Big data availability: Selective partial checkpointing for in-memory database queries

Daniel Playfair
SAP
daniel.playfair@sap.com

Amitabh Trehan
Queen's University Belfast
a.trehan@qub.ac.uk

Barry McLarnon
SAP
barry.mclarnon@sap.com

Dimitrios S. Nikolopoulos
Queen's University Belfast
d.nikolopoulos@qub.ac.uk

Abstract

Fault tolerance is an important challenge for supporting critical big data analytic operations. Most existing solutions only provide fault tolerant data replication, requiring failed queries to be restarted. This approach is insufficient for long-running time-sensitive analytic queries, due to lost query progress. Several solutions provide intra-query fault tolerance. However, these focus on distributed or row-oriented databases and are not suitable for use with the column-oriented in-memory databases increasingly used for high-performance workloads. We propose a new approach for intra-query checkpointing that produces an optimal checkpoint solution for a fixed checkpointing budget to minimise overhead on in-memory column-oriented database clusters. We describe a modified architecture for fault tolerant query execution using this approach. We present a general model for the problem, in which an adversary is free to terminate the execution of the query, eliminating all unsaved work. We present an algorithm that represents a first step towards producing checkpoint plans by optimally placing a single checkpoint. Our analysis shows this approach allows reduced checkpoint overheads while providing resilience for long-running queries.

1 Introduction

In-memory databases are high-performance databases that maintain all or a large proportion of data in main memory for their operation [1]. In-memory databases are usually column-oriented to leverage cache friendly memory access patterns. Such databases are increasingly used for big data analytics in industrial and scientific settings due to the performance benefits of reduced disk access overheads incurred with conventional databases [7]. Katal et al. identify fault tolerance as an important technical challenge for big data systems [10]. The fault tolerance approach of both conventional and in-memory database solutions largely focuses on various schemes for achieving the durability of the database through methods such as transaction logging and table snapshots [8]. In such solutions, fault tolerance of running queries relies on replicas of the database being available to allow a failed query to be restarted elsewhere. However, this approach results in all progress towards the result of such a query being lost, which can cause a significant increase in latency for long-running time-sensitive queries. Some approaches solve this by running multiple instances of important queries [2]. This increases the probability that some queries will run to completion at the cost of consuming more energy and requiring additional hardware.

While there are existing solutions that provide fault tolerance for running database queries [15, 12, 14, 4], these approaches are not suitable for use by in-memory column-store database systems either due to limitations in complexity of supported query plans or high overheads due to the large and varying memory profile of in-memory databases [3].

We propose a selective checkpointing mechanism towards providing efficient intra-query fault tolerance suited to in-memory database systems in a replicated cluster. Our approach leverages the intermediate data produced during the execution of a query plan and aims to produce checkpoint plans that more effectively use a fixed checkpointing budget than available alternatives. In-memory databases often fully materialise these intermediates, so no extra resources are consumed to produce this data. This approach allows all nodes within the cluster to be primarily focused on normal productive work and does not require any idle or non-productive backup nodes.

Our general model for reasoning about this problem consists of a bipartite directed acyclic graph (DAG) representing operations and data involved in parallel computations as well as providing a well defined order of execution. This model represents an approach that, unlike previous work, imposes no limitations on query plans, and is applicable to other models of computation. We also include an abridged version of this model that only considers the data dependencies for convenience of reasoning about checkpoint plans.

We envision that our mechanism will produce effective fault tolerance in many failure scenarios. In this paper, we take a first step towards implementing our mechanism by developing an algorithm for checkpoint planning. We believe our algorithm is the first to take a global approach to selecting an optimal single checkpoint from database query plan graphs. We analyse the algorithm, proving its correctness and complexity, and provide some preliminary results to its effectiveness compared to a naive checkpointing scheme. We suggest that our approach of effectively using a fixed checkpointing budget will reduce checkpoint overhead, as shown by the impact of a single optimally placed checkpoint.

The main contributions of our work are:

- A selective checkpointing mechanism for a replicated cluster of in-memory databases

- A novel general model for reasoning about query plans and producing checkpoint plans that treats all properties topologically
- An algorithm for selecting a single, optimal checkpoint from a query plan graph
- Theoretical analysis of our algorithm with proofs for its complexity and correctness

The layout of this paper is as follows: Section 2 describes related work and contrasts the state of the art solutions against our approach. Section 3 describes an existing system architecture and our proposed modifications (Section 3.1), a general model of computation that describes query plan graphs (Section 3.2), the execution of query plan graphs (Section 3.4) and the abridged model used by our algorithm (Section 3.3). Section 4 describes our algorithm, which is then analysed in Section 5, providing a proof of complexity and correctness. Section 6.1 describes the setup used to produce the preliminary results we discuss in Section 6.2. Finally, we give our conclusions in Section 7.

2 Related Work

As discussed in Section 1, most database fault tolerance approaches focus on achieving durability [8]. Fault tolerance for queries is generally provided by cluster-based solutions that facilitate parallel or repeat execution of failed queries on another node [2].

In solutions that employ complete re-execution in the case of failures, the performance of executing normal queries is important as improving this also impacts the repeated execution. However, these solutions are not sufficient for our specific scenario as we consider exceptionally long-running queries where trivial re-execution of failed queries is not suitable, even with performance improvements.

Ivanova et al. [9] describe an in-memory database solution for increasing the performance of TPC-H queries by reusing intermediate results produced during the execution of queries. This work demonstrates the power of using the materialised intermediate data featured in in-memory databases.

Li et al. [11] describe a method for denormalising database schemas that allows significant performance improvements for TPC-H queries over existing in-memory database solutions. This solution extends existing performance benefits of in-memory databases and shows that in many cases checkpointing is unnecessary since the cost of re-executing failed queries is insignificant.

We primarily consider solutions that provide intra-query fault tolerance, which can potentially address the issue of repeated effort by providing high-availability for long-running queries not provided by parallel or retry execution strategies. Reducing this repeated effort will enable faster execution of a query under failure conditions while using fewer resources than other strategies. This should provide a significant cost improvement to highly available big data solutions, as hardware can be more effectively utilised.

Remus [5] is a virtualisation-based solution that continuously checkpoints the CPU, I/O and main memory state of a virtual machine to another hypervisor. Upon failure of the original host, the virtual machine is automatically migrated and resumed from the last checkpoint. A database running on such a system would, as a result, inherit intra-query fault tolerance as all state is saved. However, this solution has some significant disadvantages. Firstly, a secondary host capable of storing the entire state of the protected virtual machine is required, which is a significant expense for in-memory databases each with large main memory. Additionally, the memory of this secondary host cannot be used productively as it has to remain idle to receive checkpoints. Finally, the overhead for a database system will be significant due to the quantity of memory modifications in the processing of queries that need to be checkpointed.

RemusDB [12] provides a solution to the significant checkpointing overhead of using Remus with database systems by modifying the database to become aware of the checkpointing. RemusDB flags memory pages used by the database that may not be checkpointed if they can be trivially restored later, reducing the checkpoint overhead by incurring a greater cost on restart to regenerate unsaved data. This mechanism is not able to differentiate between queries, resulting in greater overheads compared to checkpoint selection schemes as either all or no intermediate results must be checkpointed.

Osprey [15] describes a first step towards intra-query checkpointing, implementing a MapReduce-like [6] system for data warehouse databases. However, this system checkpoints every intermediate to disk for all subqueries, introducing a significant overhead. Additionally, the Osprey work only considers support for query plans that are trees operating within star schema data warehousing databases. This schema is restricted to business intelligence applications, and does not extend to general relational database applications.

FTOpt [14] develops a framework for selecting fault tolerance mechanisms for each operator within a query plan. This selectivity includes not saving or checkpointing the output of a particular operation, instead having re-execution of selected subqueries, thus removing the need to save potentially large intermediates. While this approach is more generally applicable than that of Osprey, it is still limited to tree-like query plans.

Chen and Taura [4] describe ParaLite, a distributed shared-nothing relational database system in which they implement intra-query fault tolerance using selective checkpointing employing a divide and conquer approach to decide whether an intermediate should be checkpointed. This approach relies on the assumption that a checkpoint decision for a particular node is only dependent on its successors. We suggest that this assumption does not hold for column-oriented database query plans that have a much greater sharing of intermediate results than row-oriented query plans. Leveraging this, we believe a global approach can achieve more efficient placement of checkpoints within a query plan.

3 Model

This section describes an existing architectural approach to database fault tolerance that uses a cluster of replicated database nodes, along with our proposed extension to this system. It additionally contains our models describing query plans and their execution.

3.1 System Architecture

3.1.1 Existing approach

A common existing approach consists of a cluster of read-only database nodes that each have a full copy of the data set. Multiple clients submit Online Analytical Processing (OLAP) queries to a load-balancing database middleware that assigns a node from a cluster. Once assigned to a node, a query is executed in parallel on this node, as would be the case if the database were used independently. The result is then returned to the client via the middleware. This approach is beneficial as it requires little to no modification of the underlying database implementation to support. An example of this approach is RAIDb [2], which provides this functionality to multiple databases engines through JDBC.

Should a failure occur, the middleware may decide to transparently retry the query on another node, or the client will be notified by the middleware and must resubmit it for execution. In either scenario, any progress towards the production of the result made by the previously selected node will be lost and the query is restarted from scratch.

3.1.2 Proposed extension

We propose augmenting the middleware and databases within the previously described approach with the capability to create and resume from partial checkpoints created from the execution of a query. This will address the problem of the increased query latency experienced in the case of a failure of a node in the cluster.

A database node executing a query will produce a checkpoint plan for selected query plans, which indicates a set of intermediate results created during the execution of the query that should be saved. Databases will execute queries as before but additionally save results as indicated by the checkpoint plan. These partial checkpoints will be made available to the middleware either through being directly transmitted over the network, or being saved to a persistent shared storage solution.

For in-memory databases, these checkpoints can be created by directly saving the materialised intermediate data structures that are generated during the normal execution of queries in such databases. This means that it is not necessary to explicitly materialise the intermediate results, reducing the potential performance impact of our solution. Ivanova et al. demonstrated successful use of the materialised intermediates created by in-memory databases to improve execution performance [9].

If the node executing the query should fail, another database node within the cluster will be automatically selected to finish the query by the middleware, based on its normal load-balancing strategies. This node will be sent the partial checkpoints, query plan, and initial parameters from the failed query and execution will resume and execute to completion on the replacement before returning the answer to the client via the middleware. Optionally, to protect against multiple failures, the checkpoint plan may also be transmitted and the new database node will continue to save new checkpoints as they are produced.

Incomplete checkpoints are considered invalid and are ignored. If a database node fails during the transmission of a checkpoint, this incomplete checkpoint will be discarded and the system will operate as if it was not transmitted.

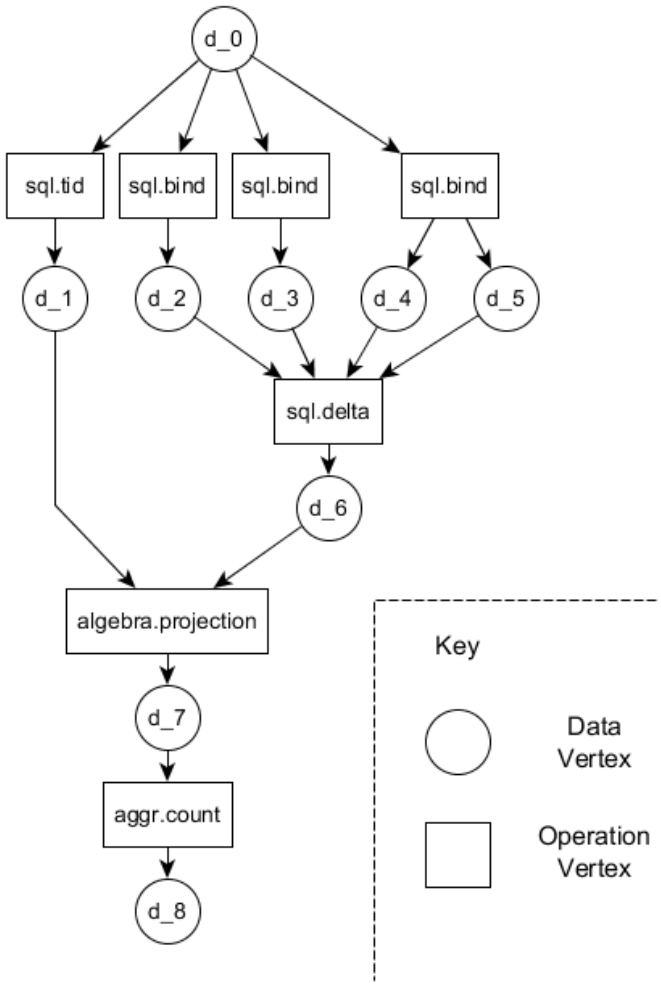

Figure 1: Example of query plan graph structure from MonetDB for query "`select count(*) from example;`" on a simple table with an integer primary key

A key requirement of this solution is the ability to select a subset of intermediate results from the query plan to be checkpointed. The selection of this subset should aim to limit the overhead of the checkpointing mechanism by selecting fewer, smaller intermediates where possible, but also to minimise the execution time of the restarted query. This selection is difficult as an approach potentially has to consider all possible subsets of vertices, as well as all possible failure positions of the query execution when using each candidate checkpoint subset.

3.2 Query Plans

A query plan describes the actions to be performed to produce the result for a given query in a database system. Query plans take different forms depending on the specific database implementation, but all can be modelled as DAGs.

Query plans are generated from SQL queries and optimised by one or more database query optimisers. Queries that will be reused can be declared as prepared statements. Query plans are usually cached for long periods of time and do not change. Automatic query optimisation may involve different goals, such as minimising run-time or achieving a consistent average case performance. Query plans may additionally be manually tweaked by database administrators using specific knowledge of workload and data distribution to achieve more desirable execution characteristics.

Generally a query can be described as being built from two primary components - a statement and parameters. The statement represents the general form of the query, and is used to construct the query plan. An example query plan graph is shown in Figure 1. The statement may use placeholders for variable parameters that will be provided at query execution time. Prepared statements enable reuse of the same query plan with different parameters. At run-time these placeholders are substituted with values provided from the query parameters provided to the prepared statement. In a query plan graph, these vertices, as well as vertices representing the data stored in the database itself, are used as the initial data vertices from which to begin execution of the query.

3.2.1 General Query Plans

Query plans (as well as many computations in general) can be represented as a bipartite directed acyclic graph, G. In these graphs, the set of vertices, V, are one of two types, V_{op} or V_{data}, representing either operations or data respectively.

Each vertex has a weight with different meaning depending on its type. The weight of operation vertices, $time(V_{\text{op}})$, represents the time required for the operation to be performed. The weight of data vertices, $size(V_{\text{data}})$, represents their size and the amount of memory they consume. This size is also proportional to the cost of checkpointing a particular vertex, as transmitting more data consumes additional resources.

The set of directed edges, E, in the graph represent the dependency relationship between the operations and data. Data vertices represent data items such as the original columns, query parameters, intermediate results, and the final result to be returned to the client. Operation vertices represent relational algebra operations performed on the data, such as selects or joins.

$$
\begin{aligned}
G &= (V, E) \\
V &= V_{\text{data}} \cup V_{\text{op}} \\
E \subseteq \{\ (v, w) \mid v, w \in V & \quad \wedge \\
v \in V_{\text{data}} &\implies w \in V_{\text{op}} \quad \wedge \\
w \in V_{\text{data}} &\implies v \in V_{\text{op}} \quad \}
\end{aligned}
$$

Each operation may have zero or more incoming data vertices, and may have zero or more output data vertices. Vertices with no ancestors are considered roots, and vertices with no descendants are considered leaves. There is a single root data vertex representing the database state and query parameters, and a single leaf data vertex representing the query result. During the execution of a query plan, an operation can only be performed if all its incoming data vertices have already been produced. This is described in more detail in Section 3.4.

An example query plan graph for a simple query is shown in Figure 1. This shows some of the operations a query will be built from, such as SQL catalogue access and low-level relational algebra functions performed on columns.

3.3 Abridged Query Plans

Section 3.2 describes a general model of computation that describes query plans. However, for the purposes of selecting data vertices within these query plans for checkpointing, a simpler graph representation composed of only the data vertices simplifies processing and reasoning.

This abridged query plan graph, G', is still a DAG but no longer bipartite, as operation vertices from the full query plan are omitted. The set of vertices in this graph, V', are instead data vertices from the original graph with edges, E', based on their indirect dependency relationships inherited from the original graph. This change allows algorithms to more conveniently consider the elements of the graph that can be checkpointed, and the relations between them.

The remaining vertices retain their original data vertex weights, representing only intermediate size. Edge weights are now used to represent execution time are derived from the weights of the operation vertices from which the edges are derived.

$$
\begin{aligned}
G' &= (V', E') \\
V' &= V_{\text{data}} \\
E' = \{\ (v, z) \mid v, z \in V_{\text{data}} & \quad \wedge \\
w \in V_{\text{op}} & \quad \wedge \\
(v, w), (w, z) \in E & \quad \wedge \\
time((v, z)) = time(w) & \quad \}
\end{aligned}
$$

3.4 Execution Model

For this work we consider the execution of a single query within the replicated cluster architecture described previously. We assume that queries are executed in parallel on a single database node (i.e. execution is not distributed across the cluster), selected by the load-balancing middleware. It is assumed that each database node has an infinite number of cores and that the time taken per operation and the size of an intermediate are both constant. When any data vertex that is marked to be checkpointed is produced during the execution of the query it will be asynchronously transmitted to the middleware.

To execute an operation, all of its dependencies must have been produced. As such, all data in the graph must be produced to calculate the result of the query. To avoid ambiguity as to when an operation is executed, we assume each operation in the query plan is performed as soon as all of its dependencies have been produced. Therefore, the execution time is equal to the weighted length of the *critical path*, as this will be the limiting factor in producing the final vertex.

Definition 1. *A **critical path** is a longest sequence of operations in the query plan. This is equivalent to a longest edge-weighted path in the abridged query plan graph.*

Adversary Strategy We consider the problem of crash failures of entire database nodes within our system during the execution of a single query. In such an event, the failure is detected by the middleware and a different database node is selected to re-execute the query. If any checkpoints were successfully created in the original execution, these are also transferred to the newly selected node.

During the execution of a single query in this model, an adversary may choose any point in the execution to trigger the failure of the node that it is executing on. We consider an adversary that, with full knowledge of the query and checkpoint plans, chooses to place a single failure to cause the worst-case execution time for the given query and checkpoint plan.

Execution time in the presence of a single failure The total execution time for a query in the presence of a single failure can be calculated as the sum of the time spent executing the query before the failure, and the time to re-execute the query to completion. If no checkpoint is available, the query must be re-executed in is entirety. The worst-case failure without a checkpoint would be to fail immediately before finishing, as this will cause the entire graph to be executed twice.

If a checkpoint is available, the re-execution time of the graph can be reduced. Operations dependent on a checkpointed data vertex may be able to execute sooner. Additionally, some operations and their intermediate results may no longer be necessary at all if they were only used in the production of the checkpoint. The execution time of the graph when using a checkpoint is equivalent to the execution time of a new graph where the checkpointed vertex is removed, along with any of its ancestors that are no longer reachable from the leaf of the graph.

Observation 1. *The total execution time with a single failure occurring before a checkpoint could be produced is equal to the sum of the amount of time spent executing the query before the failure and the uninterrupted execution time of the whole query.*

Observation 2. *The total execution time with a single failure occurring after a checkpoint was produced is equal to the sum of the amount of time spent executing the query before the failure and the uninterrupted execution time of the whole query when beginning with the checkpointed data.*

Returning to the example of the query plan in Figure 1, consider a failure after the *algebra.projection* operation with *d_6* being checkpointed. The re-execution of this query will only consist of three sequential operations producing the data vertices *d_1*, *d_7* and *d_8* (the query result). Note that *d_0* is still required to produce *d_1*, but *d_2*, *d_3*, *d_4* and *d_5* are no longer required as *d_6* was checkpointed and does not need to be reproduced.

4 Algorithm

To implement the checkpointing mechanism described in Section 3.1 we need a method of producing a checkpoint plan - a set of data vertices that should be saved during the execution of a query. As a first step towards this, we consider the production of checkpoint plans consisting of a single vertex to be checkpointed.

Definition 2. *A **single checkpoint** is a data vertex from a query plan to be saved during the execution of a query that will be the only vertex checkpointed and used during re-execution in the event of a failure.*

Definition 3. *An **effective single checkpoint**, v_{eff}, is any* single checkpoint *that will reduce the execution time of the query in the event of a single failure over simply re-running the query from the start. There is a (potentially empty) set, V_{eff}, of effective single checkpoints, which each have the same worst-case execution time.*

$$V'_{eff} = \{\ v_{eff} \in V' \mid time(v_{eff}) < distance(G', leaf(G')) \times 2\ \}$$

Algorithm 1 Single Checkpoint Selection Algorithm

```
1:  function CHECKPOINTSELECTION(G)
2:      best ← ∞
3:      candidates ← ∅

4:      P_c ← a critical path of G
5:      G_depth ← length of P_c
6:      l ← leaf vertex of P_c

7:      for v ∈ P_c do
8:          cost_before ← G_depth + LPATH(G, v, ∅)
9:          cost_after ← G_depth + LPATH(G, l, {v})

10:         cost ← MAX(cost_before, cost_after)
11:         if cost < best then
12:             best ← cost
13:             candidates ← v
14:         else if cost = best then
15:             candidates ← candidates ∪ v
16:         end if
17:     end for

18:     return candidates
19: end function
```

Definition 4. *An **optimal single checkpoint** is any* single checkpoint *that results in the smallest possible worst-case execution time of the query when used in the event of a single failure.*

$$V'_{opt} = \{\ v_{opt} \in V' \mid \neg\exists\ v \in V' \bullet time(v) < time(v_{opt})\ \}$$

The rest of this section describes our algorithm, which produces all optimal effective single checkpoint plans (if any exist) for arbitrary query plan graphs. To discuss the algorithm, we consider only the abridged query plan graphs described in Section 3.3.

Search candidate vertices The algorithm, summarised in Algorithm 1, consists of a search that calculates the worst-case execution time in the event of a single failure (the cost). A naive approach would calculate this cost for every vertex in the graph. However, this search space can be minimised by checking only vertices that lie within a critical path of the graph, as effective single checkpoints must belong to the critical path, as shown in Lemma 1. As such only the vertices within the critical path have the cost calculated.

We then calculate the cost for each candidate vertex tracking the set of vertices observed with the same minimum score. We make no attempt to further differentiate between single checkpoints that have an equal worst-case execution time, and one may be selected randomly from the set to use.

Cost calculation A naive approach for calculating the worst-case execution time for a particular checkpoint candidate would have to consider every vertex in the graph as a potential failure location. However, for plans consisting of a single checkpoint, the worst-case failure can only occur in two positions - directly before the checkpoint would be

Subroutine 1 Augmented longest path

```
1: function LPATH(G, x, exclusions)
2:     depths ← ∅
3:     for v ∈ topological ordering of v ∈ G do
4:         if v ∈ exclusions then
5:             depths[v] ← 0
6:         else if ∃ n ∈ V • (n, v) ∈ E then
7:             depths[v] ← MAX({depths[n]|(n, v) ∈ E}) +time(n, v)
8:         else
9:             depths[v] ← 0
10:        end if

11:        if v = x then
12:            return depths[v]
13:        end if
14:    end for
15: end function
```

saved or before completing the query, with the worst-case being the larger cost of failing in these two locations.

The cost of failing before creating a checkpoint is the sum of the cost of running the entire query once without any checkpoints (i.e. longest path to the leaf of the graph), and the amount of work that was lost due to the failure (i.e. the longest path to the checkpoint minus one).

The cost of failing before returning the result is the sum of the cost of running the entire query once without any checkpoints and the cost of re-running the query using the checkpoint, which must avoid computing vertices that are no longer required due to the checkpoint itself.

The calculation of these execution times requires calculating the longest weighted path from the root to a given vertex in the graph, for which there is a well known algorithm. This algorithm calculates the distance to vertices in the graph using a topological ordering, so that each vertex and edge need only be visited once. However, our problem additionally requires the calculation of the execution time that would result from using the checkpoint.

This can be calculated by transforming the graph to remove the checkpoint and its redundant ancestors, and then calculating the longest path as before. However, our longest path subroutine (Subroutine 1) extends the well known longest path algorithm to achieve this effect without requiring the transformation. This method retains the time complexity of the conventional algorithm. Our modification of the algorithm consists of the addition of lines 4 and 5, which checks if the vertex is a checkpoint and as a result should not be considered in the path length.

No optimal effective checkpoints If there exist no effective single checkpoints, all single checkpoints are then by definition optimal. In this case our algorithm will not return all optimal checkpoints, as it only searches the critical path, and so instead returns all vertices in the critical path. This satisfies the weaker guarantee that the algorithm will always return a non-empty subset of optimal checkpoints.

5 Analysis

In this section we prove the correctness and complexity of our single checkpoint algorithm.

5.1 Correctness

We aim to show that our algorithm will produce all optimal effective single checkpoints when any exist. If no effective checkpoints exist, it will return every vertex in the critical path. In this case we do not perform any checkpoint as there is no benefit in doing so.

Search As the algorithm only searches a single critical path for potential checkpoints we must first show that all potential effective checkpoints will lie within the searched vertices.

Lemma 1. *All effective single checkpoints lie on a critical path.*

Proof. The execution time of a query is defined by the length of the critical path of the query plan graph. An effective checkpoint will reduce the execution time of the query. If a single checkpoint is placed outside the critical path it will not reduce the execution time as all vertices on the critical path must still be produced. As such, an effective single checkpoint must be placed within a critical path. □

Lemma 2. *All effective single checkpoints lie within the intersection of the vertices on all critical paths.*

Proof. If a single checkpoint is placed in a critical path vertex that is not a member of all critical paths, then the execution time will not be reduced as the other critical paths must still have all their vertices produced. Therefore, only vertices that are on all critical paths will reduce the execution time when used as a single checkpoint. □

Corollary 1. *A graph with multiple critical paths and an empty intersection of vertices will have no effective single checkpoints.*

Lemma 3. *If any effective single checkpoints exist, all optimal single checkpoints will be a subset of the set of effective single checkpoints.*

Proof. Optimal single checkpoints are those that provide the best worst-case execution time of all potential single checkpoints. All non-effective single checkpoints will have a greater worst-case execution time than an effective one. □

Lemma 4. *All effective optimal single checkpoints lie within the intersection of the vertices on all critical paths.*

Proof. As per Lemma 2 and Lemma 3. □

Lemma 5. *Our algorithm considers all vertices that are potential candidates for effective single checkpoint.*

Proof. The assignment in line 4 and iteration in line 7 of Algorithm 1 considers all vertices on a single critical path. This will include all vertices that appear in any intersection of the critical path vertices, and as per Lemma 4, these are the only vertices that are candidates for being effective and optimal. □

This demonstrates that our algorithm considers all potential candidates that may form the set of effective optimal single checkpoints.

Cost calculation We must also show that our algorithm correctly calculates the worst-case execution time of the candidate vertices it considers.

Lemma 6. *The worst-case failure before a single checkpoint is produced will occur immediately before a checkpoint would be produced.*

Proof. The execution time will be maximised in a worst-case failure. In Observation 1 the total execution time of the graph from the start is fixed, so the amount of lost work must be maximised. This will occur at the furthest point from the root of the graph without taking a checkpoint. This point is maximised at the vertices before the single checkpoint. □

Lemma 7. *The worst-case failure that can occur after a single checkpoint is produced will be immediately before the query result is produced.*

Proof. The execution time will be maximised in a worst-case failure. In Observation 2 the total execution time of the graph from the start using the checkpoint is fixed, so the amount of lost work must be maximised. This will occur at the furthest point from the checkpoint of the graph. This point is maximised at last vertex in the graph. □

Lemma 8. *Subroutine 1 correctly calculates the execution time considering the use of a given checkpoint.*

Proof. The subroutine is an extension to the known algorithm for calculating the length of the longest path with the addition of lines 4 & 5. These lines alter the reported cost for vertices which are checkpoints. Due to the *max* operation on line 7, the overridden cost for a checkpointed vertex will only be considered if there are no alternative routes.

Due to the topologically ordered iteration of the conventional algorithm, these values will properly flow down to descendents as before. As such, this subroutine will report the longest path in a graph where the checkpoint and all its redundant ancestors are removed. □

Lemma 9. *Our algorithm correctly calculates the worst-case execution time of candidate vertices.*

Proof. A failure can only occur before or after a checkpoint is created. The algorithm calculates the execution time for the worst-case failures for both these cases (Algorithm 1 lines 8 & 9). It then uses the worst of these (Algorithm 1 line 10), and performs these calculations correctly (Lemmas 6, 7 & 8). □

Proof of correctness

Theorem 1. *Our algorithm returns all effective optimal checkpoints if any exist.*

Proof. The algorithm searches all potential effective optimal checkpoints (Lemma 5) and correctly calculates the worst-case execution time for these candidates (Lemma 9) and only stores and returns vertices whose cost is the smallest of the candidates checked (Algorithm 1 lines 11 to 18). □

5.2 Complexity

We show that the worst-case time complexity of our algorithm is $\mathcal{O}(n(n+m))$, where n is the number of vertices in the graph and m is the number of edges. This is based on the complexity of the outer algorithm function and the subroutine it calls.

Lemma 10. *The complexity of Subroutine 1 is* $\mathcal{O}(m+n)$.

Proof. The complexity of the subroutine is determined by the complexity of *max* operation finding the largest incoming vertex for every vertex in the graph, and the complexity of the iteration over each vertex.

The iteration in its worst-case considers every vertex in the graph, making it $\mathcal{O}(n)$. The *max* operation only considers the incoming edges to each vertex. This means that in the worst-case each edge is considered only once across the entire iteration, making this operation $\mathcal{O}(m)$. The overall complexity of the subroutine is the sum of these two complexities, $\mathcal{O}(n+m)$. □

Theorem 2. *The complexity of Algorithm 1 is* $\mathcal{O}(n(n+m))$.

Proof. The complexity defining operation in the algorithm is the iteration over the vertices in the critical path. In the worst-case, the critical path contains every vertex in the graph, making this operation $\mathcal{O}(n)$.

Inside this iteration, the most complex operation is the call to Subroutine 1, which has complexity $\mathcal{O}(n+m)$, making the overall complexity the product of these two complexities, which is $\mathcal{O}(n(n+m))$. □

6 Evaluation

In this section we discuss the potential implications and caveats of the application of our proposed mechanism its current form. We also describe the setup used to produce the simulated results discussed.

6.1 Profiling & Simulation Setup

To evaluate our approach and algorithms, we profiled the execution of queries from the TPC-H [13] benchmark on MonetDB. The profile additionally provides the query plan graphs used in the execution of the queries themselves. Output from this profiling is used to annotate the query plan graphs with the size and time parameters used in our models.

In this profiling and simulation we use queries from the TPC-H benchmark as they are easily accessible. These queries are not candidates for our checkpointing mechanism due to the insignificant cost of re-execution [9, 11]. These queries should not be considered examples of long running queries; however, they are suitable stand-ins to demonstrate the potential of our approach.

For profiling, we ran MonetDB (July 2015 SP2) on a machine with a Intel Core i5-2500 CPU, with 16 GB of RAM, using the TPC-H benchmark at scale factor 1. The queries are executed in a warm-up run before profiling to allow the database to load the relevant tables into memory. Each query is run individually with no other load on the system, and profiled using the MonetDB "stethoscope" profiling tool. This profile records intermediate data sizes execution times for each operation executed within the query plan.

We ran an implementation of our single checkpoint selection algorithm against the abridged graphs annotated with this profile. This produces a checkpoint plan containing the single optimal checkpoint for each query.

For comparison we also considered checkpoints produced by a naive checkpointing scheme that creates checkpoints, including all intermediate results that have been produced at the halfway mark of execution, but excluding any that are not required for future operations. This scheme guarantees that the maximum possible amount of work lost due to failure is 50% of the normal execution time of the query. Therefore the worst-case execution time with a single failure will be 150% of the normal execution of a single query. This provides a consistent 33% reduction in total worst-case execution time in the presence of a single failure; compared to having to re-execute the query from scratch, instead of 200% in the case of repeating it.

Using the profiled query plan and our execution model we also calculate:

- The worst-case execution time of the query using the re-execution strategy
- The worst-case execution time of the query using our single optimal checkpoints
- The size of the single optimal checkpoint
- The size of the naive checkpoint
- The worst-case execution time improvement of our single checkpoints vs query re-execution
- The effectiveness of our single checkpoints compared to the naive checkpointing scheme

The results of our profiling & simulation work is presented in Table 1. Queries 14, 16, 19 and 20 were omitted as they would not run correctly with the "stethoscope" profiling tool.

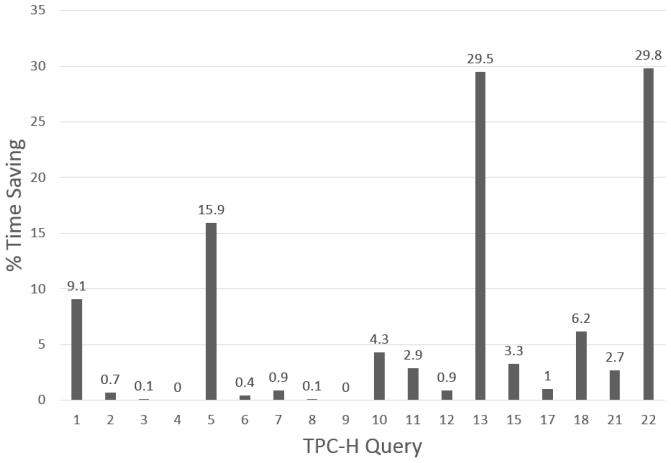

Figure 2: Predicted worst-case time saving for TPC-H queries with a optimal single checkpoint compared to query re-execution.

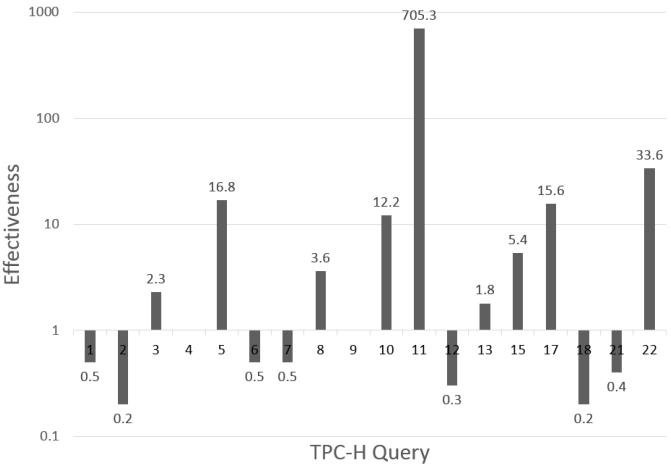

Figure 3: Effectiveness of single vs naive checkpoints.

6.2 Discussion

While our algorithm is only a first step towards selective checkpointing for intra-query fault tolerance, it provides some insights about the potential of our novel approach of using algorithms that place checkpoints considering the global topology of query plan graphs.

Figure 2 shows the percentage reduction in worst-case execution time predicted when using our checkpointing algorithm for a selection of TPC-H queries compared to the time taken to re-execute. We compare this to the naive checkpointing scheme that provides a 33% reduction in total execution time. Using only a single checkpoint we see that the execution time saving varies significantly depending on the query run. Some queries exhibit a reduction by nearly as much as 30%, while others are predicted to gain by less than 1%. Queries 13 and 22 show protection close to the 33% offered by the naive scheme, but consume less than 50% and 1% of the checkpoint size required by the naive scheme respectively. This shows that for a subset of queries, saving even just a single checkpoint can provide significant protection to queries. This is similar to the behaviour seen in Ivanova et al. [9] where their recycled intermediates only provide benefit to a subset of queries.

Figure 3 compares the effectiveness of the single checkpoint for each TPC-H query compared to the naive checkpoints. This calculation extrapolates the percentage saving provided for checkpoint size and compares the difference between the two schemes to provide the same level of protection. Values below 1 indicate that the single checkpoint is less effective than the naive scheme, while greater values indicate that the single checkpoint provided greater protection per unit space. This chart again shows that the behaviour differs for the different TPC-H queries. Queries 4 and 9 show no result as they provided little or no reduction in execution time. This identifies that these queries have no effective single checkpoints and require multiple checkpoints to provide coverage. In addition to the savings shown by Figure 2, this chart shows that while the single checkpoint for some queries did not provide a large reduction in execution time, the reduction that was provided was done so efficiently compared to the naive scheme. Query 11 only provided a 2.9% reduction in execution time, but was able to do it with a fraction of the intermediate data that the naive scheme was extrapolated to require.

This effectiveness evaluation is optimistic about the efficiency of the naive scheme, as it does not account for the additional overhead that would be required to save multiple intermediates. We can not assume the effectiveness of a single checkpoint can be directly extrapolated to coverage provided for a whole query. However, our results still suggest that the approach of more selectively checkpointing a small fixed number of vertices could be capable of reducing

Table 1: Table of simulation results

TPC-H Query	Repeat Query Time (μs)	Single Checkpoint Time (μs)	Single Checkpoint (% reduction)	Single Checkpoint) Size (MB)	Naive Checkpoint Size (MB)
1	5558090	5051882	9.1	28.2	56.4
2	258388	256650	0.7	0.8	9.2
3	707364	706751	0.1	0.1	96.7
4	979034	978770	0	0	122.5
5	1345364	1131621	15.9	4.3	153.0
6	596478	594369	0.4	4.2	210.1
7	2074210	2055670	0.9	18.4	329.9
8	810992	810138	0.1	0.2	218.2
9	12238168	12234414	0	10.9	29.9
10	840094	803552	4.3	0.4	41.0
11	171628	166574	2.9	0	30.8
12	828064	820831	0.9	25.9	293.6
13	10082486	7103963	29.5	14.6	29.3
15	300032	290238	3.3	1.1	59.5
17	1067854	1056876	1	0.2	118.1
18	22558	21153	6.2	28.6	28.6
21	2778552	2703561	2.7	42.9	208.9
22	226294	158949	29.8	0.1	2.3

checkpointing overheads while providing sufficient resilience against failure during long-running queries.

7 Conclusions

Existing approaches to providing intra-query fault tolerance suffer from significant overheads, being too restricted to apply to the general structure of query plan graphs used with in-memory column-oriented databases. Improvements in this field will contribute to providing practical high-availability solutions for big data applications.

Our model and approach are not restricted by schema or query plan complexity, and as such can be applied to any analytic database workload. Additionally, our model and approach can be used in other (non-database) parallel computing applications when a task graph is fully known in advance of execution.

Despite our single checkpoint selection algorithm only being a first step towards providing checkpoint planning, it shows promise for the approach of efficiently selecting checkpoints from a query plan graph based on a fixed budget due to the reductions in checkpoint overhead. This represents a novel approach compared to existing schemes, which can produce a variable number of checkpoints but rely on simplifications to achieve acceptable complexity in doing so.

Currently, our algorithm is limited to only selecting a single checkpoint, so necessary future work involves extending our algorithm to consider producing checkpoint plans containing multiple checkpoints. An optimistic extrapolation of our provisional results suggests that continuing our global optimisation approach may enable both greater execution time savings with smaller checkpointing overheads than existing solutions. Work is ongoing to implement our mechanism in MonetDB to provide an experimental evaluation and comparison of our approach.

References

[1] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] E. Cecchet. Raidb: Redundant array of inexpensive databases. In *Parallel and Distributed Processing and Applications*, pages 115–125. Springer, 2005.

[3] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 557–568. ACM, 2007.

[4] T. Chen and K. Taura. A selective checkpointing mechanism for query plans in a parallel database system. In *Big Data, 2013 IEEE International Conference on*, pages 237–245. IEEE, 2013.

[5] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[8] I. Gashi, P. Popov, and L. Strigini. Fault tolerance via diversity for off-the-shelf products: A study with sql database servers. *Dependable and Secure Computing, IEEE Transactions on*, 4(4):280–294, 2007.

[9] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. *ACM Transactions on Database Systems (TODS)*, 35(4):24, 2010.

[10] A. Katal, M. Wazid, and R. Goudar. Big data: issues, challenges, tools and good practices. In *Contemporary Computing (IC3), 2013 Sixth International Conference on*, pages 404–409. IEEE, 2013.

[11] Y. Li and J. M. Patel. Widetable: an accelerator for analytical data processing. *Proceedings of the VLDB Endowment*, 7(10):907–918, 2014.

[12] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: Transparent high availability for database systems. *The VLDB Journal*, 22(1):29–45, 2013.

[13] Transaction Processing Performance Council. Tpc-h benchmark specification. *Published at http://www.tpc.org/tpch/*, 2008.

[14] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 241–252. ACM, 2011.

[15] C. Yang, C. Yen, C. Tan, and S. R. Madden. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 657–668. IEEE, 2010.