THE INTERNATIONAL JOURNAL OF SCIENCE & TECHNOLEDGE

Incremental Join Aggregation Using Map Reduce

Dhananjay M. Kanade
Department of Computer, PG Student, KKWIEER, Nashik, University of Pune, India
Dr. Shirish. S. Sane
Professor**,** Department of Computer, KKWIEER, Nashik, University of Pune, India

Abstract:
Fast and efficient retrieval mechanisms that work within acceptable time frames and keep pace with the information explosion are essential for efficient and effective database systems of the future. Using more processing power, and in particular, exploiting parallel processing is the intuitive solution. While there have been many successful efforts that aim at parallelizing query evaluation, it is possible to further improve the performance of a database system using newer techniques such as Map/Reduce.
It is observed that the parallel DBMS perform well because they load and index the data before they process queries. Since this loading phase can take a long time, there arises the question if the loading phase itself is worth to process only one or two queries. This is where Map/Reduce comes into play. One such idea for processing enormous quantities of data is Google's Map/Reduce. It is a simple yet powerful framework for efficient execution of complex queries over large datasets involving joins and aggregation.
The work reported in this paper deals with the design of a proposed framework based on Map/Reduce for efficient execution of complex database queries involving joins and aggregations. The user needs to only write specialized map and reduce functions as a part of the Map/Reduce job and the framework takes care of the rest. The design of the framework has been done after exploring existing solutions and then extended to propose an efficient solution for queries involving joins. Join algorithms are of two types - Two-Way joins and Multi-Way joins. This work deals with only two-way joins. Options to pre-process data in order to improve performance at map side have also been explored. Experimental results presented provide an insight into how good a fit Map/Reduce is for evaluating joins with aggregation at reduce side.

1. Introduction

The join and aggregation operator has been a cornerstone of relational database system since their inception. As much time and effort has gone into making join and aggregation efficient. With obvious trends towards multiprocessors, attention has focused on efficient The significant amount of effort has been focused on developing efficient join and aggregation algorithms. Initially, nested loops and sort-merge were conventional algorithms of choice for join operations. Besides being faster under most conditions these join and aggregation operation required complex query to be executed. Loading huge database in memory with parallel query execution required huge resources.

The conventional database management system cannot store intermediate result in memory but it passes to the next operation, failure in one process can be a failure in the overall operation.

Map Reduce is simple and efficient for computing join aggregate. Thus, it is often compared with "filtering then group-by aggregation" query processing in a DBMS. The main idea of the Map Reduce model is to hide the details of parallel execution and allow users to focus only on data processing strategies. The Map Reduce model consists of two primitive functions: Map and Reduce, which parallelize the join and aggregation operation. But a traditional database system with query engine have some limitation running the job in a parallel way. If data added in the database the incremental computation will have to perform the traditional database management system execution of a query need to perform all join and aggregation operation from the beginning .

Using map reduce computations can respond automatically and efficiently with modifications to their input data by reusing intermediate results from previous runs, and incrementally updating the output according to the changes in the input. Which will reduce time complexity

2. Architectural Approaches

In more complex forms, map reduction jobs are broken into individual, independent units of work and spread across many servers, typically commodity hardware units, in much less complicated and easily managed by many computers connected together in a cluster. In layman's terms, when the task at hand is too big for one person, then a crew needs to be called in to complete the work. Typically, a map reduction "crew" would consist of one or more multi-processor nodes (computers) and some type of master node

or program that manages the effort of dividing up the work between nodes (mapping) and the aggregation of the final results across all the worker nodes (reduction). The master node or program could be considered the map reduction crew's foreman. In actuality, this explanation is an over-simplification of most large map reduction systems. In these larger systems, many additional indexing, I/O, and other data management layers could be required depending on individual project requirements. However, the benefits of map reduction can also be realized on a single multi-processor computer for smaller projects.

The primary benefits of any map reduction system come from dividing up work across many processors and keeping as much data in memory as possible during processing. Elimination of I/O (reading data from and writing data to disk) represents the greatest opportunity for performance gains in most typical systems. Commodity hardware machines each provide additional processors and memory for data processing when they are used together in a map reduction cluster. When a cluster is deployed however, additional programming complexity is introduced. Input data must be divided up (mapped) across the cluster's nodes (computers) in an equal manner that still produces accurate results and easily lends itself to the aggregation of the final results (reduction). The mapping of input data to specific cluster nodes is in addition to the mapping of individual units of input data work to individual processors within a single node. Reduction across multiple cluster nodes also requires additional programming complexity. In all map reduction systems, some form of parallel processing must be deployed when multiple processors are used. Since parallel processing is always involved during map reduction, thread safety is a primary concern for any system. Input data must be divided up into individual independent units of work that can be processed by any worker thread at any time during the various stages of both mapping and reduction. Sometimes this requires substantial thought during the design stages since the input data is not necessarily processed in a linear fashion. When processing text data for instance, the last sentence of a document could be processed before the first sentence of a document since multiple worker threads simultaneously work on all parts of the input data.

3. Two-Way Joins

3.1. Reduce-Side Join

In this algorithm, as the name suggests, the actual join happens on the Reduce side of the framework. The 'map' phase only pre-processes the tuples of the two datasets to organize them in terms of the join key.

3.1.1. Map Phase

The map function reads one tuple at a time from both the datasets. The values from the column on which the join is being done are fetched as keys to the map function and the rest of the tuple is fetched as the value associated with that key. It identifies the tuples' parent dataset and tags them. A custom class called Text Pair has been defined that can hold two Text values. The map function uses this class to tag both the key and the value.

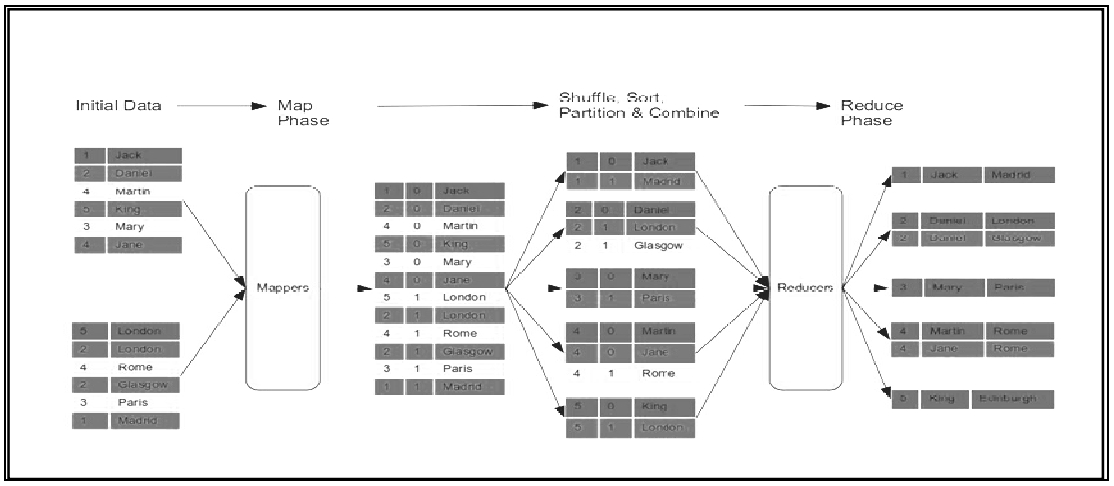

Fig. 3.1: Map and Reduce side join

3.1.2. Partitioning and Grouping Phase

The partitioner partitions the tuples among the reducers based on the join key such that all tuples from both datasets having the same key go to the same reducer. The default partitioner had to be specifically overridden to make sure that the partitioning was done only on the Key value, ignoring the Tag value. The Tag values are only to allow the reducer to identify a tuple's parent dataset.

But this is not sufficient. Even though this will ensure that all tuples with the same key go to the same reducer, there still exists a problem. The **reduce** function is called once for a key and the list of values associated with it. This list of values is generated by grouping together all the tuples associated with the same key. It is sending a composite TextPair key and hence the Reducer will consider (key, tag) as a key. This means, for eg., two different **reduce** functions will be called for [Key1, Tag1] and [Key1, Tag2]. To overcome this, we will need to override the default grouping function as well. This function is essentially the same as the partitioner and makes sure the reduce groups are formed taking into consideration only the Key part and ignoring the Tag part.

3.1.3. Reduce Phase

The framework sorts the keys (in this case, the composite TextPair key) and passes them on with the corresponding values to the Reducer. Since the sorting is done on the composite key (primary sort on Key and secondary sort on Tag), tuples from one table will all come before the other table. The Reducer then calls the reduce function for Join Algorithms. Each key group. The reduce function buffers the tuples of the first dataset. This is required since once the tuples are reading it lose access to these values unless it reinitialize the stream. But we need these tuples in order to join them with the tuples from the second dataset. Tuples of the second dataset are simply read directly from, one at a time and joined with all the tuples from the buffer (all the values with one reduce function are in the same key) both key and values need to be tagged. This is because the tag attached to the key is used to do a secondary sort to ensure all tuples from one table are processed before the other. Once the reduce function is called, it will only get the first (key, tag) pair as its key (owing to the custom grouping function written which ignores the tags). After j tuple of first dataset join with the tuple of second dataset.

The aggregation algorithm can be performed in one MapReduce job as follows. Mapper extract from each tuple values to group by and aggregate and emits them. The intermediate outputs produced by the mappers are then sorted locally for grouping key-value pairs sharing the same key. After local sort, Combine () is optionally applied to perform pre-aggregation on the grouped key-value pairs so that the communication cost taken to transfer all the intermediate outputs to reducers is minimized.

Reducer receives values to be aggregated already grouped and calculates an aggregation function. The first job does the 2-way join as it did in the problem above, then the second job does all the grouping and aggregation. Grouping is done by MapReduce; all it required to do is to output the grouping attribute(s) as the intermediate key in map:

4 . Incremental Aggregation

The system can reuse the result of prior computations transparently by using intermediate result produce at Map side. As for real time database the synthetic database is produced, which is added in an incremental way.

The algorithm is composed of the following steps:

- Import the synthetic database on the real time database.
- Assign map reduces function for aggregation operation.
- Re-compute the aggregation result using the intermediate result stored.

5.Experiments

5.1. Dataset

Data are the set of tuples. Tuple is split into a pair of a key and a value, where value is the remaining attributes. Generation of synthetic data was done. Join keys are distributed randomly and in uniform manner.

The first data set employed is "DATAMART" for the experiments which is a real world relational database. The ” DATAMART” data set consist of two tables. This database have of skewed key distribution. The Second data set employed for the experiment is a real world relational database collected from the online store.The “eSTORE” data set consists of two tables. This database have uniform key distribution.

5.2. The General Case

One of the most important functional requirements of a database system is its ability to process queries in a timely manner. This is particularly true for very large, mission critical applications such as weather forecasting, banking systems and aeronautical applications, which can contain millions and even trillions of records. The work presented in this report focus is to join the relation using Map reduce that support incremental updating.

Map Reduce is excellent in the areas of simplicity, fault tolerance and flexibility. Map Reduce consists basically of two functions, Map and Reduce, Avoids a long loading phase. If the data is processed only once and it changes occur frequently over time, this approach can be very useful and preferable. Its fault tolerance model brings an important advantage, as datasets become bigger. But this fault tolerance is not for free. The materializing of intermediate results on local disks causes a bad impact on the performance. Nevertheless, such a programming model with this fault tolerance will become more preferable as datasets grow.

The join algorithm can be implemented in a different ways. In terms of disk accesses, the join operations can be very expensive, so implementing and utilizing efficient join algorithms is important in minimizing execution time. Such datasets on which join and aggregation operation is performed produces the intermediate result on local disk. Those intermediate results are used in incremental aggregation computation as the data set grows.

A series of experiments were conducted to compare the runtime and scalability of the algorithms. Involving running two-way join algorithms on database (skewed key distribution and uniform key distribution database) of different sizes. The performance of system is measured over the real word database as well as synthetic database.

The first data set employed for the experiment is a real world relational database collected from online eStore

In incremental aggregation data added to the existing database. In a traditional database management system incremental computation will have to perform the execution of the query. All query operations like join and aggregation are performed again from beginning.

Map reduce computations responds automatically and efficiently to modifications to their input data by reusing intermediate results from previous runs, and incrementally updating the output according to the changes in the input. Which reduces time complexity. Where the traditional query approach performs all the join and aggregation from the beginning. As traditional query operation does not save intermediate result it passes the result from one process to another.

The experiment for incremental aggregation uses a synthetic database. Synthetic database is added in base table incrementally. Synthetic database tested in an incremental way for the eStore. Range of dataset given INC_RECORDS in table 5.1 which is added incrementally to original database. The key column show on which key new records are inserted.
Table 5.1 shows time in second taken by the traditional query processing approach versus Map reduce on an increasing number of records sizes on the existing database.

INC_RECORDS	Traditional Incremental (Time in Sec)	Map Reduce incremental (Time in Sec)	Key	Records
26674	250.4	239.38	6212	500
27674	135.98	128.99	6168	1000
29174	134.26	129.23	6790	1500
31174	133.38	129.12	19212	2000
33674	345.58	322.4	32264	2500
36674	438.18	405.47	6788	3000
40174	525.35	486.1	6787	3500
44174	620.72	586.05	6788	4000
48674	688.43	660.46	7551	4500
53674	830.58	728.64	6771	5000

Table 5.1.: key field shows the one which ticket_id records are added and record field indicates how many records are added.

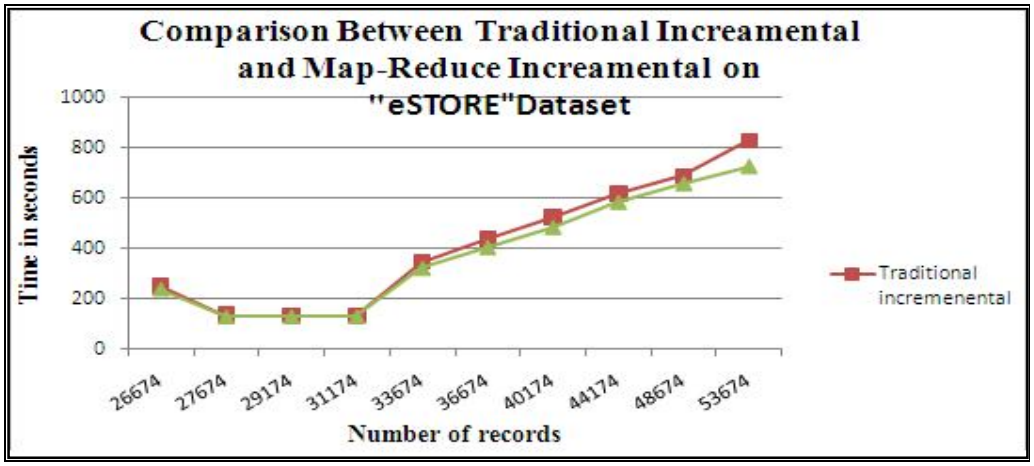

Figure5. 1: Records incrementally added in to the database and as the size of data increases Time required by Map Reduce approach is consistently less than traditional approaches

In the DATAMART" database sale_fact table import the synthetic data for customer field which Aggregate (count) the customer id with the incremental synthetic data.

INC_RECORDS	Traditional Incremental (Time in Sec)	Map Reduce incremental (Time in Sec)	Key	Records
87405	31.06	30.48	1	500
88405	29.15	27.96	5	1000
89950	24.81	23.94	8	1500
91205	28.08	27.8	15	2000
93705	29.22	28.32	21	2500
96705	29.33	28.64	25	3000
100205	30.49	29.41	30	3500
104205	31.12	30.35	35	4000

108705	32.52	31.49	40	4500
113705	36.12	33.26	45	5000
123705	33.95	32.95	55	10000
138705	37.21	34.6	60	15000
158705	42.22	40.5	65	20000

Table 5.2: key field shows the one which records are added and records field explains how many records are added

One can observe 158705 are total number of records in new dataset as a result of merging synthetic records to the original dataset consisting 87405 records.

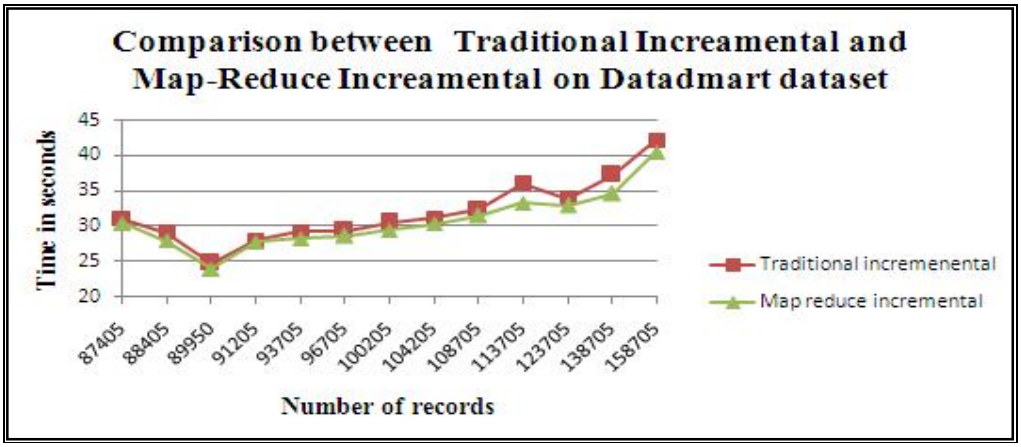

Figure 5.2: Records incrementally added in to the database and as size of database increases Time required by Map Reduce approach is consistently less than traditional approaches

The presented approach tested on dataset having uniform and others with skewed key distribution shows significant improvements with consistent results as compared to the traditional query processing technique.

6. Conclusion

The system under consideration uses an incremental algorithm for efficient execution of join and aggregation queries . The approach is based on MapReduce programming model with the goal to design, develop and implement a framework for fast and efficient retrieval mechanisms. It works within acceptable time frames and keep pace with the information explosion that is essential for efficient and effective database operation.

As database grows, incremental computation is a key for efficient execution. The framework also supports incremental computation and achieves a near optimal time efficiency in resource restricted scenarios. Through extensive experiments over both synthetic and real world data, given solution provide better query evaluation efficiency as compared to traditional system. Ability of hiding the details of parallelization, fault-tolerance, locality optimization, load balancing and accessing specific data for processing are some of the additional merit of the framework.

In particular map reduce approach optimizes the join and aggregation operation on complex database that improves performance and needs minimum resources to detect changes in the incremental data.

Experimental result shows that the time required for computation of join-aggregation using MAP/Reduce approach is lesser when compared with traditional approach.

7. References

1. A.Shatdal, C.Kant, J.F.Naughton. Cache conscious algorithms for relational query processing. VLDB,1994.
2. Michael Isard, Mihai Budiu. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys '07, 2007.
3. Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. OSDI, 2008.
4. Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. Proceedings of the VLDB Endowment, 2008.1(2):1265-1276.
5. Ashish Thusoo Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. VLDB, 2009.
6. Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, D.Stott Parker. Map-reduce-merge: simplified relational data processing on large cluster. SIGMOD'07, 2007.

7. S. Babu and J. Widom. Streamon: an adaptive engine for stream query processing. In SIGMOD Conference, pages 931–932, New York, NY, USA, 2004. ACM.
8. Chen, S., and Schlosser, S. W. Map-reduce meets wider varieties of applications. Tech. Rep. IRP-TR-08-05, Intel Labs Pittsburgh Tech Report, May 2008.
9. Domeniconi, C., and Gunopulos, D. Incremental support vector machine construction. In ICDM'01 (Washington, DC, USA,2001), pp. 589{592.
10. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, 2008