Distributed Aggregation in Cloud Databases

By: Aparna Tiwari
tiwaria@umail.iu.edu

ABSTRACT

Data intensive applications rely heavily on aggregation functions for extraction of data according to user requirements. Various cloud platforms adapt different strategies for implementing these aggregation functions. The performance of these functions is affected by various factors such as their internal implementation in systems, functionality, the size, indexing and storage of the underlying data.

The paper discusses the implementation of user defined and partial aggregation in Map Reduce. It also mentions one level, two level and three level aggregations in Dryad LINQ.

Later, it evaluates these aggregation functions on the basis of their performance, usability and functionality.

INTRODUCTION

Aggregation is an increasingly important operation in data-intensive applications that are designed to execute on large computing clusters. As data sets grow larger, there is growing emphasis on not just extracting the data, but also general characterizations of large subsets of data.

For computations such as matrix multiplication and graph traversal in distributed programming models, the most efficient available mechanism is grouped aggregation. Such algorithms typically require nonstandard aggregations that are more sophisticated than traditional built-in database functions such as Sum, Min, Max etc. As a result, the ease of programming of user-defined aggregations, and the efficiency of their implementation, is of great current interest.

DISTRIBUTED AGGREGATION

There are several functions that are used in order to perform general-purpose user-defined aggregations. The data consumed by an aggregation function can be retrieved from database, based on some key function. In some other cases, the data consumed is the output of more complex processing such as a Join or a previous aggregation.

This paper discusses the iterator based programming model adapted by MapReduce and Hadoop and discusses the alternatives adapted by Dryad-LINQ.
MAP REDUCE

The computation in Map Reduce has exactly two phases: the first phase executes a Map function on the inputs to extract keys and records, and then performs a partitioning of these outputs based on the keys of the records. The second phase collects and merges all the records with the same key, and passes them to the Reduce function. (This second phase is equivalent to GroupBy followed by Aggregate in the traditional database.)

The optimizations for distributed aggregation rely on computing and combining partial aggregations. Partial aggregations may exist, for example, when the aggregation function is commutative and associative. When there is such substantial data reduction, partial aggregation can be introduced both as part of the initial Map phase and in an aggregation tree, as shown in Figure 2, to greatly reduce network traffic.

Figure 1: A case in Map Reduce, when reduce cannot be decomposed to perform partial aggregation.
In order to enable partial aggregation a user of MapReduce must supply three functions:

1. **InitialReduce**: \(<K; \text{Sequence of } R> \rightarrow <K;X>\) which takes a sequence of records of type \(R\), all with the same key of type \(K\), and outputs a partial aggregation encoded as the key of type \(K\) and an intermediate type \(X\).

2. **Combine**: \(<K; \text{Sequence of } X> \rightarrow <K;X>\) which takes a sequence of partial aggregations of type \(X\), all with the same key of type \(K\), and outputs a new, combined, partial aggregation once again encoded as an object of type \(X\) with the shared key of type \(K\).

3. **FinalReduce**: \(<K; \text{Sequence of } X> \rightarrow \text{Sequence of } S\) which takes a sequence of partial aggregations of type \(X\), all with the same key of type \(K\), and outputs zero or more records of type \(S\).

This concept unnecessarily introduces three functions, even for simple computations like integer average.
DryadLINQ

DryadLINQ integrates relational operators with user code by embedding the operators in an existing language, rather than calling into user-defined functions from within a query language like Pig Latin or SQL. For example, a distributed grouping and aggregation can be expressed in DryadLINQ as follows:

```csharp
var groups = source.GroupBy(KeySelect);
var reduced = groups.SelectMany(Reduce);
```

In this fragment, source is a DryadLINQ collection (which is analogous to a SQL table) of .NET objects of type R. KeySelect is an expression that computes a key of type K from an object of type R, and groups is a collection in which each element is a group" (an object of type IGrouping<K,R>) consisting of a key of type K and a collection of objects of type R. Finally, Reduce is an expression that transforms an element of groups into a sequence of zero or more objects of type S, and reduced is a collection of objects of type S. DryadLINQ programs are statically strongly typed, so the Reduce expression could for example be any function that takes an object of type IGrouping<K,R> and returns a collection of objects of type S, and no type-casting is necessary. Aggregation without grouping is expressed in DryadLINQ using the Aggregate operator.

The DryadLINQ interface gives user the flexibility to implement the aggregation using one level, two level or three level aggregation. The one level and two level aggregation can be implemented using Join() operator. However, the algorithm for three level aggregation is different from the former ones and is implemented using ApplyPerPartition() function. The input is first split into partitions and each partition is stored in a separate file. Each map task processes one partition. At the end of computation, each Map task performs some initial merge for partial values and do the second level aggregation. In the last stage, Main program collect the merged values from each compute node and do the third level aggregation.

![Figure 3: Three level aggregation in DryadLINQ.](image-url)
Comparison of Map Reduce and DryadLINQ aggregation operators

The evaluation of these aggregation functions on several cloud platforms can be done in various ways such as –

1. The degree of language integration between user defined functions and the high level query language
2. The choice of programming interface
3. The programming model supplied for user defined aggregation
4. The performance of these functions on a variety of workloads.

The implementations that use bounded memory at the first stage, but achieve lower data reduction, complete faster than those which use more memory, but output less data, in the first stage. The Map Reduce strategy of using a very large number of small input partitions performs substantially worse than the other implementations due to the overhead of starting a short-lived process for each of the partitions.

CONCLUSION

When the aggregation involves more complex user-defined functions and data types, the database programming interface can become substantially more difficult to use than DryadLINQ. Databases generally adopt accumulator based interfaces. These consistently outperform the iterator interfaces used by systems like MapReduce.

When a userdefined function is easily expressed using a built-in database language the SQL interface is slightly simpler, however more complex user-defined functions are easier to implement using MapReduce or Hadoop. When sophisticated relational queries are required native MapReduce becomes difficult to program. In some ways, DryadLINQ seems to offer the best of the two alternative approaches: a wide range of optimizations; simplicity for common data-processing operations; and generality when computations do not fit into simple types or processing patterns.

An accumulator-based interface for user-defined aggregation can perform substantially better than an iterator-based alternative.
REFERENCES

1. Map Reduce - [http://en.wikipedia.org/wiki/MapReduce](http://en.wikipedia.org/wiki/MapReduce)

2. Online Aggregation and Continuous Query Support in MapReduce -
   neilconway.org/docs/sigmod2010_hop_demo.pdf

3. Distributed Aggregation -
   citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.4415.

4. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations - [http://research.microsoft.com/apps/pubs/?id=102138](http://research.microsoft.com/apps/pubs/?id=102138)

5. Online Aggregation -

6. Dryad and Dryad LINQ for data intensive research -