PhoeniQ: Failure-Tolerant Query Processing in Multi-node Environments

Yutaro Bessho[1](✉), Yuto Hayamizu[1], Kazuo Goda[1], and Masaru Kitsuregawa[1,2]

[1] The University of Tokyo, 7–3–1 Hongo, Bunkyo-ku, Tokyo, Japan
`{bessho,haya,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp`
[2] National Institute of Informatics, 2–1–2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan

Abstract. Parallel processing is a flagship approach for answering analytical queries on large-scale database. As the database scale increases, a larger number of processing nodes are likely to be incorporated to increase the degree of parallelism. However, this solution results in an increased probability of node failure. If such a failure happens during query processing, the processing often has to restart from scratch. This temporal cost may not be acceptable for the user. In this paper, we propose PhoeniQ, a fault-tolerant query processing mechanism for analytical parallel database systems. PhoeniQ takes a package-level checkpoint for every operator pipeline and replicates the output of stateful operators among different processing nodes. If a single processing node fails during processing, another node is enabled to resume the execution state of the failed node, so that the query can continue to run. This paper presents our intensive experiments based on our prototype, which demonstrate that PhoeniQ can continue the query processing in the face of node failures with significantly smaller cost than the conventional approach.

Keywords: Parallel database system · Fault tolerance · Query processing

1 Introduction

A wide spectrum of big data applications have spurred the growth of database capacity. Petabyte-scale databases are no longer uncommon, especially in cloud-scale companies [6,11,20,23]. The trend of utilizing IoT sensor data is likely to boost the growth further [2].

Parallel query processing is a standard tactic to service analytical queries on large database [5,13,15]. A parallel database system is composed of multiple processing nodes, each of which executes the processing of an assigned part of a given query in parallel. This approach has been actively studied in academia and widely deployed in industry.

Y. Bessho—Currently, he works for NTT.

© Springer Nature Switzerland AG 2020
S. Hartmann et al. (Eds.): DEXA 2020, LNCS 12391, pp. 71–85, 2020.
https://doi.org/10.1007/978-3-030-59003-1_5

A major drawback of such parallel processing is that query processing becomes vulnerable to node failures [17]. As database accommodates larger data, an increased number of processing nodes are often incorporated into the database system. This approach performs well to increase the parallelism. At the same time, it causes a higher aggregate probability of node failure. A relational query is often composed of one or more pipelined operators, each of which can hold an internal execution state. For example, an aggregation operator keeps its in-process data in the memory buffer. If a processing node fails during query processing, the database system loses such an execution state held in the failed node. The system needs to restart the query from ground zero, no matter how far the process has progressed at the point of failure. This time penalty is likely to be unacceptable, particularly for users who run hour-long or day-long analytical queries.

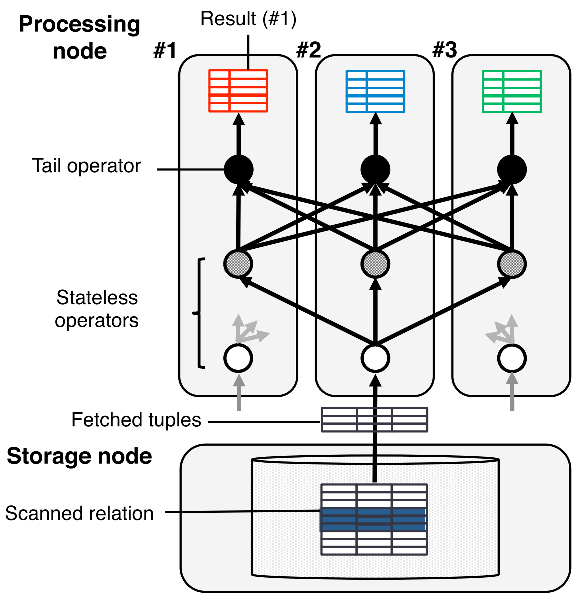
This paper proposes PhoeniQ, a novel fault-tolerant query processing mechanism for analytical parallel database systems. The technical points of PhoeniQ are two-fold. First, PhoeniQ takes a package-level checkpoint for every operator pipeline. Second, it replicates the output of stateful operators among different processing nodes. If a single processing node fails during query processing, another node is enabled to resume the execution state of the failed node, so that the query can continue to run. This paper presents an intensive experiment that we performed with our prototype in a public cloud infrastructure. The experimental result demonstrates that PhoeniQ can continue the query processing in the face of node failure with a significantly smaller time penalty than the conventional approach.

The rest of this paper is structured as follows. Section 2 presents a design overview of PhoeniQ, and Sect. 3 offers a technical deep-dive. Section 4 provides prototype-based experiments in a public cloud environment. Section 5 reviews related work and Sect. 6 concludes the paper.

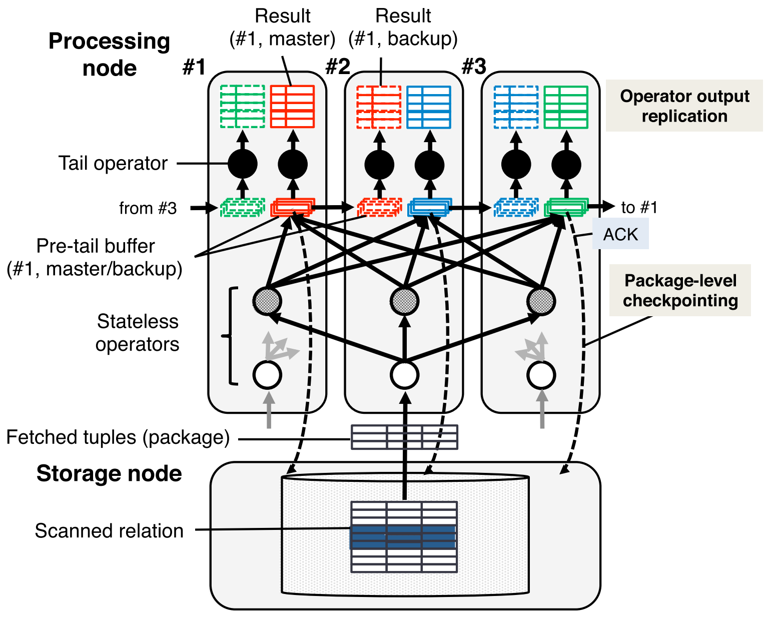
2 Overview of PhoeniQ

First of all, we present a design overview of PhoeniQ, a novel fault-tolerant query processing mechanism for analytical parallel database systems. Figure 1 highlights PhoeniQ by comparing it with the conventional execution mechanism. In this paper, we assume a shared-storage architecture [14] for simplicity[1]. As Fig. 1(a) illustrates, a parallel database system is composed of a single storage node that stores the entire database and multiple processing nodes that process query operators by fetching data from the storage node. According to the query execution plan generated from a given query, a set of pipelined operators are assigned to each node. The first operator in the pipeline is mostly a scan operator that fetches tuples from the storage node, processes them, and passes output tuples to its next operator. The next operator similarly processes received tuples and passes its output tuples to its next operator. Such data flow may travel over

[1] The idea of PhoeniQ can be easily extended to a shared-nothing architecture [26]. Due to the space limitation, we will present further discussion in a separate paper.



(a) Conventional pipeline



(b) PhoeniQ pipeline

Fig. 1. Execution comparison. PhoeniQ takes a package-level checkpoint for every operator pipeline and replicates the output of tail (sometimes stateful) operators among different processing nodes.

multiple processing nodes via the network connection. Finally, the last operator (which we refer to as the *tail operator* in this paper) generates the result of the operator pipeline; the result is often buffered in the memory to be shipped to the user or another operator pipeline or stored in the memory to be accessed later. Note that the tail operator can be stateful, while the other operators are stateless.

Assume that one processing node fails during query processing. The failed node loses intermediate data flows traveling on the node at the moment of failure. Worse, if some operators hold runtime execution states, such data is also lost and the system can no longer continue the query processing. In conventional practice, the database system terminates all the processing whenever a node failure happens, and restarts the query from scratch. This naive solution works, but all the work done so far gets discarded no matter how far the process has progressed. This restarting strategy obviously incurs a significant time penalty.

In contrast, PhoeniQ allows the database system to continue query processing even when a single processing node fails[2]. This unique feature is enabled by two novel techniques illustrated in Fig. 1(b). First, *package-level checkpointing* takes a checkpoint for every operator pipeline, so that the database system can be ready to identify the lost intermediate data flow at the moment of failure and restart merely the affected processing that is necessary to recover the lost data. Second, *operator output replication* copies the output of tail operators to another processing node, allowing the node to resume the execution states of the failed node even when the tail operator is stateful. Section 3 focuses on the technical details of these techniques.

3 Execution Mechanism of PhoeniQ

This section gives a technical deep-dive into PhoeniQ. Sections 3.1 and 3.2 explain the two techniques: *package-level checkpointing* and *operator output replication*, respectively. Section 3.3 describes a tagging technique behind them. Finally, Sect. 3.4 shows a recovery procedure for PhoeniQ.

3.1 Package-Level Checkpointing of Operator Execution States

PhoeniQ allows each processing node to take a checkpoint for every operator pipeline to the storage node. Thanks to this unique capability, whenever a processing node fails, the remaining processing nodes can identify the lost intermediate data flow and restart the only processing that is necessary to recover the lost data. In this subsection, we firstly explain how the storage node manages the execution states of every operator pipeline. The execution state is managed for each tuple of relations scanned by the first operator of the pipeline. However,

[2] For simplicity and due to the space limitation, this paper merely presumes a single-node crash failure of processing nodes. The same idea can be easily applied to other cases, such as a double-node failure. Another exploration is necessary to protect against a failure of the storage node.

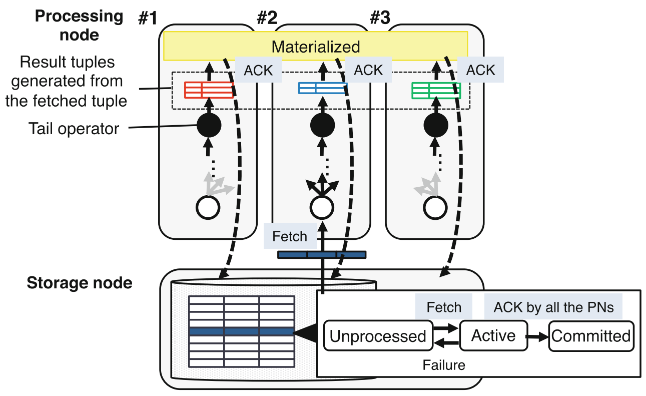


Fig. 2. The execution state management and checkpointing by PhoeniQ. *In this figure, the tail operator is stateless and result tuples are immediately materialized.*

a naive implementation leads to significant performance overhead. We secondly introduce a package-level state management technique to mitigate the overhead.

Execution State Management at the Storage Node. As mentioned earlier, we assume the shared-storage architecture. The basic function of the storage node is to store the database and to deliver a tuple upon a fetch request from a processing node. In processing a given query, each processing node requests tuple fetches to the storage node (mostly for executing a scan operator). In response to each request, the storage node feeds back a tuple to the requesting processing node in the on-demand manner [16].

Our novel idea is to let the storage node additionally manage an execution state of the operator pipeline for each tuple of scanned relations. After delivering a tuple to a processing node, the storage node tracks the execution state of the pipeline for the concerned tuple (Fig. 2). This mechanism enables the storage node to identify if the pipeline execution for each tuple has been completed or not. Thanks to this capability, the storage node can identify the lost part of the query processing in the face of failure, offering the recoverability for the failed query processing. We call this technique *checkpointing* in this paper.

PhoeniQ defines three states for a tuple in the scanned relation as follows.

Unprocessed. The initial state. An unprocessed tuple is not being processed by any of the processing nodes.

Active. An active tuple has been fetched and is being processed by the processing nodes. The tuples generated from an active tuple are not all *fault-safe* (defined later).

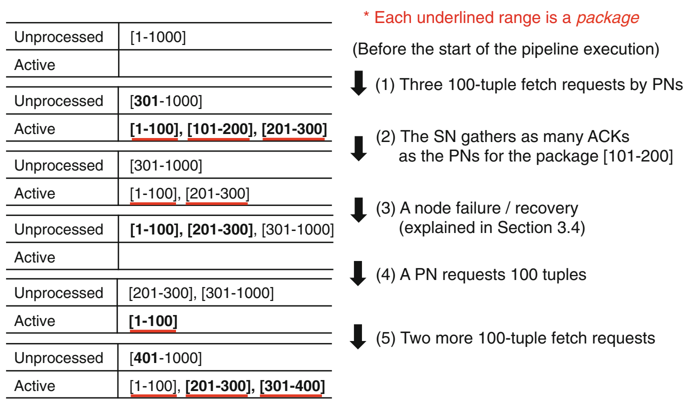
Committed. The tuples generated from a committed tuple are all fault-safe.

For a tuple in the pipeline, to be fault-safe means that it would not be lost in the face of a failure. If the tail operator is stateless and immediately materializes the result tuples (e.g. by delivering them to another processing node, a client terminal, or the storage node), the result tuples become fault-safe automatically and immediately when they have been generated and materialized. In contrast, if a tail operator holds an internal state or buffers the result tuples in the memory buffer, the result tuples do not automatically become fault-safe in the conventional approach. PhoeniQ introduces another technique for handling this case, which is explained later in Sect. 3.2.

At the start of a pipeline execution, all the tuples managed in the database are in the unprocessed state. An unprocessed tuple is turned active when fetched by a processing node. Upon a fetch request by a processing node, the storage node selects an unprocessed tuple and provides it to the requesting node in the on-demand manner.

An active tuple is turned committed when the storage node has received from every processing node a message called *ACK* associated with the tuple. Each processing node sends an ACK for a managed (active) tuple when it has finished its assigned part of the process to make the in-flight tuples generated from the managed tuple all fault-safe. As illustrated in Fig. 2, if the tail operator is stateless and result tuples are immediately materialized, each processing node sends an ACK for the tuple when all the corresponding result tuples generated in the node have been materialized. At the end of the pipeline execution, all the managed tuples have reached the committed state.

On a node failure, only the in-flight tuples corresponding to active tuples are lost and reprocessed after recovery. The recovery procedure is explained in Sect. 3.4.


* Each underlined range is a *package*
(Before the start of the pipeline execution)
(1) Three 100-tuple fetch requests by PNs
(2) The SN gathers as many ACKs as the PNs for the package [101-200]
(3) A node failure / recovery (explained in Section 3.4)
(4) A PN requests 100 tuples
(5) Two more 100-tuple fetch requests

Unprocessed	[1-1000]
Active	

Unprocessed	[**301**-1000]
Active	**[1-100], [101-200], [201-300]**

Unprocessed	[301-1000]
Active	[1-100], [201-300]

Unprocessed	**[1-100], [201-300]**, [301-1000]
Active	

Unprocessed	[201-300], [301-1000]
Active	**[1-100]**

Unprocessed	[**401**-1000]
Active	[1-100], **[201-300], [301-400]**

Fig. 3. An example of tuple processing states management at the storage node. This shows the tuple states changing over a pipeline execution where a 1000-tupled relation is scanned. *[x, y] denotes a range of tuples whose ID is between x and y (containing both ends).*

Package-Based State Management. A naive implementation of the checkpointing scheme would be to take checkpoints at the granularity of tuple. This is impractical, however, because it would incur significant memory footprint and considerable performance overhead. PhoeniQ instead takes a *range-based* approach to reduce the managed information and the processing overhead. This is achieved with the following three techniques.

– The processing nodes fetch multiple tuples from the storage node in one fetch request, instead of one tuple in one request.
– The storage node manages ranges of tuples for each state in a bulky manner, instead of managing a state of each tuple.
– The storage node explicitly manages information of unprocessed or active tuples only and omits that of committed ones. Tuples not present in the unprocessed nor active range list are regarded as committed.

Figure 3 illustrates an example of the package-based state management. Before the pipeline execution, the managed information consists of one range of tuples in the unprocessed state containing all the managed tuples. On receiving a fetch request from a processing node, the storage node cuts out a subset range from the unprocessed range, reads and sends the selected range of tuples to the processing node, and finally turns the tuples active. We call such a group of tuples read out and turned active in response to a fetch request a *package*.

This package-level checkpointing method poses the requirement to process tuples so that the tuples in every package are committed at once, i.e., the tuples generated from a package become fault-safe at once. When the tail operator is stateless and result tuples are materialized as soon as generated, every group of result tuples generated from a package are buffered, gathered, and then materialized at once. How PhoeniQ meets the requirement when result tuples are buffered in the memory during the pipeline execution is explained in Sect. 3.2.

3.2 Operator Output Replication Among Different Processing Nodes

In addition to the checkpointing method, PhoeniQ employs *operator output replication* technique, which copies the output of the tail operator to another processing node. This technique is motivated by situations where the pipeline result needs to be buffered until the end of the pipeline processing (e.g., when the tail operator is stateful). Such accumulated execution states are lost in a failure.

As shown in Fig. 1(b), when operator output replication is enabled, each processing node replicates the computation of the tail operator and its result partition in the logically neighboring node. Thanks to this redundancy, a spare node can restore the lost result as the failed node had at the moment of failure, by receiving it from the neighbors (explained in Sect. 3.4).

Tuples are not immediately input to the tail operator, but are buffered before it. We call this buffer a *pre-tail buffer* of the pipeline. Pre-tail buffers enable tuples to be replicated and fed to the tail operator at the granularity of package. Only tuples generated from committed packages can be fed to the tail operator.

When a processing node knows that it has buffered all the tuples generated from a package and are to come to the node, it sends a copy of the tuples to its logically neighboring node. The receiver again buffers the tuples before the tail operator which leads to the backup copy of the sender node's result partition. We call this buffer the backup copy of the sender node's pre-tail buffer.

When all the processing nodes have successfully replicated tuples made from a certain package, those tuples become fault-safe. Thus, operator output replication allows each processing node to send an ACK for the package to the storage node when its copying to the neighbor node has been completed.

Each processing node periodically asks the storage node which of the tuples in their pre-tail buffers can be forwarded to the tail operator. This query is performed by sending a set of package identifiers to which the buffered tuples correspond. The storage node responds by sending back the set of committed packages. On receiving the answer, the processing node proceeds to deliver the ready tuples (corresponding to committed packages) in its master pre-tail buffer to the tail operator. Similarly, the neighbor node selects tuples in the backup copy of the pre-tail buffer and deliver them to the backup tail operator.

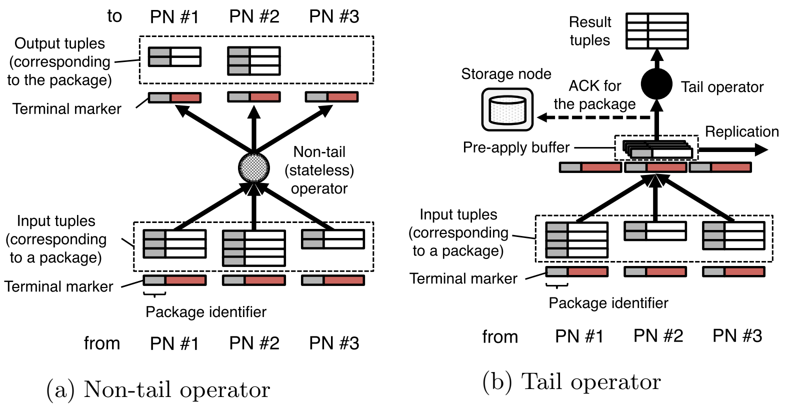


Fig. 4. PhoeniQ tags in-flight tuples so that it can identify the origin package of every tuple and detect the end of tuples generated from every package.

3.3 Tuple Tagging for Package-Level Commit

Package-level checkpointing requires additional information to be attached to in-flight tuples. For a processing node to decide when to send an ACK for a package, the following two conditions must be satisfied.

1. Each processing node can identify the origin package of every in-flight tuple.
2. Each processing node can identify whether or not all the tuples in a package have undergone the processing necessary to become fault-safe.

The first condition is satisfied by tagging each tuple in the pipeline. When a processing node fetches a package of tuples from the storage, the tuples get tagged with the identifier of the package. When these tuples undergo non-tail (stateless) operators, the result tuples are tagged with the same identifier as the input.

The second condition is satisfied by introducing marker tuples denoting that no following tuples correspond to a certain package (called *terminal marker* for a package), as shown in Fig. 4(a)(b). When a processing node has fetched a package of tuples, it appends to the fetched tuples a terminal marker tagged with the package identifier. When an operator redistributes (i.e., shuffles) output, terminal markers passed to it get broadcast.

If the pipeline has multiple shuffles, tuples made from a package can arrive from all the processing nodes. In this case, an operator knows that it has received all the tuples derived from the package only when it has received a terminal marker for the package from all the processing nodes. If the operator is non-tail (stateless) and redistributes output (Fig. 4(a)), it merges the N markers and then broadcasts it (where N is the number of processing nodes). If the operator is the tail operator (Fig. 4(b)), it knows it has gathered all the tuples corresponding to a package when it has seen N terminal markers for the package. It then can (replicate the tuples if operator output replication is enabled, and) issue its ACK for the package.

3.4 Query Processing Recovery

When a processing node fails, the remaining processing nodes invalidate all the in-flight tuples in the executed pipeline whose origin packages are in the active state. The storage node proceeds to rewind all the tuples in the active state to the unprocessed state, so that they can be refetched and reprocessed after recovery[3]. When a spare processing node has joined the query processing to replace the failed node, the pipeline can be restarted immediately if operator output replication is not enabled. Otherwise, the new processing node proceeds to receive the execution states from the two logically neighboring nodes before restarting the pipeline. In Fig. 1(b), for example, when processing node #2 fails, the new node receives the master result of processing node #1 and the backup result of processing node #3.

4 Evaluation

To demonstrate the feasibility and evaluate the effectiveness of our approach, we conducted a series of experiments with our prototype implementation. The experiments consist of two parts: an evaluation of the reduction of failure-recovery time and an evaluation of runtime overhead. In Sect. 4.1, we will describe the experimental setup and the benchmark query for the evaluations, and explain the execution plan. Then, Sect. 4.2 presents the experimental results.

[3] As long as all the non-tail operators are stateless as we have assumed, the reprocessing causes only marginal overhead compared to the entire pipeline processing.

4.1 Experimental Setup and Workload

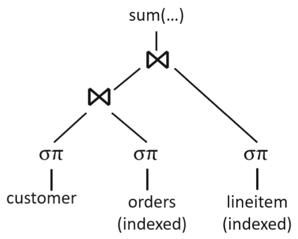
We built our experimental system on public cloud services provided by Amazon Web Services. We used EC2 instances as the processing nodes and the storage node, whose specifications are shown in Table 1. For the storage, we used the instance store of the storage node instance (a SSD drive connected via NVMe).

Table 1. Experimental setup. The prototype system consists of up to sixteen processing nodes (PNs) and one storage node (SN).

	Processing node	Storage node
Instance	t2.medium	i3.xlarge
CPU	2 vCPUs	4 vCPUs
Memory	4 GiB	30.5 GiB
Storage	8 GB	950 GB
	(General purpose, SSD)	(Instance store, NVMe SSD)
OS	Amazon Linux 2	Amazon Linux 2

```
SELECT l_orderkey,
sum(l_extendedprice * (1 - l_discount)),
o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < 1995-03-15
and l_shipdate > 1995-03-15
GROUP BY l_orderkey, o_orderdate, o_shippriority
```

(a) Test query



(b) Query plan

Fig. 5. The test query and its query plan.

Our implementation was configured to run with or without PhoeniQ enabled. When enabled, our system ran with package-level checkpointing and operator output replication introduced in the previous section. In each experiment, we compared the results obtained with our approach enabled and disabled.

We prepared three relations and two indexes in the shared storage. The relations were `customer`, `orders` and `lineitem` from the TPC-H Benchmark [3] with scale factor 100. `orders` had an index file created on its primary key field `o_custkey`, and `lineitem` on `l_orderkey`. The relations were stored as arrays of C structures, and the indexes were Berkeley DB [1] B+ trees (version 18.1.32).

In the experiments, we ran a benchmark query shown in Fig. 5(a) to this dataset. The query involved selections and a joining of the three relations, followed by an aggregation, as illustrated in Fig. 5(b).

The entire query was processed with a single pipeline. The joining of the three relations was performed by scanning `customer` and taking advantage of the indexes. Each processing node first fetched `customer` tuples and applied the selection, and queried the storage node for joining `orders` tuples by join attribute values. The storage node looked up `order` index and provided joining `orders` tuples. Similarly, each processing node demanded joining `lineitem` tuples, and the storage node answered the query by reading `lineitem` index. Joined tuples underwent a hash-based shuffle before input to the aggregation.

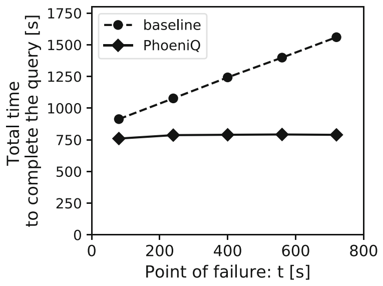


Fig. 6. PhoeniQ incurs almost zero penalties for the failure regardless of the query processing progress, whereas the conventional case incurs much longer execution time if the failure happens at later points in time.

When PhoeniQ was enabled, the computation and the result of the aggregation were replicated among two nodes. The system tracked the progress of the pipeline by managing the states of `customer` tuples. The processing nodes fetched 4096 `customer` tuples in each request, creating 4096-tupled packages.

The storage node ran worker threads, each of which was in charge of processing for each processing node. Each processing node ran two threads when PhoeniQ was disabled: one for the selections and the joining, and one for the aggregation. With PhoeniQ, each processing node ran one extra thread for checkpointing and replicating pre-tail buffers.

4.2 Experimental Results

We performed a scenario where a single node failed during a query execution. The benchmark query was run with 16 processing nodes. After t seconds, the program on one of the processing node instances was terminated. Three seconds after the failure, the failed program was restarted and joined the system. Depending on whether PhoeniQ was enabled or not, the system handled the failure differently. When PhoeniQ was disabled, all the nodes terminated their program, waited for the spare to join, and restarted the query from the beginning. When PhoeniQ was enabled, the system performed the recovery procedure and resumed the query. In either case, after the failure handling, the query was completed without failure.

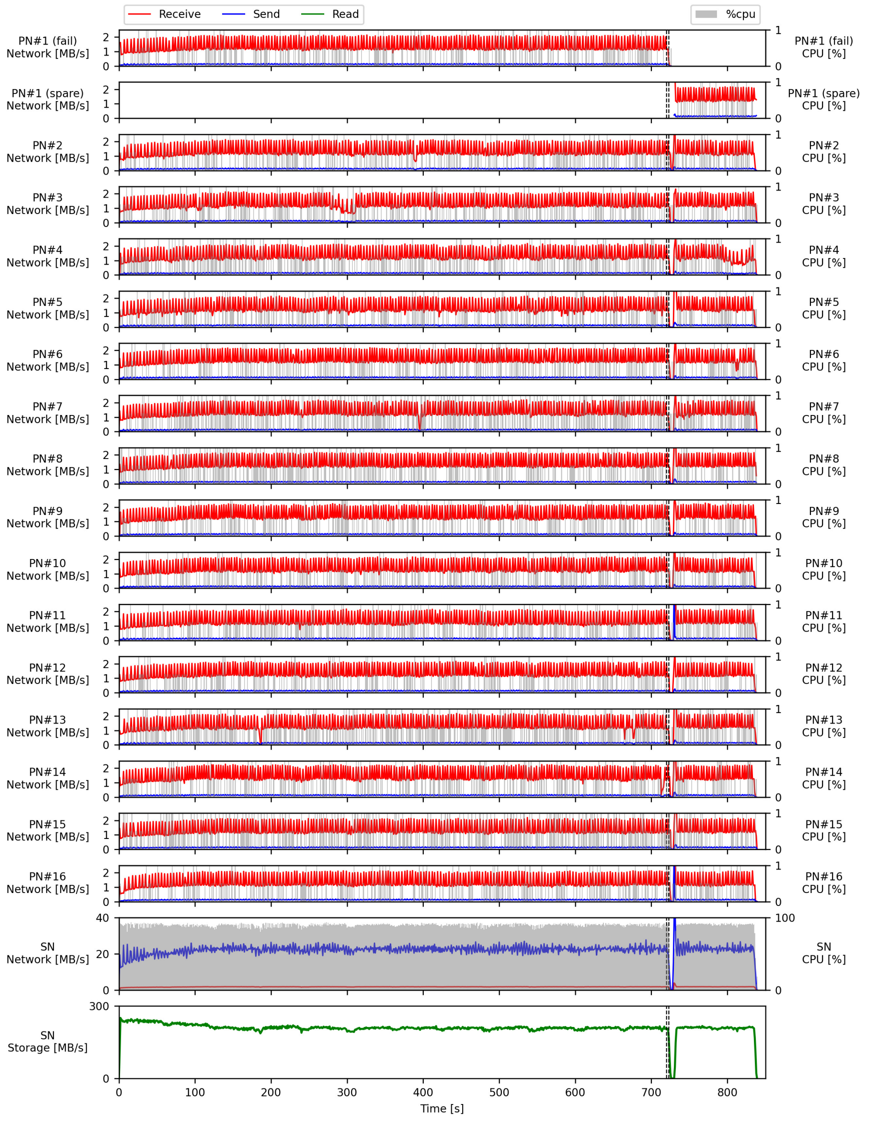


Fig. 7. PhoeniQ dynamically continues query execution even in the face of node failure. PN#1 (fail) terminates the execution at 720 s, but PN#1 (spare) immediately recovers the execution.

Figure 6 shows the total execution time with varying points of failure t (seconds). Without our approach, more time was spent to get the result when the failure took place later. In contrast, no noticeable penalty was present with our approach. At $t = 720$ s (when about 90% of `customer` relation has been scanned), our approach almost halved (−45%) the total execution time.

Figure 7 shows a recovery behavior at $t = 720$ s. In the recovery procedure, the spare node joined the cluster, and received around 80 K tuples in total from the two logically neighboring nodes in around 1.4 s. The spikes in the network throughput of the two nodes after the recovery were caused by the restoration of the result onto the new node. It could also be seen from the network and storage throughput that the system regained processing speed shortly after the recovery. During the entire execution, the CPUs of the storage node were almost fully utilized, whereas those of the processing nodes were underutilized because of the I/O bound characteristics of the workload. It can therefore be inferred that the additional CPU cost added by operator output replication did not affect the execution time in this workload.

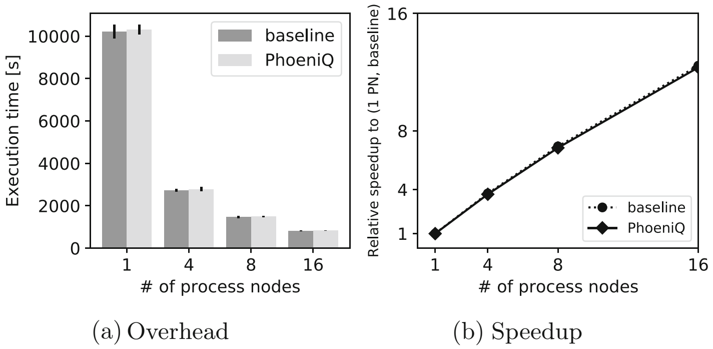


Fig. 8. PhoeniQ enables query continuation with negligible execution overhead and does not disturb scale-out.

Fig. 8(a) shows execution times without failure with varying numbers of processing nodes. Our approach incurred at most 1.6% execution overhead (at 4 processing nodes). This result implies that the overhead introduced by package-level checkpointing, as well as operator output replication, was quite marginal. Figure 8(b) shows that PhoeniQ did not disturb the scale-out.

These experiments demonstrated that PhoeniQ can continue the query processing in the face of node failures with significantly smaller cost than the conventional approach.

5 Related Work

This section outlines the previous work on query restarting techniques.

The previous work for centralized systems mostly aims to restart interrupted queries in favor of those of higher priority [8,10].

For distributed systems, a variety of methods have been proposed to support query fault tolerance. Early MapReduce [12] frameworks write out the output of every process stage to storage. While this allows the query to restart from the latest persisted state, additional I/O cost is not negligible, as demonstrated in [22]. Fault-tolerant query on systems where input data is dynamically provided (as known as stream-based systems [4,7,9]) have been relatively well studied. For example, [24] replicates every computation to backup nodes. In [19,21], master nodes take periodical checkpoints into spare nodes.

For parallel database systems that run queries on static data, several methods [18,25] aim to reduce reprocessing. However, [25] does not consider aggregation operators, and [18] allows a fair amount of recomputation of aggregation. OTPM [17] by B. Han et al. is close to our approach in that it curtails reprocessing of aggregation. In OTPM, operators track the progress of their upstream operator by monitoring the IDs of incoming tuples. The system requires additional nodes to store intermediate results. They have shown promising results from simulation-based evaluation, but a working implementation is not shown. One of the major differences between this approach and ours is that PhoeniQ does not track the progress of every operator. Furthermore, our approach replicates results in a way that does not require additional nodes. Lastly, our approach assumes shared-storage systems, while theirs and all the other work mentioned proposed for distributed settings assumes shared-nothing systems. Shared-storage approach is advantageous in that it does not require sending the data partitions of failed nodes to spare nodes as in a shared nothing system.

6 Conclusion

In this paper, we have proposed a method for parallel database systems to restore execution states on a spare node and to resume query processing. This is achieved by package-level checkpointing and operator output replication. We have implemented a prototype system and performed an experiment with up to 16 processing nodes in a cloud environment. The result shows that our approach successfully reduces restarting temporal penalty on failures with negligible overhead under I/O bound workload. Future work includes conducting experiments with an increased variety of queries. Hash join workloads, for example, are an interesting target. They involve multiple pipelines (separate pipelines for hash build and probe), and a single pipeline can involve multiple shuffles. Moreover, the performance overhead of operator output replication needs careful investigation, because hash joins are generally more CPU-heavy than index joins.

References

1. Oracle Berkeley DB. https://www.oracle.com/database/berkeley-db/db.html
2. The Internet of Things: Data from Embedded Systems Will Account for 10% of the Digital Universe by 2020. https://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm
3. The TPC-H benchmark. http://www.tpc.org/tpch/
4. Abadi, D.J., et al.: The design of the borealis stream processing engine. In: Proceedings CIDR, pp. 277–289 (2005)

5. Boral, H., et al.: Prototyping bubba, a highly parallel database system. IEEE Trans. Knowl. Data Eng. **2**(1), 4–24 (1990)
6. Borthakur, D.: Petabyte scale databases and storage systems at facebook. In: Proceedings SIGMOD, pp. 1267–1268 (2013)
7. Carney, D., et al.: Monitoring streams - a new class of data management applications. In: Proceedings VLDB, pp. 215–226 (2002)
8. Chandramouli, B., Bond, C.N., Babu, S., Yang, J.: Query suspend and resume. In: Proceedings SIGMOD, pp. 557–568 (2007)
9. Chandrasekaran, S., et al.: Telegraphcq: continuous dataflow processing for an uncertain world. In: Proceedings CIDR (2003)
10. Chaudhuri, S., Kaushik, R., Ramamurthy, R., Pol, A.: Stop-and-restart style execution for long running decision support queries. In: Proceedings VLDB, pp. 735–745 (2007)
11. Daniel Weeks: Netflix: Integrating Spark at petabyte scale. https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/43373
12. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
13. DeWitt, D.J., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)
14. DeWitt, D.J., Madden, S., Stonebraker, M.: How to build a high-performance data warehouse how to build a high-performance data warehouse. http://db.csail.mit.edu/madden/high_perf.pdf
15. Ghandeharizadeh, S., DeWitt, D.J.: Hybrid-range partitioning strategy: a new declustering strategy for multiprocessor database machines. In: Proceedings VLDB, pp. 481–492 (1990)
16. Goda, K., Tamura, T., Oguchi, M., Kitsuregawa, M.: Run-time load balancing system on san-connected PC cluster for dynamic injection of CPU and disk resource - a case study of data mining application. Proc. DEXA. **2453**, 182–192 (2002)
17. Han, B., Omiecinski, E., Mark, L., Liu, L.: OTPM: failure handling in data-intensive analytical processing. In: Proceedings CollaborateCom, pp. 35–44. IEEE (2011)
18. Hauglid, J.O., Nørvåg, K.: Proqid: partial restarts of queries in distributed databases. In: Proceedings CIKM, pp. 1251–1260. ACM (2008)
19. Hwang, J., Xing, Y., Çetintemel, U., Zdonik, S.B.: A cooperative, self-configuring high-availability solution for stream processing. In: Proceedings ICDE, pp. 176–185 (2007)
20. Jeff Barr: Migration Complete - Amazon's Consumer Business Just Turned off its Final Oracle Database. https://aws.amazon.com/blogs/aws/migration-complete-amazons-consumer-business-just-turned-off-its-final-oracle-database/
21. Kwon, Y., Balazinska, M., Greenberg, A.G.: Fault-tolerant stream processing using a distributed, replicated file system. Proc. VLDB **1**(1), 574–585 (2008)
22. Pavlo, A., et al.: A comparison of approaches to large-scale data analysis. In: Proceedings SIGMOD, pp. 165–178 (2009)
23. Reza, S.: Uber's Big Data Platform: 100+ Petabytes with Minute Latency. https://eng.uber.com/uber-big-data-platform/
24. Shah, M.A., Hellerstein, J.M., Brewer, E.: Highly available, fault-tolerant, parallel dataflows. In: Proceedings SIGMOD, pp. 827–838. ACM (2004)
25. Smith, J.E.T., Watson, P.: A rollback-recovery protocol for wide area pipelined data flow computations (2004)
26. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**, 4–9 (1985)