Spotgres - Parallel Data Analytics on Spot Instances

Carsten Binnig, Abdallah Salama, Erfan Zamanian
Muhammad El-Hindi, Sebastian Feil, Tobias Ziegler
Cooperative State University of Baden-Wuerttemberg, Mannheim, Germany

***Abstract*—Market-based IaaS offers such as Amazon's EC2 Spot Instances represent a cost-efficient way to operate a cluster. Compared to traditional IaaS offers which follow a fixed pricing scheme, the per hour price of Spot Instances changes dynamically, whereas the Spot price is often significantly less when compared to On-demand and even the Reserved Instances. When deploying a Parallel Data-Processing Engine (PDE) on a cluster of Spot Instances a major obstacle is to find a bidding strategy that is optimal for a given workload and satisfies user constraints such as the maximal budget. Moreover, another obstacle is that existing PDEs implement rigid fault-tolerance schemes which do not adapt to different failure rates resulting from different bidding strategies.**

In this paper, we present a novel PDE called *Spotgres* that tackles these issues. *Spotgres* extends a typical PDE architecture by (1) a constraint-based bid advisor which finds an optimal cluster configuration (i.e., a set of bids on Spot Instances) and (2) a cost-based fault-tolerance scheme that takes various parameters (such as the mean time between failures and query statistics) into account to efficiently execute analytical queries over the set of Spot Instances that have a varying failure rate.

I. Introduction

Motivation: Market-based IaaS offers such as Amazon's EC2 Spot Instances represent a cost-efficient way to operate a cluster. Compared to On-demand and Reserved Instances which follow a fixed pricing scheme, the per hour price of Spot instances changes dynamically, whereas the Spot price is often significantly less when compared to On-demand and even the Reserved Instances. In order to rent one or several Spot instances, the user places a bid which includes the number of instances of one particular machine type and the maximal per hour price that the user is willing to pay (called bid price). If the bid price exceeds the Spot price for the given machine type, Amazon will launch the instance(s) for the user. In that case, Amazon charges the user the current Spot price and not the bid price. Moreover, Spot Instances are billed in the granularity of an hour (using the Spot price of the last full hour if an instance is running longer than an hour). However, if the Spot price exceeds the bid price, Amazon could and will most likely kill all instance(s) without signaling the user beforehand, whereas Amazon does not charge for the last non-complete hour of uptime. Consequently, the availability of Spot Instances and the actual cost of the complete cluster are directly linked to the bidding strategy (i.e., the cost the user is willing to spend per hour per node) and the actual price.

Figure 1 shows the Spot price history in the AWS management console for an EC2 machine of type $c1.medium$ for one availability region (eu-west-1c) over the duration of one month. Compared to the On-demand price of $c1.medium$

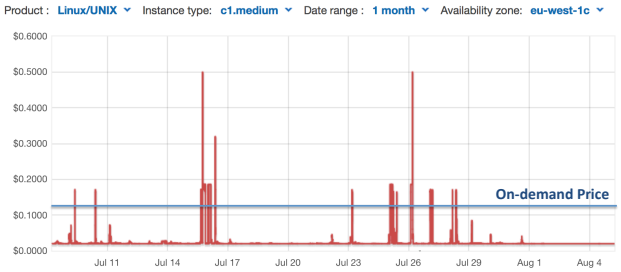


Fig. 1. Spot Price History (1 Month)

machines, which is approx. 13 cent per hour in that availability region, the Spot price is significantly less most of the time varying between 1 and 3 cent per hour. However, we can also see some individual spikes which range up to 50 cent per hour. Thus, if the user follows a strategy to place a bid which is only a little above the current Spot price (e.g.4 cent per hour), she minimizes the risk of paying a high per hour price but the user will (most likely) lose her instances when spikes occur. On the other hand, if the user bids a high price (e.g., above the highest spike), she minimizes the risk of losing an instance but increases the risk of paying a per hour price which reflects one of the spikes.

Parallel Data-Processing Engines (PDEs) such as parallel databases (e.g., SAP HANA [3], Greenplum [12] or Terradata [8]) or other modern parallel data management platforms (e.g., Hadoop [13], Scope [18], Shark [14]) are used today to analyze large amounts of data in clusters of shared-nothing machines. When deploying a Parallel Data-Processing Engine (PDE) on a cluster of Spot Instances a major obstacle is to find a bidding strategy that is optimal for a given workload and satisfies user constraints such as the maximal budget. Currently there is no support from Amazon to create a set of bids for a cluster of Spot Instances. Thus, the user has to manually find the best set of bids for her constraints and constantly needs to adapt her set of bids (i.e., re-bid) on Spot Instances if individual instances fail.

Moreover, another obstacle is that existing PDEs implement rigid fault-tolerance schemes to deal with mid query-failures. Whereas traditional parallel databases restart a query from scratch (on a replica) if a mid-query failure occurs, modern PDEs materialize intermediates to implement a more fine-grained scheme which restarts a sub-query from the last checkpoint. However, no PDE adaptively adjusts its execution strategy to a varying mean time between failures (MTBF) and other characteristics of the workload (e.g., query runtime and materialization costs for intermediates).

Contributions: In this paper, we represent a novel PDE called *Spotgres* that tackles the before mentioned issues. *Spotgres* extends a typical PDE architecture by (1) a constraint-based bid advisor which finds an optimal cluster configuration (i.e., a set of bids) based on a given set of constraints and (2) an cost-based fault-tolerance scheme that takes various parameters (such as MTBF and query statistics) into account to efficiently execute analytical queries in parallel over the set of launched Spot Instances.

The constraint-based bid advisor of *Spotgres* supports the user to specify constraints on the resources in the cluster as well as a quality constraint on the maximal budget or the minimal availability. On the one hand, if the maximal budget is given as a constraint, the bid advisor will propose a set of bids which aims to maximize the availability. On the other hand, if the minimal availability per node is given as a constraint, the bid advisor will propose a set of bids which aims to minimize the total costs per hour for the given availability.

The cost-based fault-tolerance scheme of *Spotgres* implements an efficient execution strategy for the set of launched Spot Instances. Therefore, *Spotgres* constantly collects statistics (such as the MTBF) for the individual machine types in the cluster and finds the best materialization strategy for a given query that minimizes the runtime under mid-query failures.

Our experiments show that the bid advisor effectively meets the given constraints. For example, the bid advisor suggests a cluster of Spot Instances which has an availability of 98% for only 25% of the costs of the cheapest On-demand cluster. Moreover, our cost-based fault-tolerance scheme shows that even with high failure rates (e.g., low MTBFs) queries can be executed more efficiently compared to other existing fault-tolerance schemes.

Outline: In Section II, we first present the architecture of *Spotgres* and discuss how the bid advisor and the cost-based fault-tolerance scheme are integrated into a typical PDE architecture. In Section III we discuss the details of the bid advisor which uses constraint programming to solve the optimization problem discussed above. Moreover, in Section IV we present the details of our cost-based fault-tolerance scheme. Finally, Section V shows our experimental evaluation and Section VI summarizes related work.

II. Spotgres Architecture

Figure 2 shows the architecture of our *Spotgres* PDE along two important aspects: (1) the cluster configuration using constraints and (2) the cost-based fault-tolerant query execution. In Figure 2, the blue arrows represent the control-flow of the cluster configuration, whereas the green arrows represent the control-flow of the query execution. In the following, we explain details of the individual components.

Bid Advisor: The most important component for the cluster configuration is the bid advisor which takes a set of input constraints from the administrator (e.g., resource constraints) and derives an optimal cluster configuration represented as

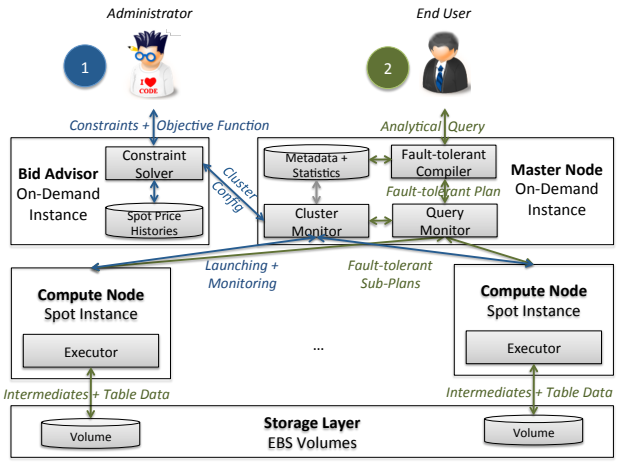


Fig. 2. *Spotgres* Architecture

a set of bids on Spot Instances (whereas each bid includes the machine type, the number of instances, and the bid price). Moreover, the administrator must define an objective function to derive a cluster configuration. Currently *Spotgres* supports only two objective functions: either maximizing the availability of each cluster node for a given total budget or minimizing the cost for the complete cluster for a given availability per node. The input constraints and the objective function are translated into a constraint program by the *Constraint solver* which then finds an optimal set of bids. Therefore the solver uses the Spot price histories (of a given window) to solve the optimization problem. The suggested cluster configuration is used by the cluster monitor (which is a part of the master node) to launch the Spot Instances.

Master Node: The master node hosts components for compiling and executing queries as well as monitoring compute nodes (which actually execute sub-queries). The *Fault-tolerant Compiler* takes a query from the end user and finds a plan that materializes a subset of intermediate results such that the runtime of the query is minimized under mid-query failures. For selecting a subset of intermediate results that should be materialized, *Spotgres* implements a cost model which uses query statistics (i.e., histograms of the data) and cluster statistics in order to find an optimal fault-tolerant plan that minimizes the runtime under mid-query failures. For query execution, the *Query Monitor* gets the fault-tolerant plan and splits it into sub-plans that are executed by the compute nodes in parallel over different partitions and materializes their output to the storage layer (EBS in our case) for fault-tolerance. For monitoring the complete cluster, the *Cluster Monitor* pings all individual compute nodes. In case of a node failure, *Cluster Monitor* uses the bid advisor to start new machines, i.e., to re-bid for failed Spot Instances. Moreover, the cluster monitor also updates the cluster statistics such as mean time between failures (MTBF) that are used by the fault-tolerant query compiler to select the subset of intermediates that should be materialized.

Since the *Master Node* and the *Bid Advisor* are central components in *Spotgres*, they are deployed on an On-Demand

Instance such that we do not need to deal with fault-tolerance in these nodes at all.

Compute Nodes: The compute nodes are executing sub-queries (i.e., sub-plans) over individual partitions and materializing the result of the sub-plans to the attached EBS volumes of the storage layer. All compute nodes are deployed on Spot Instances which means that these nodes can fail during query execution.

Storage Layer: *Spotgres* horizontally partitions a database D into n partitions P_i with $i = 1...n$ (i.e., $D = [P_1, ..., P_n]$ and replicates individual partitions $P_i \in D$ to k different nodes for fault-tolerance. Each individual copy of a partition is stored to a separate EBS volume. That way, if a node in the cluster fails, the cluster monitor can reattach the EBS volume(s) of the failed instance to a new (or an already running) instance. Afterwards, a failed sub-plan can be restarted from the most recent intermediate result that was materialized to an EBS volume.

III. Constrained-based Bid Advisor

A. Overview

In this section, we give an overview of the bid advisor which suggests a cluster configuration (i.e., a set of bids) based on the given input constraints. The input constraints of the user are shown in the following table:

Input	Description	Constr.Type
CU	Minimal number of Amazon Compute Units in the cluster	Resource
RAM_{cu}	Minimal RAM per CU in the cluster	Resource
B	Maximal budget for the complete cluster	Quality
A_{cu}	Availability per CU	Quality

The first two constraints are called *Resource* constraints, which define functional aspects of the cluster: i.e., the computational power of the complete cluster in compute units CU and the minimal main memory per CU. In order to specify the computational power per machine type, Amazon provides an abstract measure called *Compute Unit* CU (i.e., for each machine type the number of CUs is defined). That way, a CU can be seen as a virtual processor that provides a fixed computational power. Thus, in order to define the computational power of the complete cluster, the user gives the minimal number of CUs in the cluster as an input constraint. The second input constraint is the minimal RAM per CU denoted as RAM_{cu}. These two constraints heavily depend on the expected workload.

The other two constraints in the table before are called *Quality* constraints, which define non-functional aspects of the cluster: i.e, the maximal budget B (i.e., an upper bound for the actual cluster cost) and the minimal availability per compute unit A_{cu}. Out of these two constraints, the user must select one, whereas the other variable is used as the

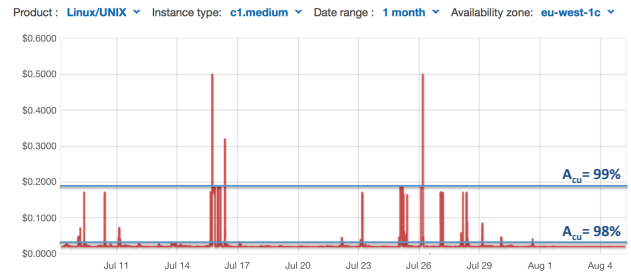


Fig. 3. Derive the Bid Price for a Machine Type

objective function: i.e., if the maximal budget B is provided as an input constraint, the objective function is to maximize the availability A_{cu}, whereas if the availability A_{cu} per CU is provided as an input constraint, minimizing the maximal budget B is the objective function.

In the following, we discuss both cases in detail.

B. Minimizing Cluster Cost

In the first case, the user provides the minimal availability per compute unit A_{cu} as quality constraint as well as the two resource constraints: CU and RAM_{cu}. Afterwards, the bid advisor executes the following steps: (1) filter all machine types $T = [t_1, \ldots, t_n]$ that qualify for the given RAM_{cu} constraint, (2) derive the bid prices $P = [p_1, \ldots, p_n]$ per machine type $t_i \in T$ to satisfy the given A_{cu} and calculate the average billed prices $A = [a_1, \ldots, a_n]$ per machine type $t_i \in T$, and finally (3) find the optimal cluster configuration using the constraint solver.

The first step is trivial since Amazon provides the number of $CU = [cu_1, \ldots, cu_n]$s and the RAM per machine type t_i. The result is the vector $T = [t_1, \ldots, t_n]$ of machine types that qualify for the given RAM_{cu} constraint. For the second step, the Spot price history per machine type $t_i \in T$ is used. Figure 3 shows the price history for one machine type over the last month (whereas the window we use for learning the bid price is configurable). Based on the history and the given A_{cu}, we can derive a bid price b_i that will (based on the history) give us the desired availability for that machine type. For example, in Figure 3 we set $b_i \geq \$0.03$ to guarantee an availability of $A_{cu} = 98\%$, whereas if we set $b_i \geq \$0.17$ we can achieve an availability of $A_{cu} = 99\%$. Moreover, we also use the Spot price history to calculate the average prices $A = [a_1, \ldots, a_n]$ per machine type $t_i \in T$ for the learning windows by dividing the history into intervals which have the same Spot price. The average billed prices instead of the bid prices are used to calculate the actual costs of a cluster configuration.

In order to find the optimal cluster configuration $C = [b_1, \ldots, b_n]$ which is a vector of bids (one for each machine type), *Spotgres* evaluates the constraint program as shown in Listing 1. A bid $b_i = [x_i, p_i]$ on a machine type i is a tuple which defines the number of instances x_i and the bid price p_i. The constraint program in step (3) returns the vector $X = [x_1, \ldots, x_n]$, whereas the bid prices $P = [p_1, \ldots, p_n]$ for each machine type are returned already by step (2).

In the following, we explain the details of the constraint program shown in Listing 1. The objective function obj

Listing 1. Constraint Program

```
Minimize:
 obj: a_1 · x_1 + ... + a_n · x_n + 0 · b_1 + ... + 0 · b_n

Subject To:
 cuConstraint: cu_1 · x_1 + ... + cu_n · x_n >= CU
 lowerBound_1: r_1 · b_1 - x_1 <= 0
 upperBound_1: MinInt · b_1 + x_1 <= 0
 ...
 lowerBound_n: r_n · b_n - x_n <= 0
 upperBound_n: MinInt · b_n + x_n <= 0

 (activeInstance_1:  b_1 = 1
 ...
 activeInstance_n:  b_n = 1)
```

calculates the total cluster costs based on the average prices per machine type. The variable b_i is a special variable which is set to $b_i = 1$ in order to indicate that machine type i must be included in the cluster configuration by adding the optional constraint $activeInstance\_i$ which is needed for re-bidding if some nodes in the cluster do not fail (see Section III-D). The first constraint (i.e., $cuConstraint$) defines that there must be in total at least CU compute units in the cluster where cu_i is the number of compute units for machine type i. The other constraints (i.e., $lowerBound\_i$ and $upperBound\_i$) represent bounds on the numbers of instances of machine type i. These constraints can be used e.g for re-bidding to make sure that instances that did not fail must be included in the new cluster configuration again.

C. Maximizing Availability

Compared to the first case (in Section III-B) where we minimize the total cluster costs, the second case has a maximal budget as an input constraint and the objective function is to maximize the availability per compute unit A_{cu}. Since there is no simple function that maps the average cost per machine type to its actual availability, we apply a binary search over the procedure shown in Section III-B. Thus, we start with $A_{cu} = 50\%$ and depending on the total cluster costs, we either increase or decrease the availability to $A_{cu} = 75\%$ or $A_{cu} = 25\%$ etc. until we find the highest value for A_{cu} that still satisfies the maximal budget B.

D. Optimizations and Variants

Diversity Optimization: One variant of the procedure in Listing 1 is that we add a diversity constraint as shown below:

```
diffTypesConstraint: b_1 + ... + b_n = k
```

This constraint can be used to force the solver to choose k different machine types for the cluster configuration. Together with the $lowerBoundi$ and $upperBoundi$ constraint the total number for each of the k machine types can be defined. The diversity constraint can be used for *Spotgres* cluster configurations where partitions should be replicated. The reason is that typically all instances of the same machine type fail at the same time when using Spot Instances (i.e., when the Spot price exceeds the bid price). Thus, replication only helps if data is copied to different machine types or the same machine type with a higher bid price. Currently, setting $k \geq 1$ only returns k different machine types in the cluster configuration. Returning the same machine type with a higher bid price is an avenue for future work.

Uptime Optimization: Another variant is an optimization that leverages the fact that a user is not charged for instances which have an uptime of less than an hour. Therefore, we use the cluster configuration we get from one of the procedures described in Section III-B and III-C and add/subtract a cost-tolerance to/from the given total cluster costs. We use that cost tolerance as a new constraint for the constraint program shown in Listing 1. Moreover, we change the objective function to maximize the number of intervals where the uptime is less than an hour. Thus, the main idea is to reduce the total cluster cost by adding the cost tolerance (i.e., in the worst case we are willing to pay a little more).

IV. Cost-based Fault-tolerance

In this section, we present a high-level overview of our cost-based fault-tolerance scheme. Compared to the existing fault-tolerance schemes, our scheme selects a subset of intermediates (called materialization configuration) to be materialized such that the query runtime is minimized under the presence of mid-query failures taking the mean time between failures (MTBF) per cluster node into account since the MTBF can strongly vary over time.

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

In order to find an optimal plan, cost-based optimizers typically enumerate different equivalent execution plans P and apply a cost function to find the best plan with the minimal runtime cost. However, they neither enumerate different materialization configurations for each plan 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 with minimal runtime under mid-query failures. 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 4): (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. 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

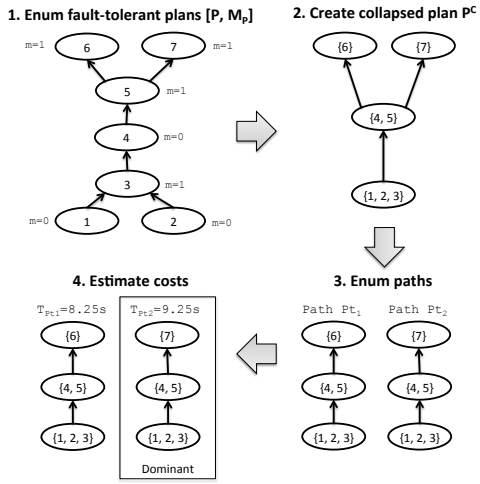


Fig. 4. Finding the best Fault-tolerant Plan

finished successfully it does not need to be re-executed again). (3) Third, for a collapsed plan, all execution paths (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 (called the dominant path of that configuration) is selected. The total cost of the dominant path represents the total runtime of the fault-tolerant plan. In Figure 4, for example, path Pt_2 is the dominant path and thus the estimated runtime for the given fault-tolerant plan $[P, M_P]$ would be $9.25s$. To that end, our procedure finds an execution plan P and a corresponding materialization configuration M_P which has minimal total costs under the presence of mid-query failures.

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, as well as the cluster size (i.e, the number of nodes). In this paper, we assume that all these parameters are given.

V. Experimental Evaluation

In this section we report the results of our experimental evaluation which shows (1) the effectiveness of our bid advisor for different input constraints and different objective functions (Section V-A), and (2) the effectiveness of our cost-based fault-tolerance scheme by comparing the runtime of queries over the TPC-H schema for different cluster configurations (Section V-B).

In both experiments, we use the Spot price history of the last 12 months ranging from 1st of September 2013 until 8th of August 2014 for all 18 machine types where the price history was available in that time frame. Moreover, we use the following resource constraints as an input to the bid advisor: $CU \geq 10$ and $RAM_{cu} \geq 1.5GB$. Adding more CUs would not change the results of the bid advisor since this has no effect on the optimization problem.

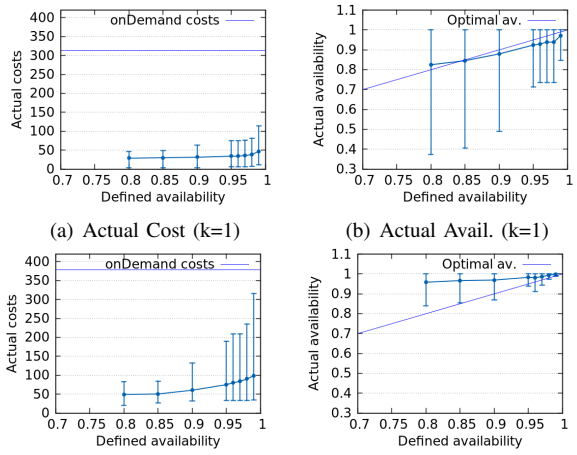
A. Effectiveness of Bid Advisor

In this experiment, we show the effectiveness of our bid advisor for two scenarios: Exp. 1a minimizes cluster costs for a given availability (Figure 5) and Exp. 1b maximizes the availability for a given budget (Figure 6). For finding an optimal cluster configuration, we use one month of the price history for learning and use the subsequent month for testing. We repeat each experiment multiple times by shifting the learning and testing window forward for one week until the testing window reaches the end of the Spot price history (mentioned before).

For Exp. 1a, we use the given availabilities of 0.8, 0.85, 0.9, and from 0.95 to 0.99 every 1% step. For Exp. 1b, we use the given prices of 1%, 5%, 25%, 40%, 70%, 90%, and 100% of the cheapest On-demand cluster that satisfies the given resource constraints (i.e., $CU \geq 10$ and $RAM_{cu} \geq 1.5GB$). Moreover, we execute each experiment (1a and 1b) using the diversity variant of our algorithm mentioned in Section III-D; using the diversities of $k = 1$ and $k = 3$ (i.e., a database without replication and one with replication factor 3). Moreover, for $k = 3$ we repeat Exp. 1a with the uptime optimization enabled to analyze for potential cost savings (Figure 7).

Exp. 1a - Minimizing Costs (Figure 5): Figure 5(a) and 5(b) compare the defined availability (i.e., the input constraint) and the actual availability per node as well as the defined availability and the actual total costs of the cluster (which was the objective function in this experiment) for $k = 1$. Both figures show the average as well as the the lower and upper quartile for the Spot cluster. For the actual total costs of the Spot cluster that we minimize in this experiment, we see that the average costs are below $\$50$ (compared to more than $\$300$ for the cheapest On-demand cluster). For the actual availability of the Spot cluster, we see that the average is very close to the defined availability (i.e, the input constraint). However, the lower and upper quartile show a high interquartile range (IQR).

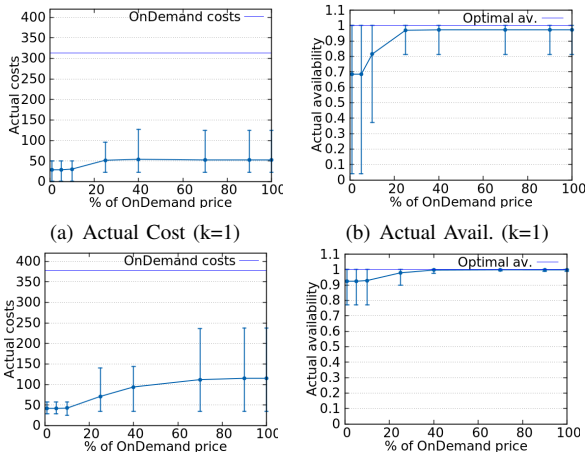
Figure 5(c) and 5(d) show the results for a diversity of $k = 3$. The positive effect of $k = 3$ is that the average actual availability gains when comparing to average actual availability of $k = 1$. In fact, the average of the actual availability is always higher than the defined availability and has a much smaller IQR. However, the average total costs for the Spot Cluster and the cheapest On-demand cluster are rising both about $\$50$. This is clear, since for $k = 3$ the bid advisor must pick 3 different machine types and can not compose the complete cluster only using the



(a) Actual Cost (k=1) (b) Actual Avail. (k=1)

(c) Actual Cost (k=3) (d) Actual Avail. (k=3)

Fig. 5. Exp. 1a - Minimizing Cluster Costs (Avgs and Quartiles)



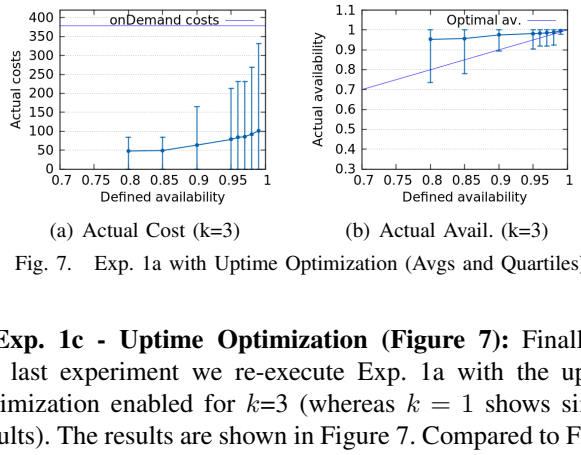
(a) Actual Cost (k=1) (b) Actual Avail. (k=1)

(c) Actual Cost (k=3) (d) Actual Avail. (k=3)

Fig. 6. Exp. 1b - Maximizing Availability (Avgs and Quartiles)

cheapest machine type. In the extreme case (i.e., for a defined availability of $A_{cu} = 0.99$) the total costs thus almost double for the Spot cluster (i.e., \$100 for $k = 3$ instead of \$50 for $k = 1$).

Exp. 1b - Maximizing Availability (Figure 6): In this experiment, we set the maximal budget of the user for the entire cluster as a constraint and analyze the actual costs of the Spot cluster as well as the actual availability of each node (which is the objective function in this experiment). If we set the budget to 25% of the cheapest On-demand cluster, we can see that we already get an actual average availability per node of more than 0.96 for $k = 1$ (see Figure 6(b)) and even 0.99 for $k = 3$ (see Figure 6(d)). Moreover, the actual costs do not rise higher than 20% (i.e., \$50) of the cheapest On-demand cluster for $k = 1$ respectively 30% (i.e., \$120) for $k = 3$ on average even if we set the maximal budget to 100% of the cheapest On-demand cluster. The reason is that the Spot price is typically much lower than the bid price if we set the maximal budget to 100% of the cheapest On-demand cluster. For the IQR, we see similar effects as for Exp. 1a.



(a) Actual Cost (k=3) (b) Actual Avail. (k=3)

Fig. 7. Exp. 1a with Uptime Optimization (Avgs and Quartiles)

Exp. 1c - Uptime Optimization (Figure 7): Finally, in the last experiment we re-execute Exp. 1a with the uptime optimization enabled for k=3 (whereas $k = 1$ shows similar results). The results are shown in Figure 7. Compared to Figure 5(c) and 5(d) (i.e., the same experiment without the uptime optimization), the most important difference is that the lower quartile of the actual costs in this experiment is at \$0 (i.e., in the best case we do not pay anything for our cluster).

B. Effectiveness of Cost-based Fault-Tolerance

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. 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 do 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 compared in this experiment, are:

- **all-mat:** This represents the strategy of Hadoop, where all intermediates are materialized to a fault-tolerant storage medium. 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 intermediates are not used to recover. Moreover, lineage information is used to re-compute failed sub-plans.
- **no-mat (restart):** This represents the strategy of a parallel database, where intermediates are not used to recover. Moreover, for recovering a coarse-grained strategy is used (i.e.,the complete query plan is restarted once a sub-plan fails).
- **cost-based:** This represents our cost-based strategy that materializes intermediates based on a cost-model. Moreover, for recovering a fine-grained strategy is used (i.e., only sub-plans that fail are restarted).

In the following, we report the overhead of all these strategies: (a) when running queries with varying runtime for a fixed MTBF to show the effect of the fault-tolerance

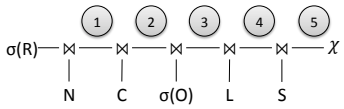


Fig. 8. TPC-H query 5 (Free operators 1-5)

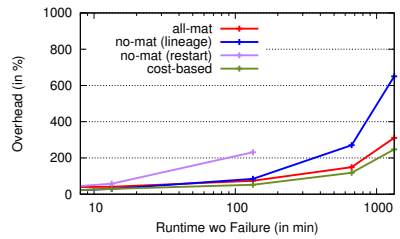
strategies short- and long-running queries and (b) when running the same query for different MTBFs to show the effect of different cluster setups. For measuring the actual runtime of queries under mid-query failures, we recorded 10 traces to inject failure using a poisson distribution per machine type with a given MTBF and used the same set of traces for all fault-tolerance schemes. Moreover, failures for nodes of the same machine type were correlated (i.e., all nodes of the same type failed together). In all experiments, we used a diversity constraint of $k = 3$. We used this method for all other experiments in this paper to inject failures.

Exp. 2a - Varying Query Runtime (Figure 9): In this experiment, we executed TPC-H query 5 over different scaling factors using parameters. 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 8). We also used other queries of the TPC-H benchmark but they showed similar results when varying their runtime.

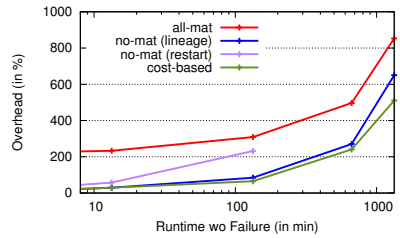
For this experiment, the output of every join operator was defined to be a free operator (marked with the numbers 1-5 in Figure 8) and thus could be selected by our cost-model to be materialized. Thus, for each enumerated plan, our procedure enumerated 2^5 materialization configurations when pruning was deactivated. Moreover, we injected mid-query failures using a MTBF of 1 day per node.

The result of this experiment is shown in Figure 9(a). The x-axis shows the baseline-runtime of the query (i.e., when no failure is happening) and the y-axis shows the overhead as discussed before. 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. Thus, our scheme effectively finds the best materialization configuration for different queries of different length. Compared to our cost-based scheme, the other schemes impose a higher overhead depending on the query runtime. Both no-mat schemes pay 0% overhead for short-running queries. However, for queries with a higher runtime, the overhead increases for both no-mat schemes whereas the restart-based scheme shows a much stronger increase. In fact, queries with a runtime higher than $7,820$s never finished in our experiment. The all-mat scheme pays 30% overhead for short-running queries which is pure materialization overhead (i.e., no failure occurred). However, for a higher baseline-runtime the overhead of this scheme does not increase as strong as for the no-mat schemes but still is 30% higher compared to overhead of the cost-based scheme.

Figure 9(b) shows the same experiment as before with increased I/O costs (i.e., materialization costs were approx. $7\times$ higher than those used in Figure 9(a))[1] The rationale of this experiment is to show the effect of slower external storage systems (such as HDFS) on the overhead of all schemes. As before, the cost-based scheme has the lowest overhead for all queries (i.e., from short- to long-running queries). However, the overhead for queries with a higher runtime is increasing stronger compared to Figure 9(a) since fewer intermediates are materialized and thus higher recovery costs need to be paid. Moreover, another difference is that the all-mat scheme has to pay the higher materialization overhead for all queries making this scheme unattractive in this setting. Finally, the two no-mat schemes (lineage and restart) are not affected by the increased I/O costs since they do not materialize additional intermediates.



(a) Normal I/O Cost



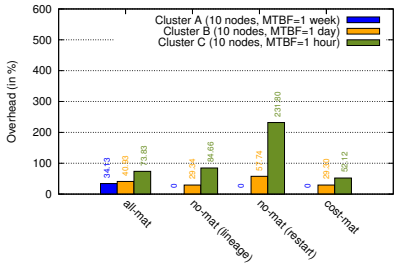
(b) High I/O Cost

Fig. 9. Varying Runtime

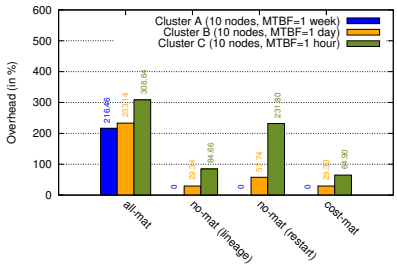
Exp. 2b - Varying MTBF (Figure 10): This experiment shows the overhead of the fault-tolerance schemes mentioned before when varying the MTBF. In this experiment, we executed TPC-H query 5 over $SF = 100$ using a low selectivity. This resulted in a query execution time of 905.33s (i.e., approx. 15 minutes) 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 when applying different fault-tolerance strategies and injected mid-query failures using the following MTBFs per node to cover a range of different failure rates: 1 week, 1 day, and 1 hour.

Figure 10 shows the overhead of the individual schemes under varying MTBFs. We executed this experiment for normal I/O costs (see Figure 10(a)) and high I/O costs where materialization costs were approx. $7\times$ higher as in the normal I/O case (see Figure 10(b)). Both figures show the same trends as already reported before in Figure 9(a) and Figure 9(b): The cost-based scheme has the lowest overhead for all MTBFs when compared to the other schemes using the same MTBF.

[1]We use different I/O rates of the EBS volumes.



(a) Normal I/O Cost



(b) High I/O Cost

Fig. 10. Varying MTBF

Both not-mat schemes are again independent of the I/O cost. Moreover, they do show a strong increase in the overhead for lower MTBFs (i.e., higher failure rates) compared to our cost-based scheme. As to be expected, the efficiency of the all-mat scheme depends mainly on the I/O costs. When having high I/O costs (Figure 10(b)), this scheme again has the highest overhead compared to other schemes since it pays the materialization costs for all operators independent of the MTBF.

VI. Related Work

Amazon EC2 Spot Instances: There exist different approaches on (cost-efficient) checkpoint techniques for Spot Instances [15], [16], [7]. Moreover, different approaches for bidding strategies have been proposed and evaluated [9], [10]. However, to the best of our knowledge there is no work that is particular targeting PDEs on Spot Instances nor has there been work on automatically finding the best cluster configuration for given user constraints on the budget and the availability as described in this paper.

Fault-tolerance Schemes: Fine-granular fault-tolerance scheme are typically found in modern PDEs (such as Hadoop [1], Shark[14], Dryad [6]) as well as in many stream processing engines [5], [11]. While stream processing engines checkpoint the internal state of each operator for recovering continuous queries, MapReduce-based systems [4] 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 results are persistent even when a node in the cluster fails, which requires expensive replication and prevents support for latency-sensitive queries. Other systems, like Impala [2] and Shark [14] 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 [17], which store their lineage in order to enable re-computation instead of replicating intermediates. However, all of these systems still materialize the output of each operator.

VII. Conclusions

In this paper, we present a novel PDE called *Spotgres* that can be deployed in a cost-efficient and reliable manner on Amazon's Spot Instances. *Spotgres* has two important features: (1) a constraint-based bid advisor which finds an optimal cluster configuration (i.e., a set of bids) based on a given set of constraints and (2) an cost-based fault-tolerance scheme that takes various parameters (such as MTBF and query statistics) into account to efficiently execute analytical queries under mid-query failures. Our experiments show that *Spotgres* is able to deliver the same query performance as a traditional PDE on an On-demand cluster with only 1% of the total costs.

References

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

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

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

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

[5] 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.

[6] 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.

[7] S. Khatua and N. Mukherjee. Application-Centric Resource Provisioning for Amazon EC2 Spot Instances. In *Euro-Par*, pages 267–278, 2013.

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

[9] S. Tang, J. Yuan, and X.-Y. Li. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance, booktitle = IEEE CLOUD. pages 91–98, 2012.

[10] S. Tang, J. Yuan, C. Wang, and X.-Y. Li. A Framework for Amazon EC2 Bidding Strategy under SLA Constraints. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):2–11, 2014.

[11] 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.

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

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

[14] 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.

[15] S. Yi, A. Andrzejak, and D. Kondo. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE T. Services Computing*, 5(4):512–524, 2012.

[16] S. Yi, D. Kondo, and A. Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *IEEE CLOUD*, pages 236–243, 2010.

[17] 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.

[18] 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.