Fault Tolerant Distributed Hash-Join in Relational Databases

1st Arsen Nasibullin
Saint-Petersburg State University
Saint-Petersburg, Russia
nevskyarseny@yandex.ru

2nd Boris Novikov
HSE University
Saint-Petersburg, Russia
bnovikov@hseu.ru

***Abstract*—The business today are facing with immense challenges due to complex applications and rapid growth in data volumes. Many of applications use data for computing statistical data to make proper decision in other applications such as machine learning or social networking. Mostly, these applications assume performing sophisticated client queries with such operators as aggregation and join. State-of-the-art distributed relational databases get over these challenges. Unfortunately, distributed database management systems suffer from failures. Failures causes sophisticated queries with joining large tables are re-executed so that enormous volume of resources must be leveraged.**

In this paper, we introduce a new fault tolerant join algorithm for distributed RDBMS. The algorithm based on mechanisms of data replication and heartbeat messages. We compare our algorithm with traditional, unreliable distributed join algorithm. This paper demonstrates how we achieved trade-off between time required to perform tasks of failed sites and extra resources needed to carry it out.

***Index Terms*—databases, hash-join, fault-tolerance, query processing, replication**

I. Introduction

Let us consider the definition of distributed database systems. *Distributed database systems* are database management systems, consisting of local database systems [1], [2]. These systems with their own disks are located and dispersed over a network of interconnected computers. In this paper, we deal with shared-nothing architecture of distributed database systems.

Fault tolerance is the property of system to keep on carrying out tasks in the event of the failure [1], [3]. A system is able to detect a failure and properly handle it without affecting correct execution of tasks.

Distributed systems, based on *MapReduce* framework, provide high availability, improved performance. They also provide fault-tolerance in case of any site is down. In contrast to distributed systems, parallel RDBMS cannot handle occurred fails so that entire work has to be re-executed. In our work we will focus on how achieve fault tolerance for RDBMS.

Modern distributed database systems are applicable for a variety of tasks such as business intelligence [1], [4]. Mostly, these tasks assume performing sophisticated client queries with such operators as aggregation and join.

The join operation has been studied and discussed extensively in research efforts because it is one of the most time-consuming and data-intensive operations in relational query processing [4]–[7].

Distributed database systems have the main goal is processing an enormous amount of data in a short time, by transmitting data, handling its and passing the final outcome to applications. In failure-free scenarios, database systems may achieve good results. Given that failures are common at large-scale, distributed database systems remain fault tolerance as built-in feature. As example, modern data processing systems have algorithms of recovering tasks of a failed site [8].

According to studies [9], [10], the most common failure types are divided into the following:

- *Communication failure*. This type of failures is specific for distributed systems. There are many types of communication failures - errors in messages, unordered messages, undelivered message, and lost connection. The first two types are responsibility of computer network. Lost messages and/or connection can be the consequence of failed sites or communication failure.
- *System failure*. A hardware of a software failure may cause a system failure. For example, may be either CPU crashed or the presence of bugs in application code which cause failures occur. Once a system failure occurred, content of main memory is loosen.
- *Media failure*. It refers to the failures of the secondary storage devices that store the database. Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures. The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible.
- *Transaction failure*. Transactions can fail for a number of reasons. For instance, a failure can occur due to either incorrect input data or potential deadlock.

As state in [9], hardware and software failures make up more than 90% of all failures. This percentage has not changed significantly since the early days of computing. What is more interesting is the occurrence of soft failure is significantly

higher than that of hardware failures.
In contrast to the occurrence of system failures, the occurrence of transaction failures, as it states in [10], is 3%. Modern database management systems are capable of recovering execution of abrupt interrupted transactions in a short time. In this paper, we do not consider transactions failures.

Unfortunately, state-of-the-art data processing systems do not have algorithms for handling failures when a client performs a query joining two of more tables. Currently, the modern relational and NoSQL distributed database systems re-execute an entire query even if a one node of systems becomes inactive. For example, a node with stored data becomes unavailable when transmitting data to sites where join operation are performed. It causes re-execution of a query with join leads to spend enormous resources and time to end up a join query.

The paper addresses the challenges of recovering tasks at failed sites. Our solution achieves the fault tolerant execution of join queries through making trade-off between completing recovery tasks for the least time and extra resources to be leveraged for this. Data replication insure to reallocate a client query to available site with stored data. An approach of sending heartbeat messages allows to detect a failure of any site and undertake required steps to recover undone work.

We leverage the following notations used in the paper:

TABLE I
NOTATIONS USED IN THE PAPER

Symbol	**Explanation**
R	Input relation R
S	Input relation S
b	Block size
$B(R)$	The number of blocks of relation R
$B(S)$	The number of blocks of relation S
$T(R)$	The number of tuples of relation R
$T(S)$	The number of tuples of relation S
$\|K\|$	The number of keepers
$\|W\|$	The number of workers
$I(A)$	The number of distinct join attribute values
Keepers	Nodes, where data is stored
Workers	Nodes, where join operation are performed

This paper is organized as follows. Section II describes related work. Description of the distributed cost model, used in the paper, is given in Section III. In Section IV, we address to classical distributed hash-join algorithm and examine cost of the execution. Fault tolerant distributed hash-join algorithm and its cost is introduced in Section V. Section VI provides the description and evaluations of how fault tolerant algorithm gets over failures. Comparison of estimations is presented in Section VII. This paper is concluded by Section VIII.

II. Related Work

In this section, we briefly review works dedicated to different approaches of tackling failures in modern framework for processing vast amount of data in parallel called *Hadoop MapReduce* [8]. We examined algorithms [6], [11]–[13] of parallel join for different kind of systems. Analyzed works do not consider the task of handling failures. Authors supposed algorithms work on fail-free systems.

Review [14] describes the state of the art approaches to improve the performance of parallel query processing using MapReduce. Even more, authors discussed significant weaknesses and limitations of the framework. A classification of existing studies over MapReduce is provided focusing on the optimization objective.

Authors proposed a strategy of doubling each task in execution [15]. This stands for if one of the tasks fails, the second backup task will end up on time, reducing the job completion time by using larger amounts of resources. Intuitively, readers may guess that doubling the tasks leads to approximately doubling the resources.

In work [16], authors have devised two strategies to improve the failure detection in Hadoop via heartbeat messages in the worker side. The first strategy is an adaptive interval which will dynamically configure the expiry time adapted to the various sizes of jobs. As authors state, the adaptive interval is advantageous to the small 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.

To remove single point of failure in Hadoop, a new approach of a metadata replication based solution 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 such as version file and file system image are caught up with those of active/primary node; in replication phase, the runtime metadata (such as outstanding operations and lease states) for failover in future are replicated; in failover phase, standby/new elected primary node takes over all communications.

III. Distributed Cost Model

One of the challenges of multi objective query optimization in distributed database management systems is to find Pareto set of solutions or the best possible trade-offs among the objective functions [18]. Objective functions are defined as total time of query execution, I/O operations, CPU instructions and a number of messages to be transmitted.

In distributed database systems the total time of query execution is expressed through mathematical model of weighted average. This model consists of time to perform I/O operations, CPU instructions and time to exchange a number of messages among involved sites. In this work we are going to find trade-off between the least time of the execution in case of failure occurrence and extra resources required to recover failed tasks. We will evaluate time to perform I/O operations, CPU instructions, and communication separately.

In this paper, we consider joining two relations R and S, and *R.A* = *S.B* is the join condition. Assume all data of each relation evenly distributed throughout the system; each node stores $\frac{T(R)}{|K|}$ and $\frac{T(S)}{|K|}$ respectively.

IV. Classical Distributed Hash-Join Algorithm

In this section, we describe distributed hash-join algorithm. We will often refer to distributed hash-join algorithm as classical join algorithm.

At the highest level, the working of distributed hash join in distributed database systems of a shared-nothing architecture is straightforward:

1) A coordinator receives a client query. Then, the coordinator populates messages with a client query across 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...$|K|-1$. If a tuple hashes to value *i*, then it is sent to a worker W_i. Once a keeper ends up reading its partitions of relation R, it notifies the coordinator about the status of work.
3) Each worker W_i 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 the 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) Each keeper reads its partitions of relation S, applies a hash function *h1* to the join attribute of each attribute as it is in the step 2. If a tuple hashes to value *i*, then it has to be sent to a worker W_i.
6) A worker receives a tuple of relation S, probes the hash table built in step 2 to find out a tuple of relation S joins with any tuple of relation R. If so, tuples join and an outcome tuple is generated.

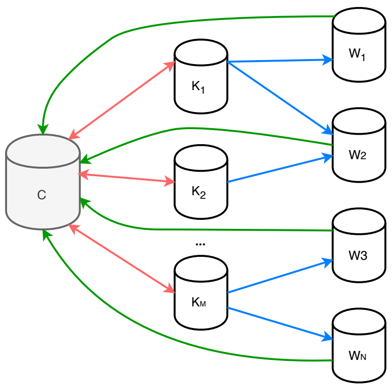

Fig. 1. Scheme of working of distributed join

Figure 1 illustrates a scheme of working distributed join algorithm. The red lines stands for a communication between the coordinator and a keeper at two phases *building* and *probing*. The direction of a red line from a keeper to the coordinator means process of sending a notification to the coordinator with message *a keeper stopped reading its partitions on a particular phase*. The blue lines among keepers and workers denote the process of sending tuples at two phases. Lines of green color, directed from workers to the coordinator, signify submitting outcome tuples.

Here and further in the paper we do not consider the single coordinator receives all client queries. Instead, a new coordinator is assigned for each query.

A. Cost of Classical Distributed Hash-Join

1) Communication cost: the coordinator sends a message to K keepers to take up reading their partitions of relation R. One more message the coordinator sends to keepers to start reading partitions of relation S. At the building phase, keepers send $T(R)$ messages to workers, and at the probing phase they send $T(S)$ messages. The total communication cost:

$$T_{msg} = 2 + T(R) + T(S) \tag{1}$$

2) I/O cost: keepers read $B(R)$ blocks at the *building* phase and workers write $B(R)$ blocks into their disks. At the *probing* phase, keepers read $B(S)$ blocks of relation S. To write the result of matching, it will take

$$IO_{matches} = \frac{B(R)T(S) + B(S)T(R)}{I(A)} \tag{2}$$

The total I/O cost:

$$T_{I/O} = 2B(R) + B(S) + IO_{matches} \tag{3}$$

3) CPU cost: at the both building and probing phases, the coordinator sends 2 messages to keepers. Each keeper receives 2 messages. For reading both relations, keepers perform $B(R) + B(S)$ operations. To write blocks of relation R on each worker, it is required to perform $B(R)$ operations. At the building and probing phases, keepers perform $T(R) + T(S)$ operations to submit messages to workers and workers execute $T(R) + T(S)$ operations to receive messages. Each worker executes

$$CPU_{compare} = \frac{B(R)T(S) + B(S)T(R) + T(R)T(S)}{I(A)|W|} \tag{4}$$

comparing of tuples both relations during the matching, and writing join result. The total CPU cost:

$$T_{CPU} = B(S) + 2(T(R) + T(S) + B(R) + |K| + 1) + CPU_{compare} \tag{5}$$

V. Fault Tolerant Distributed Hash-Join

We identified a few causes the classical distributed join may interrupt at any phase of algorithm.

- The coordinator became unreachable. For example, due to a communication or a system failure. All stored data may be loosen.

- A media or system failure occurred at a keeper or a worker performing operations over its disk.
- A site was suddenly turned off in the middle of the probing phase so the entire work has to be re-executed.

To detect and properly handle the failures above, we decided to leverage the concept of sending heartbeat messages. The concept of heartbeat is widely used in the consensus algorithm Raft [19]. In a heartbeat message, a site may inform another site about status of carried out tasks or pass crucial metadata information.

The approach of heartbeat messages benefits detecting any failures of sites. The coordinator's duty is to send heartbeat messages to all sites and receive responses. Once the coordinator receives heartbeat messages, it has to sort out inputs. If an input contains error message, it stands for an error occurred in a site, for example, software or media failure. If there is no response from a site, it means either a site is unreachable due to communication failure or hardware issues occurred.
We injected the approach of heartbeat messages into our algorithm. Based on content of a received message, the coordinator undertakes required steps to recover work of a failed site.

Apart from telling about communication and system failures, consider failures with disks. A failure with a disk can be caused by power loss, media faults, I/O errors, or corruption of information on the disk. Particularly, a failure with a disk of a worker may cause the loss of intermediate join result. To prevent the loss of data stored in sites, we leverage the strategy of full data replication [20], [21]. Keepers form a ring. Keeper $(i+1) \mod |K|$ stores a full copy of data of the previous keeper $i \mod |K|$. To protect intermediate data of workers from the loss, each keeper copy data for joining to both workers. The first worker executes join operation of both tables whereas the second worker stays on until the coordinator signals to join reserved data.

A. Algorithm

Summarizing said all above, we introduce fault tolerant distributed hash-join algorithm. The 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, the coordinator populates messages with a client query across all nodes. Once messages are sent, the coordinator sets 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 is sent 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 the 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 is in the step 2. If a tuple hashes to value *i*, then it is sent 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 to find out a tuple of relation S joins with any tuple of relation R. 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 heartbeat message the following worker. In this message, it points a position of the last successfully joined tuple of relation S.

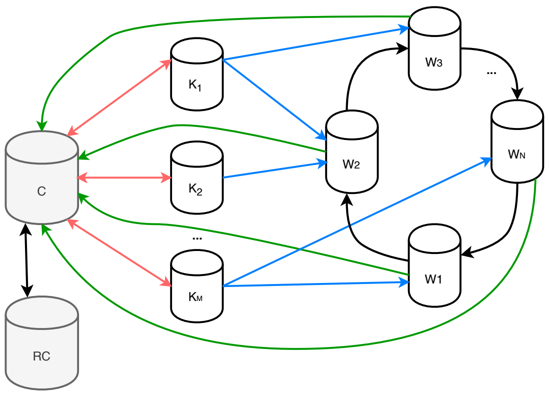

Fig. 2. Scheme of working of fault tolerant distributed join

In the Figure 2 shown a scheme of working of fault tolerant distributed join algorithm. There is the same principle of working as it is shown in Figure 1 for the classical join algorithm. The difference is that there is added a reserved coordinator *RC* and it synchronizes with the primary coordinator *C*. Workers comprise a ring of nodes. Each worker is aware of the following node. It facilitates a worker submits info about proceeded work during the join to the following worker. In case of $i \mod |W|$ worker is failed, $(i+1) \mod |W|$ worker takes over tasks of a failed worker.

B. Cost of Fault Tolerant Distributed Hash-Join Algorithm

1) Communication cost: similar to the classical distributed join algorithm, in fault tolerant algorithm the coordinator sends one message to keepers to take up reading their partitions of relation R and one message to start reading partitions of relation S. At the building phase, keepers send $T(R)$ messages to workers, and at the probing phase they send $T(S)$ messages.

$$T_{msg} = 2 + T(R) + T(S) \tag{6}$$

2) I/O cost: keepers read $B(R)$ blocks at the *building* phase and workers write $B(R)$ blocks into their disks in parallel including reserved blocks of relation R. At the *probing* phase, keepers read $B(S)$ blocks and workers write $B(S)$ blocks as reserved. To write the result of matching, it's required to perform

$$IO_{matches} = \frac{B(R)T(S) + B(S)T(R)}{I(A)} \tag{7}$$

I/O operations. The total I/O cost:

$$T_{I/O} = 2(B(S) + B(R)) + IO_{matches} \tag{8}$$

3) CPU cost: the coordinator carries out *2* operations to submit messages to keepers at the both phases. All in all, keepers perform $2|K|$ operations to receive messages. To read and write both relations by keepers at two phases, it will takes

$$IO_{RW} = 2(B(S) + B(R)) \tag{9}$$

At the building and probing phases, to submit and receive messages with tuples, it will takes

$$C_{SR} = 2(T(R) + T(S)) \tag{10}$$

operations to submit and receive messages with tuples. Each worker executes

$$CPU_{compare} = \frac{T(R)T(S)}{I(A)|W|} \tag{11}$$

operations comparing of tuples both relations during the matching, and

$$CPU_{join} = \frac{B(R)T(S) + B(S)T(R)}{I(A)|W|} \tag{12}$$

operations to write join result. After an outcome tuple is generated and sent to the coordinator, two operations are needed to send a message from $i \mod W$ worker to $(i+1) \mod |W|$ worker.
The total CPU cost:

$$\begin{aligned} T_{CPU} = IO_{RW} + C_{SR} + 2(|K| + 1) + \\ CPU_{compare} + CPU_{join} \end{aligned} \tag{13}$$

VI. Handling Failures

In this section, we examine steps to handle failures during the execution of fault tolerant hash join algorithm.

A. Failure of The Coordinator

As we said before, the coordinator may fail at any phase. For instance, the coordinator becomes unreachable before *probing* phase or even in the middle of the execution of building phase. To eliminate any of those scenarios, the secondary coordinator takes over a work of the failed coordinator. Both keepers and workers have to be aware of the secondary coordinator and be able to quickly join it.
We suggest adding both coordinators' addresses to keepers and workers. When a heartbeat message is sent from a site to the failed coordinator, a node should handle it by re-sending a message to the secondary coordinator and keeping on work with it. A database manager should be aware of a failed coordinator and recover it.

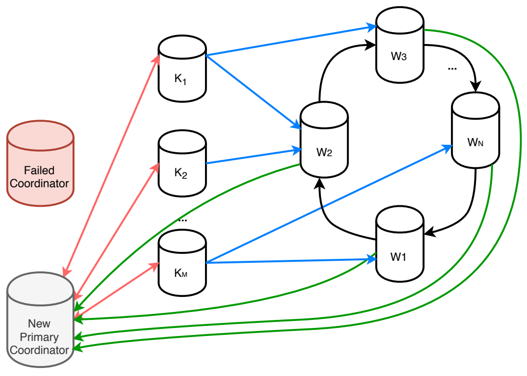

Fig. 3. All messages route to the new coordinator

Figure 3 depicts a case when the coordinator is failed and all messages from nodes re-route to the reserved coordinator.

B. Failure of a Keeper

A keeper may fail at any phase:

1) *Before performing a query*. Once the coordinator knows a keeper is failed, it will redirect tasks to another keeper where reserved data of a failed keeper is stored.
2) *Building or probing*. Despite the phase, a keeper stops sending heartbeat messages to the coordinator. The latter reassigns task to another keeper with stored data of an inactive keeper and says its to keep scanning a particular relation.

To handle the first case, all the coordinator needs to carry out is to mark the failed coordinator as unreachable and not to submit messages to it. We do not calculate cost of performing this case because of we assume the coordinator is aware of the address of the failed keeper and the cost will not affect the query execution.
Let us consider the second case. As we pointed out in Section III, the total cost of execution is the sum of communication cost, I/O cost and CPU cost. Communication cost is composed of sending a message from the coordinator to a keeper and submitting messages of a particular relation to workers.

$$T_{msg} = 1 + \frac{T(R) + T(S)}{|K|} \tag{14}$$

I/O cost implies scanning blocks of two relations, writing blocks of relation R twice in parallel and once blocks of relation S.

$$T_{I/O} = \frac{2(B(R) + B(S))}{|K|} \tag{15}$$

As for CPU cost, it is composed of two operations to send and receive message from the coordinator to a keeper, reading and writing blocks of two relations and submitting and receiving messages with tuples.

$$T_{CPU} = 2 + 2\frac{B(R) + B(S) + T(R) + T(S)}{|K|} \tag{16}$$

The total cost of the execution of recovering work of a failed keeper is the sum of

$$T_{recovery} = (14) + (15) + (16) \tag{17}$$

C. Failure of a Worker

If a worker becomes unavailable at *building* phase, the coordinator submits a notification in the following heartbeat message *stop communicating with an unavailable worker*.
If worker $i \mod |W|$ is in unreachable state at the probing phase, the coordinator sends out a message to worker $(i+1) \mod |W|$ so that it may take over task of joining tuples.
The cost of communication:

$$T_{msg} = 1 \tag{18}$$

I/O cost is composed of the following assumption. During performing join operation, once a worker $i \mod |W|$ sends an outcome tuple to the coordinator, it submits index of a tuple *V* which just joined and sent to the coordinator. Worker $(i+1) \mod |W|$ starts reading from *V + 1* tuple.

$$T_{I/O} = 2\frac{\frac{T(S)}{|W|} - V}{b} \tag{19}$$

CPU cost is the sum of performing sending and receiving messages, reading untreated tuples and comparing them

$$T_{CPU} = 2 + \frac{\frac{T(S)}{|W|} - V}{b} + \frac{(\frac{T(R)}{|W|} - V)(\frac{T(S)}{|W|} - V)}{I(A)} \tag{20}$$

The total cost to recover work of a failed worker is the sum of

$$T_{recovery} = (18) + (19) + (20) \tag{21}$$

VII. Evaluation and Comparison

In this section we computed cost of both algorithms.

The average time of the execution of joining both tables is defined by introducing the probability *P* that the total time of the execution is the sum of probabilities of fail-free execution and total time of work required to recover a work at a failed site at any phase.
The formula below depicts the average time of execution

$$\textit{The average time} = P * NW + (1 - P) * RW \tag{22}$$

where *NW* is the total time of fail-free execution of an algorithm whereas *RW* is the total time of recovering work. In case of fault tolerant join algorithm, under *RW* we assume the execution of recovery work at any phase described in Section VI. For classical distributed join, the recovery work is to re-execute the entire query across all sites.
We consider the following cases: fail-free work, a keeper fails, and a worker fails. For simplicity, we assume that the probability of fault tolerant work of any site equals. As we said in Section III, we analyze different objective functions separately. Table II provides parameters used during evaluations.

TABLE II
Parameters used for computing of costs

T(R) = T(S)	256	50000	100000
$\lvert W\rvert = \lvert K\rvert$	5		
B(R)	16	1000	5000
B(S)	16	500	1000
b	16	20	100

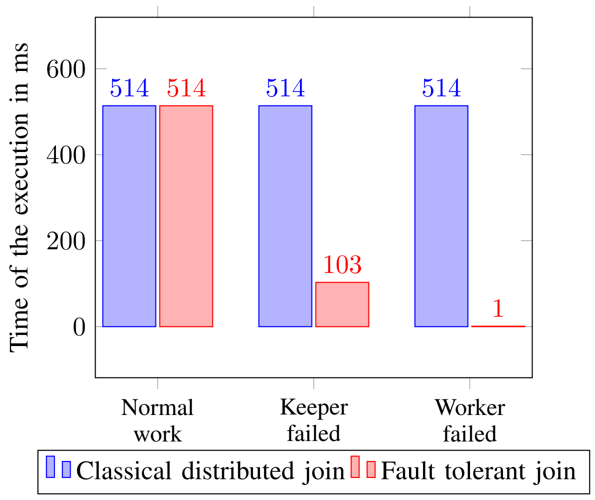

Fig. 4. Comparison of communication time required to execute both algorithms. T(R) = T(S) = 256

Working with disks in fail-free mode of work, the execution of fault tolerant join takes from 5 to 11% more time than classical join takes. This is due to tuples of the second table have to be written and marked as reserved. In the rest cases, fault tolerant join algorithm works with disks 97% time faster.

As for CPU cost, classical join wins in term of time of the execution at fail-free mode. 81% time less requires to recover work of the execution of client query if a keeper fails. In case of a worker fails, it will take 93% less time to assign a task of a failed site to another worker.

The more fascinating results are got when comparing communication costs. In fail-free case, time required to pass data across all sites for both algorithms equals. When a keeper fails, fault tolerant algorithm works 80% time faster than classical algorithm works. In case of a worker fails, fault tolerant join algorithms demonstrates impressive time of the execution. It will 99% faster than its competitor. The results of communication evaluations are shown in Figures 4, 5, 6.

VIII. Conclusion

In this paper, we introduced a fault-tolerant distributed hash-join algorithm. Cost model has been provided, classical and fault tolerant algorithms are compared with each other. In case of failure-free, the unstable algorithm demonstrates better results. In contrast to classical distributed hash join, estimations showed that fault tolerant join algorithm takes precedence over unreliable distributed hash join algorithm.

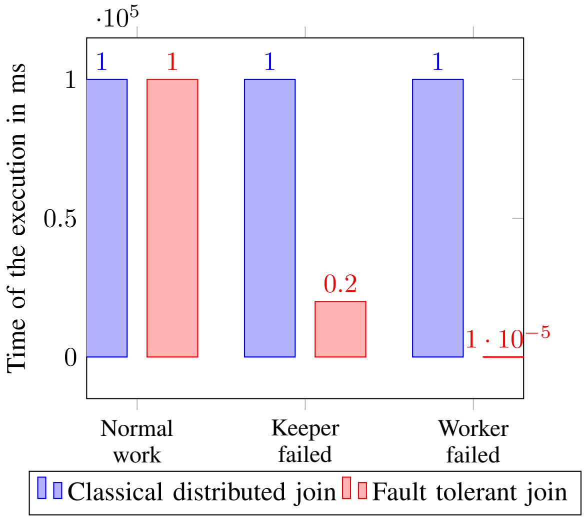

Fig. 5. Comparison of communication time required to execute both algorithms. T(R) = T(S) = 50.000

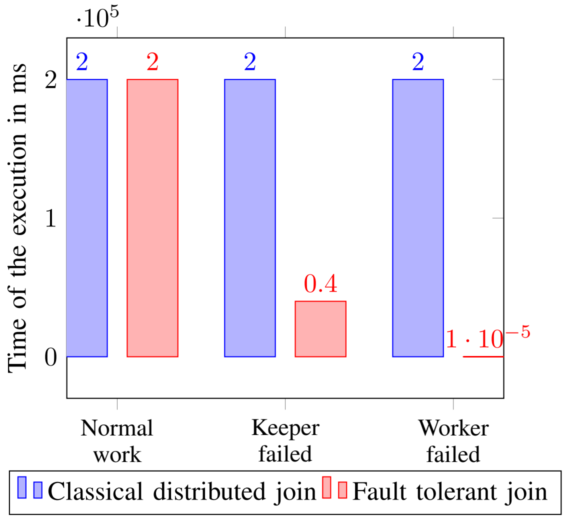

Fig. 6. Comparison of communication time required to execute both algorithms. T(R) = T(S) = 100.000

With double copying tuples of relations to servers, where join is performed, we may omit abrupt failures of workers and keep on joining tuples in available ones.

Future work includes adapting our algorithm to data skew. We will also consider implementing fault tolerant algorithm and conduct experiments with other RDBMS.

References

[1] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[2] S. K. Rahimi and F. S. Haug, *Distributed Database Management Systems: A Practical Approach*. Wiley-IEEE Computer Society Pr, 2010.

[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, Jan. 2004. [Online]. Available: https://doi.org/10.1109/TDSC.2004.2

[4] B. Catania and L. Jain, *Advanced Query Processing: An Introduction*, 01 2012, vol. 36, pp. 1–13.

[5] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, Jun. 1993. [Online]. Available: http://doi.acm.org/10.1145/152610.152611

[6] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 517–528, Jan. 2017. [Online]. Available: https://doi.org/10.14778/3055540.3055545

[7] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687553.1687564

[8] A. S. Foundation. (2019) Apache hadoop. [Online]. Available: https://hadoop.apache.org/

[9] M. T. zsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2011.

[10] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–242, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/356842.356847

[11] G. Gardarin and P. Valduriez, "Join and semijoin algorithms for a multiprocessor database machine," *ACM Transactions on Database Systems*, vol. 9, 03 1984.

[12] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proc. VLDB Endow.*, vol. 7, no. 1, p. 85–96, Sep. 2013. [Online]. Available: https://doi.org/10.14778/2732219.2732227

[13] J. Teubner and G. Alonso, "Main-memory hash joins on modern processor architectures," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1754–1766, July 2015.

[14] C. Doulkeridis and K. Norvaag, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1007/s00778-013-0319-9

[15] P. Costa, M. Pasin, A. Bessani, and M. Correia, "Byzantine fault-tolerant mapreduce: Faults are not just crashes," 11 2011, pp. 32–39.

[16] H. Zhu and H. Chen, "Adaptive failure detection via heartbeat under hadoop," 12 2011, pp. 231–238.

[17] F. Wang, J. Qiu, J. Yang, B. Dong, X. Li, and Y. Li, "Hadoop high availability through metadata replication," in *Proceedings of the First International Workshop on Cloud Data Management*, ser. CloudDB '09. New York, NY, USA: ACM, 2009, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/1651263.1651271

[18] V. Singh, "Multi-objective parametric query optimization for distributed database systems," in *Proceedings of Fifth International Conference on Soft Computing for Problem Solving*, M. Pant, K. Deep, J. C. Bansal, A. Nagar, and K. N. Das, Eds. Singapore: Springer Singapore, 2016, pp. 219–233.

[19] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666

[20] S. H. Son, "Replicated data management in distributed database systems," *SIGMOD Rec.*, vol. 17, no. 4, pp. 62–69, Nov. 1988. [Online]. Available: http://doi.acm.org/10.1145/61733.61738

[21] G. Alonso and B. Kemme, "How to select a replication protocol according to scalability, availability and communication overhead," 08 2001.