Fault Tolerant Distributed Join Algorithm in RDBMS

Arsen Nasibullin

Saint-Petersburg State University
nevskyarseny@yandex.ru

Abstract. Many of applications use vast volume of data for computing in business intelligence applications. Mostly, these applications handle queries with such operators as aggregation and join. State-of-the-art distributed RDBMS get over these tasks in assumption no errors occur. Unfortunately, distributed database management systems suffer from failures. Failures causes queries with joining large tables re-execute so that enormous volume of resources must be leveraged.
In this paper we propose a new fault tolerant join algorithm for distributed RDBMS. The results which have been already obtained and a detailed plan of further research are discussed.

Keywords: Databases · Join · Query processing · Fault tolerance · Replication.

1 Introduction

Nowadays, known RDBMS work with assumption that no any kind of failures may occur. If a database fails, query should be re-executed. In this work, we assume that a client runs query with join over enormous value of data of two tables dispersed among many servers.

Distributed systems based on Map-Reduce were invented to assist handling vast volume of data on unstable distributed systems. Such kind of systems do not interrupt the execution of query. Instead, they re-execute a part of failed sub-tasks. Unfortunately, Map-Reduce systems do not do it in the best way [1].

The goal of this research is to come up with and implement a fault tolerant distributed join algorithm for unstable RDBMS. Existent RDBMS solutions do not fit to be used because of queries have to be re-executed in case of failure occurrence. Map-Reduce solutions are capable of recovering failed tasks but do not do it effectively. The main task of our work is to seek an intermediate solution.

This paper is organized as follows. Section 2 defines the key terms and notations used in this work. Problem statement and research questions are defined in Section 3. Section 4 provides a review of state of the art related work. Research process, results and further plans are described in Sections 5 and 6. This paper is concluded by Section 7.

2 The Key Terms and Notations Used

The following definitions and notations are used in this paper.
Consider the definition of distributed database systems. *Distributed database systems* are database management systems, consisting of local database systems. Each of these local databases has its disks. Databases are located and dispersed over a network of interconnected computers. In this paper, the configuration of system is based on shared-nothing architecture.
There is the single entry point named *coordinator*. It receives client queries and returns an outcome of an executed query. *Keepers* are nodes where data is stored. *Workers* are nodes where join operation is performed. $|W|$ stands for amount of workers in the configuration of a system. R,S are relations to be joined.
In this paper under *classical algorithm* will be often assumed *classical, unstable, distributed join algorithm.*

3 Problem Statement

There are a few causes the classical distributed join may interrupt [2, 3].

- The coordinator became unreachable because of a communication or a system failure.
- A media or system failure occurred at a keeper or a worker site.
- A site was suddenly turned off during the performing of query.

In this work, the main focus is on coming up with an algorithm which could detect and properly handle causes listed above. The following algorithms of parallel distributed join [4–7] for different sort of systems are used in assumption that a system is fail-free. Examined works do not consider task of handling failures from the list above. In contrast to fail-free RDBMS algorithm, there are many research efforts [8] are dedicated to Hadoop for detecting and proper handling of failures.

Based on the said above, the following research questions are defined in this paper:

- Will doubling tasks increase execution time of query with join?
- What patterns and mechanisms exist for identifying and monitoring the availability of a site?
- How effectively do existent fault tolerant algorithms of Hadoop do their work?
- How data replication can be used in order to design and implement fault tolerant join algorithm?

4 State of the Art

Two main parts of this work to be considered - join algorithms in Map-Reduce and RDBMS, and mechanisms ensuring fault-tolerance.

4.1 Join algorithms

The competitive analysis and description of join algorithms of Map-Reduce are presented in works [9, 10].
Repartition join is a simple algorithm which performs data pre-processing in Map phase, and direct join is done during the Reduce phase. The algorithm has several drawbacks: the algorithm is more time consuming and it requires a lot of memory during the reduce phase. *Repartition join* is widely used in [11].
Broadcast join does the following. It populates the smaller input and proceeds joining during map phase. The disadvantage of this algorithm is that if a smaller input does not fit into memory to build a hash-table, an additional joining phase must be performed [10]. This algorithm is used in Hadoop Pig [12].
Semi-join algorithm is used to prevent transferring data that does not take part in join phase. Approach of deleting unused tuples reduces amount of data to be submitted and joined. The disadvantage of this algorithm is that an extra phase is required to perform joining. Moreover, additional scanning is needed to drop out unwanted data.

4.2 Fault-tolerance mechanisms

In work [13] authors proposed a strategy of doubling each task during the query execution. This stands for if one of the tasks fails, the second backup task will end up on time. It reduces the job completion time by using larger amounts of resources. Tasks are doubled at *map* and *reduce* phases. Readers may guess that doubling the tasks leads to approximately doubling the resources.

Haopeng Chen and Hao Zhu proposed two strategies to improve the failure detection in Hadoop via heartbeat messages in the worker side [14]. The first strategy is an adaptive interval which dynamically configures the expiry time adapted to the various sizes of jobs. The second strategy is to evaluate the reputation of each worker according the reports of the failed fetch-errors from each worker. If a worker failures, it lows its reputation. Once the reputation becomes equal to some bound, the master node marks this worker as failed.
Another taking research [15] proposes a solution based on consensus algorithm Raft. The key point of a system is that each node periodically transfers messages with metadata to other sites. During the execution of a client query, a quorum must take place to handle a client query fully. Raft algorithm is successfully applied in well-known distributed system CockroachDB [16].

To remove single point of failure in Hadoop, a new approach of a metadata replication was proposed in [17]. The solution involves three major phases. In initialization phase, each secondary node is registered to primary node and its initial metadata is caught up with active/primary node. At replication phase, such metadata as outstanding operations and lease states are replicated across all sites. During the fail-over phase, standby/new elected primary node takes over all communications.

To defend stored data from being crashed or lost, mechanism of full data replication has to used. Initially, data can be horizontally partitioned. As example, PostgreSQL [18] provides model of streaming replication. There are two

roles defined in replication mechanism. The first role is *master*. The master server receives client queries, gathers data from others servers and populates *WAL* entries across involved servers. The second role is *standby*. It receives replicated data and stores them in its own disks.

5 Evaluation Plan and Preliminary Results

Given the problem and research questions, the following plan has been performed:

1. Conducted a survey of academic works made in this field. Reviewed abilities of state of the art RDBMS and NoSQL solutions. We checked out how these solutions handle fault occurrences.
2. Reviewed distributed hash-join algorithms. Outlined a cost model and then evaluated the distributed algorithm by applying the cost model to reviewed algorithms. Highlighted possible emerging faults during the execution of join algorithms.
3. Come up with the fault tolerant join algorithm. Applied cost model and conducted a comparison of our algorithm with an unstable distributed join algorithm.

5.1 Fault Tolerant Distributed Join Algorithm

As the basement, classical distributed hash-join algorithm has been taken from work [19]. The fault tolerant distributed hash-join algorithm is similar to classical hash-join for distributed database systems in a shared-nothing architecture.

1. *Building*. A coordinator receives a client query. To initiate a build phase, it populates messages with a client query across all nodes. Once messages are sent, the coordinator sets the status of performing a client query as *processing* for all keepers.
2. Each keeper reads its partitions of relation R, applies a hash function *h1* to the join attribute of each attribute. Hash function *h1* has its range of values $0...|W|-1$. If a tuple hashes to value i, then it goes to $i \mod |W|$ and $(i+1) \mod |W|$ workers. For the latter, a message has to contain message *reserved data*. Once a keeper ends up reading its partitions of relation R, it notifies the coordinator about the status of work.
3. Each worker builds a hash table, allocated in memory, and fills in it with tuples received from step 2. In this step, each worker uses a different hash function *h2* than the one used in step 2.
4. Once all keepers stopped reading their partitions of relation R, the coordinator initiates a probing phase by sending notifications to keepers.
5. *Probing*. Each keeper reads its partitions of relation S, applies a hash function *h1* to the join attribute of each attribute as it does in step 2. If a tuple hashes to value i, then it goes to $i \mod |W|$ and $(i+1) \mod |W|$ workers.

6. Worker $i \mod |W|$ receives a tuple of relation S, probes the hash table built in step 2. If so, tuples join and an outcome tuple is generated. The other worker $(i+1) \mod |W|$ puts reserved data into its disk.
7. Once an outcome tuple is generated, a worker sends a heartbeat message to the following worker. In this message, it points a position of the last successfully joined tuple of relation S.

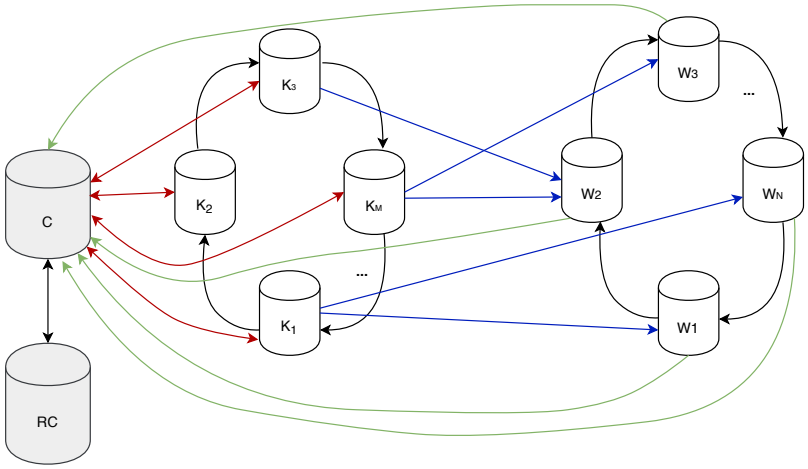

Fig. 1. Scheme of working of fault tolerant distributed join

In the Figure 1 shown a scheme of working of fault tolerant distributed join algorithm. There is added a reserved coordinator *RC*. It synchronizes with the primary coordinator *C*. Workers and keepers comprise a ring of nodes. Each site is aware of the following node. It facilitates a site submits info about proceeded work during the join to the following node. In case of $i \mod |W|$ worker is failed, $(i+1) \mod |W|$ worker takes over tasks of a failed worker. If keeper $i \mod |K|$ fails, site $i \mod |K+1|$ takes over jobs of the failed keeper.

5.2 Comparison and Evaluation

In multi-objective query optimization distributed database systems process finding Pareto set of solutions or the best possible trade-offs among the objective functions [20]. Objective functions might be total time of query execution, I/O operations, CPU instructions and a number of messages to be transmitted. In this work we found trade-off between the least time of the execution in case of failure occurrence and extra resources needed to recover failed tasks. In distributed database systems the total time of query execution is expressed through

mathematical model of weighted average. This model consists of sum of time to perform I/O operations, CPU instructions and time to exchange a number of messages among involved sites. Our work consider evaluating cost of total time of the query execution.

Figures 2, 3 depict time of the execution both algorithms in different cases. The first case is fail-free. Other cases simulate a keeper failed situation, a worker failed and case with failed both keeper and worker. In fail-free case, classical algorithm has benefit in front of fault tolerant algorithm. As for the rest cases, on the average 9% fault tolerant algorithm takes less time to perform a client query even if at least one of site is down.

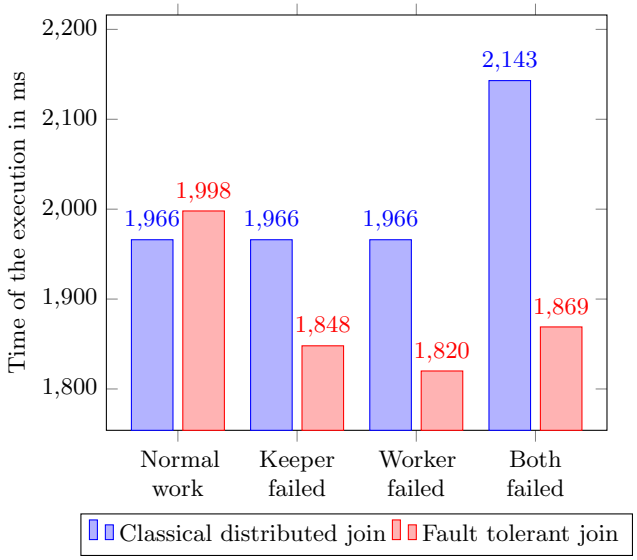

Fig. 2. Comparison of time execution of both algorithms for four cases. T(R) = T(S) = 256

6 Future work

Based on obtained results, our further plans look like the following:

- Design and implement distributed fault tolerant hash-join algorithm. Conduct experiments with other solutions.
- Perform comparison of the performance of our extension with Hadoop. Make use of different volumes of data.
- Evaluate and compare I/O, CPU, and memory costs.
- Consider combining the developed fault-tolerant algorithm with other join algorithms.

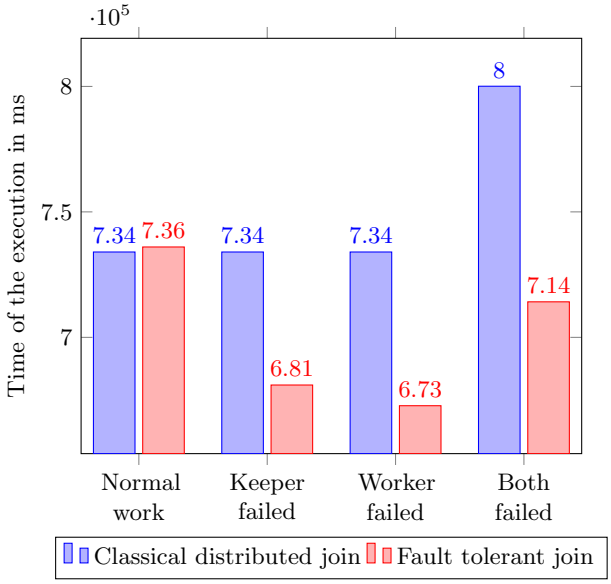

Fig. 3. Comparison of time execution of both algorithms for four cases. T(R) = T(S) = 100.000

– Define benchmarks to evaluate and compare developed fault tolerant algorithms with existent solutions.

As example, developed algorithms might be compared with Hadoop Map-Reduce Join algorithms. Evaluation should be performed with different volume of data.

7 Summary

In this paper the fault tolerant distributed join algorithm has been proposed. Results of comparison demonstrates that proposed algorithm lead to less time to re-execute a failed task at a failed site than time needed to re-execute the query using classical algorithm. Also future work is provided.

Acknowlegements. Author thanks Boris Novikov for his helpful comments that have significantly improved this paper.

References

1. Christos Doulkeridis and Kjetil Norvaag. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, June 2014.

2. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.
3. Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the system r database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.
4. Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013.
5. Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, January 2017.
6. Georges Gardarin and Patrick Valduriez. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9, 03 1984.
7. J. Teubner and G. Alonso. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1754–1766, July 2015.
8. Bunjamin Memishi, Shadi Ibrahim, María Pérez, and Gabriel Antoniu. *Fault Tolerance in MapReduce: A Survey*, pages 205–240. 10 2016.
9. Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 975–986, New York, NY, USA, 2010. Association for Computing Machinery.
10. A. Pigul. Comparative study parallel join algorithms for mapreduce environment. *Proceedings of the Institute for System Programming of RAS*, 23:285–306, 01 2012.
11. *APACHE HIVE*, 2020.
12. *APACHE PIG*, 2020.
13. Pedro Costa, Marcelo Pasin, Alysson Bessani, and Miguel Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. pages 32–39, 11 2011.
14. Hao Zhu and Haopeng Chen. Adaptive failure detection via heartbeat under hadoop. pages 231–238, 12 2011.
15. Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.
16. *CockroachDB official website*, 2020.
17. Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. Hadoop high availability through metadata replication. In *Proceedings of the First International Workshop on Cloud Data Management*, CloudDB '09, pages 37–44, New York, NY, USA, 2009. ACM.
18. *PostgreSQL official website*, 2020.
19. David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.
20. Vikram Singh. Multi-objective parametric query optimization for distributed database systems. In Millie Pant, Kusum Deep, Jagdish Chand Bansal, Atulya Nagar, and Kedar Nath Das, editors, *Proceedings of Fifth International Conference on Soft Computing for Problem Solving*, pages 219–233, Singapore, 2016. Springer Singapore.