Cost-based Fault-tolerance for Parallel Data Processing

Abdallah Salama* Carsten Binnig*† Tim Kraska† Erfan Zamanian†

*Baden-Wuerttemberg Cooperative State University
Mannheim, Germany

† Brown University
Providence, USA

ABSTRACT

In order to deal with mid-query failures in parallel data engines (PDEs), different fault-tolerance schemes are implemented today: (1) fault-tolerance in parallel databases is typically implemented in a coarse-grained manner by restarting a query completely when a mid-query failure occurs, and (2) modern MapReduce-style PDEs implement a fine-grained fault-tolerance scheme, which either materializes intermediate results or implements a lineage model to recover from mid-query failures. However, neither of these schemes can efficiently handle mixed workloads with both short running interactive queries as well as long running batch queries nor do these schemes efficiently support a wide range of different cluster setups which vary in cluster size and other parameters such as the mean time between failures. In this paper, we present a novel cost-based fault-tolerance scheme which tackles this issue. Compared to the existing schemes, our scheme selects a subset of intermediates to be materialized such that the total query runtime is minimized under mid-query failures. Our experiments show that our cost-based fault-tolerance scheme outperforms all existing strategies and always selects the sweet spot for short- and long running queries as well as for different cluster setups.

1. INTRODUCTION

Motivation: Parallel Data Processing Engines (PDEs) such as parallel databases (e.g., SAP HANA [7], Greenplum [20] or Teradata [15]) or other modern parallel data management platforms (e.g., Hadoop [21], Scope [25], Spark [24]) are used today to analyze large amounts of data in clusters of shared-nothing machines. While traditional parallel database systems have been designed to run in small clusters with highly available hardware components, modern PDEs are often deployed on large clusters of commodity machines where mid-query failures are a more common case.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749437.

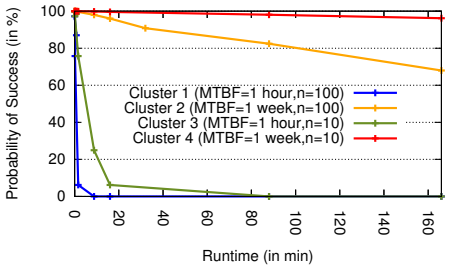


Figure 1: Probability of Success of a Query

In order to deal with mid-query failures the before mentioned systems implement different strategies: (1) fault-tolerance in parallel databases is implemented in a coarse-grained manner by restarting a query completely when a mid-query failure occurs, whereas (2) modern MapReduce-style PDEs implement a more fine-grained fault-tolerance scheme, which either materializes each intermediate result (e.g., Hadoop) or use lineage information to be able to recover sub-plans from mid-query failures (e.g., Spark).

However, none of the before mentioned fault-tolerance schemes can efficiently handle analytical workloads which consist of a mix of queries with a strongly varying runtime ranging from seconds to multiple hours as commonly found in real deployments [16]. For example, while short running queries in Hadoop typically suffer from high materialization costs, long-running queries in parallel databases need to pay high recovery costs when a mid-query failure occurs right before the query is about to finish its execution. Moreover, other parameters such as the size of the cluster (or more precisely the number of nodes typically participating in executing a query) and the mean time between failures (MTBF) also have a strong influence on the efficiency of a fault-tolerance scheme which is not reflected by existing schemes.

Figure 1 shows the relationship between the query runtime and the probability that no mid-query failure is occurring while executing the query in different cluster setups varying in the cluster size (i.e., number of nodes) and the MTBF per cluster node.[1]

[1]In this paper, we assume exponential arrival times between failures and accordingly model the probability of having n-failures in time interval t as a poisson process. That is, assuming that a query is executed on n nodes with independent failure rates, the likelihood of at least one failure within the cluster is given as $P(N_t^n > 0) = 1 - P(N_t^1 = 0)^n = 1 - e^{\frac{-tn}{MTBF}}$ where N_t^n is the number of failures in time interval t on n servers.

Cluster 1 in this figure represents a cluster setup with a large number of nodes and a low MTBF per node (which can typically be found in IaaS offerings such Amazon's Spot Instances). For this type of cluster setup we can see that even short-running queries already have a very low probability to succeed. Thus, materializing intermediates to recover from mid-query failures would be beneficial in this setup. Cluster 4 represents a cluster setup with only a few nodes and a much higher MTBF than cluster 1. In this cluster setup materializing intermediates for queries results in unnecessary execution costs since the probability that queries in this setup succeed is always very high. However, for the other two cluster setups (i.e., cluster 2 and 3) the probability that a query succeeds in one attempt strongly depends on the query runtime.

Contributions: We present a novel cost-based fault-tolerance scheme. Compared to the existing fault-tolerance schemes discussed before, our scheme selects a subset of intermediates to be materialized (further referred to as *materialization configuration*) such that the query runtime is minimized under the presence of mid-query failures. In order to select which intermediates should be materialized, we present a cost model, which helps finding an optimal materialization configuration for a given query using statistics about the query and the cluster (such as the cluster size and the MTBF). Moreover, we present efficient pruning techniques to reduce the search space of potential materialization configurations enumerated by our cost model. We integrated our approach as well as the existing fault-tolerance schemes in an existing open-source parallel database called *XDB* [8] and executed a comprehensive experimental evaluation. In our evaluation, we show that our cost-based fault-tolerance scheme can effectively deal with different kinds of queries (i.e., varying runtime and costs for materialization) as well as with different cluster setups and thus outperforms all existing strategies.

Outline: First, in Section 2 we discuss the assumptions of our cost-based fault-tolerance scheme. In Section 3, we present our cost-based fault-tolerance scheme in detail: We show how our cost-based fault-tolerance scheme enumerates different materialization configurations for a given query and how this enumeration can be integrated into existing cost-based optimizers. Moreover, we discuss details of how our cost model estimates the runtime costs of a query under mid-query failures in order to find the optimal materialization configuration. Afterwards, in Section 4, we discuss pruning techniques to reduce the search space of the enumeration process. Our comprehensive experimental evaluation with the TPC-H benchmark is then discussed in Section 5. Finally, we conclude with related work in Section 6 and a summary in Section 7.

2. ASSUMPTIONS

In this section, we first discuss our assumptions regarding the parallel execution model. Secondly, we discuss which types of failures our cost-based fault-tolerance scheme can handle effectively.

2.1 Parallel Execution Model

Based on the typical design of Parallel Data Processing Engines (PDEs) we assume that the parallel execution plan P is represented as a directed acyclic graph (DAG) and that

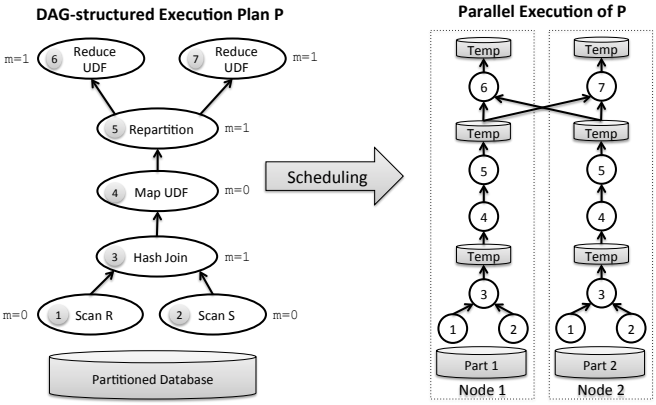


Figure 2: Parallel Execution Model

the data is horizontally partitioned over multiple nodes of a shared-nothing cluster. We further assume that one or more operators $o \in P$ are executed in parallel over partitions on individual nodes Figure 2 shows an example of a plan and its partition-parallel execution using two nodes.

In order to apply our cost-based fault-tolerance scheme in a PDE, we require that for each operator $o \in P$ there exists a property $m(o)$ which defines whether the output of o should be materialized (i.e., $m(o) = 1$) or not (i.e., $m(o) = 0$) during execution. The set of all properties $m(o)$ for each $o \in P$ is called materialization configuration M_P for plan P. For example, for the plan in Figure 2 the output of the operators 3 and 5, 6 7 is set to be materialized.

For operators $o \in P$ with $m(o) = 1$, we assume that these are blocking operators (i.e., these operators materialize their output completely before the consumer starts its execution). For operators $o \in P$ where $m(o) = 0$, we assume that the PDE uses its standard input/output behavior such as pipelining data to the consuming operator. That way our cost-based fault-tolerance scheme can be easily integrated into many existing PDEs by simply materializing the output of sub-plans.

Moreover, our cost-based fault-tolerance scheme supports DAG-structured execution plans which contain arbitrary operators $o \in P$ (i.e., UDFs as well as standard relational operators) as long as the following estimates can be provided for each operator as input to our cost model: the runtime cost $t_r(o)$ for executing an operator o and the materialization cost $t_m(o)$ for materializing its output to a given storage medium. Both $t_r(o)$ and $t_m(o)$ are given for partition parallel execution (i.e., an operator is executed in parallel over all partitions) Typically, these estimates are calculated based on input/output cardinalities of each operator [14].

Finally, in order to support certain platform specific properties of individual PDEs, operators can be marked as non-materializable (i.e., $m(o)$ is constantly set to 0) or it can be marked as always-materialized (i.e., $m(o)$ is constantly set to 1) before our cost model is applied. For example, some PDEs always materialize the output of a repartitioning-operator. In this case, these operators will be marked as always-materialized. Operators that are either marked as non-materializable or as always-materialized are called bound operators, denoted as $f(o) = 0$. All other operators are called free operators, denoted as $f(o) = 1$. All bound operators are excluded by the enumeration process our cost-based fault-tolerance scheme.

Notation	Description
P	DAG-structured execution plan P.
M_P	Materialization configuration for execution plan P.
o	Operator $o \in P$.
$m(o)$	Indicates if an operator $o \in P$ should be materialized (i.e., $m(o) = 1$) or not (i.e., $m(o) = 0$).
$f(o)$	Indicates if an operator $o \in P$ is free and optimizable ($f(o) = 1$) or bound ($f(o) = 0$).
$t_r(o)$	The estimated accumulated execution cost of an operator $o \in P$.
$t_m(o)$	The estimated accumulated materialization cost of an operator $o \in P$.
$t(o)$	The total accumulated runtime cost of an operator $o \in P$; i.e. $t(o) = t_r(o) + t_m(o) \cdot m(o)$.
P^c	Collapsed plan resulting from P and M_P.
c	Collapsed operator $c \in P^c$.
Pt	Path from a source to a sink in P^c.
R_{Pt}	Total execution cost of a path Pt without recovery costs for mid-query failures.
T_{Pt}	Total execution cost of a path Pt with recovery costs for mid-query failures.
$CONST_x$	An internal constant x used by our cost model.

Table 1: Terminology and Description

2.2 Failure Model

As other work [18], we assume exponential arrival times between failures and that failures are independent. Furthermore, our cost-based scheme is designed to handle process and node failures in clusters of shared-nothing machines. In order to give accurate estimates for the expected total runtime in the presence of mid-query failures, our cost model depends on the assumption that intermediates are not lost if a mid-query failure occurs; i.e., our cost-based scheme assumes that we can restart a query always from the last successfully materialized intermediate result after a given mean time to repair (MTTR). Consequently, if all intermediates are materialized to a separate fault-tolerant storage medium (e.g., Hadoop materializes its results to HDFS), our cost model will be accurate since intermediates are not lost in case of a mid-query failure. In case intermediate results can be lost due to a mid-query failure, our cost model which estimates the runtime of a query under failure will be too optimistic. For example, if intermediate results are stored locally per node to main memory, our cost model will not be accurate since failures lead to a loss of intermediates. As a future avenue of work, we plan to extend our cost model to also be accurate in these cases.

It is important to note, that the assumption that intermediates are not lost by mid-query failures, does not limit the applicability of our cost-based strategy in PDEs. For example, lineage information [24] makes it possible to reconstruct lost intermediate results of individual nodes without re-executing the whole query plan. Since typically only one or two nodes fail at the same time our cost-based fault-tolerance scheme is still applicable though slightly optimistic. For the rest of the paper, we assume that intermediate results are not lost due to a mid-query failure.

3. COST-BASED FAULT-TOLERANCE

In this section, we first give an overview of our cost-based fault-tolerance scheme and then explain the individual steps in details.

3.1 Overview

The main goal of our cost-based fault-tolerance scheme to find an execution plan P and a materialization configuration M_P is for a given query such that the total runtime of that query under mid-query failures is minimized. Moreover, our cost-based fault-tolerance scheme is a fine-grained strategy which restarts only sub-plans on nodes that actually failed.

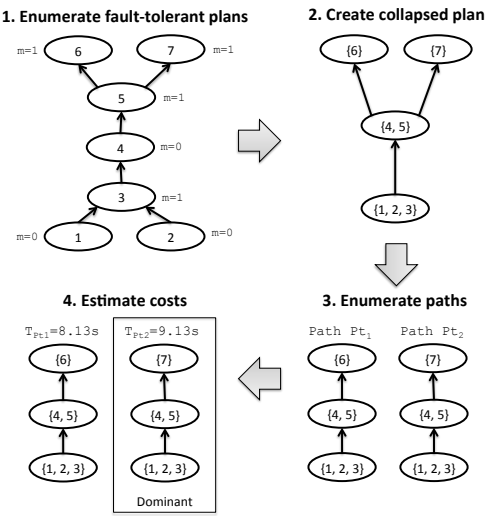


Figure 3: Steps of our Procedure

In order to find an optimal plan, cost-based optimizers of PDEs typically enumerate different equivalent execution plans P and apply a cost function to find the best plan with the minimal runtime cost. However, neither do they enumerate different materialization configurations for fault-tolerance nor do they consider the costs for recovering from mid-query failures. Therefore, we propose to change the cost-based optimizer to use an enumeration procedure that finds the best combination of a plan P and a materialization configuration M_P to minimize the overall run-time under the previously described failure model. We call this combination $[P, M_P]$ a fault-tolerant plan.

In the following, we give a high-level description of our procedure $findBestFTPlan(Q)$, which finds the best fault-tolerant plan for a given query (see Figure 3): (1) First, our procedure enumerates different fault-tolerant plans $[P, M_P]$ for the given query Q. (2) Second, for each enumerated fault-tolerant plan, our procedure creates a collapsed plan P^c where all operators in M_P that do not materialize their output are collapsed into the next materializing operator(s). The idea of the collapsed plan is to represent the granularity of re-execution using these collapsed operators (i.e., if a collapsed operator has finished successfully it does not need to be re-executed again). (3) Third, for a collapsed plan, all execution paths $Pt \in P^c$ (i.e., paths from each source to each sink in P^c) are enumerated and (4) the total cost T_{Pt} is estimated for each execution path Pt using a cost function that takes statistics about the operators and the cluster (e.g., the mean time between failures) into account. The path Pt with the maximal estimated total cost for a given materialization configuration is marked as dominant path of the fault-tolerant plan $[P, M_P]$. The dominant path is a good representative for the total runtime of the fault-tolerant plan under failures.[2] In Figure 3, for example, path Pt_2 is marked as the

[2]To estimate the cost we do not use the average expected cost but a more pessimistic estimate using percentiles as described in Section 3.5, which simplifies the model and allows to avoid complex models involving the maximum over stochastic variables.

dominant path and thus the estimated runtime for the given fault-tolerant plan $[P, M_P]$ is the runtime of this path under mid-query failures. Finally, our procedure selects that fault-tolerant plan, which has the shortest dominant path among all enumerated fault-tolerant plans.

Listing 1 shows the pseudo code of our procedure to implement the before mentioned steps. The input is a given query Q and the output is a fault-tolerant plan $[P, M_P]$. In order to estimate the total cost of a path Pt under mid-query failures, the cost function *estimateCost* requires that the following statistics are given: The runtime costs $t_r(o)$ to execute each operator $o \in P$ and the costs $t_m(o)$ to materialize the output of each operator $o \in P$. Both cost values can be derived from cardinality estimates that are calculated by a cost-based optimizer. Moreover, other parameters that are needed for the cost estimation are the following cluster statistics: the mean time between failures (MTBF) and the mean time to repair (MTTR) for one cluster node. In this paper, we assume that all these parameters are given by the function call *getCostStats* in line 6 of Listing 1.

In the following sections, we present the details for each of the before mentioned steps (1-4) and explain the pseudo code in detail. Table 1 summarizes the most important terminology used in the remainder of this paper.

Listing 1: Find Best Fault-tolerant Plan

```
1  function findBestFTPlan(Query  Q){
2    Plan bestP = null;
3    Mat.Conf. bestM = null;
4    int bestT = MAX_INTEGER;
5    for(each [P, M_P] in enumFTPlans(Q)){
6      cost stats = getCostStats(P);
7      int domT_Pt = 0;
8      Plan P^c = collapsePlan(P, M_P, stats);
9      for(each path Pt in P^c){
10       int T_Pt = estimateCost(Pt, stats);
11       if(T_Pt > domT_Pt) //dominant path
12          domT_Pt = T_Pt;
13     }
14     if(domT_Pt<bestT){ //store if dom. path is shorter
15       bestT = domT_Pt;
16       bestP = P
17       bestM = M_P
18     }
19   }
20   return [bestP, bestM];
21 }
```

3.2 Step 1: Enumerating Fault-tolerant Plans

Our procedure in Listing 1 enumerates potential fault-tolerant plans $[P, M_P]$ for a given query Q using the function *enumFTPlans* (line 5 in Listing 1). A naive implementation of this function would first enumerate all plan and then for each plan use exhaustive search to explore the 2^n variations n being the number of free operators in P to find the optimal plan.

The problem complexity of enumerating all join orders in DAG-structured plans is already $\mathcal{NP}$-hard [14]. Therefore, we use an approximate algorithm to implement the enumeration function *enumFTPlans*. In the first phase, this function uses dynamic programming to find the top-k plans (produced by the last iteration) ordered ascending by their cost without mid-query failures. In the second phase, function *enumFTPlans* then enumerates all potential materialization configurations for these plans. The intuition to analyze the top-k plans is that a plan P that has slightly higher costs than a plan P' in the first phase, can have lower costs when including the costs to recover from mid-query failures. For example, a plan P that has an operator o with low materialization costs (i.e., $t_m(o)$ is small) at a position in a plan right before a failure is likely to happen (based on the given MTBF) will "waste" much less time to recover than a plan P' which does not have this property.

However, enumerating all materialization configurations for the top-k plans can still be very expensive since the search space for potential materialization configurations is growing exponentially with the number of free operators in a plan. Though an interesting observation is, that the top-k plans might have a lot of paths in common. Based on that observation we present techniques to prune the search space in Section 4.

3.3 Step 2: Creating a Collapsed Plan

In its second step, our procedure creates a collapsed plan P^c for the given fault-tolerant plan $[P, M_P]$ by calling the function *collapsePlan* (line 8 in Listing 1). As mentioned before, the collapsed plan is the basis to estimate the total cost of a given fault-tolerant plan $[P, M_P]$ including the cost to recover from mid-query failures.

In order to construct the collapsed plan P^c from the given plan P and the materialization configuration M_P, all operators $o \in P$ that are defined by M_P to be not materialized are collapsed into the next consuming operators in the DAG-structured plan, which materialize their output. In Figure 3, we show the collapsed plan P^c (step 2) for the given fault-tolerant plan $[P, M_P]$ (step 1). To put it differently, a collapsed operator $c \in P^c$ represents a sub-plan of P that, once it has materialized its output, does not need to be re-executed again if a mid-query failure occurs. The set of operators of P collapsed into one operator $c \in P^c$ is denoted by $coll(c)$.

Moreover, for each collapsed operator $c \in P^c$, function *collapsePlan* also calculates the runtime (without costs for mid-query failures). The runtime of a collapsed operator c is defined as $t(c) = t_r(c) + t_m(c)$ (i.e., runtime costs plus materialization costs). This runtime is used in step 4 (see Section 3.5) to estimate the runtime of the collapsed plan P^c under mid-query failures. In the following, we show how each component is calculated.

The runtime costs $t_r(c)$ of a collapsed operator are determined by the longest execution path in $coll(c)$ called the dominant path $dom(c)$ of a collapsed operator c. For example, in Figure 3 (step 2) the dominant path $dom(\{1,2,3\})$ of the collapsed operator $\{1,2,3\}$ is represented by the two operators $\{2,3\}$ if $t_r(2) \geq t_r(1)$ holds. The runtime costs $t_r(c)$ are thus calculated as shown by Equation 1. For example, the execution cost of the collapsed operator $\{1,2,3\}$ in Figure 3 (step 2) is $t_r(\{1,2,3\}) = (t_r(2) + t_r(3)) \cdot CONST_{pipe}$ if $t_r(2) \geq t_r(1)$ holds. The constant $CONST_{pipe}$ with a value in the interval $(0,1]$ is used to reflect the effects of pipeline parallelism in the sub-plan represented by a collapsed operator c.[3]

[3]Since the constant $CONST_{pipe}$ strongly depends on the execution strategy implemented by the PDE and the underlying hardware, it must be derived individually by calibration experiments.

$$t_r(c) = \left(\sum_{o \in dom(c)} t_r(o) \right) \cdot CONST_{pipe} \quad (1)$$

The materialization costs $t_m(c)$ of a collapsed operator $c \in P^c$ are the materialization costs of the final operator in the longest path. For example, in Figure 3 (step 2) $t_m(\{1,2,3\}) = t_m(3)$.

3.4 Step 3: Enumerating Paths

Once the collapsed plan P^c is derived from P (as described before), our procedure enumerates all potential execution paths $Pt \in P^c$ and estimates the costs for each path (line 9-13 in Listing 1). An execution path Pt is defined as a path from a source operator (i.e., operators with no incoming edges) to a sink operator (i.e., operators with no outgoing edges) in the collapsed plan P^c.

For each enumerated path Pt, we estimate the total runtime cost T_{Pt} using the function call *estimateCost* (line 9), which we describe in the next section in detail. From all enumerated execution paths the dominant execution path Pt' is selected. The dominant execution path is the path which has the maximal estimated runtime cost under mid-query failures. The intuition is that in a PDE with inter-operator parallelism the dominant path is a good candidate to represent the total runtime of the complete collapsed plan P^c.

3.5 Step 4: Cost Estimation

In this section, we discuss how we estimate the total runtime cost of a given execution path Pt under mid-query failures (line 9 in Listing 1). The cost can be generally split down into three components: (1) the runtime cost without failures, (2) the expected runtime that is lost/wasted for each failure and (3) the number of attempts required to finish a query. While we already know (1) we focus here on (2) and (3).

Wasted Runtime Cost: If we assume that queries start at time t_0, the runtime wasted because of failure for an operator in path Pt is a linear function of time t from the start (t_0) until the operator finishes ($t_0+t(c)$). Figure 4 illustrates it for our running example; the potentially wasted runtime increases linear with time t until the operator finishes and the result is successfully materializes, which resets it for the next operator. Consequently the average wasted runtime for a failure of an operator c is the likelihood of a failure f at time t given that the failure happens during the execution of the operator, times the current execution time $(t - t_0)$:

$$\overline{w}(c) = \int_{t_0}^{t_0+t(c)} (t - t_0) \cdot P(f_t \mid f_{t_0<t<t_0+t(c)})\, dt \quad (2)$$

Here $P(f_t \mid f_{t_0<t<t_0+t(c)})$ corresponds to the likelihood of a failure f at time t given that a failure happens during the execution of the operator ($f_{t_0<t<t_0+t(c)}$). As outlined earlier we assume exponential arrival time between failures and that failures are independent, which further allows us to simplify the average cost for a single machine to:[4]

[4] $MTBF_{cost} = MTBF \cdot CONST_{cost}$ represents the MTBF transformed into an internal cost value of the PDE.

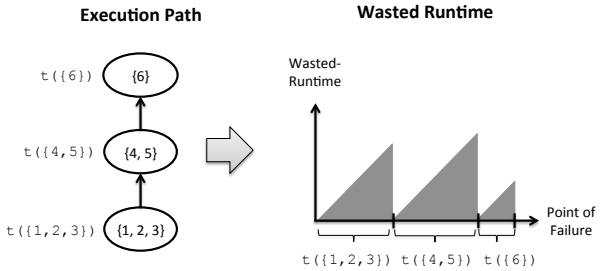


Figure 4: Wasted Runtime Cost

$$\begin{aligned}\overline{w}(c) &= \int_{t_0}^{t_0+t(c)} (t - t_0) \cdot \frac{P(f_t)}{P(f_{t_0<t<t_0+t(c)})}\, dt \\ &= \int_{t_0}^{t_0+t(c)} \frac{(t - t_0) \cdot e^{-\frac{t}{MTBF_{cost}}}}{MTBF_{cost} \cdot \left(-e^{-\frac{t_0+t(c)}{MTBF_{cost}}} + e^{-\frac{t_0}{MTBF_{cost}}} \right)}\, dt \\ &= MTBF_{cost} - \frac{t(c)}{e^{\frac{t(c)}{MTBF_{cost}}} - 1} \end{aligned} \quad (3)$$

First, it should be noted that $\overline{w}(c)$ does no longer depend on t_0 because of the fact that we have a stationary process (i.e., a Poisson process). Secondly, a limit analysis for $MTBF_{cost} \to \infty$ shows that:

$$\overline{w}(c) \to \frac{1}{2} \cdot t(c) \quad (4)$$

In fact, already for $MTBF_{cost} > t(c)$ the average wasted time $\overline{w}_0$ is close to $t(c)/2$. While potentially surprising, the reason is simple: the higher the $MTBF_{cost}$ is compared to the execution time of the operator, the more evenly distributed is the failure rate during the execution, resulting in an average closer to the middle ($t(c)/2$) of the execution time. As our main goal is not a precise failure model, but a reasonable fast to calculate cost model, we use $t(c)/2$ as an approximation of $\overline{w}(c)$ in the remainder of the paper.

Number of Attempts: Given our estimate for the average wasted runtime for a failure of operator c, we now estimate the number of additional attempts we need because of failures to successfully run the operator.

The likelihood that we have a failure in time-interval t given the exponential arrival times is $F(t) = 1 - e^{\frac{-t}{MTBF}}$ [15, 18]. Accordingly the probability that an operator $c \in Pt$ fails is $\eta(c) = F(t(c)) = 1 - e^{\frac{-t(c)}{MTBF_{cost}}}$ and that it succeeds $\gamma(c) = 1 - \eta(c) = e^{\frac{-t(c)}{MTBF_{cost}}}$. As a result the likelihood that operator c succeeds in N attempts is given as:

$$S(A \leq N) = \underbrace{\gamma(c)}_{A=0} + \underbrace{\eta(c) \cdot \gamma(c)}_{A=1} + ... + \underbrace{\eta^N \cdot \gamma(c)}_{A=N} \quad (5)$$

Here A corresponds to the number of attempts for operator c, not counting the first attempt (to not count the case that we do not have any failures). As it can be noted, $S(A \leq N)$ is a geometric series. For a given finite N, the cumulative probability of success can be presented as the following closed-form expression: $S(A \leq N) = \gamma(c) \cdot (1 - \eta(c)^{(N+1)})/(1 - \eta(c)) = (1 - \eta(c)^{(N+1)})$. Moreover, for $N \to \infty$ the cumulative probability of success is $\gamma(c)/(1 -$

c	{1, 2, 3}	{4, 5}	{6}	{7}
t(c)	4	3	1	2
$\overline{w}$(c)	2	1.5	0.5	1
γ(c)	0.94	0.95	0.98	0.96
a(c)	0.0648	0	0	0
T(c)	4.13	3	1	2

Table 2: Example - Cost Estimation

$\eta(c)) = \gamma(c)/\eta(c) = 1$ (i.e., at some point every operator will succeed).

Using the closed-form expression, we now derive the number of attempts $a(c)$ that operator c needs to achieve a desired probability of success $\hat{S}$ (i.e. $S(A \leq N) \geq \hat{S}$) as shown by Equation 6.

$$a(c) = max\left(\left(\frac{ln(1-\hat{S})}{ln(\eta(c))} - 1\right), 0\right) \quad (6)$$

In all our experiments in Section 5, we use $\hat{S} = 0.95$ (i.e., the 95th percentile) that is often used in literature [18] to represent the worst case.

Total Runtime: In order to estimate the total runtime T_{Pt} of an execution path Pt under the presence of mid-query failures, the idea is to sum up the estimated total runtime $T(c)$ of each operator in path Pt as shown by Equation 7.

$$T_{Pt} = \sum_{c \in Pt} T(c) \quad (7)$$

Based on the given number of attempts $a(c)$ per operator $c \in Pt$, we can estimate the total runtime $T(c)$ of an operator c (under node failures) as follows where $MTTR_{cost}$ is the mean time to repair a failure represented as an internal cost value:[5]

$$T(c) = \underbrace{t(c)}_{(1)} + \underbrace{a(c) \cdot \overline{w}(c)}_{(2)} + \underbrace{a(c) \cdot MTTR_{cost}}_{(3)} \quad (8)$$

The idea of $T(c)$ is that an operator c needs at least the time $t(c)$ shown as component (1) in Equation 8 to finish its execution (i.e., its pure execution time without mid-query failures). Moreover, the second component (2) represents the additional wasted runtime up to attempt $a(c)$ which is given as a sum of the average wasted runtime cost $\overline{w}(c)$ of c and the number of attempts $a(c)$. The last component (3) of $T(c)$ represents the costs needed to redeploy an operator (i.e, $a(c) \cdot MTTR_{cost}$). Based on $T(c)$, we are now able to calculate the total estimated cost T_{Pt} of path Pt under failures using Equation 7 and thus determine the dominant path of a fault-tolerant plan $[P, M_P]$.[6]

[5]As for $MTBF_{cost}$, $MTTR_{cost} = MTTR \cdot CONST_{cost}$ represents the MTTR transformed into an internal cost value of the PDE.

[6]It should be noted, that this model is an approximation and not an exact model, mainly for performance reasons. For instance, we assume that operations between machines are not blocking (e.g., one machine can always move ahead without waiting for another). Hence, we might underestimate the execution time. Furthermore, we do not model the different paths as stochastic variables, which they ultimately are. Overall, this leads to faster calculation times, but also imprecision. However, as our experimental evaluation will show for many scenarios the estimates are close enough.

Example: In the following we estimate the costs for the two execution paths Pt_1 and Pt_2 shown in Figure 3 to calculate T_{Pt1} and T_{Pt2}. For this example, we assume that $t(c)$ for each collapsed operator c is given (as shown in the following table). Moreover, we use $MTBF_{cost} = 60$ and $MTTR_{cost} = 0$. Based on this information, we can derive the average wasted time $\overline{w}(c)$ and the probability of success using the Equations 4 and 5 as shown in Table 2. Based on these values and $\hat{S} = 0.95$, we can calculate the number of attempts $a(c)$ per operator and the total runtime $T(c)$ using Equations 6 and 8. Consequently, we get $T_{Pt1} = 8.13$ and $T_{Pt2} = 9.13$ for the two paths and thus the dominant path is Pt_2.

4. PRUNING OF SEARCH SPACE

In this section, we discuss pruning rules to reduce the search space of potential fault-tolerant plans $[P, M_P]$. All pruning techniques are based on the previously described cost model.

4.1 Rule 1: High Materialization Costs

The first pruning rule is applied to a DAG-structured execution plan P that is returned by the first phase of function $enumFTPlans(Q)$ as described in Section 4.2 (i.e., before enumerating different materialization configurations).

The intuition of this pruning rule is to mark an operator $o \in P$ as non-materializable if materializing o is guaranteed to lead to a higher runtime costs under mid-query failures. When applying this pruning rule, we differentiate two cases: (1) the operator o is a child of a unary parent operator p and (2) the operator o is a child of a n-ary parent operator p (i.e., p has more than one child operator). In the following, we explain the pruning rule for these two cases more formally.

In the first case (1), we mark an operator o as non-materializable (i.e., we set $m(o) = 0$ and $f(o) = 0$), if $t(\{o, p\}) \leq t(\{o\})$ holds where $\{o, p\}$ and $\{o\}$ are collapsed operators. The runtime for collapsed operators without the additional costs for mid-query failures is calculated as discussed in Section 3.3. In the following we show that if $t(\{o, p\}) \leq t(\{o\})$ holds, we can guarantee that the runtime with recovery costs T_Pt of an arbitrary path Pt that contains $\{o, p\}$ will always be predicted to be less or equal the runtime of the same path where we replace $\{o, p\}$ by two separate operators $\{o\}$ and $\{p\}$ (where each output is materialized). Therefore, we mark such an operator o as non-materializable.

Following the equations in Section 3, if $t(\{o, p\}) \leq t(\{o\})$ holds, the wasted runtime $\overline{w}(\{o, p\})$ is less or equal the wasted runtime $\overline{w}(\{o\})$. Moreover, for the same reason the number of attempts $a(\{o, p\})$ is also less or equal $a(\{o\})$ since the probability of success $\gamma(\{o, p\})$ is greater equal than $\gamma(\{o\})$. To that end, the total estimated runtime (including the additional costs for mid-query failures) T_{Pt^1} of an arbitrary path Pt^1 that contains a collapsed operator $\{o, p\}$ is less or equal the total estimated runtime of a variant of Pt^1 called Pt^2 where we replace $\{o, p\}$ by only $\{o\}$. Adding operator $\{p\}$ to the path Pt^2 only increases the total runtime T_{Pt^2}. Thus, $T_{Pt^1} \leq T_{Pt^2}$ holds if $t(\{o,p\}) \leq t(\{o\})$ holds.

Figure 5 (left hand side) shows an example with two operators, where the total runtime costs of the collapsed operator $\{o, p\}$ is less than the estimated total runtime costs of the child operator $\{o\}$. For calculating the execution costs $t_r(\{o, p\})$, we use $CONST_{pipe} = 0.8$ in this example. Since $t(\{o, p\}) < t(\{o\})$ holds, we mark operator o as non-

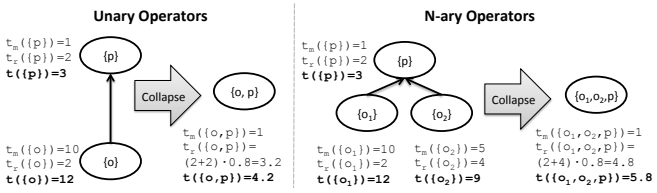


Figure 5: Rule 1 - High Materialization Costs

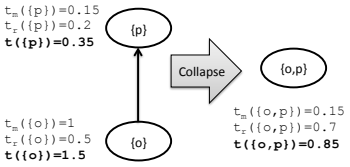


Figure 6: Rule 2 - Short-Running Operators

materializable and thus exclude operator o from the enumeration of materialization configurations.

Case (2) is similar to case (1): In oder to simplify the presentation, but without loss of generality, we assume that p has two child operators o_1 and o_2. In this case, we mark the two child operators o_1 and o_2 as non-materializable, if and only if it is guaranteed that the operator $\{o_1, o_2, p\}$, which results from collapsing o_1, o_2 and p, has a total estimated runtime costs less than o_1 and less than o_2. This is the case, if $t(\{o_1, o_2, p\}) \leq t(\{o_1\}) \wedge t(\{o_1, o_2, p\}) \leq t(\{o_2\})$ holds. In order to proof this pruning rule, we can use the same argument as before individually for $\{o_1\}$ and $\{o_2\}$. This proof can be extended to an arbitrary number of child operators of p.

Figure 5 (right hand side) shows an example for this case where the total runtime costs of the collapsed operator $\{o_1, o_2, p\}$ is less than the estimated total runtime costs of both operators $\{o_1\}$ and $\{o_2\}$. For calculating the execution costs $t_r(\{o_1, o_2, p\})$ we use $CONST_{pipe} = 0.8$ in this example. Since $t(\{o_1, o_2, p\}) \leq t(\{o_1\}) \wedge t(\{o_1, o_2, p\}) \leq t(\{o_2\})$ holds, we mark operator 1 and 2 as non-materializable and exclude those operators from the enumeration of materialization configurations.

4.2 Rule 2: High Probability of Success

The second pruning rule is applied to a DAG-structure execution plan P enumerated by the first phase of function $enumFTPlans(Q)$ described in Section (i.e., before enumerating different materialization configurations). This rule is only applied to operators o that are a child of a unary parent operator p. The idea of this pruning rule is to mark an operator o as non-materializable if the collapsed operator $\{o, p\}$ has a probability of success higher than the desired probability of success.

More formally, we mark o as non-materializable (i.e., we set $m(o) = 0$ and $f(o) = 0$ for all materialization configurations), if and only if $\gamma(\{o, p\}) \geq \hat{S}$ whereas $\hat{S}$ is the desired probability of success as discussed in Section 3.5. In that case, we expect that the collapsed operator finishes without an additional attempt (i.e., no mid-query failure happens). Since we save the materialization cost of o when collapsing o into p, the total runtime for any possible execution path Pt^1 which contains $\{o, p\}$ is under our model guaranteed to be less or equal total runtime of a variant of Pt^1 called Pt^2 where we replace $\{o, p\}$ by $\{o\}$ and $\{p\}$. For a high MTBF (which lead to higher success probabilities), this rule marks operators with even high total execution costs as non-materializable.

Figure 6 shows an example for this pruning rule. For the collapsed operator $\{o, p\}$ we get a probability of success $\gamma(\{o, p\}) = 0.999$ when applying the cost equation in Section 3 using a mean-time-between-failures of $MTBF_{cost} = 3600$. Since this probability is higher than our $\hat{S} = 0.95$, we mark operator $o \in P$ as non-materializable and thus exclude operator o from the enumeration of materialization configurations in our cost model.

4.3 Rule 3: Long Execution Paths

The last pruning rule is applied during the enumeration of execution paths for a given fault-tolerant plan $[P, M_P]$ to stop the path enumeration early without analyzing all execution path (i.e., the pruning can be applied after line 9 in Listing 1). The idea of this rule is that we stop the path enumeration early, if we find an execution path Pt, which has a runtime that is longer than the best memoized dominant path of all previously enumerated fault-tolerant plans for a given query Q.

More formally, if the analyzed path Pt of the current fault-tolerant plan has guaranteed higher total execution cost than the best memoized dominant path of a previously analyzed fault-tolerant plan (i.e., if $T_{Pt} \geq bestT$ holds), then we do not need to analyze any other execution path of the the current fault-tolerant plan. Consequently, in this case, we can stop the path enumeration for the current fault-tolerant plan early and continue with the next fault-tolerant plan without analyzing all paths.

In order to find out that if we can stop the path enumeration, we analyze if one of the following conditions holds: (1) $R_{Pt} \geq bestT$: If for the runtime R_{Pt} of path Pt without any failure, $R_{Pt} \geq bestT$ holds, we can skip all remaining paths for the given fault-tolerant plan $[P, M_P]$. The reason is that the dominant path (called Pt' further on) has a runtime $T_{Pt'}$ under mid-query failures that is greater equal the runtime T_{Pt} of the current path Pt. Thus, since $T_{Pt'} \geq T_{Pt}$, $T_{Pt} \geq R_{Pt}$ and $R_{Pt} \geq bestT$ holds, we can conclude that $T_{Pt'} \geq bestT$ holds as well. Moreover, since the runtime of a path without mid-query failures can be computed as $R_{Pt} = \sum_{c \in Pt} t(c)$, we can calculate R_{Pt} easily without calling the $estimateCosts$ function (in line 9 in Listing 1). (2) $T_{Pt} \geq bestT$: If for any path $T_{Pt} \geq bestT$ holds, we can skip all remaining paths for the given fault-tolerant plan $[P, M_P]$ since then the dominant path (called Pt') of the the given fault-tolerant plan $[P, M_P]$ will have a runtime $T_{Pt'}$ under mid-query failures that is guaranteed to be greater equal $bestT$. Thus, since $T_{Pt'} \geq T_{Pt}$ and $T_{Pt} \geq bestT$ holds, we can conclude that $T_{Pt'} \geq bestT$ holds as well. Compared to the rule before, however, we have to call the $estimateCosts$ function to compute T_{Pt}.

Additionally, instead of memoizing only one best dominant path and its runtime $bestT$, we could memoize multiple best dominant paths with different numbers of collapsed operators for pruning even more aggressively. These memoized dominant paths can then be used to check for a given execution path Pt, if we find a memoized dominant path Pt_m with the same number of collapsed operators where the following equation holds:

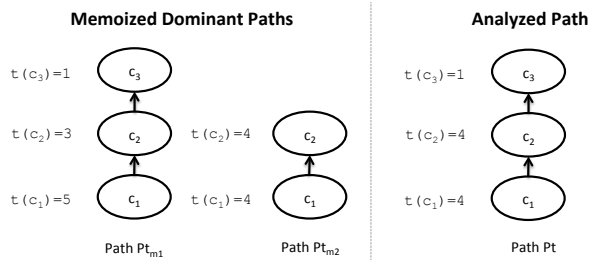


Figure 7: Rule 3 - Memoizing best Dominant Paths

$$\forall(i \in \{0...|Pt|\}) : sort(Pt).getT(i) \geq sort(Pt_m.getT(i)) \quad (9)$$

This means, we sort the operators in the paths Pt_m and Pt descending by their total execution costs (using function $sortTDesc$) and pairwise compare the total costs of the operators with the same index i in both sorted lists. If the condition in Equation 9 holds, we can derive that $T_{Pt} \geq T_{Pt_m}$. If the condition holds, we say $Pt \geq Pt_m$ for short. The rationale why this holds is that the average wasted runtime $\overline{w}(c)$ as well as the number of attempts $a(c)$ will be greater than or equal for each operator c in path Pt compared to path Pt_m and thus T_{Pt} will also be greater than or equal T_{Pt_m}. Moreover, we also can compare path Pt to any memoized dominant path Pt_m with a smaller number of collapsed operators, since we always can add further collapsed operators with total costs $t(\{o\}) = 0$ to Pt_m .

Figure 7 shows an example for this pruning rule. The left hand-side shows two memoized dominant paths: Pt_{m1} is the best dominant path with three collapsed operators (c_1' to c_3') and Pt_{m2} is the best dominant path with two collapsed operators (c_1'' and c_2'') that are already sorted descending by runtime to simplify the presentation. For the given path Pt with three collapsed operators (c_1 to c_3), we see that $Pt \geq Pt_{m1}$ does not hold but $Pt \geq Pt_{m2}$ holds. Thus, we can skip the enumeration of the remaining paths of the given fault-tolerant plan $[P, M_P]$.

Finally, since multiple equivalent DAG-structured physical execution plans are enumerated for the same query (as described in Section 3) in cost-based enumeration, this rule will lead to an even better reduction of the search space if we store $bestT$ as well as the memoized best dominant paths for the complete enumeration process of all equivalent execution plans for a given query. In our experiments in Section 5, we analyze the efficiency of this pruning rule when it is applied to all equivalent execution plans for a given query.

5. EXPERIMENTAL EVALUATION

The goal of the experimental evaluation is to show: (1) the efficiency of our scheme compared to existing fault-tolerance schemes for different types of queries and different failure rates (see Section 5.2 and Section 5.3), (2) the accuracy of our cost-based fault-tolerance scheme and its robustness (see Section 5.4), and (3) the effectiveness of our pruning techniques to reduce the search space of potential materialization configurations (see Section 5.5).

5.1 Setup and Workload

Cluster Setup: For our experiments, we used a cluster of 10 commodity machines each having 2 CPUs (Intel Xeon E5345, 2.33GHz, 4 cores), 8 GB RAM, and two local SCSI hard disks (10,000rpm) each with 73.4 GB storage capacity. As fault-tolerant storage medium, we used an external iSCSI storage with 12 disks (5,400rpm) each having 4 TB storage capacity attached via 1 GB Ethernet. Each cluster node was running the following software stack: openSUSE 13.1, MySQL 5.6.16, and Java 8.

Workload: Moreover, for the workload in our experiments, we used the TPC-H benchmark schema and data. We replicated the two small tables `NATION` and `REGION` to all cluster nodes. All other tables were horizontally partitioned into 10 partitions and distributed to different nodes (i.e., one partition per node). We co-partitioned the two tables `LINEITEM` and `ORDERS` using `HASH`-partitioning on the `orderkey` attribute. For all other tables, we used the `RREF`-partitioning [8] that partially replicates individual tuples in order to minimize distributed joins as follows: `CUSTOMER RREF` by `ORDERS` (on `custkey`), `PARTSUPP RREF` by `LINEITEM` (on `suppkey` and `partkey`), as well as `SUPPLIER RREF` by `PARTSUPP` (on `suppkey`).

Implementation: For actually running queries in parallel, we implemented our cost-based fault-tolerance scheme as well as the other existing fault-tolerance schemes in the open source PDE called *XDB* [8], which provides DAG-structured execution plans.*XDB* is implemented as a middleware that executes queries over sharded single-node MySQL database instances. The middleware provides the fault-tolerant query optimizer, which determines the subset of intermediates that will be materialized during execution. For execution, the query coordinator splits the execution plan according to the materialization configuration into sub-plans over different partitions, which are then executed on the corresponding cluster nodes. Sub-plans where configured to store their output to the external iSCSI storage. In order to achieve fault-tolerance with regard to mid-query failures, a query coordinator monitors the execution of individual sub-plans and restarts them once a failure is detected.

Statistics: In order to show the effect of different failure rates, we injected failures using different MTBFs that we also use as input to our fault-tolerant query optimizer. In all experiments, we used a monitoring interval to $2s$ in *XDB*. Thus, in average a failed operator was redeployed in $1s$ and we therefore use MTTR=$1s$ in all experiments. Moreover, we also use perfect cost estimates for $t_r(o)$ and $t_m(o)$ for our experiments in Section 5.2 and Section 5.3. In order to derive perfect query statistics, we executed all queries in *XDB* (w/o injecting failures) and measured $t_r(o)$ and $t_m(o)$ for each operator o. Additionally, in order to show the effects of non-exact estimates in Section 5.4, we introduced errors in these statistics. Finally, as constants, we use $CONST_{cost} = 1$ since the estimates represent the real time as well as $CONST_{pipe} = 1$ that we derived using a calibration experiment in *XDB*.

5.2 Efficiency for Different Queries

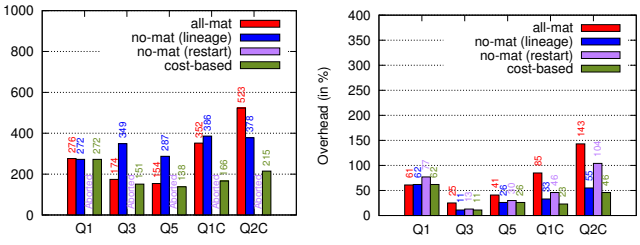
In this experiment we compare the overhead of different existing fault-tolerance schemes to our cost-based scheme when mid-query failures happen while executing queries over a partitioned TPC-H database of $SF = 100$. The reported overhead in this experiment represents the ratio of the runtime of a query under a given fault-tolerance scheme (i.e.,

including the additional materialization costs and recovery costs) over the baseline execution time. The baseline execution time for all schemes is the pure query runtime without additional costs (i.e., no extra materialization costs and no recovery costs due to mid-query failures). Thus, if we report that a scheme has 50% overhead, it means that the query execution under mid-query failures using that scheme took 50% more time than the baseline. The fault-tolerance schemes, which we compare in this experiment, are:

- **all-mat:** This represents the strategy of Hadoop, where all intermediates are materialized. Moreover, for recovering a fine-grained strategy is used (i.e., only sub-plans that fail are restarted).
- **no-mat (lineage):** This represents the strategy of Shark, where lineage information is used to re-compute failed sub-plans. Moreover, for recovering a fine-grained strategy is used.
- **no-mat (restart):** This represents the (coarse-grained) strategy of a parallel database, where the complete query plan is restarted once a sub-plan fails.
- **cost-based:** This represents our strategy that materializes intermediates based on a cost model. Moreover, for recovering a fine-grained strategy is used.

We compare the overhead of these schemes for different TPC-H queries with varying complexity: Q1 (no join), Q3 (3-way join) and Q5 (6-way join). Moreover, we run two complex queries: a variant of Q1(called Q1C) and a variant of Q2 (called Q2C). Q1C is a nested query that uses Q1 as inner query and joins the result with the `LINEITEM` table to count the individual items with a given status that have a price above the calculated average. Thus, Q1C is a query that has an aggregation operator in the middle of the plan. Q2C modifies Q2 (which is already nested) such that the inner aggregation query (4-way join) is used as a common-table-expression (CTE) which is consumed by two outer queries. For the two outer queries, we used the original outer query (5-way join) with different filter predicates on the `PART` table. This query represents a DAG-structured plan.

Moreover, for injecting failures, we use the following two settings per query: (1) an MTBF per node which is 10% higher than the baseline runtime of each query to simulate high failure rates, and (2) an MTBF per node which is 10× the baseline runtime to simulate low failure rates. For measuring the actual runtime, we created 10 failure traces for each unique MTBF using an exponential distribution where $\lambda = 1/MTBF$ and used the same set of traces for injecting failures to compare the overhead of different fault-tolerance schemes. We used this method in all experiments.



(a) Low MTBF (b) High MTBF

Figure 8: Varying Queries

The result of this experiment is shown in Figure 8. The **cost-based model always selects the sweet spots for materialization** for different queries and different failure rates. Thus, the cost-based scheme has the least or comparable overhead as the best existing fault-tolerance scheme. Query Q1 is an exception in this experiment since it has no free operator that can be selected for materialization. Thus, all schemes show almost the same overhead except for no-mat (restart).

Low MTBF: For all queries (except Q1), the not-mat (lineage) scheme has a higher overhead than the cost-based scheme. Moreover, the no-mat (restart) scheme does not finish any query (i.e., we aborted them after 100 restarts). Another interesting pattern is that both star-join queries (Q3 and Q5) have a similar overhead for the cost-based and the all-mat scheme. The reason is that the cost-based scheme materializes most intermediate results (except the most expensive ones) and therefore the resulting overhead is similar to the all-mat scheme. For more complex queries (Q1C and Q2C), the cost-based scheme, has clearly the best overhead. The reason is that these queries contain an aggregation operator in the middle of the plan, which has low materialization costs. This aggregation operator is selected by the cost-based scheme as a checkpoint that efficiently minimizes the overhead under mid-query failures. Moreover, for Q1C and Q2C, the all-mat scheme has a much higher overhead compared to the cost-based scheme. The reason is that the total materialization costs for many operators are relatively high and the cost-based scheme thus does not materialize these operators.

High MTBF: For low failure rates the results are different. The no-mat (lineage) scheme and the cost-based scheme are the best schemes for Q3 and Q5. The reason is that the cost-based scheme only materializes few small intermediates and thus is similar to the no-mat (lineage) scheme. For Q1C and Q2C, the no-mat (lineage) scheme is slightly worse since the cost-based scheme materializes the small aggregation operator in the middle of the plan and has thus a lower overhead if a failure occurs. The no-mat (restart) scheme also tends to have a slightly higher overhead than the cost-based scheme since it is a coarse-grained scheme. Interestingly, the all-mat scheme also has only a slightly higher overhead than the cost-based scheme for Q3 and Q5. The rationale is that these queries have moderate total materialization costs (approx. $20-30\%$ of the runtime costs) under all-mat. Q1C and Q2C, however, have much higher materialization costs (approx. $60-100\%$ of the runtime costs) for the all-mat scheme leading to a much higher overhead.

5.3 Efficiency for Varying Statistics

In this experiment, we compare the overhead of the different fault-tolerance strategies (a) when running the same query with varying runtime for a fixed MTBF to show the effect of short- and long-running queries and (b) when running the same query under MTBFs to show the effect of different cluster setups.

Exp. 2a - Varying Query Runtime (Figure 10): In this experiment, we executed TPC-H query 5 over different scaling factors ranging from $SF = 1$ to $SF = 1000$. This resulted in query execution times ranging from a few seconds up to multiple hours. We selected TPC-H query 5 in this experiment since this is a typical analytical query with multiple join operations and an aggregation operator on top (see Figure 9). For this experiment, the output of every join

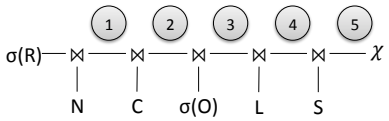


Figure 9: TPC-H Query 5 (Free operators 1-5)

operator was defined to be a free operator (marked with the numbers 1-5 in Figure 9) and thus could be selected by our cost model to be materialized. Thus, for each enumerated plan, our procedure in Section 3 enumerated 2^5 materialization configurations when pruning was deactivated. Moreover, we injected mid-query failures using a MTBF of 1 day (1440 minutes) per node.

The result of this experiment is shown in Figure 10. The x-axis shows the baseline-runtime of the query (i.e., when no failure is happening) and the y-axis shows the overhead under mid-query failures. The **cost-based scheme has the lowest overhead for all queries**; starting with 0% for short-running queries and ending with 247% for long-running queries. Compared to our cost-based scheme, the other schemes impose a higher or comparable overhead depending on the query runtime. Both no-mat schemes also start with 0% overhead for short-running queries. However, for queries with a higher runtime, the overhead increases. As expected, for the restart-based no-mat scheme, queries with a high runtime tend to not finish since the complete query is restarted over and over. The lineage-based not-mat scheme degrades a more gracefully. However, it still has the second highest overhead since sub-plans need to be restarted from scratch. The all-mat scheme behaves very similar to the cost-based scheme for short- and long-running queries. The reason is, that the total materialization costs of all operators (1-5 in Figure 9) represent only 34.13% of the total runtime costs. For long-running queries, the cost-base scheme materializes the intermediates 2 and 3. As a result, the cost-mat scheme has 63% less overhead than the all-mat scheme resulting from lower materialization costs and less attempts to finish the query under mid-query failures. For short running queries the overhead of all-mat is exactly 34% higher since the cost-based scheme does not materialize any intermediate.

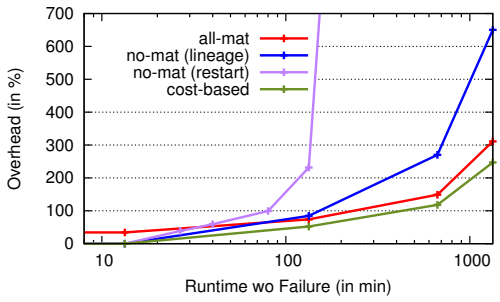


Figure 10: Varying Runtime

Exp. 2b - Varying MTBF (Figure 11): This experiment shows the overhead of the fault-tolerance schemes mentioned before when varying the MTBF. In this experiment, we executed TPC-H query 5 over $SF = 100$ using a low selectivity. This resulted in a query execution time of 905.33s (i.e., approx. 15 minutes) as a baseline-runtime when injecting no failures and adding no additional materializations in the plan. In order show the overhead, we executed the same query using the following MTBFs per node: 1 week, 1 day, and 1 hour.

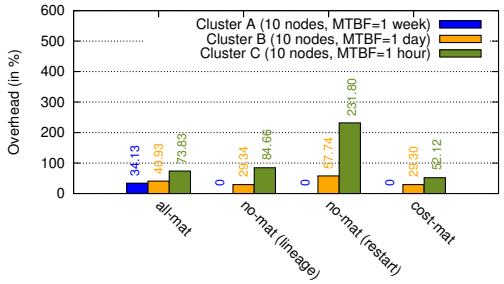


Figure 11: Varying MTBF

Figure 11 shows the overhead of the individual schemes under varying MTBFs. This figure shows the same trends as already reported before in Figure 10. The **cost-based scheme has the lowest overhead for all MTBFs** when compared to the other schemes using the same MTBF. Both not-mat schemes show a higher increase of the overhead under high failure rates (i.e., a low MTBF). The all-mat scheme again imposes unnecessary overhead for low failure rates and is the second best for high failure rates since the materialization overhead for all operators of Q5 is only 30% of the query runtime.

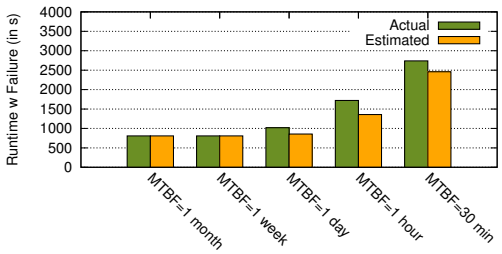
5.4 Accuracy and Robustness of Cost Model

In this experiment we show the accuracy and robustness of our cost model: (a) For showing the accuracy of our cost model, we compare the actual runtime with the estimated runtime for different fault-tolerant plans (enumerated by our cost-based scheme) and for different MTBFs. (b) For showing the robustness of our cost model, we introduce errors in the statistics and analyze the effects on the plan selection.

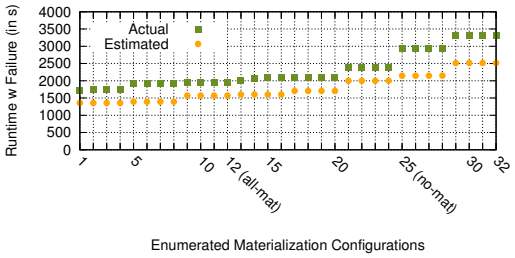
Exp. 3a - Accuracy of Cost Model (Figure 12): In this experiment, we executed TPC-H query 5 over $SF = 100$ using a low selectivity. This resulted in a query execution time of 905.33s (i.e., approx. 15 minutes) as a baseline-runtime when injecting no failures and adding no additional materializations in the plan. In order to cover a wide range of MTBFs, we added extreme MTBFs to cover a wide range from 30 minutes to 1 month (different from Experiment 1b in Section 5.4).

Figure 12(a) shows the accuracy results (i.e., actual vs. estimated runtime) for different MTBFs. While for high MTBFs (i.e., low failure rates) the error is 0%, we get an error of 30% for low MTBFs. In general the cost model tends to underestimate the runtime when injecting failures. However, with an increasing estimated runtime, we see also an increase in the actual runtime. This behavior is crucial for a good cost model to select plans with a minimal actual runtime.

Figure 12(b) shows the accuracy when enumerating different 2^5 materialization configurations for the plan of TPC-H query 5 shown in Figure 9 for a fixed MTBF of 1 hour. The x-axis shows the 2^5 enumerated materialization configurations sorted ascending by their estimated runtime. The y-axis shows the estimated/actual runtime for each of the enumerated plans. The plot shows, that there is a **high correlation of the estimated and actual runtime** for all enumera-



(a) Varying MTBF



(b) Varying Mat. Confs.

Figure 12: Accuracy of Cost Model

ted materialization configurations which validates our cost model (i.e., a plan which has lower estimated cost has also lower actual cost). As discussed before, this behavior is crucial for a good cost model to select plans with a minimal actual runtime.

Exp. 3b - Robustness of Cost Model (Table 3): In this experiment, we evaluate the sensitivity of our cost model to inaccurate statistics. We again use the plan of TPC-H query 5 as shown in Figure 9. The reason is that the runtime of this plan under mid-query failures strongly depends on the materialization configuration chosen by our cost-based scheme. As shown before in Figure 12(b), the runtime varies for all enumerated materialization configurations from 1358s to 2517s (for $SF = 100$ and an MTBF of 1 hour). To evaluate the sensitivity of our scheme, we vary the input statistics of our cost model and report how these changes affect the top-5 plans. When changing the I/O costs, we multiplied the materialization costs ($t_m(o)$) of each operator with a given perturbation factor, before applying our cost model. When changing all costs, we multiplied the all operator costs ($t_r(o)$ and $t_m(o)$) with a given perturbation factor, before applying our cost model.

The baseline is the ranking of materialization configurations shown in Figure 12(b) that represents the case with exact statistics. Table 3 shows the results of this experiment. Each line in this table shows which materialization configuration of the baseline ranking moved to the top-5 positions when perturbing the statistics (i.e., the higher the number, the worse is the selected plan). In general, perturbations with small factors (i.e., 0.5× and 2×) often **change the order within the top-5 materialization configurations only slightly**. This shows that our cost-based scheme is robust towards typical perturbations. In this case, our cost-based approach does not select the most optimal fault-tolerant plan but it selects a fault-tolerant plan that is close to the optimal plan in terms of its runtime under mid-query failures as shown in Figure 12(b). However, for extreme perturbations (i.e., 0.1× and 10×) our cost model is more sensitive. In the worst case, a materialization configuration which was on position 28 in the baseline ranking (out of 32) is placed on rank 1 after perturbation, resulting in a materialization configuration which has a 1.7× higher runtime compared to the optimal materialization configuration. Moreover, perturbations in the I/O costs have a much stronger effect as perturbations of the other two categories. This is also clear, since our cost-based scheme then favors configurations with less materializations when compared to the perfect ranking.

Ranking w exact statistics	**1**	**2**	**3**	**4**	**5**
MTBF ×0.1	3	4	1	2	5
MTBF ×0.5	3	4	1	2	5
MTBF ×2	8	7	5	6	2
MTBF ×10	28	27	25	26	18
I/O costs ×0.1	11	12	9	10	13
I/O costs ×0.5	3	1	2	4	11
I/O costs ×2	5	7	6	8	2
I/O costs ×10	27	25	28	26	17
Compute & I/O costs ×0.1	28	27	25	26	8
Compute & I/O costs ×0.5	7	8	5	6	2
Compute & I/O costs ×2	3	4	1	2	5
Compute & I/O costs ×10	3	4	1	2	5

Table 3: Robustness of Cost Model

5.5 Effectiveness of Pruning Rules

In our final experiment, we show the effectiveness of our pruning rules presented in Section 4. Therefore, we enumerate all 1344 equivalent join orders of TPC-H query 5 (i.e., we do not enumerate plans with cartesian products) and apply our cost model with and without pruning rules enabled for $SF = 10$ and three different cluster setups with varying MTBFs: 1 week, 1 day, and 1 hour. We analyze the pruning efficiency for different cluster setups since the pruning rules 2 and 3 depend on the given MTBF.

Figure 13 shows the percentage of fault-tolerant plans that are pruned by the individual rules and the overall percentage of pruned fault-tolerant plans accumulated for all pruning rules 1-3. If we do not apply any pruning rule (no pruning), then 2^5 different materialization configurations need to be analyzed for each enumerated execution plan since TPC-H query has 5 free operators. Thus, without pruning $43,008$ fault-tolerant plans must be enumerated when no pruning is activated. When activating all pruning rules, **in the best case 36% of the fault-tolerant plans can be pruned** for a MTBF of 1 week whereas in the worst case 26% can be pruned for a MTBF of 1 hour. In the following, we report the results when activating the pruning rules one after each other and explain the decreased effectiveness of the pruning rules for lower MTBFs.

The first rule (i.e., rule 1 in Section 4) is the most efficient rule and prunes constantly (i.e., independent of the MTBF) 25% of all fault-tolerant plans. The reason is that some of the join operators in TPC-H query 5 have a quite large intermediate result (e.g., when joining `LINEITEM` and `SUPPLIER`). Thus, materializing the output of those joins is more expensive than running the subsequent operator, which means that we can set these join operators with large intermediate results to be not materialized.

The second rule (i.e., rule 2 in Section 4) is less efficient than rule 1 and prunes only $0.74\% - 7.15\%$ depending on the given MTBF. First, this rule generally prunes less materialization configurations since only the very last operators of a query tree (i.e., aggregation and projection) are typically short running and thus can be set to be not materialized.

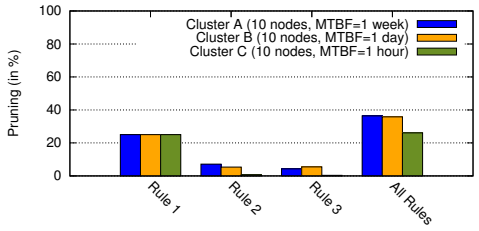


Figure 13: Effectiveness of Pruning

Moreover, for a higher MTBF, the probability of success of an operator increases even for longer running operators. In that case, more operator can be pruned (i.e., they are set to be not-materialized).

Compared to rule 1 and 2, rule 3 actually does not prune fault-tolerant plans eagerly before they are enumerated but it prunes them lazily during the enumeration of execution paths. More precisely, it prunes a fault-tolerant plan, once t finds an execution path Pt which has a total runtime T_{Pt} longer than the best dominant path found so far. Thus, the efficiency of this pruning rule depends strongly on the enumeration order of execution paths. In this experiment, we count those fault-tolerant plans where this rule can be applied at all and at the end regard only the half of the fault-tolerant plan as being pruned. The reason why we only count only the half of the fault-tolerant plans is, that pruning has two extreme cases: (1) the rule can be already applied for the first enumerated execution path (i.e., we skip all other execution paths of the same fault-tolerant plan), or (2) the rule is applied only for the very last enumerated execution path (i.e., we do not skip any other execution path of the same fault-tolerant plan). Thus, in average half of the costs for analyzing the paths can be avoided by this rule.

Figure 13 shows that the pruning efficiency of rule 3 is also increasing for higher MTBF. The reason is that the best dominant path has a lower total runtime $bestT$ for an increasing MTBF. Thus, the pruning condition 1 of rule 3 (see Section 4.3), which compares the runtime R_{Pt} of a path Pt without mid-query failures to the memoized best dominant path, holds more often.

6. RELATED WORK

Parallel databases [4, 2, 6, 5] typically can handle various types of failures. One of the main issues discussed in the literature is to maintain consistency in transactional workloads despite failures [15]. Moreover, another well studied field in parallel databases is how to achieve high-availability of the database by replicating partitions to multiple nodes in a cluster [11]. When introducing replication in transactional workloads, modern parallel databases often relax consistency (e.g. by using eventual consistency) in favor of better supporting availability or network partitioning [9] in order to being able to scale in large clusters. However, to the best of our knowledge there is no published work on how parallel databases handle mid-query failures in analytical workloads. The main approach in most parallel databases is to restart a query once a mid-query failure happens. However, this scheme is not efficient when running on large clusters of commodity machines or IaaS offerings such as Amazon's Spot Instances where a mid-query failure is a much more common case. This requires a more fine-granular fault-tolerance scheme, which checkpoints intermediate results.

Fine-granular fault-tolerance scheme are typically found in modern PDEs (such as Hadoop [1], Shark[22], Dryad [13]) as well as in many stream processing engines [12, 17]. While stream processing engines checkpoint the internal state of each operator for recovering continuous queries, MapReduce-based systems [10] such as Hadoop [1] typically materialize the output of each operator to handle mid-query failures. For being able to recover, they rely on the fact that the intermediate result is persistent even when a node in the cluster fails, which requires expensive replication and prevents support for latency-sensitive queries. Other systems, like Impala [3] and Shark [22] store their intermediates in main-memory in order to better support short running latency-sensitive queries. Moreover, Shark uses the idea of resilient distributed datasets [24], which store their lineage in order to enable re-computation instead of replicating intermediates. In contrast, in Dryad [13] operators can be configured to either pipeline their output to the next operator(s) or materialize the intermediate results. However, in Dryad this must be configured manually; i.e., there is no optimizer that decides which results should be materialized and which should be pipelined for an efficient execution under failures.

Two more recent approaches, which tackle the same problem as discussed in this paper are FTOps [19] and Osprey [23]. FTOps also presents an optimizer to find the best fault-tolerance strategy for each operator of a given plan. However, compared to our approach, FTOps only supports tree-structured plans and not DAG-structured plans. Another difference is that FTOps only supports plans with aggregations at the top, while we support arbitrary positions of aggregation operators in the plan. Moreover, FTOps runs as a post-processing step and uses a brute-force enumeration technique to analyze different fault-tolerance strategies for input plan while our approach analyses the top-k plans and efficiently prunes the search space of different fault-tolerant plans. Compared to FTOps, Osprey applies MapReduce-style fault- tolerance to parallel databases: Osprey splits analytical queries over a star schema into multiple sub-queries over individual partitions and executes a final merge of all sub-queries. If a sub-query fails it is re-started on a different replica. However, Osprey does not provide a cost-based optimizer to select an optimal fault-tolerant plan that includes additional materializations.

7. CONCLUSIONS AND OUTLOOK

In this paper, we presented our novel cost-based fault-tolerance scheme for parallel data processing. Compared to the existing strategies which either materialize all intermediates or no intermediates, our scheme selects an optimal subset of intermediates to be materialized such that the query runtime is minimized under the presence of mid-query failures. Our experiments show, that our scheme is efficient for different queries as well different cluster setups whereas the existing schemes only have their sweet-spot for certain cluster setups and query workloads.

One main avenue of future work, is to integrate other fault-tolerance strategies (e.g., check-pointing of the operator state to also support mid-operator failures) into our cost-based fault-tolerance scheme. This could be helpful especially for long running operators which otherwise are likely to fail often. Moreover, we also want to look into more dynamic decision for cases where data is skewed or statistics are hard to estimate (e.g., for user-defined functions).

8. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] HP Vertica Database. http://www.vertica.com/.

[3] Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[4] Pivotal Greenplum Database. http://www.gopivotal.com/big-data/pivotal-greenplum-database.

[5] SAP HANA Database. www.sap.com/HANA.

[6] Teradata Database. http://www.teradata.com/.

[7] C. Binnig, N. May, and T. Mindnich. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *BTW*, pages 363–382, 2013.

[8] C. Binnig, A. Salama, A. C. Müller, E. Zamanian, H. Kornmayer, and S. Lising. XDB: a novel database architecture for data analytics as a service. In *IEEE Big Data*, 2014.

[9] E. A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[11] H.-I. Hsiao and D. J. DeWitt. A Performance Study of Three High Available Data Replication Strategies. *Distributed and Parallel Databases*, 1(1):53–80, 1993.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. B. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, pages 779–790, 2005.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[14] G. Moerkotte. *Building Query Compilers*. University of Mannheim, http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, 2014.

[15] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[16] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's Adolescence. *PVLDB*, 6(10):853–864, 2013.

[17] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. B. Zdonik. Load Management and High Availability in the Borealis Distributed Stream Processing Engine. In *GSN*, pages 66–85, 2006.

[18] P. Tobias and D. Trindade. *Applied Reliability, Third Edition*. Taylor & Francis, 2011.

[19] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD Conference*, pages 241–252, 2011.

[20] F. M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE (Informal Proceedings)*, 2008.

[21] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[22] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

[23] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, pages 657–668, 2010.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[25] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *VLDB J.*, 21(5), 2012.