (say GS1) with a set of dimensions, the total number of entries created in the hash table for computing the grouping set entry is equal to the product of the cardinality of the dimensions in it. This is also equal to the number of keys emitted from the mappers to the reducers. Note, however, that this estimation of the number of keys is a lower bound. If the in-memory size of the keys generated by a mapper is bigger than the available memory capacity of the mappers, then it requires frequent flushing of the partial results to the reducers leading to increase in the total number of keys shuffled.

Following the logic above, the total number of keys to be maintained in the hash table and shuffled, across all the grouping set entries, is simply the sum of the cardinality of each entry. More formally: Let d be the set of dimensions, and gs be the grouping set. The number of keys in the hash table for a grouping set entry gs_i is $s(gs_i) = \prod_{d_j \in gs_i} |d_j|$, where d_j is the cardinality of the dimension d_j in gs_i. The total number of HT keys, and the keys shuffled, for the entire grouping set is: $ts = \sum_{i=1}^{n} s(gs_i)$.

Our objective is to bring down these numbers. Next we propose three strategies for organizing and partitioning the data before using the MeshJoin and Cube operators that can lead to remarkable reduction in these numbers.

4.2 Promoting a Key to MeshJoin Prefix Key

If all the grouping set entries to be computed have a certain grouping key in them, then this key can be moved to be a prefix key of the mesh join. This means that the blocks input to the MeshJoin operator is sorted using this prefix key as the sort key. By moving this key to be the part of the MeshJoin, the aggregates can be computed after the join for each sub-block containing the mesh prefix key. This means that the hash table size needed to store the aggregates is equal to the product of the cardinality of the remaining dimensions (excluding the mesh prefix). This is because MeshJoin processes the sub-block with the same prefix and this means that the hash table for aggregation is keyed only on non-prefix keys. In many cases, this strategy, in fact, makes it *feasible* to do the calculation and keep the keys in the hash table. This strategy does not, however, lead to a reduction in the volume of keys shuffled though.

Example. In the example in Table 1, cd0, the first context dimension is present in all of the grouping set entries. As a result, we can apply this strategy and move it to be a part of the MeshJoin key. This brings down the HT size requirement by a fact of 10, which is the cardinality of cd0. To compute GS1, for e.g., for a sub-block of the two blocks, during MeshJoin, the HT only needs to store the number of keys equal to the cardinality of $|md0| * |md1|$ which is 100 (coming down from 1000).

4.3 Member Dimension-Aware Partitioning

Partitioning the dataset on the member dimensions leads to a significant reduction in both the costs. By partitioning on the member dimensions, the range of the member dimensions calculated in each mapper is constrained to a small part of the entire dimension range. As a result, each mapper needs to keep a much smaller number of keys in its hash table for aggregation.

We can perform member dimension partitioning of the dataset as follows. Consider two datasets, TableA for member data and TableB for fact data to be partitioned on member dimensions. We first do a DUGen on TableA on memberId, then DUGen by index on TableB, then we join the appropriate Data Units of TableA and TableB on the mapper of the subsequent job and perform a DUGen with member dimensions as partition and sort keys. During this DUGen, by setting appropriate cost functions, we can roughly assign equal amount of work to the mappers/reducers.

This partitioning brings down the cost on both the metrics by a significant factor. If N is the total number of member level dimension key combinations and B is the total number of data units, the number of member dimension combinations per unit is reduced to N/B. Typically B is in 1000s, which means if there are a million unique dimension key combinations, they come down to 1000s per mapper (as we usually assign a small number of data units per mapper).

An important property of partitioning on the member dimensions is that it also partitions the memberId space. That is, the set of member Ids are disjoint in different data units. As a result, computation of holistic (partitioned-additive) aggregates becomes *additive across different data units*. We take advantage of this property in the next primitive also.

Given that context dimensions of a member can have many different values, partitioning on context dimensions generally leads to a huge skew as well as it is useless for computing holistic measures (because a member can end up in multiple mappers and we cannot combine holistic aggregates across mappers). As a result, when data is partitioned on dimensions, we only partition them on member dimensions and not on context dimensions.

Example. Continuing with the example in Table 1, when the dataset is not partitioned on the member dimensions, potentially each mapper can generate a key for each of the member dimension combinations – and there are 120 of those (100 for GS1, 10 for GS3, 10 GS4). When we partition the dataset on the member dimensions, we can bring this down by a significant factor (proportional to the number of mappers). In fact, we can set the cost function such that the generated keys can fit in the memory of the mapper, thus speeding up the computation. This partitioning also partitions the ancestor keys and thus leads to proportional reduction in the number of keys shuffled.

4.4 Cubing on Context Dimensions using Staged Aggregation

Cube computation can be made more efficient by breaking down the computation into stages. By computing the Cube on the product dimensions in one MR job, and then cubing the member dimensions in a subsequent job, we can significantly reduce the memory requirement and the volume of the keys shuffled across M/R stages.

Consider, for e.g., a cube computation on dimensions (md0, md1, cd0, cd1). Assuming all of them have a cardinality of 10, the Cube on md0 and md1 will have totally 122 unique keys. Similarly (cd0, cd1) cube will have 122 keys. If the Cube on all four dimensions are calculated, the total number of keys in the hash table, and the number of keys shuffled, would be about 14644 (per mapper).

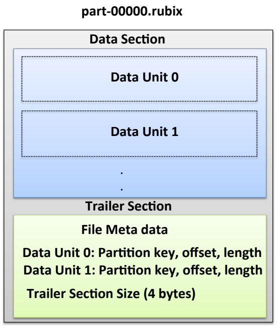

part-00000.rubix
Data Section
Data Unit 0
Data Unit 1
Trailer Section
File Meta data
Data Unit 0: Partition key, offset, length
Data Unit 1: Partition key, offset, length
Trailer Section Size (4 bytes)

Figure 3: Rubix file format.

Assume that the previous strategy of member dimension partitioning is applied and the blocks are created by partitioning and sorting on (md0, md1). Now in the mapper we can cube only on (cd0, cd1). By doing so, we have effectively brought down the hash table to 122 keys per unique key value of (md0, md1) seen. This is much smaller memory explosion compared to full cube on all four dimensions. As a result, this job can aggregate the results for the product dimension cube much better in the available memory without having to flush partial results frequently.

Since we know that the member dimension aggregates (even partitioned-additive) across data units can be combined. In the second job, we compute the cube on (md0, md1). Here also we have small memory requirement compared to full cube computation. Because of better aggregation, we can expect significantly smaller number of keys to be shuffled in both the jobs combined compared to single-stage execution. The final number of result tuples produced, though, would be the identical in both the approaches.

5. IMPLEMENTATION

So far we have described the data units and the main operators that lead to improved efficiency. Now we describe our implementation of the framework, how these operators are executed and how a user can program in our framework.

We have implemented the Rubix framework in Java, using Hadoop [4] 1.1.2 with MapReduce [9] APIs. The entire framework is about 65K lines of code. In the framework we have implemented (a) the operators for creating data units and the operators for join and aggregation on these units, and (b) an executor service that takes in a script consisting of the operators to run, compiles it to a sequence of map reduce jobs and executes the jobs on Hadoop. We describe the key highlights of our platform implementation starting with the custom file format we use to store data units.

5.1 Rubix File Format

We have implemented Custom RubixInputFormat and RubixOutputFormat classes in Hadoop. This enables us to store the data generated after DUGen in our custom format, called the Rubix File Format. This is only for the intermediate DUGen data, though. The final results can be in whichever format the user desires (AVRO, TEXT, etc.).

Using this custom output format, all the data the reducer of DUGen receives is stored by creating a Rubix file. The data it receives is organized into data units, which are all stored in a single output file. The file has enough index information for us to learn the boundaries of the block or data units. The Rubix file format is shown in the Figure 3.

Each Rubix file has data section (containing data units) and a trailer section (containing partition key range, offset within file and length of each unit). The data section has data units written as a byte stream of serialized tuples. Whenever a data unit ends, we add an entry in the index indicating the end of the previous data unit. The current position in the file is the start offset for the new data unit created. The index that is maintained while writing the data units is stored in the trailer section. After writing the index section, we reserve four bytes to indicate the size of the trailer section.

When a Rubix file is read, using the custom input format, we read the last four bytes first to figure out the size of the trailer section. We seek to the begin of this section and then read the metadata (or the index) to learn the data unit locations in the Rubix file. This index is then used to either figure out the number of splits for MapReduce job execution or to read a specific data unit (for load block operation).

5.2 Rubix Job Specification and Execution

Specification. Our framework implements the following operators: load, store, join, group by, cube, dugen, dugen-by-index. A user can specify the operations to perform on a data set via a Rubix script consisting of a series of job. Each Rubix job consists of two stages: Map stage and Reduce stage. The input script specifies the sequence of operators for both these stages.

In addition to specifying the sequence of operators, the script also specifies the various inputs and output files for the jobs, as well as specify how the data is partition and sorted between the Map and Reduce stages.

An example job is shown in the Algorithm 7. The code specifies a single map-reduce job, with 100 mappers and 50 reducer processes. The paths to datasets for input and output are specified, along with the storage format (AvroStorage and Rubix in this example). Two set of operator sequences are defined, for mappers and reducers, respectively. The output of mapper operator sequence is partitioned on (colA) and sorted on (colB, colA) before delivering it as input to the reducer operator chain.

A computation may comprise of multiple jobs, and each job can be represented by a block of code between begin and end job. Dependencies between the jobs can be established by their input and output requirements.

Execution. A *job executor* parses the input script and executes these jobs, in the order of their dependencies on each other, as separate Map-Reduce programs. First, the job executor determines the order in which the jobs need to be run by performing a topological sort based on the input and output of the jobs. Once the order of the jobs is determined, the execution of each job is the responsibility of the *phase executor*. Independent jobs are run in parallel.

The phase executor first topologically sorts the (directed acyclic graph of) operators with in a phase. That is, the operators themselves are ordered in the order in which they need to be run (for instance, in a left to right ordering, an operator can depend only on the operators to its left in the ordering).

Algorithm 7 Example job specification

```
Begin Job
Name "example job"
Num-Mappers 100
Num-Reducers 50
Input inputpath Using AvroStorage
Output outputpath Using Rubix
Begin Map // sequence of operators to execute
b1 = LOAD Input
End Map
PARTITION b1 ON colA SORT ON colB, colA
Begin Reduce // reducer operator sequence
b2 = CREATE BLOCK FROM b1 BYSIZE 100MB ON colA;
STORE b2 as Output;
End Reduce
End Job
```

The phase executor instantiates the operators in order of the topological sort of the operator sequence. The ordering ensures that when an operator is instantiated, all its parent operators are already instantiated. The phase executor then determines the last operator, which is the output of this sequence, and now it pulls data from this operator; and this operator, in turn, pulls data from its parent operators. The data is, therefore, successively pulled from child operator from their respective parent operators. The phase executor pulls data from the last operator, until no more data is available. When this happens, the phase executor determines that the operator sequence is exhausted and the current phase is terminated.

6. CASE STUDY: LINKEDIN'S EXPERIMENTATION PLATFORM

Rubix is actively used in production by various use cases. We describe one particular use case in detail where we have applied the proposed primitives and used all aspects of the system describe so far.

Rubix is a key component powering LinkedIn's experimentation platform. This is the A/B testing platform for experimenting with new features on the site and measure their impact on various different metrics. We describe the problem at an abstract level to keep the description simple.

Rubix is used to address joins and aggregations over two large (independent) streams of datasets. One stream of data set, let us call it, Table A, has information about various metrics measured on the site and the value of these metrics. These metric values are at a per member level, and hence contain (metric, value, memberId, date). The second stream, called Table B, contains details about the members' assignment to experiments and associated attributes, such as the variant of the experiment (control/treatment), segment, and date (experiment, segment, variant, memberId, date). Experiments are added and retired daily, and each experiment chooses a set of users based on various dynamic conditions. These tables already have data aggregated on a per-day basis (by upstream jobs).

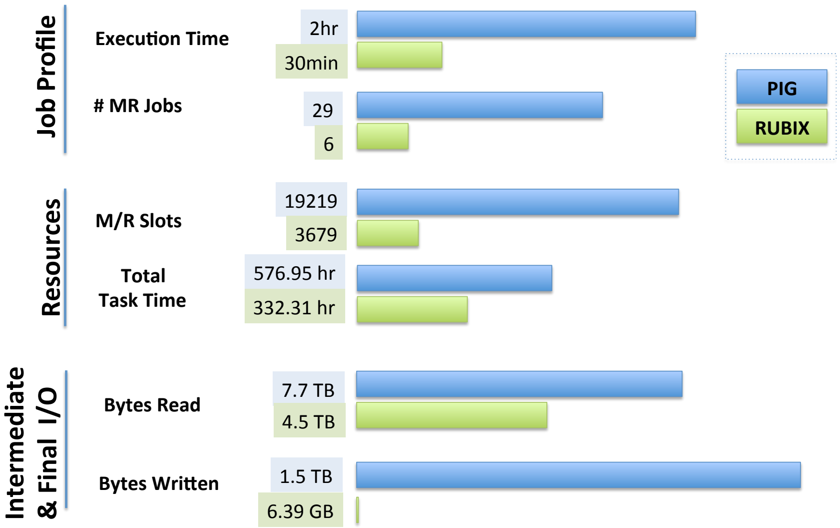


Figure 4: Performance comparison of a script in production running on Pig vs Rubix.

Analyzing an A/B test involves statistical calculations such as mean, variance, etc. These calculations are run not just on a daily basis, but also run across days, even up to months. These calculations are driven by another dataset, Table C, which includes the (multiple) date ranges for which an experiment's impact needs to be measured: (experiment, start date, end date). While joining the first two datasets is complex enough, due to the dimensionality of the fields, this third dataset effectively leads to another explosion in the joins along time dimension. Finally, the time ranges we process the total data for can go as far back as three months.

To give an idea of the number of records, Table A and B contain about 70 Billion (about .5TB in size) and more than 500 Billion records (about 10TB in size) over three months. There are about 6000 entries in Table C. Ultimately we need to compute mean and variance to generate (experiment, segment, variant, metric, start date, end date, mean, variance). Lift and statistical significance calculation for a set of metrics for all members who were exposed to multiple variants of the same test involves a join on memberId and time range. Variance involves the sum of square calculation (which is quite well-known). In this case when generating lift and statistical significance reports for multiple time ranges, the correct way to perform the calculation is to compute the total member response for each time range and then take the sum of squares since the memberId is both the randomization and analysis unit [13]. This requires going back to the base dataset and hence cannot be done using daily aggregates. This means that we need to process all these datasets each day in entirety.

Experience Running on Hive Before building the Rubix framework, in fact, we tried to compute these aggregates on Hive (about a year ago). Despite months of effort by several developers and heavy optimizations, Hive did not process all these queries successfully, even though the data was much smaller back then. At the time, even optimized hive queries on a smaller dataset took over 40 hours and still did not finish successfully. In order to execute these queries faster, we started building out the basics of a framework, which lead to Rubix.

Performance on Rubix. To give an intuition on the savings in memory size of the hash table during computation, here are some concrete numbers. We see about 500 distinct metrics, about 2000 unique (experiment, segment, variant) combination, about 30 date ranges per grouping key. This means that we see about 30M unique grouping keys. This is without any dimension-based drill downs. 30M keys do not fit in 1GB of memory (usual configuration we use for mapper and reducers). MeshJoin brings down the memory requirement to O(1). In another flow, we perform the same computation with member dimension drill downs. With 5 different dimensions, each having about 10 values on an average, even if we assume only 10% of the values are seen, we see a 5x explosion of the hash table size in naive joins/aggregations. In other flows we see drill downs on 2-level and 3-level combinations of dimensions which further increase the number of entires by 10s of times.

Rubix is able to perform these computations in the order of hours. About 2 hours are spent in doing DUGens on the two datasets. The calculation of the aggregates take between 1.5 hours to 4 hours depending on the level of drill downs. All the mappers and reducers are configured to consume only 1GB of memory. The times reported include quite a bit of wait time for mappers and reducers to get scheduled (which depends on the load on the cluster). These calculation jobs have been in production since August 2013 on Rubix, as a key component powering the A/B testing platform.

7. EVALUATION

We present evaluation results from two applications that are using Rubix.

7.1 Join and Grouping Set Computation

We translated an existing script that used to run on Pig into Rubix and then ran the two scripts on identical datasets. The script mainly involved joining two large datasets (after a set of filters) on memberId and computing 8 group bys with count distinct aggregate on memberId. The input datasets were of sizes 2TB and nearly 200 GB. These datasets had about 20B records and over 350M records respectively. We present a comparison of the resources consumed and the time taken by the two scripts in Figure 4.

When we executed the script in Pig, it compiled our script down into a series of 29 MapReduce jobs. Notably, each group by was computed in a separate job where each job read the results of the join. Rubix, on the other hand, was able to perform everything in the script in 6 MR jobs by packing the operators more compactly. As a concrete example, our Cube operator could compute the 8 group bys in a single job, there by reducing the 7 jobs just from aggregation part of the script. Other optimizations include not materializing the join results to disk, and avoiding duplicate reads of the same dataset, which Pig performed due to what seemed like a sub-optimal plan generation.

In terms of the total execution time, Rubix was about 4x faster than Pig. The reasons for this speed up includes fewer jobs spawned, less data read, less data written, and much fewer overall number of map and reduce slots. Reducing the number of jobs not only eliminates the job run time, but also the wait times in the queue. The MR slots were significantly reduced to 3.6K from nearly 20K, a big chunk of that was due to avoiding separate jobs for group bys and avoiding unnecessary reads. Only final results and DUGen results were written out by Rubix, while Pig wrote temporary results from each job. The total task time was only cut down by about half because computing the group bys were quite time consuming compared to reading or writing the data (in this script).

We have seen similar advantages in other scripts we have translated to Rubix. As a result, Rubix is seeing faster adoption to replace such Pig scripts within LinkedIn.

7.2 Cube Computation

This application is using Rubix to run OLAP Cube on a fact table with 10 dimensions to compute count distinct on the measure. This team started using Rubix after Pig was unable to complete this query in under a day despite significant optimizations. We took a sample data, of about 100M tuples, from this application and ran it for varying number of dimensions on both the optimized Pig script and the corresponding Rubix script to compare their performance.

Figures 5 and 6 show the run time and the number of tuples shuffled between the M/R stages during cube computation. Note that both the graphs are in log scale. Rubix script used the Cube operator for this query. Even though Rubix had an additional job to do DUGen on the fact table, the run time taken was nearly 2 orders of magnitude smaller. For 8 dimensions, Rubix took about 13 mins including DUGen but Pig took over 5 hours. For 10 dimensions, Rubix took a little less than an hour, but Pig was projected to take over 43 hours. Rubix's remarkable performance is due to in-memory aggregation performed by the Cube operator using the hash table before flushing partial results to the reducer. This aggregation reduces the number of tuples shuffled between M/R stages by about 2 orders of magnitude as seen in Figure 6, which leads to the corresponding improvement in run time. For instance, for 8 dimensions, Pig shuffled about 15B tuples while Rubix only shuffled 82M tuples. We had set the hash table size to be 2M entries (with 1GB memory for the mappers), but this can be increased to improve aggregation (and hence reduce the tuples shuffled and improve performance) further.

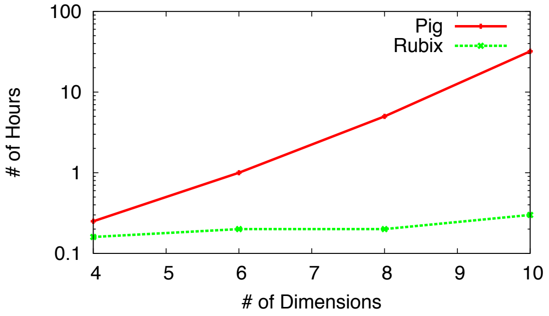


Figure 5: Time taken for Cube computation.

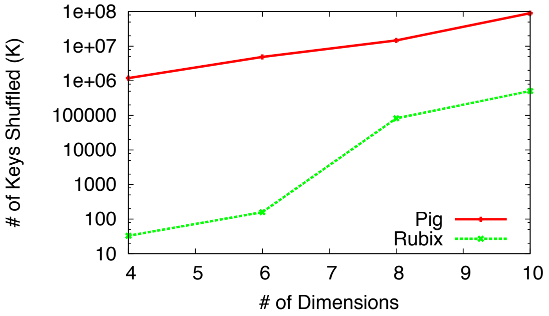


Figure 6: Number of Keys shuffled across M/R.

8. DISCUSSION

Data Units need to be created first in our model before running computations. There are common scenarios where we can reduce or even remove the overhead of DUGen.

Incremental DUGen. When a calculation processes data spread over weeks or even months, the time spent during DUGens re-organizing the data can be expensive. To alleviate this we can adapt an incremental strategy that only re-organizes the data generated for the current day using the same index as the consolidated and re-organized data set inclusive of all days other than the current day. Thus we have matching pairs of DUs, a large unit from the original consolidated data set, and a small unit from the daily delta which was DUGen-ed using the same index. We then provide a merged Input format, which offers an abstraction of a unified data unit that contains tuples from both the large and the small data unit in the order corresponding to the sort key.

The incremental delta DUs created on a daily basis, however, should be merged into one consolidate unit at some point. This could either be done after the daily calculation or on a periodic basis (once every week, etc.). This way the DUGen overhead is removed form the path of the calculation SLA.

Data Units using Persistent Indices. Shuffle join is a commonly used operator and we can completely avoid the jobs needed for DUGen and DUGen by Index by caching indices on frequently seen join keys and datasets. We can maintain these indices on join keys as persistent objects (created upon first run) and use them to shuffle the two datasets (member data and another fact table for join, for e.g.), we can then dynamically create DUs on the reducer and join the

DUs. The index used for DUGen constrains the size of the DUs of the primary data set to be memory resident. This primary data unit tuples appear first on the reducer (accomplished by setting appropriate comparator properties). Then an in memory block is created for the primary. The secondary data unit tuples are streamed and we can do a merge join.

9. RELATED WORK

Hive [12, 5] and Pig [6] have a few optimization to improve the efficiency of joins on large tables. Hive's "skewed" join aims to reduce the skew in joins by sampling the data first and computing a histogram of the key space. Then it partitions one of the tables using this histogram to partition the key space in a skew-aware manner, and streams the other table. Hive's bucket map join is an optimization where the tables are divided into buckets (or partitions), and the appropriate buckets from the two data sets are loaded on the mappers to perform the join. Both these optimizations, however, are not effective in improving the performance of aggregation. Due to lack of control on the sort order of the data within a bucket or partition, they still require a shuffle of the joined data in order to compute the aggregates.

A recent paper, Nandi et al. [11] proposed partitioning strategies for scaling the computation of OLAP Cubes in MapReduce setting. Specifically, they proposed value partitioning for semi-algebraic aggregates, where in a sampling program estimates the number of records for different regions of the cube and uses it to partition the tuples shuffled to avoid over-burdening a reducer. While this is an improvement over the naive cubing implementation on MapReduce, this still requires shuffling a large number of tuples (each tuple explodes to as many tuples as the number of ancestors to compute) across mappers and reducers. Our proposed operators, on the other hand, performs aggregation on the mappers and shuffles only partially aggregated results.

Cascading [1] and Scalding [2] are frameworks designed with a focus on making it easier to program MapReduce jobs via simple Java and Scala APIs. While there are a few optimizations specifically on join strategies, the primitives in Rubix are more general and can improve the efficiency of join and aggregation operators.

Apache Tez [7] is a framework that aims to improve the performance of the Hadoop framework itself by offering patterns such as Map-Reduce-Reduce, and providing the ability to run a DAG of jobs etc. The primitives offered by Rubix and Tez are complimentary in nature. While Rubix currently works on Hadoop, by implementing the same primitives on Tez, we believe, that the applications can see amplified performance improvements.

Apache Spark [3] is a framework specifically designed for interactive and iterative computations such as machine learning and graph computations. Resilient Distributed Datasets (RDDs) in Spark, are particularly efficient when the cached data fits in memory and it is reused in future iterations. Rubix on the other hand has focused on the problem of performing joins and aggregations on large data sets, that do not fit in memory, efficiently. To our knowledge, Spark, as is, will shuffle datasets for joins and afterwards before group bys, there by hitting the same limitations as that of Hive or Pig. Rubix blocks, on the other hand, might be useful for iterative computations, which we plan to explore as a part of future work.

10. CONCLUSIONS

We have described Rubix, LinkedIn's contribution to Big Data analytics platforms. Rubix has several novel primitives for scaling complex joins and aggregations on Big Data. We described the data organization primitives, computation primitives, and primitives for partitioning the data to reduce skews and aggregation overheads. We have shown how these primitives help in reducing the cost of aggregation inherent in existing implementations of Map Reduce-based frameworks for query execution. We are leveraging the advantages of Rubix in production at LinkedIn for various applications. We wish to offer these advantages to a larger community and are working on open-sourcing Rubix in the near future.

11. ACKNOWLEDGMENTS

We are grateful to many folks who have helped us in building Rubix. We wish to thank the Data Analytics Infrastructure team and the management for all their support. We specially want to thank our early adapters in the company: the XLNT team, the Plato team, and the SPI team.

12. REFERENCES

[1] Cascading. https://github.com/Cascading/cascading.

[2] Scalding. https://github.com/twitter/scalding/wiki.

[3] Spark. http://spark.apache.org/.

[4] Apache. Apache hadoop. http://hadoop.apache.org/.

[5] Apache. Apache hive. http://hive.apache.org/.

[6] Apache. Apache pig. http://pig.apache.org/.

[7] Apache. Apache tez. http://hortonworks.com/hadoop/tez/.

[8] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. In *Proceedings of VLDB*, 2011.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[10] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[11] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Data cube materialization and mining over mapreduce. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1747–1759, 2012.

[12] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Proceedings of ICDE*. IEEE, 2010.

[13] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer, 2004.