Instant Graph Query Recovery on Persistent Memory

Alexander Baumstark
alexander.baumstark@tu-ilmenau.de
TU Ilmenau
Germany

Muhammad Attahir Jibril
muhammad-attahir.jibril@tu-ilmenau.de
TU Ilmenau
Germany

Philipp Götze
philipp.goetze@tu-ilmenau.de
TU Ilmenau
Germany

Kai-Uwe Sattler
kus@tu-ilmenau.de
TU Ilmenau
Germany

ABSTRACT

Persistent memory (PMem) – also known as non-volatile memory (NVM) – offers new opportunities not only for the design of data structures and system architectures but also for failure recovery in databases. However, instant recovery can mean not only to bring the system up as fast as possible but also to continue long-running queries which have been interrupted by a system failure. In this work, we discuss how PMem can be utilized to implement query recovery for analytical graph queries. Furthermore, we investigate the trade-off between the overhead of managing the query state in PMem at query runtime as well as the recovery and restart costs.

ACM Reference Format:
Alexander Baumstark, Muhammad Attahir Jibril, Philipp Götze, and Kai-Uwe Sattler. 2021. Instant Graph Query Recovery on Persistent Memory. In *International Workshop on Data Management on New Hardware (DAMON'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3465998.3466011

1 INTRODUCTION

During the execution of long-running database queries such as analytical queries on large databases, there is a risk of system failures, e.g., due to system errors, concurrency conflicts, or insufficient resources. In the interest of guaranteeing the ACID properties, most DBMSs simply restart the entire query from the beginning after failure because managing the state of the queries and the intermediate results on external storage is expensive and slows down the actual query processing. However, with this approach, not only is the progress made by a query until the failure completely lost, but also expensive resources (computation time) are wasted. Nevertheless, with the recent advent of persistent memory (PMem), there are new opportunities to establish practical near-instant query recovery strategies in databases. But the other challenge still holds: maintaining intermediate results and query states in PMem. The higher latency that comes with the usage of PMem requires adapting a strategy with low overhead to not slow down the actual query processing.

This work is licensed under a Creative Commons Attribution International 4.0 License.

DAMON'21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8556-5/21/06.
https://doi.org/10.1145/3465998.3466011

In this paper, we investigate the trade-off between performance of query processing and benefits of query recovery. Query recovery requires that the query state and intermediate query results (snapshots) exist before the failure. Both the state and snapshots are used during recovery to continue the execution of the query. In addition, it offers benefits beyond query recovery such as reusing the intermediate results to answer other queries and updating query states to re-run only the remaining query parts, similar to materialized views.

We consider query recovery for analytical graph queries on our graph database engine Poseidon [1] which uses PMem as main storage. The main contributions of this paper can be summarized as follows: *(1)* We present a chunk-based snapshot approach in order to store intermediate results of query execution to PMem with low overhead. *(2)* We demonstrate a near-instant strategy for recovering a query after a failure, based on push-based and morsel-driven query processing.

Aside from [4], to our knowledge, there has been little work on query recovery using persistent memory so far. NVGraph [4] uses multi-version data structures to store snapshots of a node index table and an edge table, as well as versions of application-defined data and runtime states. It employs hybrid storage, where the most frequently accessed snapshots are kept in a snapshot cache in DRAM. Besides graph engines, some other storage architectures have been proposed for PMem. FOEDUS [2] uses dual pages where the most recent version of a page is in DRAM (if available) and periodic snapshots are created in PMem. SOFORT [5] is a columnar transactional storage engine leveraging PMem by minimizing logging and updating data in place. Primary data is placed on PMem while secondary in DRAM and recovered in case of failure.

2 GRAPH PROCESSING MODEL

Poseidon is a PMem-resident native graph database based on the property graph data model. It is designed to exploit the idiosyncrasies of PMem for accelerated graph processing. We realize the property graph data model via separate persistent tables of nodes, relationships, and property sets. A table is a linked-list of fixed-sized chunks. A chunk is an array of equally-sized object records of the same type - node, relationship, or property set. Chunks are cache-line aligned and multiples of 256 bytes for optimized sequential PMem access and PMem bandwidth utilization [1].

Node and relationship objects are stored as equally-sized records by outsourcing properties to separate tables. On top of that, property keys and variable-length property values are encoded using a persistent dictionary with bidirectional translation, effectively reducing expensive writes to PMem. Furthermore, equally-sized records allow for accessing records using their 8-byte integer offsets instead of 16-byte persistent pointers.

For efficient reclamation of deleted records, a bitmap is used in each chunk to mark empty and used slots. Allocations and deallocations are done at the chunk-level instead of record-wise in order to amortize the overhead of expensive PMem allocations. The chunks are linked by persistent pointers, forming a linked list. A scan over all nodes in a graph is achieved by traversing the linked chunks.

The byte-addressability of PMem is particularly useful for the access pattern of graph traversal operators. However, since PMem data access is no longer block-oriented, there is need to make optimizations for sequential data access. Moreover, PMem access latency is higher than that of DRAM. Hiding this higher latency requires efficient cache utilization, multithreaded processing, and various execution modes. To this end, we opted for a multithreaded push-based query processing model. Ranges of the node/relationship chunks are assigned to fine-grained task packages from a task pool. A worker thread pulls a task to execute the query on the assigned chunks. The execution starts with threads scanning their respective ranges of chunks, in the form of parallel scans. Thereafter, for each task, all subsequent operators following the scan are also performed within the task until a pipeline breaker like a join or sort operator is reached. Thus, we adopt the morsel-driven parallelism approach [3]. Our engine provides a set of graph-specific algebra operators such as NodeScan, RelationshipScan, ForeachRelationship, and Expand; as well as standard relational operators like Filter, Project, and several Join variants.

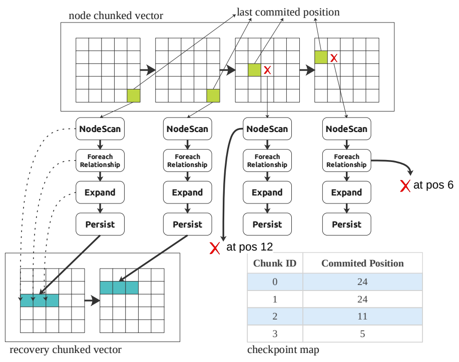
Processing a query starts with node (or relationship) table scan or node creation. Traversal queries (Match in Cypher), for example, are typically initiated by scans on the node tables followed by optional label and property filters. Each node matching the filter predicate is forwarded to the traversal (ForeachRelationship) operator to navigate their connected relationships. Subsequent operators are applied in a similar way.

3 QUERY RECOVERY

In order to recover a query after a complete system failure during the execution, the state of the query before the failure as well as the intermediate results produced until then must be restored. This requires that necessary information about the current state of the query processing, e.g., the current position of a scan, is stored during execution. Additionally, the results achieved up to a certain (check)point in the query pipeline must be persistently stored. The choice of this persisting of tuples is a trade-off between computation time and memory usage, which needs to be made appropriately.

3.1 Chunk-based Snapshots

For the actual storage of intermediate results, we implement a chunked-based data structure similar to the storage of graph data in Poseidon [1]. However, a modified design must be adopted here, since the length of intermediate result tuples varies, depending on the query in question. Therefore, we store each tuple element individually in the intermediate storage. For this purpose, we use a wrapper structure (cache-line-aligned and multiples of 256 bytes) that stores the actual elements and necessary information for processing, like the element type and tuple id. This way, we properly restore tuples from their constituent elements.



Chunk ID	Commited Position
0	24
1	24
2	11
3	5

Figure 1: Recovery storage model.

Storing a complete node or relationship object in the intermediate storage would consume too much memory. Because of this, we only store the node or relationship identifiers in the wrapper, i.e., the offsets of the objects in the node table or relationship table, respectively. For string values, we similarly only store the corresponding dictionary code. The raw value is stored in the wrapper for all other types. Thus we ensure that each tuple element in the PMem intermediate storage has the same length.

A sequential scan is sufficient to restore the intermediate results from PMem. For this, a map is used which maps a tuple id to its corresponding tuple. The tuple value in the wrapper element is converted according to its type, e.g., retrieving the node with a certain id from the node list, and then inserting it into the respective tuple in the map. The order of the tuples and their elements is maintained during recovery since they are inserted according to that order.

3.2 Query Processing

To store the intermediate results during the execution of the query, we make use of the push-based query engine of Poseidon. Since each operator pushes the processed tuple to the next operator, it is convenient to add another operator at a certain point in the pipeline which stores the intermediate results to PMem. For this purpose, we implement a Persist operator. The Persist operator stores the passing tuples to the persistent intermediate result storage and pushes them afterward to the next operator. Additionally, we implement special operators for Aggregations and Joins to store intermediate results that are not directly pushed to the next pipeline. The Aggregation operator buffers the tuples in a list in DRAM, as the results are coming in, before computing the aggregates. Only after the computation are the aggregated results pushed individually to the subsequent operator. It can be realized that if a failure is to occur at this point, the entire computation is lost. With a persistence-enabled Aggregation operator, the buffered tuples are rather maintained in the persistent intermediate storage. This handling allows persisting intermediate results at any point in the

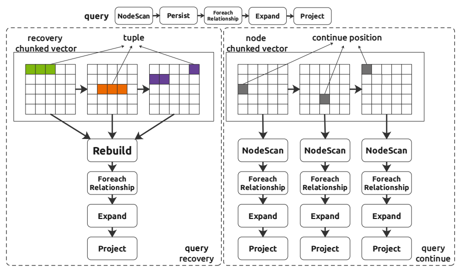


Figure 2: Query recovery procedure.

query pipeline with low overhead, as our evaluation in Section 4.2 has shown. Another requirement for query recovery is maintaining the current query execution state. Likewise, we take advantage of the push-based query processing for this. A graph query in Poseidon typically starts with a NodeScan, which sequentially scans the vector of node chunks in PMem.

When a position in the vector contains a node that matches a certain label or property predicate, it is pushed to the next operator until a pipeline-breaker is reached, which materializes the tuple. This completes the processing of the scanned position in the chunk vector and signifies a new query execution state. Internally, we traverse the chunk vector sequentially, managing checkpoints of the currently processed chunk (chunk id and pointer) and the current position of the object within the chunk. After the current position is processed, the checkpoints are stored to a persistent map data structure in PMem. An overview of the complete result persisting approach is shown in Figure 1.

3.3 Recovery

The recovery of a query after a failure consists mainly of restoring the intermediate results and the execution state before failure in order to continue to query. As we are storing the last processed position of the chunk vector in a persistent map, continuing the query is possible by starting from the checkpoints obtained from the persistent map. Here, we simply create the iterator objects for each not completed chunk using the last committed checkpoint as starting position. Using this approach allows for continuing the failed query while recovering the intermediate results for further processing, as illustrated in Figure 2.

The recovery of intermediate results is also optimized for PMem by scanning the intermediate storage sequentially, similar to the sequential scan of the NodeScan operator. Again, this has the advantage that we can utilize multithreading with the morsel-driven approach. An additional Rebuild operator is automatically set for the appropriate pipeline to restore the intermediate results from PMem and materialize them into a map structure in DRAM. The Rebuild operator is set at the same position as the Persist operator in the original query. With this, the query execution continues from the point where the tuples were persisted to PMem. Figure 2 shows an example of query rewriting for recovery. The scanned nodes are directly stored in the intermediate storage. After rewriting, the query processing continues with rebuilding the nodes and passing them to the same ForeachRelationship operator. This is necessary because the length of tuples is unknown and the individual elements of a tuple may not appear in direct order in the storage, due to multithreading during query execution (concurrent access from multiple Persist operators).

The Rebuild operator represents the point where the stored intermediate results are processed further, continuing from the point where they were persisted in the failed query instance. After the recovery of the tuples into an intermediate list, all tuples are forwarded individually to the next operator, following the push-based query processing model. At the same time as the rebuild, the query is also continued from the last committed checkpoints before failure. For this purpose, we implement a ContinueScan operator, which creates the iterator for each persisted checkpoint and continues the processing for each chunk from the position before the failure.

4 EVALUATION

In this section, we evaluate our recovery approach on selected queries. The goals of this evaluation are as follows:

(1) We evaluate the Persist operator on PMem and investigate the extent to which the additional overhead affects query processing.
(2) We demonstrate the effectiveness of our query recovery approach from failures occurring at different progress levels of query execution.

4.1 Environment & Workload

Our evaluations were carried out on a dual-socket Intel Xeon Gold 5215 with 10 cores per socket running at a maximum of 3.40 GHz, 384 GB DRAM, 1.5 TB Intel Optane DCPMM operating in AppDirect mode and 4x 1.0 TB Intel SSD DC P4501 Series connected via PCIe 3.1. The system runs on CentOS 7.9 with Linux Kernel 5.10.6. We use the Intel Persistent Memory Development Kit (PMDK) version 1.9.1 and libpmemobj-cpp version 1.11 for accessing the PMem device. Furthermore, we have created an ext4 filesystem on the PMem DIMMs, mounted with the DAX option to enable direct loads and stores.

For our workload, we selected the Linked Data Benchmark Council's Social Network Benchmark Business Intelligence (LDBC-SNB BI) [6] queries. This choice was informed by the nature of these queries, which makes them suitable for query recovery. However, we limit our evaluation to BI queries 1, 4, 9 & 16, which altogether are representative of the aspects of the BI queries mentioned above and also within the limits of our model. Therefore, they are sufficient for our evaluation purposes. We executed the queries on the synthetic yet realistic LDBC social network graph data at scale factor 10.

To simulate of execution failures, we place a special Crash operator that throws an exception when a given tuple count is reached. The query executor catches the exception and measures the elapsed time. This is especially useful for reproducibility of results.

4.2 Overhead of Persisting Results

We first study the effect of persisting intermediate results, using the Persist operator, on the total query execution time. We measured the time to execute the query without any intermediate result storage for recovery, and compared against two instances of the execution time with intermediate result storage at different points

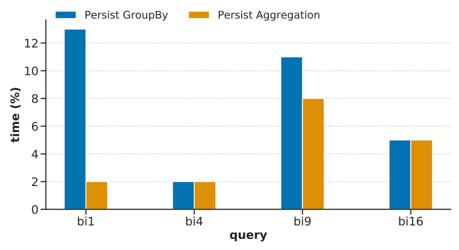


Figure 3: Overhead of persisting intermediate results.

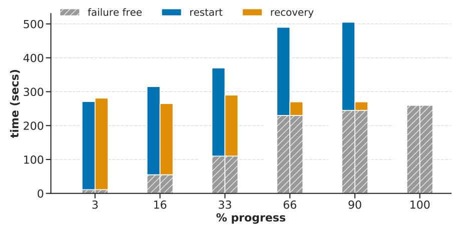


Figure 4: Recovery overhead of BI1 query at different progress.

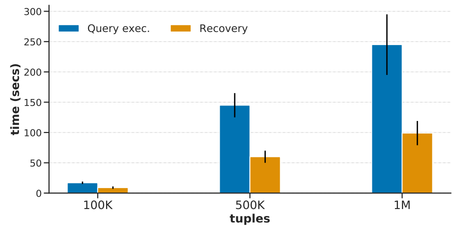


Figure 5: Recovery costs compared to restart query.

along the query pipeline – after the GroupBy and the Aggregation operators. This was done by appropriately positioning the Persist operator in the query pipeline. Although the overhead of persisting intermediate results increases with increasing number of tuples, persisting intermediate results in PMem at suitable points, in addition to our PMem-aware approach to storage, have only a small overhead, as shown in Figure 3.

Furthermore, since the occurrence of system failures is not common, it underscores the importance of placing the Persist operator at a suitable position in the pipeline, to minimize the storage overhead. Suitable positions are points with low tuple count like after Filter or Aggregation operators, or after compute-intensive operators like Joins.

4.3 Recovery Decision

Figure 5 compares the restart and recovery times for tuples with 5 elements. It shows clearly that recovering stored results is more than 2 times faster than obtaining the data from the graph again. Computation time can be saved when certain progress was achieved before failure. Therefore, we evaluate the recovery decision at different stages of query progress.

Figure 4 shows the execution of the BI1 query. The execution time shown for recovery includes both the actual time for intermediate result recovery and the continuing of the remaining parts. Intermediate results of the GroupBy operator are stored in PMem. Storing the intermediate results at this position is useful, as most of the computation time is spent aggregating the results. The percentage is the progress of the query execution before the system failure. Recovering intermediate results starts much more worthy from around 33% of execution progress before failure, when approximately 10 million tuples have been processed, with 20 million tuples remaining.

Considering the execution time for progress over 66% reveals that query recovery and continue is nearly instantly, i.e., recovery and query continue finished the remaining query parts in only a few seconds.

5 CONCLUSION

In this paper, we have presented our query recovery approach in the context of graph databases and persistent memory. We have discussed the additional necessary data structures and materialization steps in a query pipeline following the PMem-based design goals of our previous work. Our evaluation has shown that the materialization overhead is minimal when using PMem while significantly accelerating the failure recovery. The point at which the intermediate results are materialized and the progress level of the query execution before failure play a crucial role in the trade-off between overhead and benefits of recovery. The approach presented already provides a sound basis, which we will extend in future work to include a precise cost model and adaptable materialization granularities. Lastly, the advantage of persisting intermediate results to PMem is not limited to recovery after a system failure. The stored intermediate results can also be reused for the execution of other sub-queries and queries. We also leave this for future work.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) in the context of the project "Hybrid Transactional/Analytical Graph Processing in Modern Memory Hierarchies (#TAG)" (SA 782/28-2) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037) and by the Carl-Zeiss-Stiftung under the project "Memristive Materials for Neuromorphic Electronics (MemWerk)".

REFERENCES

[1] Muhammad Attahir Jibril, Alexander Baumstark, Philipp Götze, and Kai-Uwe Sattler. 2021. JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation. In *EDBT 2021*. 37–48. https://doi.org/10.5441/002/edbt.2021.05
[2] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 691–706. https://doi.org/10.1145/2723372.2746480
[3] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD 2014*. 743–754.
[4] Soklong Lim, Tyler Coy, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2020. NVGraph: Enforcing Crash Consistency of Evolving Network Analytics in NVMM Systems. *IEEE Trans. Parallel Distributed Syst.* 31, 6 (2020), 1255–1269. https://doi.org/10.1109/TPDS.2020.2965452
[5] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN 2014*. 8:1–8:7. https://doi.org/10.1145/2619228.2619236
[6] Gábor Szárnyas, Arnau Prat-Pérez, et al. 2018. An early look at the LDBC social network benchmark's business intelligence workload. In *SIGMOD 2018*. ACM, 9:1–9:11. https://doi.org/10.1145/3210259.3210268