Computer tools in education, 2022
№ 4: 68–82
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-4-68-82

FAULT TOLERANT HASH JOIN FOR DISTRIBUTED SYSTEMS

Nasibullin A.[1], PhD candidate, ✉ nevskyarseny@yandex.ru

[1]Saint Petersburg State University, 28 Universitetskiy pr., Stary Peterhof, 198504, Saint Petersburg, Russia

Abstract

Nowadays, enterprises are inclined to deploy data processing and analytical applications from well-equipped mainframes with highly available hardware components to commodity computers. Commodity machines are less reliable than expensive mainframes. Applications deployed on commodity clusters have to deal with failures that occur frequently. Mostly, these applications perform complex client queries with aggregation and join operations. The longer a query executes, the more it suffers from failures. It causes the entire work has to be re-executed.

This paper presents a fault tolerant hash join (FTHJ) algorithm for distributed systems implemented in Apache Ignite. The FTHJ achieves fault tolerance by using a data replication mechanism, materializing intermediate computations. To evaluate FTHJ, we implemented the baseline, unreliable hash join algorithm. Experimental results show that FTHJ takes at least 30% less time to recover and complete join operation when a failure occurs during the execution. This paper describes how we reached a compromise between executing recovery tasks for the least amount of time and using additional resources.

Keywords: *Distributed systems, Hash join, Fault tolerance, Replication.*

Citation: A. Nasibullin, “Fault Tolerant Hash Join for Distributed Systems,” *Computer tools in education*, no. 3, pp. 68–82, 2022; doi: 10.32603/2071-2340-2022-3-68-82

1. INTRODUCTION

From year to year, companies rely on the results of massive data analytics for critical business decisions. For making decisions, businesses utilize distributed processing systems. In this paper, we consider the shared-nothing architecture of distributed computing systems.

Distributed processing systems are used for a variety of tasks such as processing data from patients to track the spread of chronic diseases, marketing analytics, and gathering a massive volume of data out of Internet of Things sensors for transportation applications [1, 2]. Most of these tasks assume performing complex queries using such operators as aggregation and join.

Distributed systems can provide improved performance, high availability, and fault tolerance. In our work, we focus on *fault tolerance* defined as a property of a system to keep on performing tasks in the event of the failure [1, 3]. A system can detect a failure and properly handle it without affecting the correct execution of tasks.

The join operation is one of the most time-consuming operations. It has been studied and discussed widely in works [2, 4–6]. The join algorithms implemented in parallel relational DBMSs deliver excellent performance and are scalable, but do not, in case of any failure, an application has to re-submit the whole transaction.

Modern scalable distributed systems such as [7–10] have built-in features of detecting and handling faults without affecting the work they do. However, the granularity of units of work to be repeated is too coarse, and the performance of join algorithms is significantly lower than in relational DBMSs.

The modern distributed database systems re-execute an entire query in case of a node of a system fails. For example, during the execution of the join operation, a node with stored data becomes unavailable when transmitting data to sites where the join operation has to be executed. The re-execution of a query with a join results in spending significant resources and time to end up with a join query.

In this work, we consider distributed fault tolerant algorithm for the join operation [11].

A generic fault tolerant hash join algorithm was introduced in [11]. In this paper, we extend and specialize this algorithm for the scalable, distributed main memory system and experimentally evaluate potential performance gains. Our solution achieves the fault tolerant execution queries with join operation through making a trade-off between completing recovery tasks for the last time and extra resources to be used for this.

The rest of the paper is organized as follows. The basic concepts are provided in section 2. The description of our algorithms, presented in section 3, is followed by experimental results in section 4. Section 5 describes related works. Finally section 6 concludes the paper.

2. BACKGROUND

2.1. Hash Join Algorithms

The join operation is one of the most time-consuming operations. In this paper, we focus on the hash join algorithm. Hereafter, consider the join of two datasets, L and R.

A simple hash join algorithm consists of two phases:

Build: In this step, rows of L disperse across hash buckets depending on hash values computed from the join attributes as a hash key.

Probe: During the probe phase, the algorithm scans R sequentially and forms an output row for each row of L matching the current row of R.

In the era of Big Data, combining two or more large datasets requires effective algorithms. For this challenge, researchers consider parallel and distributed [12] join algorithms, and MapReduce-based [13] join algorithms.

MapReduce and Distributed Hash Joins. The common idea of MapReduce join algorithms [13, 14] is to split two datasets into small chunks, distribute them across nodes, and join a subset of received chunks on each machine in parallel. In this way, the *Repartition join* algorithm works. Another algorithm, named *Broadcast join*, spreads the smaller dataset L across cluster nodes and joins each partition of R with the passed dataset. A detailed overview of other MapReduce joins algorithms can be found in work [15].

In contrast to MapReduce-based hash joins, join algorithms in distributed systems look as follows. In the first phase, called *distribution*, all nodes repartition data by a joining key. Once a node obtains a subset of data, a local join runs build and probe phases sequentially. Distributed join algorithms may rely on the fact that data is already repartitioned by some key of distribution [16]. It helps avoid the phase of distribution. This approach works if a distribution key matches the key of the join operation.

2.2. Fault Tolerance Techniques

In accordance with studies [17, 18], failures are divided into the following:

Communication failure: This type of failure appears in distributed systems. There are many kinds of communication failures: lost messages or connections, unordered messages, and undelivered messages.

Media failure: It refers to the failure of the secondary storage devices that allocate the database. Such type of failure may be due to operating system errors.

System failure: A hardware of a software failure may cause a system failure. Once a system failure occurs, the content of the main memory is lost.

In works [17, 18], authors assert that hardware and software failures comprise more than 90 % of failures. This percentage has not been changed substantially since the early days of computing.

Fault-tolerant support consists of two main key concepts: fault detection and recovery [19, 20].

Fault detection enables detecting faults as soon as they occur. The heartbeat approach relies on a periodic exchange of heartbeat messages between two system components during a fail-free period. The approach is widely used in such systems as Apache Hadoop [8], Apache Ignite [9], Apache Spark [10], Google Spanner [21].

Fault recovery brings the faulty component to its normal state after a fault occurs. To ensure data availability and reliability, it applies checkpointing and replication techniques.

Data replication ensures reallocating a client request to an available site with stored data. Checkpointing preserves the state of an active node to store it on another standby node. In case of the node becomes inactive, the standby node takes over on the latest recorded state of the active node for fast fault recovery.

Such distributed databases and processing systems as [8, 9, 21–24] are based on those concepts. Particularly, works [25, 26] deeply examine these mechanisms.

2.3. The Base System

Apache Ignite is a distributed database and processing system for high-performance computing with in-memory speed. It targets analytical, streaming, and transactional workloads. Apache Ignite runs on the shared-nothing architecture. The system can add or remove nodes non-disruptively. In contrast to systems, based on the primary-secondary architecture [10, 27], no assigned node acts in the role of the primary. Instead, all nodes of a cluster are identical.

Apache Ignite provides an API that allows developers to distribute computations across multiple nodes to gain high performance. These APIs include:

Distributed computing API: Ignite provides an ability to launch either individual tasks or a task based on the MapReduce paradigm.

Load balancing: Ignite automatically load balances jobs produced not only by a compute task but also by individual tasks submitted to the cluster.

Asynchronous communication: Communication among cluster nodes might be asynchronous to increase the availability of the nodes.

Data in the Apache Ignite cluster represents a set of key-value pairs that constitute *cache*. The cache's key may represent a set of attributes of some database table. By default, Ignite takes the primary key of a database table as a cache key. The value consists of a set of rest database table attributes. An application can create cache dynamically and distribute it across cluster nodes by partitions. Ignite disperses data across partitions but not nodes. The core of Ignite takes care of the even distribution of partitions across cluster nodes.

Distributed computing API comes with failover capabilities that may handle a failed job. By default, the system automatically transfers computations of a failed cluster node to other available nodes for re-execution. Additionally, Failover API allows using other strategies to handle failed jobs.

Every task has its session object that stores some attributes and a list of jobs performed. The task session can be replicated so that the session might be restored in case of a node failure. Each job has its context that preserves data or any other attributes that may help in computations. A job context may be restored within the cluster nodes.

3. FAULT TOLERANT HASH JOIN

This section introduces the fault tolerant hash join (FTHJ) algorithm. First, we describe the baseline algorithm DHJ and then provide a detailed description of FTHJ.

3.1. The baseline algorithm

As the baseline, we implemented distributed hash join (DHJ) algorithm on top of Apache Ignite distributed computing API. The implemented algorithm uses the default data distribution that Apache Ignite provides.

At the top level, we split distribution, build, and probe phases of the distributed hash join between two stages — map and reduce. The map stage distributes data across machines. Whereafter, the reduce stage performs the build, probe phases, and returns outputs. In overall, DHJ looks as follows:

1. A client node initiates a task by sending a request to Ignite cluster.
2. One of the cluster nodes receives the client request and takes the responsibility to act the role of the master node for performing the client request.
3. The master node initiates the map stage. It creates a redistribution task. This task redistributes data of both caches across all nodes in the cluster by the join key.
4. After the redistribution task is done, the master node initiates the reduce stage. It creates a reduce task. This task consists of 2 sequential jobs — *perform local hash join* and *send the output*.
5. Each worker executes the local hash join of stored partitions of both caches. At the build phase, a worker creates a hash table where a join key is the key of a bucket.
6. During the probe phase, a worker looks for all matches by the join condition.
7. Once a worker performs hash join algorithm, it sends the output to the master node.
8. The master node aggregates results obtained from workers and returns them to the client node.

3.2. Fault tolerant algorithm

FTHJ uses the baseline algorithm with the following enhancements. Before running a task, Ignite cluster creates a new cluster group. Let us name this group as *Backup Data Storage* (BDS).

This cluster group allocates a reserved cache where all workers put intermediate results they computed. Any worker or the master node may reach out to that group. FTHJ uses out-of-the-box replication capability. It guarantees that a partition has one or more replicas within the cluster.

Summarizing all above, FTHJ looks as follows:

1. A client node initiates a task by sending a request to Ignite cluster.
2. One of the cluster nodes receives the client request and takes the responsibility to act the role of the master node for performing the client request.
3. The master node initiates the map stage. It creates a redistribution task. This task redistributes data of both caches across all nodes in the cluster by the join key.
4. As soon as data is redistributed, the master node persists metadata that keeps the status of the map stage. This step allows going by the redistribution phase during the recovery.
5. After the redistribution task is done, the master node initiates the reduce stage. It creates a reduce task. This task consists of 2 sequential jobs — *perform local hash join* and *send the output*.
6. In parallel, the Ignite cluster creates a BDS cluster group.
7. After the map phase is done, the master node broadcasts jobs across workers.
8. Each worker executes the local hash join of stored partitions of both caches. At the build phase, a worker creates a hash table where a join key is the key of a bucket.
9. Once a partition of R is handled at the probe phase, a worker asynchronously sends computed data to BDS. After that, a worker memorizes the number of the last handled partition in the job context.
10. Once a worker performs hash join algorithm, it sends the output to the master node.
11. The master node aggregates results obtained from workers and returns them to the client node.

Figure 1 illustrates the schema of FTHJ. Peach rectangles represents the BDS cluster group. Node W_1 with the crown is assigned to act the role of the master node per the given task.

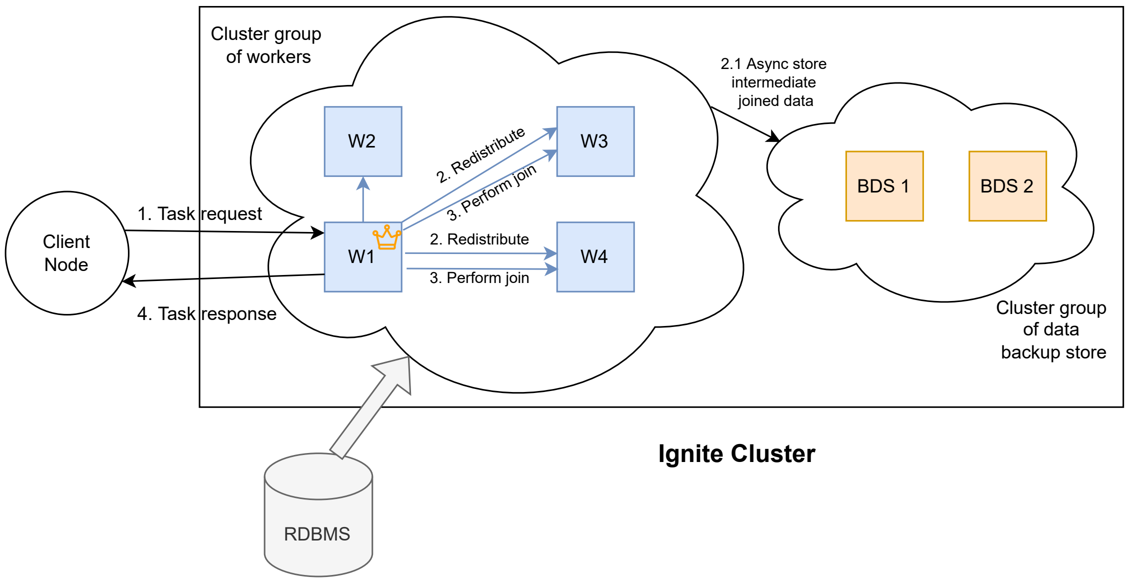


Figure 1. Scheme of fault tolerant hash join

The pseudocode of the FTHJ 1 is presented below:

Algorithm 1 Fault Tolerant Hash Join Algorithm

```
procedure AssignMasterNodeForTask(client task)
return A random node to be the master node
end procedure

procedure RunMapStage
Fan out the client task across all workers
Redistribute both caches across all workers by the join key
    for all workers in the cluster do
        Redistribute both local caches partitions across other workers
        Notify the worker distributed its local data
    end for
end procedure

procedure RunReduceStage
Fan out the client task across all workers
Redistribute both caches across all workers by the join key
    for all workers in the cluster do
        Perform local hash join
        Asynchronously send output to BDS
        Memorize the last successfully handled partition
        Output the obtained result to the master node
    end for
end procedure

procedure Main(client task)
AssignMasterNodeForTask(client task)
Persists metadata of the Map stage
RunMapStage
Result = RunReduceStage
Send aggregated result to the client
end procedure
```

3.3. Worker Failure Handling

A worker may fail at any of the possible cases:

— before receiving a job from the master node,
— during the execution of a job.

To handle the first case, the master node becomes aware of the failure of a worker via received heartbeat messages. At the map phase, the master node simply omits the absence of a failed worker and distributes jobs across the rest of the cluster nodes. As for the failure of a worker during the execution of a job, the master receives join context metadata along with the last handled partition at the result phase. Hereafter, the master does the following:

- *Loading computed data from BDS*. The master node asynchronously request all computed data by the failed node.
- *Choose a node where the failed job will be resumed*. By default, a random node is chosen. With the help of *load balancing* API, the cluster may switch to another strategy to pick up a new node for work.

- *Resume the failed job*. A worker, assigned to handle the failed job, resumes the execution. It takes the last processed partition number p_i, handled by the failed worker, and resumes the execution from p_{i+1} partition.

Figure 2 depicts a case of a worker's failure.

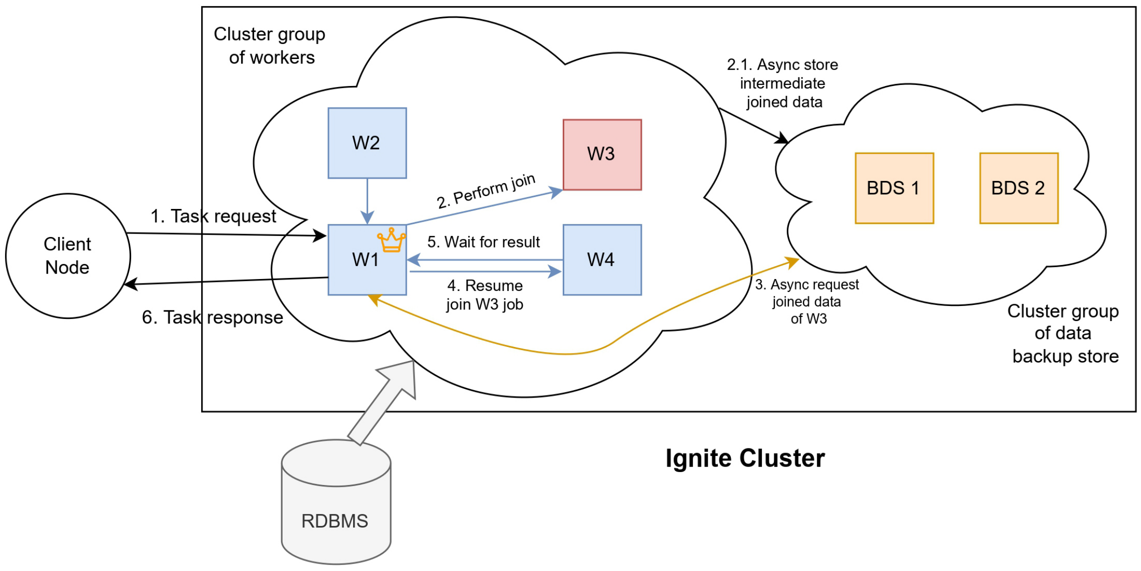


Figure 2. Scheme of handling a worker's failure

The pseudocode 2 of the algorithm is shown below:

Algorithm 2 Worker Failure Handling Algorithm

```
1: procedure ResumeJob(JobId)
2:     Load job metadata by id
3:     start the job for $p_{i+1}$ partition
4: end procedure

5: procedure HandleWorkerFailure(WorkerId)
6:     Asynchronously load computed data of the failed worker
7:     ResumeJob(jobId)
8: end procedure
```

3.4. Master Failure Handling

As we mentioned in section 2, Ignite cluster does not have a special master role. Once a client node sends a request to cluster nodes, an arbitrary node is assigned to be in the role of the master of the client's task. During the task execution, the master node may fail at any phase. For each case, FTHJ does the following:

- *Map phase*. Before sending jobs to workers, the master node may fail. It causes jobs will not execute on nodes.
- *Job execution phase*. During job executions, the connection between master and workers may be abrupt.
- *Reduce phase*. While collecting data from workers, the master node may fail down. It causes all computed data will be lost.

To get over failures above, *FTHJ* does the following. In *job execution*, each worker suspends its work preserves computed data of the last processed number of partition p_i in BDS. Then, the client node rejoins to a new master node. In turn, the new master recovers the failed task, asynchronously requests all computed data from BDS, and sends out jobs to workers. Each worker resumes their job starting from p_{i+1} partition of R.

As for the rest two phases, FTHJ handles master failure in the following manner. A client request goes to a new master and that master simply checks the status of the task from metadata. If the task status is *REDUCE*, the master node requests computed data from BDS. Considering the failure at the *Map* phase, a task completely re-executes, because no computations were not started yet. Figure 3 illustrates a scheme of handling a failure of node W_1 that used to act the role of the master node for a task. The client node resubmits the task to W_2 node with the crown. This node is chosen as a new master node for the task.

The pseudocode 3 of the algorithm is depicted below:

Algorithm 3 Master Failure Handling Algorithm

```
 1: procedure HadleFailureAtTheJobExecutionPhase
 2:     for all workers in the cluster do
 3:         Suspend all computations
 4:         Wait for the request from a new master node
 5:         Resume computations from $p_{i+1}$ partition
 6:     end for
 7: end procedure

 8: procedure HandleMasterFailure
 9:     HadleFailureAtTheJobExecutionPhase
10:     if $thefailurehappenedatthejobexecutionphase$ then
11:         HandleFailureAtTheJobExecutionPhase
12:     else if $thefailureoccurredatthemapphase$ then
13:         Completely re-execute the client task
14:     else
15:         Request data from BDS async
16:     end if
17: end procedure
```

4. EXPERIMENTAL EVALUATION

This section examines the environment setup, the dataset, and queries we implemented as distributed tasks. Then, we describe the performed experiments and compare the results of the proposed algorithm to the baseline distributed hash join algorithm.

4.1. Experimental Setup

We used Apache Ignite and Java to implement algorithms.

Regarding a workstation for running experiments, we used laptop MacBook Pro 16 inches, 16 GB RAM, 512 GB SSD. Locally, the cluster consisted of 4 virtual nodes. Each of the nodes acquired 3GB of RAM. Additionally, we deployed one more node with 1 GB RAM to test FTHJ. The used operating system was Mac OS Monterey version 12.0.1, and the software stack consisted of Apache Ignite version 2.12.0, AWS RDS PostgreSQL version 13.4, and Java version 15.

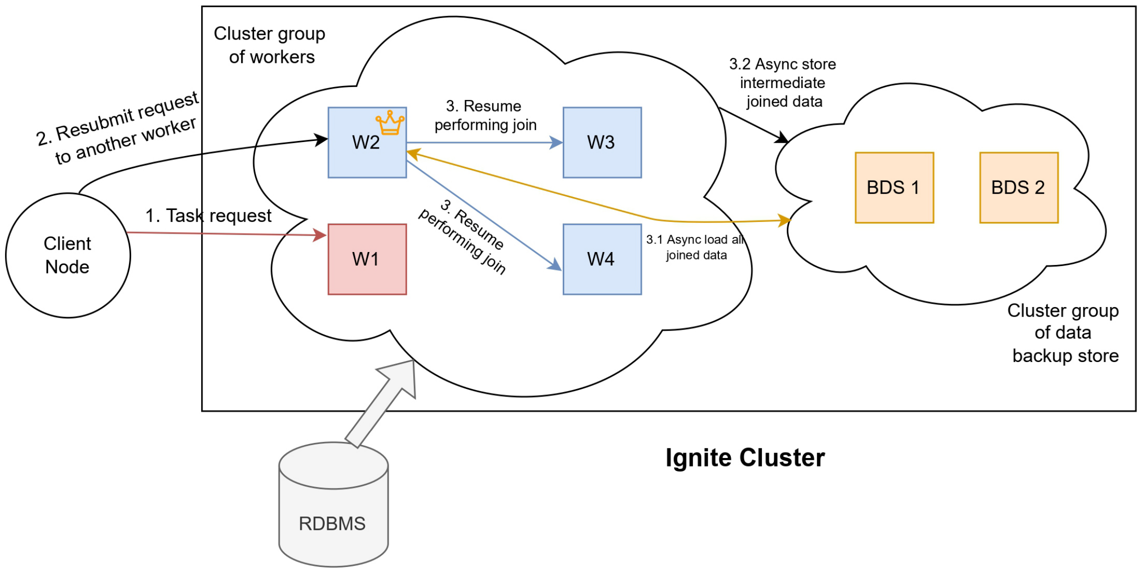

Cluster group of workers
2. Resubmit request to another worker
W2
3. Resume performing join
W3
3.2 Async store intermediate joined data
BDS 1
BDS 2
Client Node
1. Task request
W1
3. Resume performing join
W4
3.1 Async load all joined data
Cluster group of data backup store
Ignite Cluster
RDBMS

Figure 3. Scheme of handling a master's failure

In our experiments, we took advantage of TPC-H Benchmark [28]. Out of 8 tables of the benchmark, we took up three ones: Part, PartSupp, and LineItem. We set the replication to 2 partitions for all partitions in both cluster groups. Based on the laptop has hardware limitations, we considered the scale factor of generated data equal to 1. The bigger scale factor caused the laptop to be stuck.

For experiments, we leveraged two SQL queries. The first query looks as follows:

```
SELECT *
FROM part AS p
JOIN partsupp AS ps
ON p.p_partkey = ps.ps_partkey
```

Listing 1. Join of two tables Part and Partsupp

TPC-H Benchmark comes with 22 queries. Not all queries use join operation. For experiments, we selected Query 14 which is defined below:

```
SELECT
100.00 * SUM(CASE
WHEN p_type LIKE ’PROMO%’
THEN l_extendedprice * (1 - l_discount)
ELSE 0
END) / SUM(l_extendedprice * (1 - l_discount))
AS promo_revenue
FROM ineitem, part
WHERE l_partkey = p_partkey
AND l_shipdate >= DATE ’1995-09-01’
AND l_shipdate < DATE ’1995-09-01’
+ INTERVAL ’1’ MONTH;
```

Listing 2. Query 14 of TPC-H Benchmark

We implemented both queries as distributed tasks, written in Java code. For reference in this paper, let us name SQL query 1 as Task 1, and query 2 marked as Task 2.

4.2. Results and Discussion

We compared both implemented algorithms DHJ and FTHJ in the following scenarios:

- No failures happen.
- A failure of a worker in both phases.
- A failure of the master node in the join phase.
- Multiple failures. The master fails in the redistribution phase, and a worker fails in the join phase.

To perform the last two scenarios, we decided to get rid of the process of a random node to simulate the failure of a node. We considered the following percentage of the job execution made by a worker — 50 %. As soon as the assigned worker handles the needed percentage of done work, it has to be turned off immediately.

No Failures Happen. In this scenario, we executed two tasks in fail-free mode. Figure 4 illustrates both algorithms. We observe that FTHJ demonstrates a higher execution time for each of the tasks compared to DHJ. We assume it might be due to a) additional resources needed for initialization and b) extra communication with the BDS cluster group.

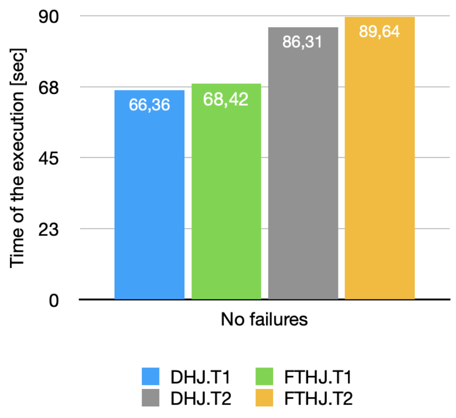
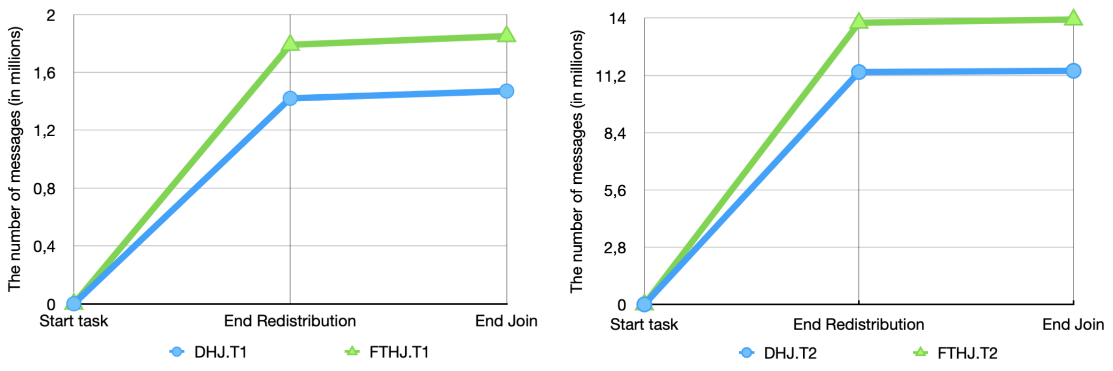


Figure 4. Fail-free scenario

We counted the number of messages that all nodes send out during the fail-free work at the end of each phase. Figure 5 a and 5 b depict results. Compared with DHJ, FTHJ stores intermediate join results and metadata required to resume computations after the failure. All features of FTDJ mentioned above impact the number of messages. Out algorithm sends 17 % and 19% more messages than DHJ does for each task accordingly.

A Worker Failure. As shown in Figures 6 a and 6 b, FTHJ works better for both tasks. In case of a random worker fails on a half of done work in the redistribution phase, FTHJ works 35 %, 29 % faster than DHJ does for each task respectively. This is because another worker takes responsibility to resume the work over a job of a failed node not from the beginning. As for the join phase, FTHJ demonstrates 40 %, 37 % better results for both tasks correspondingly. The master node takes care of loading all computed data from the BDS cluster group.

The Master Node Failure. Figures 7 shows results of the executions of both tasks. For both tasks, FTHJ works 37 % more quickly than DHJ does in case of a random worker fails once 50 % of



(a) Task 1 **(b)** Task 2

Figure 5. Total number of messages that are sent by both algorithms

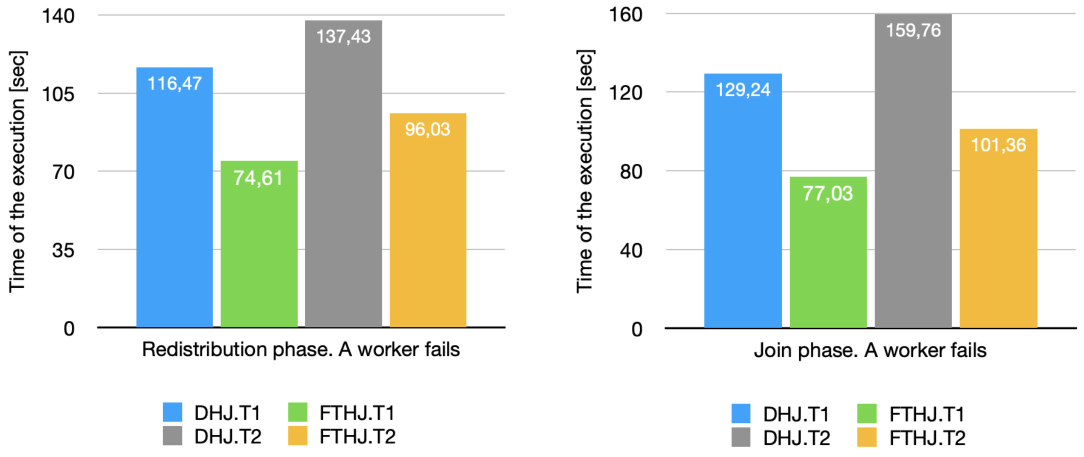


Figure 6. Scenarios when a worker fails in the redistribution and the join phases

its work is done. DHJ works worse because the entire task has to be re-run. In contrast, all FTHJ needs to do is to reassign the client's request to another node, restore task metadata, and resume the work. From the results, we see that the more time task executes, the more time needs DHJ to recover from the failure.

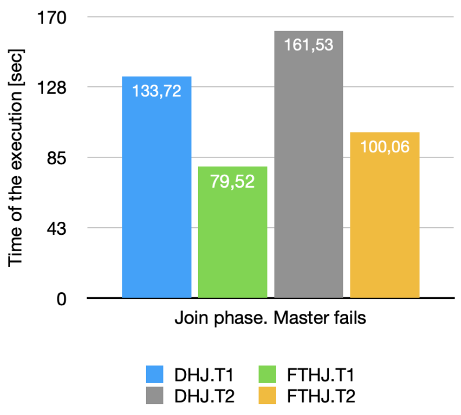


Figure 7. Scenario when the master fails in the join phase

Multiple Failures. In this scenario, we turned off both the master node in the redistribution phase, and a worker in the probe phase. Figure 8 shows results for both tasks. FTHJ performs better than its competitor on 47 % and 41 % for each task respectively.

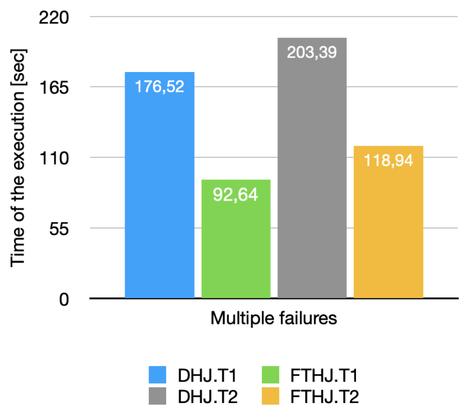


Figure 8. Multiple failures

Whatever scenario and task we considered, a cluster needs no more than 1.04 seconds on average to recover a failed job at any phase. This time includes itself the following: the master node becomes aware of the failure of a node, a job re-submits to another worker, and a new worker notifies a new job is received.

5. RELATED WORK

In this section, we briefly review works dedicated to joining algorithms. Also, we studied works that are related to different approaches to handling failures.

5.1. Join Algorithms

Researchers proposed the bucker-sorted hash join algorithm [29]. The algorithm searches the corresponding record during the probe phase. Authors assume the records are allocated in a sorted order only within each bucket, not across the buckets. It helped researchers to gain up to 300% performance compared to the hybrid hash join.

In [30] authors presented a scalable distributed hash join on GPUs. The algorithm involves a two-level shuffle phase for data redistribution.

5.2. Fault Detection and Recovery

A schema for recovering from mid-query faults in a distributed shared-nothing database was proposed in [31]. Instead of aborting and restarting queries, the authors suggest dividing queries into subqueries and replicating data such that each subquery can be rerun on a different node.

In [32], authors proposed a family of Recovery Algorithms for Fast-Tracking (RAFT) MapReduce in the presence of these failures. The family of algorithms exploits the checkpoint approach to preserve query metadata, task progress computation, and intermediate result.

A new approach of a metadata replication-based solution was proposed in [33]. This was done to remove a single point of failure in the Hadoop cluster. The proposed solution leverages metadata replication and checkpoint mechanism so that all metadata of a failed primary node might be caught by a standby node. Also, the standby node takes over all communications from the rest nodes.

6. CONCLUSION

This paper describes and analyzes an implementation FTHJ algorithm on top of the baseline algorithm DHJ. In contrast to DHJ, FTHJ showed that even if an abrupt failure happens, the entire work does not interrupt. Our solution uses the backup data storage cluster group with Ignite Failover API. The results of the experiments demonstrated the effectiveness of FTHJ in terms of the overall execution time.

This work shows the results of experiments in the laptop we used. This is due to we could not have had availability to run our experiments in a bunch of commodity machines. In future work, we will conduct comprehensive experiments in the cloud cluster with multiple nodes. Further experiments will use an increased scale factor value to generate more test data.

References

1. A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, Upper Saddle River, NJ, USA: Prentice-Hall, 2006.
2. B. Catania and L. Jain, "Advanced Query Processing: An Introduction," in *Intelligent Systems Reference Library*, vol. 36, pp. 1–13, 2012; doi:10.1007/978-3-642-28323-9_1
3. A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004; doi:10.1109/tdsc.2004.2
4. G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, 1993; doi:10.1145/152610.152611
5. C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores," in *Proc. of the VLDB Endowment*, vol. 10, no. 5, pp. 517–528, 2017; doi:10.14778/3055540.3055545
6. C. Kim et al., "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," in *Proc. of the VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009; doi:10.14778/1687553.1687564
7. Volt Inc., "VOLT ACTIVE DATA," in *voltdb.com*, 2022. [Online]. Available: https://voltdb.com/
8. The Apache Software Foundation, "Apache hadoop," in *apache.org*, 2019. [Online]. Available: https://hadoop.apache.org/
9. The Apache Software Foundation, "Apache ignite," *apache.org*, 2022. [Online]. Available: https://ignite.apache.org/
10. The Apache Software Foundation, "Apache spark," *apache.org*, 2020. [Online]. Available: https://spark.apache.org/
11. A. Nasibullin and B. Novikov, "Fault tolerant distributed hash join in relational databases," in *5th Conf. on software engineering and information management (SEIM-2020), Aahen, Germany*, pp. 11–17, no. 2691, 2020.
12. D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," *ACM SIGMOD Record*, vol. 18, no. 2, pp. 110–121, 1989.
13. S. Blanas et al., "A comparison of join algorithms for log processing in mapreduce," in *Proc. of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 975–986, 2010.
14. K. Shim, "Mapreduce algorithms for big data analysis," in *Int. Workshop on Databases in Networked Information Systems*, pp. 44–48, 2013.
15. D. Van Hieu, S. Smanchat, and P. Meesad, "Mapreduce join strategies for key-value storage," in *2014 11th Int. Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 164–169, 2014.
16. Citus Data, Microsoft Company, "Citusdb," in *citusdata.com*, 2022. [Online]. Available: https://citusdata.com/

17. M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, Springer Publishing Company Inc., 2011.
18. J. Gray et al. "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–242, 1981; doi:10.1145/356842.356847
19. M. Saadoon et al., "Fault tolerance in big data storage and processing systems: A review on challenges and solutions," *Ain Shams Engineering Journal*, vol. 13, no. 2, p. 101538, 2022; doi:10.1016/j.asej.2021.06.024
20. N. Ayari, D. Barbaron, L. Lefevre, and P. Primet, "Fault tolerance for highly available internet services: concepts, approaches, and issues," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 2, pp. 34–46, 2008.
21. D. F. Bacon et al., "Spanner: Becoming a sql system," in *Proc. of the 2017 ACM International Conference on Management of Data*, pp. 331–343, 2017.
22. Amazon Web Services Inc., "Aws rds," in *amazon.com*, 2022. [Online]. Available: aws.amazon.com/rds/
23. ClickHouse Inc., "Clickhouse," in *clickhouse.com*, 2022. [Online]. Available: https://clickhouse.com/
24. The Apache Software Foundation, "Apache hive," *apache.org*, 2022. [Online]. Available: https://hive.apache.org/
25. D. Playfair, A. Trehan, B. McLarnon, and D. S. Nikolopoulos, "Big data availability: Selective partial checkpointing for in-memory database queries," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 2785–2794, 2016.
26. T. Chen and K. Taura, "A selective checkpointing mechanism for query plans in a parallel database system," in *2013 IEEE Int. Conf. on Big Data*, pp. 237–245, 2013.
27. MongoDB, Inc., "Mongo db," in *mongodb.com*, 2021. [Online]. Available: https://www.mongodb.com/
28. TPC, "Tpc-h benchmark," in *tpc.org*, 2022. [Online]. Available: http://www.tpc.org/tpch/
29. H. Shin, I. Lee, and G. S. Choi, "Bucket-sorted hash join," *Journal of Information Science & Engineering*, vol. 36, no. 1, pp. 171–190, 2020.
30. H. Gao and N. Sakharnykh, "Scaling joins to a thousand gpus. in Bordawekar," in *Int. Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, pp. 55–64, 2021.
31. C. Yang, C. Yen, C. Tan, and S. Madden, "Osprey: Implementing mapreducestyle fault tolerance in a shared-nothing distributed database," in *Proc. of the 26th Int. Conf. on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pp. 657–668, 2010; doi:10.1109/ICDE.2010.5447913
32. J. A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "Rafting mapreduce: Fast recovery on the raft," in *2011 IEEE 27th Int. Conf. on Data Engineering*, pp. 589–600, 2011.
33. F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop high availability through metadata replication," in *Proc. of the First Int. Workshop on Cloud Data Management, CloudDB '09*, pp. 37–44, 2009; doi:10.1145/1651263.1651271

Received 12-11-2022, the final version — 21-12-2022.

Arsen Nasibullin, PhD candidate, Postgraduate graduate of the Faculty of Mathematics and Mechanics, Saint Petersburg State University, ✉ nevskyarseny@yandex.ru

Компьютерные инструменты в образовании, 2022
№ 4: 68–82
УДК: 004.4
http://cte.eltech.ru
doi:10.32603/2071-2340-2022-4-68-82

Отказоустойчивый алгоритм hash join в распределенных системах

Насибуллин А. Р.[1], соискатель, ✉ nevskyarseny@yandex.ru

[1]Санкт-Петербургский государственный университет, Университетский пр., д. 28,
Старый Петергоф, 198504, Санкт-Петербург, Россия

Аннотация

В настоящее время компании развертывают приложения для обработки данных и анализа не на мейнфреймах с производительными аппаратными компонентами, а на обычных кластерах из персональных компьютеров. Персональные компьютеры менее надежны, нежели дорогие мейнфреймы. Приложениям, развернутым в кластерах, приходится иметь дело с частыми сбоями. В основном эти приложения выполняют сложные клиентские запросы с операциями агрегирования и объединения. Чем дольше выполняется запрос, тем больше он подвержен сбоям системы. Это приводит к тому, что вся работа должна быть выполнена заново. В этой статье представлен алгоритм отказоустойчивого hash join (FTHJ) для распределенных систем, реализованный в Apache Ignite. FTHJ обеспечивает отказоустойчивость за счет использования механизма репликации данных, реализующего промежуточные вычисления. Для оценки FTHJ мы внедрили подверженный к отказам алгоритм hash join. Экспериментальные результаты показывают, что FTHJ требуется как минимум на 30 % меньше времени для восстановления и завершения операции соединения в случае, если сбой произошел во время работы алгоритма. В этой работе описывается, как мы достигли компромисса между выполнением задач восстановления за наименьшее количество времени и использованием дополнительных ресурсов.

Ключевые слова: *распределенные системы, hash join, отказоустойчивость, репликация.*

Цитирование: Насибуллин А. Р. Отказоустойчивый алгоритм hash join в распределенных системах // Компьютерные инструменты в образовании. 2022. № 4. С. 68–82. doi:10.32603/2071-2340-2022-4-68-82

Поступила в редакцию 12.11.2022, окончательный вариант — 21.12.2022.

Насибуллин Арсен Радикович, соискатель, выпускник аспирантуры математико-механического факультета СПбГУ, ✉ nevskyarseny@yandex.ru