2017

Analyzing Spark Performance on Spot Instances

Jiannan Tian
University of Massachusetts Amherst

ANALYZING SPARK PERFORMANCE ON SPOT INSTANCES

A Thesis Presented

by

JIANNAN TIAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2017

Department of Electrical and Computer Engineering

ANALYZING SPARK PERFORMANCE ON SPOT INSTANCES

A Thesis Presented

by

JIANNAN TIAN

Approved as to style and content by:

David Irwin, Chair

Russell Tessier, Member

Lixin Gao, Member

Christopher V. Hollot, Head
Department of Electrical and Computer Engineering

ABSTRACT

ANALYZING SPARK PERFORMANCE ON SPOT INSTANCES

SEPTEMBER 2017

JIANNAN TIAN
B.Sc., DALIAN MARITIME UNIVERSITY, CHINA
M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor David Irwin

Amazon Spot Instances provide inexpensive service for high-performance computing. With spot instances, it is possible to get at most 90% off as discount in costs by bidding spare Amazon Elastic Computer Cloud (Amazon EC2) instances. In exchange for low cost, spot instances bring the reduced reliability onto the computing environment, because this kind of instance could be revoked abruptly by the providers due to supply and demand, and higher-priority customers are first served.

To achieve high performance on instances with compromised reliability, Spark is applied to run jobs. In this thesis, a wide set of spark experiments are conducted to study its performance on spot instances. Without stateful replicating, Spark suffers from cascading rollback and is forced to regenerate these states for ad hoc practices repeatedly. Such downside leads to discussion on trade-off between compatible slow checkpointing and regenerating on rollback and inspires us to apply multiple fault tolerance schemes. And Spark is proven to finish a job only with proper revocation rate. To validate and evaluate our work, prototype and simulator are designed and implemented. And based on real history price records, we studied how various checkpoint write frequencies and bid level affect performance. In case study, experiments show that our presented techniques can lead to ~20% shorter completion time and ~25% lower costs than those cases without such techniques. And compared with running jobs on full-price instance, the absolute saving in costs can be ~70%.

TABLE OF CONTENTS

Page

CHAPTER

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Cloud computing has become an overwhelmingly effective solution to build low-cost, scalable online services (*Infrastructure as a Service* or *IaaS*). Providers such as AWS Elastic Compute Cloud (AWS EC2) [2], Google Compute Engine [3] and Microsoft Azure [4] manage large-scale distributed computing infrastructures and rent this compute capacity to customers. Compute capacity, abstracted from computing resource, storage, and network bandwidth, etc., is rented out as virtual server instance. There are situations when cloud providers have unused, active resources, and put their idle capacity up at a cleaning price to maximize revenue. Compared to those full-price instances, spot instances are much (usually 80%) cheaper for compromised reliability [2]. In the literature, the terms *spot instance*, *transient server*, *preemptible instance* have been used interchangeably to represent virtual server that can be revoked by the provider. In this paper, we will use nomenclature *spot instance* for simplicity. Spot instance allows customers to bid at any expected price [1]. The provider sets a dynamic base price according to the supply and demand of compute capacity, and accepts all the bids over the base price. On acceptance, customers who bid are granted those instances. On the other hand, if later the base price exceeds that user's bid, those instances are revoked by the provider.

In nature, spot instance cannot compete with always-on instance in sense of QoS; such a fact forces customers put non-critical background jobs on spot instances. Among multiple QoS metrics, particularly *availability* and *revocability* are the main concern. Availability is defined as the ratio of the total time a functional unit is capable of being used during a given interval to the length of the interval [18]. In comparison, revocability indicates whether a spot instance is revoked under certain circumstance. For instance, if there are high-rate price alteration in a short time, the high availability can still exist, however, revocation numbers can be large. Moreover, revocation can be severe and abrupt; in a short period, the amplitude of the price change can be large and the price does not rise gradually. And spikes can be extensively observed in figure of price history. In our concern, working against *revocability* of spot instances while most prior work focuses on availability as indicated in Section 3.

On revocation, all the data and application that are deployed on instances are lost permanently. This incurs overhead from not only downtime, restart time but time to recover from loss and rollback as well. Therefore, job completion time increases when using spot instances. Rising bid effectively decrease the possibility of hitting base price and hence rate of instance revocation. Such a cost-reliability trade-off can lead to some sophisticated bidding strategy to minimize the total resource cost. On the other hand, with software supported fault tolerance schemes, the job completion time can also be minimized.

To seek feasibility of complete jobs on spot instances in decent time, we deployed Spark and utilized its fault tolerance mechanism. Unlike checkpoint, Spark does not recover from disk snapshot by default; nor does it recovers from duplicate memory states that are transferred to other networked machines before failure. On submission of application, Spark yields a list of function calls in order from the program code and hosts it on the always-on driver node. Such a list is called *lineage* and is used for task scheduling and progress tracking. An implication is that when the current job is interrupted, intermediate states are lost but regenerated in order according to the lineage. Such a rollback, if there is no other supplementary fault tolerance mechanism in use, can hit the very beginning of the lineage. With lineage-based recomputing, Spark would handle occasional interruption well [29], however, revocation triggered node failure is much more frequent, and Spark is not specifically designed for such an unreliable computing environment. Theoretically, if rollback to the very beginning occurs can possibly make the job exceed timeout and never end. This brought about the first question that leads to the thesis: what is the impact of node revocation on Spark job completion time and what are factors that affect performance?

To alleviate painful repeated rollbacks, we applied compatible checkpoint mechanism on Spark. By default, checkpoint is not utilized due to overhead from I/O operation between memory and low-speed disk; if there is no interruption, routine checkpoint write does nothing but increase the job completion time. However, by dumping snapshot onto disk and later retrieving to the working cluster, checkpoint makes it possible that job continues at the most recently saved state, and this would benefit those long jobs even more. Therefore, trade-off lies between routine checkpoint write overhead and painful rollback. A question emerges naturally: is there optimum that minimizes job completion time? Noticed that the optimization is based on natural occurrence failure that approximately satisfies Poisson Distribution, and it is different from that of market-based revocation. So the question is that whether the mechanism still works on *spot market* where instances are bid. These questions lead to the thesis. Contributions of this thesis are listed below.

- *Effectiveness experiment* is designed based on prototype Spark program. It proves the effectiveness that Spark cluster can get over frequent revocations. We tested 10, 20, 30 and 60 seconds as *mean time between node number alteration* (MTBA), and we found cases with MTBA above 30 seconds can meet time restriction to recover.

Noticed that this MTBA is much less that price change (not necessarily making node revoked) from the spot market.

- *factors* from the cluster configuration and job property are discussed since they may affect Spark performance. They are namely partition number, job iteration number, and mean time between node number alteration. We figured out that higher partition degree leads to less processed partition loss and hence shorter recovery time. And as is pointed out, shorter MTBA impacts on complete time more. And longer task suffers even more for the recovery process is even longer than those short jobs.

- *Mixed fault tolerance scheme* is developed and extensively discussed. With the inspiration of optimal checkpoint write interval in single-node batch-job case, we found that such optimum is valid for distributed MapReduce job. Noticed that in both cases revocation occurrence satisfies Poisson Distribution. In later case studies, we can see that checkpointing with proper optimal interval according to different market information can help lower costs when using spot instances.

- *Analytic Experiments* based on real price history (A collection of example price history records are hosted on the repository of this project [5].) are conducted. To validate and evaluate our work, prototype and simulator are designed and implemented. We studied how various checkpoint write frequencies and bid level affect performance. Results from experiments show that our presented techniques can lead to ~20% shorter completion time and ~25% lower costs than those cases without such techniques. And compared with running jobs on full-price instance, the absolute saving in costs can be ~70%.

CHAPTER 2

BACKGROUND

2.1 Spot Instance

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable computing capacity in unit of *instance* . Amazon EC2 provides a wide selection of instance types to meet different demands. There are three basic pricing models for instances from Amazon EC2: Reserved Instance, On-demand Instance and Spot Instance.

- Reserved instances allow customers to reserve Amazon EC2 computing capacity for 1 or 3 years, in exchange for up to 75% discount compared with On-demand (full-price) instance pricing.

- On-demand (hereinafter interchangeable with full-price) instance is more flexible. Customers pay for compute capacity by the hour so that they can request instance when instances are needed.

- Spot instances allow customers to bid on spare compute capacity at discounted price. Customers pay willingly any price per instance hour for instances by specifying a bid.

Spot instance can be acquired when there are idle instances from Reserved and On-demand pools. Since the performance of spot instance is equivalent to that of full-price instance, customers can save a lot on performance-thirsty required jobs. The provider sets dynamic spot price for each instance type in different geographical and administrative

type	**Reserved**	**On-demand**	**Spot**
price	high, w/ discount	high	low
volatility	N/A	N/A	high
availability	guaranteed	not guaranteed	not guaranteed
revocability	N/A	N/A	when underbid

Table 2.1: Cost-availability trade-off among instance pricing models

zone. Customers bid at desired price for spot instances. If a customer's bid is over that base price, the customer acquires the instances. On the other hand, if later spot price goes up and exceed the original bid, the customer's instances are revoked and permanently terminated. In consequence, hosted data and deployed applications are lost, and job suffers from rollback. If bid is risen, customers are more safe to meet less revocations and hence shorter job completion time. We can see that in exchange for low cost, the reliability of spot instances is not guaranteed. Table 2.1 shows comparison of instance pricing models.

2.1.1 Spot Market

Spot market is a fair market where the provider and customers mutually agree on the service price above an base price. The base price fluctuates according to supply and demand. Spot price ranges from 0.1x to 10x full price of the same instance type. On rare occasions, although it goes over 1.0x full price, it is far below 1.0x on average. Despite of the average low price, the price change can be severe; price change abruptly to a high level and fall to a rather low level in a short period (short enough so that a job cannot even be finished).

Table A1 in Appendix shows pricing for On-demand (full-price) instance in `east-us-1` as of year 2014. and Table A2 in Appendix chapter shows pricing for newly released fixed-duration as complementary pricing model.

types		mean	3^{rd}	5^{th}	10^{th}	25^{th}	median	75^{th}	90^{th}	95^{th}	97^{th}
c3	**large**	0.179	0.159	0.160	0.161	0.165	0.170	0.176	0.187	0.198	0.210
	xlarge	0.207	0.165	0.167	0.170	0.177	0.191	0.214	0.252	0.292	0.329
	2xlarge	0.232	0.181	0.184	0.189	0.202	0.221	0.250	0.287	0.312	0.339
	4xlarge	0.251	0.168	0.172	0.178	0.191	0.214	0.254	0.327	0.417	0.498
	8xlarge	0.215	0.162	0.163	0.166	0.172	0.185	0.208	0.247	0.281	0.326
d2	**xlarge**	0.172	0.103	0.103	0.103	0.106	0.160	0.205	0.259	0.305	0.341
	2xlarge	0.130	0.105	0.106	0.107	0.112	0.121	0.132	0.145	0.173	0.205
	4xlarge	0.126	0.103	0.103	0.104	0.105	0.109	0.122	0.156	0.194	0.226
	8xlarge	0.122	0.102	0.102	0.103	0.104	0.108	0.129	0.145	0.173	0.181
g2	**2xlarge**	0.197	0.126	0.129	0.134	0.148	0.175	0.215	0.267	0.307	0.353
	8xlarge	0.355	0.151	0.160	0.174	0.201	0.269	0.385	0.651	1.000	1.000
i2	**xlarge**	0.123	0.100	0.101	0.101	0.104	0.115	0.140	0.152	0.160	0.167
	2xlarge	0.125	0.103	0.103	0.104	0.108	0.118	0.133	0.148	0.159	0.169
	4xlarge	0.139	0.103	0.104	0.104	0.106	0.115	0.147	0.185	0.205	0.218
	8xlarge	0.122	0.101	0.101	0.102	0.103	0.107	0.129	0.156	0.161	0.169
m3	**medium**	0.156	0.131	0.131	0.134	0.139	0.148	0.169	0.185	0.200	0.210
	xlarge	0.164	0.138	0.140	0.144	0.151	0.161	0.172	0.185	0.196	0.206
	2xlarge	0.170	0.139	0.141	0.145	0.154	0.166	0.180	0.198	0.212	0.224
	large	0.151	0.132	0.133	0.135	0.138	0.144	0.154	0.175	0.199	0.218
r3	**large**	0.129	0.100	0.101	0.102	0.106	0.114	0.128	0.150	0.179	0.210
	xlarge	0.186	0.104	0.106	0.112	0.126	0.147	0.191	0.284	0.379	0.474
	2xlarge	0.168	0.111	0.114	0.119	0.131	0.151	0.183	0.227	0.268	0.303
	4xlarge	0.145	0.099	0.100	0.102	0.107	0.117	0.140	0.192	0.267	0.344
	8xlarge	0.165	0.112	0.114	0.119	0.130	0.151	0.181	0.218	0.256	0.288

Table 2.2: Mean, median spot price and other percentiles in 90 days

2.1.2 Market Volatility

Same-type instances are priced approximately the same across different geographical regions. Here we take *us-east-1* as example to analyze on spot market volatility in the Unites States.

Instances are differentiated by purpose, e.g. general-purpose, memory-optimized for intensive in-memory computing, and GPU-optimized for graph algorithms and machine learning. For full-price instances, all same-purpose instances are price the same for unit performance. A unit performance is defined by price per *EC2 Compute Unit* (ECU), and it can be represented alternatively as ratio of spot price to full price. So we adopted this ratio as standardized price to measure the spot price as illustrated in Equation 2.1.

$$\mathit{ratio} = \frac{\text{spot price}}{\text{on-demand price}} = \frac{\text{spot price/ECU number}}{\text{OD price/ECU number}} = \frac{\text{spot price per ECU}}{\text{OD price per ECU}}, \tag{2.1}$$

where full-price is fixed for each type.

Due to supply and demand, the ratio for same-purpose instance can be different. An example of comparison between `m3.medium` and `m3.xlarge` is shown in Figure 2.1. On bidding strategies, we may bid for several small instances or a single large instance delivering the same performance. Which to bid may depend on the granularity to which a job is partitioned. And it is related to Section 3.2. This brings forth a critical question: high revocation rate causes cascading node failure and data loss, is it even feasible to deploy application even with abundant fault-tolerant mechanisms? This leads to observation on volatility of the market. Although this can lead to a sophisticated bidding strategies, in this paper, we are not going to discuss further on this.

We also gave a general comparison among all instance types in Figure 2.2. In spot market, bidding level determines availability. To give an intuitive view over availability, we supposed in the past three months, we bid for each type of instance at exactly the mean

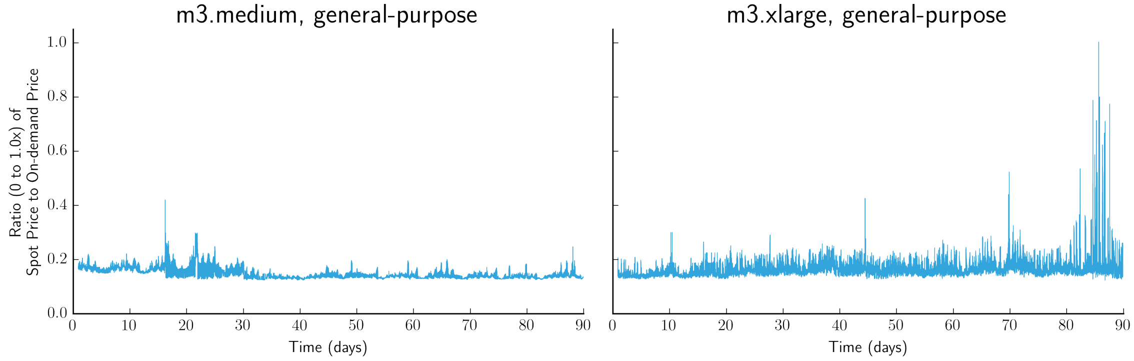


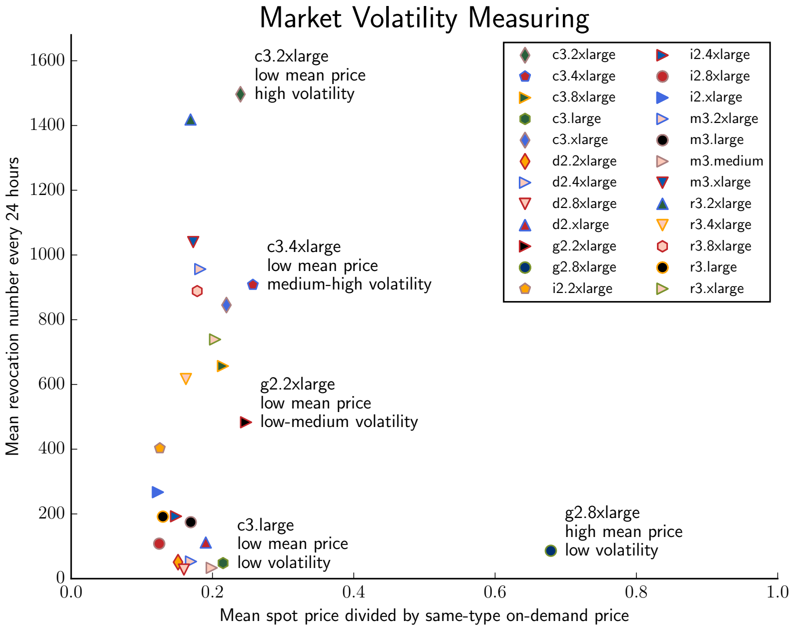
Figure 2.1: Price history comparison of `m3.medium` and `m3.xlarge`

price and count revocation number; thus revocation rate due to underbids can reflect the spot market volatility. We defined revocation rate as revocation number per 24 hours. (only records in most recent three months can be retrieved from official source; however, 3rd-party communities maintain much longer history).

Figure 2.2 shows widely distributed bid-revocation information. In this Figure, X-axis is given by mean spot price during 90 days (in this project, it is March 13 to June 13, 2016), and the data is standardized as ratio of spot price to full-price. Y-axis is given by mean revocation number every 24 hours when bid level is set to the aforementioned mean price. As we can see, most instance types (`g2.8xlarge` type is the only exception in this study) are lowly priced but revocation rates are widely distributed. We can take `c3.2xlarge`, `c3.4xlarge`, `g2.2xlarge` and `c3.large` as examples.

2.1.3 Alternative Service

Preemptible instance from *Google Compute Engine* (GCE) is an alternative option of the spot instances. Customers also create and run virtual machines on its infrastructure [3]. GCE might terminate (preempt) these instances if it requires access to those resources for other tasks, although pricing is not auction based (fixed instead). Additionally, Compute Engine has a finite number of available preemptible instances, so customer might not be



type	mean price	revoc. rate
c3.large	0.215	48.1
c3.xlarge	0.220	845.2
c3.2xlarge	0.240	1496.5
c3.4xlarge	0.257	907.9
c3.8xlarge	0.215	656.8
d2.xlarge	0.191	111.6
d2.2xlarge	0.151	51.0
d2.4xlarge	0.170	52.9
d2.8xlarge	0.160	28.1
g2.2xlarge	0.248	483.1
g2.8xlarge	0.679	86.2
i2.xlarge	0.123	267.1
i2.2xlarge	0.126	403.0
i2.4xlarge	0.148	192.7
i2.8xlarge	0.125	108.1
m3.medium	0.199	33.3
m3.large	0.169	174.5
m3.xlarge	0.173	1039.8
m3.2xlarge	0.183	956.3
r3.large	0.130	191.5
r3.xlarge	0.204	739.0
r3.2xlarge	0.169	1418.5
r3.4xlarge	0.162	616.7
r3.8xlarge	0.178	888.5

Figure 2.2: Market volatility comparison

able to create them during peak usage [15]. Comparison of AWS Spot Instance and GCE preemptible instance is listed in Table 2.3.

provider	**AWS Spot Instance**	**Preemptible Instance**
pricing	fluctuating, bidding required	fixed
condition of yielding	bidding failure	preempted by higher high-priority tasks
on yielding	instance terminated	(same) instance terminated

Table 2.3: Comparison of Spot Instance and Preemptible Instance

2.2 Spark the Framework

Apache Spark is a general-purpose parallel-compute framework that supports extensive data processing primitives. Spark Core, a collection of core functionality, drives high-level applications. There is an optimized engine that supports general execution graphs, *Spark SQL* for SQL and structured data processing, *MLib* for machine learning, *GraphX* for graph processing, and Spark Streaming. Spark structure is shown in Figure 2.3.

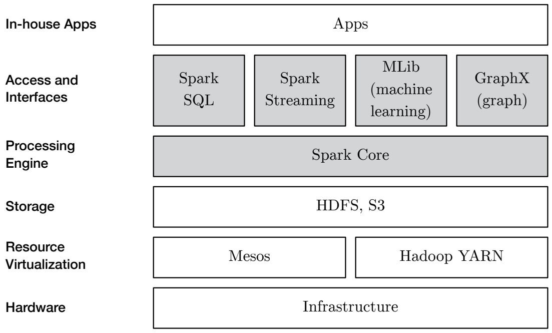


Figure 2.3: Spark cluster components

In this paper, we focus on designing programs with primitives from Spark Core. These primitives are classified into two categories, *transformation* and *action*. A complete list of transformation and action is shown in Table A2.

2.2.1 In-memory Computing

Traditional *Hadoop Distributed File System* (HDFS) is an abstract distributed file system primarily for managing data. Although HDFS is primarily for Hadoop application, it is ubiquitously used by distributed frameworks. Due to the fact that for read operation is much frequent than write operation, it is designed write-once-many-access feature for simple coherence and derived intermediate states are written back to disk. For those applications that mainly work over data access rather than data write, HDFS contributes high throughput; however, it is against the nature of those applications that generate vast of intermediate results. Particularly, when it comes to iterative tasks, it incurs severe overhead of swapping transient states out and in to low-speed storage, thus it deteriorates the overall performance.

Spark incorporates popular MapReduce methodology. Compared with traditional Hadoop MapReduce, Spark does not write intermediate results back to low-speed disk. Instead, Spark maintains all necessary data and volatile states in memory.

2.2.2 Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is the keystone data structure of Spark. Partitions on Spark are represented as RDD. By default, necessary datasets and intermediate states are kept in memory for repeated usage in later stages of the job. (Under rare circumstance, with insufficient physically memory, in-memory states are swapped out onto low-speed disk, resulting in severely downgraded performance.) RDDs can be programmed *persistent* for reuse explicitly, such an operation is *materialization*; otherwise, RDDs are left *ephemeral* for one-time use.

On job submission to Spark, the program code is unwound and recorded as a list of procedural function calls, terminologically *lineage*. On execution, lineage is split into stages. A stage can start with either a *transformation* or an *action*. A *transformation* literally transform a type of data hosted in RDD into another type in RDD while an *action* in the end output data in regular types that are not used for in-memory computing. With syntactical support of lazy evaluation, Spark starts executing *transformation* operations only when the program interpreter hits *action* after those *transformations*. Such a scheme is used for scheduling and fault tolerance (see details in Section 2.3). *Scala* programming language [14] is used to call function in Spark program.

2.3 Fault Tolerance

2.3.1 Recomputing from Lineage

Consistent with in-memory computing, fault tolerance is accomplished by utilizing lineage as preferred. To simplify question, Spark driver program is hosted on supposedly always-on instance. Thus lineage generated in driver program is never lost and fault tolerance system can fully work towards recovery.

On node failure, volatile states in memory are lost. Rather than recover from duplicate hosted on other machine before failure, this part of lost node can be computed from other states; specifically, it can be generated from original datasets. With progress tracked in lineage, recovery can start from the very beginning of the lineage and finally reaches the failure point. Programmatically, Spark supports recomputing from lineage and checkpoint mechanism. And these are discussed in Section 2.3.3 and 2.3.4. Multiple fault tolerance mechanisms and schemes are also compared in Section 3.3.

2.3.2 Node Failure Difference

There are several differences lying between natural node failure in datacenter and revocation triggered failure.

- in industry, *mean time to fail* (MTTF) are used measure failure interval in unit of hundreds of days, which is much longer (10,000x) than interval for a price change thus potential revocation.

- natural node failure occurrence obeys non-memorizing distribution. In the single-node case, Poisson Distribution is reasonable approximation. However, there is no evidence showing that revocation triggered node failure obey such distribution.

- Spot prices fit in to Pareto and exponential distributions well [32] while revocation distribution is more complex for different bidding schemes.

Some sophisticated bidding strategies [32, 23] are derived. While some argued there is no need to bid the cloud [24, 26] for different reason (see details in Section 3.2). We focus on invariant in running Spark job on spot instances no matter how we bid the cloud.

2.3.3 Naïve Fault Tolerance Scheme

Recomputing from lineage makes it possible to recover from failure without external backups. However, the effectiveness of the exploiting recomputing scheme is undetermined. There are some positive factors from the cluster configuration that help recover.

- data storage and application are deployed differently. Data is hosted on HDFS cluster other than the compute cluster, or hosted in S3 bucket.
- it is inexpensive and preferred to deploy driver program on a single always-on node to avoid lineage loss.

More related cluster configuration is listed in Section 4.1.

However, there many negative factors that undermines the recovery severely.

- Revocation is much more frequent than natural node failure in datacenter, and
- Despite the strong resilience of Spark (recovering when there is only small number of nodes in the cluster), revocations in sequence applies cascading state losses on the cluster, making it even harder to recover.

A fault tolerance scheme is application with specified parameter of its cornerstone mechanism. Compared to natural node failure, this fault tolerance mechanism is not designed for high failure rate. It is highly possible to exceed system-specified timeout, and the job is terminated. This leads to a later effectiveness experiment stated in Section 4.2. As we pointed out later, although it is not guaranteed to complete job without exceeding timeout, we can cut off those timeout tasks by configuring mean time between failure.

2.3.4 Checkpoint

Compatible checkpoint write is disabled in Spark by default for performance consideration. This supplemental mechanism can be enabled both in program code and configuration. Technically, RDD can be differentiated by storage level (see details in Table A1). By default, `MEMORY_ONLY` is preferred to use to achieve better performance. Flexible on-disk materialization for specific RDDs can be done by programming rather than hard-setting `ON-DISK` for all RDDs. On job failure, disk-cached states will be immediately ready after loading. This alleviate cascading rollbacks and recompute from beginning. However, if there is no failure, routine checkpoint write is wasteful, only to extend job completion time. This motivate us to utilize mixed fault tolerance scheme.

2.3.5 Mixed Fault Tolerance Scheme

As discussed earlier, we can balance overhead of routine disk write and rollback. This arise the second question, what the optimum of checkpoint write interval is if any. Inspired by single-node batch-job case, we applied a first-order approximation on finding optimum of checkpoint write interval to minimize the total job completion time. The evaluation is shown in Chapter 6.

CHAPTER 3

RELATED WORKS

This thesis focuses on analyzing the performance and cost of running distributed data-intensive workloads, such as Spark jobs, on transient servers, such as AWS Spot Instances and GCE Preemptible Instances. Below, put our work in the context of prior work that has examined a variety of bidding strategies and fault-tolerance mechanisms for optimizing the cost and performance on such transient servers.

3.1 Cloud Computing

There are several topics related to cloud computing infrastructure.

- *In-memory computing* Data reuse is common in many iterative machine learning and data mining [29]. Pessimistically, the only way to reuse before computations is to write it to external stable storage system, e.g. HDFS [8]. Specialized frameworks, such as Pregel [21] for iterative graph computations and HaLoop [9] for iterative MapReduce, have been developed. However, these frameworks support limited computation patterns. In contrast, Spark is general-purposed and offers primitives for data processing. The abstraction for data reuse as well as fault tolerance is (RDD). Materialization can be toggled by programming in sense of data reuse with the support of RDDs. In the programmed application, a series of data processing procedure along with explicit materialization of intermediate data is logged as lineage. Such a setting lead to quick recovery and does not require costly replication [29].

- *Multi-level storage* Although materialization of reused data boosts performance, node loss annihilates such efforts and makes it useless on high-volatile cluster. In our work, we took a step back. We took advantage of multiple storage level (see Table A1); not only low latency in the process but the global minimizing completion time is the goal. To resolve such issue, we employ checkpointing along with built-in recovery form other RDDs. Despite the fact that overhead from disk-memory swapping is introduced again, we leverage its short recovery and avoidance of recompute from very early stage of a logged lineage.

- *Practice* In-memory computing requires abundant memory capacity in total. Spark official claimed that the framework is not as memory-hungry as it sounds and the needed original datasets are not necessary to loaded into memory instantly; in addition, multiple storage level, including memory and/or disk and the mixed use of them, can be configured to resolved the issue of materialization required capacity [6]. It could be true if base memory capacity is satisfied when the cluster node availability is stable, however, when node availability is low, performance suffers from both the limited memory capacity and memory state loss such that swapping in and out happens frequently and thus latency becomes much more serious. Such overhead is also discussed in Chapter 6.

3.2 Bidding the Cloud

Spot price alteration reflects and regulates supply and demand. This is proven and discussed further in [10]: for the provider, it is necessary to reach market equilibrium such that QoS-based resource allocation can be accomplished.

- *Strategic bidding* Zheng et al. [32] studied pricing principles as a critical prerequisite to derive bidding strategies and fit the possibility density function of spot price of some main types by assuming Pareto and exponential distributions. Such fitting helps predict future spot prices. He et al. [16] implemented a scheduler for bidding and migrate states between spot instances and always-on on-demand instances.

Analysis in [22] shows the sensitivity of price change; a small increase (within a specific range) in bid can lead to significant increment in performance and decrement in cost. Though the sensitivity to price is also observed in our experiment (as shown in Chapter 6), it is more than aforementioned reason. 1) qualitative change occurs when bid is slightly increased to the degree where it is above price in most of time. And scarcely can revocation impact on performance and thus total cost; instead the dominating overhead is from routine checkpoint write to disk. 2) on the other hand, when bid is not increased high enough to omit most of revocations, a dramatically high performance is accomplished by much less rollback when checkpointed at appropriate frequency.

- *Not bidding* Some argued not biding is better without knowing the market operating mechanisms deeply. Not developing bidding strategies can be attributed to several reasons: 1) Technically, IaaS providers can settle problem of real-time response to market demand [33], and short-term prediction is hard to achieve, 2) customers can always find alternative instances within expected budget [24] for market is large enough, 2) there are abundant techniques that [25, 24] ensure state migration within the time limit and 3) some pessimistically deemed that it is not even effective to bid the cloud since cascading rollbacks caused by revocation is so painful to recover from and framework improvement is the key point to solution [26].

3.3 Fault Tolerance

Bidding strategy is helpful and we need specified bidding schemes to conduct experiments and to compensate less effective bidding strategies, we fully utilized fault tolerance mechanisms to archive equivalent effectiveness. And despite of intention of not bidding the cloud, we set different bid levels for 1) it is related performance and sometime performance is sensitive to the corresponding availability, and 2) data-intensive MapReduce batch jobs has been studied in [20, 16, 11]. Our part of job is not the traditional MapReduce with static original datasets that is pre-fetched and processed, rather some job does not really rely on old intermediate states, i.e. streaming, although QoS is not guaranteed.

Most of the prior work focuses on improving availability and thus QoS by developing bidding strategies. Nevertheless, higher availability does not necessarily result in low revocation rate. Yet Spark is employed to process data-intensive jobs, high-rate price alteration may lead to high revocation rate. There are several main fault-tolerance approaches to minimize impact of revocations (i.e. intermediate state loss and progress rollback): checkpointing, memory state migration and duplicate and recomputing from original datasets.

- *Live migration/duplication* Prior work of migration approaches is presented in [24, 25]. And fast restoration of memory image is studied in [31, 19]. In contrast, our origin working dataset is hosted on always-on storage while intermediate is mostly generated online for ad hoc practices expect the checkpointed portion to avoid overhead from network [30]. And these static integrity, i.e. integrity is ensured due to complete duplication differs from freshly regenerated intermediate states. Such difference lead to our investigation on more than checkpointing schemes.

- *Fault tolerance schemes* Checkpointing for batch jobs [12, 13] and its application on spot instances [27] are studied. We adopt the origin scheme into distributed case and mixed use of both checkpoint read and regeneration.

  [28] gives four basic and various derived checkpointing schemes with mean price bidding. In our work, mean price bidding is only used for illustrating market volatility(see Section 2.1.2); yet mean price bidding is not key to optimize. Listed basic checkpointing schemes includes hour-boundary, rising edge-driven, and adaptively deciding checkpointing. Results from [28] shows empirical comparison among cost-aware schemes; however, 1) before extensive discussion on other three basic methods, hour-boundary checkpointing can still be deeply investigated by changing checkpoint write interval and 2) for different bidding-running cases, the optimal checkpoint write interval can be different, which implies routing checkpoint write of variable interval can be employed; such a method along with its derived variable-interval checkpoint write can be effective while maintaining its simplicity.

  In addition, compared to [20, 16, 11] where given grace period of 2 minutes is used for live migration, in our case, the grace period is mainly used to finish writing checkpoint to external HDFS. (Otherwise, even the next stage can be finished, it is lost in the next moment.)

CHAPTER 4

DESIGN

4.1 Cluster

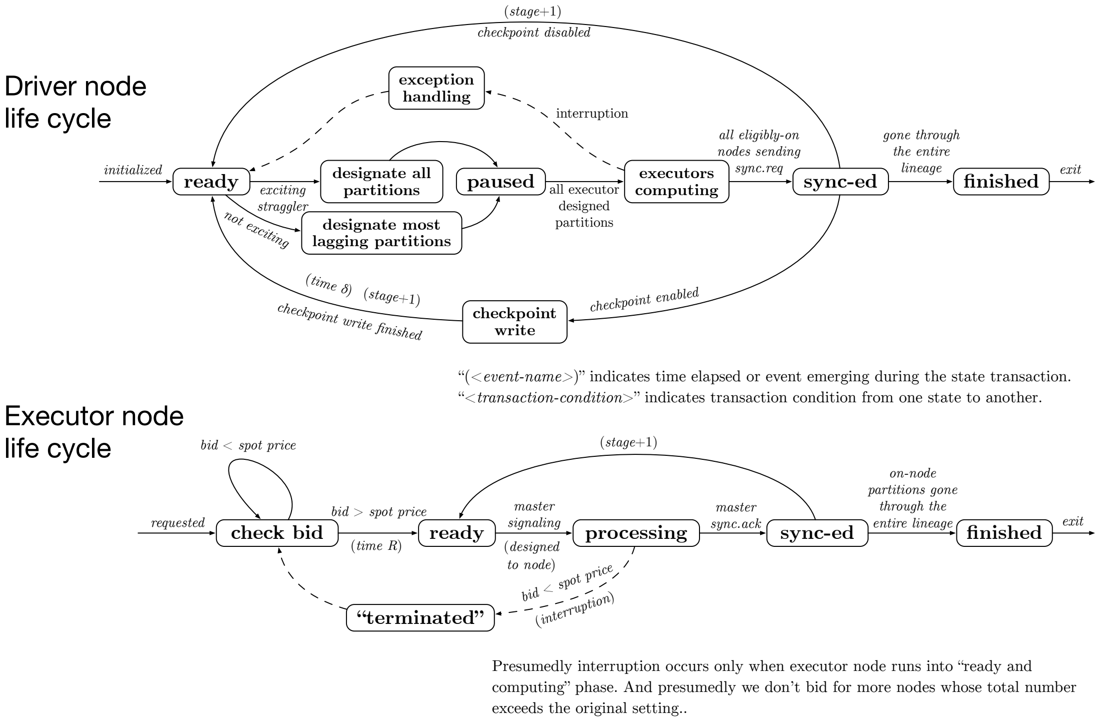
Suppose we choose a cluster of nodes from a node pool. And this cluster comprises a single master node (driver node) and multiple slave nodes (executor nodes). Via control panel we can control over the cluster in the remote datacenter. Noticed that a node registered under a framework can be easily replaced since compute capacity is ubiquitously multiplexed and we can always migrate workload from one to another [17]. Before we run Spark jobs on instances and recover job from failure, we first figured out how driver and executor nodes work in the cluster.

4.1.1 Driver Node Life Cycle

Driver node goes with the cluster until the cluster is terminated or expires. The driver node handles 1) partition designation as well as balance workload throughout the cluster, 2) catching exceptions catch, 3) recovering from node failure, 4) issuing checkpoint write if appropriate, and 5) synchronizing progress through all the executor nodes. Spark driver node life cycle is depicted in Figure 4.1.

4.1.2 Executor Node Life Cycle

As we can see, after acquiring the executor node once its bidding is over the threshold price set by the service provider. After being acquired, executor node is under control of driver node and is to be designated workloads. If there is no interruption caused by underbid, the node runs and finally exits peacefully; otherwise it is terminated and its alternative is requested to the cluster. Executor node life cycle is depicted in Figure 4.1.



“(<*event-name*>)” indicates time elapsed or event emerging during the state transaction.
“<*transaction-condition*>” indicates transaction condition from one state to another.

Presumedly interruption occurs only when executor node runs into “ready and computing” phase. And presumedly we don’t bid for more nodes whose total number exceeds the original setting..

Figure 4.1: Life cycles of nodes in cluster

4.1.3 Job Classification

Real-world jobs can be roughly classified into two categories:

1. Iterative MapReduce application as an example is one kind; when executed on Spark cluster, stages are inter-dependent since input for a stage is always the output from previous stage. Obviously in such cases, the all the intermediate and final results can be attributed to the first stage and the very input datasets. In this way, if a revocation occurs, all the active nodes are paused until the lost intermediate are generated from the very beginning.

2. Unlike stage-interdependent tasks, when the node number decreases, there is no need to start over, rather, old lost RDDs is simply not needed any more; instead, the processing capacity shrinks. A good example would be streaming; although there is no iteration that forms a stage, streaming often comes with data retrieving and analyzing online, which could be coded into transformations and actions.

4.1.4 Cluster Prototype

We built a prototype dynamic cluster whose node number always changes. A specific number of full-price (always-on) instances to ensure full control over the node availability. Cluster can be manipulated via control panel such that Spark executor processes are manually terminated and restarted on need basis. Such a design simulates node loss and new node requests in the spot market.

Suppose Spark runs under periodic pattern of fluctuating node availability. And such a given pattern is discretized to fit in to integer node number (see Figure 4.2). Thus job completion time in such a dynamic cluster can be observed and compared to that in static cluster with no node number change. The sample rate determines *mean time between mandatory pattern alteration* and the *interval* is defined as a unit time. Noticed that in a periodic pattern, there are two phases, 1) on ascending phase, new nodes are added and 2) on descending phase, nodes are revoked. So shrinking MTBA can either boost computing (on ascending phase) or deteriorate node loss even more and vice versa. In later results (see Section 6.2), we can see that MTBA is key parameter and may determine whether Spark can survive cascading/consecutive revocations or not.

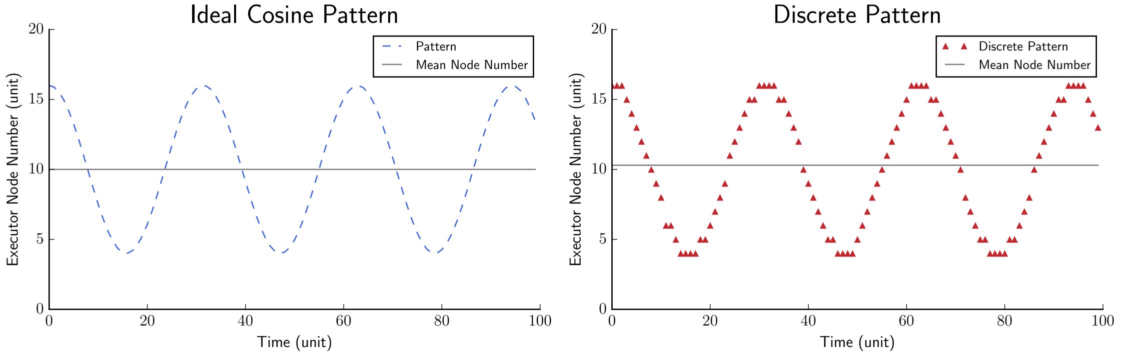


Figure 4.2: Pattern to apply on Spark cluster

4.2 Effectiveness Experiment

We conduct experiments to prove that it is possible to run Spark job in decent time with proper parameters. Noticed that number of data partitions, or RDD, are constant from the view of the system; rather than in a queue to be designated on new nodes, these RDDs are crammed on existing active nodes. For discussing effectiveness and more details, the amplitude, cached RDD number, and mean time to fail are manipulated. We hard-set some factors to reasonably simplify the problem (see Table 4.1). And we conduct experiments over parameters that listed below.

4.2.1 Amplitude

Amplitude of pattern is a direct parameter that impacts. We first set a (10 ± 6)-node dynamic cluster, which in long term average node number is 10. A stage holds 0+ transformation and 1+ action calls; recall that lazy evaluation lying in the scheduling basis and RDD, if lost, is regenerated from the lineage back to a specific stage (need action to trigger). Thus with the cached and to-be-regenerated RDD number constant, theoretically, if the job recoverable, a stage with less active executor node would run for long time to finish this stage. To exemplify the varying situation, we first set a (10 ± 4)-node dynamic cluster whose mean node number in long term is the same with a 10-node static cluster

parameters	how it affects
performance in static cluster	Performance in the static cluster outlines the best performance that can be possibly achieved in the dynamic cluster. In the dynamic cluster, if there is no node failure and thus rollback, job completion by stage whose time determined by the performance in the static cluster would not be repeated. So avoiding revocation as much as possible lead to optimal results.
timeout	Timeout is criterion for the system to terminate the job and time limit within which node connectivity issues must be resolved. By default, after three attempts on reconnection with the failed node, the current job will be killed by driver program.
CPU core	More available CPU cores are almost positive for everything. In our experiment, we restricted CPU core per node (using `m3.medium` instances).
checkpoint write	Checkpointed job does not need to start over. However, if there is no failure, checkpoint write time is wasteful. In the effectiveness experiment, to test if Spark without high-latency checkpointing can complete jobs.

Table 4.1: Factors that potentially affect resilience

without node loss and addition. Later a change in amplitude are discussed. Results of these sub-experiments are stated in Chapter 6.

4.2.2 Parallelism Degree

Cached RDD number (or *parallelism degree*) in total is set to 20, making maximum of hosted RDD number on each executor node less than 2.0. By default, an equivalent CPU core can process 2 RDDs at the same time thus as active node decreases, average number of RDD hosted on executor node exceeds 2.0 and simply lengthen job completion time for this stage by at least 100%. There is also an auxiliary experiment to see how RDD per node impacts performance.

4.2.3 Mean Time to Fail/revoke

The interval, or mean time to fail/revoke, is the key impact from the exterior environments, and whether the Spark cluster could recover from the turbulent technically depends on whether the capacity to recover meet the deadline (there is a timeout in the system).

4.2.4 Mean Time to Write Checkpoint

Later when we combined usage of both lineage and traditional checkpoint mechanisms, how often we conduct checkpoint write also affect Spark cluster performance. From [13], we know that for a single-node batch-job, the job completion time is given by

$$T_w(\tau) = \underbrace{T_s}_{\text{solve time}} + \underbrace{\left(\frac{T_s}{\tau} - 1\right)\delta}_{\substack{\text{checkpointing}\\\text{dump time}}} + \underbrace{[\tau + \delta]\,\phi(\tau + \delta)\,n(\tau)}_{\text{recovery time}} + \underbrace{Rn(\tau)}_{\text{restart time}}\ , \tag{4.1}$$

where T_s denotes job completion time without failure (solve time), $n(\tau)$ interruption time, δ time to write a checkpoint file, $\phi(\tau + \delta)$ fraction of interruption averagely, and R time to restart. And the optimum of mean time to write checkpoint is given by $\tau_{opt} = \sqrt{2\delta M}$, where M denotes mean time to interrupt. Not only can it be used for verification that the simulator reflects real-world cases, we expect to extend its scope to distributed cases. On the other hand, when real history price is used to simulate the cluster, Equation 4.1 does not quite apply any more, and hidden mathematically representation is still to be discovered.

4.3 Simulator

For real-world tasks it takes at least 10 minutes to finish a task, and even longer time to repeatedly get reasonable result with less deviations. To speed up development, we

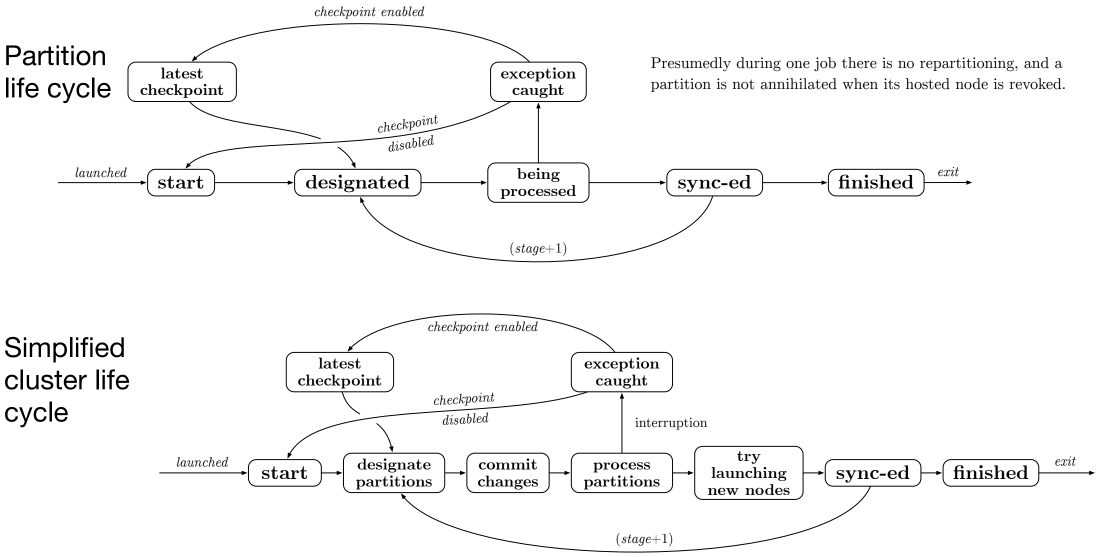


Figure 4.3: Simpler cluster life cycle description

designed a simulator. An intuitive idea to simulate the cluster is to multithread the simulator program. In details, we can deploy one thread for driver node and multiple for executor nodes. However, to stick with the goal rather than simply emphasize on the mechanism or implementation, as well as ability to extend the program in the future, we prioritize the observation of partition progress; in comparison, node is container where partitions of workload is hosted, and node life cycle that later as we can see, could be logically integrated as a whole cluster.

In Figure 4.1, we can see that life cycle mostly coincides with executor node in the cluster except for the partition is designed to live until the job is finished. After tentatively implementing a multi-threading prototype, we found it was neither easy to extend nor necessary: 1) stage completion time for an iteration is determined by the longest partition processing time from a specific node in the cluster, thus the competing process is trivial to record in the simulator, and 2) cost exists as long as instances are on. Thus, in sense of optimization, we can simply calculate the longest processing time for that stage. And checkpoint mechanism would pause the processing, thus processing and checkpoint if any are executed in serial under the scheduling from driver node. Thus a much simpler as well as much faster single-threaded simulator is implemented from the angle of the while cluster. In the description of the cluster, we focus on how partition state is transited. See details in Figure 4.3.

CHAPTER 5

IMPLEMENTATION

Most parts for this project is implemented in Python, Shell Script and illustrative applications are in Scala. Also project platform is available and open-sourced at https://github.com/JonnyCE/project-platform. And this chapter is organized in three parts: 1) Cluster setting, 2) platform and 3) pattern-based controller implementation.

5.1 Cluster Setup

Components listed in Table 5.1 are necessary to set up a cluster. Unfortunately, there is no handy deploy tool from Amazon official; in fact, Amazon's command line tools are quite fault-prone when deploying manually. At this stage we use both Spark EC2 (released by Spark group) and implemented console tools based on Python Boto 2.3.8, and this will be the part comprising our abstraction interface.

component	version	usage
Spark	1.2.x or 1.3.x	Framework where applications submitted
HDFS	Hadoop 2.4+	Delivering distributed file system
Mesos	0.18.0 or 0.21.0	Working as resource allocator
YARN	Hadoop 2.4+	Mesos alternative, negotiator
Scala	2.10	Front end for Java runtime
Python	2.6+	Boto 2 package is employed for customization
Java	6+	Backend for Hadoop, Scala and Spark
Bash	built-in	Built-in script interpreter

Table 5.1: Components and compatibility

- *EC2 Spot Instances* With a pool of spot instances [1], we can request flexible number of node to use. At this stage, we use Spark official EC2 deployment tool to automate authorization between driver and executor nodes. To manipulate the execute node, an ancillary control panel is also implemented based on AWS Boto API and *Secure Shell* (SSH) pipe as supplement. And to deliver better performance, in the effectiveness experiment, we employ a `m3.large` instance as driver node, and `m3.medium` as executor instances.

- *Storage* Master-slave modeled HDFS cluster consists of a single `namenode` that manages the file system namespace and regulates access to file by clients and a number of `datanode`. HDFS exposes a file system namespace and allows user data to be stored in files [7]. The existence of a single HDFS `namenode` in a cluster simplifies the architecture of the system. the `namenode` is designed to be the arbitrator and repository for all HDFS meta-data and user data never flows through the `namenode`.

  In this paper We presume that the HDFS cluster (storage) the Spark cluster do not overlap. At this stage, we also can use AWS S3 Bucket for easier deployment. Now, we host Spark application (`.jar`) with experiment dataset and tarball of Spark framework in the bucket.

- *Resource Allocator* Mesos or YARN could be used to multiplex resource usage due to the essence that there are multiple frameworks running on each single node. Mesos is designed to offer resources and collect feedback (accepted or refused) from multitenant frameworks which do nothing against the nature of frameworks [17]. Yet YARN is an alternative choice that we did not take a close look at. To port Mesos on our target operating system, we compiled Mesos of both 0.18.0 and 0.21.0 and one of them is chosen to be installed as default one.

Spark, the Framework This experiment is to focus on fault tolerance and resilience features of Spark. Among different distributions of Spark, we choose binary package that is pre-built for Hadoop 2.4+. And two most recent versions, 1.2.2 and 1.3.1, in regard to compatibility.

- *Control panel* We have implemented different components for this project platform shown in Table 5.2.

component	**description**
console	based on AWS Boto 2.38 to request lookups and make snapshot/user image on current cluster
experiment	a spot market request simulator, generating and propagating availability pattern to the Spark framework
logger	recording and analyzing availability pattern impact
graphic library	supporting data visualization
math library	containing price analysis tools

Table 5.2: Control panel

- *PageRank demo application* The lineage of example PageRank consists 13 stages: 2 `distinct` actions, 10 `flatmap` transformations for there are 10 iterations and 1 `collect` action.

- *Cluster setting* The cluster is set as shown in Table 5.3. Noticed that time factor setting is based on such a cluster. In the experiments based on simulation in Section 6.3, a time unit (40 seconds) is based on stage completion time.

5.2 Simulator Implementation

The behavioral pseudo-code for the simulator essence is list below.

The simulator, as core part of the experiment, is implemented in C++ for better performance, while analytical jobs are done in Python and shell scripts.

overview	driver	`m3.large`
	executor	`m3.medium`, with 2.4 GiB memory per node for Spark worker
usage	cores	unlimited, 10 for most of time
	memory	300 to 500 MiB/12.8 GB in total
	disk	0 B for we did not set up checkpoint write
application	description	PageRank with 10 iterations
	variable	*iteration count*, in this case we set it constant 10; *partition number* as known as *RDD caching degree*, or degree of parallelism
	language	Scala 2.10 with Java 1.7 as backend
	package	`.jar` package to submit
dataset	source	`https://snap.stanford.edu/data/web-Google.html`
	filesystem	hosted on S3 bucket: `s3n://spark-data-sample/web-Google.txt`
	description	containing 875713 nodes, 5105039 edges

Table 5.3: Cluster setting

```
initialization

while not all partitions finished processing:
if time to interrupt:
chosen victim nodes are "down"
hosted partitions roll back to checkpoint

if iteration-based:
select only lagging partitions to resume
else:
select all partitions to resume
designate corresponding partitions to active nodes

overhead of resume applied if any
bring back nodes if appropriate
process partitions

if checkpoint enabled and time to write:
checkpoint write

done
```

CHAPTER 6

EVALUATION

6.1 Evaluation of Effectiveness Experiment

Job completion time is lengthened when there is loss and fallback and varies according to specific parameters. Presumably, there is no re-partitioning that changes parallelism degree, i.e. partition number of a task. In a dynamic cluster, with constant compute capacity of a single node (we only focus on CPU related capacity), stage completion time always varies due to fluctuating node number of the cluster.

Quantitatively, we set a cluster of constant 10 nodes, or a 10-node static cluster as pivot. In the effectiveness experiment, we set a node number fluctuating according to a periodic pattern with average value 10, i.e. a cluster of $(10 \pm m)$ nodes. With such technique, in sense of node availability (the number of available node for computing), these two clusters are at the same cost in average. Nevertheless, a $(10 \pm m)$-node cluster should not be the equivalence of a 10-node static cluster; a $(10 + m)$-node cluster loses $2m$ nodes due to revocations on purpose.

We would show the impacts from multiple aspects:

- Amplitude of the node availability varies in different scenarios; a $10 \pm m_1$- and a $10 \pm m_2$-node cluster ($m_1 \neq m_2$) share the same cost on average if running for the same time in the long term. However, to finish a exactly same jobs, the completion time may varies.

- An implication of node availability decrement undermines performance; such a decrement happens in the descending phase of the pattern. If there is no change in node availability and the node number remains at a certain level, the completion time is only determined by the workload and compute capacity. And if the dynamic cluster, within a short duration, the average compute capacity is the same with one in the static cluster but job completion time increases, we assume there is extra overhead for node availability fluctuation

- Reservation of always on node. (unfinished) There has been discussion on whether to employ always-on node to guarantee the performance or not. For the sake of simplicity, only an illustration is shown in Figure 6.2, and we choose not to utilize such alway-on instances for simplicity.

6.1.1 Base Completion Time

To settle the question of existence of overhead from node availability change, we first measured job completion time in a static cluster as pivot. Job completion time comprises each stage completion time. To standardize, we measured stage completion time where constant partitions are mapped onto various number of executor nodes. And such measurement guided the development of the simulator for parameter configuration. The static cluster for measuring base completion time is configured as: 1) 10 `m3.medium` executor nodes, or 10 active CPU cores, 2) each instance has 1 CPU core, able to process 2 partitions in the same time and 3) demo MapReduce application contains 10 iterations. Job completion time is shown in Table A5, and Figure 6.1.

In this experiment, we designated 20 partitions onto 10 nodes. As *partition number* is increased from 2 to 20, job completion time drops; hosted partition number decreased from 10.0 to 1.0. Noticed that stage completion time slightly increases when less than 2.0

partitions are hosted on a CPU core on average. In addition, job completion time sum total is approximately the same as what is given in the Spark WebUI (a built-in graphical control panel) Result is shown in Table A5 and Figure 6.1.

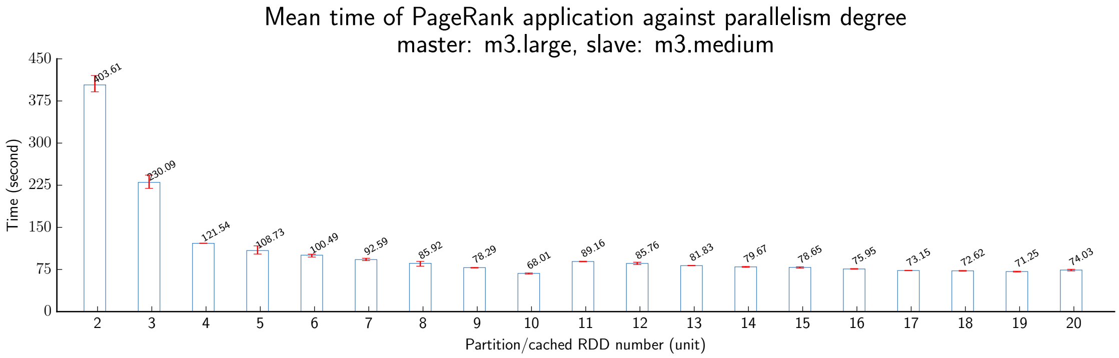


Figure 6.1: Figure for Table A5

6.1.2 Job Completion in Dynamic Cluster

In the effectiveness experiment, we applied a pattern to node availability to a cluster with at most $10 + m$ executor nodes, making it a dynamic cluster. And there is no extra fault tolerance mechanisms applied except the internal one. We set the amplitude of pattern from $\{4, 6, 8\}$, making the (maximum, minimum) of a cluster node number $(14, 6)$, $(16, 4)$ and $(18, 2)$ respectively. For each case, we also set comparison of cases with and without reserved always-on nodes in the cluster. The discrete pattern is in unit of 30 seconds; node number is changed compulsorily every 30 seconds. Below 30 seconds, revocation is intensified and the cluster can hardly recover and exceed the timeout caused by cascading fallback. Timeline of each case is shown in Figure 6.2 and it shows the feasibility of completing job with appropriate parameters.

We ran the same application (10 iterations) in the dynamic cluster for four times. Trend shows that small drop from maximum of the pattern lead to shorter completion time. Comparing a (10 ± 4)- and a (10 ± 6)-node cluster, we noticed that gap in performance is small and even negligible with these case study; however, a (10 ± 8)-node alteration shows obvious violation on the executing and the completion time is lengthened much more in contrast to (10 ± 4) case. Trend also shows that running job in the ascending phase of the pattern is much shorter than in the descending phase, which is intuitive and expected. Nevertheless, in this illustrative evaluation, we accessed to full control over the node availability, otherwise, in the real-world, we cannot predict on the phase change of the market and the alteration of price is not gradually but abruptly. Moreover, the absolute overhead is dense, even the (10 ± 4) cluster ran the task for much longer time than the bad cases shown in Figure 6.1. Such a result can be attributed to the lack of proper fault-tolerant mechanisms.

In addition, reserved always-on (on-demand) instances boost the performance. And on rare occasions, if node availability is extremely low, and memory capacity is far more below abundant, even loading dataset on need basis cannot be smooth; rather, virtual memory swapping between memory and disk is automatically invoked and the latency is magnified. In sense of guaranteeing enough memory capacity, always-on instances can be put into use. However, on balancing the complexity of design, the cost and income, and such technique is not applicable to all types of jobs. We proceed later experiments without such technique.

6.2 Impacts of Parameters

In each experiment, we have 3 dynamic cluster with different pattern amplitude; a single parameter is varying while others are unaltered. Also in each experiment consists of at least 20 submissions of the example PageRank application. To simulate the real-word cases, we submit application to the cluster at arbitrary phase of periodical availability pattern.

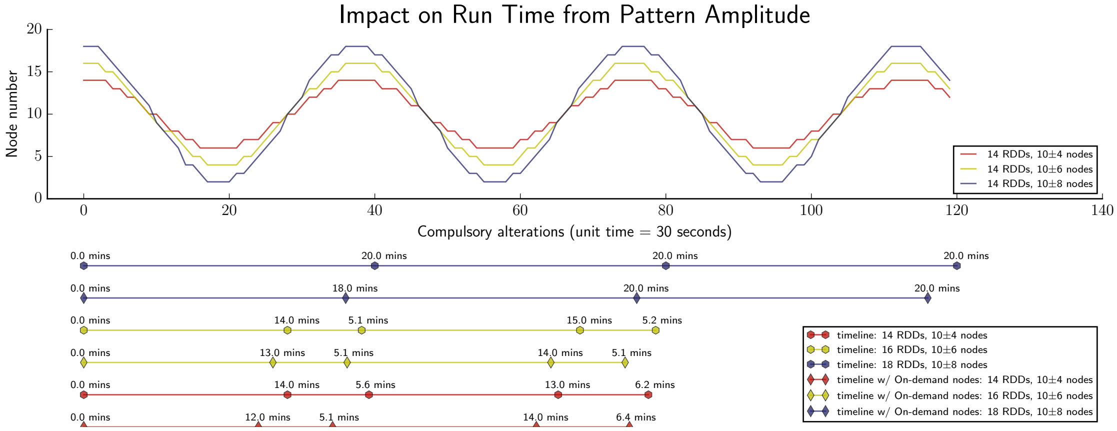


Figure 6.2: Running time in dynamic cluster

So far we gain some illustrative results as shown in Figure 6.3. The first figure shows the impact on job completion time by changing MTBA. Trending is that longer MTBA interval leads to smaller variance of job completion time although sometimes some scattered cases have much longer job completion time. The second figure shows the impact on job completion time by changing lineage length, in this case, the iteration number. The trending reflects the correctness of intuition that either larger amplitude (corresponding to less availability) or longer iteration makes cluster even harder to recover. If we compare amplitude varying and iteration varying separately, we find that variance beyond 25 to 75 percentile increasing holds, although as iteration number increases, monotonicity of job completion time within 1.5 IQRs no longer valid. The third figure shows the impact on job completion time by changing partition number. It is straight forward that increasing parallelism degree from 10 to 20 leads to lower overhead and faster time finishing job. Yet it is not always valid that amplitude increasing surely deteriorate recovery. More scrutiny is needed on this part.

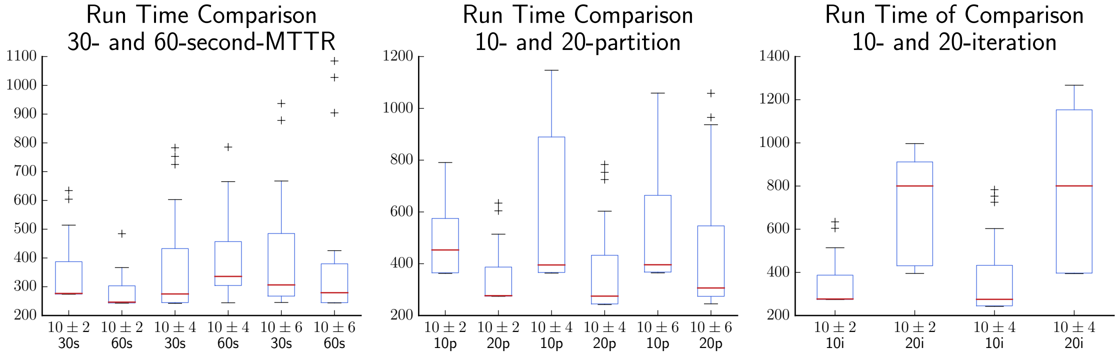


Figure 6.3: Parameter impacts on job completion time

6.3 Results from Simulation

Verification With inspiration of optimization in single-node batch-job case, we were to apply optimum for distributed jobs. Before that, we first verified the simulator by running a single-node batch job. After the correctness is proven, we extended the experience to distributed cases and conducted a simple MapReduce to gain result, and it turned out to be applicable. Both cases are under such restrictions: 1) revocation occurrence satisfies the approximation of Poisson distribution, 2) a state of the job at one moment is dependent on previous states, and 3) revocation failure rate is proper such that with checkpoint write a job could be finished. Both cases are shown in Figure 6.4.

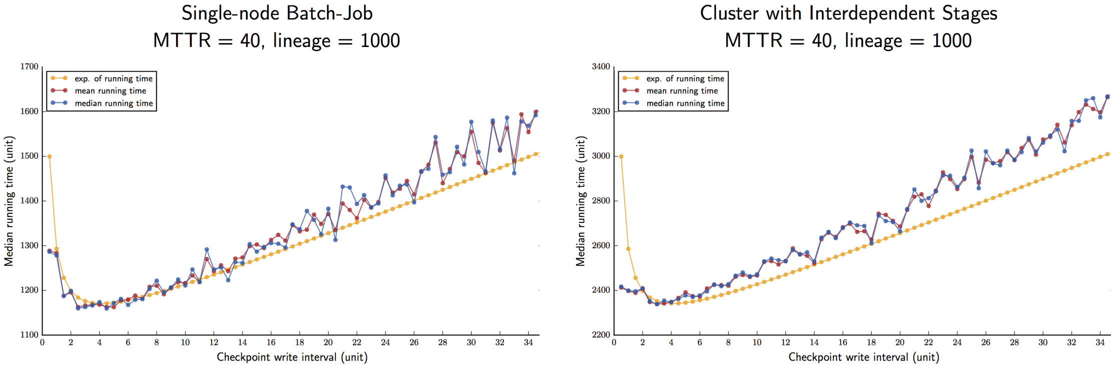


Figure 6.4: Verification and extension

Experiments based on simulation From the actual execution on real Spark Instances, we gathered some data: 1) in a static cluster, stage completion time is around 40 seconds when average RDD number on an executor node is less than 2.0. and 2) Spark cluster can recover from a revocation every 30 seconds averagely (based on both pre-selected pattern and Poisson Distribution). With these a posteriori experience, we did some case studies with simulations of `m3.large` instance, and we get some sample results listed below. And these results are main patterns selected various experiments.

In Figure 6.5 we can see that overall trend shows that overhead from checkpoint write impact on performance when checkpoint writing too frequently but alleviated when the write interval set to appropriate value; however, when there are inadequate checkpoints, severe performance deterioration takes place and becomes even worse when checkpoint write is towards absolutely absent. Thus we see a small drop to local minimum in both job completion time and total cost, and it becomes global minimum.

Figure 6.6 shows a pattern that resembles one in Figure 6.5. As we can see, the pattern goes flat because there is the short duration of price alteration, where limited revocations impact on job completion time thus total cost.

In Figure 6.7, we see that at bid of 0.16x, like patterns shown in Figure 6.5 and Figure 6.6, a small drop occurs, leading to local minimum in both job completion time and total cost, after that, both rises. Another observation is that when we slightly rise the bid, we can see then the only overhead is from routine checkpoint write.

Figure 6.6 shows drop and steady trending toward situation in which there is no checkpoint write. This is attributed to constant number of revocations exist during the job processing. Recall that if there are cascading revocations, Spark may hit timeout and failed the job (see Section 2.1.2). So we use this to determine to what degree shorter completion time and cost saving can be achieved. In this case, with mixed fault tolerance scheme, ~20% shorter completion time and ~25% lower cost are achieved than the situation of no checkpoint write; and compared with cases of running jobs on full-price instance, the absolute saving in costs can be ~75%.

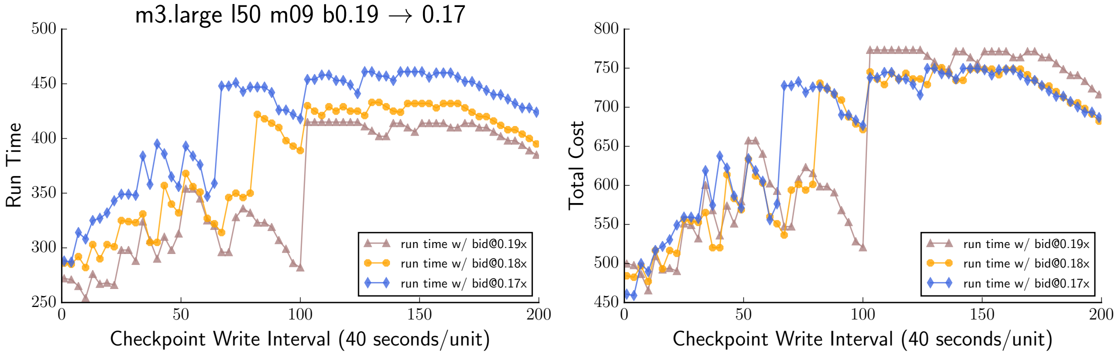
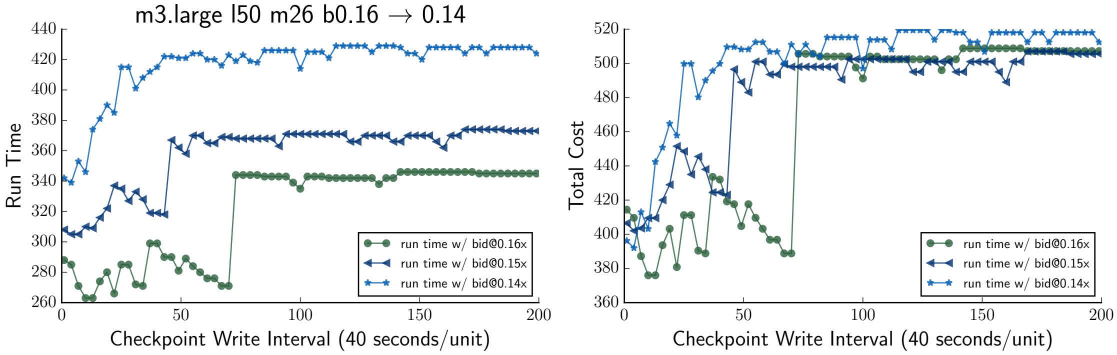


Figure 6.5: Pattern of small drop 1

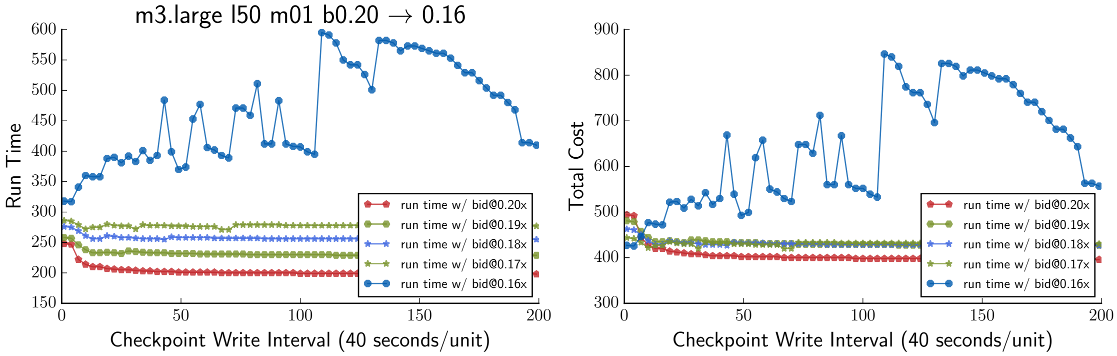


	completion time			cost		
bid	min/overhead	max/overhead	trending	min/discounted	max/discounted	trending
0.16	260 / 21.5%	342 / 59.8%	342 / 59.8%	375 / 17.5%	508 / 23.7%	500 / 23.4%
0.15	304 / 42.1%	370 / 72.9%	370 / 72.9%	392 / 18.3%	500 / 23.4%	500 / 23.4%
0.17	337 / 57.5%	425 / 98.6%	425 / 98.6%	400 / 18.7%	517 / 24.2%	571 / 26.7%

Figure 6.6: Pattern of small drop and constant

Noticed that result can be changed a lot when parameters are slightly tweaked. For example, starting timeframe can lead to good or bad timing when encountering price change, and d grace period also contributes to the timing.

Presumably, all clusters go through revocations. We conclude that:



	completion time			**cost**		
bid	**min/overhead**	**max/overhead**	**trending**	**min/discounted**	**max/discounted**	**trending**
0.20	N/A	250/16.8%	215/0.5%	N/A	295/13.8%	400/18.7%
0.19	N/A	260/21.5%	235/9.8%	N/A	485/22.7%	440/20.6%
0.18	N/A	275/28.5%	260/21.5%	N/A	460/21.5%	440/20.6%
0.17	N/A	285/33.2%	280/30.8%	N/A	440/20.6%	440/20.6%
0.16	320/49.5%	600/180.4%	N/A	420/19.6%	850/39.7%	N/A

Figure 6.7: Price-sensitive pattern

1. Optimum checkpoint write interval tends to be small, i.e. overhead from checkpoint write is much smaller than that from rolling back with RDD recovery.

2. The robust of checkpointing in sense of interval could help lower the price of using spot instances and work according to market information.

3. Sometimes a small rise in bid can lead to qualitative change and lower bid does not always mean lower cost.

APPENDIX A

SPOT INSTANCE PRICE RECORDS

purpose	type	vCPU	ECU	RAM (Gib)	disk (GB)	price according to usage (USD per hour)					
						Linux/UNIX			Windows	w/ SQL	
						general	RHEL	SUSE	general	std.	web
general	`t2.micro`	1	var.	1	EBS Only	0.01	0.07	0.02	0.02		0.07
	`t2.small`	1	var.	2	EBS Only	0.03	0.09	0.06	0.04		0.14
	`t2.medium`	2	var.	4	EBS Only	0.05	0.11	0.15	0.07		0.27
	`t2.large`	2	var.	8	EBS Only	0.10	0.16	0.20	0.13		0.43
	`m3.medium`	1	3	3.75	1 x 4 SSD	0.07	0.13	0.17	0.13	0.35	0.18
	`m3.large`	2	6.5	7.5	1 x 32 SSD	0.13	0.19	0.23	0.26	0.70	0.37
	`m3.xlarge`	4	13	15	2 x 40 SSD	0.27	0.33	0.37	0.52	1.27	0.73
	`m3.2xlarge`	8	26	30	2 x 80 SSD	0.53	0.66	0.63	1.04	2.53	1.47
	`m4.large`	2	6.5	8	EBS Only	0.13	0.19	0.23	0.25	0.93	0.26
	`m4.xlarge`	4	13	16	EBS Only	0.25	0.31	0.35	0.50	1.12	0.44
	`m4.2xlarge`	8	26	32	EBS Only	0.50	0.63	0.60	1.01	2.35	0.90
	`m4.4xlarge`	16	53.5	64	EBS Only	1.01	1.14	1.11	2.02	4.64	1.84
	`m4.10xlarge`	40	124.5	160	EBS Only	2.52	2.65	2.62	5.04	11.81	4.58
compute optmized	`c3.large`	2	7	3.75	2 x 16 SSD	0.11	0.17	0.21	0.19	0.56	0.27
	`c3.xlarge`	4	14	7.5	2 x 40 SSD	0.21	0.27	0.31	0.38	1.07	0.54
	`c3.2xlarge`	8	28	15	2 x 80 SSD	0.42	0.55	0.52	0.75	2.13	1.08
	`c3.4xlarge`	16	55	30	2 x 160 SSD	0.84	0.97	0.94	1.50	4.26	2.17
	`c3.8xlarge`	32	108	60	2 x 320 SSD	1.68	1.81	1.78	3.01	8.52	4.33
	`c4.large`	2	8	3.75	EBS Only	0.11	0.17	0.21	0.19	1.41	0.42
	`c4.xlarge`	4	16	7.5	EBS Only	0.22	0.28	0.32	0.39	1.68	0.79
	`c4.2xlarge`	8	31	15	EBS Only	0.44	0.57	0.54	0.77	3.35	1.64
	`c4.4xlarge`	16	62	30	EBS Only	0.88	1.01	0.98	1.55	5.58	2.23
	`c4.8xlarge`	36	132	60	EBS Only	1.76	1.89	1.86	3.09	12.57	4.27
GPU instance	`g2.2xlarge`	8	26	15	60 SSD	0.65	0.78	0.75	0.77	3.82	0.96
	`g2.8xlarge`	32	104	60	2 x 120 SSD	2.60	2.73	2.70	2.88		
memory optmized	`r3.large`	2	6.5	15	1 x 32 SSD	0.18	0.24	0.28	0.30	0.96	0.40
	`r3.xlarge`	4	13	30.5	1 x 80 SSD	0.35	0.41	0.45	0.60	1.40	0.76
	`r3.2xlarge`	8	26	61	1 x 160 SSD	0.70	0.83	0.80	1.08	2.78	1.56
	`r3.4xlarge`	16	52	122	1 x 320 SSD	1.40	1.53	1.50	1.94	4.66	2.37
	`r3.8xlarge`	32	104	244	2 x 320 SSD	2.80	2.93	2.90	3.50	8.76	4.00
storage optmized	`i2.xlarge`	4	14	30.5	1 x 800 SSD	0.85	0.91	0.95	0.97	1.23	0.99
	`i2.2xlarge`	8	27	61	2 x 800 SSD	1.71	1.84	1.81	1.95	2.46	1.99
	`i2.4xlarge`	16	53	122	4 x 800 SSD	3.41	3.54	3.51	3.89	4.92	3.97
	`i2.8xlarge`	32	104	244	8 x 800 SSD	6.82	6.95	6.92	7.78	9.84	7.94
	`d2.xlarge`	4	14	30.5	3 x 2000 HDD	0.69	0.75	0.79	0.82		
	`d2.2xlarge`	8	28	61	6 x 2000 HDD	1.38	1.51	1.48	1.60		
	`d2.4xlarge`	16	56	122	12 x 2000 HDD	2.76	2.89	2.86	3.06		
	`d2.8xlarge`	36	116	244	24 x 2000 HDD	5.52	5.65	5.62	6.20		

Table A1: `east-us-1` On-demand instance pricing

data collected at 11:16 PM on October 8, 2015, `us-east-1`

type	discounted price			type	discounted price		
	spot	1-hr fixed	6-hr fixed		spot	1-hr fixed	6-hr fixed
`m3.medium`	14%	55%	70%	`c4.xlarge`	15%	55%	70%
`m3.large`	20%	55%	70%	`c4.2xlarge`	17%	55%	70%
`m3.xlarge`	15%	55%	70%	`c4.4xlarge`	16%	55%	70%
`m3.2xlarge`	14%	55%	70%	`c4.8xlarge`	23%	55%	70%
`m4.large`	12%	55%	70%	`d2.xlarge`	10%	55%	70%
`m4.xlarge`	11%	55%	70%	`d2.2xlarge`	11%	55%	70%
`m4.2xlarge`	11%	55%	70%	`d2.4xlarge`	10%	55%	70%
`m4.4xlarge`	12%	55%	70%	`d2.8xlarge`	11%	55%	70%
`m4.10xlarge`	14%	55%	70%	`g2.2xlarge`	11%	55%	70%
`c3.large`	16%	55%	70%	`g2.8xlarge`	18%	55%	70%
`c3.xlarge`	18%	55%	70%	`r3.large`	15%	55%	70%
`c3.2xlarge`	20%	55%	70%	`r3.xlarge`	14%	55%	70%
`c3.4xlarge`	19%	55%	70%	`r3.2xlarge`	20%	55%	70%
`c3.8xlarge`	19%	55%	70%	`r3.4xlarge`	3%	55%	70%
`c4.large`	16%	55%	70%	`r3.8xlarge`	11%	55%	70%

Table A2: `east-us-1` Spot and Fixed-duration instance pricing

bid	c3.2xl	c3.4xl	c3.8xl	c3.l	c3.xl	d2.2xl	d2.4xl	d2.8xl	d2.xl	g2.2xl	g2.8xl
0.12	4208.30	3941.77	2889.94	836.22	3116.36	291.73	170.89	117.63	245.48	2952.92	706.53
0.13	4208.30	3941.77	2889.94	836.22	3116.36	163.37	125.82	86.46	231.32	2815.10	704.86
0.14	4208.30	3941.77	2889.94	836.22	3116.36	91.53	99.93	58.24	218.29	2516.51	701.34
0.15	4208.30	3941.77	2889.94	836.22	3116.36	51.86	70.76	33.93	203.58	2183.04	691.92
0.16	4208.30	3934.80	2835.78	795.31	3108.72	40.96	60.23	27.38	181.41	1900.16	676.66
0.17	4198.54	3842.84	2289.62	417.47	2859.20	31.80	52.52	19.51	149.42	1629.76	655.06
0.18	4095.47	3594.43	1703.28	199.52	2341.02	25.69	44.23	11.27	127.90	1394.20	622.41
0.19	3763.73	3189.54	1270.56	109.40	1814.31	23.58	39.07	10.10	111.63	1159.24	589.04
0.20	3308.16	2719.54	966.24	74.84	1372.86	19.41	33.63	7.47	95.73	990.44	542.97
0.21	2763.84	2260.20	732.17	54.79	1050.86	16.78	28.84	5.44	80.03	840.39	511.72
0.22	2250.29	1844.71	560.76	42.29	815.73	14.32	26.57	4.37	71.14	721.26	484.78
0.23	1820.61	1508.37	436.27	31.68	647.33	12.27	21.34	3.69	61.41	618.10	457.91
0.24	1458.20	1234.52	337.92	26.61	517.17	10.98	19.08	3.36	50.34	531.99	435.38
0.25	1171.07	1022.90	263.46	22.72	425.46	9.60	18.40	3.11	43.22	462.84	412.98
0.26	933.74	861.98	199.20	20.03	358.77	8.66	16.62	3.08	36.44	400.56	393.48
0.27	730.83	731.29	149.81	17.57	300.87	8.21	15.89	3.04	31.92	341.27	364.93
0.28	572.54	623.68	120.31	16.18	253.64	7.68	15.36	3.03	28.29	296.03	352.46
0.29	446.94	538.90	99.14	14.64	217.19	6.58	13.13	2.47	24.94	256.20	329.16
0.30	349.77	447.66	82.86	13.76	187.81	6.29	12.72	2.37	21.38	226.19	314.61
0.31	267.04	399.98	72.23	12.74	163.92	5.99	11.17	2.33	17.98	191.20	295.22
0.32	216.26	358.72	64.88	12.14	142.60	5.73	10.93	2.29	15.73	180.84	281.80
0.33	172.47	326.46	59.32	11.69	126.06	5.44	10.66	2.29	14.13	165.57	270.28
0.34	139.04	296.04	54.71	11.32	108.92	5.09	10.50	2.27	12.71	156.62	256.86
0.35	110.14	270.64	49.94	11.04	97.50	4.89	10.27	2.26	11.32	150.16	245.81
0.36	86.17	246.52	45.73	10.68	87.10	4.69	9.97	2.24	10.47	142.07	233.74
0.37	70.61	227.31	42.70	10.38	78.50	4.37	9.61	2.24	9.94	134.20	220.81
0.38	58.17	209.86	40.02	10.07	70.60	4.29	9.36	2.24	9.02	130.44	213.07
0.39	47.67	194.92	37.72	9.72	61.97	4.22	8.58	2.24	8.09	121.57	174.12
0.40	40.04	180.93	35.59	9.56	55.89	4.16	8.43	2.24	7.39	117.71	169.46

Table A3: Market volatility 01, highlighted if 10 revocations per hour

bid	i2.2xl	i2.4xl	i2.8xl	i2.xl	m3.2xl	m3.l	m3.m	m3.xl	r3.2xl	r3.4xl	r3.8xl	r3.l	r3.xl
0.12	501.09	366.49	117.09	282.46	3251.41	1245.54	498.22	3659.60	3844.26	1545.18	2622.84	294.83	2447.93
0.13	322.32	306.62	88.19	232.22	3249.80	1231.91	486.34	3655.48	3414.50	1188.36	2273.86	180.21	2069.16
0.14	190.69	240.62	69.54	159.23	3134.78	834.16	341.01	3478.44	2845.17	944.61	1924.46	107.87	1738.32
0.15	98.48	182.38	48.66	75.39	2688.93	431.16	246.89	2809.06	2289.04	759.98	1586.87	74.54	1462.18
0.16	52.12	135.67	20.81	31.29	2085.14	256.97	184.70	1958.68	1805.79	637.66	1298.44	54.27	1249.51
0.17	28.71	104.33	10.83	14.22	1487.50	170.09	120.43	1151.16	1383.41	546.06	1058.79	42.77	1093.04
0.18	16.60	87.87	9.73	2.94	1026.47	123.63	72.62	658.77	1061.92	475.58	844.49	34.79	957.97
0.19	12.50	69.29	9.02	2.93	718.37	91.68	44.86	406.11	835.73	420.97	633.72	28.81	854.50
0.20	11.33	48.20	8.33	2.23	508.89	69.63	29.71	261.24	659.86	381.20	524.49	25.03	762.62
0.21	10.20	34.11	7.59	2.11	379.22	54.64	20.67	181.14	530.21	344.24	438.66	21.72	687.50
0.22	9.64	23.24	7.09	2.06	284.66	40.44	15.88	124.84	426.58	314.60	370.76	19.51	619.24
0.23	8.70	19.30	6.38	1.30	216.24	31.51	12.54	89.01	344.61	284.27	314.30	17.29	563.00
0.24	7.60	17.83	5.96	1.30	163.46	25.44	9.56	64.78	288.72	258.80	269.92	15.66	516.74
0.25	7.01	16.36	5.52	1.29	123.39	21.03	7.92	50.66	242.19	237.83	235.92	14.51	472.94
0.26	5.99	12.68	4.86	1.28	94.77	17.31	6.21	42.31	203.48	218.54	204.71	13.28	434.99
0.27	5.72	11.37	4.51	1.28	71.92	13.82	5.67	37.40	174.86	201.68	180.16	11.86	403.32
0.28	5.49	10.20	4.09	1.28	49.36	11.20	5.14	35.18	150.38	186.51	156.81	11.04	372.71
0.29	5.38	9.14	3.73	1.01	41.19	9.66	3.24	32.37	130.38	174.11	136.53	10.54	346.48
0.30	5.17	7.77	3.52	1.01	31.72	8.42	2.06	29.66	113.01	162.09	118.88	10.09	322.22
0.31	4.52	7.06	2.92	0.98	26.26	7.10	2.06	25.58	98.81	148.93	103.41	9.68	301.22
0.32	4.41	6.67	2.77	0.98	21.60	6.10	2.06	21.20	85.82	139.12	92.68	8.53	279.34
0.33	4.23	6.33	2.66	0.98	18.32	5.13	2.06	16.84	75.50	130.38	83.10	8.30	261.26
0.34	4.19	6.10	2.48	0.98	14.67	4.23	2.06	14.48	66.61	120.61	74.47	8.00	243.47
0.35	4.01	5.80	2.42	0.98	11.93	3.70	2.04	13.14	58.59	112.62	67.90	7.84	228.26
0.36	3.91	5.58	2.36	0.98	10.09	3.20	2.04	12.13	50.62	105.11	62.23	7.63	214.63
0.37	3.88	5.42	2.29	0.98	8.72	2.83	2.04	11.36	42.92	97.41	57.48	7.41	202.59
0.38	3.71	5.07	2.22	0.98	7.54	2.51	2.04	10.69	36.00	86.54	52.03	7.21	191.20
0.39	2.77	4.31	2.06	0.96	6.11	2.37	2.04	10.09	29.53	80.57	48.29	7.06	181.66
0.40	2.74	4.13	1.97	0.96	5.79	2.33	2.04	9.60	22.12	74.46	44.79	6.89	170.71

Table A4: Market volatility 02, highlighted if 10 revocations per hour

RDD	run time/second			statistics				
caching	1st	2nd	3rd	average	upper error		lower error	
degree	instance	instance	instance		value	percent	value	percent
2	399.320	391.292	420.226	403.613	16.613	4.12%	12.321	3.05%
3	243.068	219.362	227.840	230.090	12.978	5.64%	10.728	4.66%
4	122.002	121.276	121.354	121.544	0.458	0.38%	0.268	0.22%
5	102.479	117.092	106.608	108.726	8.366	7.69%	6.247	5.75%
6	97.164	102.284	102.032	100.493	1.791	1.78%	3.329	3.31%
7	91.984	90.778	95.010	92.591	2.419	2.61%	1.813	1.96%
8	87.494	80.876	89.383	85.918	3.465	4.03%	5.042	5.87%
9	78.674	77.551	78.640	78.288	0.386	0.49%	0.737	0.94%
10	68.813	68.366	66.861	68.013	0.800	1.18%	1.152	1.69%
11	88.529	89.188	89.776	89.164	0.612	0.69%	0.635	0.71%
12	83.776	88.001	85.499	85.759	2.242	2.61%	1.983	2.31%
13	81.546	82.397	81.544	81.829	0.568	0.69%	0.285	0.35%
14	79.858	78.711	80.425	79.665	0.760	0.95%	0.954	1.20%
15	77.439	78.753	79.757	78.650	1.107	1.41%	1.211	1.54%
16	75.719	75.456	76.676	75.950	0.726	0.96%	0.494	0.65%
17	73.128	73.595	72.721	73.148	0.447	0.61%	0.427	0.58%
18	72.592	72.050	73.233	72.625	0.608	0.84%	0.575	0.79%
19	71.956	71.341	70.464	71.254	0.702	0.99%	0.790	1.11%
20	72.473	74.254	75.373	74.033	1.340	1.81%	1.560	2.11%

Table A5: Baseline job completion time

APPENDIX B

SPARK WORKING MODES

storage level	description
`MEMORY_ONLY`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
`MEMORY_ONLY_SER`	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
`MEMORY_AND_DISK_SER`	Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
`DISK_ONLY`	Store the RDD partitions only on disk.
`MEMORY_ONLY_2`, `MEMORY_AND_DISK_2`	Same as the levels above, but replicate each partition on two cluster nodes.

Table A1: Storage level of RDD

transformations		
map(f : T → U)	:	`RDD[T]` → `RDD[U]`
filter(f : T → Bool)	:	`RDD[T]` → `RDD[T]`
flatMap(f : T → Seq[U])	:	`RDD[T]` → `RDD[U]`
sample($fraction$: Float)	:	`RDD[T]` → `RDD[T]` (Deterministic sampling)
groupByKey()	:	`RDD[(K, V)]` → `RDD[(K, Seq[V])]`
reduceByKey(f : (V, V) → V)	:	`RDD[(K, V)]` → `RDD[(K, V)]`
union()	:	`(RDD[T], RDD[T])` → `RDD[T]`
join()	:	`(RDD[(K, V)], RDD[(K, W)])` → `RDD[(K, (V, W))]`
cogroup()	:	`(RDD[(K, V)], RDD[(K, W)])` → `RDD[(K, (Seq[V], Seq[W]))]`
crossProduct()	:	`(RDD[T], RDD[U])` → `RDD[(T, U)]`
mapValues(f : V → W)	:	`RDD[(K, V)]` → `RDD[(K, W)]` (Preserves partitioning)
sort(c : Comparator[K])	:	`RDD[(K, V)]` → `RDD[(K, V)]`
partitionBy(p : Partitioner[K])	:	`RDD[(K, V)]` → `RDD[(K, V)]`
actions		
count()	:	`RDD[T]` → `Long`
collect()	:	`RDD[T]` → `Seq[T]`
reduce(f : (T, T) → T)	:	`RDD[T]` → `T`
lookup(k : K)	:	`RDD[(K, V)]` → `Seq[V]` (On hash/range partitioned RDDs)
save($path$: String)	:	Outputs RDD to a storage system, e.g., HDFS

Table A2: Transformations and actions

BIBLIOGRAPHY

[1] Amazon EC2 Instance Purchasing Options. http://aws.amazon.com/ec2/purchasing-options/.

[2] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[3] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[4] Microsoft Azure. https://azure.microsoft.com/.

[5] Sample Price History of Spot Instance. https://github.com/JonnyCE/project-platform/tree/master/price_history.

[6] Spark Configuration. http://spark.apache.org/docs/latest/configuration.html.

[7] Apache Hadoop. HDFS Architecture. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[8] Apache Hadoop Project. Hadoop mapreduce. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[9] Bu, Yingyi, Howe, Bill, Balazinska, Magdalena, and Ernst, Michael D. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 285–296.

[10] Buyya, Rajkumar, Yeo, Chee Shin, and Venugopal, Srikumar. Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. *arXiv.org* (Aug. 2008).

[11] Chohan, N, Castillo, C, Spreitzer, M, and Steinder, M. See Spot Run: Using Spot Instances for MapReduce Workflows. *HotCloud* (2010).

[12] Daly, John. A model for predicting the optimum checkpoint interval for restart dumps. In *ICCS'03: Proceedings of the 2003 international conference on Computational science* (June 2003), Raytheon, Springer-Verlag, pp. 3–12.

[13] Daly, John T. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems 22*, 3 (2006), 303–312.

[14] École Polytechnique Fédérale de Lausanne (EPFL). Scala. http://www.scala-lang.org.

[15] Google Cloud Platform. Preemptible Instancee. https://cloud.google.com/compute/docs/instances/preemptible.

[16] He, Xin, Shenoy, Prashant, Sitaraman, Ramesh, and Irwin, David. Cutting the cost of hosting online services using cloud spot markets. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), ACM, pp. 207–218.

[17] Hindman, B, Konwinski, A, Zaharia, M, and Ghodsi, A. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *NSDI* (2011).

[18] Institute for Telecommunication Sciences. Federal Standard 1037C: Glossary of Telecommunications Terms - Avalibility. http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm.

[19] Lagar-Cavilla, Horacio Andrés, Whitney, Joseph Andrew, Scannell, Adin Matthew, Patchin, Philip, Rumble, Stephen M, de Lara, Eyal, Brudno, Michael, and Satyanarayanan, Mahadev. *SnowFlock: rapid virtual machine cloning for cloud computing*. ACM, Apr. 2009.

[20] Liu, H. Cutting MapReduce Cost with Spot Market. *HotCloud* (2011).

[21] Malewicz, Grzegorz, Austern, Matthew H, Bik, Aart JC, Dehnert, James C, Horn, Ilan, Leiser, Naty, and Czajkowski, Grzegorz. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[22] Mattess, M, Vecchiola, C, and Buyya, R. Managing peak loads by leasing cloud infrastructure services from a spot market. *... (HPCC)* (2010), 180–188.

[23] Salama, Abdallah, Binnig, Carsten, Kraska, Tim, and Zamanian, Erfan. *Cost-based Fault-tolerance for Parallel Data Processing*. ACM, New York, New York, USA, May 2015.

[24] Sharma, Prateek, Irwin, David, and Shenoy, Prashant. How not to bid the cloud. *University of Massachusetts Technical Report UMCS-2016-002* (2016).

[25] Subramanya, Supreeth, Guo, Tian, Sharma, Prateek, Irwin, David, and Shenoy, Prashant. SpotOn: a batch computing service for the spot market. In *SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing* (Aug. 2015), University of Massachusetts Amherst, ACM.

[26] Yan, Ying, Gao, Yanjie, Chen, Yang, Guo, Zhongxin, Chen, Bole, and Moscibroda, Thomas. Tr-spark: Transient computing for big data analytics. accepted in 2016 ACM Symposium on Cloud Computing, 2016.

[27] Yi, S, Kondo, D, and Andrzejak, A. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. *2010 IEEE 3rd International . . .* (2010).

[28] Yi, Sangho, Kondo, Derrick, and Andrzejak, Artur. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. *2010 IEEE International Conference on Cloud Computing (CLOUD)* (2010), 236–243.

[29] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J, Shenker, Scott, and Stoica, Ion. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Apr. 2012), USENIX Association, pp. 2–2.

[30] Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J, Shenker, Scott, and Stoica, Ion. Spark: cluster computing with working sets. In *HotCloud'10: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (June 2010), USENIX Association.

[31] Zhang, Irene, Garthwaite, Alex, Baskakov, Yury, Barr, Kenneth C, Zhang, Irene, Garthwaite, Alex, Baskakov, Yury, and Barr, Kenneth C. *Fast restore of checkpointed memory using working set estimation*, vol. 46. ACM, July 2011.

[32] Zheng, Liang, Joe-Wong, Carlee, Tan, Chee Wei, Chiang, Mung, and Wang, Xinyu. How to Bid the Cloud. In *SIGCOMM '15: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, New York, USA, Aug. 2015), ACM Request Permissions, pp. 71–84.

[33] Zhou, Yanqi, Wentzlaff, David, Zhou, Yanqi, Wentzlaff, David, Zhou, Yanqi, and Wentzlaff, David. The sharing architecture: sub-core configurability for IaaS clouds. *ACM SIGARCH Computer Architecture News 42*, 1 (Apr. 2014), 559–574.