Dynamic Fault Tolerance for Multi-Node Query Processing

Yutaro BESSHO†*, Yuto HAYAMIZU†, Nonmembers, Kazuo GODA††, Member, and Masaru KITSUREGAWA†, Fellow

SUMMARY Parallel processing is a typical approach to answer analytical queries on large database. As the size of the database increases, we often try to increase the parallelism by incorporating more processing nodes. However, this approach increases the possibility of node failure as well. According to the conventional practice, if a failure occurs during query processing, the database system restarts the query processing from the beginning. Such temporal cost may be unacceptable to the user. This paper proposes a fault-tolerant query processing mechanism, named PhoeniQ, for analytical parallel database systems. PhoeniQ continuously takes a checkpoint for every operator pipeline and replicates the output of each stateful operator among different processing nodes. If a single processing node fails during query processing, another can promptly take over the processing. Hence, PhoeniQ allows the database system to efficiently resume query processing after a partial failure event. This paper presents a key design of PhoeniQ and prototype-based experiments to demonstrate that PhoeniQ imposes negligible performance overhead and efficiently continues query processing in the face of node failure.

key words: parallel database systems, fault tolerance, query processing

1. Introduction

A variety of big data applications are spurring database capacity growth. Petabyte scales are no longer uncommon, especially in cloud-scale enterprises [1]–[4]. Emerging IoT sensors have the potential to further drive the growth. Surmounting the increasing scale is still technically challenging for database systems.

Intra-query parallelism is a key factor to quickly answer analytical queries on large-scale database [5]–[7]. One major direction is multi-node processing, which scatters the query processing work among multiple processing nodes [8]–[11]. Employing a higher number of processing nodes potentially allows the query processing to utilize more underlying resources such as processing capacity and I/O bandwidth, thus speeding up the query. So far, this approach has been actively studied by researchers. Actually, multi-node processing is widely deployed in industry.

Multi-node processing is vulnerable to node failure [12]. Increasing the number of processing nodes potentially increases the parallelism of query processing; at the same time, it increases the aggregate probability of node failure. If a processing node fails during query processing, its execution status is totally lost and cannot be recovered by any other. Thus, the database system has to restart the query processing again from scratch, regardless of the progress of the query processing at the point of failure. The temporal penalty is not acceptable for many analytic users, who need to wait for the query processing to complete for hours and days at times.

PhoeniQ is a query execution method, which we have developed, for providing dynamic fault tolerance to parallel database systems [13]. PhoeniQ allows parallel database systems to continue the query processing even in the face of a node failure event. This paper presents the design and prototype based evaluation of PhoeniQ. PhoeniQ continuously takes a checkpoint for every operator pipeline and replicates the output of stateful operators among different processing nodes. If a single processing node fails during query processing, PhoeniQ allows another node to quickly take over the processing. We have prototyped a database system based on PhoeniQ and experimentally validated its effectiveness with up to twenty-four processing nodes plus a shared storage node in a public cloud environment. The experiment has clarified that PhoeniQ only imposes negligible performance overhead in normal times and quickly resumes the query processing after node failure. To our knowledge, similar attempts have not been reported in literature by others.

The rest of this paper is structured as follows. Section 2 presents a design overview of PhoeniQ, and Sects. 3 and 4 offer a technical deep-dive into PhoeniQ. Section 5 provides prototype-based experiments in a public cloud environment. Section 6 reviews related work and Sect. 7 concludes the paper.

2. PhoeniQ

This section gives an overall design of PhoeniQ, a query execution method for providing dynamic fault tolerance to parallel analytical database systems. Figure 1 comparatively highlights the execution mechanism of PhoeniQ.

Figure 1 (a) presents a conventional execution method for providing dynamic fault tolerance to parallel analytical database systems. Figure 1 (b) presents a conventional execution method for providing dynamic fault tolerance to parallel analytical database systems.
Fig. 1 Comparison of execution mechanisms. (a) Conventional parallel database systems fetch tuples from storage and feed them into multiple operator pipelines to obtain a query result. (b) Additionally, PhoeniQ continuously tracks the execution progress of the operator pipelines, and replicates their execution states to allow the run-time recovery from a node failure.

In each processing node, the query operators are often pipelined; typically, a head operator in the pipeline is a scan operator, which fetches the tuples from the storage node and feeds them into the pipeline.

Suppose that an unrecoverable event such as hardware failure and software crash occurs at one of the processing nodes performing query processing. The database system is not able to continue the query processing, because the failed node loses the data flowing through the operator pipeline and the execution states held by stateful operators at the point of the failure. The common practice is to terminate the in-flight query processing and then restart the processing from the beginning, regardless of the processing progress at the point of failure. This naive strategy produces an unacceptably significant temporal penalty at times.

PhoeniQ is our solution to allow parallel database systems to overcome such failure. Figure 1 (b) illustrates how PhoeniQ prepares for a failure event. First, PhoeniQ continuously tracks the execution progress of every operator pipeline running in the processing nodes. Second, PhoeniQ replicates the output state of the operator pipelines to another processing node. Thus, even when a node failure happens during query processing, PhoeniQ can identify the lost part of the query processing work done so far and invoke a takeover process to efficiently recover the lost work and resume the query processing. Sections 3 and 4 present its technical deep-dive. As an early challenge, this paper mainly describes a single-node crash failure of processing nodes. Further exploration is necessary for other types of failures such as a double-node failure and a failure of the storage node.

3. Checkpointing for Stateless Operator Pipeline

Here, we describe how PhoeniQ continues query processing in an event of node failure. For simplicity, this section starts the description for a query only comprising a stateless operator pipeline. Section 4 then extends it to stateful cases.

3.1 Stateless Operator Pipeline and Its Recoverability

Again, we assume that a query is composed of a stateless operator pipeline. Figure 2 presents two examples. Figure 2 (a)
presents a simple selection query, which scans a relation \( R \) and answers tuples satisfying a given selection condition. Figure 2 (b) describes its operator pipeline. The first \( \text{scan} \) operator fetches tuples from storage and feeds them into the pipeline, the second \( \text{selection} \) operator passes only tuples which meet the selection condition, and finally the last \( \text{return} \) operator answers the tuples to a query requestor (e.g., a database client). Similarly, Figs. 2 (c) and 2 (d) present a two-way nested-loop join query, which scans an outer relation \( R \), searches matching tuples in an inner relation \( S \), and answers them to a query requestor. The property of these operator pipelines is that an output has been generated from a part of the input tuples.

Suppose that, while a stateless operator pipeline scans all the tuples \( T \) from a relation, a failure happens at the processing node. Until the point of failure, the operator pipeline has fetched tuples \( T_f(\in T) \) from the relation and answered an output that has been generated from a part \( T_a(\in T_f) \) of the input tuples. Obviously, an efficient solution for resuming this processing from the failure is to execute again the operator pipeline only for the unanswered part \( T - T_a \), so that the query requestor can eventually obtain a complete answer. A primary idea of PhoeniQ is to allow the storage node to identify the unanswered part \( T - T_a \) when a failure happens at a processing node.

3.2 Checkpointing of Operator Pipelines

PhoeniQ takes a checkpoint for every operator pipeline of processing nodes and materializes it to the storage node during query processing. Suppose a single processing node stops its processing due to failure, but the other surviving nodes keep going. PhoeniQ can identify the lost part of processing at the failed node by using the materialized checkpoint information and invoke the processing that is necessary for recovering the lost part to continue the query processing. Here, we firstly describe a basic framework of execution state management done at the storage node. Sections 3.3 and 3.4 then present its efficient implementation techniques for reducing the performance overheads involved with the checkpointing.

This paper assumes the shared-storage architecture, which places multiple processing nodes and a single storage node. The storage node manages shared storage space and, upon a request, delivers a requested tuple to a processing node for query processing. In general, a scan operator, namely a head operator of an operator pipeline, typically iteratively utilizes a fetch-next command to scan the tuples from a relation and feed them into the pipeline. The fetch-next command requests a tuple that has not been fetched for query processing. The storage node manages the fetching status (e.g., the information regarding which tuples have been already fetched and not yet fetched), and decides a tuple to be delivered for every request in the on-demand manner [16].

PhoeniQ extends this mechanism at the storage node to allow to keep track of the execution progress of an operator pipeline for every tuple that is fed into the pipeline by the fetch-next command. Figure 3 illustrates the state management at the storage node. Specifically speaking, in the conventional scheme, the storage node only manages two states for the fetch-next command; a tuple is in a state of having not been fetched by any processing nodes, or in a state of having been fetched by a certain processing node. In contrast, PhoeniQ extends this state management, enabling the storage node to additionally identify if a result of an operator pipeline becomes committed regarding every tuple delivered by the fetch-next command. Herein, being committed means that an output has become durable or recoverable even if a concerned node fails, such as being answered to a query requestor. Assume that a processing node fails during query processing. The recovery process can identify the lost part of the processing at the failed node by enumerating the tuples that have been delivered to the failed node but have not yet become committed at the point of failure. This checkpoint mechanism offers the recoverability for the failed query processing.

The list below summarizes the three states that the storage node manages for the fetch-next command system in PhoeniQ.

- An \textit{unprocessed} state means that a tuple has not been fetched by any processing nodes.
- An \textit{in-process} state means that a tuple has been already fetched but an output of an operator pipeline associated with the tuple has not been committed.
- A \textit{committed} state means that an output of an operator pipeline associated with the tuple has been committed.

When PhoeniQ starts the execution of an operator pipeline, the storage node sets all the tuples to be scanned to the unprocessed state. Upon the fetch-next request given
by processing nodes, the storage node distributes fetched tuples to the processing nodes, each of which then executes an assigned operator pipeline for each incoming tuple and, every time an output is committed (e.g., delivered to the requestor), notifies the acknowledgement to the storage node for the concerned tuple. In turn, upon the notification of acknowledgement, the storage node transitions the corresponding tuple into the committed state. This iteration continues until all the tuples turn into the committed state.

If a failure event happens at a processing node during query processing, the storage node simply moves back all the in-process tuples into the unprocessed state. The same operator pipeline running on another node (e.g., a surviving node or a spare node) would automatically fetch and feed those tuples so that the lost part of the query processing would be recovered and the remaining processing could be resumed eventually.

### 3.2.1 Package-Level Checkpointing

The checkpointing mechanism actually works, but it incurs significant memory footprint and communication overhead because a checkpoint is managed at the granularity of tuple; the overall performance of query processing is likely to degrade. We have incorporated an efficient implementation technique for PhoeniQ, which manages a state at a package level instead of a tuple level, to reduce the state management overhead. Here, a package denotes a set of contiguous tuples being identified with a disjoint tuple range. How to specify the range depends on the software implementation, but in many cases, we can use logical reference keys (e.g., primary keys and index keys) and physical pointers (e.g., RowIDs) [17].

In the package-level checkpointing, the communication between the processing nodes and the storage node is based on packages. Upon the fetch-next command, the storage node delivers a package of multiple tuples to a processing node at a time. Similarly, the acknowledgement notification informs that a package of multiple tuples is committed.

Figure 4 illustrates how PhoeniQ manages a tuple state at the package level. Initially, all tuples are in a single package. Every time the fetch-next command is requested, a concerned package in the unprocessed state is divided and the fetched part moves into the in-process state. Similarly, every time the acknowledgement is notified, a concerned package in the in-process state is divided and the acknowledged part comes into the committed state. On failure, packages in the in-process state turn back into the unprocessed state.

The package-level approach keeps the recoverability nature of query processing and reduces the state management overhead.

#### 3.2.2 Package Tagging

When delivering an output from an operator pipeline, the processing node must identify an origin package (fetched by a scan operator) corresponding to the output to make a decision on whether to notify the acknowledgement for the package and, if necessary, to perform the notification. We have incorporated a package tagging technique for PhoeniQ, which appends a package terminal marker behind each in-flight package flowing in an operator pipeline. The marker contains identification information of the origin package. Thus, when processing an incoming package, the last operator of the pipeline can identify its origin package.

Figure 5 illustrates how the package tagging technique works. A scan operator at the head of an operator pipeline appends a package terminal marker behind each outgoing package. If the scan operator transmits an outgoing package only to the same processing node, the package terminal marker also goes to the same node as presented in Fig. 5 (a). In contrast, assuming that the scan operator transmits outgoing packages to all the processing nodes in order to shuffle the tuples, the package terminal marker is also distributed to all the processing nodes as presented in Fig. 5 (b).

An intermediate operator (not staying at the head or the tail of an operator pipeline) basically lets an incoming package terminal marker pass through itself. If this operator accepts an incoming package and its terminal marker from multiple processing nodes (i.e., assuming that its preceding operator does the shuffling), it merges the package terminal marker. Similarly to the scan operator, this operator replicates the package terminal marker if distributing outgoing packages to all the processing nodes as presented in Fig. 5 (c).

A tail operator makes a decision on whether to notify the acknowledgement for each incoming package. If the operator pipeline does not contain any shuffling operator, all the packages arriving at the tail operator are supposed to originate from an origin package fed by the scan operator running on the same node. In this case, when transmitting an output for each incoming package, the tail operator notifies the acknowledgement for the origin package that is identified by the package terminal marker. In contrast, if the operator pipeline contains a shuffling operator,

---

*Even if a package delivers no tuple to a certain processing node, its package terminal marker is delivered so that origin package information can be correctly propagated.*
Fig. 5 Package tagging. (a) A scan operator puts a package terminal marker (having a package identifier internally) at the tail of every tuple package fetched from storage and forwards it to the next operator, (b) the scan operator duplicates the package terminal marker when shuffling packages, (c) a stateless operator merges and replicates in-coming package terminal markers in order, and (d) a tail operator notifies acknowledgement for every in-coming package after processing its package terminal marker.

packages are supposed to be shuffled among all the processing nodes. In this case, the tail operator temporally buffers an output for each in-coming package; when the in-coming packages originating from an identical origin package (to be fetched by the scan operator) from all the processing nodes are ready in the buffer, the tail operator transmits the output and notifies the acknowledgement for the origin package as presented in Fig. 5 (d).

Note that this paper assumes that each operator pipeline running in processing nodes keeps the tuple processing order for simplicity. This assumption is held in the design of many database engines, but in reality, known exceptional techniques change the tuple processing order. For example, Graefe proposed the exchange operator, being placed in an operator pipeline, which allows database engines to reorder the tuple processing order and to improve the vertical parallelism [6]. One solution to apply PhoeniQ to such database engines is to put an additional package identification value (hopefully consuming a small amount of space) to each in-flight tuple, so that the origin package can be identified even though the tuple order changes. Hopefully, the additional value consumes a small amount of space, but its possible performance overhead should be carefully investigated. Another solution is to synchronize the operator pipeline at a point of the package terminal marker, so that no package terminal markers do not come ahead its dependent tuples. With sufficiently large package sizes, the tuple reordering still potentially work, but possible performance overhead due to the limited reordering should be carefully investigated, too.

3.3 Resumption of Query Processing

On recognizing the failure, the processing nodes invalidates the work of the in-process tuples and the storage node rolls back the in-process tuples to the unprocessed state. Then, PhoeniQ restarts the surviving processing nodes, which automatically fetch those tuples and apply them to the operator pipeline. In this step, a designated spare node or a newly allocated node may be injected to replace the failed node. Hence, the on-going query processing can continue and eventually return its complete answer to the designated requestor.

In summary, with a focus on stateless operator pipelines, this section describes the preparatory and recovery techniques employed in PhoeniQ so far. Being put together, these techniques allow the parallel database systems to prepare for a possible failure event with a small amount of performance overhead, and even if a fatal failure happens on a processing node, to quickly recover the lost part of the query work and resume the processing.

4. Output Replication for Stateful Operator Pipelines

The description presented in Sect. 3 is limited to a query only comprising a stateless operator pipeline. In reality, many queries necessitate stateful operations, which mostly stay at the tail of operator pipelines. This section extends PhoeniQ to query processing that contains such stateful cases.

4.1 Stateful Operator Pipeline

Figure 6 presents two examples of stateful operator pipelines. Figure 6 (a) presents an aggregation query, which scans a relation and outputs the number of tuples satisfying with a given selection condition. Figure 6 (b) describes its operator pipeline. The last count operator counts up the number of the in-coming tuples, which is later notified to a query requestor. Similarly, Figs. 6 (c) and 6 (d) present a build phase of a hash join query, which scans an outer relation and builds a hash table, so that a later probe phase can probe the hash table to perform a join. The unique property of these operator pipelines is that an output (e.g., the count number and the hash table) of an operator pipeline is deter-
mined by multiple input tuples, and then it is later fetched or referenced by another pipeline. In this paper, we use the term stateful for referring to this type of operator pipelines, because the pipeline holds an output state that is shared between different input tuples.

Merely utilizing the mechanism presented in Sect. 3 cannot provide the recoverability for such stateful operator pipelines. First, the checkpointing technique only assumes that the operator pipeline is stateless; it cannot trace the output state of stateful operator pipelines. Second, if a fatal failure happens in a system, the failed node loses an output state of the running stateful operator pipeline, but any other nodes cannot recover it. To solve these issues, we have extended the checkpointing mechanism to stateful operator pipelines and have incorporated a replication technique.

4.2 Revised Checkpointing

The checkpointing mechanism introduced in Sect. 3 only considers stateless operator pipelines; an in-coming tuple fetched by the scan operator is assumed to be committed when an output originating from the tuple is answered to a query requestor. We extend this to stateful operator pipelines, in which the resulting tuples are accumulated into the output state. As a result, PhoeniQ identifies that an incoming tuple fetched by the scan operator is committed and notifies the acknowledgement

- when an output originating from the tuple is answered to a query requestor (for a stateless operator pipeline); or
- when an output originating from the tuple is applied to the output state (for a stateful operator pipeline).

4.3 Output State Replication

PhoeniQ replicates the output state of the stateful operator pipeline to another processing node. Even if a processing node fails due to hardware errors or software crash and loses its output state, another processing node having its replica can recover the original state, so that the query processing can continue.

As the examples noted above indicate, the output state is mostly generated and held by the tail operator. Our software implementation has opted to duplicate the tail operator between a pair of processing nodes. Specifically speaking, for a tail operator placed in each processing node, its backup tail operator is placed in its neighboring processing node as illustrated in Fig. 1 (b). The preceding operator delivers the identical data flow to the original tail operator and the backup tail operator. The pair of operators work identically, except that the original tail operator finally delivers the output state, whereas the backup tail operator does not in normal times, but only keeps the redundancy. The acknowledgement is notified separately; the storage node identifies that an origin tuple is finally committed when the acknowledgement arrives both from the original tail operator and its backup.

When a failure happens at a processing node, PhoeniQ restores the output state held in the failed node by using the replicated output state, and then resumes the query processing. Thus, PhoeniQ is allowed to continue queries containing a stateful operator pipeline.

5. Experimental Evaluation

This section presents prototype based experiments that we conducted to clarify the effectiveness of PhoeniQ.

5.1 Experimental Setup

We implemented an experimental database system based on PhoeniQ. The system employed the shared-storage architecture, namely consisting of multiple processing nodes running query operators and a single storage node managing the database storage. For comparative studies, we configured the experimental system so that we could change the number of processing nodes, and activate or deactivate PhoeniQ for every query processing test. When PhoeniQ was activated, the query processing was performed with the fault-tolerant mechanism as illustrated in Fig. 1 (b); otherwise, the query processing was performed in the conventional mechanism as illustrated in Fig. 1 (a).

The experimental system was built on public cloud services provided by Amazon Web Services. Table 1 summarizes the specification of the built system. We tested two different server instances, c4.large and m4.large, for the processing nodes, and we only used i3.xlarge for the storage node. All the server instances were placed in an identical placement group so that the inter-server communication could perform efficiently. The storage server was configured to manage all the database content in its local NVMe-connected storage device because it provided much higher bandwidth than other networked storage. In the storage
Table 1  Experimental setup. The prototype system was built with twenty-four m5.large / c4.large EC2 instances for the processing nodes and a i3.xlarge EC2 instance for the storage node, which all were placed at an identical placement group in Amazon Web Services’ Asia Pacific (Tokyo) region.

| Processing node | Storage node |
|-----------------|--------------|
| Instance type   |              |
| Number of instances |     |
| Processor       | Instance storage |
| Memory          | EBS storage  |
| Network         | OS           |
| m5.large        | Up to 10 Gbps |
| c4.large        | moderate     |
| i3.xlarge       | Up to 10 Gbps |
| m5.large        | 950 GB       |
| c4.large        | 8 GB         |
| i3.xlarge       | –            |
| m5.large        | 8 GB         |
| c4.large        | –            |
| i3.xlarge       | gp2          |
| m5.large        | gp2          |

Fig. 7  Test query and query plan. The test query is a simplified version of TPC-H Query 3, joining major three relations, CUSTOMER, ORDERS and LINEITEM. The query joins these relations in a nested-loop manner and performs aggregation.

node, we ran multiple worker threads, each being in service for every processing node. In contrast, in each processing node, we invoked a single worker thread for every operator.

5.2 Test Workload

We generated the TPC-H [18] dataset with the scale factor of 100 and loaded it to the system. Tuples were stored in the row-oriented manner, and indices were built for the primary keys and the foreign keys with the assistance of Berkeley DB [19] B+ trees.

Figure 7 (a) presents a test query that we employed in this study. The test query is a simplified version of the original Query 3, which joins three major relations, CUSTOMER, ORDERS and LINEITEM, and then performs an aggregation. The reasons why we decided to use this query are threefold. First, a major technical characteristic of PhoeniQ is in its capability of tracking the internal state of operator pipelines and recovering it in a case of node failure. A query having multiple stateless operators and a stateful operator in a pipeline is a good candidate to demonstrate the effectiveness of PhoeniQ. Query 3 meets this condition because it contains multiple joins and aggregation. Second, Query 3 and its variants have been studied in other database performance research [20]–[22]. This is probably because Query 3 is the earliest-numbered query that performs a join operation with LINEITEM, the largest table in TPC-H. Third, the three-way join of CUSTOMER, ORDERS and LINEITEM (a major portion of Query 3) is a typical join pipeline in the TPC-H schema. Actually this join pipeline is a major portion of Queries 5, 7, 10 and 18 and it is also contained in Queries 8 and 9. The original Query 3 contains an “order by” operation, which has been omitted in the test query for simplicity. Figure 7 (b) illustrates a query plan of the test query; CUSTOMER, ORDERS and LINEITEM are joined in the nested-loop manner and the joined tuples are aggregated using a hash table. All these operators are pipelined. The head operator scans the base relation, CUSTOMER, and invokes the pipeline execution. Thus, we opted to track the execution progress along with CUSTOMER. We organized a single package for every 4096 CUSTOMER tuples. We performed the preliminary experiment for deciding the package size. Figure 8 presents the execution time that PhoeniQ took for the test query with different package sizes on eight c4.large instances. Note that failure was not injected. The query execution time increased as we chose package sizes of 1024 and less tuples. This was seemingly because of the communication overhead. As a result, we decided to use 4096 tuples per package because this configuration seemed sufficient to remove a majority of the communication overhead.

5.3 Total Query Execution Time

We experimentally tested the failure recoverability of PhoeniQ and its efficiency by emulating a single-node failure during query processing. We initially started the query processing by using eight processing nodes and an additional spare node. After t seconds passed, we terminated a certain processing node (PN#1). In the case of the conventional method, the database system discarded all the processing done so far and restarted the query processing from the beginning. In contrast, PhoeniQ immediately started the recovery process; specifically replacing the failed node with the spare node and taking over the lost processing on the spare node. We measured the total query execution time including the recovery time while changing t from 100 seconds to 900 seconds.

Figure 9 demonstrates the conventional method (baseline) incurred a significant time penalty if node failure happened at later points in time, whereas PhoeniQ offered negligible penalty consistently. Figure 9 (a) presents the case where eight c4.large instances were utilized for processing nodes, whereas 9 (b) presents the case where eight
Total query execution time. (a) Eight c4.large instances were utilized for processing nodes, whereas (b) eight m5.large instances were utilized similarly. The conventional method (baseline) incurred a significant time penalty if the node failure happened at later points in time, whereas PhoeniQ offered negligible penalty consistently.

Runtime resource utilization. Eight c4.large instances started query processing with an additional instance as a spare. PN#1 (fail) terminated the execution at the point of 900 seconds, but PN#1 (spare) immediately recovered the execution. PN and SN denote processing node and storage node respectively.

Query execution time and speed up ratio. One to 24 c4.large instances ran query processing according to the conventional method (baseline) and PhoeniQ. PhoeniQ did not disturb the node scalability.

Scalability Study

We conducted a scalability study on PhoeniQ. PhoeniQ imposes additional processing (e.g., package-level checkpointing and output replication) during query processing, which may produce performance overhead and degrade the node scalability.

Figure 11 demonstrates that PhoeniQ does not disturb the node scalability. Figure 11 (a) comparatively presents the query execution time by the conventional method and by PhoeniQ. We did this test with one to 24 processing nodes. Note that no failure event happened in this test, even though PhoeniQ conducted package-level checkpointing and output replication. PhoeniQ only presented negligible additional execution time in comparison with the conventional method consistently. Similarly, Fig. 11 (b) comparatively presents the speedup ratio. Again, PhoeniQ only presented negligible overhead.

In summary, this series of experiments clarified that PhoeniQ efficiently resumed query processing in the face of node failure, and PhoeniQ imposed negligible performance overhead due to checkpointing and replication.

6. Related Work

Early MapReduce[11] and its derivatives such as...
Hadoop [23] opted to intensively materialize operator outputs to the storage. This design allows query processing to easily recover from the latest persisted state, but it incurs significant I/O overheads [24]. A recent parallel execution engine Spark [25] offers a new programming interface to manage resilient distributed datasets (RDDs) [26] on the distributed memory space. Spark potentially offers fault tolerance with improved performance by avoiding the intensive materialization of intermediate data.

Fault-tolerant mechanisms for stream or data-flow processing systems were well studied [27]–[33]. They basically replicate computation and checkpoint the computation periodically to backup or spare nodes.

Chandramouli et al. and Chaudhuri et al. separately presented a checkpoint mechanism for query operators at runtime [34], [35]. Their aim is to allow the database user to suspend and resume the query processing, while PhoeniQ offers fault tolerance for query processing.

Hauglid et al. proposed PROQID, a partial query restart technique for the distributed database [36]. PROQID enables recovery from multi-site query failure and incurs only marginal extra work for the recovery. Their recovery mechanism assumes that all the tuple deliveries are deterministic in the distributed environment. In contrast, PhoeniQ does not request the determinism of tuple delivery.

Han et al. proposed OTPM for providing the dynamic query recoverability in parallel database systems [12]. OTPM is close to PhoeniQ in that query operators track the progress of their upstream operators by monitoring the identifiers of incoming tuples. They presented simulation-based evaluation, implying promising results. In contrast, PhoeniQ tracks the execution progress for every operator pipeline rather than every operator to reduce the checkpointing overhead, and replicates the operator output among different processing nodes to remove the necessity of dedicated tracking nodes. In addition, this paper presents the prototype-based experiment to clarify the effectiveness.

Recent work on parallel database systems studied the efficient materialization strategy in preparation of failure by considering the materialization cost. Upadhyaya et al. proposed a cost-based fault-tolerance optimizer that automatically selected additional work (e.g., whether to materialize the output to the storage or not) for each operator [37]. Chen et al. and Salama et al. separately proposed a cost-based optimization method for deciding operators, where the output was materialized for DAG-structured query plans [38], [39]. Ji et al. proposed the consideration of query success rates into the optimization of materialization strategies [40]. Zhu et al. proposed the application of machine learning prediction into the optimization process [41]. These work assumes that database systems materialize the intermediate results of selected operators mostly in the storage. In contrast, PhoeniQ takes a checkpoint of the execution state for every operator pipeline (i.e., at the last operator of the pipeline). It does not materialize the output of the operator to the storage, but it replicates the output to another processing node.

7. Conclusion

This paper has proposed PhoeniQ, a method for allowing parallel database systems to efficiently resume query processing after a partial failure event. PhoeniQ continuously takes a checkpoint for every operator pipeline and replicates the output of stateful operators among different processing nodes. If a single processing node fails during query processing, another can promptly take it over. We have prototyped a database system based on PhoeniQ and experimentally validated its effectiveness with up to twenty-four processing nodes plus a shared storage node in a public cloud environment. The validation test has confirmed that

- PhoeniQ resumes query processing in the face of node failure;
- PhoeniQ reduces a temporal penalty that used to be necessary for the conventional approach (restarting a query from scratch); and
- PhoeniQ imposes negligible performance overhead for checkpointing and replication.

As an early study, this paper has focused on introducing the idea of PhoeniQ, but has presented validation in a limited case. The present experiment only presumed a nested-loop join query, which induced the I/O bound situation. However, if we applied other types of queries, the performance bottleneck might occur at processors or network; different performance properties might be observed and they necessitate further careful design exploration. We would like to extend the test query variety to clarify the query space where PhoeniQ works effectively and efficiently. In addition, this paper only assumes that each operator pipeline running in processing nodes keeps the tuple processing order for simplicity. Removing this limitation is an open problem to apply PhoeniQ to other systems where the tuple order may not be kept. As described in Sect. 3.2.2, we are studying two ideas, appending additional tuple identification values and limiting the reordering opportunity. Hopefully, they only impose negligible performance overhead. We would like to carefully investigate their effects.

Acknowledgements

This work was in part supported by Impulsing Paradigm Change through Disruptive Technologies Program (IMPACT) by Cabinet Office, Japan and JSPS Grant-in-Aid for Scientific Research (B) JP20H04191.

References

[1] R. Shiftehfar, “Uber’s big data platform: 100+ petabytes with minute latency,” https://eng.uber.com/uber-big-data-platform/
[2] D. Weeks, “Netfliex: Integrating Spark at petabyte scale.” https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/43373
[3] J. 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/

[4] D. Borthakur, “Petabyte scale databases and storage systems at face-
book,” Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, ed. K.A. Ross, D. Srivastava, and D. Papadis, pp.1267–1268, ACM, 2013.

[5] D.J. DeWitt and J. Gray, “Parallel database systems: The future of high performance database systems,” Commun. ACM, vol.35, no.6, pp.85–98, 1992.

[6] G. Graefe, “Encapsulation of parallelism in the volcano query pro-
cessing system,” Proc. SIGMOD, vol.19, no.2, pp.102–111, ACM Press, 1990.

[7] K. Goda, Y. Hayamizu, H. Yamada, and M. Kitsuregawa, “Out-of-
order execution of database queries,” Proc. VLDB Endow., vol.13, no.12, pp.3489–3501, 2020.

[8] S. Ghandeharizadeh and D.J. DeWitt, “Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database storage,” Proc. VLDB 1990, pp.481–492, Morgan Kaufmann, 1990.

[9] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, “Prototyping bubbap, A highly parallel database system,” IEEE Trans. Knowl. Data Eng., vol.2, no.1, pp.4–24, 1990.

[10] T. Tamura, M. Oguchi, and M. Kitsuregawa, “Parallel database pro-
cessing on a 100 node PC cluster: Cases for decision support query processing and data mining,” Proc. SC 1997, p.49, ACM, 1997.

[11] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Proc. OSDI 2004, pp.137–150, USENIX Association, 2004.

[12] B. Han, E. Omiecinski, L. Mark, and L. Liu, “OTPM: failure han-
dling in data-intensive analytical processing,” Proc. CollaborateCom 2011, pp.35–44, ICST / IEEE, 2011.

[13] Y. Bessho, Y. Hayamizu, K. Goda, and M. Kitsuregawa, “Phoeniq: Failure-tolerant query processing in multi-node environments,” Proc. DEXA 2020, Lecture Notes in Computer Science, vol.12391, pp.71–85, Springer, 2020.

[14] M. Youssf, “Shared-storage clusters,” Clust. Comput., vol.2, no.4, pp.249–257, 1999.

[15] M. Stonebraker, “The case for shared nothing,” Proc. HPTS 1985, 1985.

[16] K. Goda, T. Tamura, M. Oguchi, and M. Kitsuregawa, “Run-time load balancing system on san-connnected PC cluster for dynamic in-
jection of CPU and disk resource —A case study of data mining application—,” Proc. DEXA 2002, Lecture Notes in Computer Science, vol.2453, pp.182–192, Springer, 2002.

[17] D.E. Shasha, Database Tuning - A Principled Approach, Prentice-
Hall, 1992.

[18] Transaction Processing Performance Council, “The TPC-H bench-
mark.” http://www.tpc.org/tpch/

[19] Oracle Corporation, “Oracle Berkeley DB.” https://www.oracle.com/database/berkeley-db/db.html

[20] F. Li, B. Wu, K. Yi, and Z. Zhao, “Wander join: Online aggregation for joins,” Proc. SIGMOD 2016, pp.2121–2124, ACM, 2016.

[21] D. Jiang, A.K.H. Tung, and G. Chen, “MAP-JOIN-REDUCE: to-
ward scalable and efficient data analysis on large clusters,” IEEE Trans. Knowl. Data Eng., vol.23, no.9, pp.1299–1311, 2011.

[22] D. Tsiorogiannis, S. Harizopoulos, M.A. Shah, J.L. Wiener, and G. Graefe, “Query processing techniques for solid state drives,” Proc. SIGMOD 2009, pp.59–72, ACM, 2009.

[23] Apache Software Foundation, “Apache Hadoop.” https://hadoop.apache.org/

[24] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” Proc. SIGMOD 2009, pp.165–178, ACM, 2009.

[25] Apache Software Foundation, “Apache Spark: Unified engine for large-scale data analytics.” https://spark.apache.org/

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCaulley, M.J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster comput-
ing,” Proc. NSDI 2012, pp.15–28, 2012.

[27] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S.B. Zdonik, “Monitoring streams - A new class of data management applications,” Proc. VLDB 2002, pp.215–226, Morgan Kaufmann, 2002.

[28] F. Khan, N. Akhtar, and M.A. Qadeer, “RFID enhancement in road traffic analysis by augmenting reciever with telegraphq,” Proc. WKDD 2009, pp.331–334, IEEE Computer Society, 2009.

[29] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S.B. Zdonik, “The design of the bcralils stream pro-
cessing engine,” Proc. CIDR 2005, pp.277–289, www.cidrdb.org, 2005.

[30] M.A. Shah, J.M. Hellerstein, and E. Brewer, “Highly-available, fault-tolerant, parallel dataflows,” Proc. SIGMOD 2004, pp.827–838, ACM, 2004.

[31] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. Zdonik, “A cooperative, self-configuring high-availability solution for stream processing,” Proc. ICDE 2007, pp.176–185, IEEE Computer Society, 2007.

[32] Y. Kwon, M. Balazinska, and A. Greenberg, “Fault-tolerant stream processing using a distributed, replicated file system,” Proc. VLDB Endow., vol.1, no.1, pp.574–585, 2008.

[33] A. Nguyen-Tuong, A.S. Grimshaw, and M. Hyett, “Exploiting data-
flow for fault-tolerance in a wide-area parallel system,” Proc. SRDS 1996, pp.2–11, IEEE Computer Society, 1996.

[34] B. Chandramouli, C.N. Bond, S. Babu, and J. Yang, “Query suspend and resume,” Proc. SIGMOD 2007, pp.557–568, ACM, 2007.

[35] S. Chaudhuri, R. Kaushik, R. Ramamurthy, and A. Pol, “Stop-and-
restart style execution for long running decision support queries,” Proc. VLDB 2007, pp.735–745, ACM, 2007.

[36] J.O. Haaglid and K. Nørvåg, “PROQID: partial restarts of queries in distributed databases,” Proc. CIKM 2008, pp.1251–1260, ACM, 2008.

[37] P. Upadhyaya, Y. Kwon, and M. Balazinska, “A latency and fault-
tolerance optimizer for online parallel query plans,” Proc. SIGMOD 2011, pp.241–252, ACM, 2011.

[38] T. Chen and K. Taura, “A selective checkpointing mechanism for query plans in a parallel database system,” Proc. IEEE Big Data 2013, pp.237–245, IEEE Computer Society, 2013.

[39] A. Salama, C. Binnig, T. Kraska, and E. Zamanian, “Cost-based fault-tolerance for parallel data processing,” Proc. SIGMOD 2015, pp.285–297, ACM, 2015.

[40] Y. Ji, Y. Chai, X. Zhou, L. Ren, and Y. Qin, “Smart intra-query fault tolerance for massive parallel processing databases,” Data Sci. Eng., vol.5, no.1, pp.65–79, 2020.

[41] Y. Zhu, M. Interlandi, A. Roy, K. Das, H. Patel, M. Bag, H. Sharma, and A. Jindal, “Phoebe: A learning-based checkpoint optimizer,” Proc. VLDB Endow., vol.14, no.11, pp.2505–2518, 2021.
Yuto Hayamizu is a project research associate at Institute of Industrial Science, The University of Tokyo. His research focuses on database systems and performance analysis of computer systems. He received his B.E. in electrical engineering, his M.E. in information and communication engineering and his Ph.D. in information science and technology from The University of Tokyo in 2009, 2011 and 2014 respectively. He is a member of IPSJ and DBSJ.

Kazuo Goda is an associate professor at Institute of Industrial Science, The University of Tokyo. He received his B.E. in electrical engineering, his M.E. in information and communication engineering and his Ph.D. in information science and technology from The University of Tokyo in 2000, 2002 and 2005 respectively. His research interests include database systems and storage systems. He has been awarded IEICE Best Paper Award, 21st Century Invention Award, and IPSJ Outstanding Paper Award. He is a member of IEICE, ACM, DBSJ, IEEE, IPSJ, and USENIX.

Masaru Kitsuregawa is Director General of National Institute of Informatics and a university professor at The University of Tokyo. He received Ph.D. degree from The University of Tokyo in 1983. He served in various positions such as President of Information Processing Society of Japan (2013–2015) and Chairman of Committee for Informatics, Science Council of Japan (2014–2016). He has wide research interests, especially in database engineering. He has received many awards including ACM SIGMOD E.F. Codd Innovations Award, IEICE Contribution Award, IPSJ Contribution Award, 21st Century Invention Award of National Commendation for Invention, Japan and C&C Prize, IEEE Innovation in Societal Infrastructure Award and Japan Academy Award. In 2013, he was awarded Medal with Purple Ribbon from Japanese Government, and in 2016, the Chevalier de la Legion D’Honneur. He is a fellow of ACM, IEICE and IPSJ, CCF honorary member, and IEEE Life fellow.