Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Trond Humborstad

Database and storage layer integration for cloud platforms

Master's thesis in Computer Science
Supervisor: Jon Olav Hauglid, Øystein Grøvlen
June 2019

NTNU
Kunnskap for en bedre verden

Trond Humborstad

Database and storage layer integration for cloud platforms

Master's thesis in Computer Science
Supervisor: Jon Olav Hauglid, Øystein Grøvlen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Preface

I would like to express my sincere thanks to my advisor at Oracle, Jon Olav Hauglid, for motivating me and reviewing my progress for my master thesis, as well as providing much valued resources and insights for the Amazon Web Services platform. I would also like to thank Alibaba and Øystein Grøvlen for providing preview access to the PolarDB database service, as well as input and feedback on drafts of this report.

Trond Humborstad

June 2019

Abstract

Services and software deployments have in recent years increasingly moved to the cloud. Platforms such as Amazon and Alibaba offer databases as a managed service. These databases must be able to scale well to meet customers needs. One of the innovations done is to separate the compute and storage components of the traditional database system, allowing for independent scaling of these components. Various other innovations have also been made to facilitate the underlying architecture. We will in this paper present some of the systems that have made these changes, and see where their approaches are similar and where they differ. We also present benchmark tests to see how well these systems perform under various workloads, and tie these results up to the architectural changes made. This gives an insight into how well theory matches practice. Alibaba has additionally granted exclusive access to their PolarDB database offering, which is at this point only commercially available in Asia. The results show that while Amazon Aurora performs reasonably well; the benchmarks are slightly lower than the results obtained by Amazon. We will see that Aurora is first able to perform well with many database connections. The RDS MySQL service is able to follow Aurora in terms of read and write performance, and sometimes outperform Aurora on similar hardware. PolarDB show strong read and write performance, and tops out most tests when compared to similar hardware from the Amazon services.

Sammendrag

Tjenester og programvareløsninger har i senere år sett en overgang til skyplattformer. Flere plattformer, som Amazon og Alibaba, tilbyr databaseløsninger som egne tjenester. Disse databasene må kunne skalere godt for å møte forventningene til krevende kunder. En av innovasjonene som er gjort er å separerere databehandlingslaget fra lagringslaget, og slik tillate uavhengig skalering av disse. Andre grep som typisk gjøres, er å tilpasse systemene for å passe inn med den underforliggende arkitekturen til skyplattformene. Vi vil i denne oppgaven presentere enkelte av de databasessystemene som har gjort disse endringene, og se hvor tilnærmingene deler fellestrekk, og hvor de skiller seg fra hverandre. Vi vil også utføre ytelsestester under forskjellig last, og knytte resultatene opp mot endringene i arkitektur. Alibaba har i tillegg gitt eksklusiv tilgang til tjenesten PolarDB, som på dette tidspunktet i utgangspunktet kun er tilgjengelig i Asia. Resultatene viser at Amazon Aurora yter relativt bra, men tallene er noe lavere enn hva Amazon selv oppgir. Vi vil se at Aurora først yter bra ved mange tilkoblinger til databasen. RDS MySQL-tjenesten er på mange måter kapabel i å følge Aurora både for lese og skrivelast, og forbigår av og til Aurora. PolarDB viser god lese og skriveytelse, og kommer ut på topp for de fleste tester når sammenlignet mot lignende maskinvare og oppsett som Amazons tjenester.

Contents

1 Introduction **1**
1.1 Report structure . . . 2

2 What is the cloud, and how does it work? **3**
2.1 Shared nothing vs. shared disk . . . 4

3 Background theory **6**
3.1 ACID and BASE - two design philosophies . . . 6
3.2 CAP theorem . . . 7
3.3 Remote direct memory access (RDMA) . . . 9
3.4 NVMe disks . . . 10
3.5 NoSQL and NewSQL systems . . . 10
3.6 Distributed consensus . . . 12
3.7 Paxos . . . 12
3.8 Raft . . . 14
3.9 Quorum models . . . 18
3.9.1 Quorumbased voting in commit protocols . . . 19
3.9.2 Quorumbased voting for replica control . . . 19

4 Amazon RDS **20**
4.1 Storage solutions for RDS . . . 21
4.1.1 General purpose SSD's . . . 21
4.1.2 Provisional IOPS SSD's . . . 22
4.1.3 Magnetic storage . . . 22
4.2 Read replicas and hot-standby replicas . . . 22
4.2.1 Single AZ deployments . . . 24
4.3 Logging and performance metrics in RDS . . . 25

5 Amazon Aurora **27**

6 Alibaba PolarDB **32**

7 Benchmarking RDS MySQL, AWS Aurora, and Alibaba PolarDB **40**
7.1 Relevant applications and tools . . . 40
7.1.1 Terraform . . . 40
7.1.2 Sysbench . . . 41
7.2 Test setup . . . 43
7.2.1 Amazon AWS . . . 43
7.2.2 PolarDB . . . 46

8 Results **48**
8.1 Results with 250 tables, 25000 records per table 49
8.1.1 Amazon Aurora 50
8.1.2 Amazon RDS MySQL (Multi AZ) 54
8.1.3 PolarDB 57
8.1.4 Summary - 250 tables, 25000 records per table 59
8.2 Results with 250 tables, 2700000 records per table 60
8.2.1 Amazon Aurora 61
8.2.2 Amazon RDS MySQL (Multi AZ) 64
8.2.3 PolarDB 67
8.2.4 Summary - 250 tables, 2700000 records per table 68
8.3 Comparison 69
8.3.1 How RDS MySQL (both with and without Multi AZ), Aurora, and PolarDB compare under similar circumstances 69
8.3.2 How Aurora and PolarDB compares with 250 tables and 200000 records per table data set (10.8GB) 72
8.3.3 How RDS MySQL (both with and without Multi AZ), Aurora, and PolarDB compare with high thread counts 73

9 Discussion **76**
9.1 Amazon Aurora 77
9.2 Alibaba PolarDB 82
9.3 Comparison 84

10 Future work **87**
10.1 Look into other cloud database systems 87
10.2 Test other database configurations 87

11 Conclusion **88**

Appendices **89**

A Terraform test setup **89**
A.1 Terraform files for Aurora 89
A.2 Terraform files for MySQL 94

B Sysbench results **101**
B.1 mysql-db.r5.large-32-read-run-1 101
B.2 mysql-db.r5.large-32-read-run-2 101
B.3 mysql-db.r5.large-32-read-run-3 101
B.4 mysql-db.r5.large-32-read-run-4 101

B.5 mysql-db.r5.large-32-read-run-5 101
B.6 mysql-db.r5.xlarge-32-read-run-1 101
B.7 mysql-db.r5.xlarge-32-read-run-2 102
B.8 mysql-db.r5.xlarge-32-read-run-3 102
B.9 mysql-db.r5.xlarge-32-read-run-4 102
B.10 mysql-db.r5.xlarge-32-read-run-5 102
B.11 mysql-db.r5.xlarge-32-read-run-1 102
B.12 mysql-db.r5.xlarge-32-read-run-2 102
B.13 mysql-db.r5.xlarge-32-read-run-3 103
B.14 mysql-db.r5.xlarge-32-read-run-4 103
B.15 mysql-db.r5.xlarge-32-read-run-5 103
B.16 mysql-db.r5.4xlarge-32-read-run-1 103
B.17 mysql-db.r5.4xlarge-32-read-run-2 103
B.18 mysql-db.r5.4xlarge-32-read-run-3 103
B.19 mysql-db.r5.4xlarge-32-read-run-4 104
B.20 mysql-db.r5.4xlarge-32-read-run-5 104
B.21 mysql-multiaz-db.r5.large-16-read-run-1 104
B.22 mysql-multiaz-db.r5.large-16-read-run-2 104
B.23 mysql-multiaz-db.r5.large-16-read-run-3 104
B.24 mysql-multiaz-db.r5.large-16-read-run-4 104
B.25 mysql-multiaz-db.r5.large-16-read-run-5 105
B.26 mysql-multiaz-db.r5.xlarge-16-read-run-1 105
B.27 mysql-multiaz-db.r5.xlarge-16-read-run-2 105
B.28 mysql-multiaz-db.r5.xlarge-16-read-run-3 105
B.29 mysql-multiaz-db.r5.xlarge-16-read-run-4 105
B.30 mysql-multiaz-db.r5.xlarge-16-read-run-5 105
B.31 mysql-multiaz-db.r5.xlarge-16-read-run-1 106
B.32 mysql-multiaz-db.r5.xlarge-16-read-run-2 106
B.33 mysql-multiaz-db.r5.xlarge-16-read-run-3 106
B.34 mysql-multiaz-db.r5.xlarge-16-read-run-4 106
B.35 mysql-multiaz-db.r5.xlarge-16-read-run-5 106
B.36 mysql-multiaz-db.r5.4xlarge-16-read-run-1 106
B.37 mysql-multiaz-db.r5.4xlarge-16-read-run-2 107
B.38 mysql-multiaz-db.r5.4xlarge-16-read-run-3 107
B.39 mysql-multiaz-db.r5.4xlarge-16-read-run-4 107
B.40 mysql-multiaz-db.r5.4xlarge-16-read-run-5 107
B.41 mysql-multiaz-db.r5.large-32-read-run-1 107
B.42 mysql-multiaz-db.r5.large-32-read-run-2 107
B.43 mysql-multiaz-db.r5.large-32-read-run-3 108
B.44 mysql-multiaz-db.r5.large-32-read-run-4 108

B.45 mysql-multiaz-db.r5.large-32-read-run-5 108
B.46 mysql-multiaz-db.r5.xlarge-32-read-run-1 108
B.47 mysql-multiaz-db.r5.xlarge-32-read-run-2 108
B.48 mysql-multiaz-db.r5.xlarge-32-read-run-3 108
B.49 mysql-multiaz-db.r5.xlarge-32-read-run-4 109
B.50 mysql-multiaz-db.r5.xlarge-32-read-run-5 109
B.51 mysql-multiaz-db.r5.xlarge-32-read-run-1 109
B.52 mysql-multiaz-db.r5.xlarge-32-read-run-2 109
B.53 mysql-multiaz-db.r5.xlarge-32-read-run-3 109
B.54 mysql-multiaz-db.r5.xlarge-32-read-run-4 109
B.55 mysql-multiaz-db.r5.xlarge-32-read-run-5 110
B.56 mysql-multiaz-db.r5.4xlarge-32-read-run-1 110
B.57 mysql-multiaz-db.r5.4xlarge-32-read-run-2 110
B.58 mysql-multiaz-db.r5.4xlarge-32-read-run-3 110
B.59 mysql-multiaz-db.r5.4xlarge-32-read-run-4 110
B.60 mysql-multiaz-db.r5.4xlarge-32-read-run-5 110
B.61 mysql-multiaz-db.r5.large-64-read-run-1 111
B.62 mysql-multiaz-db.r5.large-64-read-run-2 111
B.63 mysql-multiaz-db.r5.large-64-read-run-3 111
B.64 mysql-multiaz-db.r5.large-64-read-run-4 111
B.65 mysql-multiaz-db.r5.large-64-read-run-5 111
B.66 mysql-multiaz-db.r5.xlarge-64-read-run-1 111
B.67 mysql-multiaz-db.r5.xlarge-64-read-run-2 112
B.68 mysql-multiaz-db.r5.xlarge-64-read-run-3 112
B.69 mysql-multiaz-db.r5.xlarge-64-read-run-4 112
B.70 mysql-multiaz-db.r5.xlarge-64-read-run-5 112
B.71 mysql-multiaz-db.r5.xlarge-64-read-run-1 112
B.72 mysql-multiaz-db.r5.xlarge-64-read-run-2 112
B.73 mysql-multiaz-db.r5.xlarge-64-read-run-3 113
B.74 mysql-multiaz-db.r5.xlarge-64-read-run-4 113
B.75 mysql-multiaz-db.r5.xlarge-64-read-run-5 113
B.76 mysql-multiaz-db.r5.4xlarge-64-read-run-1 113
B.77 mysql-multiaz-db.r5.4xlarge-64-read-run-2 113
B.78 mysql-multiaz-db.r5.4xlarge-64-read-run-3 113
B.79 mysql-multiaz-db.r5.4xlarge-64-read-run-4 114
B.80 mysql-multiaz-db.r5.4xlarge-64-read-run-5 114
B.81 aurora-db.r5.large-16-read-run-1 114
B.82 aurora-db.r5.large-16-read-run-2 114
B.83 aurora-db.r5.large-16-read-run-3 114
B.84 aurora-db.r5.large-16-read-run-4 114

B.85 aurora-db.r5.large-16-read-run-5 115
B.86 aurora-db.r5.xlarge-16-read-run-1 115
B.87 aurora-db.r5.xlarge-16-read-run-2 115
B.88 aurora-db.r5.xlarge-16-read-run-3 115
B.89 aurora-db.r5.xlarge-16-read-run-4 115
B.90 aurora-db.r5.xlarge-16-read-run-5 115
B.91 aurora-db.r5.2xlarge-16-read-run-1 116
B.92 aurora-db.r5.2xlarge-16-read-run-2 116
B.93 aurora-db.r5.2xlarge-16-read-run-3 116
B.94 aurora-db.r5.2xlarge-16-read-run-4 116
B.95 aurora-db.r5.2xlarge-16-read-run-5 116
B.96 aurora-db.r5.4xlarge-16-read-run-1 116
B.97 aurora-db.r5.4xlarge-16-read-run-2 117
B.98 aurora-db.r5.4xlarge-16-read-run-3 117
B.99 aurora-db.r5.4xlarge-16-read-run-4 117
B.100 aurora-db.r5.4xlarge-16-read-run-5 117
B.101 aurora-db.r5.large-32-read-run-1 117
B.102 aurora-db.r5.large-32-read-run-2 117
B.103 aurora-db.r5.large-32-read-run-3 118
B.104 aurora-db.r5.large-32-read-run-4 118
B.105 aurora-db.r5.large-32-read-run-5 118
B.106 aurora-db.r5.xlarge-32-read-run-1 118
B.107 aurora-db.r5.xlarge-32-read-run-2 118
B.108 aurora-db.r5.xlarge-32-read-run-3 118
B.109 aurora-db.r5.xlarge-32-read-run-4 119
B.110 aurora-db.r5.xlarge-32-read-run-5 119
B.111 aurora-db.r5.2xlarge-32-read-run-1 119
B.112 aurora-db.r5.2xlarge-32-read-run-2 119
B.113 aurora-db.r5.2xlarge-32-read-run-3 119
B.114 aurora-db.r5.2xlarge-32-read-run-4 119
B.115 aurora-db.r5.2xlarge-32-read-run-5 120
B.116 aurora-db.r5.4xlarge-32-read-run-1 120
B.117 aurora-db.r5.4xlarge-32-read-run-2 120
B.118 aurora-db.r5.4xlarge-32-read-run-3 120
B.119 aurora-db.r5.4xlarge-32-read-run-4 120
B.120 aurora-db.r5.4xlarge-32-read-run-5 120
B.121 aurora-db.r5.large-64-read-run-1 121
B.122 aurora-db.r5.large-64-read-run-2 121
B.123 aurora-db.r5.large-64-read-run-3 121
B.124 aurora-db.r5.large-64-read-run-4 121

B.125 aurora-db.r5.large-64-read-run-5 121
B.126 aurora-db.r5.xlarge-64-read-run-1 121
B.127 aurora-db.r5.xlarge-64-read-run-2 122
B.128 aurora-db.r5.xlarge-64-read-run-3 122
B.129 aurora-db.r5.xlarge-64-read-run-4 122
B.130 aurora-db.r5.xlarge-64-read-run-5 122
B.131 aurora-db.r5.2xlarge-64-read-run-1 122
B.132 aurora-db.r5.2xlarge-64-read-run-2 122
B.133 aurora-db.r5.2xlarge-64-read-run-3 123
B.134 aurora-db.r5.2xlarge-64-read-run-4 123
B.135 aurora-db.r5.2xlarge-64-read-run-5 123
B.136 aurora-db.r5.4xlarge-64-read-run-1 123
B.137 aurora-db.r5.4xlarge-64-read-run-2 123
B.138 aurora-db.r5.4xlarge-64-read-run-3 123
B.139 aurora-db.r5.4xlarge-64-read-run-4 124
B.140 aurora-db.r5.4xlarge-64-read-run-5 124
B.141 polardb-manythreads-2-read-run-1 124
B.142 polardb-manythreads-2-read-run-2 124
B.143 polardb-manythreads-2-read-run-3 124
B.144 polardb-manythreads-2-read-run-4 124
B.145 polardb-manythreads-2-read-run-5 125
B.146 polardb-manythreads-4-read-run-1 125
B.147 polardb-manythreads-4-read-run-2 125
B.148 polardb-manythreads-4-read-run-3 125
B.149 polardb-manythreads-4-read-run-4 125
B.150 polardb-manythreads-4-read-run-5 125
B.151 polardb-manythreads-8-read-run-1 126
B.152 polardb-manythreads-8-read-run-2 126
B.153 polardb-manythreads-8-read-run-3 126
B.154 polardb-manythreads-8-read-run-4 126
B.155 polardb-manythreads-8-read-run-5 126
B.156 polardb-manythreads-16-read-run-1 126
B.157 polardb-manythreads-16-read-run-2 127
B.158 polardb-manythreads-16-read-run-3 127
B.159 polardb-manythreads-16-read-run-4 127
B.160 polardb-manythreads-16-read-run-5 127
B.161 polardb-manythreads-32-read-run-1 127
B.162 polardb-manythreads-32-read-run-2 127
B.163 polardb-manythreads-32-read-run-3 128
B.164 polardb-manythreads-32-read-run-4 128

B.165 polardb-manythreads-32-read-run-5 128
B.166 polardb-manythreads-64-read-run-1 128
B.167 polardb-manythreads-64-read-run-2 128
B.168 polardb-manythreads-64-read-run-3 128
B.169 polardb-manythreads-64-read-run-4 129
B.170 polardb-manythreads-64-read-run-5 129
B.171 polardb-150GB-2-read-run-1 129
B.172 polardb-150GB-2-read-run-2 129
B.173 polardb-150GB-2-read-run-3 129
B.174 polardb-150GB-2-read-run-4 129
B.175 polardb-150GB-2-read-run-5 130
B.176 polardb-150GB-4-read-run-1 130
B.177 polardb-150GB-4-read-run-2 130
B.178 polardb-150GB-4-read-run-3 130
B.179 polardb-150GB-4-read-run-4 130
B.180 polardb-150GB-4-read-run-5 130
B.181 polardb-150GB-8-read-run-1 131
B.182 polardb-150GB-8-read-run-2 131
B.183 polardb-150GB-8-read-run-3 131
B.184 polardb-150GB-8-read-run-4 131
B.185 polardb-150GB-8-read-run-5 131
B.186 polardb-150GB-16-read-run-1 131
B.187 polardb-150GB-16-read-run-2 132
B.188 polardb-150GB-16-read-run-3 132
B.189 polardb-150GB-16-read-run-4 132
B.190 polardb-150GB-16-read-run-5 132
B.191 polardb-150GB-32-read-run-1 132
B.192 polardb-150GB-32-read-run-2 132
B.193 polardb-150GB-32-read-run-3 133
B.194 polardb-150GB-32-read-run-4 133
B.195 polardb-150GB-32-read-run-5 133
B.196 polardb-150GB-64-read-run-1 133
B.197 polardb-150GB-64-read-run-2 133
B.198 polardb-150GB-64-read-run-3 133
B.199 polardb-150GB-64-read-run-4 134
B.200 polardb-150GB-64-read-run-5 134
B.201 polardb-manythreads-1GB-500-read-run-1 134
B.202 polardb-manythreads-1GB-500-read-run-2 134
B.203 polardb-manythreads-1GB-500-read-run-3 134
B.204 polardb-manythreads-1GB-500-read-run-4 134

B.205 polardb-manythreads-1GB-500-read-run-5 135
B.206 mysql-db.r5.large-32-write-run-1 135
B.207 mysql-db.r5.large-32-write-run-2 135
B.208 mysql-db.r5.large-32-write-run-3 135
B.209 mysql-db.r5.large-32-write-run-4 135
B.210 mysql-db.r5.large-32-write-run-5 135
B.211 mysql-db.r5.xlarge-32-write-run-1 136
B.212 mysql-db.r5.xlarge-32-write-run-2 136
B.213 mysql-db.r5.xlarge-32-write-run-3 136
B.214 mysql-db.r5.xlarge-32-write-run-4 136
B.215 mysql-db.r5.xlarge-32-write-run-5 136
B.216 mysql-db.r5.xlarge-32-write-run-1 136
B.217 mysql-db.r5.xlarge-32-write-run-2 137
B.218 mysql-db.r5.xlarge-32-write-run-3 137
B.219 mysql-db.r5.xlarge-32-write-run-4 137
B.220 mysql-db.r5.xlarge-32-write-run-5 137
B.221 mysql-db.r5.4xlarge-32-write-run-1 137
B.222 mysql-db.r5.4xlarge-32-write-run-2 137
B.223 mysql-db.r5.4xlarge-32-write-run-3 138
B.224 mysql-db.r5.4xlarge-32-write-run-4 138
B.225 mysql-db.r5.4xlarge-32-write-run-5 138
B.226 mysql-multiaz-db.r5.large-16-write-run-1 138
B.227 mysql-multiaz-db.r5.large-16-write-run-2 138
B.228 mysql-multiaz-db.r5.large-16-write-run-3 138
B.229 mysql-multiaz-db.r5.large-16-write-run-4 139
B.230 mysql-multiaz-db.r5.large-16-write-run-5 139
B.231 mysql-multiaz-db.r5.xlarge-16-write-run-1 139
B.232 mysql-multiaz-db.r5.xlarge-16-write-run-2 139
B.233 mysql-multiaz-db.r5.xlarge-16-write-run-3 139
B.234 mysql-multiaz-db.r5.xlarge-16-write-run-4 139
B.235 mysql-multiaz-db.r5.xlarge-16-write-run-5 140
B.236 mysql-multiaz-db.r5.xlarge-16-write-run-1 140
B.237 mysql-multiaz-db.r5.xlarge-16-write-run-2 140
B.238 mysql-multiaz-db.r5.xlarge-16-write-run-3 140
B.239 mysql-multiaz-db.r5.xlarge-16-write-run-4 140
B.240 mysql-multiaz-db.r5.xlarge-16-write-run-5 140
B.241 mysql-multiaz-db.r5.4xlarge-16-write-run-1 141
B.242 mysql-multiaz-db.r5.4xlarge-16-write-run-2 141
B.243 mysql-multiaz-db.r5.4xlarge-16-write-run-3 141
B.244 mysql-multiaz-db.r5.4xlarge-16-write-run-4 141

B.245 mysql-multiaz-db.r5.4xlarge-16-write-run-5 141
B.246 mysql-multiaz-db.r5.large-32-write-run-1 141
B.247 mysql-multiaz-db.r5.large-32-write-run-2 142
B.248 mysql-multiaz-db.r5.large-32-write-run-3 142
B.249 mysql-multiaz-db.r5.large-32-write-run-4 142
B.250 mysql-multiaz-db.r5.large-32-write-run-5 142
B.251 mysql-multiaz-db.r5.xlarge-32-write-run-1 142
B.252 mysql-multiaz-db.r5.xlarge-32-write-run-2 142
B.253 mysql-multiaz-db.r5.xlarge-32-write-run-3 143
B.254 mysql-multiaz-db.r5.xlarge-32-write-run-4 143
B.255 mysql-multiaz-db.r5.xlarge-32-write-run-5 143
B.256 mysql-multiaz-db.r5.xlarge-32-write-run-1 143
B.257 mysql-multiaz-db.r5.xlarge-32-write-run-2 143
B.258 mysql-multiaz-db.r5.xlarge-32-write-run-3 143
B.259 mysql-multiaz-db.r5.xlarge-32-write-run-4 144
B.260 mysql-multiaz-db.r5.xlarge-32-write-run-5 144
B.261 mysql-multiaz-db.r5.4xlarge-32-write-run-1 144
B.262 mysql-multiaz-db.r5.4xlarge-32-write-run-2 144
B.263 mysql-multiaz-db.r5.4xlarge-32-write-run-3 144
B.264 mysql-multiaz-db.r5.4xlarge-32-write-run-4 144
B.265 mysql-multiaz-db.r5.4xlarge-32-write-run-5 145
B.266 mysql-multiaz-db.r5.large-64-write-run-1 145
B.267 mysql-multiaz-db.r5.large-64-write-run-2 145
B.268 mysql-multiaz-db.r5.large-64-write-run-3 145
B.269 mysql-multiaz-db.r5.large-64-write-run-4 145
B.270 mysql-multiaz-db.r5.large-64-write-run-5 145
B.271 mysql-multiaz-db.r5.xlarge-64-write-run-1 146
B.272 mysql-multiaz-db.r5.xlarge-64-write-run-2 146
B.273 mysql-multiaz-db.r5.xlarge-64-write-run-3 146
B.274 mysql-multiaz-db.r5.xlarge-64-write-run-4 146
B.275 mysql-multiaz-db.r5.xlarge-64-write-run-5 146
B.276 mysql-multiaz-db.r5.xlarge-64-write-run-1 146
B.277 mysql-multiaz-db.r5.xlarge-64-write-run-2 147
B.278 mysql-multiaz-db.r5.xlarge-64-write-run-3 147
B.279 mysql-multiaz-db.r5.xlarge-64-write-run-4 147
B.280 mysql-multiaz-db.r5.xlarge-64-write-run-5 147
B.281 mysql-multiaz-db.r5.4xlarge-64-write-run-1 147
B.282 mysql-multiaz-db.r5.4xlarge-64-write-run-2 147
B.283 mysql-multiaz-db.r5.4xlarge-64-write-run-3 148
B.284 mysql-multiaz-db.r5.4xlarge-64-write-run-4 148

B.285 mysql-multiaz-db.r5.4xlarge-64-write-run-5 148
B.286 aurora-db.r5.large-16-write-run-1 148
B.287 aurora-db.r5.large-16-write-run-2 148
B.288 aurora-db.r5.large-16-write-run-3 148
B.289 aurora-db.r5.large-16-write-run-4 149
B.290 aurora-db.r5.large-16-write-run-5 149
B.291 aurora-db.r5.xlarge-16-write-run-1 149
B.292 aurora-db.r5.xlarge-16-write-run-2 149
B.293 aurora-db.r5.xlarge-16-write-run-3 149
B.294 aurora-db.r5.xlarge-16-write-run-4 149
B.295 aurora-db.r5.xlarge-16-write-run-5 150
B.296 aurora-db.r5.2xlarge-16-write-run-1 150
B.297 aurora-db.r5.2xlarge-16-write-run-2 150
B.298 aurora-db.r5.2xlarge-16-write-run-3 150
B.299 aurora-db.r5.2xlarge-16-write-run-4 150
B.300 aurora-db.r5.2xlarge-16-write-run-5 150
B.301 aurora-db.r5.4xlarge-16-write-run-1 151
B.302 aurora-db.r5.4xlarge-16-write-run-2 151
B.303 aurora-db.r5.4xlarge-16-write-run-3 151
B.304 aurora-db.r5.4xlarge-16-write-run-4 151
B.305 aurora-db.r5.4xlarge-16-write-run-5 151
B.306 aurora-db.r5.large-32-write-run-1 151
B.307 aurora-db.r5.large-32-write-run-2 152
B.308 aurora-db.r5.large-32-write-run-3 152
B.309 aurora-db.r5.large-32-write-run-4 152
B.310 aurora-db.r5.large-32-write-run-5 152
B.311 aurora-db.r5.xlarge-32-write-run-1 152
B.312 aurora-db.r5.xlarge-32-write-run-2 152
B.313 aurora-db.r5.xlarge-32-write-run-3 153
B.314 aurora-db.r5.xlarge-32-write-run-4 153
B.315 aurora-db.r5.xlarge-32-write-run-5 153
B.316 aurora-db.r5.2xlarge-32-write-run-1 153
B.317 aurora-db.r5.2xlarge-32-write-run-2 153
B.318 aurora-db.r5.2xlarge-32-write-run-3 153
B.319 aurora-db.r5.2xlarge-32-write-run-4 154
B.320 aurora-db.r5.2xlarge-32-write-run-5 154
B.321 aurora-db.r5.4xlarge-32-write-run-1 154
B.322 aurora-db.r5.4xlarge-32-write-run-2 154
B.323 aurora-db.r5.4xlarge-32-write-run-3 154
B.324 aurora-db.r5.4xlarge-32-write-run-4 154

B.325 aurora-db.r5.4xlarge-32-write-run-5 155
B.326 aurora-db.r5.large-64-write-run-1 155
B.327 aurora-db.r5.large-64-write-run-2 155
B.328 aurora-db.r5.large-64-write-run-3 155
B.329 aurora-db.r5.large-64-write-run-4 155
B.330 aurora-db.r5.large-64-write-run-5 155
B.331 aurora-db.r5.xlarge-64-write-run-1 156
B.332 aurora-db.r5.xlarge-64-write-run-2 156
B.333 aurora-db.r5.xlarge-64-write-run-3 156
B.334 aurora-db.r5.xlarge-64-write-run-4 156
B.335 aurora-db.r5.xlarge-64-write-run-5 156
B.336 aurora-db.r5.2xlarge-64-write-run-1 156
B.337 aurora-db.r5.2xlarge-64-write-run-2 157
B.338 aurora-db.r5.2xlarge-64-write-run-3 157
B.339 aurora-db.r5.2xlarge-64-write-run-4 157
B.340 aurora-db.r5.2xlarge-64-write-run-5 157
B.341 aurora-db.r5.4xlarge-64-write-run-1 157
B.342 aurora-db.r5.4xlarge-64-write-run-2 157
B.343 aurora-db.r5.4xlarge-64-write-run-3 158
B.344 aurora-db.r5.4xlarge-64-write-run-4 158
B.345 aurora-db.r5.4xlarge-64-write-run-5 158
B.346 polardb-manythreads-2-write-run-1 158
B.347 polardb-manythreads-2-write-run-2 158
B.348 polardb-manythreads-2-write-run-3 158
B.349 polardb-manythreads-2-write-run-4 159
B.350 polardb-manythreads-2-write-run-5 159
B.351 polardb-manythreads-4-write-run-1 159
B.352 polardb-manythreads-4-write-run-2 159
B.353 polardb-manythreads-4-write-run-3 159
B.354 polardb-manythreads-4-write-run-4 159
B.355 polardb-manythreads-4-write-run-5 160
B.356 polardb-manythreads-8-write-run-1 160
B.357 polardb-manythreads-8-write-run-2 160
B.358 polardb-manythreads-8-write-run-3 160
B.359 polardb-manythreads-8-write-run-4 160
B.360 polardb-manythreads-8-write-run-5 160
B.361 polardb-manythreads-16-write-run-1 161
B.362 polardb-manythreads-16-write-run-2 161
B.363 polardb-manythreads-16-write-run-3 161
B.364 polardb-manythreads-16-write-run-4 161

B.365 polardb-manythreads-16-write-run-5 161
B.366 polardb-manythreads-32-write-run-1 161
B.367 polardb-manythreads-32-write-run-2 162
B.368 polardb-manythreads-32-write-run-3 162
B.369 polardb-manythreads-32-write-run-4 162
B.370 polardb-manythreads-32-write-run-5 162
B.371 polardb-manythreads-64-write-run-1 162
B.372 polardb-manythreads-64-write-run-2 162
B.373 polardb-manythreads-64-write-run-3 163
B.374 polardb-manythreads-64-write-run-4 163
B.375 polardb-manythreads-64-write-run-5 163
B.376 polardb-150GB-2-write-run-1 163
B.377 polardb-150GB-2-write-run-2 163
B.378 polardb-150GB-2-write-run-3 163
B.379 polardb-150GB-2-write-run-4 164
B.380 polardb-150GB-2-write-run-5 164
B.381 polardb-150GB-4-write-run-1 164
B.382 polardb-150GB-4-write-run-2 164
B.383 polardb-150GB-4-write-run-3 164
B.384 polardb-150GB-4-write-run-4 164
B.385 polardb-150GB-4-write-run-5 165
B.386 polardb-150GB-8-write-run-1 165
B.387 polardb-150GB-8-write-run-2 165
B.388 polardb-150GB-8-write-run-3 165
B.389 polardb-150GB-8-write-run-4 165
B.390 polardb-150GB-8-write-run-5 165
B.391 polardb-150GB-16-write-run-1 166
B.392 polardb-150GB-16-write-run-2 166
B.393 polardb-150GB-16-write-run-3 166
B.394 polardb-150GB-16-write-run-4 166
B.395 polardb-150GB-16-write-run-5 166
B.396 polardb-150GB-32-write-run-1 166
B.397 polardb-150GB-32-write-run-2 167
B.398 polardb-150GB-32-write-run-3 167
B.399 polardb-150GB-32-write-run-4 167
B.400 polardb-150GB-32-write-run-5 167
B.401 polardb-150GB-64-write-run-1 167
B.402 polardb-150GB-64-write-run-2 167
B.403 polardb-150GB-64-write-run-3 168
B.404 polardb-150GB-64-write-run-4 168

B.405 polardb-150GB-64-write-run-5 168
B.406 polardb-manythreads-1GB-500-write-run-1 168
B.407 polardb-manythreads-1GB-500-write-run-2 168
B.408 polardb-manythreads-1GB-500-write-run-3 168
B.409 polardb-manythreads-1GB-500-write-run-4 169
B.410 polardb-manythreads-1GB-500-write-run-5 169

References **170**

List of Figures

1 Differences between on-premise, IaaS, PaaS, and SaaS. Source: [24] . 3
2 AWS regions and availability zones Source: [21] 20
3 AWS RDS Multi AZ Architecture Source: [35] 23
4 Example dashboard in AWS Performance Insights 25
5 Example dashboard in AWS CloudWatch 26
6 Segments and protection groups in Aurora 28
7 Overall PolarDB architecture. Source: [7] 32
8 PolarFS - I/O throughput under different loads. Source: [7] 38
9 PolarDB - read/writes per second. Source: [7] 39
10 AWS test setup 44
11 Aurora read performance 50
12 Aurora write performance 52
13 MySQL (Multi AZ) read performance 54
14 MySQL (Multi AZ) write performance 56
15 PolarDB read performance 58
16 PolarDB write performance 59
17 Aurora read performance 61
18 Aurora write performance 62
19 MySQL (Multi AZ) read performance 64
20 MySQL (Multi AZ) write performance 65
21 PolarDB read performance 67
22 PolarDB write performance 68
23 Comparison read performance 70
24 Comparison write performance 71
25 Comparison read performance 72
26 Comparison write performance 73
27 Comparison read performance 74
28 Comparison write performance 75
29 Network IO for mirrored MySQL. Source: [2] 77
30 Network IO for Aurora. Source: [2] 78
31 PolarFS - I/O latency under different loads. Source: [7] 84

1 Introduction

The cloud has changed much of how large scale computing is done. Databases have been faced with new challenges for dealing with scalability and performance for the new workloads of an increasingly online world. New systems following the NoSQL movement have come to the market to solve some of these issues, where traditional database systems have lagged behind. However, relational-based systems have seen developments. Increasingly, cloud databases have gained foothold, pioneered by Amazon Aurora which was introduced in 2015. Other competitors have also followed suit, with Alibaba's PolarDB as an example.

Some major trends can be identified for the new cloud based distributed database systems. Modern databases have realized the need to be able to scale dynamically. One approach to implement this, is to decouple the compute and storage components of the system, allowing for independent scaling of the two respective layers. The cloud systems also use space efficient shared disk architectures, where instances within a cluster access a single database image. Similarly, network bandwidth is in many cases a limited resource, and systems might therefore move functionality to the storage layer to perform operations close to data and reduce usage of network bandwidth. Systems also make use of redundant storage of data to ensure resiliency against failures and to improve availability. This paper will take a closer look at two systems that fall in this category; Amazon Aurora and Alibaba PolarDB. The systems have chosen different approaches to solve the points mentioned above, with various following trade-offs. This paper will revolve around how these approaches differ and where they share similarities.

In this paper we will present the Amazon Aurora and Alibaba PolarDB offerings, before doing a short evaluation of their approach to the topics mentioned above. Additionally, we will present performance tests on comparative database instances of the Aurora, RDS, and PolarDB services. These tests has been performed using the *sysbench* database benchmarking tool, which is widely recognized within the industry for measuring database performance[41]. The tests run against the databases consists of basic OLTP tests, as well as some more specific tests for the systems in question. The main goals are to

- Independently verify the performance for Amazon's Aurora and RDS service, as well as Alibaba's PolarDB
- Compare how the database systems perform on similar hardware running similar tests

Further, we try to map these results to the underlying architecture, and see where each system excel and where they have their weaknesses.

1.1 Report structure

The report is divided into the following sections:

Chapter 2 will give a basic intro into what is meant by "the cloud", and how the services provided typically are divided into several layers such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Chapter 3 present some basic background theory to understand the current landscape and how we got here. This includes topics such as the distinction between the ACID/BASE philosophies, the CAP theorem and its implications, emerging technology such as RDMA and NVMe, and the consensus problem with solutions.

Chapters 4, 5, and 6 will present the Amazon RDS service, the Aurora database service, as well as the PolarDB/PolarFS database service, respectively.

Chapter 7 presents the methodology, tools, and setup used to carry out the performance tests for the various systems.

Chapter 8 present the results obtained from the various tests.

Chapter 9 discusses the architecture, results, and implications these carry with them.

Chapter 10 outlines future work, and how the results can be carried further.

Chapter 11 sums up and discusses the most important results.

The presentation of the Aurora and PolarDB database systems, as well as some background theory, is based on the specialization project report [31].

2 What is the cloud, and how does it work?

The cloud is generally referred to as a shared pool of compute resources together with higher level services that can be provisioned fast and with little management [6]. In general, cloud compute can be described as on-demand compute resources, that can be elastically scaled up and down to meet demand [32].

There's typically been a distinction between public and private cloud. This distinction depends on who are managing the cloud infrastructure. In a private cloud, the infrastructure is run by the company who utilize it. This is typical in large companies, which may already have infrastructure and expertise for managing it, or have special security requirements [36]. A public cloud is on the other hand run by a separate company that provides and manages its infrastructure, making it available for third-parties [36]. The major contenders here are Amazon Web Services, Alicloud, Google Cloud Platform, and Microsoft Azure.

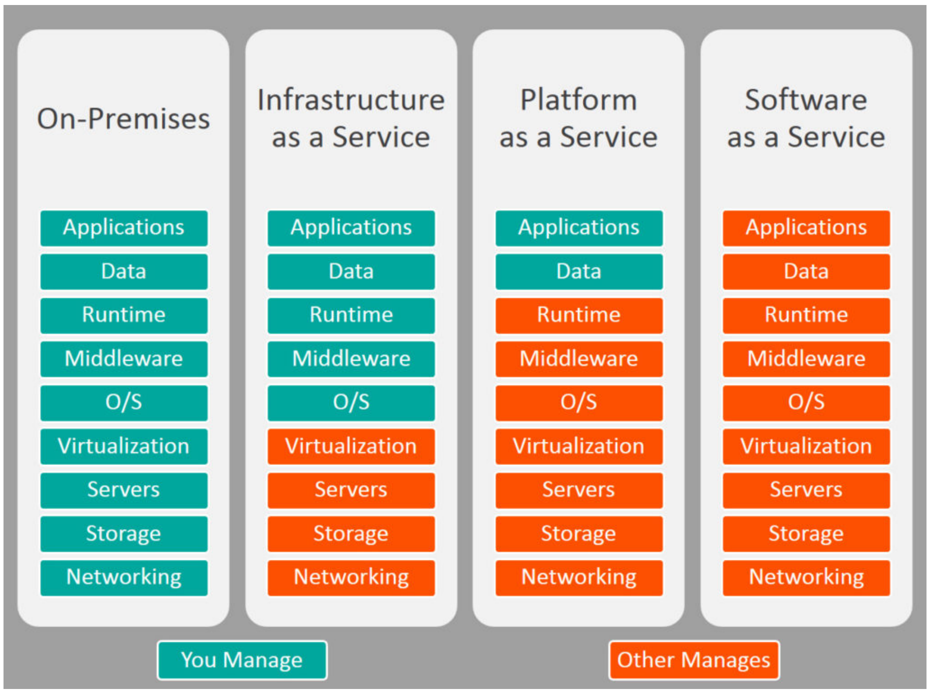


Figure 1: Differences between on-premise, IaaS, PaaS, and SaaS. Source: [24]

Cloud platforms can typically divide the services they provide into a fixed hierarchy. These are Infrastructure-as-a-service (IaaS), Platform-as-a-service (PaaS), and

Software-as-a-service. Each level differentiate what is managed by the cloud provider and what is managed by the customer, as seen in figure 1.

IaaS is typically the most low-level service provided in a cloud environment. This level provide the basic infrastructure such as compute resources such as servers, networking, and storage through virtualization technology [32] [24]. This is the most flexible level in the hierarchy, giving the customer full control of their infrastructure. The infrastructure can easily scale to demand, and the customer is only billed for the resources that they actually use.

PaaS, however, provides a framework that developers can build upon and deploy their applications on top of, without the complexity of managing and provisioning underlying hardware and software [32] [24]. This greatly reduces the burden on the application developer in maintaining and managing the underlying architecture.

SaaS provide complete cloud-based services provided by the cloud platform and consumed directly by the consumer. This is the most abstracted layer in the hierarchy [32].

Many of these cloud services are managed through a separate user interface, typically through a web page. The customer only pays for the services he consumes, and can actively monitor the performance and diagnostics of the service. Cloud platforms such as AWS provide services across the whole spectre, making it possible to integrate the services in various ways, and can be utilized for building more complex and complete applications.

The main points that separate cloud computing from the traditional setting is that the cloud is highly scalable, elastic, easier to manage, and with a cost model where you only pay for the resources you use [6].

Most cloud providers have a database service, which allows for deploying databases as a SaaS. In the case of Amazon, the Relational Database Service (RDS) supports many different database systems (i.e. MySQL, PostgreSQL, and Aurora) [14]. Alibaba has a similar offering with Alibaba Cloud. Alibaba's PolarDB is already available in some data centers in Asia, but they're also in the process of rolling out international support for PolarDB as well.

2.1 Shared nothing vs. shared disk

Shared nothing vs. shared disk has been a widely discussed topic in distributed database architecture and design. The traditional approaches have been shared nothing and shared disk, which each describe separate ways on how the data is stored and accessed. Each of these architectures have various trade-offs when it comes to read and write performance.

In a shared nothing architecture, each individual node stores its own set of data. The data is fully segregated with each node having full autonomy over its own dataset. In contrast, a shared disk architecture allows for any node to access the data stored on any disk [23].

A shared disk architecture might face problems when multiple nodes want to write to the shared disk. This has traditionally been solved either by distributed locks or a disk based lock table [33]. These measures impact performance and introduce scaling problems. For shared nothing architectures, the optimal solution is to direct every write directly to a specific node. However, in cases where the data touches on multiple nodes, a shared nothing system will still need to implement distributed locks [23]. This solution however, has better options for scaling. The better the data is partitioned, the less you have to make use of the distributed locks. Shared disk can also make use of logical partitioning of data to remedy some of its scaling issues.

A shared disk system can easily experience bottlenecks during reads, as all nodes need to access the data from the same point [33]. Caching might also be less efficient in this setting for the same reasons. Shared nothing systems however typically have slow response times for reads with joins where the data spans multiple nodes.

A point made for shared nothing systems is that they have less complexity at scale. Since nodes in a shared nothing system store their own data with full autonomy, it's easier for a node to handle failures independently and on their own with no system-wide knowledge [23]. Additionally, shared nothing systems won't have a single point of contention or failure, may be self-healing and easier to perform non-disruptive updates for. Shared disk systems however, generally have more complex failure handling, with management of locks etc.

3 Background theory

This section will explain some of the concepts that a modern database system depend on, namely the ACID/BASE philosophies, the CAP theorem, emerging technologies such as NVMe disks and RDMA, the NewSQL movements, as well as the distributed consensus problem and its solutions. These are all fundamental background theory for understanding the current landscape and how we got here. This section will serve as a primer for some of the concepts and relevant technologies which is part of the systems we will be discussing later in this report.

3.1 ACID and BASE - two design philosophies

There are two major design philosophies for database systems which describe the guarantees the database system gives. The ACID properties have been the dominant for traditional systems, but in recent years, the BASE approach has significantly gained tracking. This is mainly in conjunction with the rise of NoSQL systems, which trades off some of the traditional guarantees of ACID in order to provide high-availability, scalable, distributed systems.

ACID
ACID is an acronym for Atomicity, Consistency, Isolation and Durability, and describes a set of properties that are desirable for transaction processing in database systems. ACID has held a prominent place in database systems for the past 30 years, and most traditional systems provide the ACID guarantees in slight variations [42].

- Atomicity - a transaction should either be performed fully, or not at all.
- Consistency - a transaction should take the database from one consistent state to another, obeying constraints and other limitations.
- Isolation - a transaction should be seen as executing in isolation from other transactions.
- Durability - Once a transaction has completed, the changes should be made durable, so it's not lost at some later point.

BASE
The BASE guarantees relaxes on some of the properties of ACID. Many NoSQL systems have adopted BASE since they don't need the strict guarantees of ACID, and can therefore make a trade off for other properties, such as higher availability [42].

- Basic availability - the database should be available for operations at nearly all times
- Soft state - the state of the database could change over time, even with no explicit input
- Eventual consistency - the database will eventually converge on a consistent state.

Compared to ACID, the BASE approach severely relaxes the guarantees on data consistency. BASE opens for the possibility to read stale or inconsistent data, and pushes the task of managing consistency up to the application [25].

3.2 CAP theorem

In a distributed system with data replication, concurrency control becomes more complex than in a traditional setting. If an update is performed, the changes need to be performed in a consistent manner across all data items. An example would be that a transaction T_1 updates data item X. If another transaction T_2 updates a copy of the same data item, following transactions might read different values for X.

The CAP theorem describes three sets of desirable properties for a distributed system with data replication. The C is consistency among replicated copies, A is availability of the system to perform operations, and P is partition tolerance in the face of a network partition where the system is unable to communicate with all its nodes. The CAP theorem states that a distributed system with data replication cannot guarantee all three properties at the same time, so that a trade-off has to be made among the properties.

Consistency, in the CAP theorem, refers to consistency across replicated data copies. It's important to point out that this is different from the consistency guarantee of ACID. In the CAP theorem, the "consistency" refers to that the same item will be visible across multiple replicas. Availability refers to the system's ability to serve incoming requests. Either a request will be successfully served, or the system will display an error. Partition tolerance refers to the system being able to withstand partitioning of the nodes in the event of a network failure [25]. The traditional line of thought is that you would need to choose two of these three properties when designing a system. Generally, traditional relational databases have covered the consistency and partition tolerance angles. NoSQL systems are generally more lax on the consistency guarantee, instead offering "eventual consistency" which is, as the name implies, that the data might be inconsistent for some time, but consistent in the end. Instead, NoSQL systems guarantee availability and partition tolerance [42].

However, often the CAP theorem has been used more or less as an excuse for opting for an "eventually consistent" model, even when better consistency guarantees could've been made [25]. Eric Brewer, who originally proposed the CAP theorem, argues that the term "choose 2 of 3" is misleading. He argues that the desirable properties of CAP isn't binary, but rather continuous. A system shouldn't straight up forfeit one of the properties, but rather make a more balanced tradeoff. He also argues that software systems should be more dynamic in how it handles this tradeoff, and adjust the balance between the three properties as necessary. Finally, he argues that partitions are an rare event, and when handled explicitly, consistency and availability can be optimized. He argues that perfect consistency and availability can be achieved most of the time [25]. When a software system believes a partition is present, it should make steps to detect and remedy the situation.

Brewer argues that CAP and latency is closely interconnected. Each program must at some point make a partition decision, on whether to cancel the operation and decrease availability, or proceed with the operation and risk inconsistency [25]. A partition takes form as a very long delay. This may be experienced differently by the nodes in the system. There is therefore no global notion of a partition. Some nodes might detect a partition, while others won't. When a node detects a partition, it should enter a special partition mode, which will give it more flexibility in balancing consistency against availability [27]. Brewer makes a point that it might be beneficial to relax on consistency in order to maintain some degree of availability at this point. When the partition is solved, the nodes would need to perform a partition recovery which must restore consistency and revert mistakes the system might have made during the partition.

In a recent blog post, Abadi supports Brewer's claim that most NoSQL systems unnecessarily sacrifice consistency [27]. The reasoning is the misunderstood notion of "choose 2 of 3" of the CAP theorem, and that this is the most common path for NoSQL systems. Abadi, however, argue that putting the task of ensuring consistency on the application developer is a heavy burden, and it's very difficult to do bug-free in practice. Further, A real-life system never will be able to achieve absolute availability, but will rather be a percentage. The notion that you have to give up availability in order to guarantee consistency is flawed. Guaranteeing consistency would rather imply slightly reduced availability, not a total loss of it [25]. Abadi further supports the claim that network partitions are a rare event, and becoming even more so with the rapid development of network infrastructure. Further, the loss of availability when guaranteeing consistency wouldn't be dramatic. In the face of a partition, the system would still be available from the majority partition, but unavailable from the minority partition. For the reduced availability to be noticed, clients must be able to reach the minority partition but not the majority.

3.3 Remote direct memory access (RDMA)

Remote direct memory access is an emerging technology which allows computers to transfer data directly from ones memory to anothers. No work needs to be done by the CPU, caches or context switches, allowing for zero-copy of data. Instead, its the network adapters are the main component for performing RDMA transfers [7]. This allows for high throughput and very low latency networking between computers. For a large distributed system, where the nodes are closely located, this would be a huge performance gain over conventional TCP/IP networking.

There's been done much work in standardizing the RDMA technology. Currently, its still being worked on by the RDMA Consortium together with the DAT Collaborative. There's also been much hardware development for network interfaces supporting RDMA. RDMA is currently used to a degree in some systems. Red Hat Enterprise Linux, VMWare, and Microsoft's Windows Server all have support for RDMA. RDMA is also extensively used in systems such as PolarFS.

Typically, RDMA provides both one-sided and two-sided communication. Two-sided communication is i.e. a send/receive operation that will need the remote process to respond back with a receive operation to the sending process. One-sided communication is i.e. read/write operations where the NIC manipulates the memory directly without involvement from the remote process [7].

In practice, a combination of one-sided and two-sided operations will be performed. For smaller transfers, a two-sided operation can be used directly. However, for larger transfers, a two-sided operation will be used for negotiating the destination address on the remote system. A one-sided operation will then perform the actual transfer using the negotiated values.

RDMA has increasingly seen adoption in data centers across the globe [7], and multiple new systems can therefore take advantage of this technology. One of these systems, are PolarDB and its PolarFS storage layer, which extensively use this new technology. PolarDB is currently being rolled out to an international audience, but as of writing, only select data centers in Asia has full PolarDB support.

3.4 NVMe disks

Non-volatile Memory Express (NVMe) is a new device interface specification aimed at taking advantage of the multiple levels of parallelism in modern SSD storage technology [7]. Previous specifications were in general designed mainly for harddisk drives, and are thus unsuited for solidstate drives. The NVMe specification aims at reducing these bottlenecks, and has a much smaller I/O overhead, with higher bandwidth and lower latency interconnects.

A study shows that a simple 4KB I/O request can require up to 20 000 instructions. As SSDs become faster, they will exacerbate the limitations of the traditional I/O stack. An NVMe SSD will be able to perform about 500 000 IOPS at a 100μs latency [7]. There's also been introduced a new 3D XPoint SSD that further reduces this latency to around 10 μs.

To make leveraging NVMe storage easier from the application layer, Intel has introduced a Storage Performance Development Kit (SPDK). It achieves high performance by moving many drivers into user mode, and using polling instead of interrupts. This removes the need for expensive kernel context switches and interrupt overhead.

Similar to RDMA, this is also an emerging technology extensively used by PolarDB/PolarFS. Many other large companies have also started experimenting with this technology, and it's likely it will become even more popular in the coming years.

3.5 NoSQL and NewSQL systems

The term NoSQL in its current form was coined around 2009, and describes databases systems that use other models and data storage mechanisms than those found in traditional relational SQL databases.

NewSQL is a term used for modern relational database systems which tries to solve problems regarding to scaling OLTP workload in an efficient manner [40]. The NewSQL movement came as an response to NoSQL, where NewSQL aims at maintaining the ACID guarantees of the traditional database systems. Some of the traits typically found in NewSQL systems are (1) being able to scale out to multiple nodes, (2) keep SQL with ACID guarantees, even in a multiple shard environment, (3) replication doesn't interfere with the properties listed above [40]. Multiple new systems have come to the market, and existing systems have tried to adapt to the changing infrastructure and workload that the cloud architecture demands. One of the main systems born out of this movement is Amazon Aurora, which also is a managed service, removing much of the traditional

However, researches point out not all NewSQL systems that claim to provide consistency, actually do. This leads back to the same problem where the application developer would need to implement consistency checks in the application code, which was exactly what CP systems aimed at avoiding [27]. Most modern systems use a consensus protocol for assuring consistency in a distributed system. There are several such protocols available, i.e. Raft and Paxos. These systems work by a majority vote mechanism, where each data replica is assigned a vote, and an operation on the data needs the majority of replicas to agree on the operation [3]. As a consequence, the minority in a partition can be offline, while the majority keeps processing operations. However, exactly how these protocols are used in practice is up to the specific systems implementation. Some systems use a single global consensus protocol for the entire database, while others divide the data into shards and use a separate consensus protocol per shard [27].

The global consensus protocol approach easily runs into scaling problems. If the whole fleet has to vote over every transaction, this would incur heavy network load and be a severe bottleneck for the system. In practice, most systems instead batch transactions, and instead of voting over single transactions, votes over a batch of transactions. This performs relatively well, and many systems with a global consensus protocol have opted for this approach [27].

The per-shard consensus protocol approach has obvious consistency problems when a transaction touches data across multiple shards. One of the first systems to use this approach was Google Spanner, which resolved the problem by it's TrueTime API. The TrueTime API is used by Spanner to determine the timewise relationship between two transactions, and this works even when a transaction involves data across multiple shards. To eliminate problems regarding clock skew across multiple servers, Spanner introduces an "uncertainty window", which represents the maximum clock skew that can occur across the servers. Only after this window has passed, is the data visible for the client [5]. This is backed up by sophisticated hardware involving GPS and atomic clocks, in order to hold the guarantee that the clock skew never exceeds the "uncertainty window" [5].

However, many systems such as YugaByte and CockroachDB have tried to build upon some of the architectural foundations laid down by Spanner. They rely on the same principle that clock skew across servers doesn't grow too big. Spanner manages this by having its hardware component that handles this particular problem [5]. However, the other systems building on Spanner architecture are fully software based solutions, and lacks the hardware feature that ensures the "uncertainty window" condition is kept [27]. It is therefore possible there might occur inconsistencies, and the systems can't guarantee CAP consistency.

3.6 Distributed consensus

One of the main problems of distributed computing is the consensus problem. The problem boils down to having a set of nodes in a distributed system to agree on some common data value [3]. In a database setting, this would typically be for a set of nodes to agree on whether to commit a transaction or not. Especially for a cloud database system, this question is of essential importance. There's been various protocols suggested for dealing with the consensus problem. The goal is to achieve system reliability in the event of process failures.

More formally, the consensus problem can be described by a system consisting of processes $P_i i = (1, 2, ..., N)$. Each process starts out in an undecided state and propose a value v_i taken from a set of D. The processes then communicate with eachother and exchanges values. Each process then sets a value it has decided upon [3]. The following traits describe a simple consensus protocol that can tolerate halting failures:

- Termination - Every correct process eventually decides upon a value
- Integrity - If all correct processes propose the same value v, then any correct process must decide upon v
- Agreement - Every correct process must agree on the same value

There's typically a distinction between crashes and Byzantine failures. A crash is simply that a process suddenly stops and doesn't resume. A Byzantine failure, however, is the process may fail in any way possible [3]. This might be sending invalid or contradicting messages, or be an act of a malicious third-party. These types of failures may be harder to defend against. In order to tolerate byzantine failures, the Integrity property may be strengthened such that

- If a correct process decides upon v, then v must have been proposed by a correct process.

3.7 Paxos

Paxos includes a set of protocols aimed at solving the consensus problem in a distributed system [46]. Paxos was first introduced in 1989, and encompasses several different protocols with corresponding trade-offs such as the number of participants, message delays and various fault types. Paxos aims at being fault tolerant and safe, with few possibilities to have a stuck state where the system can't make progress [37]. Paxos has been widely adopted by many systems, and have become close to a de-facto protocol used for consensus in fault-tolerant distributed systems. Paxos has, however, laid the groundwork for further developments and alternative protocols. Because of its dominant position, other systems are often compared against Paxos. In order to understand the systems derived from Paxos, this section will briefly outline the basic Paxos protocol.

In Paxos, a system consists of processors. A processor can have one or several roles, namely client, acceptor, proposer, learner, and leader [46].

- A client proposes requests to the system, such as write operations
- Acceptors vote on requests. If a client request fails to reach a quorum majority, it is disregarded.
- Proposers advocates a client request for the Acceptors. It acts as a coordinator in case conflicts should arise in order to move the protocol forward.
- Learners act on enforcing the decision by the Acceptors, such as executing a request and respond to the client.
- A leader is a distinguished Proposer which is tasked to ultimately make the protocol progress forwards.

Paxos also ensures the consensus safety property. This is done by adhering to the following points at all times [46]:

- Validity - only proposed values can be chosen
- Agreement - no two learners can learn different values
- If a value has been proposed, eventually learners will learn some value

The simplest variant of Paxos is known as "Basic Paxos" and is used for a system to agree on a single proposed value. The execution may span over multiple rounds, where each round typically consists of two phases. The first phase ensures acceptors promise to receive a request, and the second will accept a value for the given request from the previous phase [37].

Phase 1 starts with a Proposer receiving a request from the client. The Proposer then sends a Prepare message with a unique identifying value n to a quorum of Acceptors. The number n is monotonically incremented for each Proposer, so that the Proposer hasn't used that number before. When an Acceptor receives a Prepare message, it will look at the number n. If the number n is larger than any of the previous received Prepare messages, the Acceptor will return a Promise message [46]. This means that the Acceptor will not allow any other Prepare messages having a number less than n. If an Acceptor has previously accepted a proposal, it will return the identifying number and the accepted value to the current Proposer. Otherwise, if n is less than any previous received Prepare message, the Acceptor will ignore it [37].

Phase 2 starts by the Proposer obtaining quorum, and then choosing a value to propose. In case any of the Acceptors have previously accepted a Prepare message, the Proposer will choose the returned value with the highest identifying number for its value. Otherwise, the Proposer will choose the value it had originally received from the client [37]. The Proposer then sends an Accept message to a quorum of the Acceptors with the identifying number n and chosen value. The Acceptor will then accept the Accept message if it hasn't agreed to any Prepare messages with a higher n. The Acceptor will then register the value and reply back to the Proposer as well as the Learners with an Accepted message [46].

The "Basic Paxos" protocol can be extended and optimized to work for agreeing on a series of decisions. This is useful for instance for agreeing on a distributed log used in conjunction with a state machine [4]. The distributed log would in this case increment the state machine in a unison manner across the system.

The idea of Multi-Paxos is to run "Basic Paxos" multiple times. A typical optimization is to have a stable, sticky leader. In this case, the Prepare phase only has to be done once in the start. For subsequent protocol instances can skip the Prepare phase, since there only is a single leader to increment the unique identifying number n [4].

It has been argued that even single-decree Paxos is very difficult to understand, and can only be done with large effort and difficulty. This is due to a very opaque and dense description, with many subtleties. This is further exacerbated in multi-Paxos, which is lacking with many missing details [29]. Additionally, Paxos doesn't provide a good ground for building systems. Lamport's descriptions of Paxos is mostly aimed at single-decree Paxos, which isn't too practical in real-world systems. Typically, Paxos is used as a starting point, and then each system independently try to make Paxos "sane" with their own optimizations and thus drifting significantly from the original base [28].

3.8 Raft

Raft is an alternative consensus algorithm, aiming at reducing some of the complexity of Paxos. Raft is an acronym for "Reliable, Replicated, Redundant, And Fault-Tolerant", and provides similar assurances as Paxos for fault tolerance and performance [26]. Raft offers a generic way at distributing a state machine across a set of nodes, so that all nodes agree on the chain of state transitions. Some of the motivation behind Raft lies within the frustration of working with Paxos; both system builders and students found it difficult to reason about and build systems on top of [28]. Raft aims at reducing the complexity compared to Paxos. Raft is cleanly divided into subcomponents for leader election, log replication and safety. It also has a reduced state space compared to Paxos, which reduces the degree of non-determinism and ways nodes can be inconsistent [29]. PolarFS uses a derivative of the Raft protocol, and this section will therefore describe the Raft protocol in order to understand what improvements Alibaba has contributed to it.

Due to the problems with Paxos, Raft set out to make a more understandable and useful consensus algorithm with the same guarantees as Paxos. The Raft team has put understandability as one of its primary goals, being easy to learn and reason about. The users should be able to intuitively understand it, so it will be easy to implement in real-world systems [29]. This point has influenced the design process of the Raft algorithm. When facing multiple different approaches to a problem, the Raft team usually tried to opt for the solution which was most intuitive and easy to explain. This involved decomposing large problems into smaller components, such as leader election, membership changes and log replication [29]. The team also tried to reduce the state space, making the system more coherent and with reduced non-determinism. For instance, logs are not allowed to have holes, and Raft limits the ways logs can become inconsistent [28].

Some of the main features in the Raft consensus algorithm is their approach to leadership, leader election, and membership changes [26]. Raft has opted for a strong leader approach, where log entries only flow from the leader to other followers. This simplifies log management and thus makes it easier to reason about. Raft also use random timers for leader election, which introduces little overhead, while easily resolving conflict problems. Raft utilizes a new joint consensus approach for handling membership changes, which allows a cluster to keep operating normally during configuration changes [29].

Raft aims at providing the following features for its consensus algorithm [28]:

- Safety - Never return an incorrect result, even under non-Byzantine failures such as partitions, packet delays/losses and reordering.
- Availability - It remains functional as long as a majority of servers work as expected.
- Does not rely on timing for consistency. Network delays and faulty clocks shouldn't cause availability problems.
- A command can complete as soon as a majority of the servers agree upon it, and not be bound by the slowest responding server.

Like many consensus systems, Raft uses replicated state machines for its consensus algorithm. Each server typically has its own state machine that can be used to compute the same state across the system, and continue operating in case a few servers should fail. These state machines typically use a replicated log, which the state machines execute in order. The state machines are deterministic and will therefore end up on the same result across the system [26]. The consensus algorithm ensures the replicated log is consistent by communicating with the other servers to ensure that the log entries are in the same order. After the consensus module has ensured the log entries are consistent, the state machines execute the entries, giving the same state across the system [28].

A Raft cluster typically spans multiple servers, where each server may be in one of the following states: leader, follower or candidate [29]. Normally in the Raft cluster, there's always one distinguished leader, with the other servers as followers. The leaders handles all outside client requests. The followers are passive, in that they issue no requests of their own and only respond to the requests of the leader. The candidate state is typically only used during the leader election process [28]. Raft uses a distinguished leader with complete responsibility for managing the replicated log. It (1) accepts log entries from clients (2) replicates the entries to the followers and (3) notifies the followers when its safe to apply the log entries to their state machines [29]. Raft divides time into terms of arbitrary length, where each term is designated by an incrementing number. Each term starts with an election, where a new leader is chosen. The elected leader then leads until the end of the term. Each server stores the current term number, and is included in all server communication. If a leader or candidate notices its term is less than the others, it will automatically revert to the follower state [28]. Similarly, followers will reject requests with stale term numbers.Servers communicate via RPCs, which are executed in parallel and retried upon failures.

Raft utilizes a heartbeat mechanism for leader election. Servers remain as followers as long as they receive valid heartbeats from a leader. If a follower don't receive a heartbeat for some time, a timeout will occur, and the follower will then commence a new election [26]. The server that experienced the timeout will transition to the candidate stage. It will then increment its term counter, and send out a RequestVote RPC to the other servers in parallel. The three outcomes of this are the following [29]:

- A candidate wins the election if the majority of the servers in the system vote in favour for it. A server will vote for maximum one leader per term on a first-come-first-serve basis. Once a candidate has received enough votes, it will assume the leader role, and send out a heartbeat to the servers, informing them that it won and is now the acting leader.

- The candidate may receive RPCs from another server claiming to be a leader. In this case, the term number is compared. If the term number of the claimed leader is equal or higher to the candidate, the candidate will revert to the follower state and accept the leader. If the number is less than to the candidate, the candidate will reject the claimed leader and continue the vote.
- There may also be a tie between different candidates. In this case, none of the candidates will get a majority and become leader. The candidates will time out and start a new election by incrementing the term and issuing new RequestVote RPCs

In order to avoid split votes could repeat indefinitely, Raft employs a random timeout mechanism. Each candidate chooses a random timeout from a fixed interval [28]. This drastically reduces the chances for a split vote occurring/repeating itself.

The election process also includes a mechanism for securing that the leader elected has all the latest committed log entries present. This is done during the voting process, where a candidate would need to obtain a majority vote [29]. This implies that at least one server that hold all committed entries. In the case where the candidate is out-of-date compared to another follower, that follower will deny its vote for the candidate [26].

Once elected, the leader will serve client requests. The leader appends commands to its log, and then replicates the log in parallel using the AppendEntry RPC [26]. In case of follower failure or delays, the leader will retry the RPC until all followers store the correct log sequence. The leader will apply the log entries to the state machine and return the result to the client. The log entries contain the state machine command, current term number, the index of the leader's highest known committed entry, and an integer index identifying the previous log entry. A log record is seen as committed once a majority of the followers have replicated the entry [29]. A consequence of this is that the previous log entries of the leader also are committed. The leader also keeps track of the highest index to have committed, and this is relayed to the followers through the AppendEntry RPC. Once the followers become aware of this, the log entries are applied to the state machine in order.

Raft can then ensure [29]:

- That two entries with the same term and index contain the same state machine command, because a leader creates at most one entry with the index for a given term.
- That two entries with the same term and index, then the logs are identical in all

preceding entries, because the follower checks when receiving an AppendEntry RPC that the included previous entry index exists in its log. If it doesn't, the follower will reject new entries.

However, leader crashes can leave the logs in an inconsistent state, i.e. if the leader fails while replicating a log entry to the followers. To deal with this, a leader will force a follower to duplicate the log of the leader [29]. This includes overwriting conflicting entries. The leader will find the followers latest log entry that agrees with the leaders, delete any following log entries from the follower, and then replicate the missing entries from the leader to the follower [26]. One thing to notice is that leaders won't commit log entries from previous terms by majority counting. Only entries from the leader's current term are committed this way, and thus ensures that all prior entries are committed as well [29].

Raft manages changes to the cluster using a joint consensus approach. A leader will propagate the latest cluster configuration to the followers using special entries in the replicated log. Each server in the cluster uses the latest cluster configuration sent by the leader, disregarding whether the entry has yet been committed [29]. Raft uses this mechanism such that a cluster will first go over to a transitional configuration, before moving on the the new configuration [28]. This aims at avoiding issues where the cluster might split into two independent majorities during configuration changes and allow for continued serving requests during a cluster change. In the transitional configuration, both new and old configurations are combined, such that (1) all servers receives log entries, (2) any follower can become leader, and finally (3) any agreements such as leader election need a separate majority from both the new and old configuration [29]. Once the joint consensus has been reached, the cluster leader can then move on and commit the new configuration [28]. The transition is then complete, and the servers possessing the old configuration can then be shut down or upgraded.

3.9 Quorum models

Quorum models can also be used to enforce consistent operations in a distributed system. Similar to consensus protocols, each node is assigned a vote, but in the case for quorum models, only a number of nodes have to agree to perform the operation. This allows quorum models to work in the face of network partitions. There are two main use cases for quorum models, namely quorumbased voting in commit protocols and quorumbased voting for replica control. In each case, the quorum exhibits slightly different rules in order to perform as desired. Quorumbased voting for replica control is heavily utilized by Amazon Aurora, and makes possible amny of the optimizations Amazon has done.

3.9.1 Quorumbased voting in commit protocols

In this case, a distributed transaction can be executed on multiple nodes. The atomicity property needs the transaction to either be performed in its entirety on all nodes it touches upon, or on none. The quorum model describes that a transaction can be performed if the majority of nodes vote for it [45]. This allows a system to remain operational in the face of network partitions, where the system is unable to communicate with all the nodes in the system. This can be solved by using a quorum approach where each node in the system is assigned a vote. In this example, we'll have a total of V nodes. We'll also have a commit quorum, V_c, and an abort quorum, V_a. The quorums must abide to the following rules [45]:

- $V_a + V_c > V$ and $V_c > 0, V >= V_a$
- Before a transaction commits, it must obtain a commit quorum. Nodes prepared to commit or waiting must be more than or equal to V_c
- Before a transaction aborts, it must obtain an abort quorum. Nodes prepared to abort or waiting must be more than or equal to V_a

The first rule indicates that a transaction cannot be committed and aborted at the same time. The two second rules ensure a quorum for either action is reached before proceeding [45].

3.9.2 Quorumbased voting for replica control

In this case, a database system can store copies of a data item across multiple sites. Due to serializability, two transactions shouldn't be allowed to either read or write the same data item at the same time. This can be solved by using a quorum approach where each copy of a data item is assigned a vote. In this example, we'll have a total of V copies. In order to perform a read on the data item, the transaction would need to obtain a read quorum V_r. Similarly, in order to perform a write on the data item, the transaction would need a write quorum, V_w. The quorums must abide to the following rules [2]:

- $V_r + V_w > V$
- $V_w > V/2$

The first rule ensures that no copy is read and written concurrently, as well as that the read quorum contains a node with the latest version of a data item. The second rule ensures that no data item is written concurrently by two transactions [1].

4 Amazon RDS

The Amazon Web Service (AWS) include many different services covering most aspects of modern cloud architecture. One main point is that different services can communicate with each other, and be combined to create complex applications. One of the services offered is the RDS (Relational database service). The RDS can be used with various popular databases. At the time of writing, RDS supports the following relational databases: MySQL, MariaDB, PostgreSQL, SQL Server, Oracle, and Amazon Aurora[22].

The AWS platform is divided into geographically separated regions. Within each region, there's multiple data centers (in AWS terms, Availability zones, or AZs). AZs are relatively isolated from each other in most aspects, such as power, networking, software deployment etc. The availability zones are interconnected with low latency links[1][22]. With this layout, Amazon achieves great fault tolerance and stability[20].

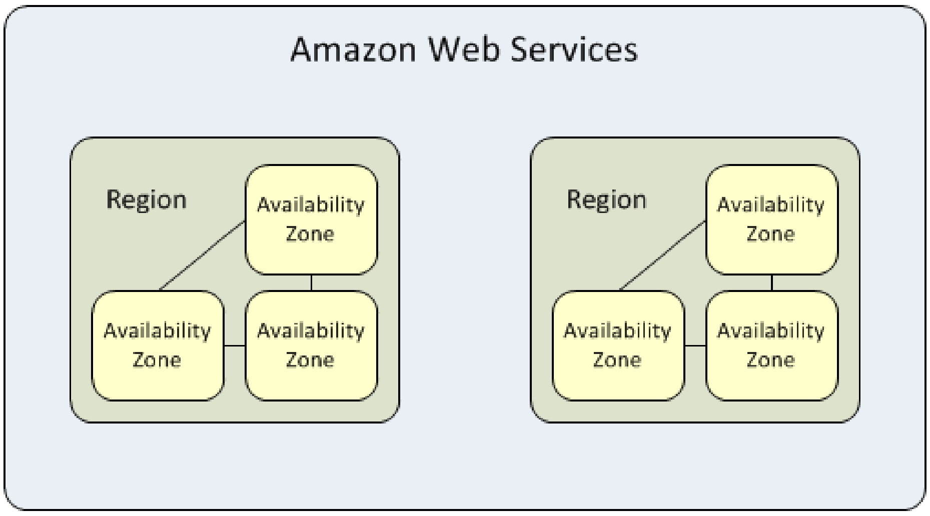


Figure 2: AWS regions and availability zones Source: [21]

Like most AWS offerings, RDS is a fully managed service, which implies Amazon takes care of needed hardware provisioning, software patching and most setup tasks. RDS aims to provide a highly scalable database service, with ability to adjust compute and storage resources to demand[20]. Additionally, RDS instances aim at being highly available and durable, with the ability to i.e. synchronously replicate data to a standby instance at a different datacenter. Additionally, RDS is easy to administer either through Amazon's web dashboard, or via APIs. RDS can also be interconnected with other AWS services[22]. This could either be to insert new data into the database, or generate statistics and logs from the database performance.

4.1 Storage solutions for RDS

For the most of the RDS-supported database systems (MySQL, MariaDB, PostgreSQL, and SQL Server), RDS instances run on an Amazon EC2 instance backed by Amazon's Elastic Block Storage service for database and log storage. There are three separate storage types available for RDS database instances[16], where each has specific traits depending on the cost/performance needs of the database. These are

- General purpose SSD's
- Provisional IOPS SSD's
- Magnetic storage

General purpose SSD's is meant to be a good all-round choice for various workloads while remaining cost effective. It can perform up to 3000 IOPS for short periods of time. Provisional IOPS aims at providing low I/O latency and consistent I/O throughput. Magnetic storage is provided for legacy reasons, and Amazon recommends other storage solutions for new deployments[16].

4.1.1 General purpose SSD's

As previously mentioned, general purpose SSD's aims at providing a good all-round storage solution for most workloads. With the exception of SQL Server, general purpose SSD's can support database volumes between 20GB-32TB. For SQL Server, database volumes between 20GB-16TB are supported[44].

The IOPS performance the database is able to accomplish is governed by a rather complex system. The volume size determines the baseline IOPS the system can perform, but for volumes less than 1TB, the database can perform bursts up to 3000 IOPS for a short period of time[44]. The length of this duration is determined by a separate IO credits balance. Every volume starts out with a credits balance of 5.4 million I/O credits. Once the database instance exceeds the baseline IOPS, the database instance can use of the IO credit to gain additional IOPS[16]. The size of the volume also determines how fast new IO credits are accumulated; larger volumes accumulate faster than smaller ones. When the database instance uses fewer IOPS than the baseline, the "remaining" IOPS are added to your IO credits. The IO credits is capped to the starting amount (5.4 million I/O credits). If the IO credits pool is emptied, the database instance will be capped to the baseline IOPS level. Amazon informs most workloads won't empty out their IO credits pool [17].

4.1.2 Provisional IOPS SSD's

Provisioned IOPS, on the other hand, takes a more straight forward approach. In this case, you provision the IOPS you need upfront and Amazon will provide that number until its manually changed. Amazon recommends this kind of provisioning for database instances that require fast and consistent IO performance[44]. With the exception of SQL Server, you can specify volume sizes between 100GB-32TB and IOPS rates between 1000 - 40 000 IOPS. Amazon additionally allows for use of provisioned IOPS in a Multi-AZ setup, but also for read replicas. The storage option for the read replica is separate from the master[16]. It's worth noting that there might be other factors (such as network bandwidth, CPU, memory) that might pose as a bottleneck, meaning the database won't be able to fully utilize the provisioned IOPS. However, increasing IOPS might also decrease latency, as an IO request doesn't spend much time waiting in a queue.

4.1.3 Magnetic storage

Amazon also supports magnetic storage for legacy reasons, but they strongly suggest to use either general purpose SSD's or provisioned IOPS SSD's. Some of the drawbacks with magnetic storage is that it is constrained to 1000 IOPS and a volume size of 4TB. It also doesn't support elastic volumes[16].

4.2 Read replicas and hot-standby replicas

RDS also has functionality to add replicas to a database setup. Amazon support both hot-standby replicas, as well as traditional read replicas. In Amazon terminology, a database deployment with a hot-standby replica is called a "Multi AZ" deployment[18]. Upon creation of a Multi AZ deployment, Amazon will set up your primary database as well as a standby instance in a separate availability zone. The primary database will synchronously replicate data to the standby[34]. Since the primary and standby are in separate AZ's, they will be isolated for most faults. In case the primary experiences a failure, RDS will automatically perform a failover to the standby replica[19]. The database endpoint will remain the same, so an application will be able to resume operation once the failover is complete. Another difference is that only the database engine on the primary is active, unlike with read replicas that can accept read operations. Multi AZ deployments also always span across two AZ's in the same region, while read replicas allow for placements in the same AZ, across AZ's and cross-region[34]. It is worth noting that Multi AZ and read replicas can be combined, so that Multi AZ could be used to achieve high availability, while read replicas can help to offload read operations from the master[19].

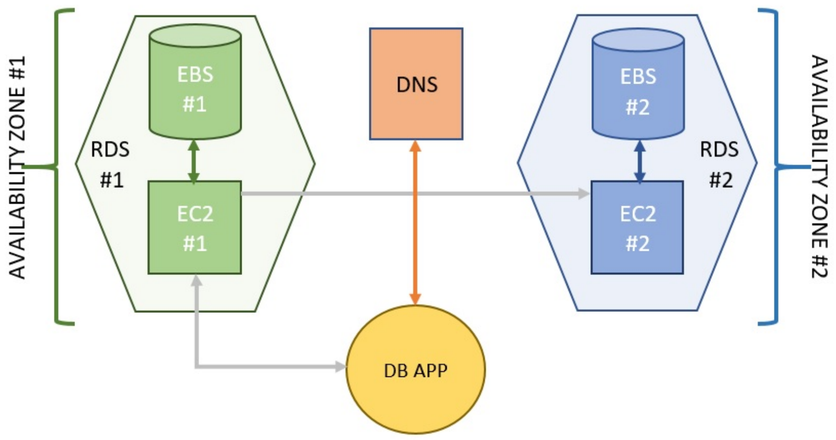


Figure 3: AWS RDS Multi AZ Architecture Source: [35]

With the exception of SQL Server, Multi-AZ deployments also leverage synchronous physical replication to replicate data to the standby replica. This is unlike traditional read replicas, which use asynchronous replication[18]. Multi AZ is implemented using a custom replication layer which sits between the database application EBS volumes. The layer handles reads and writes for the database instance and applies it to two EBS volumes, where one is local, while the other is remote. Each EBS volume is managed by a dedicated EC2 instance[34]. The EC2 instances are connected to each other with a TCP connection. See figure 3. The standby doesn't have a database server process running, but will instead just synchronously write the data it receives from the primary. A database write operation will be applied to both volumes before a successful response is sent back, but read operations will be handled by the primary directly[34]. The replication layer is additionally unaware of any higher level issues, such as connectivity issues etc. The EC2 instances are therefore managed by an external observer which makes sure availability and performance requirements are met. Amazon argue that the incurred performance cost of running a Multi AZ configuration is minimal. Their tests show approx. 2-5 ms increase in database commit latency, but for general workloads, the practical difference will be minimal[34]. If the primary and the standby lose connection with each other, the instances will momentarily pause, and the observer will direct an available instance to resume the role as primary and proceed without replication for the time being. Once connection is restored, the primary and standby will resynchronize. This is done by the primary keeping track of which blocks are modified while the connection is down. During resynchronization, the primary only sends the modifications needed to the standby[34].

4.2.1 Single AZ deployments

Amazon in general recommends using a Multi AZ deployment, due to its higher resiliency to failures[18]. It is, however, possible to deploy single AZ configurations for scenarios where you either don't need the added HA requirements or the associated costs. In both cases, the database will be more vulnerable to failures. Amazon uses the metrics "recovery time objective" (RTO) and "recovery point objective" (RPO) to measure the impact of running a single AZ configuration vs a Multi AZ configuration. RTO corresponds to the amount of time for a recovery to complete in the event of failure, while RPO refers to the amount of time during which data is at risk for loss in the event of a failure[47].

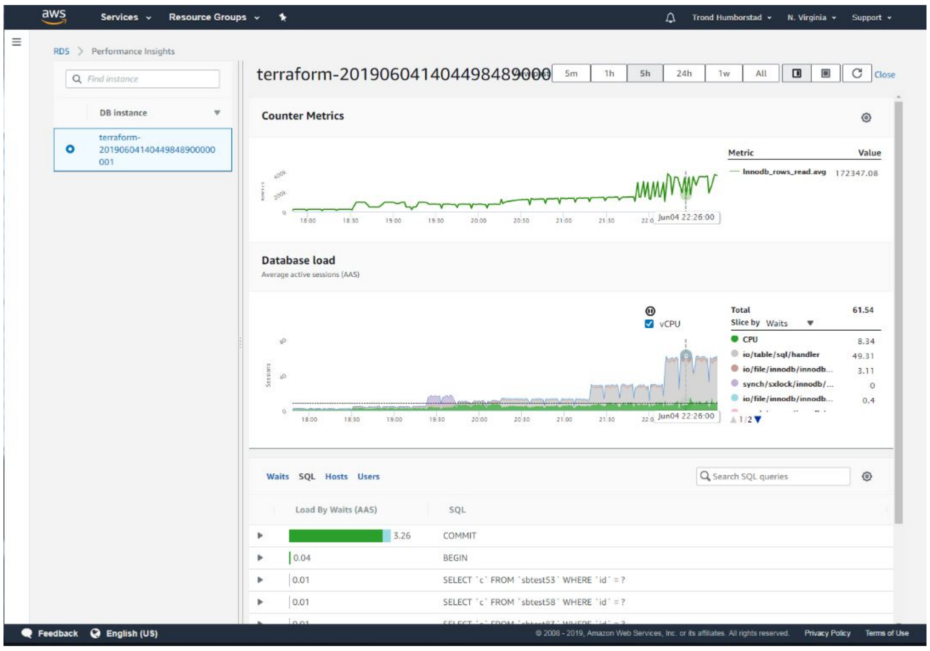
Amazon divides the possible failure modes into four categories. These are

- Recoverable instance failures, where the underlying EC2 instance is experiencing a failure and RDS is able to automatically recover from it
- Non-recoverable instance failures, where the underlying EC2 instance is experiencing a failure, but RDS couldn't recover from it automatically
- EBS failures, where the EBS volume has experienced a data loss failure
- AZ disruption which affects the RDS instance

For recoverable instance failures, RDS will launch a new EC2 instance and attach the underlying EBS volume and recover. This gives an RTO of approx. 30 minutes, but no impact on the RPO[47]. In the case the recovery fails, or an EBS failure happens, a point-in-time recovery will be necessary. The RTO in this case will amount to the time it takes to launch a new instance and apply all the changes since the last backup. The RPO can vary from minutes to hours, depending on the size of the database, the number of changes made since the last backup, and the workload on the database[47]. It is also worth mentioning that I/O will be impacted during snapshot and backup creation for a single AZ configuration. For most of the RDS offerings, all other I/O is temporarily suspended during this period. Similarly, when performing OS patching, a single AZ configuration would be unavailable while its maintenance window[18]. These are major limitations that can be avoided in Multi AZ configuration.

4.3 Logging and performance metrics in RDS

The RDS console has some basic statistics readily available. These include CPU utilization, number of DB connections, amount of free storage space, amount of freeable memory, read IOPS, and write IOPS. However, additional metrics can be obtained both for the OS and DB performance. OS metrics, such as CPU utilization, disk IO, memory, and network performance can be monitored in detail when the "Enable monitoring" option is chosen for the database.



Metric	Value
Innodb_rows_read.avg	172347.08

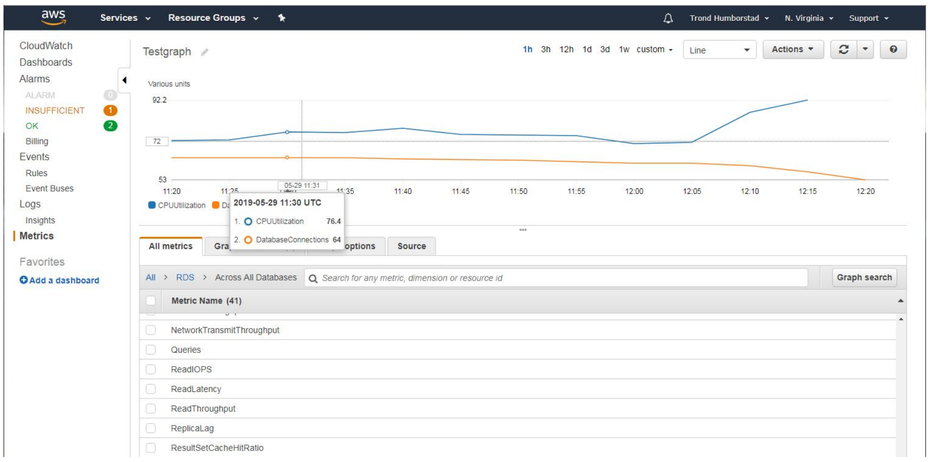
Total	61.54
Slice by Waits	
● CPU	8.34
● io/table/sql/handler	49.31
● io/file/innodb/innodb...	3.11
● synch/sxlock/innodb/...	0
● io/file/innodb/innodb...	0.4

1/2

Load By Waits (AAS)	SQL
3.26	COMMIT
0.04	BEGIN
0.01	SELECT `c` FROM `sbtest53` WHERE `id` = ?
0.01	SELECT `c` FROM `sbtest58` WHERE `id` = ?

Figure 4: Example dashboard in AWS Performance Insights

DB performance metrics can be enabled by turning on "Performance insights" for the database. This will in turn enable performance_schema for MySQL databases, unless explicitly turned off. The performance_schema information will then be extensively used in order to provide insights into the database performance. Additionally, database logs such as the slow query log, audit log, general log, and error log can be extracted from the database instance and used for other performance statistics.



2019-05-29 11:30 UTC	
1. CPUUtilization	76.4
2. DatabaseConnections	64

All > RDS > Across All Databases

Search for any metric, dimension or resource id

Metric Name (41)

- NetworkTransmitThroughput
- Queries
- ReadIOPS
- ReadLatency
- ReadThroughput
- ReplicaLag
- ResultSetCacheHitRatio

Figure 5: Example dashboard in AWS CloudWatch

AWS has a separate service for logging these metrics, AWS CloudWatch, that can be used for monitoring the performance over time. This service can be combined with the RDS service. CloudWatch provides both a dashboard with graphs and plots, as well as an API to access the data directly. An example dashboard can be seen in figure 5.

5 Amazon Aurora

Aurora is one of Amazon's multiple database offerings for the AWS cloud platform. Launched in late 2014, and being generally available in 2015, Aurora has changed the game on how database systems work in the cloud. Aurora is a shared-disk, relational database system, which claims compatibility with MySQL. It is specially tailored for a cloud environment, with high performance and scalability as major priorities. Aurora leverages a distributed, fault-tolerant, self-healing storage system that auto-scales up to 64 TB of data [13].

The Aurora database system is proprietary, and Amazon limits access to the underlying hardware of the Aurora database. Despite this lack of insight into the actual implementation, Amazon has released several papers on the inner workings of the Aurora architecture. Aurora was initially based of MySQL, but a PostgreSQL variant is now also available. For this paper, we will focus on the MySQL variant. The MySQL version of Aurora has been adapted and optimized to work in Amazon's cloud environment. The main changes lie in how InnoDB perform reads and writes [1]. The Aurora database offers MySQL compatibility up to MySQL 5.7, but with some missing features, which we will see in section 7.2.

One of the main changes in Aurora is that it has decoupled the compute and storage components of the database system, and replicates storage across multiple nodes [1]. To ensure consistency, Aurora uses a quorum model [2].

As described in section 4, each AZ in a region is isolated for most faults, but are interconnected with low latency links. For Aurora, distributing data replicas across AZ's should isolate failures to affect only one data replica [2].

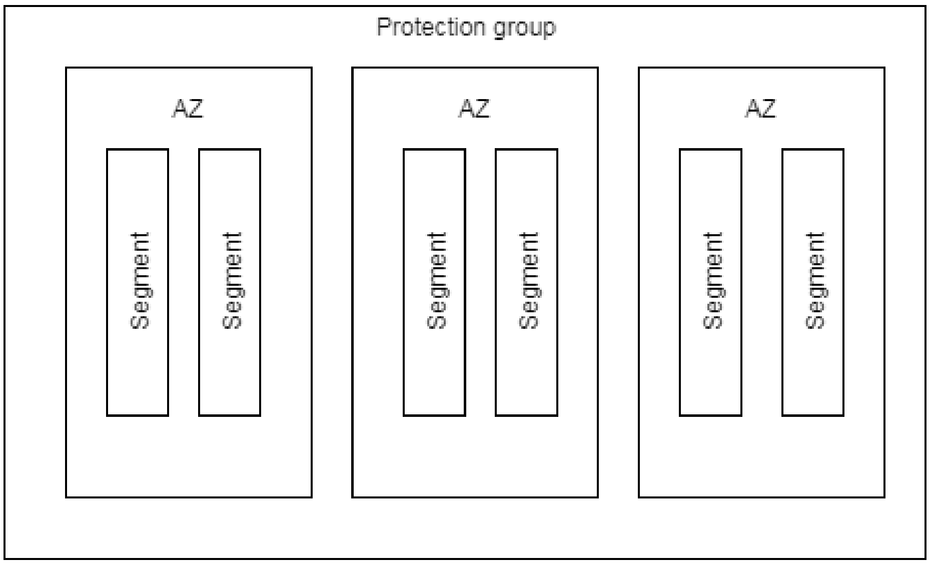


Figure 6: Segments and protection groups in Aurora

Aurora is designed to withstand the loss of an entire AZ and an additional single storage node without losing data. Aurora should also retain write ability when losing an entire AZ [2]. They replicate the data item across three AZ's, where each AZ stores two copies of each data item. This translates into a quorum model with $V = 6$, $V_r = 3$, and $V_w = 4$. This will ensure the design principles can be upheld [1]. Furthermore, Aurora divides the database volume into small segments of 10GB in size. These segments are each replicated six ways into Protection Groups (PG), see figure 6. The Protection Groups are organized across three AZs, so that each PG has two segments in each AZ [2].

Another main change in Aurora compared to traditional systems is that they have pushed the log processing down to the storage layer. The redo log is the only part that is written from the database layer to the storage layer [1]. One main design principle is to minimize the latency of the foreground write request. This is done by prioritizing foreground write requests. Aurora will i.e. not perform background activities such as garbage collection when the storage node is busy processing a foreground write request. Most operations, such as materializing pages, are moved to the background on the storage node.

Amazon argues protocols like two-phase commit and the variants of Paxos can be both expensive and incur excessive network overhead. Instead, Aurora uses a combination of quorum I/O, locally observable state, and log-sequence numbers for its commit and membership processing [2]. Log-sequence numbers (LSN) are used for uniquely identifying a log record. The LSN space is monotonically increasing, assigned by the database instance, and common across the database volume. Aurora uses a segmented redo log, where the database instance storage driver shuffles the log records to write buffers individual to each storage node. The storage driver asynchronously sends the data to the nodes, receives acknowledgements, and establishes consistency points [2]. All parts of log writes execute asynchronously, including sending the log to the storage node, processing the log on the storage node, and acknowledging back to the database instance.

In Aurora, a log record additionally includes the following LSNs:

- The LSN of the preceding log record in the volume. Typically only used for disaster recovery.
- The previous LSN of the segment. This is used to identify missing records, and gossip with other storage nodes to fill these holes.
- The previous LSN of the block being modified. This is used to materialize blocks on demand.

Aurora keeps track of various LSN numbers for its commit and membership processing. When a storage node receives new log records, it may advance a Segment Complete LSN (SCL). This represents the latest point where it knows it has received all log records, and is used to identify missing writes. The SCL is in turn sent to the database instance as a write ACK [2]. Once four of the six nodes in a PG respond with its SCL, the database instance advances a Protection Group Complete LSN (PGCL) [2]. This represents the point where the PG has made the writes durable. The database instance also advances a Volume Complete LSN (VCL), when there are no pending writes preventing PGCL from advancing for one of the PGs in the volume [2]. A commit is acknowledged once all data modified by the transaction is made durable. This happens when the commit redo record (System Commit Number/SCN) is below the VCL [2].

During crash recovery, Aurora will need to recompute the LSN state values. Aurora has to first establish a read quorum. The database can then recompute the VCL and PGCL values by finding read quorum consistency points across SCLs [2]. Any records which are past the newly computed VCL will be truncated. If write quorum can't be met, Aurora will start repair and rebuild the failed segments [1]. When both read and write quorum is met, Aurora increments an epoch in its storage metadata service and sends this to the storage nodes. Storage nodes will in turn not accept requests for stale volume epochs [2].

Aurora has taken steps to optimize the read performance. This is mainly achieved by avoiding read quorums where possible. The latest version of a data block can either be found in cache or in the latest durable version of the block in one of the segments of the PG it belongs to. The database instance will always keep track of which segments have the last durable version of a data block. It can thus directly request it directly from that segment [2]. This allows Aurora to avoid quorum reads, which greatly reduces network overhead.

Aurora is also able to avoid distributed consensus in the general case for writes and commits by managing consistency points in the database [2]. In Aurora, storage nodes do not have a vote in whether or not to accept a write. If a node misses a write, they will gossip with other nodes in the PG to catch up [1]. For commit processing, the worker thread will hand off the transaction to a commit queue and return to the task queue to find new requests. A dedicated commit thread will scan the commit queue and send acknowledgements once the data is durable [2].

Similarly, Aurora uses quorum sets to determine group membership. Like with volume epochs, Aurora maintains a membership epoch which is monotonically incremented for each change. Clients with stale membership epochs will have their requests rejected and must update their membership information. All membership epoch increments must have a write quorum, and be passed within the correct epoch [2]. This is just like any other regular request. The segments of a PG make up a quorum set, and boolean logic is used to ensure read/write quorum for the PG. If a segment fails, the segment can be replaced with a new segment. In this case, a new quorum set will be formed with the healthy segments and the new segment. Aurora will transition from one quorum set to another by incrementing the membership epoch [2].

The numbers provided by Amazon show that Aurora can scale linearly with instance sizes for both write-only and read-only workloads. Each increment in instance size doubles the available memory and vCPUs. In the case of the highest available instance size, Aurora performs up to 5x better than MySQL 5.7 for reads and writes per second [1].

For varying database sizes, Amazon heavily outperforms MySQL even for out-of-cache sizes. With a 100GB database, Aurora performs 67x better for writes/s than MySQL. With a 1TB database, Aurora performs 34x better. This is with a buffer cache of 170GB [1].

Read replica lag is also significantly reduced. When scaling from 1000 writes per sec to 10 000 writes per sec, Aurora's replica lag only grows from 2.62 ms to 5.38 ms. MySQL on the other hand grows from less than 1000 ms to over 300 000 ms [1].

Amazon has also performed TBC benchmarks to investigate the impact of hot row contention. They experience a 2.3x to 16.3x improvement of throughput with Aurora compared to MySQL 5.7 as the workload ranges from 500 connections with a 10GB data size to 5000 connections and 100GB data size [1].

6 Alibaba PolarDB

Similar to Aurora, PolarDB also aims at decoupling compute and storage in order to be more flexible and allow for shared storage in a cloud environment. However, most traditional database systems fail to utilize the full potential of emerging technologies such as RDMA and NVMe SSD's. Alibaba has therefore created its own distributed file system, called PolarFS, which provides low latency, high throughput, and high availability by fully utilizing these emerging technologies [7].

PolarFS also implements a lightweight network and I/O stack which is kept entirely in user space. PolarFS provides a POSIX-like API which should be compiled into the database process and replace the regular file system calls [11]. DMA is heavily used to transfer data between main memory and RDMA NICs/NVMe disks. Furthermore, they use a specialized version of Raft, called ParallelRaft, to achieve consensus among nodes [7].

PolarFS is divided into two layers. The storage layer handles the data resources on the storage nodes and provides a database volume for each database instance. The file system layer supports file management of the volume, in addition to handling mutexes and synchronization of file system metadata access [7].

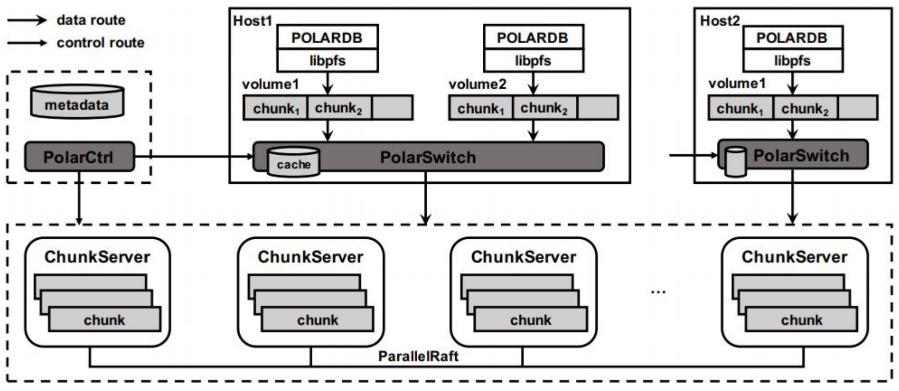


Figure 7: Overall PolarDB architecture. Source: [7]

Figure 7 illustrates PolarFS's architecture. libpfs is a user space file system with a POSIX-style API. This is linked into PolarDB. PolarSwitch are located on the compute nodes and redirect I/O from the application to the ChunkServers. ChunkServers are located on the storage node and serve I/O requests. PolarCtrl is the control plane, and has a set of masters in addition to agents deployed on the storage and compute nodes [7].

The file system should provide shared and parallel file access, where multiple storage nodes should be able to access the system concurrently. There is also a need for synchronizing file system metadata across to the various storage nodes in a serializable manner [11]. In PolarDB, each database instance has a volume, consisting of a list of chunks. The chunks are distributed across multiple ChunkServers. Each chunk is at 10GB, and is the smallest unit of data distributed across the nodes. The size is chosen so the chunk metadata can be cached in main memory by PolarSwitch. The chunks are replicated across three replicas, which are on separate racks [7]. A chunk should also not span across disks. The replicas of a chunk forms a consensus group, with one leader and the rest are followers [11].

A chunk is further divided locally on the ChunkServer. The ChunkServer divides the chunk into blocks, where each block has a size of 64 kB. The chunks use thin provisioning, which means new blocks are allocated and mapped to the chunk on demand [10]. This is the opposite to traditional fat provisioning, where the storage space is allocated up-front. The ChunkServer also handles the mapping of the chunk's LBA (logical block address) to blocks, and is cached in main memory. In addition, the ChunkServer also handles the bitmap of free blocks on each disk.

In the PolarFS architecture, the PolarSwitch is a daemon which runs on the compute nodes. The libpfs layer forwards I/O requests to PolarSwitch, which PolarSwitch translates into a new request which is sent to the underlying leading ChunkServer where the chunk resides [11]. Only the leader can answer I/O requests for a chunk. PolarCtrl keeps track of the leadership changes, and PolarSwitch uses a locally cached and synchronized copy in order to resolve the I/O requests [7].

The ChunkServer's responsibility is to store and provide access to chunks. Typically, multiple ChunkServer run on each storage node. Each ChunkServer has a dedicated NVMe disk and CPU core to avoid any contention between each other [10]. Modifications to a chunk is written to a WAL before the changes are written to the chunk blocks. A ChunkServer will typically write the WAL to a 3D XPoint SSD buffer. However, if the buffer is full and no place can be freed, the log may be written to a NVMe disk instead [7]. ChunkServers replicate I/O requests to each other using a ParallelRaft consensus group [43]. If a ChunkServer goes down, it will autonomously try to reconnect to the group. PolarCtrl will also probe for faulty servers, and remove it if so deemed necessary.

PolarCtrl is the control plane in the PolarFS architecture [11]. It provides cluster management services such as node/volume management, resource allocation, and metadata synchronization [10]. PolarCtrl is designed to handle the following responsibilities

- Keeping track of membership and liveness of ChunkServers and handling ChunkServer migrations
- Maintaining volume and chunk locations in the metadata cache
- Volume creation and chunk allocation to ChunkServers
- Synchronizing metadata with PolarSwitch
- Monitoring various metrics for volumes and chunks
- Scheduling various CRC checks periodically

PolarSwitch will fetch location metadata from PolarCtrl and store it in a local cache. If the local cache becomes outdated, a new fetch from PolarCtrl will occur.

A typical write request goes through multiple steps. libpfs and PolarSwitch has a shared memory segment with multiple ring buffers. When PolarDB writes to libpfs, it splits up the write request into multiple block requests and enqueues this in the shared ring buffer. PolarSwitch constantly polls the ring buffers, and upon a new request, dequeues the request and sends it to the corresponding leading ChunkServer based on metadata and routing information obtained from PolarCtrl. On the ChunkServer, the RDMA NIC moves the request into a buffer and notifies the ChunkServer through a request queue. The ChunkServer constantly polls this queue, and upon finding a new request, it starts processing said request at once. The ChunkServer then writes the request to the log block on disk, and propagates the request to the follower nodes. A similar process happens at the follower nodes; the RDMA NIC adds the request to a buffer and notifies the ChunkServer through a request queue. The follower ChunkServers then write the request to disk and respond back to the leader ChunkServer. When a majority of followers successfully reply back to the leader, the write request is applied to the data blocks. In turn, the ChunkServer replies back to PolarSwitch through RDMA, which marks the request done, and respond to the client [7]. Read requests are on the other hand handled directly on the ChunkServer leader [10].

PolarFS started out using Raft for group consensus. However, Raft showed to have multiple limitations in its design that would severely impact both throughput and latency. Some of the limitations experienced were Raft's lack of support for having holes in the log for both leader and follower [7]. This means that followers would need to acknowledge a request, the leader would commit, and then the followers would apply the change in sequence. Furthermore, a follower must acknowledge log entries in sequence, which results in the follower being unable to accept log entries arriving out of order.

The PolarDB/FS team therefore chose to build upon Raft to remedy some of these limitations. The new proposed protocol is named ParallelRaft [43]. ParallelRaft leverages the database's own mechanisms for ensuring consistency, and can therefore relax on some of the constraints of Raft. ParallelRaft has several guarantees that consistency is maintained, among others that none of the committed modifications will be lost in any corner cases and that it is compliant with the storage semantics of traditional database systems. In contrast to Raft, ParallelRaft does both acknowledgement and commit out of order [10]. This also implies you may have holes in the log. In order to solve the missing holes issue, PolarFS uses a look-behind-buffer which contains the LBA modified by the previous N log entries. A follower can then tell if a log entry conflicts with a missing log entry if its overlapping with some missing previous log entries. In this case, PolarFS will put the log entry in a list and complete it once the missing log entry is retrieved [7].

Since there may be holes in the log, special care needs to be taken when electing a new leader. PolarFS solves this by having an additional merge stage where the leader candidate obtains it's unseen entries from a quorum and commits them locally [7]. In short, a leader election will go through the following steps.

1. The leader candidate receives the log entries from the followers and merges these with its local entries.
2. The leader candidate synchronizes the state with the followers.
3. The leader candidate commits the changes and sends a notification to the followers to commit as well

ParallelRaft makes use of checkpoints, where log entries before the checkpoint are applied to disk. However, a checkpoint might also contain some log entries that are committed after the checkpoint. ParallelRaft uses this checkpointing mechanism for leader election, where the node with the latest checkpoint is elected to become a leader candidate [10].

If a follower is lagging or has become stale, the follower would need to catch up to the leader. PolarFS provides two different mechanisms for catch-up. The method chosen depends upon how stale the follower is [7]. A *fast catch up* is done if the follower is only slightly lagging behind the leader. However, if it has been stale for some time, a *streaming catch up* needs to be done. The fast catch up only works if the leaders checkpoint is older than the latest log index on the follower. Otherwise, the leader might have pruned the log entries previous to its checkpoint.

With fast catchup, log holes between the followers checkpoint and the leaders checkpoint are identified by using the look behind buffer mentioned previously. Once identified, the follower copies these missing modifications directly from the leaders data blocks [10]. The holes after a leader's checkpoint is filled by copying directly from the leaders log blocks.

With streaming catchup, the follower copies the datablocks and content from after the leaders checkpoint. This is done by dividing the chunks into relatively small pieces of 128KB in small tasks. The motivation behind this is claimed to be more controllable resynchronization [7].

Most modern databases support taking snapshots of the database for a given moment in time. PolarFS provides a so-called disk outage consistency snapshot. For a snapshot initiated at timepoint T, all I/O operations before a timepoint T_0 is included, while operations after T is excluded. Operations inside the interval $[T_0, T]$ is undetermined. During crash recovery, the latest disk outage consistency snapshot is used as a building foundation for the database nodes. PolarFS even allows for user load during snapshot creation. This is done by PolarCtrl telling PolarSwitch to make a snapshot. PolarSwitch then attaches a tag to indicate that this request happened after the snapshot creation begun. When receiving a request with this special tag, a ChunkServer will make a snapshot by copying block mapping meta information and handle further requests in a Copy-On-Write fashion. Once the snapshot creation is finished, PolarSwitch will quit adding this special tag [7].

PolarDB has also made significant changes to the InnoDB storage engine to facilitate physical replication. For this, PolarDB utilizes InnoDB's redo logs[38]. InnoDB uses the redo log to store physical page level operations for crash recovery. PolarDB extends this functionality to deploy multiple read replicas for read load sharing. Instead of just doing redo application during crash recovery, PolarDB does redo application to data pages at runtime where the replica applies redo logs generated on the primary[39]. The replication lag can in this case thus be represented as primary.written_lsn - replica.applied_lsn. A stated goal is to reduce the replication lag. This is both for better service, but also due to the replication lag imposing some constraints on the primary when it comes to flushing. Additionally, small replication lag will help reduce memory usage/redo application time on the replica[39].

Alibaba has also taken steps to optimize the redo application code. Traditionally, redo application was only used for crash recovery and was implemented in a single threaded fashion. PolarDB instead read the redo logs in a separate asynchronous reader threads. The parsing of redo logs is single threaded, and are stored in multiple hash tables. Then, multiple threads apply the redo logs to the pages in the buffer pool[38]. Alibaba has purposefully made the decision to apply the redo logs without any consideration of atomicity or ordering, and then deal with the consequences at a later stage[39]. When a replica receives a batch of redo logs from the primary, the replica will only apply the changes made to the pages in the buffer pool. The replica will keep track of the received batch of redo logs, in case new pages are read in from disk. In practice, this means PolarDB won't incur any extra IO for redo log application[38]. Alibaba has also made the design decision not to apply the batches atomically, but instead handle the instances where physical inconsistency may occur on the replica. This would typically happen

- on the primary when multiple pages are modified (such as a B-tree merge or split)
- on the replica when multiple pages are read (such as a range scan)

In essence, PolarDB is able to solve this problem by introducing a new log entry (MLOG_INDEX_LOCK_ACQUIRE), so that an mini-transaction (MTR) on the replica will be able to detect when it's touching on pages that has undergone a split during a batch that is being applied. In this case, the MTR will close and restart the MTR[39]. The advantages to this approach is that there are no system level locking for atomic batch application, no index level locking for page splits/merges, only affected MTRs have to retry, but no transaction level retries.

Additionally, the primary has some constraints on when it is able to flush pages. A replica doesn't know of any changes that are greater than replica.applied_lsn. This implies that the primary cannot flush a page if page.newest_modification > replica.applied_lsn[39]. This creates an issue when dealing with hot pages. When this happens, the page's newest_modification will be frequently updated, so the primary cannot flush it. A consequence of this is that the primary cannot make a checkpoint. PolarDB makes two approaches to deal with this problem. The first is to pin known hot pages in the replica's buffer pool at startup, so that the replicas will never read them from the disk. This will allow the primary to flush the pages when needed[38]. It doesn't, however, solve the issue of random hot pages. The way this is dealt with, is that the primary makes a copy of the flushed page, and once the copied page is flushable, it will be written to disk and the flush list accordingly[39].

PolarDB has also made some changes to how the read view of open read/write transactions. This is due to the fact that the replica only performs reads and has no local read view, will thus need to know the open transactions at the primary at the current applied_lsn[39]. The initial read view is sent by the primary as part of the handshake. The primary will also send out redo log entries on each transaction start and commit. The replica will then parse these and build a new read view. The replicas use a global read view, so the read view on the replicas will be shared among all transactions until the applied_lsn counter is moved[38]. Additionally to this, PolarDB has also made some changes to how purging is done.

Alibaba has also performed various tests, both for measuring PolarDB and PolarFS performance. The PolarFS system was tested against ext4 on local SSD and CephFS. CephFS and PolarFS was run with 6 storage nodes and one client. However, Ceph's filesystem only support TCP/IP communication, so this had to be used for the tests. PolarFS came out relatively close to the ext4 system, and outperformed CephFS substantially when measuring latency [7]. This is mainly because PolarFS avoids thread context switching and rescheduling. Additionally, PolarFS optimizes memory allocation and paging. That CephFS is only able to run in a TCP/IP configuration is also very likely a contributing factor.

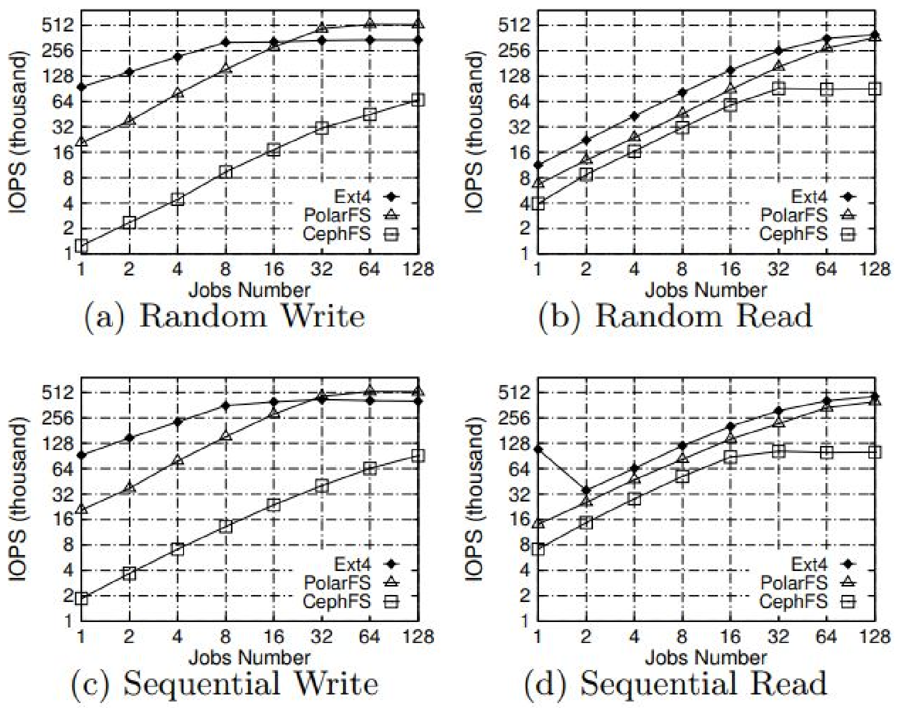


Figure 8: PolarFS - I/O throughput under different loads. Source: [7]

With regards to I/O throughput, the same characteristics were produced. PolarFS came out relatively close to the ext4 system, and outperformed CephFS substantially also for I/O throughput.

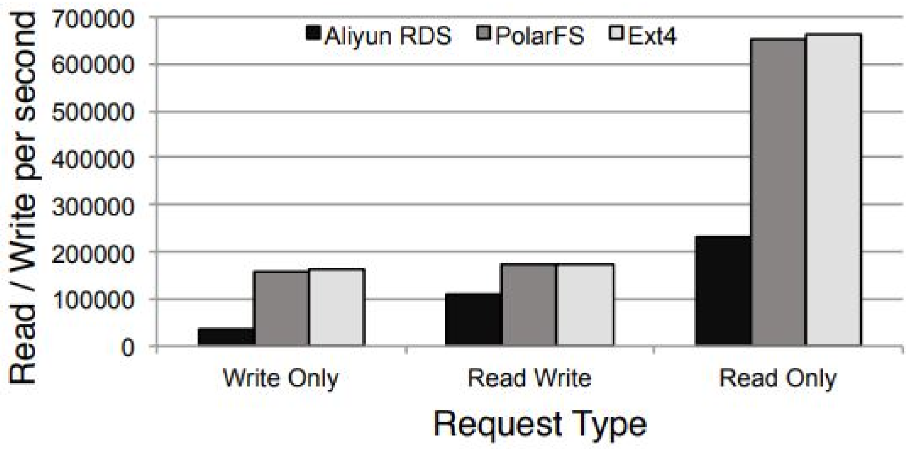


Figure 9: PolarDB - read/writes per second. Source: [7]

PolarDB was tested against both local SSD, PolarDB when run on PolarFS and the Alibaba MySQL cloud service. The tests were run with Sysbench for both RO, WO, and mixed read/write on a database with 250 tables, where each table had 8 500 000 records. As seen in figure 9, PolarDB on PolarFS showed very close performance to local SSD. This shows the benefits of the ultra-low I/O latency of PolarFS [7].

7 Benchmarking RDS MySQL, AWS Aurora, and Alibaba PolarDB

In the recent years, there has been made significant innovations in how databases can utilize the infrastructure of the cloud. Alibaba's PolarDB with its underlying shared file system PolarFS, and Amazon's RDS and Aurora services are some of the major players in this field. The benchmarks and tests presented by Amazon and Alibaba show that these systems perform significantly better when compared to traditional setups. The tests published are mostly done by the respective owners, and might therefore be susceptible of showing their own database offering in a favourable light. There are also few published tests and benchmarks that compare the new cloud based database services directly. Instead, most benchmarks compare the database services to a traditional database setup, typically with MySQL 5.6 or 5.7 with a hot standby or read replica. The goal of this thesis is to do a fair benchmark comparison of the database offerings of Amazon and Alibaba. We have therefore carried out similar tests on comparable hardware on Amazon's RDS MySQL and Aurora, as well as Alibaba's PolarDB offering. These tests gives an insight into how the systems compare to each other, as well as verifying the benchmarks published by Amazon and Alibaba.

One of the main challenges for testing and comparing results from different systems is to create a level test ground. This is necessary in order to give a fair comparison between the systems, where the results can be compared in a meaningful manner. This section will describe the steps taken to create such an environment and the various features used in AWS and Alicloud to achieve this.

7.1 Relevant applications and tools

The following section will present some of the tools used for performing the benchmarks. The main goal was to create an easily reproducible setup, which lends itself well to automation and scripting. The main tools used for this task are Terraform [30] for easy setup and configuration of the test infrastructure, and Sysbench [8] for performing the actual tests towards the databases. These tools are some of the most common tools used in the market.

7.1.1 Terraform

Terraform is a tool developed by HashiCorp used for describing infrastructure as code. The stated main goals are to safely and predictably create, change and improve infrastructure by describing it as declarative configuration files which can be easily shared and versioned. Terraform is able do build a dependency graph of the infrastructure, allowing it to create and change the infrastructure in a safe way. This is useful for instance where a small change could have cascading effects. Another benefit is that since the infrastructure is codified, it becomes easier to automate.

Additionally, Terraform supports a range of different "providers", allowing for interfacing between Terraform and external resources. In Terraform, a provider is tasked with understanding API interactions and exposing resources. Notably, many cloud platforms have Terraform providers for their services, allowing for easy interaction. In example, the provider for AWS exposes almost the entire platform to Terraform, making setup and infrastructure management very easy.

7.1.2 Sysbench

Sysbench is a commonly used tool for performing benchmarking on databases, as well as other system aspects such as file I/O, CPU, memory etc. When testing databases, Sysbench will connect to the database through the usual client interfaces (network or socket-based communication) and perform the tests like normal clients would. The current version of Sysbench is based on LuaJIT, and as a consequence supports user-scriptable tests. By default, sysbench provides various tests for OLTP transactions, namely:

- oltp_delete
- oltp_insert
- oltp_point_select
- oltp_read_only
- oltp_write_only
- oltp_read_write
- oltp_update_index
- oltp_update_non_index
- select_random_points
- select_random_ranges

The Lua scripts are relatively simple. Each script is composed of two functions, namely *prepare_statements()* and *event()*. The *prepare_statements()* function is responsible for creating global nested tables for prepared statements and their parameters. The *event()* function on the other hand is responsible for performing the actual queries.

Taking a closer look at *oltp_read_only*, we see that it will perform point SELECT queries and, if enabled, a set of range queries. The range queries include a set of simple range SELECTs, SELECT SUM() ranges, SELECT ORDER BY ranges, and SELECT DISTINCT ranges.

The *oltp_read_write* test performs point SELECTs and, if enabled, the same range queries as the *oltp_read_only* test. Additionally, it will perform UPDATE index and non-indexed queries, as well as a set of DELETE/INSERT combinations.

The *oltp_write_only* test performs the same UPDATE index and non-indexed queries, in addition to the DELETE/INSERT combinations found in the *oltp_read_write* test.

The default settings have 10 point SELECTS per transaction, 1 query of each range category, and then 1 query of of the UPDATE categories, as well as 1 DELETE/INSERT query per transaction. The default range size for the range SELECTs is set to 100.

Typical usage of sysbench consists of preparing a dataset in a separate test database (typically named 'sbtest'), running tests towards this database using the various test types and tuning different parameters, before doing a cleanup which wipes the test database. The size of the database is determined by how many tables are created and how many rows are inserted in each respective table, controlled by the '–tables' and '–table-size' parameters. The tables have the following form

```
CREATE TABLE `sbtest1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `k` int(11) NOT NULL DEFAULT '0',
  `c` char(120) NOT NULL DEFAULT '',
  `pad` char(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `k_1` (`k`)
) ENGINE=InnoDB AUTO_INCREMENT=25001 DEFAULT CHARSET=utf8mb4
```

, with random values for the fields.

7.2 Test setup

As stated above, one of the main goals were to create a level environment for testing. This was somewhat challenging, given that Alibaba's and Amazon's cloud offerings differ in aspects such as which services they provide and their underlying infrastructure. The main goal was to compare Amazon Aurora and Alibaba's PolarDB. As a reference point, tests were also performed against a 'regular' Amazon RDS MySQL 5.7 instance running on Amazon EBS storage, as a baseline reference measurement.

7.2.1 Amazon AWS

As mentioned in section 4, AWS is divided into multiple regions, each having a set of Availablity Zones (AZs) with low latency high throughput communication links. Each of the AZ's operate independently, with separate power and communications infrastructure. The tests were conducted using a Amazon EC2 instance in conjunction with the database being tested. The EC2 instance was located in the same region, and when a single-instance database was tested, also in the same AZ as the database. This was done in order to minimize latency. For convenience, the *US-East-1* region was used. The test setup made use of the following AWS resources:

- EC2 compute instance located in the same region as the database being tested.
- AWS Aurora/RDS MySQL database, both single-instance and with multiple read replicas.
- CloudWatch alarms for monitoring performance of both the EC2 and database instances.
- VPC for allowing the EC2 instance to connect to the DB

The following figure demonstrates the test setup:

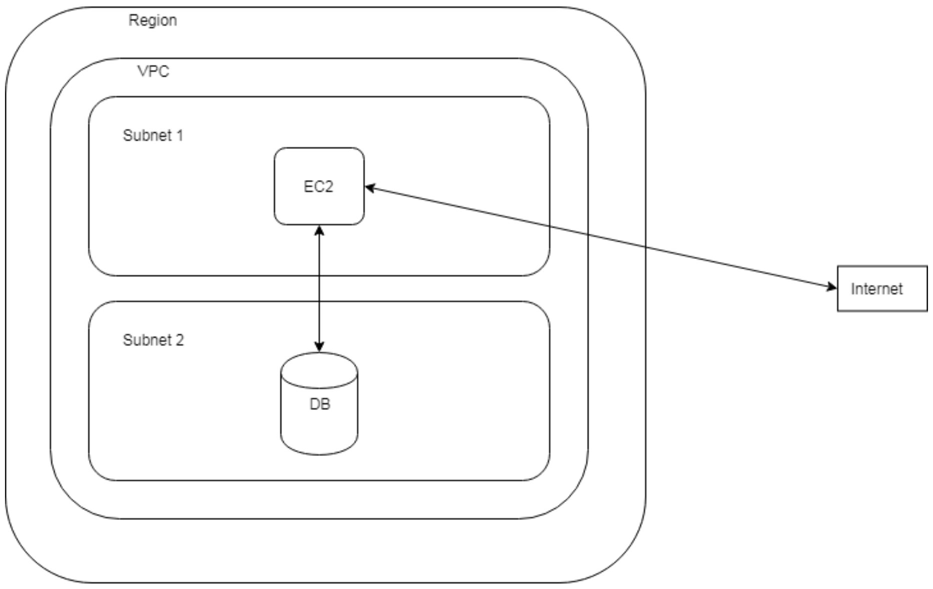


Figure 10: AWS test setup

Aurora is offered in various versions. Aurora is available with both MySQL and PostgreSQL compatibility. The user is also able to choose which MySQL version Aurora should be based on. The options are either 5.6 or 5.7. The 5.6 version also provides the option of enabling parallel query processing. There are some notable missing features for Aurora based on MySQL 5.7 compared to Aurora based on 5.6 for the time being. These include

- Asynchronous key prefetch
- Hash joins
- Native functions for synchronously invoking AWS Lambda functions
- Scan batching
- Migrating data from MySQL to the Amazon S3 storage service

For the tests we chose to opt for the MySQL 5.7 compatible version, for having a more level environment.

Aurora has multiple instance types, each optimized for different workloads. The instance types currently offered are the following

- db.m* series - an all-round instance type that provide a balance of compute, memory, and network resources. Amazon deems this a good fit for most applications.

- db.r* and db.x* series - an instance type optimized for memory-intensive applications. The db.r* series has previously been used in Amazon's own benchmarking tests.
- db.t* series - an instance type that utilize the burst IOPS capabilities, where an instance is able to burst up above baseline performance for a small amount of time

There are multiple generations of each series. For the db.m series, the db.m5.* is currently the latest generation. Similarly are db.r5.* and db.x1e.* the latest for the memory intensive types. db.t3.* is the latest for the burstable instance types. Amazon still feature older generations for backwards compatibility, but there's signs these are being phased out. For instance, the EU (Paris) and South America (São Paulo) regions don't support the older db.r3.* instance types.

The instance types are further divided by the amount of CPU cores, memory, and a few additional factors. Amazon deals with compute resources a bit differently than most other providers. Amazon uses a term named "vCPU", short for virtual CPU, which is described as a "unit of capacity" that can be used to compare instances. The stated goal is to provide an abstract scale for compute power, separated from the underlying hardware. In practice, the vCPU count is mapped to the amount of CPU cores and the amount of threads per core each database instance has available. For the db.r5.* instance series, for instance, the vCPU count is mapped to the number of cores multiplied by the number of threads per core.

Amazon also has an "enhanced networking" feature, giving high performance networking capabilities to supported instances. This feature makes use of single root I/O virtualization which gives higher I/O performance and lower CPU utilization than traditional virtualized network interfaces. Some of the benefits include higher bandwidth, higher packet per second count, and lower inter-instance latency. There are two available interfaces. Most instance types support the Elastic Network Adapter (ENA), which gives speeds up to 100Gbps. Some lower-tiered instances instead support the Intel 82599 Virtual Function interface, with speeds up to 10Gbps. Additionally, when an RDS database instance is launched into a subnet which contain another instance (such as a EC2 compute instance), the database instance will automatically be upgraded to use of the "enhanced networking" feature. The feature also comes with no additional costs for usage.

Amazon allow customers to choose between burstable IOPS and provisioned IOPS. When using burstable IOPS, Amazon will use a credits system, where an instance may draw from credits pool that is refilled at a certain rate. If the credits pool becomes empty, the instance will fallback to a baseline IOPS rate. This can lead to various problems when conducting performance tests. Once the credits pool is emptied, the baseline rate will act as a bottleneck.

It is therefore important that both the database instance and compute instance isn't limited by this rate. This caused problems for the early tests where the EC2 instance exhausted the credits pool, and thus limited the performance. This was easily remedied by choosing the m5.xlarge, m5.2xlarge and c5.2xlarge instance sizes, which isn't bound by the credits system. The m5.xlarge was used when running tests on the smaller database instance sizes, while the m5.2xlarge and c5.2xlarge was used for the larger instance sizes. The database instances had provisioned IOPS selected. CloudWatch was then set up for monitoring that none of the instances ran out of available IOPS. The buffer pool size for all tests was set to 3/4 of the available memory for the instance.

For these tests, we focused on the db.r5.* range. This is the latest generation of memory-optimized database instances available, and the ones Amazon uses for their own benchmarking. This would as a consequence be ideal for verifying the performance of Aurora and RDS MySQL. The database instances in the db.r5.* range have some of the following properties:

Instance	vCPU	Memory (GB)
db.r5.large	2	16
db.r5.xlarge	4	32
db.r5.2xlarge	8	64
db.r5.4xlarge	16	128
db.r5.12xlarge	48	384

Terraform is used to instrument and automate the tests for the AWS setup. Amazon has a Terraform "provider" that makes it possible to access nearly all AWS services from Terraform. This allows for easily reproducible tests.

7.2.2 PolarDB

Alibaba's new database is currently limited to China and some other Asian countries. As a consequence, international customers wouldn't normally be able to access it from abroad. For this masters project, Alibaba granted access to, and free compute credits for PolarDB and their compute service, ECS. However, there turned out to be some practical difficulties in order to access the systems. The PolarDB instances can only be created using a Chinese user account. Account registration further require a Chinese phone number, which isn't particularly easy to obtain. The end result was that Alibaba created a PolarDB instance using their own accounts, and provided me with the login credentials and endpoints. Additionally, the user interface for PolarDB in Alicloud is only available in Chinese, which made it relatively hard to navigate. Due to these difficulties, the resources and access to perform tests on PolarDB were ready only first in early April. The test setup made use of the following PolarDB resources:

- ECS compute instance located in the same region as the database being tested.
- PolarDB database instance with a read replica.

Alibaba provided me with one instance size, namely polar.mysql.x8.xlarge (8 cores 64GB RAM). This is similar to Aurora's db.r5.2xlarge. However, the setup provided consisted of a master with a replica. A consequence is that no comparisons to how well PolarDB scales with instance sizes can be performed. Instead, we have focused on how PolarDB performs under various workloads, with varying database sizes and thread counts.

8 Results

As mentioned in section 7.2.1, a various set of tests has been carried out, mainly revolving around RO and RW tests. For the AWS setup, we want to compare how the various instance sizes scaled, and how they scaled with different number of threads. We want to compare how Aurora and regular RDS MySQL compare in similar setup configuration. The tests has been based on the db.r5.* series of database instances, as these are the latest generation memory-optimized databases, which Amazon uses for their own benchmark tests. We then contrast the findings with how PolarDB performs on the same tests. Since we only had access to a single instance size for the PolarDB database, we will not be able to compare how it scales with increasing instance sizes. However, we will be able to see what impact multiple threads have on the system.

The tests were run 5 times on each instance size for 600 seconds. There was also a short warmup period run before the actual tests. The figures below show the average of these five runs, with error bars corresponding to the standard deviation in the measurement runs. Tests with varying database sizes were performed. The sizes ranged from a database containing 25 tables with 25000 records per table, to 250 tables and 2700000 records per table. The thread count varied between 2, 4, 8, 16, 32, 64. The test with 2, 4, and 8 threads were only carried out on PolarDB. This is due to an early assumption that PolarDB would scale best for a smaller number of threads. The results gathered and presented in the following sections show that this isn't necessarily the case. We also conducted some extra tests to compare the systems directly to each other, such as Aurora compared to PolarDB for a database size of 10GB. Additionally, a test with a larger number of threads (500 threads) was performed. This was done because Aurora seems to perform better for higher thread counts, and the tests with 64 threads showed signs of not exploiting the full potential of Aurora.

The following tests were carried out:

- 250 tables and 25000 records per table for RDS MySQL, Aurora (16, 32, and 64 threads scaling across db.r5.large to db.r5.4xlarge).
- 250 tables and 25000 records per table for PolarDB (2, 4, 8, 16, 32, and 64 threads for the polar.mysql.x8.xlarge instance)
- 250 tables and 2700000 records per table for RDS MySQL, Aurora (16, 32, and 64 threads scaling across db.r5.large to db.r5.4xlarge).
- 250 tables and 2700000 records per table for PolarDB (2, 4, 8, 16, 32, and 64

threads for the polar.mysql.x8.xlarge instance)

- 250 tables and 200000 records per table for Aurora's db.r5.2xlarge and PolarDB's polar.mysql.x8.xlarge with 64 threads.
- 250 tables and 25000 records per table for RDS MySQL and Aurora db.r5.2xlarge and PolarDB's polar.mysql.x8.xlarge with 500 threads

8.1 Results with 250 tables, 25000 records per table

The figures shown below are the result of tests with 250 tables with 25000 records per table. This is equal to a database size that is close to 1.6GB, and is thus easily held in memory for all instance sizes. These tests will be presented using queries per second as metric, as this allows for better mapping to the results presented by Amazon and Alibaba.

Amazon reported their achieved performance for the various instance sizes in their SIGMOD 2017 paper[1], which was conducted on a similar dataset with similarly scaling instance sizes. The main caveats are that Amazon didn't specify how many connection threads they performed the tests with, nor did they disclose the exact configuration the Aurora instances were set up with. It is also worth noting that the tests were performed on the db.r3.* series of instances, while our new tests were performed on the db.r5.* instances.

8.1.1 Amazon Aurora

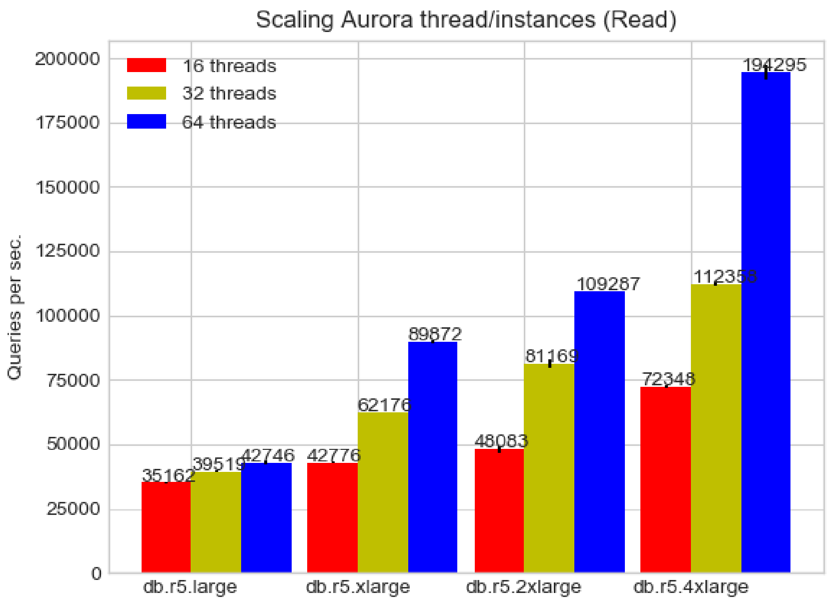


Figure 11: Aurora read performance

Aurora's read performance for these tests show that each step up in instance size brings an increase to performance, but the effect is best seen for the larger thread counts. With 16 threads, the queries per second are around $\sim$ 35000 for the db.r5.large database, but scale up to $\sim$ 72000 for the db.r5.4xlarge instance size, which correspond to a 2x increase in performance. With 64 threads, however, the base performance is at $\sim$ 43000 qps for the db.r5.large, but climb up to $\sim$ 194000 qps for the db.r5.4xlarge. This in turn correspond to a 4.6x increase in performance.

We see from the result that the instances scale as thread count is increased. Due to network latencies, a small but significant amount of time will be lost as the query and response propagates through the network. As a consequence, a larger count of database connections can be facilitated than there are hardware threads. This is especially of significance for writes, where the database must wait for a quorum to be formed and the changes made durable before being able to respond. The same is true to a lesser extent for PolarDB, where low-latency RDMA is able to reduce the impact of this effect.

CPU characteristics:

Instance	CPU load
Aurora large read 16	85-95%
Aurora large read 64	90-100%
Aurora xlarge 16	67-74%
Aurora xlarge 16	90-100%
Aurora 2xlarge 16	20-25%
Aurora 2xlarge 32	35-40%
Aurora 2xlarge 64	65-70%
Aurora 4xlarge 16	21-25%
Aurora 4xlarge 32	38-42%
Aurora 4xlarge 64	60-70%

Monitoring the CPU utilization for the various runs, we see that both the db.r5.large and db.r5.xlarge have very high CPU utilization, especially for the 64 threads runs. Especially, the db.r5.large is being CPU-bottlenecked for nearly all tests, staying somewhere between 85-95% for the 16 threads test, and spiking to 90-100% for the 64 threads test. This explains why the db.r5.large instance see little scaling for the different thread counts. The db.r5.xlarge, however, start at 67-74% for 16 threads, and average around 90% for the 64 threads run. The larger instances, such as the db.r5.2xlarge and db.r5.4xlarge, don't suffer from this issue, with CPU utilization being generally low/medium, with a a max utilization of 70% for their respective 64 threads testruns.

The read performance shown here differs somewhat from the presented performance by Amazon. The db.r5.large and db.r5.xlarge performs slightly better than what is reported by Amazon. For the db.r5.large instance, Amazon reports approx. 30000 queries per second, but the highest result we achieved is 42746 queries per second. Similarly for the db.r5.xlarge instance, Amazon achieves close to 70000 queries per second, while our results show 89872 queries per second. For the larger instance sizes, our results are lower than that presented by Amazon. Our results for the db.r5.2xlarge is 109287 queries per second, while Amazon reports around 180000 queries per second. Similarly, they claim that Aurora's 4xlarge instance size is able to perform over 300 000 queries per second, while our result tops out on 194295 queries per second. This might indicate that Amazon's own tests were carried out with a larger number of threads, which the larger instances benefit from, but as a consequence also CPU-bottleneck the smaller instances.

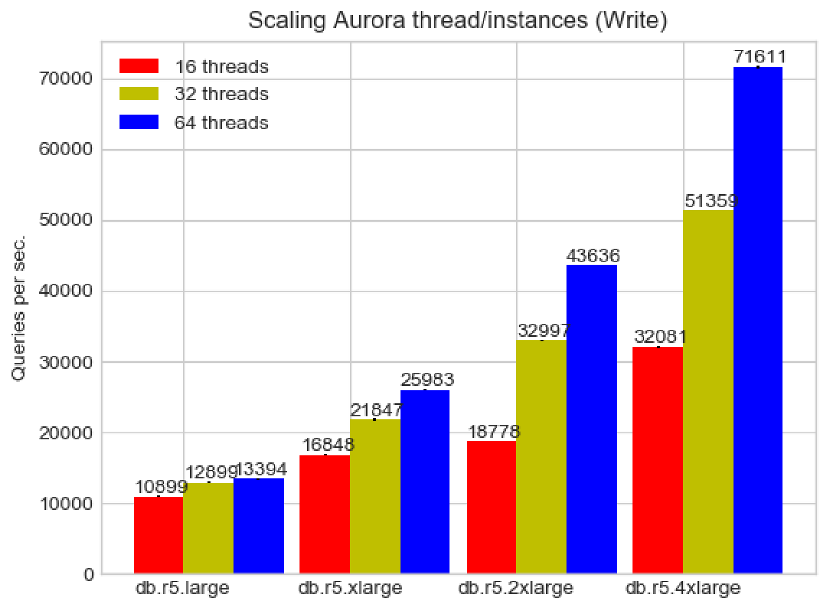


Figure 12: Aurora write performance

Similarly to the read performance, each increase in instance size brings an increase in write performance. The largest scaling improvements are seen for the large thread counts. For the db.r5.large instance size, it is worth noting that the difference between 16 threads and 64 threads is only roughly $\sim$ 2500 queries per second. The 16 thread tests see a $\sim$ 2.95 x increase in performance when going from the db.r5.large instance size to the db.r5.instance size, while the 64 thread tests show a factor of $\sim$ 5.34 x increase between the two instances.

CPU characteristics:

Instance	CPU load
Aurora large 16	85-90%
Aurora large 32	85-95%
Aurora large 64	90-100%
Aurora xlarge 16	63-70%
Aurora xlarge 32	74-80%
Aurora xlarge 64	85-95%
Aurora 2xlarge 16	45-50%
Aurora 2xlarge 32	65-73%
Aurora 2xlarge 64	90-100%
Aurora 4xlarge 16	34-38%
Aurora 4xlarge 32	54-60%
Aurora 4xlarge 64	71-80%

Again, by observing the CPU utilization for this test, we notice that the db.r5.large instance is running low on CPU, with a utilization of 85-90 for the 16 threads test, and up to 90-100% for the 64 thread run. This is the reason the db.r5.large instance doesn't scale too well with thread counts. The larger instances doesn't seem to have any CPU limiting problems for the 16 and 32 thread runs, but some instances show high utilization for their 64 thread runs. These are the db.r5.xlarge and db.r5.2xlarge instances.

When compared to Amazon's results, the achieved write performance is very similar, or even slightly better with our new tests. The db.r5.large instance is reported by Amazon to achieve approximately 10000 queries per second, which is close to our 13394 qps. The db.r5.xlarge is reported to achieve results close to 20000 qps, while our tests show 25983 qps. For the db.r5.2xlarge, Amazon is able to achieve 40000 qps, with our test topping out on 43636. Lastly, for the r5.4xlarge, Amazon reports a result close to 75000 queries per second, which is comparatively close to our result at 71611 queries per second. This confirms the results reported by Amazon for Aurora's write performance, even in some cases doing slightly better. We see a similar trend to the read performance in that our performance tests obtain better results for the smaller instances, while Amazon's own tests outperform ours on the larger instances. This support our theory that Aurora might've used a larger thread count to obtain better results for the larger instances, on the sacrifice of CPU-bottlenecking the smaller ones.

8.1.2 Amazon RDS MySQL (Multi AZ)

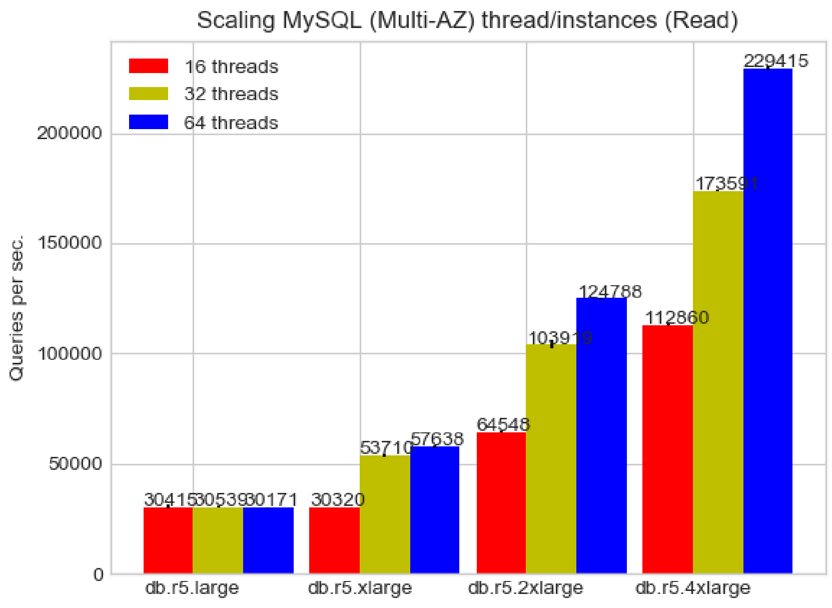


Figure 13: MySQL (Multi AZ) read performance

This figure shows the read performance for MySQL when run in the Multi AZ mode. For the db.r5.large instance, the tests results for all thread counts fall very close to each other. They are only separated by a margin of 368 queries per second. When we contrast this against the CPU load, it is clear that the db.r5.large instance is bound by the CPU for the 32 and 64 thread tests. The 32 and 64 thread test runs all utilize about 80-100% of the CPU, while the 16 thread test run is a little lower. For the db.r5.xlarge instance, both the test with 32 threads and the test with 64 threads achieve relatively close results, with the 64 threads test being better by 4000 queries per second. The CPU load for the 64 thread test run, however, is close to 90-100%, which shows that the 64 thread test run is limited by the CPU. With the db.r5.2xlarge and the db.r5.4xlarge, the differences become more pronounced, with the 64 threads test peaking out on the db.r5.4xlarge with close to 230000 queries per second. The scaling factor for the 16 threads test is ~ 3.71x, while it is up to ~ 7.6x for the 64 threads tests.

CPU utilization:

Instance	CPU load
MySQL large 16	80-100%
MySQL large 32	80-100%
MySQL large 64	80-100%
MySQL xlarge 16	47-54%
MySQL xlarge 32	84-90%
MySQL xlarge 64	90-100%
MySQL 2xlarge 16	48-55%
MySQL 2xlarge 32	70-77%
MySQL 2xlarge 64	90-100%
MySQL 4xlarge 16	38-43%
MySQL 4xlarge 32	64-72%
MySQL 4xlarge 64	82-92%

The results for the smaller instance sizes (db.r5.large and db.r5.xlarge) are lower than their Aurora counterparts. RDS MySQL achieve 30539 qps and 57638 qps for their respective instance sizes, while Aurora has 42746 qps and 89872 qps for the same instance sizes. A very interesting aspect however, as shown in the figure 13, is that the larger instances (db.r5.2xlarge and db.r5.4xlarge) perform better than their Aurora counterpart. The RDS MySQL db.r5.2xlarge performs with 124788 qps, while the Aurora equivalent has 109287 qps. Similarly, for the RDS MySQL db.r5.4xlarge performs with 229415 qps, while Aurora has 194295. These numbers show that in this case, RDS MySQL performs equal to, or better than Aurora. If contrasted with Amazon's reported Aurora performance, Aurora would still outperform RDS MySQL, with a claimed 180000 qps and over 300000 qps for the 2xlarge and 4xlarge respectively. One possible explanation to this phenomena is that Aurora first perform well with a higher thread count than 64 threads. Both results presented by Amazon and our tests indicate this. Either way, RDS MySQL follow Aurora relatively well, as instance size increases, with the larger instances performing better than Aurora's equivalent with the same configuration.

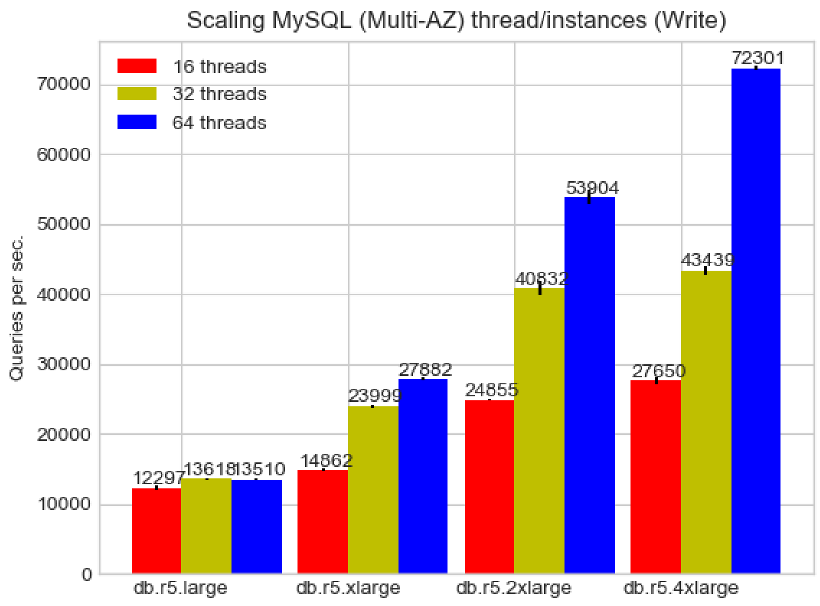


Figure 14: MySQL (Multi AZ) write performance

The figure for write performance shows many of the similar traits as seen in the previous figures. The db.r5.large instance show little difference between the various thread counts, with the differences being more easily visible for the larger instances. The scaling factor between the large and 4xlarge for the 16 threads tests are ~ 2.25x, while it is ~ 5.35x for the 64 threads tests.

CPU utilization:

Instance	CPU load
MySQL large 16	71-86%
MySQL large 32	85-94%
MySQL large 64	86-96%
MySQL xlarge 16	44-52%
MySQL xlarge 32	72-83%
MySQL xlarge 64	84-95%
MySQL 2xlarge 16	35-42%
MySQL 2xlarge 32	60-71%
MySQL 2xlarge 64	81-90%
MySQL 4xlarge 16	17-20%
MySQL 4xlarge 32	29-32%
MySQL 4xlarge 64	48-55%

Again, similar to read performance, RDS MySQL performs similar to, or sometimes better than Aurora with write performance as well. The db.r5.large metric is very alike to Aurora's, with both being on the 13000 qps mark. The next instances perform slightly better with RDS MySQL than with Aurora, with RDS MySQL having 27882 qps, 53904 qps, and 72301 qps for the db.r5.xlarge, db.r5.2xlarge, and db.r5.4xlarge respectively. For comparison, Aurora has 25983 qps, 43636 qps, and 71611 qps for the same instance sizes. This might again, as mentioned above, be down to thread count. As previously mentioned, Aurora shows better performance with a higher thread count, where most other databases start to fall off.

8.1.3 PolarDB

As previously mentioned, the PolarDB instance is a polar.mysql.x8.xlarge (8 cores 64GB RAM), which corresponds to the Amazon Aurora db.r5.2xlarge instance size. Since we didn't have access to more instances, the figures show how that specific instance scales with the number of threads.

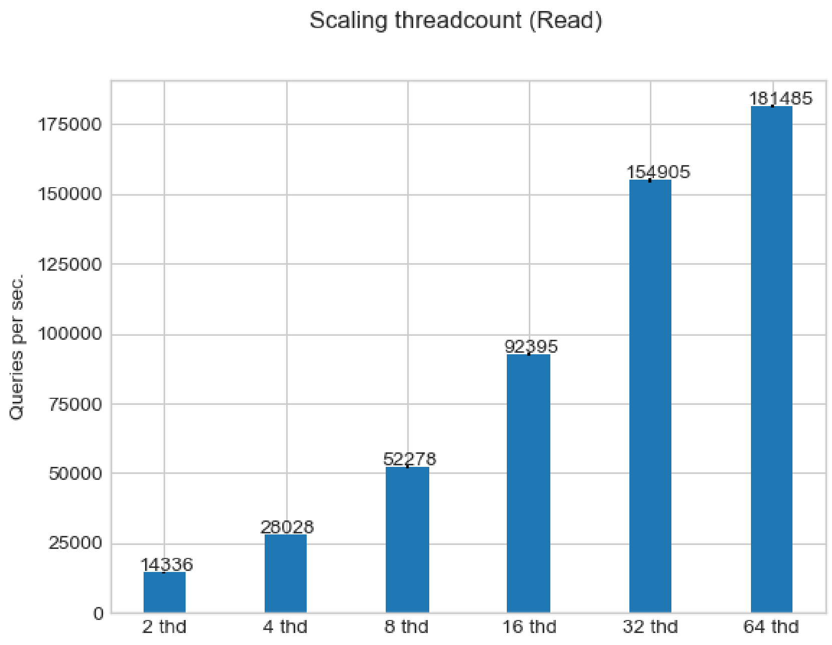


Figure 15: PolarDB read performance

With this smaller data set, PolarDB show exceptional performance. With just 2 threads, the PolarDB instance is able to perform 14336 queries per second. This rises close to linearly, so that with 64 threads, the system performs 181485 queries per second. It is worth mentioning that the scaling factor drops somewhat between 32 and 64 threads, but the achieved queries per second is still impressive. Our tests show equal, or slightly better results, than those presented [9] by Alibaba in 2018. Alibaba obtain about 130000+ qps for a 32 thread read only test, while we're able to achieve about 150000 qps under the same workload. It's worth noting that Alibaba's test didn't prewarm the data, which might explain the gap.

Similarly, as explained for the Aurora results, PolarDB will naturally experience some network latency. This latency comes as a consequence of the testing host and database instance are two separate instances and thus will need to communicate over the network. This could also happen when PolarDB is accessing its storage nodes, but in this case the effect would be smaller for PolarDB than with Aurora, since PolarDB makes use of RDMA technology. This explains why we see scaling beyond the amount of available hardware threads.

Compared to Amazon's similar instance sizes, PolarDB outperforms them rather heavily. With 64 threads, PolarDB achieved 181485 qps in read performance, while

MySQL tops out at 124788 qps, and Aurora clocking in at 109287 qps. For Aurora's part, this is, again, probably due to the low thread count.

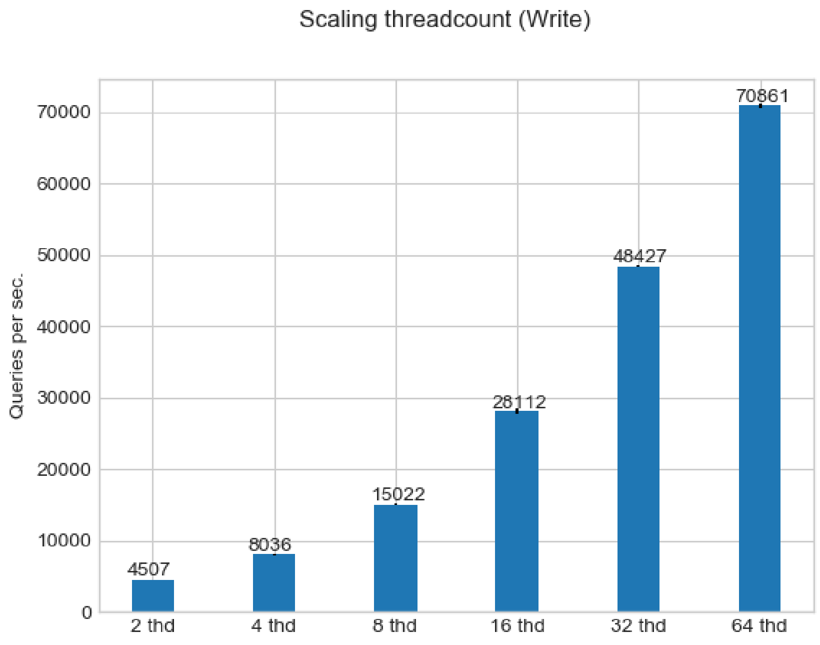


Figure 16: PolarDB write performance

Showing similar signs as with the read performance, the write performance also scales very well and close to linearly for the first four tests. The tests with 2 threads give a resultant 4507 queries per second, and with 64 threads, this is up to 70861. In this case, the scaling factor drops for the 32 threads to 64 threads jump, but the performance is still very good.

PolarDB also heavily outperform Aurora and MySQL for write performance. With 64 threads, PolarDB show writes up to 70861 qps, while Aurora and MySQL clock in at 43636 and 53904 qps, respectively.

8.1.4 Summary - 250 tables, 25000 records per table

We can identify a few trends from the results

- We have better read and write performance compared to Amazon's own tests for .large and .xlarge instances of Aurora, while lower for the .2xlarge and .4xlarge instances. This indicates larger instances benefit from larger thread counts.

- It may look like Amazon's tests were conducted with larger thread counts, giving better performance for the larger instances, but CPU-bottlenecking the smaller ones.
- Aurora performs better than MySQL for smaller instances, since they aren't that heavily CPU-bound, unlike MySQL.
- MySQL performs better than Aurora for .2xlarge and .4xlarge due to Aurora first being able to catch up for larger thread counts.
- PolarDB outperforms both Aurora and MySQL for similar thread and instance sizes.

8.2 Results with 250 tables, 2700000 records per table

The figures shown below are the result of tests with 250 tables with 2700000 records per table. This is equal to a database size that is close to 150GB. The dataset is larger than the allocated memory of the tested instances. The largest instance size tested, the db.r5.4xlarge, has 128GB of memory. These results will therefore show how the databases handle out-of-cache database sizes. As mentioned previously, the tests only show how PolarDB scaled with various thread counts, as we only had access to one instance size.

8.2.1 Amazon Aurora

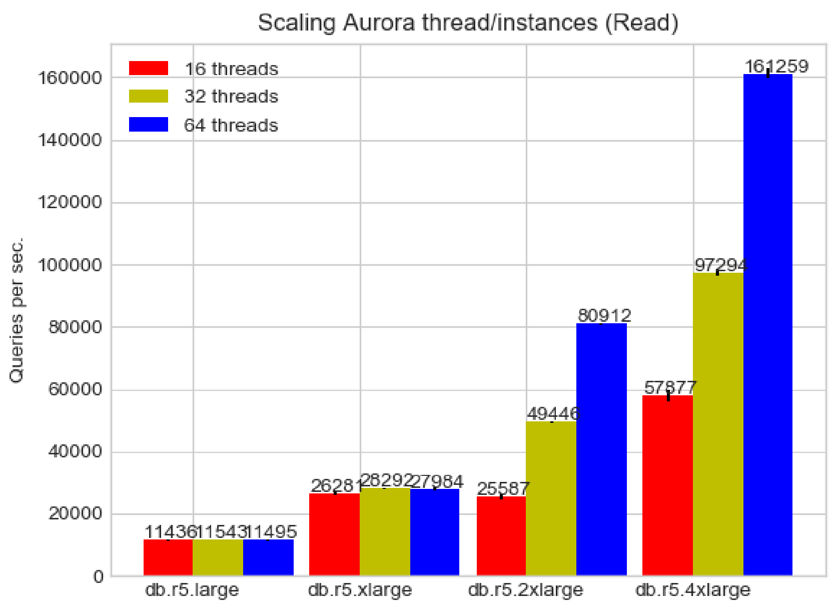


Figure 17: Aurora read performance

The results seen in figure 17 show how Aurora scales under read workload with different thread counts and instance sizes. An interesting point is that larger thread counts don't seem to affect the smaller instances sizes (db.r5.large and db.r5.xlarge), as we will explain in the next paragraph. The db.r5.large tops out at 11543 queries per second, while the db.r5.xlarge tops out at 28292 queries per second. The larger instances show better scaling with thread counts, however. The db.r5.2xlarge starts at 25587 qps for 16 threads, but climb to 80912 for 64 threads. The db.r5.4xlarge starts at 57877 qps for 16 threads, and climb to 161259 threads for for 64 threads. The read performance scales well across instance sizes, with a scale factor of 14x from the smallest to the largest instance size. When compared to the in-cache dataset, it's clear that there has been a decrease in performance, as expected. The difference is although relatively minor. For the db.r5.4xlarge instance, a decrease of 17%, with the other instance sizes following a similar trend.

The read tests for the db.r5.large clearly topped out its CPU utilization for all thread counts. The CloudWatch statistics showed the average laying somewhere in between

90 and 100 for the entire test run. This explains why the graphs in figure 17 are so even for the db.r5.large instance; the load is clearly CPU bound.

Similarly, the db.r5.xlarge showed to be peaking in CPU utilization as well. During the read run, for all threads, the utilization varied between 90-100%. This indicates the load is CPU bound, and is thus the cause for the flat graphs in fig. 17.

The CPU utilization for the db.r5.2xlarge is with 16 threads hovering around 30-40%. With 32 threads, the utilization is around 60-70%. For 64 threads, the utilization hovers between 80-90%, and sometimes spiking out to close to 100%. Throughout the run, the db.r5.2xlarge has reasonable freeable memory available. This might indicate that the db.r5.2xlarge is well capable of taking load up to 64 threads, where the utilization appears to become close to CPU bound.

For the db.r5.4xlarge instance, CPU utilization is generally low under this load. The 16 thread read test uses approx. 30 %, while the 32 thread increase to 50%, before reaching 80% for the 64 thread test run. The freeable memory is also overall high. This indicates that Aurora possibly would perform even better for higher thread counts.

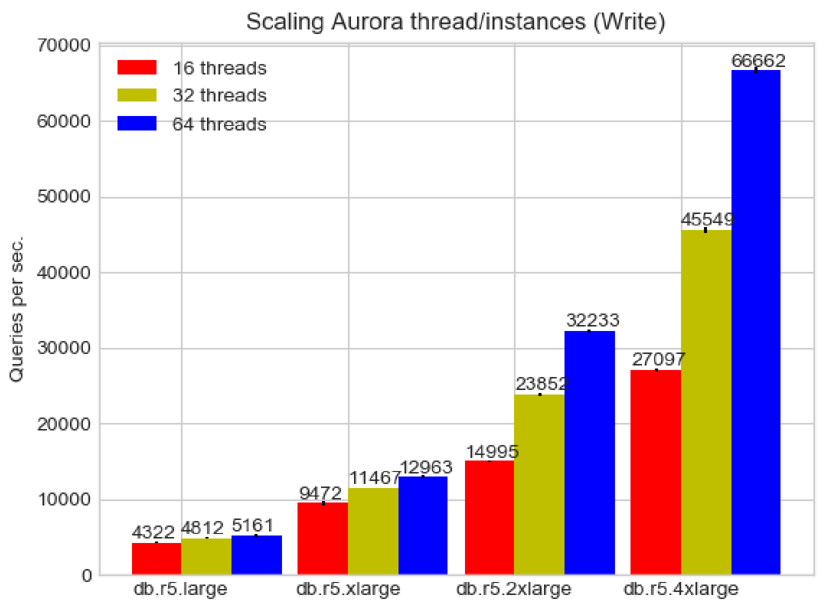


Figure 18: Aurora write performance

Figure 18 shows how Aurora scales writes with varying thread counts and instance sizes. The amount of threads have little impact for the smaller instances, such as the db.r5.large and db.r5.xlarge. For the larger instances however, the distinction between the thread count is clearly visible. The db.r5.large tops out at 5161 qps, while the db.r5.xlarge is able to perform close to 13000 qps. The db.r5.2xlarge and db.r5.4xlarge show best performance with the higher thread counts, The db.r5.2xlarge performs 32233 qps with 64 threads, and the db.r5.4xlarge is able to perform 66662 qps with 64 threads. To contrast this to the small in-cache dataset, we also see a slight performance decrease. The db.r5.4xlarge sees a 7% decrease, with similar statistics for the other instance sizes.

The db.r5.large instance showed a slight decrease in CPU utilization for the write tests. With 16 threads, it lies somewhere between 80-90% in average. This rises to 85-90 for the 32 thread count. The 64 thread tests are bouncing between 93-100%, so there's a clear CPU bound impact here as well.

The CPU utilization go slightly down for the db.r5.xlarge write performance, averaging around 80% for the 16 thread run. This is further increased to a bit over 90% for the 32 and 64 thread count test runs. This shows the instance again is CPU bound.

During the write tests, the db.r5.2xlarge averaged around 70% CPU utilization for the 16 thread test run, but increased to 80-90% for the following 32 and 64 threads. However, it didn't peak out to 100, so Aurora was probably just barely capable of sustaining this load.

The write tests showed the db.r5.4xlarge averaging around 40% for the 16 thread count, while rising to 55-60% for the 32 thread count. The 4xlarge instance topped out at 75-80% for the 64 thread test run, with the freeable memory correspondingly relatively high. This again indicates that a higher thread count might achieve even slightly better results.

This is in line with Amazon's own findings. They conducted a write test, which showed a decrease in performance with a large, out-of-cache, database size. The numbers we've achieved back this claim up, with a 17% decrease in read, and only 7% decrease for write loads.

8.2.2 Amazon RDS MySQL (Multi AZ)

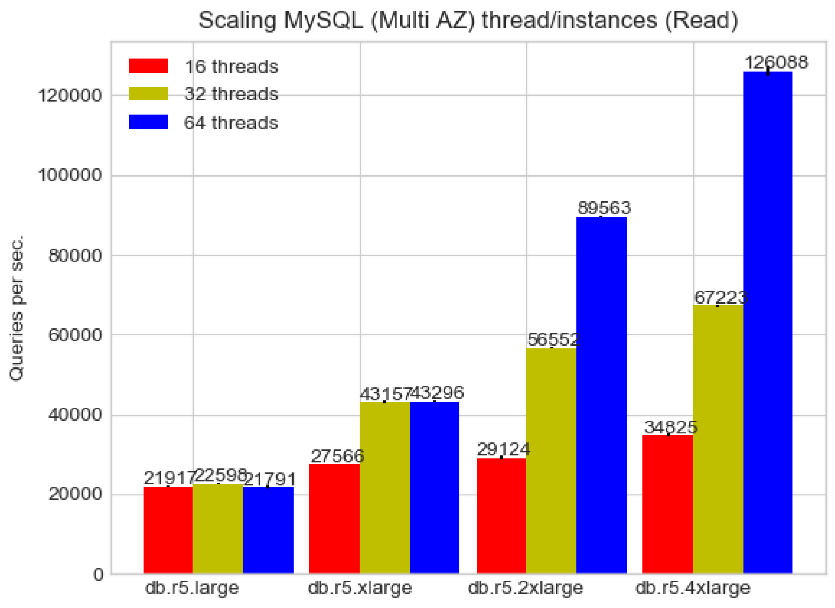


Figure 19: MySQL (Multi AZ) read performance

MySQL read performance is relatively good. As seen by figure 19, MySQL scales close to linear with instance sizes, but thread count doesn't seem to have too big an impact on the results for the smaller instances. The larger instances, namely db.r5.2xlarge and db.r5.4xlarge, however scale well with increasing thread counts. The read performance for the smaller instances actually outperform Aurora slightly. Aurora, however, catch up and surpasses MySQL for the db.r5.4xlarge instance, which it outperforms by more than 27%.

The db.r5.large instance had relatively high CPU utilization for the read tests. The 16 thread, 32 thread, and 64 thread tests all measured around 80-90% CPU utilization. This is reflected in the results shown in the graph, where the thread size doesn't make any significant impact on the results. This shows the db.r5.large instance performance is CPU bound.

The db.r5.xlarge starts off with a CPU utilization of 63-65% for the 16 thread test, but quickly rise to 80-100% for both the 32 and 64 thread tests. The graph reflects this, as both tests give rather even results at 43000 qps.

For the db.r5.2xlarge read tests, the CPU utilization increases with the thread count, as expected. With 16 threads, it is at approx. 30% utilization. This increases to 55 % for the 32 threads, and then further increasing to 80-95% for the 64 threads test run. This is shown in the graphs, where the db.r5.2xlarge instance see increased performance with higher thread counts.

The db.r5.4xlarge has relatively low CPU utilization, starting at 15-20 % for the 16 thread tests. The 32 thread tests show about 25-30%, while the 64 thread consumes approx. 50-60%. The tests indicates that perhaps an even greater thread count would be needed to stress the systems.

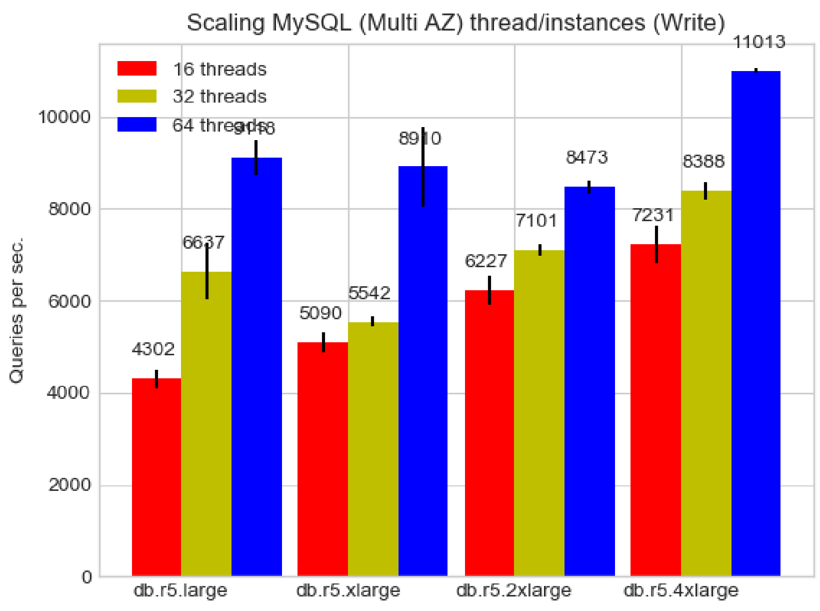


Figure 20: MySQL (Multi AZ) write performance

MySQL's write performance is however somewhat weird, as seen in figure 20. The three largest instance types seem to have very similar performance characteristics, where they perform relatively equal regardless of instance size. The db.r5.2xlarge performs slightly worse than the db.r5.xlarge instance. The db.r5.large in turn outperform both of them. This is a phenomena also seen in Amazon's own benchmarks. Compared to Aurora, the write performance is abysmal. Even the largest instances are only close to Aurora's smaller instances, db.r5.large and db.r5.xlarge.

When looking at the CPU utilization for the db.r5.large instance, it starts off at around 50% for the 16 thread write test. This increases to 60% for the 32 thread test, and further to 80-85% for the 64 threads test run. When looking at the performance insight statistics gathered, we see three events that cause long waits. This is CPU, a "io/table/sql/handler" event, as well as "/io/file/innodb/innodb_data_file". The "io/table/sql/handler" event indicates that the storage engine is processing an I/O request against a table, and the "io/file/innodb/innodb_data_file" is for accessing accessing the InnoDB data files. The performance insights show that the I/O events are the ones generating the longest waits, and we can therefore deduce that this test is mainly limited by I/O. This trend gets worse as the thread count increases.

The CPU percentage for the db.r5.xlarge instance hovers around 30% for both the 16 and 32 thread tests, before climbing slightly to 30-40% for the 64 thread tests. When looking at the performance insights for this instance, we see a similar trend, with comparatively heavy I/O which generate long waits.

The db.r5.2xlarge CPU utilization is generally very low throughout these tests, with a utilization averaging around 15-22%, which implies that the load is I/O bound. The CPU utilization increases slightly for the 64 threads, but not very significantly. Similarly, the I/O events dominate the performance insights for this instance as well.

Identical to the db.r5.2xlarge, the CPU utilization is low for the db.r5.4xlarge tests as well. For the db.r5.4xlarge, the lies between 7-12% and increases slightly for the larger thread counts. This is the same as seen with the previous instances. The only instance that see any major increase is the db.r5.4xlarge. The performance insights show similarly high waits for I/O.

8.2.3 PolarDB

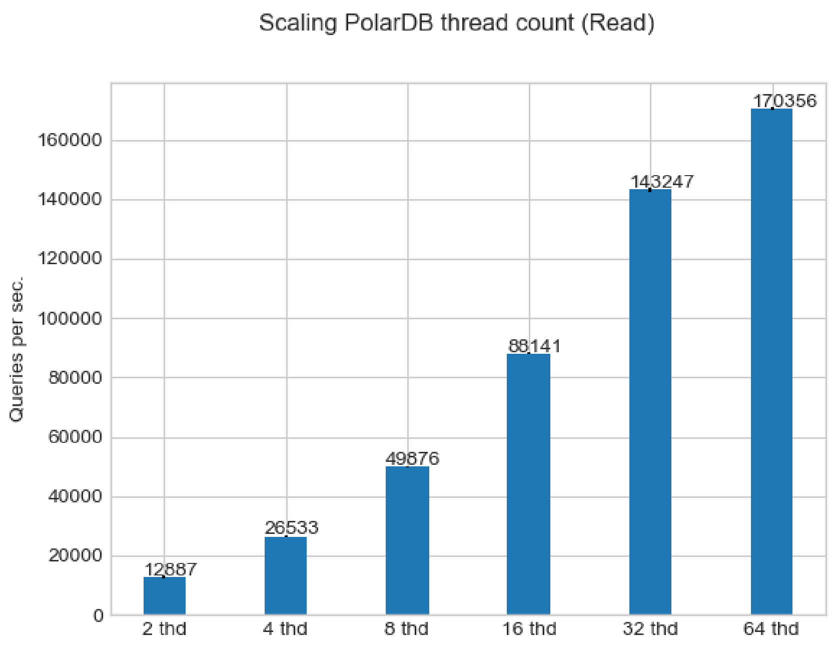


Figure 21: PolarDB read performance

As seen in figure 21, PolarDB's read performance scales close to linearly for increasing thread counts. While starting at 12887 qps with 2 threads, it quickly increase to 88141 qps with 16 threads, and tops out at 170356 qps for 64 threads. Compared to how Aurora and RDS MySQL's db.r5.2xlarge instance, PolarDB heavily outperforms the systems with close to 40%. This is very strong performance for an instance that is the equivalent of Aurora's db.r5.2xlarge instance class.

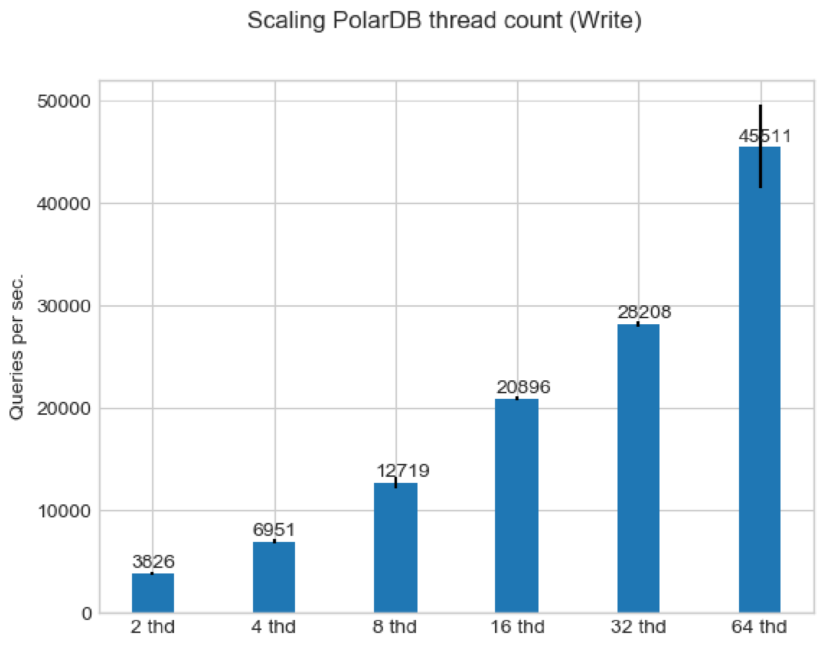


Figure 22: PolarDB write performance

PolarDB's write performance is similarly strong, as seen in fig. 22. The instance starts at 3826 qps for 2 threads, which is increased to 20896 qps by 16 threads, before topping out at 45511 qps with 64 threads. It is worth mentioning that there's some uncertainty in the measurement for the 64 threads test run, indicated by the error bar. However, compared to the other systems, PolarDB performs very well, but the difference to the in-cache dataset is lesser than with the read tests. Compared to Aurora, it performs approx. 40% better.

8.2.4 Summary - 250 tables, 2700000 records per table

We can identify a few trends from the results

- Overall a decrease in performance compared to in-cache dataset. Aurora read performance down with 17%/write performance down with 7% for the .4xlarge instance, with other instance sizes following suit.
- For read and write tests, both Aurora's .large and .xlarge see high CPU load, and are CPU-bottlenecked. .2xlarge manages well up to 64 threads where it is close to becoming CPU bound. .4xlarge doesn't see these problems, and

could possible perform even better with higher thread counts. A similar trend is observed for MySQL read performance.

- MySQL read performance is better than Aurora up to .4xlarge, where Aurora catches up and outperforms MySQL with more than 27%.
- MySQL write performance is very poor, the larger instances only come close to Aurora's .large and .xlarge.
- The MySQL write tests are clearly I/O-bound, as indicated by the performance insights metrics and CPU load.
- PolarDB shows very strong performance, outmatching both RDS MySQL and Aurora for read and write performance by close to 40% on equivalent hardware.

8.3 Comparison

The tests and figures presented above give an image of how the systems perform when scaling thread and instance sizes. The following section focuses on how the various systems compare to each other. These will be based on both the tests presented above, and additional head-to-head tests which were carried out. These tests include

- How RDS MySQL and Aurora compare at high thread counts
- How RDS MySQL (both with and without Multi AZ), Aurora, and PolarDB compare under similar circumstances
- Which impact Multi AZ has on RDS MySQL performance

8.3.1 How RDS MySQL (both with and without Multi AZ), Aurora, and PolarDB compare under similar circumstances

Comparison with 250 tables and 25000 records per table data set (1.6GB)
As mentioned previously, Alibaba only provided access to a single PolarDB instance which should be comparable to db.r5.2xlarge. The tests presented in section 8.1.3 thus only present how said PolarDB instance scales with various thread counts. For a fair comparison to the other systems, we therefore need to compare tests with the same number of threads and tests done on the same data set. The figures below show how MySQL (both in Multi AZ and regular mode), Aurora, and PolarDB compare when tests were run with 64 threads on the 250 tables/25000 records per table dataset. Since the PolarDB instance type is equal to AWS's db.r5.2xlarge, the AWS instances

used will naturally be of this type. The figures present both the read and write performance.

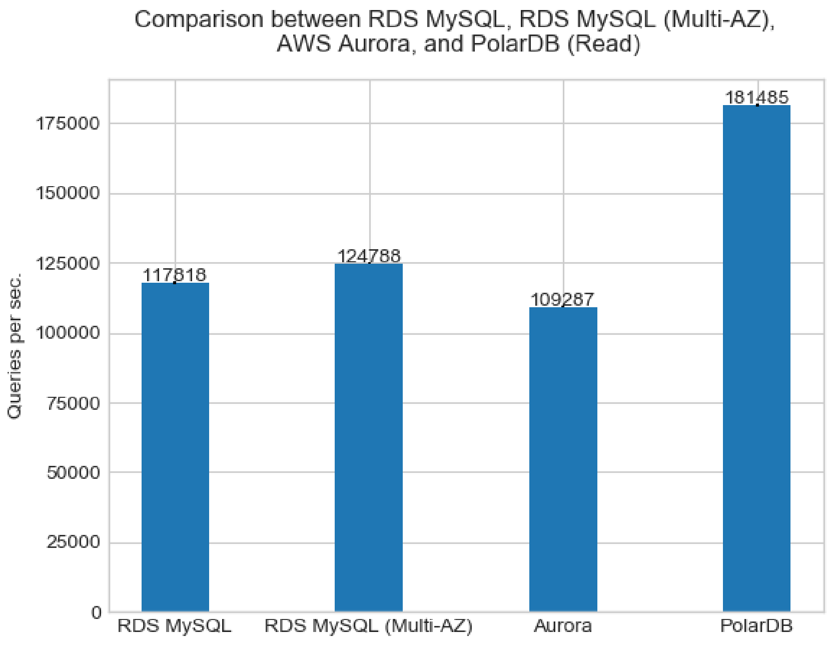


Figure 23: Comparison read performance

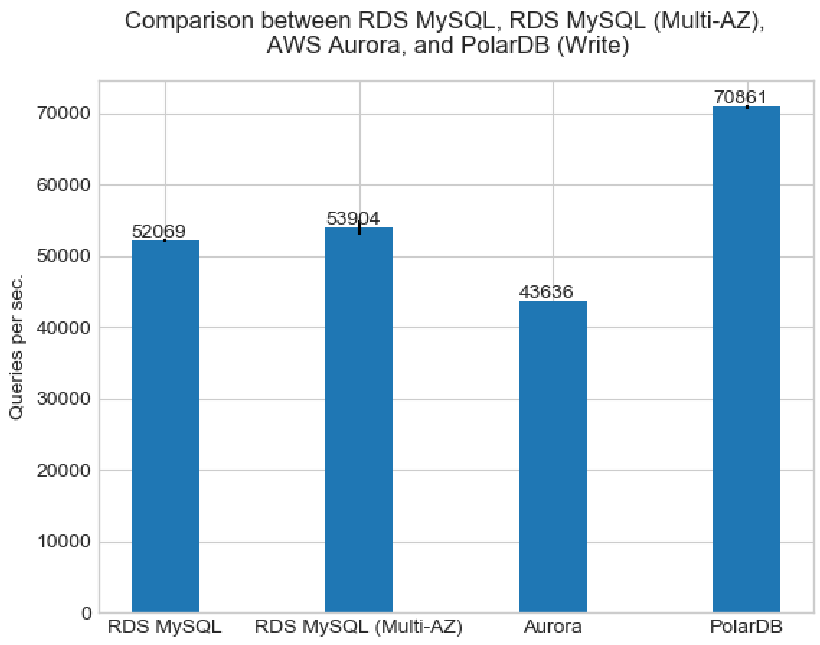


Figure 24: Comparison write performance

Aurora's low read and write performance is most likely a consequence of the low thread count, and relatively small instance size. As mentioned in section 8.1.1, Aurora must wait longer per thread due to network latencies, which facilitate a larger thread count. And indeed, Aurora only seems to perform close to AWS's stated results when run with a higher thread count. Additionally, there isn't any significant impact to running RDS MySQL with Multi-AZ mode enabled. This is also in line with Amazon's findings.

As mentioned in section 4.2, Amazon showed that a Multi-AZ deployment would be penalized by a 2-5ms commit latency, which for most workloads would be of minimal consequence. We see a slight decrease in write performance for the Multi-AZ deployment, probably is down to the need to write to an additional volume in a secondary AZ. This is however, not a severe penalty in performance, with only a 3% decrease in performance.

8.3.2 How Aurora and PolarDB compares with 250 tables and 200000 records per table data set (10.8GB)

The tests shown in figures 25 and 26 are conducted on a dataset of 250 tables and 200000 records per table. These tests were only carried out on Aurora's db.r5.2xlarge and the polar.mysql.x8.xlarge with 64 threads. This was done to give additional confirmation to the results on a somewhat larger database than the 250 tables + 25000 records per table dataset.

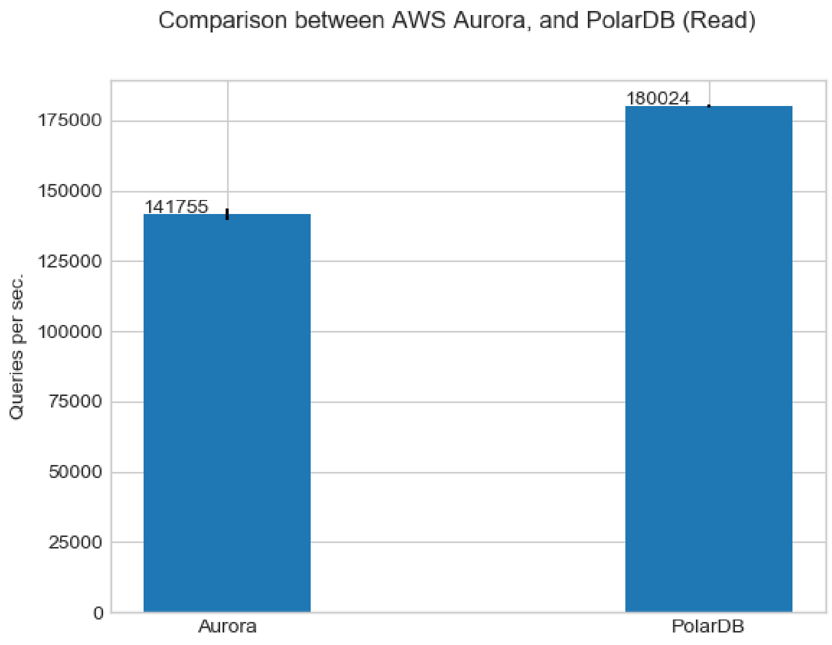


Figure 25: Comparison read performance

The read performance shows that Aurora performs about 141755 qps, while the PolarDB instance shows 180024 qps, which is around 27% better than the Aurora instance. When compared to the 250 tables + 25000 records per table dataset, we see that PolarDB's performance is nearly unaltered, whereas the Aurora performance is up from 109287 to 141755, which is an increase at around 29%. This increase is likely due to the 250 table 25000 records per table testrun had a relatively low CPU load on the db.r5.2xlarge, and thus had potential to perform even better, as seen in the results.

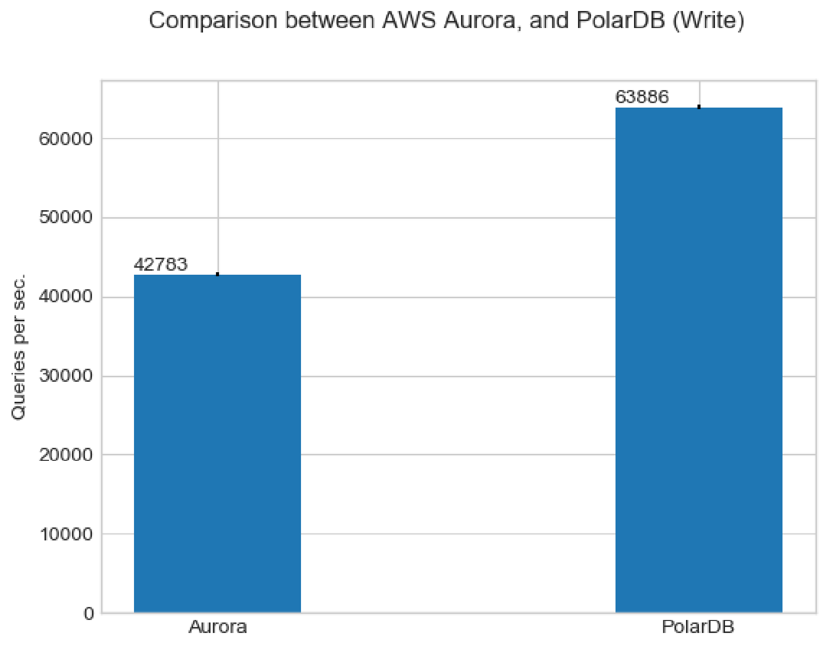


Figure 26: Comparison write performance

The write results are very identical to those of the previous 250 table, 25000 records per table test. The PolarDB instance see a drop from 70861 qps to 63886 qps, while the Aurora instance drops from 43636 qps to 42783. This is virtually the same results as in the previous test, and confirm the trend of PolarDB outperforming Aurora for similar hardware on 64 threads.

8.3.3 How RDS MySQL (both with and without Multi AZ), Aurora, and PolarDB compare with high thread counts

The figures shown below represent test runs with 250 tables and 25000 records per table run on db.r5.2xlarge instances with a 500 thread count. The goal with these tests were to see how a higher thread count affected the systems, especially Aurora. This is because, as indicated in the previous sections, a higher thread count would negate some of the network latencies experienced for AWS.

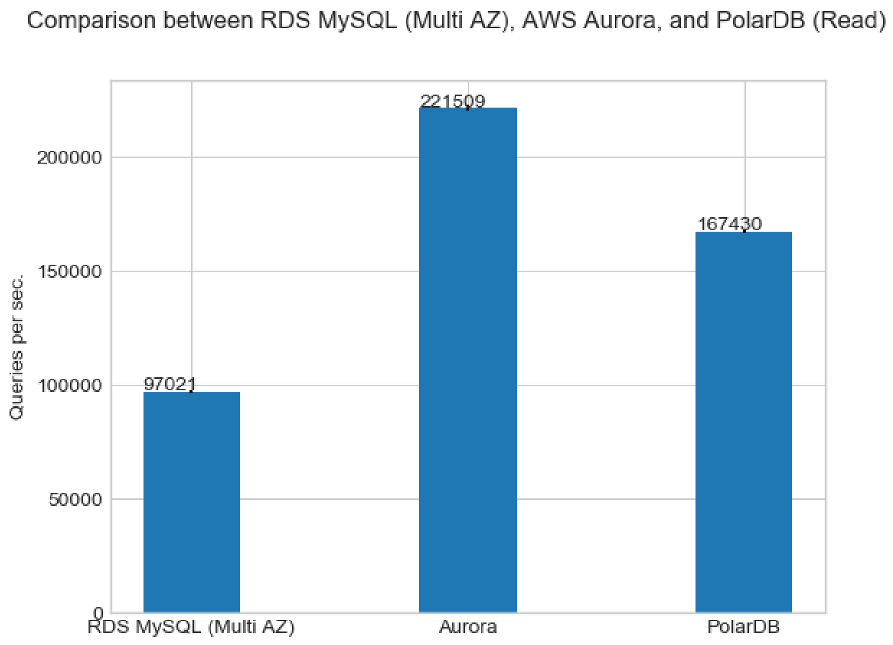


Figure 27: Comparison read performance

In this test, we see Aurora performing best of all three systems, with a read performance of 221509 qps, which is significantly higher than PolarDB's 167430 qps and RDS MySQL's 97021 qps. Aurora see an increase of 102% compared to the 64 thread result. The PolarDB instance also see an decrease in performance, from approx. 184000 qps to 167430 qps, which correspond to a 7% decrease. The RDS MySQL, however, drop sharply with this high thread count. We see a decrease from 124788 qps to 97021 qps, which is a 22% decrease. The use of a thread pool would in this case probably limit RDS MySQL's reduction in performance for low thread counts. These results is as expected, as the larger thread count allow Aurora to better utilize the available resources.

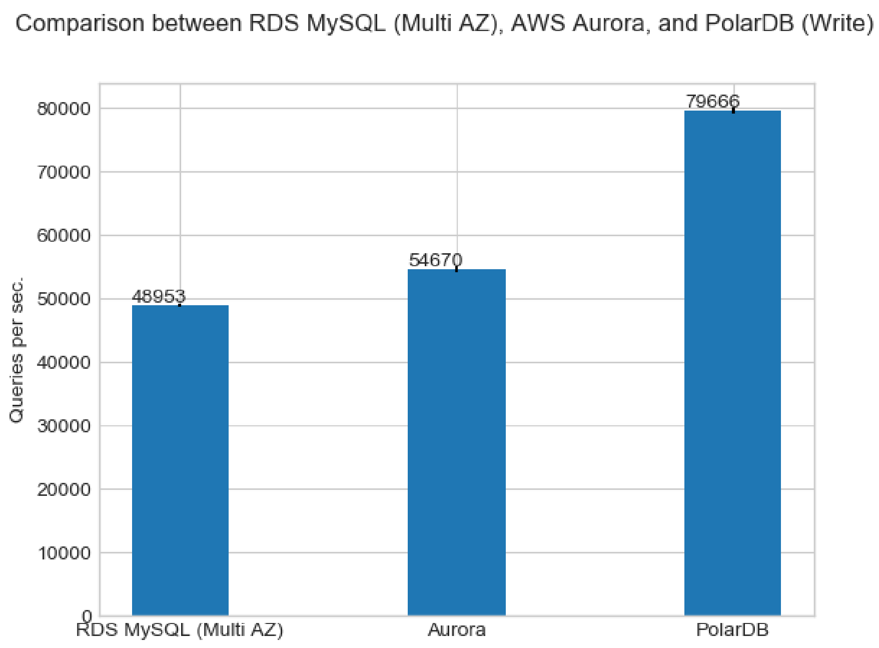


Figure 28: Comparison write performance

When comparing the write results, we see that PolarDB actually tops out of the three systems. PolarDB sees an increase from 70861 qps to 79666 qps, an increase of 12%. Similarly, Aurora also see an increase in performance, from 43636 qps to 54670 qps, which is a 25% increase. RDS MySQL, however, drops from 53904 to 48953, which is close to a 9% decrease. These results confirm that there is an increase in performance for Aurora when using a higher thread count. There is a possibility that even better write performance for Aurora could be achieved with even greater thread numbers, as the CPU utilization was around 70-80%.

9 Discussion

As previously mentioned, distributed database systems have evolved significantly in the face of cloud and the new workloads experienced by the rapid growth of the Internet. Traditional systems haven't been able to follow and adapt to this development, and we've seen the introduction of cloud based distributed database systems provided from major companies such as Amazon and Alibaba. These solutions are heavily optimized for the infrastructure the system runs on top of. Since cloud providers control the underlying infrastructure, they can design the database systems to both leverage and work optimally for exactly that infrastructure. By leveraging the infrastructure, the systems can introduce major design changes compared to previous solutions, both with regards to the compute and storage components in the system. Leveraging existing infrastructure also means that the cloud providers can reduce development time and maintenance costs. Additionally, emerging technologies have gradually been adopted and implemented in these systems, and may increase the performance of these systems drastically.

This discussion will focus on an architectural review of each databases system, as well as how they perform in benchmark test for various workloads. We will look at how the decoupling of compute and storage is done. Where does the split happen? What tasks are offloaded to the storage components? Furthermore, we will look at how data is stored on the storage nodes. How is consensus maintained, what measures are taken to provide redundancy of data? Additionally, how do the systems recover from a crash scenario? In the following sections we'll see how these approaches differ and where they share similarities. The discussion will be based around the previously mentioned systems, namely Amazon Aurora and Alibaba PolarDB.

9.1 Amazon Aurora

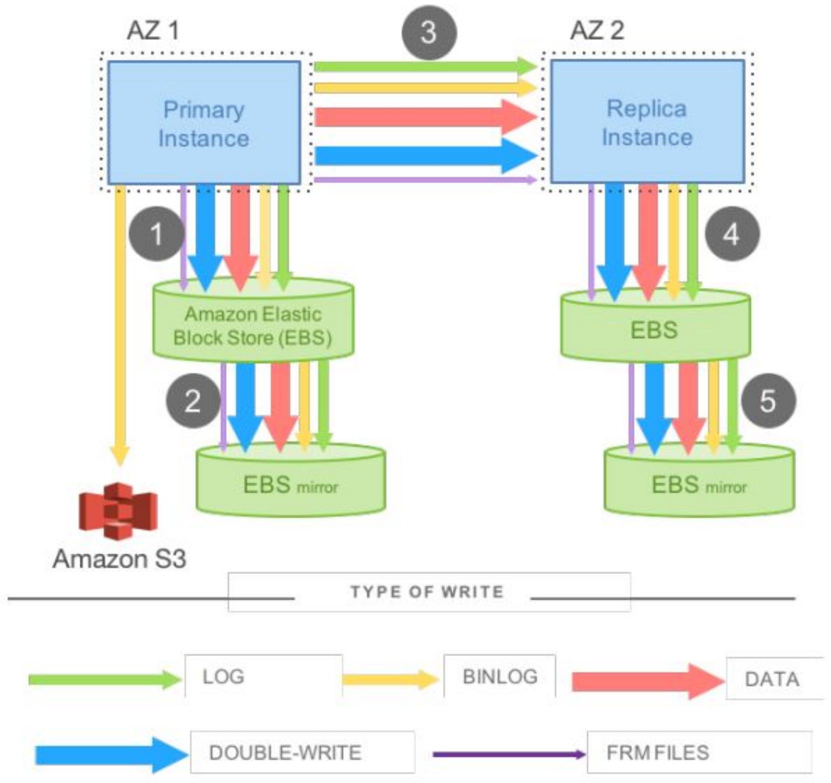


Figure 29: Network IO for mirrored MySQL. Source: [2]

Amazon's motivation for splitting the compute and storage components is that network bandwidth is a limited resource, and by offloading multiple tasks to the storage component, Aurora is able to reduce the write amplification problem that usually arise in traditional database settings. This problem is illustrated by figure 29, where we see a traditional setup with a primary instance and an active standby replica deployed on Amazon RDS. This setup results in many synchronous and sequential steps, which would impact both latency and jitter. This is clearly undesirable. By offloading tasks such as redo log processing, crash recovery, and backup/restore, Amazon is able to greatly reduce this effect[1]. In Aurora, the only writes that cross the network are redo log records, as seen in figure 30. The local log processor continuously materialize database pages in the background, which removes the need for materializing pages from scratch on demand every time[1]. This severely reduces network load. The storage layer is able to scale independently, without significantly impacting the database engine's write throughput. Additional benefits of moving the log processing to the storage component is reduced jitter by checkpointing and other background activities, as well as faster crash recovery times[1].

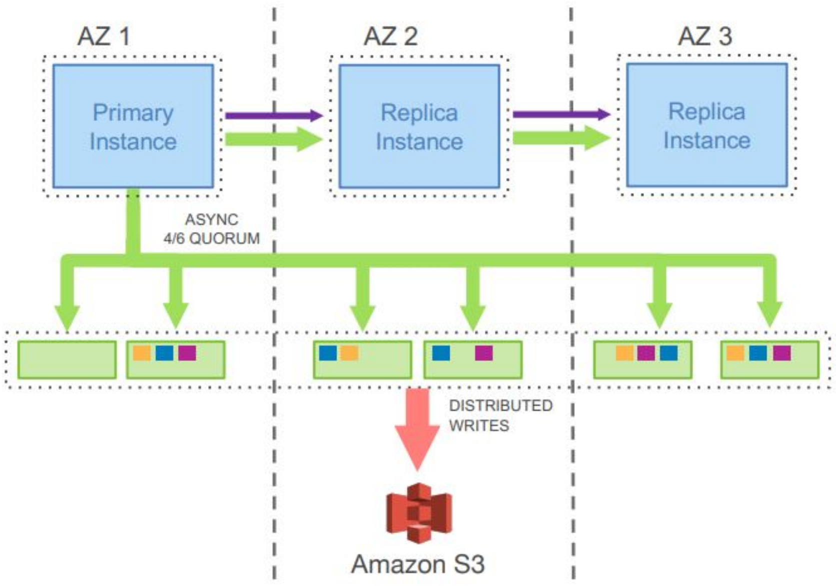


Figure 30: Network IO for Aurora. Source: [2]

We see this in both our tests, and the tests presented by Amazon. Amazon compared how a mirrored MySQL configuration spread across multiple AZs compared to Aurora with replicas across multiple AZs. The tests consisted of WO transactions with a 100GB dataset [1]. When run on similar hardware, Aurora was able to sustain 35 times more transactions over a 30-minute test period, as well as 7.7 times less I/O per transaction [1]. Our out-of-cache dataset with 250 tables and 2700000 records per table, as presented in section 8.2, show that Aurora far excel beyond RDS MySQL in write performance. MySQL write performance is very poor, the larger instances only come close to Aurora's db.r5.large and db.r5.xlarge. The MySQL write tests are clearly I/O-bound, as indicated by the performance insights metrics and CPU load. This confirms the effectiveness of the approach taken by Aurora.

Aurora has recognized the problem of constant background noise of failures when running large data centers. They have therefore taken a relatively cautious approach when it comes to data resiliency. Aurora uses a quorum model for replicas, as described in section 3.9.2. However, most systems that use the quorum model only have 3 data copies with a quorum of 2/3 both for read and write. Amazon argue that this in practice provides too little resiliency to failures. They have instead opted for a cautious approach with 6 data copies, where 3 AZ's store 2 copies of each data item. With this approach, Aurora can withstand the loss of three nodes without losing data, or two nodes and still maintain write quorum.

Having this great resilience give additional benefits to maintenance operations. This allows Amazon to perform actions such as OS and security patching and other software upgrades. Amazon does this by updating one Availability Zone at a time, and also ensuring only one storage segment is updated simultaneously [1]. This will, to Aurora, look like short failures, which Aurora can resiliently handle.

Aurora has found a novel way to maintain consensus in a distributed system. Instead of opting for the traditional distributed consensus algorithms, such as 2PC, Paxos, and Raft, Aurora uses local transient state variables, as described in section 5. Amazon motivate this choice in that the traditional systems are too expensive and result in additional network overhead[2]. Further, even though they may scale well, they more than often result in worse performance and cost with potentially high latency. However, Amazon argues that their approach gives improved performance and variability at a lower cost compared to other similar systems[1].

Aurora optimizes read performance by avoiding read quorums where possible [2]. The latest version of a data block can either be found in cache or by looking at which segments have the latest durable version of a data block, which the database instance keeps track of. In the latter case, this allows Aurora to request it directly from that segment [2]. By avoiding quorum reads in most circumstances, network overhead can be greatly reduced.

Our tests indicate that while network latency is reduced, a small but significant amount of time will be lost as the query and response propagates through the network. As a consequence, a larger count of database connections can be facilitated than there are hardware threads. For our read tests, we see clear indications that a larger number of threads would indeed be beneficial. We have also confirmed that Aurora will perform astoundingly 102% better when run with 500 threads compared to 64 threads, as described in section 8.3.3.

Additionally we see for both the in-cache and out-of-cache dataset that Aurora performs better than MySQL for smaller instances with similar numbers of threads, since they aren't that heavily CPU-bound, unlike MySQL.

For the in-cache dataset, our 64 thread tests show that MySQL performs better than Aurora for .2xlarge and .4xlarge, due to Aurora first being able to catch up for larger thread counts. With the out-of-cache dataset, however, MySQL read performance is better than Aurora up to .4xlarge, where Aurora catches up and outperforms MySQL with more than 27%.

Similarly, Aurora is able to avoid distributed consensus for writes and commits by managing consistency points in the database. In Aurora, storage nodes do not have a vote in whether or not to accept a write. If a node misses a write, they will gossip with other nodes in the PG to catch up [1]. As shown in section 5, this can also be accomplished for commit processing, even without stalling the worker thread in order to wait for the data modified by the transaction has been made durable. The worker thread will hand off the transaction to a commit queue and return to the task queue to find new requests. A dedicated commit thread will scan the commit queue and send acknowledgements once the data is durable [1].

As mentioned for the read results, Aurora is able to facilitate a large number of threads due to network latencies, but this is especially of significance for writes, where the database must wait for a quorum to be formed and the changes made durable before being able to respond. We see this holds true for our write tests as well. Section 8.3.3 show that Aurora experience a 25% increase in performance when compared to the same run with 64 threads. Also, the 500 thread test run we see that RDS MySQL start to struggle with the high thread count, decreasing with 9%. It is worth noting that Aurora in this case possibly could've achieved even better results with a higher number of threads, as the CPU utilization was around 70-80%.

For the in-cache data sets we see a similar trend for for the write performance as seen with the read performance. For smaller thread counts, RDS MySQL is able to perform equal to or better than Aurora. For the out-of-cache dataset, we see Aurora performing at a whole different level than its RDS MySQL counterpart. Aurora only see a decrease of around 7% when comparing the in-cache to the out-of-cache test runs. The smaller Aurora instances even perform better than the larger RDS MySQL instances. When looking at the data provided by Amazon's Performance Insights feature, we see that RDS MySQL is heavily I/O-bound for nearly all the tests, while Aurora isn't. This confirms Amazon's statements with regards to the effectiveness of their approach.

Consistency must be reestablished on crash recovery. On startup, the database must recompute the PGCL and VCL variables before proceeding [2]. For this to be possible, the database instance must be able to reach read quorum for each protection group the volume consists of. The recovery process will also truncate any uncompleted partial writes above the newly recomputed Volume Complete LSN. Additionally, redo records after crash recovery will be allocated LSNs over this truncation range. If the database instance is unable to reach write quorum, repair is initiated to rebuild the failed segments. Furthermore, undo of previously active transactions can then be performed. This can happen after the database has built the list of the in-flight transactions from the undo segments, and the undo can be done in parallel with other user activity [2]. Amazon argues the price of reestablishing consistency on crash recovery is a trade-off worth making because crashes are a relatively rare event. Recoveries in Aurora can be done very quickly. Amazon states Aurora can recover in about 10 sec from a crash where the database processed 100 000 write requests per second [1].

The Aurora engine performs well, both in theory and in practice. The results seen in this report slightly less, but still relatively close to the performance presented by Amazon. It is however worth mentioning that it is obvious that the comparison presented in Amazon's 2017 SIGMOD paper is biased to put Aurora in a favourable light. As we've seen in our tests, RDS MySQL is capable of rivalling Aurora for both read and write performance. The main differentiating factor is with high thread counts. Aurora performs better with higher thread counts than most other systems.

A point worth noting is that Aurora is significantly cheaper than the comparative RDS MySQL instances, so while RDS MySQL is capable of delivering equal performance, the price point is also much higher. Amazon claims Aurora costs the 1/10th of other similar database offerings.

Even though Aurora performs well, there's still work left to be done for Aurora. Since Aurora heavily relies on low latency and high throughput networking, Aurora has typically been used in a single region deployments. This is because each AZ in a region is interconnected with fast high capacity network links. Amazon has, however, added support for replicating Amazon Aurora MySQL DB Clusters across up to five AWS regions [15]. It is somewhat limited, as only cross-region read replicas are supported. You can, however, have multiple read replicas per region [15]. This can be used for scaling reads in an effective manner, and additionally increase disaster recovery capabilities. This comes at the cost of greatly increased replica lag, as well as additional costs charged by Amazon for data traffic leaving a region [15].

Similarly, currently Aurora only support single-master. There has been work done to allow for multi-master deployments in Aurora, This is aimed at being able to horizontally scale writes, as well as improve HA in case of failures [12]. The Multi-master functionality entered private preview in 2017, but hasn't been made available to the general public as of the writing of this article. Little is known of the actual implementation; no research papers or blog posts have so far outlined this functionality.

9.2 Alibaba PolarDB

Alibaba's motivation for decoupling the compute and storage components is mainly to allow for a more flexible architecture. The stated goals are to gain a single storage pool where the storage layer could be independently scaled, with the removal of local persistent state of from the compute node, as well as allowing nodes to utilize different hardware[7]. Most traditional distributed file systems, such as HDFS and Ceph, don't make use of emerging technology such as NVMe and RDMA. This is technology that has seen increased use and deployment in datacenters. Alibaba also points out that most blockstorage solutions have lacking protection mechanisms for disk failure, as well as not utilizing emerging technology such as RDMA and NVMe, and thus being harder to scale effectively [7].

In light of the drawbacks of traditional distributed file systems, Alibaba decided to design and implement their own system, PolarFS. PolarFS aims at solving the issues of traditional FS's by levering these emerging technologies to obtain low latency, high throughput, and high availability. PolarDB utilizes PolarFS as a shared storage system, where i.e. a primary and a replica would access the same underlying database directory in PolarFS [11], as described in section 4.

Alibaba developed ParallelRaft, a modification to Raft, to manage replica consistency while maximizing I/O throughput [11]. Chunks are replicated across multiple ChunkServers on separate racks. In the typical case, this will result in three data copies. This replication group form a consensus group where I/O is replicated using ParallelRaft. Reads can be satisfied directly by the leading ChunkServer, while writes will need to be acknowledged by a majority of the followers before the leader replies back to the client [7]. RDMA and NVMe is heavily used for fast communication and buffering for communication within the system.

PolarDB also shows excellent performance for both read and write tests, far excelling both RDS MySQL and Aurora for most tests on comparable hardware. For the in-cache read tests, PolarDB outperforms RDS MySQL and Aurora with up to 66%. Similarly, for the write tests, PolarDB outperform the other systems with up to 62%. For both the out-of-cache read and write tests, PolarDB again outperform the other contenders with close to 40%. This is very strong performance, and goes to show the effectiveness of PolarDB and ParallelRaft. This further backs up Alibaba's claims on the benefits of emerging technology, such as RDMA and NVMe.

PolarFS originally set out using the Raft consensus protocol, but soon discovered that Raft didn't work well with highly concurrent I/O workloads for a number of reasons [7]. Raft is designed to be easy to understand and reason about, and has therefore made trade-offs that don't work to well in a concurrent system, as described in section

3.8 and 6. Alibaba's solution with ParallelRaft was therefore to relax on some of Raft's constraints for PolarFS, making it more suitable for high I/O concurrency. Alibaba has allowed the ParallelRaft to do out of order acknowledgements and commits, as well as apply with holes in the log. Additionally, ParallelRaft has slightly changed the leader election process to add an additional merge stage to deal with the possibility that there might be holes in the log. With the steps taken by Alibaba, the correctness of ParallelRaft can be guaranteed [7].

Alibaba has tested how PolarFS perform when the I/O queue depth grows large. With a large queue depth, the difference between Raft and ParallelRaft becomes apparent. With an I/O queue depth of 32, Raft has 2.5x longer latency and 0.5x of the IOPS compared to ParallelRaft [7]. The results indicate that ParallelRaft's out-of-order acknowledgements and commits can significantly improve performance under heavy workloads.

We see this confirmed for the tests with high thread count. For the read tests with 500 threads, we see an increase of 12% in write performance from the same test with 64 threads. For the write performance, PolarDB actually beat the other systems in our tests. This confirms that PolarDB is well capable of handling intensive workloads in an efficient manner.

Recovery is relatively simple in PolarDB. Once the underlying PolarFS is ready for operations, PolarDB can perform it's usual recovery mechanism similar to that of a traditional database system. Little detail is given on the particulars of PolarFS recovery in the material published this far.

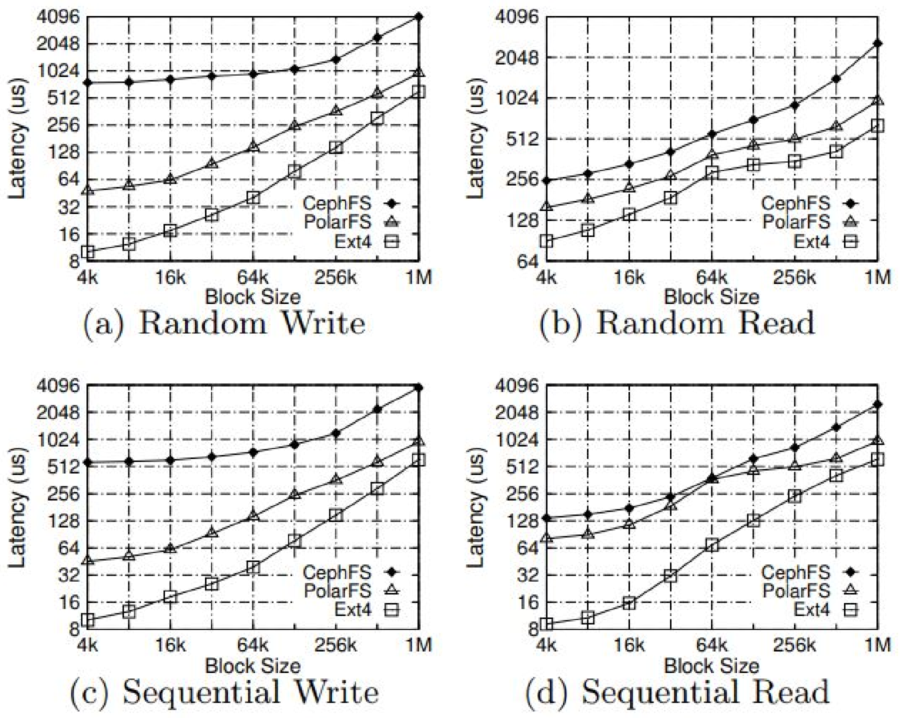


Figure 31: PolarFS - I/O latency under different loads. Source: [7]

Similarly to Aurora, Alibaba currently only support single-master with multiple read replicas. No current support seems to be available for multi-master deployments. This will need addressing in future work.

Since PolarDB is heavily reliant on NVMe and RDMA for its network communication, it can have problems covering cross-datacenter communications. This is also a problem that needs to be addressed in future work.

9.3 Comparison

Aurora and PolarDB have different goals in mind when splitting the compute and storage components. Amazon has mainly focused on minimizing amplified writes, and opted for a new approach with only replicating the log across the storage servers. PolarDB on the other hand, has aimed at making an optimized distributed file system that utilize emerging technologies such as RDMA and NVMe. This is two very different approaches. Aurora has heavily moved functionality from the compute layer to the storage layer. PolarDB, however, keeps most of this intact.

From an architectural point of view, Aurora has probably taken a better approach. By moving these components to the storage node, you are able to perform operations close to data and reduce usage of network bandwidth without the need for specialized hardware. PolarFS, however, tries to optimize these points by utilizing RDMA and NVMe, and thus remove the problems. For most of the tests we've performed, we see that PolarDB in practice is able to exceed Aurora on similiar hardware when run with 64 threads. We see however that with a larger number of threads, Aurora is able to outperform PolarDB for read performance, while PolarDB wins out for write performance.

The systems provide different resilience guarantees to failures. Aurora holds six copies of a data item across three AZ's. It can withstand the loss of three nodes without data loss, and still keep write quorum in the face of the loss of two nodes. In contrast, PolarFS only maintain three data copies by default, where the copies must be stored on separate racks. It will not be able to perform write requests in case of two simultaneous ChunkServer failures.

Amazon has in this case taken a relatively cautious approach. They believe failures will happen too frequently to be handled by a quorum model with three data copies, and has therefore safeguarded themselves by having six copies across three AZ's. PolarFS can configure how many nodes the data can be spread across, but defaults to three copies, which will work in the cases where failures are infrequent.

Aurora is able to avoid distributed consensus for writes and commits, and is able to avoid read quorums where possible. These optimizations are possible because of their quorum model, and clever use of transient state variables. This is a huge improvement over traditional systems, and greatly reduce the network consumption. PolarDB has aimed at optimizing Raft to perform better for high I/O concurrency, which is done by relaxing some of the constraints of Raft. As we saw in section 9.2, this was beneficial when the I/O queue grew large. These optimizations are great in their own respect. PolarDB has also realized they can relax some of Raft's rather strict rules in their favor while still maintaining the same guarantees as Raft.

We see that for the out-of-cache dataset, PolarDB is able to outperform Aurora for similar hardware and thread count. Again, this is probably due to the low thread count of 64, but it illustrates the benefit of the RDMA technology. As we've seen, when run with a larger thread count, Aurora outperforms PolarDB for read performance, showing the effectiveness of Aurora's ability to avoid read quorums.

Crash recovery in Aurora is relatively easy, as Aurora only need a read quorum to recompute the transient state variables and perform a cleanup. PolarDB can in this case leverage the similar recovery techniques of traditional database systems once the underlying distributed file system is ready for operations. These approaches come as a result of the architecture of the respective system.

10 Future work

This section will describe some potential areas of further research.

10.1 Look into other cloud database systems

Amazon and Alibaba are two of the main cloud platforms available, and was thus the primary goal for this thesis. That doesn't mean to say there aren't other interesting systems out there. Microsoft has recently released Hyperscale (Citrus) for their Azure platform, as well as other database offerings. Google is also a major contender in the cloud platform market, with multiple offerings ranging from Cloud Spanner and Bigtable, to the more traditional Cloud SQL service for MySQL, PostegreSQL, and SQL Server. Oracle is also in the works of developing their own database service, with coming support for MySQL. All these cloud databases are interesting in their own right, and would be interesting to look into both from an architectural standpoint, as well as from a performance standpoint.

10.2 Test other database configurations

The performance tests carried out in this thesis have mainly revolved around a master with a standby replica. The instance types have also been narrowed down to db.r5.* series for AWS, and the polar.mysql.x8.xlarge instance for Alicloud. This was chosen because it's a relatively standard setup, with typical instance types. Additionally, these instance types were chosen on the basis of being relatively equal in terms of compute power and memory size. A point we've seen is that especially Aurora's performance is sensitive to the number of connections the tests are run with. A larger number of threads, might be able to exploit the system better. There are of course other instance types available that might suit a given workload better. Equally, many different configurations might be used, such as multi-master, multi-region, and multiple replicas. An offspring of this thesis would be to see how other configurations compare, both based on the architecture and how they perform in practice.

11 Conclusion

Amazon's RDS cloud service was one of the first major steps in developing a database platform for the cloud. With the introduction of Aurora in 2015, Amazon has been able to develop a cloud database that is tailored to and can exploit the full infrastructure of AWS. Additionally, Aurora has made key innovations in splitting the compute and storage components, allowing for better and more dynamic scaling. Aurora has made interesting developments in how they manage the storage nodes, and the use of quorums with transient state has allowed Aurora to avoid distributed consensus for most cases. It's highly optimized, and avoids in most cases the need for read quorums. The AZ's are interconnected with low latency high throughput network links. Aurora also only write a segmented log over the network, greatly reducing network congestion and chatty behaviour.

One of the main takeaways from Aurora's performance in tests, is that Aurora performs generally well, though slightly less than what reported by Amazon. However, network latencies result in that Aurora sees better performance with a larger number of database connections. This is especially of significance for writes, where the database must wait for a quorum to be formed and the changes made durable before being able to respond. This is reflected in the tests carried out, as Aurora first really excel with a large number of threads.

Alibaba's PolarDB/PolarFS system is also relatively new, just introduced on VLDB last year. In comparison with Aurora, PolarDB has taken the more traditional approach with having a distributed file system under a database system. The PolarFS system utilize emerging technology such as RDMA and NVMe disks to great effect, with low latency and high throughput. Alibaba has also developed an improvement over the Raft consensus protocol, named ParallelRaft, that relax some of the constraints of Raft while still remaining correct.

In benchmarks, PolarDB/FS show strong performance. We see that for similar hardware and configurations, PolarDB is able to outperform both RDS MySQL and Aurora for both the in-cache and out-of-cache datasets, only surpassed by Aurora for the 500 thread read test. PolarDB also perform close to the benchmarks provided by Alibaba. This confirms the benefits of ParallelRaft and emerging technologies such as RDMA and NVMe.

Appendices

A Terraform test setup

The tests done on the AWS platform were conducted using the Terraform for easy automation, setup, configuration of the needed infrastructure. Two separate Terraform configurations were used, one for the RDS MySQL setup, and one for the Aurora setup. This was necessary due to Terraform's AWS provider being slightly different for the two setups. The scripts created the database instances, EC2 compute instances and configured the VPC network for allowing communication between the EC2 and database instances. The EC2 instances were accessed remotely through SSH, where a bash script configured sysbench and performed the tests against the database instances. The resultant output was then saved to file. Additionally, alarms were set up to monitor the performance during the tests, making sure neither the EC2 nor database instances ran into any resource bottlenecks.

A.1 Terraform files for Aurora

The files listed below are "main.tf", which describe the main infrastructure, followed by "alarms.tf", which describe the logging and CloudWatch alarms. The files below are for setting up the Aurora infrastructure.

Main.tf:

```
variable "region" {}
variable "shared_credentials_file" {}
variable "profile" {}
variable "my_ami" {
    type = "map"
}
variable "rds_user" {}
variable "rds_pw" {}
variable "rds_instance" {}
variable "alarms_email" {}


output "ip" {
  value = "${aws_rds_cluster.default.endpoint}"
}

output "instance" {
    value = "${var.rds_instance}"
}

provider "aws" {
    region = "${var.region}"
    shared_credentials_file = "${var.shared_credentials_file
       }"
    profile = "${var.profile}"
}

resource "aws_eip_association" "eip_assoc" {
  instance_id    = "${aws_instance.web.id}"
  allocation_id = "eipalloc-93391ff6"
}

resource "aws_instance" "web" {
    ami = "${lookup(var.my_ami, var.region)}"
    instance_type = "c5.4xlarge"
    /*associate_public_ip_address = "True"*/
    subnet_id = "<redacted>"
    vpc_security_group_ids = ["<redacted>"]
    key_name = "<redacted>"
    tags = {
        Name = "sysbench"
    }
}


resource "aws_rds_cluster_instance" "cluster_instances" {
    count = 2
    engine = "aurora-mysql"
    identifier = "aurora-cluster-demo-1"
    cluster_identifier = "${aws_rds_cluster.default.id}"
    instance_class = "${var.rds_instance}"
}


```

```
resource "aws_rds_cluster" "default" {
    cluster_identifier = "aurora-cluster-demo"
    engine = "aurora-mysql"
    availability_zones = ["us-east-1b"]
    db_subnet_group_name = "<redacted>"
    vpc_security_group_ids = ["<redacted>"]
    skip_final_snapshot = true
    database_name = "sbtest"
    master_username = "${var.rds_user}"
    master_password = "${var.rds_pw}"
}
```

Alarms.tf:

```
resource "aws_sns_topic" "alarm" {
  name = "alarms-topic"
  delivery_policy = <<EOF
{
  "http": {
    "defaultHealthyRetryPolicy": {
      "minDelayTarget": 20,
      "maxDelayTarget": 20,
      "numRetries": 3,
      "numMaxDelayRetries": 0,
      "numNoDelayRetries": 0,
      "numMinDelayRetries": 0,
      "backoffFunction": "linear"
    },
    "disableSubscriptionOverrides": false,
    "defaultThrottlePolicy": {
      "maxReceivesPerSecond": 1
    }
  }
}
EOF

  provisioner "local-exec" {
    command = "aws sns subscribe --topic-arn ${self.arn} --
        protocol email --notification-endpoint ${var.
        alarms_email}"
  }
}

resource "aws_cloudwatch_metric_alarm" "cpu" {
  alarm_name                = "web-cpu-alarm"
  comparison_operator       = "GreaterThanOrEqualToThreshold"
  evaluation_periods        = "2"
  metric_name               = "CPUUtilization"
  namespace                 = "AWS/EC2"
  period                    = "120"
  statistic                 = "Average"
  threshold                 = "80"
  alarm_description         = "This metric monitors ec2 cpu
     utilization"
  alarm_actions             = [ "${aws_sns_topic.alarm.arn}"
     ]

  dimensions = {
    InstanceId = "${aws_instance.web.id}"
  }
}

resource "aws_cloudwatch_metric_alarm" "health" {
  alarm_name                = "web-health-alarm"
  comparison_operator       = "GreaterThanOrEqualToThreshold"
  evaluation_periods        = "1"
  metric_name               = "StatusCheckFailed"
  namespace                 = "AWS/EC2"
  period                    = "120"
  statistic                 = "Average"
  threshold                 = "1"
  alarm_description         = "This metric monitors ec2
     health status"
  alarm_actions             = [ "${aws_sns_topic.alarm.arn}"
     ]

  dimensions = {
    InstanceId = "${aws_instance.web.id}"
  }
}
```

```

resource "aws_cloudwatch_metric_alarm" "cpucredit" {
  alarm_name                = "web-cpu-credit-alarm"
  comparison_operator       = "LessThanThreshold"
  evaluation_periods        = "1"
  metric_name               = "CPUCreditBalance"
  namespace                 = "AWS/EC2"
  period                    = "300"
  statistic                 = "Sum"
  threshold                 = "50"
  alarm_description         = "Triggered on low CPU credit
     balance"
  alarm_actions             = [ "${aws_sns_topic.alarm.arn}"
     ]

  dimensions = {
    InstanceId = "${aws_instance.web.id}"
  }
}

resource "aws_cloudwatch_metric_alarm" "swap_usage_too_high"
   {
  alarm_name          = "swap_usage_too_high"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = "1"
  metric_name         = "SwapUsage"
  namespace           = "AWS/RDS"
  period              = "600"
  statistic           = "Average"
  threshold           = "16384"
  alarm_description   = "Average database swap usage over
     last 10 minutes too high, performance may suffer"
  alarm_actions       = ["${aws_sns_topic.alarm.arn}"]

  dimensions = {
    DBInstanceIdentifier = "${aws_rds_cluster_instance.
       cluster_instances.id}"
  }
}

```

```
resource "aws_cloudwatch_metric_alarm" "
   cpu_credit_balance_too_low" {
  alarm_name                = "cpu_credit_balance_too_low"
  comparison_operator = "LessThanThreshold"
  evaluation_periods   = "1"
  metric_name             = "CPUCreditBalance"
  namespace                = "AWS/RDS"
  period                     = "600"
  statistic                 = "Average"
  threshold                 = "50"
  alarm_description    = "Average database CPU credit balance
     over last 10 minutes too low, expect a significant
     performance drop soon"
  alarm_actions           = ["${aws_sns_topic.alarm.arn}"]

  dimensions = {
    DBInstanceIdentifier = "${aws_rds_cluster_instance.
       cluster_instances.id}"
  }
}
```

A.2 Terraform files for MySQL

Similarly, the files listed below are "main.tf", followed by "alarms.tf". They describe the main infrastructure, followed by the infrastructure for the logging and CloudWatch alarms. The files below are for setting up the MySQL infrastructure.

Main.tf:

```
variable "region" {}
variable "shared_credentials_file" {}
variable "profile" {}
variable "my_ami" {
    type = "map"
}
variable "rds_user" {}
variable "rds_pw" {}
variable "rds_instance" {}
variable "alarms_email" {}
```

```

output "ip" {
  value = "${aws_db_instance.rds_test_mysql.endpoint}"
}

output "instance" {
    value = "${var.rds_instance}"
}

provider "aws" {
    region = "${var.region}"
    shared_credentials_file = "${var.shared_credentials_file
       }"
    profile = "${var.profile}"
}

resource "aws_eip_association" "eip_assoc" {
  instance_id   = "${aws_instance.web.id}"
  allocation_id = "eipalloc-93391ff6"
}

resource "aws_instance" "web" {
    ami = "${lookup(var.my_ami, var.region)}"
    instance_type = "c5.4xlarge"
    /*associate_public_ip_address = "True"*/
    subnet_id = "<redacted>"
    vpc_security_group_ids = ["<redacted>"]
    key_name = "<redacted>"
    tags = {
        Name = "sysbench"
    }
}

resource "aws_db_instance" "rds_test_mysql" {
    allocated_storage = 230
    storage_type = "io1"
    iops = 4500
    engine = "mysql"
    engine_version = "5.7"
    instance_class = "${var.rds_instance}"
    name = "sbtest"
    username = "${var.rds_user}"
    password = "${var.rds_pw}"
    parameter_group_name = "default.mysql5.7"
    db_subnet_group_name = "<redacted>"
    vpc_security_group_ids = ["<redacted>"]
    apply_immediately = true
    skip_final_snapshot = true
    multi_az = true
    performance_insights_enabled = true
    monitoring_interval  = 5
    monitoring_role_arn  = "${aws_iam_role.rds_enhanced_monitoring.arn}"

    //backup_retention_period = 1
    //availability_zone = "us-east-1b"
}

resource "aws_iam_role" "rds_enhanced_monitoring" {
  name               = "rds-enhanced_monitoring-role"
  assume_role_policy = "${data.aws_iam_policy_document.rds_enhanced_monitoring.json}"
}

resource "aws_iam_role_policy_attachment" "rds_enhanced_monitoring" {
  role       = "${aws_iam_role.rds_enhanced_monitoring.name}"
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonRDSEnhancedMonitoringRole"
}

data "aws_iam_policy_document" "rds_enhanced_monitoring" {
  statement {
    actions = [
      "sts:AssumeRole",
    ]

    effect = "Allow"

    principals {
      type          = "Service"
      identifiers = ["monitoring.rds.amazonaws.com"]
    }
  }
}

/*
resource "aws_db_instance" "rds_test_mysql_replica" {
    allocated_storage = 100
    storage_type = "io1"
    iops = 3000
    engine = "mysql"
    engine_version = "5.7"
    instance_class = "${var.rds_instance}"
    parameter_group_name = "default.mysql5.7"
    vpc_security_group_ids = ["<redacted>"]
    apply_immediately = true
    skip_final_snapshot = true
    multi_az = false

    replicate_source_db = "${aws_db_instance.rds_test_mysql.
       id}"
    username = ""
    password = ""
    backup_retention_period = 0
    availability_zone = "us-east-1a"
}
*/
```

Alarms.tf:

```
resource "aws_sns_topic" "alarm" {
  name = "alarms-topic"
  delivery_policy = <<EOF
{
  "http": {
    "defaultHealthyRetryPolicy": {
      "minDelayTarget": 20,
      "maxDelayTarget": 20,
      "numRetries": 3,
      "numMaxDelayRetries": 0,
      "numNoDelayRetries": 0,
      "numMinDelayRetries": 0,
      "backoffFunction": "linear"
    },
    "disableSubscriptionOverrides": false,
    "defaultThrottlePolicy": {
      "maxReceivesPerSecond": 1
    }
  }
}
EOF

  provisioner "local-exec" {
    command = "aws sns subscribe --topic-arn ${self.arn} --
        protocol email --notification-endpoint ${var.
        alarms_email}"
  }
}

resource "aws_cloudwatch_metric_alarm" "cpu" {
  alarm_name                = "web-cpu-alarm"
  comparison_operator       = "GreaterThanOrEqualToThreshold"
  evaluation_periods        = "2"
  metric_name               = "CPUUtilization"
  namespace                 = "AWS/EC2"
  period                    = "120"
  statistic                 = "Average"
  threshold                 = "80"
  alarm_description         = "This metric monitors ec2 cpu
      utilization"
  alarm_actions             = [ "${aws_sns_topic.alarm.arn}"
      ]

  dimensions = {
    InstanceId = "${aws_instance.web.id}"
  }
}

resource "aws_cloudwatch_metric_alarm" "health" {
  alarm_name                = "web-health-alarm"
  comparison_operator       = "GreaterThanOrEqualToThreshold"
  evaluation_periods        = "1"
  metric_name               = "StatusCheckFailed"
  namespace                 = "AWS/EC2"
  period                    = "120"
  statistic                 = "Average"
  threshold                 = "1"
  alarm_description         = "This metric monitors ec2
     health status"
  alarm_actions             = [ "${aws_sns_topic.alarm.arn}"
     ]

  dimensions = {
    InstanceId = "${aws_instance.web.id}"
  }
}

resource "aws_cloudwatch_metric_alarm" "cpucredit" {
  alarm_name                = "web-cpu-credit-alarm"
  comparison_operator       = "LessThanThreshold"
  evaluation_periods        = "1"
  metric_name               = "CPUCreditBalance"
  namespace                 = "AWS/EC2"
  period                    = "300"
  statistic                 = "Sum"
  threshold                 = "50"
  alarm_description         = "Triggered on low CPU credit
     balance"
  alarm_actions             = [ "${aws_sns_topic.alarm.arn}"
     ]

  dimensions = {
    InstanceId = "${aws_instance.web.id}"
  }
}

resource "aws_cloudwatch_metric_alarm" "swap_usage_too_high"
   {
  alarm_name          = "swap_usage_too_high"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = "1"
  metric_name         = "SwapUsage"
  namespace           = "AWS/RDS"
  period              = "600"
  statistic           = "Average"
  threshold           = "16384"
  alarm_description   = "Average database swap usage over
     last 10 minutes too high, performance may suffer"
  alarm_actions       = ["${aws_sns_topic.alarm.arn}"]

  dimensions = {
    DBInstanceIdentifier = "${aws_db_instance.rds_test_mysql.
       id}"
  }
}

resource "aws_cloudwatch_metric_alarm" "
   cpu_credit_balance_too_low" {
  alarm_name          = "cpu_credit_balance_too_low"
  comparison_operator = "LessThanThreshold"
  evaluation_periods  = "1"
  metric_name         = "CPUCreditBalance"
  namespace           = "AWS/RDS"
  period              = "600"
  statistic           = "Average"
  threshold           = "50"
  alarm_description   = "Average database CPU credit balance
     over last 10 minutes too low, expect a significant
     performance drop soon"
  alarm_actions       = ["${aws_sns_topic.alarm.arn}"]

  dimensions = {
    DBInstanceIdentifier = "${aws_db_instance.rds_test_mysql.
       id}"
  }
}
```

B Sysbench results

B.1 mysql-db.r5.large-32-read-run-1

```
transactions: 1501385 (2502.26 per sec.)
queries: 18016620 (30027.10 per sec.)
```

B.2 mysql-db.r5.large-32-read-run-2

```
transactions: 1583813 (2639.63 per sec.)
queries: 19005756 (31675.61 per sec.)
```

B.3 mysql-db.r5.large-32-read-run-3

```
transactions: 1571151 (2618.53 per sec.)
queries: 18853812 (31422.37 per sec.)
```

B.4 mysql-db.r5.large-32-read-run-4

```
transactions: 1541962 (2569.89 per sec.)
queries: 18503544 (30838.64 per sec.)
```

B.5 mysql-db.r5.large-32-read-run-5

```
transactions: 1584257 (2640.37 per sec.)
queries: 19011084 (31684.43 per sec.)
```

B.6 mysql-db.r5.xlarge-32-read-run-1

```
transactions: 2787900 (4646.41 per sec.)
queries: 33454800 (55756.91 per sec.)
```

B.7 mysql-db.r5.xlarge-32-read-run-2

```
transactions: 2823181 (4705.21 per sec.)
queries: 33878172 (56462.51 per sec.)
```

B.8 mysql-db.r5.xlarge-32-read-run-3

```
transactions: 2840713 (4734.43 per sec.)
queries: 34088556 (56813.14 per sec.)
```

B.9 mysql-db.r5.xlarge-32-read-run-4

```
transactions: 2875523 (4792.44 per sec.)
queries: 34506276 (57509.33 per sec.)
```

B.10 mysql-db.r5.xlarge-32-read-run-5

```
transactions: 2870306 (4783.75 per sec.)
queries: 34443672 (57405.00 per sec.)
```

B.11 mysql-db.r5.xlarge-32-read-run-1

```
transactions: 2787900 (4646.41 per sec.)
queries: 33454800 (55756.91 per sec.)
```

B.12 mysql-db.r5.xlarge-32-read-run-2

```
transactions: 2823181 (4705.21 per sec.)
queries: 33878172 (56462.51 per sec.)
```

B.13 mysql-db.r5.xlarge-32-read-run-3

```
transactions: 2840713 (4734.43 per sec.)
queries: 34088556 (56813.14 per sec.)
```

B.14 mysql-db.r5.xlarge-32-read-run-4

```
transactions: 2875523 (4792.44 per sec.)
queries: 34506276 (57509.33 per sec.)
```

B.15 mysql-db.r5.xlarge-32-read-run-5

```
transactions: 2870306 (4783.75 per sec.)
queries: 34443672 (57405.00 per sec.)
```

B.16 mysql-db.r5.4xlarge-32-read-run-1

```
transactions: 5683153 (9471.74 per sec.)
queries: 68197836 (113660.82 per sec.)
```

B.17 mysql-db.r5.4xlarge-32-read-run-2

```
transactions: 5698652 (9497.56 per sec.)
queries: 68383824 (113970.71 per sec.)
```

B.18 mysql-db.r5.4xlarge-32-read-run-3

```
transactions: 5627710 (9379.33 per sec.)
queries: 67532520 (112551.91 per sec.)
```

B.19 mysql-db.r5.4xlarge-32-read-run-4

```
transactions: 5680778 (9467.77 per sec.)
queries: 68169336 (113613.30 per sec.)
```

B.20 mysql-db.r5.4xlarge-32-read-run-5

```
transactions: 5641503 (9402.32 per sec.)
queries: 67698036 (112827.85 per sec.)
```

B.21 mysql-multiaz-db.r5.large-16-read-run-1

```
transactions: 1515140 (2525.20 per sec.)
queries: 18181680 (30302.41 per sec.)
```

B.22 mysql-multiaz-db.r5.large-16-read-run-2

```
transactions: 1569451 (2615.72 per sec.)
queries: 18833412 (31388.63 per sec.)
```

B.23 mysql-multiaz-db.r5.large-16-read-run-3

```
transactions: 1516419 (2527.33 per sec.)
queries: 18197028 (30327.99 per sec.)
```

B.24 mysql-multiaz-db.r5.large-16-read-run-4

```
transactions: 1554832 (2591.36 per sec.)
queries: 18657984 (31096.26 per sec.)
```

B.25 mysql-multiaz-db.r5.large-16-read-run-5

```
transactions: 1448180 (2413.60 per sec.)
queries: 17378160 (28963.23 per sec.)
```

B.26 mysql-multiaz-db.r5.xlarge-16-read-run-1

```
transactions: 1511682 (2519.43 per sec.)
queries: 18140184 (30233.14 per sec.)
```

B.27 mysql-multiaz-db.r5.xlarge-16-read-run-2

```
transactions: 1512733 (2521.19 per sec.)
queries: 18152796 (30254.24 per sec.)
```

B.28 mysql-multiaz-db.r5.xlarge-16-read-run-3

```
transactions: 1511667 (2519.41 per sec.)
queries: 18140004 (30232.88 per sec.)
```

B.29 mysql-multiaz-db.r5.xlarge-16-read-run-4

```
transactions: 1524832 (2541.35 per sec.)
queries: 18297984 (30496.18 per sec.)
```

B.30 mysql-multiaz-db.r5.xlarge-16-read-run-5

```
transactions: 1519324 (2532.17 per sec.)
queries: 18231888 (30386.04 per sec.)
```

B.31 mysql-multiaz-db.r5.xlarge-16-read-run-1

```
transactions: 1511682 (2519.43 per sec.)
queries: 18140184 (30233.14 per sec.)
```

B.32 mysql-multiaz-db.r5.xlarge-16-read-run-2

```
transactions: 1512733 (2521.19 per sec.)
queries: 18152796 (30254.24 per sec.)
```

B.33 mysql-multiaz-db.r5.xlarge-16-read-run-3

```
transactions: 1511667 (2519.41 per sec.)
queries: 18140004 (30232.88 per sec.)
```

B.34 mysql-multiaz-db.r5.xlarge-16-read-run-4

```
transactions: 1524832 (2541.35 per sec.)
queries: 18297984 (30496.18 per sec.)
```

B.35 mysql-multiaz-db.r5.xlarge-16-read-run-5

```
transactions: 1519324 (2532.17 per sec.)
queries: 18231888 (30386.04 per sec.)
```

B.36 mysql-multiaz-db.r5.4xlarge-16-read-run-1

```
transactions: 5631658 (9386.02 per sec.)
queries: 67579896 (112632.21 per sec.)
```

B.37 mysql-multiaz-db.r5.4xlarge-16-read-run-2

```
transactions: 5659505 (9432.43 per sec.)
queries: 67914060 (113189.13 per sec.)
```

B.38 mysql-multiaz-db.r5.4xlarge-16-read-run-3

```
transactions: 5643466 (9405.69 per sec.)
queries: 67721592 (112868.31 per sec.)
```

B.39 mysql-multiaz-db.r5.4xlarge-16-read-run-4

```
transactions: 5648478 (9414.05 per sec.)
queries: 67781736 (112968.57 per sec.)
```

B.40 mysql-multiaz-db.r5.4xlarge-16-read-run-5

```
transactions: 5632216 (9386.94 per sec.)
queries: 67586592 (112643.28 per sec.)
```

B.41 mysql-multiaz-db.r5.large-32-read-run-1

```
transactions: 1537281 (2562.08 per sec.)
queries: 18447372 (30744.96 per sec.)
```

B.42 mysql-multiaz-db.r5.large-32-read-run-2

```
transactions: 1526942 (2544.85 per sec.)
queries: 18323304 (30538.19 per sec.)
```

B.43 mysql-multiaz-db.r5.large-32-read-run-3

```
transactions: 1539045 (2565.02 per sec.)
queries: 18468540 (30780.26 per sec.)
```

B.44 mysql-multiaz-db.r5.large-32-read-run-4

```
transactions: 1530477 (2550.74 per sec.)
queries: 18365724 (30608.91 per sec.)
```

B.45 mysql-multiaz-db.r5.large-32-read-run-5

```
transactions: 1501238 (2502.01 per sec.)
queries: 18014856 (30024.15 per sec.)
```

B.46 mysql-multiaz-db.r5.xlarge-32-read-run-1

```
transactions: 2673621 (4455.94 per sec.)
queries: 32083452 (53471.26 per sec.)
```

B.47 mysql-multiaz-db.r5.xlarge-32-read-run-2

```
transactions: 2729938 (4549.80 per sec.)
queries: 32759256 (54597.57 per sec.)
```

B.48 mysql-multiaz-db.r5.xlarge-32-read-run-3

```
transactions: 2691213 (4485.26 per sec.)
queries: 32294556 (53823.11 per sec.)
```

B.49 mysql-multiaz-db.r5.xlarge-32-read-run-4

```
transactions: 2678955 (4464.83 per sec.)
queries: 32147460 (53577.92 per sec.)
```

B.50 mysql-multiaz-db.r5.xlarge-32-read-run-5

```
transactions: 2654307 (4423.74 per sec.)
queries: 31851684 (53084.91 per sec.)
```

B.51 mysql-multiaz-db.r5.xlarge-32-read-run-1

```
transactions: 2673621 (4455.94 per sec.)
queries: 32083452 (53471.26 per sec.)
```

B.52 mysql-multiaz-db.r5.xlarge-32-read-run-2

```
transactions: 2729938 (4549.80 per sec.)
queries: 32759256 (54597.57 per sec.)
```

B.53 mysql-multiaz-db.r5.xlarge-32-read-run-3

```
transactions: 2691213 (4485.26 per sec.)
queries: 32294556 (53823.11 per sec.)
```

B.54 mysql-multiaz-db.r5.xlarge-32-read-run-4

```
transactions: 2678955 (4464.83 per sec.)
queries: 32147460 (53577.92 per sec.)
```

B.55 mysql-multiaz-db.r5.xlarge-32-read-run-5

```
transactions: 2654307 (4423.74 per sec.)
queries: 31851684 (53084.91 per sec.)
```

B.56 mysql-multiaz-db.r5.4xlarge-32-read-run-1

```
transactions: 8671792 (14452.78 per sec.)
queries: 104061504 (173433.42 per sec.)
```

B.57 mysql-multiaz-db.r5.4xlarge-32-read-run-2

```
transactions: 8660634 (14434.19 per sec.)
queries: 103927608 (173210.27 per sec.)
```

B.58 mysql-multiaz-db.r5.4xlarge-32-read-run-3

```
transactions: 8715910 (14526.31 per sec.)
queries: 104590920 (174315.77 per sec.)
```

B.59 mysql-multiaz-db.r5.4xlarge-32-read-run-4

```
transactions: 8681152 (14468.37 per sec.)
queries: 104173824 (173620.45 per sec.)
```

B.60 mysql-multiaz-db.r5.4xlarge-32-read-run-5

```
transactions: 8668996 (14448.13 per sec.)
queries: 104027952 (173377.50 per sec.)
```

B.61 mysql-multiaz-db.r5.large-64-read-run-1

```
transactions: 1517586 (2529.20 per sec.)
queries: 18211032 (30350.39 per sec.)
```

B.62 mysql-multiaz-db.r5.large-64-read-run-2

```
transactions: 1514653 (2524.32 per sec.)
queries: 18175836 (30291.89 per sec.)
```

B.63 mysql-multiaz-db.r5.large-64-read-run-3

```
transactions: 1473043 (2454.98 per sec.)
queries: 17676516 (29459.70 per sec.)
```

B.64 mysql-multiaz-db.r5.large-64-read-run-4

```
transactions: 1516598 (2527.57 per sec.)
queries: 18199176 (30330.83 per sec.)
```

B.65 mysql-multiaz-db.r5.large-64-read-run-5

```
transactions: 1521308 (2535.41 per sec.)
queries: 18255696 (30424.95 per sec.)
```

B.66 mysql-multiaz-db.r5.xlarge-64-read-run-1

```
transactions: 2929399 (4882.14 per sec.)
queries: 35152788 (58585.66 per sec.)
```

B.67 mysql-multiaz-db.r5.xlarge-64-read-run-2

```
transactions: 2874429 (4790.53 per sec.)
queries: 34493148 (57486.39 per sec.)
```

B.68 mysql-multiaz-db.r5.xlarge-64-read-run-3

```
transactions: 2904308 (4840.33 per sec.)
queries: 34851696 (58083.95 per sec.)
```

B.69 mysql-multiaz-db.r5.xlarge-64-read-run-4

```
transactions: 2868739 (4781.04 per sec.)
queries: 34424868 (57372.51 per sec.)
```

B.70 mysql-multiaz-db.r5.xlarge-64-read-run-5

```
transactions: 2833278 (4721.95 per sec.)
queries: 33999336 (56663.38 per sec.)
```

B.71 mysql-multiaz-db.r5.xlarge-64-read-run-1

```
transactions: 2929399 (4882.14 per sec.)
queries: 35152788 (58585.66 per sec.)
```

B.72 mysql-multiaz-db.r5.xlarge-64-read-run-2

```
transactions: 2874429 (4790.53 per sec.)
queries: 34493148 (57486.39 per sec.)
```

B.73 mysql-multiaz-db.r5.xlarge-64-read-run-3

```
transactions: 2904308 (4840.33 per sec.)
queries: 34851696 (58083.95 per sec.)
```

B.74 mysql-multiaz-db.r5.xlarge-64-read-run-4

```
transactions: 2868739 (4781.04 per sec.)
queries: 34424868 (57372.51 per sec.)
```

B.75 mysql-multiaz-db.r5.xlarge-64-read-run-5

```
transactions: 2833278 (4721.95 per sec.)
queries: 33999336 (56663.38 per sec.)
```

B.76 mysql-multiaz-db.r5.4xlarge-64-read-run-1

```
transactions: 11537383 (19228.50 per sec.)
queries: 138448596 (230741.98 per sec.)
```

B.77 mysql-multiaz-db.r5.4xlarge-64-read-run-2

```
transactions: 11447493 (19078.67 per sec.)
queries: 137369916 (228944.09 per sec.)
```

B.78 mysql-multiaz-db.r5.4xlarge-64-read-run-3

```
transactions: 11496019 (19159.56 per sec.)
queries: 137952228 (229914.72 per sec.)
```

B.79 mysql-multiaz-db.r5.4xlarge-64-read-run-4

```
transactions: 11468609 (19113.87 per sec.)
queries: 137623308 (229366.44 per sec.)
```

B.80 mysql-multiaz-db.r5.4xlarge-64-read-run-5

```
transactions: 11405875 (19009.33 per sec.)
queries: 136870500 (228111.93 per sec.)
```

B.81 aurora-db.r5.large-16-read-run-1

```
transactions: 1934444 (3224.01 per sec.)
queries: 23213328 (38688.17 per sec.)
```

B.82 aurora-db.r5.large-16-read-run-2

```
transactions: 1947570 (3245.91 per sec.)
queries: 23370840 (38950.88 per sec.)
```

B.83 aurora-db.r5.large-16-read-run-3

```
transactions: 1887892 (3146.44 per sec.)
queries: 22654704 (37757.32 per sec.)
```

B.84 aurora-db.r5.large-16-read-run-4

```
transactions: 1950697 (3251.12 per sec.)
queries: 23408364 (39013.41 per sec.)
```

B.85 aurora-db.r5.large-16-read-run-5

```
transactions: 1919590 (3199.27 per sec.)
queries: 23035080 (38391.24 per sec.)
```

B.86 aurora-db.r5.xlarge-16-read-run-1

```
transactions: 2932605 (4887.61 per sec.)
queries: 35191260 (58651.38 per sec.)
```

B.87 aurora-db.r5.xlarge-16-read-run-2

```
transactions: 2936041 (4893.34 per sec.)
queries: 35232492 (58720.14 per sec.)
```

B.88 aurora-db.r5.xlarge-16-read-run-3

```
transactions: 2935896 (4893.09 per sec.)
queries: 35230752 (58717.11 per sec.)
```

B.89 aurora-db.r5.xlarge-16-read-run-4

```
transactions: 2929047 (4881.67 per sec.)
queries: 35148564 (58580.09 per sec.)
```

B.90 aurora-db.r5.xlarge-16-read-run-5

```
transactions: 2932008 (4886.63 per sec.)
queries: 35184096 (58639.51 per sec.)
```

B.91 aurora-db.r5.2xlarge-16-read-run-1

```
transactions: 1537217 (2562.00 per sec.)
queries: 18446604 (30744.00 per sec.)
```

B.92 aurora-db.r5.2xlarge-16-read-run-2

```
transactions: 1536761 (2561.23 per sec.)
queries: 18441132 (30734.80 per sec.)
```

B.93 aurora-db.r5.2xlarge-16-read-run-3

```
transactions: 1535421 (2559.00 per sec.)
queries: 18425052 (30708.01 per sec.)
```

B.94 aurora-db.r5.2xlarge-16-read-run-4

```
transactions: 1536403 (2560.64 per sec.)
queries: 18436836 (30727.67 per sec.)
```

B.95 aurora-db.r5.2xlarge-16-read-run-5

```
transactions: 1539938 (2566.53 per sec.)
queries: 18479256 (30798.35 per sec.)
```

B.96 aurora-db.r5.4xlarge-16-read-run-1

```
transactions: 4541698 (7569.42 per sec.)
queries: 54500376 (90833.06 per sec.)
```

B.97 aurora-db.r5.4xlarge-16-read-run-2

```
transactions: 4482763 (7471.20 per sec.)
queries: 53793156 (89654.44 per sec.)
```

B.98 aurora-db.r5.4xlarge-16-read-run-3

```
transactions: 4637420 (7728.96 per sec.)
queries: 55649040 (92747.47 per sec.)
```

B.99 aurora-db.r5.4xlarge-16-read-run-4

```
transactions: 4615660 (7692.70 per sec.)
queries: 55387920 (92312.37 per sec.)
```

B.100 aurora-db.r5.4xlarge-16-read-run-5

```
transactions: 4609188 (7681.90 per sec.)
queries: 55310256 (92182.77 per sec.)
```

B.101 aurora-db.r5.large-32-read-run-1

```
transactions: 2188290 (3647.07 per sec.)
queries: 26259480 (43764.81 per sec.)
```

B.102 aurora-db.r5.large-32-read-run-2

```
transactions: 2171600 (3619.25 per sec.)
queries: 26059200 (43431.03 per sec.)
```

B.103 aurora-db.r5.large-32-read-run-3

```
transactions: 2194521 (3657.45 per sec.)
queries: 26334252 (43889.44 per sec.)
```

B.104 aurora-db.r5.large-32-read-run-4

```
transactions: 2180721 (3634.46 per sec.)
queries: 26168652 (43613.48 per sec.)
```

B.105 aurora-db.r5.large-32-read-run-5

```
transactions: 2154863 (3590.69 per sec.)
queries: 25858356 (43088.30 per sec.)
```

B.106 aurora-db.r5.xlarge-32-read-run-1

```
transactions: 3933454 (6555.62 per sec.)
queries: 47201448 (78667.42 per sec.)
```

B.107 aurora-db.r5.xlarge-32-read-run-2

```
transactions: 3917773 (6528.13 per sec.)
queries: 47013276 (78337.60 per sec.)
```

B.108 aurora-db.r5.xlarge-32-read-run-3

```
transactions: 3973434 (6622.24 per sec.)
queries: 47681208 (79466.94 per sec.)
```

B.109 aurora-db.r5.xlarge-32-read-run-4

```
transactions: 3976620 (6627.55 per sec.)
queries: 47719440 (79530.62 per sec.)
```

B.110 aurora-db.r5.xlarge-32-read-run-5

```
transactions: 3922834 (6537.92 per sec.)
queries: 47074008 (78455.08 per sec.)
```

B.111 aurora-db.r5.2xlarge-32-read-run-1

```
transactions: 2937202 (4894.84 per sec.)
queries: 35246424 (58738.03 per sec.)
```

B.112 aurora-db.r5.2xlarge-32-read-run-2

```
transactions: 2941758 (4902.85 per sec.)
queries: 35301096 (58834.20 per sec.)
```

B.113 aurora-db.r5.2xlarge-32-read-run-3

```
transactions: 2947068 (4911.70 per sec.)
queries: 35364816 (58940.38 per sec.)
```

B.114 aurora-db.r5.2xlarge-32-read-run-4

```
transactions: 2946985 (4911.57 per sec.)
queries: 35363820 (58938.79 per sec.)
```

B.115 aurora-db.r5.2xlarge-32-read-run-5

```
transactions: 2961252 (4935.34 per sec.)
queries: 35535024 (59224.07 per sec.)
```

B.116 aurora-db.r5.4xlarge-32-read-run-1

```
transactions: 6686282 (11143.63 per sec.)
queries: 80235384 (133723.58 per sec.)
```

B.117 aurora-db.r5.4xlarge-32-read-run-2

```
transactions: 6633615 (11055.86 per sec.)
queries: 79603380 (132670.28 per sec.)
```

B.118 aurora-db.r5.4xlarge-32-read-run-3

```
transactions: 6713095 (11188.31 per sec.)
queries: 80557140 (134259.75 per sec.)
```

B.119 aurora-db.r5.4xlarge-32-read-run-4

```
transactions: 6648030 (11079.89 per sec.)
queries: 79776360 (132958.66 per sec.)
```

B.120 aurora-db.r5.4xlarge-32-read-run-5

```
transactions: 6696640 (11160.89 per sec.)
queries: 80359680 (133930.71 per sec.)
```

B.121 aurora-db.r5.large-64-read-run-1

```
transactions: 2193875 (3656.31 per sec.)
queries: 26326500 (43875.76 per sec.)
```

B.122 aurora-db.r5.large-64-read-run-2

```
transactions: 2066346 (3443.78 per sec.)
queries: 24796152 (41325.33 per sec.)
```

B.123 aurora-db.r5.large-64-read-run-3

```
transactions: 2140669 (3567.63 per sec.)
queries: 25688028 (42811.59 per sec.)
```

B.124 aurora-db.r5.large-64-read-run-4

```
transactions: 2136602 (3560.87 per sec.)
queries: 25639224 (42730.44 per sec.)
```

B.125 aurora-db.r5.large-64-read-run-5

```
transactions: 2149448 (3582.28 per sec.)
queries: 25793376 (42987.42 per sec.)
```

B.126 aurora-db.r5.xlarge-64-read-run-1

```
transactions: 4493128 (7488.29 per sec.)
queries: 53917536 (89859.44 per sec.)
```

B.127 aurora-db.r5.xlarge-64-read-run-2

```
transactions: 4463505 (7438.90 per sec.)
queries: 53562060 (89266.77 per sec.)
```

B.128 aurora-db.r5.xlarge-64-read-run-3

```
transactions: 4489672 (7482.51 per sec.)
queries: 53876064 (89790.13 per sec.)
```

B.129 aurora-db.r5.xlarge-64-read-run-4

```
transactions: 4468455 (7447.15 per sec.)
queries: 53621460 (89365.79 per sec.)
```

B.130 aurora-db.r5.xlarge-64-read-run-5

```
transactions: 4554236 (7590.09 per sec.)
queries: 54650832 (91081.13 per sec.)
```

B.131 aurora-db.r5.2xlarge-64-read-run-1

```
transactions: 5473537 (9122.31 per sec.)
queries: 65682444 (109467.72 per sec.)
```

B.132 aurora-db.r5.2xlarge-64-read-run-2

```
transactions: 5457045 (9094.85 per sec.)
queries: 65484540 (109138.20 per sec.)
```

B.133 aurora-db.r5.2xlarge-64-read-run-3

```
transactions: 5444577 (9074.04 per sec.)
queries: 65334924 (108888.47 per sec.)
```

B.134 aurora-db.r5.2xlarge-64-read-run-4

```
transactions: 5474813 (9124.46 per sec.)
queries: 65697756 (109493.51 per sec.)
```

B.135 aurora-db.r5.2xlarge-64-read-run-5

```
transactions: 5472622 (9120.79 per sec.)
queries: 65671464 (109449.54 per sec.)
```

B.136 aurora-db.r5.4xlarge-64-read-run-1

```
transactions: 9846945 (16411.15 per sec.)
queries: 118163340 (196933.79 per sec.)
```

B.137 aurora-db.r5.4xlarge-64-read-run-2

```
transactions: 9856589 (16427.23 per sec.)
queries: 118279068 (197126.73 per sec.)
```

B.138 aurora-db.r5.4xlarge-64-read-run-3

```
transactions: 9763962 (16272.84 per sec.)
queries: 117167544 (195274.10 per sec.)
```

B.139 aurora-db.r5.4xlarge-64-read-run-4

```
transactions: 9614825 (16024.31 per sec.)
queries: 115377900 (192291.71 per sec.)
```

B.140 aurora-db.r5.4xlarge-64-read-run-5

```
transactions: 9492753 (15820.83 per sec.)
queries: 113913036 (189849.99 per sec.)
```

B.141 polardb-manythreads-2-read-run-1

```
transactions: 711003 (1185.00 per sec.)
queries: 8532036 (14219.96 per sec.)
```

B.142 polardb-manythreads-2-read-run-2

```
transactions: 721548 (1202.57 per sec.)
queries: 8658576 (14430.87 per sec.)
```

B.143 polardb-manythreads-2-read-run-3

```
transactions: 718696 (1197.82 per sec.)
queries: 8624352 (14373.82 per sec.)
```

B.144 polardb-manythreads-2-read-run-4

```
transactions: 716228 (1193.71 per sec.)
queries: 8594736 (14324.46 per sec.)
```

B.145 polardb-manythreads-2-read-run-5

```
transactions: 716741 (1194.56 per sec.)
queries: 8600892 (14334.73 per sec.)
```

B.146 polardb-manythreads-4-read-run-1

```
transactions: 1401414 (2335.67 per sec.)
queries: 16816968 (28028.03 per sec.)
```

B.147 polardb-manythreads-4-read-run-2

```
transactions: 1400264 (2333.75 per sec.)
queries: 16803168 (28005.05 per sec.)
```

B.148 polardb-manythreads-4-read-run-3

```
transactions: 1396134 (2326.87 per sec.)
queries: 16753608 (27922.49 per sec.)
```

B.149 polardb-manythreads-4-read-run-4

```
transactions: 1406671 (2344.43 per sec.)
queries: 16880052 (28133.20 per sec.)
```

B.150 polardb-manythreads-4-read-run-5

```
transactions: 1402650 (2337.73 per sec.)
queries: 16831800 (28052.80 per sec.)
```

B.151 polardb-manythreads-8-read-run-1

```
transactions: 2603011 (4338.29 per sec.)
queries: 31236132 (52059.51 per sec.)
```

B.152 polardb-manythreads-8-read-run-2

```
transactions: 2620029 (4366.67 per sec.)
queries: 31440348 (52400.10 per sec.)
```

B.153 polardb-manythreads-8-read-run-3

```
transactions: 2626009 (4376.64 per sec.)
queries: 31512108 (52519.67 per sec.)
```

B.154 polardb-manythreads-8-read-run-4

```
transactions: 2587242 (4312.03 per sec.)
queries: 31046904 (51744.35 per sec.)
```

B.155 polardb-manythreads-8-read-run-5

```
transactions: 2633504 (4389.13 per sec.)
queries: 31602048 (52669.58 per sec.)
```

B.156 polardb-manythreads-16-read-run-1

```
transactions: 4587733 (7646.11 per sec.)
queries: 55052796 (91753.29 per sec.)
```

B.157 polardb-manythreads-16-read-run-2

```
transactions: 4637825 (7729.60 per sec.)
queries: 55653900 (92755.15 per sec.)
```

B.158 polardb-manythreads-16-read-run-3

```
transactions: 4640727 (7734.44 per sec.)
queries: 55688724 (92813.23 per sec.)
```

B.159 polardb-manythreads-16-read-run-4

```
transactions: 4639116 (7731.75 per sec.)
queries: 55669392 (92781.02 per sec.)
```

B.160 polardb-manythreads-16-read-run-5

```
transactions: 4593912 (7656.41 per sec.)
queries: 55126944 (91876.90 per sec.)
```

B.161 polardb-manythreads-32-read-run-1

```
transactions: 7695923 (12826.24 per sec.)
queries: 92351076 (153914.90 per sec.)
```

B.162 polardb-manythreads-32-read-run-2

```
transactions: 7755410 (12925.38 per sec.)
queries: 93064920 (155104.59 per sec.)
```

B.163 polardb-manythreads-32-read-run-3

```
transactions: 7721812 (12869.40 per sec.)
queries: 92661744 (154432.75 per sec.)
```

B.164 polardb-manythreads-32-read-run-4

```
transactions: 7824645 (13040.78 per sec.)
queries: 93895740 (156489.31 per sec.)
```

B.165 polardb-manythreads-32-read-run-5

```
transactions: 7729359 (12881.96 per sec.)
queries: 92752308 (154583.54 per sec.)
```

B.166 polardb-manythreads-64-read-run-1

```
transactions: 9087806 (15145.66 per sec.)
queries: 109053672 (181747.97 per sec.)
```

B.167 polardb-manythreads-64-read-run-2

```
transactions: 9077099 (15127.80 per sec.)
queries: 108925188 (181533.65 per sec.)
```

B.168 polardb-manythreads-64-read-run-3

```
transactions: 9071517 (15118.50 per sec.)
queries: 108858204 (181421.98 per sec.)
```

B.169 polardb-manythreads-64-read-run-4

```
transactions: 9042363 (15069.90 per sec.)
queries: 108508356 (180838.78 per sec.)
```

B.170 polardb-manythreads-64-read-run-5

```
transactions: 9094647 (15157.06 per sec.)
queries: 109135764 (181884.74 per sec.)
```

B.171 polardb-150GB-2-read-run-1

```
transactions: 601289 (1002.14 per sec.)
queries: 7215468 (12025.71 per sec.)
```

B.172 polardb-150GB-2-read-run-2

```
transactions: 626077 (1043.46 per sec.)
queries: 7512924 (12521.46 per sec.)
```

B.173 polardb-150GB-2-read-run-3

```
transactions: 652003 (1086.66 per sec.)
queries: 7824036 (13039.97 per sec.)
```

B.174 polardb-150GB-2-read-run-4

```
transactions: 672753 (1121.25 per sec.)
queries: 8073036 (13454.97 per sec.)
```

B.175 polardb-150GB-2-read-run-5

```
transactions: 669660 (1116.09 per sec.)
queries: 8035920 (13393.10 per sec.)
```

B.176 polardb-150GB-4-read-run-1

```
transactions: 1318893 (2198.14 per sec.)
queries: 15826716 (26377.66 per sec.)
```

B.177 polardb-150GB-4-read-run-2

```
transactions: 1343048 (2238.40 per sec.)
queries: 16116576 (26860.76 per sec.)
```

B.178 polardb-150GB-4-read-run-3

```
transactions: 1329133 (2215.20 per sec.)
queries: 15949596 (26582.46 per sec.)
```

B.179 polardb-150GB-4-read-run-4

```
transactions: 1326493 (2210.81 per sec.)
queries: 15917916 (26529.67 per sec.)
```

B.180 polardb-150GB-4-read-run-5

```
transactions: 1315901 (2193.15 per sec.)
queries: 15790812 (26317.82 per sec.)
```

B.181 polardb-150GB-8-read-run-1

```
transactions: 2486416 (4143.99 per sec.)
queries: 29836992 (49727.85 per sec.)
```

B.182 polardb-150GB-8-read-run-2

```
transactions: 2499809 (4166.31 per sec.)
queries: 29997708 (49995.69 per sec.)
```

B.183 polardb-150GB-8-read-run-3

```
transactions: 2486262 (4143.73 per sec.)
queries: 29835144 (49724.78 per sec.)
```

B.184 polardb-150GB-8-read-run-4

```
transactions: 2501454 (4169.05 per sec.)
queries: 30017448 (50028.63 per sec.)
```

B.185 polardb-150GB-8-read-run-5

```
transactions: 2495410 (4158.98 per sec.)
queries: 29944920 (49907.72 per sec.)
```

B.186 polardb-150GB-16-read-run-1

```
transactions: 4384244 (7306.97 per sec.)
queries: 52610928 (87683.66 per sec.)
```

B.187 polardb-150GB-16-read-run-2

```
transactions: 4422999 (7371.56 per sec.)
queries: 53075988 (88458.74 per sec.)
```

B.188 polardb-150GB-16-read-run-3

```
transactions: 4390492 (7317.38 per sec.)
queries: 52685904 (87808.60 per sec.)
```

B.189 polardb-150GB-16-read-run-4

```
transactions: 4419221 (7365.26 per sec.)
queries: 53030652 (88383.10 per sec.)
```

B.190 polardb-150GB-16-read-run-5

```
transactions: 4418691 (7364.38 per sec.)
queries: 53024292 (88372.58 per sec.)
```

B.191 polardb-150GB-32-read-run-1

```
transactions: 7156728 (11927.61 per sec.)
queries: 85880736 (143131.30 per sec.)
```

B.192 polardb-150GB-32-read-run-2

```
transactions: 7125212 (11875.09 per sec.)
queries: 85502544 (142501.12 per sec.)
```

B.193 polardb-150GB-32-read-run-3

```
transactions: 7211676 (12019.20 per sec.)
queries: 86540112 (144230.34 per sec.)
```

B.194 polardb-150GB-32-read-run-4

```
transactions: 7195627 (11992.45 per sec.)
queries: 86347524 (143909.35 per sec.)
```

B.195 polardb-150GB-32-read-run-5

```
transactions: 7123551 (11872.32 per sec.)
queries: 85482612 (142467.86 per sec.)
```

B.196 polardb-150GB-64-read-run-1

```
transactions: 8497017 (14161.06 per sec.)
queries: 101964204 (169932.73 per sec.)
```

B.197 polardb-150GB-64-read-run-2

```
transactions: 8556074 (14259.48 per sec.)
queries: 102672888 (171113.80 per sec.)
```

B.198 polardb-150GB-64-read-run-3

```
transactions: 8532114 (14219.57 per sec.)
queries: 102385368 (170634.85 per sec.)
```

B.199 polardb-150GB-64-read-run-4

```
transactions: 8489539 (14148.62 per sec.)
queries: 101874468 (169783.48 per sec.)
```

B.200 polardb-150GB-64-read-run-5

```
transactions: 8516196 (14193.00 per sec.)
queries: 102194352 (170316.05 per sec.)
```

B.201 polardb-manythreads-1GB-500-read-run-1

```
transactions: 8343905 (13901.58 per sec.)
queries: 100126860 (166818.97 per sec.)
```

B.202 polardb-manythreads-1GB-500-read-run-2

```
transactions: 8330306 (13878.96 per sec.)
queries: 99963672 (166547.56 per sec.)
```

B.203 polardb-manythreads-1GB-500-read-run-3

```
transactions: 8366140 (13938.75 per sec.)
queries: 100393680 (167265.03 per sec.)
```

B.204 polardb-manythreads-1GB-500-read-run-4

```
transactions: 8394940 (13986.80 per sec.)
queries: 100739280 (167841.61 per sec.)
```

B.205 polardb-manythreads-1GB-500-read-run-5

```
transactions: 8436993 (14056.74 per sec.)
queries: 101243916 (168680.89 per sec.)
```

B.206 mysql-db.r5.large-32-write-run-1

```
transactions: 457173 (761.88 per sec.)
queries: 9144037 (15238.58 per sec.)
```

B.207 mysql-db.r5.large-32-write-run-2

```
transactions: 459900 (766.45 per sec.)
queries: 9198801 (15330.39 per sec.)
```

B.208 mysql-db.r5.large-32-write-run-3

```
transactions: 458713 (764.47 per sec.)
queries: 9174806 (15290.36 per sec.)
```

B.209 mysql-db.r5.large-32-write-run-4

```
transactions: 460530 (767.46 per sec.)
queries: 9211493 (15350.62 per sec.)
```

B.210 mysql-db.r5.large-32-write-run-5

```
transactions: 456208 (760.30 per sec.)
queries: 9124747 (15206.90 per sec.)
```

B.211 mysql-db.r5.xlarge-32-write-run-1

```
transactions: 866061 (1443.34 per sec.)
queries: 17321942 (28867.99 per sec.)
```

B.212 mysql-db.r5.xlarge-32-write-run-2

```
transactions: 865601 (1442.54 per sec.)
queries: 17312893 (28852.25 per sec.)
```

B.213 mysql-db.r5.xlarge-32-write-run-3

```
transactions: 871768 (1452.84 per sec.)
queries: 17436007 (29057.89 per sec.)
```

B.214 mysql-db.r5.xlarge-32-write-run-4

```
transactions: 871895 (1453.06 per sec.)
queries: 17439009 (29063.01 per sec.)
```

B.215 mysql-db.r5.xlarge-32-write-run-5

```
transactions: 881327 (1468.76 per sec.)
queries: 17627201 (29376.38 per sec.)
```

B.216 mysql-db.r5.xlarge-32-write-run-1

```
transactions: 866061 (1443.34 per sec.)
queries: 17321942 (28867.99 per sec.)
```

B.217 mysql-db.r5.xlarge-32-write-run-2

```
transactions: 865601 (1442.54 per sec.)
queries: 17312893 (28852.25 per sec.)
```

B.218 mysql-db.r5.xlarge-32-write-run-3

```
transactions: 871768 (1452.84 per sec.)
queries: 17436007 (29057.89 per sec.)
```

B.219 mysql-db.r5.xlarge-32-write-run-4

```
transactions: 871895 (1453.06 per sec.)
queries: 17439009 (29063.01 per sec.)
```

B.220 mysql-db.r5.xlarge-32-write-run-5

```
transactions: 881327 (1468.76 per sec.)
queries: 17627201 (29376.38 per sec.)
```

B.221 mysql-db.r5.4xlarge-32-write-run-1

```
transactions: 1952418 (3253.83 per sec.)
queries: 39050524 (65080.14 per sec.)
```

B.222 mysql-db.r5.4xlarge-32-write-run-2

```
transactions: 1981335 (3302.02 per sec.)
queries: 39628917 (66044.15 per sec.)
```

B.223 mysql-db.r5.4xlarge-32-write-run-3

```
transactions: 2032445 (3387.20 per sec.)
queries: 40650981 (67747.37 per sec.)
```

B.224 mysql-db.r5.4xlarge-32-write-run-4

```
transactions: 1965038 (3274.18 per sec.)
queries: 39302428 (65486.35 per sec.)
```

B.225 mysql-db.r5.4xlarge-32-write-run-5

```
transactions: 1905262 (3174.89 per sec.)
queries: 38107111 (63501.00 per sec.)
```

B.226 mysql-multiaz-db.r5.large-16-write-run-1

```
transactions: 376951 (628.22 per sec.)
queries: 7539099 (12564.45 per sec.)
```

B.227 mysql-multiaz-db.r5.large-16-write-run-2

```
transactions: 364554 (607.56 per sec.)
queries: 7291342 (12151.60 per sec.)
```

B.228 mysql-multiaz-db.r5.large-16-write-run-3

```
transactions: 352446 (587.27 per sec.)
queries: 7049148 (11745.88 per sec.)
```

B.229 mysql-multiaz-db.r5.large-16-write-run-4

```
transactions: 374083 (623.44 per sec.)
queries: 7481887 (12469.21 per sec.)
```

B.230 mysql-multiaz-db.r5.large-16-write-run-5

```
transactions: 376676 (627.75 per sec.)
queries: 7533677 (12555.28 per sec.)
```

B.231 mysql-multiaz-db.r5.xlarge-16-write-run-1

```
transactions: 443439 (739.02 per sec.)
queries: 8868911 (14780.62 per sec.)
```

B.232 mysql-multiaz-db.r5.xlarge-16-write-run-2

```
transactions: 449827 (749.67 per sec.)
queries: 8996740 (14993.67 per sec.)
```

B.233 mysql-multiaz-db.r5.xlarge-16-write-run-3

```
transactions: 437178 (728.59 per sec.)
queries: 8743847 (14572.26 per sec.)
```

B.234 mysql-multiaz-db.r5.xlarge-16-write-run-4

```
transactions: 452234 (753.68 per sec.)
queries: 9044829 (15073.77 per sec.)
```

B.235 mysql-multiaz-db.r5.xlarge-16-write-run-5

```
transactions: 446740 (744.52 per sec.)
queries: 8934968 (14890.66 per sec.)
```

B.236 mysql-multiaz-db.r5.xlarge-16-write-run-1

```
transactions: 443439 (739.02 per sec.)
queries: 8868911 (14780.62 per sec.)
```

B.237 mysql-multiaz-db.r5.xlarge-16-write-run-2

```
transactions: 449827 (749.67 per sec.)
queries: 8996740 (14993.67 per sec.)
```

B.238 mysql-multiaz-db.r5.xlarge-16-write-run-3

```
transactions: 437178 (728.59 per sec.)
queries: 8743847 (14572.26 per sec.)
```

B.239 mysql-multiaz-db.r5.xlarge-16-write-run-4

```
transactions: 452234 (753.68 per sec.)
queries: 9044829 (15073.77 per sec.)
```

B.240 mysql-multiaz-db.r5.xlarge-16-write-run-5

```
transactions: 446740 (744.52 per sec.)
queries: 8934968 (14890.66 per sec.)
```

B.241 mysql-multiaz-db.r5.4xlarge-16-write-run-1

```
transactions: 828649 (1381.00 per sec.)
queries: 16573659 (27621.13 per sec.)
```

B.242 mysql-multiaz-db.r5.4xlarge-16-write-run-2

```
transactions: 847287 (1412.09 per sec.)
queries: 16946463 (28243.03 per sec.)
```

B.243 mysql-multiaz-db.r5.4xlarge-16-write-run-3

```
transactions: 842589 (1404.26 per sec.)
queries: 16852480 (28086.41 per sec.)
```

B.244 mysql-multiaz-db.r5.4xlarge-16-write-run-4

```
transactions: 825515 (1375.80 per sec.)
queries: 16510782 (27516.78 per sec.)
```

B.245 mysql-multiaz-db.r5.4xlarge-16-write-run-5

```
transactions: 803551 (1339.17 per sec.)
queries: 16071446 (26784.20 per sec.)
```

B.246 mysql-multiaz-db.r5.large-32-write-run-1

```
transactions: 404236 (673.66 per sec.)
queries: 8085333 (13474.21 per sec.)
```

B.247 mysql-multiaz-db.r5.large-32-write-run-2

```
transactions: 412699 (687.77 per sec.)
queries: 8254388 (13755.99 per sec.)
```

B.248 mysql-multiaz-db.r5.large-32-write-run-3

```
transactions: 408288 (680.41 per sec.)
queries: 8166382 (13609.18 per sec.)
```

B.249 mysql-multiaz-db.r5.large-32-write-run-4

```
transactions: 407965 (679.86 per sec.)
queries: 8159920 (13598.29 per sec.)
```

B.250 mysql-multiaz-db.r5.large-32-write-run-5

```
transactions: 409674 (682.72 per sec.)
queries: 8193949 (13655.23 per sec.)
```

B.251 mysql-multiaz-db.r5.xlarge-32-write-run-1

```
transactions: 712448 (1187.30 per sec.)
queries: 14249412 (23746.81 per sec.)
```

B.252 mysql-multiaz-db.r5.xlarge-32-write-run-2

```
transactions: 719549 (1199.14 per sec.)
queries: 14391448 (23983.48 per sec.)
```

B.253 mysql-multiaz-db.r5.xlarge-32-write-run-3

```
transactions: 724611 (1207.57 per sec.)
queries: 14492865 (24152.41 per sec.)
```

B.254 mysql-multiaz-db.r5.xlarge-32-write-run-4

```
transactions: 718041 (1196.60 per sec.)
queries: 14361453 (23933.15 per sec.)
```

B.255 mysql-multiaz-db.r5.xlarge-32-write-run-5

```
transactions: 725442 (1208.93 per sec.)
queries: 14509390 (24179.59 per sec.)
```

B.256 mysql-multiaz-db.r5.xlarge-32-write-run-1

```
transactions: 712448 (1187.30 per sec.)
queries: 14249412 (23746.81 per sec.)
```

B.257 mysql-multiaz-db.r5.xlarge-32-write-run-2

```
transactions: 719549 (1199.14 per sec.)
queries: 14391448 (23983.48 per sec.)
```

B.258 mysql-multiaz-db.r5.xlarge-32-write-run-3

```
transactions: 724611 (1207.57 per sec.)
queries: 14492865 (24152.41 per sec.)
```

B.259 mysql-multiaz-db.r5.xlarge-32-write-run-4

```
transactions: 718041 (1196.60 per sec.)
queries: 14361453 (23933.15 per sec.)
```

B.260 mysql-multiaz-db.r5.xlarge-32-write-run-5

```
transactions: 725442 (1208.93 per sec.)
queries: 14509390 (24179.59 per sec.)
```

B.261 mysql-multiaz-db.r5.4xlarge-32-write-run-1

```
transactions: 1309751 (2182.77 per sec.)
queries: 26196797 (43658.43 per sec.)
```

B.262 mysql-multiaz-db.r5.4xlarge-32-write-run-2

```
transactions: 1295922 (2159.71 per sec.)
queries: 25920176 (43197.03 per sec.)
```

B.263 mysql-multiaz-db.r5.4xlarge-32-write-run-3

```
transactions: 1326942 (2211.44 per sec.)
queries: 26540579 (44231.60 per sec.)
```

B.264 mysql-multiaz-db.r5.4xlarge-32-write-run-4

```
transactions: 1312488 (2187.32 per sec.)
queries: 26251536 (43749.27 per sec.)
```

B.265 mysql-multiaz-db.r5.4xlarge-32-write-run-5

```
transactions: 1270827 (2117.88 per sec.)
queries: 25418413 (42360.81 per sec.)
```

B.266 mysql-multiaz-db.r5.large-64-write-run-1

```
transactions: 404865 (674.62 per sec.)
queries: 8098160 (13493.89 per sec.)
```

B.267 mysql-multiaz-db.r5.large-64-write-run-2

```
transactions: 403392 (672.14 per sec.)
queries: 8068600 (13444.16 per sec.)
```

B.268 mysql-multiaz-db.r5.large-64-write-run-3

```
transactions: 404237 (673.58 per sec.)
queries: 8085845 (13473.45 per sec.)
```

B.269 mysql-multiaz-db.r5.large-64-write-run-4

```
transactions: 404921 (674.63 per sec.)
queries: 8099347 (13494.12 per sec.)
```

B.270 mysql-multiaz-db.r5.large-64-write-run-5

```
transactions: 409387 (682.17 per sec.)
queries: 8188740 (13645.11 per sec.)
```

B.271 mysql-multiaz-db.r5.xlarge-64-write-run-1

```
transactions: 841621 (1402.41 per sec.)
queries: 16833821 (28050.49 per sec.)
```

B.272 mysql-multiaz-db.r5.xlarge-64-write-run-2

```
transactions: 838945 (1397.98 per sec.)
queries: 16780466 (27962.14 per sec.)
```

B.273 mysql-multiaz-db.r5.xlarge-64-write-run-3

```
transactions: 838189 (1396.71 per sec.)
queries: 16765072 (27936.45 per sec.)
```

B.274 mysql-multiaz-db.r5.xlarge-64-write-run-4

```
transactions: 835151 (1391.63 per sec.)
queries: 16704500 (27835.15 per sec.)
```

B.275 mysql-multiaz-db.r5.xlarge-64-write-run-5

```
transactions: 829095 (1381.33 per sec.)
queries: 16583368 (27629.11 per sec.)
```

B.276 mysql-multiaz-db.r5.xlarge-64-write-run-1

```
transactions: 841621 (1402.41 per sec.)
queries: 16833821 (28050.49 per sec.)
```

B.277 mysql-multiaz-db.r5.xlarge-64-write-run-2

```
transactions: 838945 (1397.98 per sec.)
queries: 16780466 (27962.14 per sec.)
```

B.278 mysql-multiaz-db.r5.xlarge-64-write-run-3

```
transactions: 838189 (1396.71 per sec.)
queries: 16765072 (27936.45 per sec.)
```

B.279 mysql-multiaz-db.r5.xlarge-64-write-run-4

```
transactions: 835151 (1391.63 per sec.)
queries: 16704500 (27835.15 per sec.)
```

B.280 mysql-multiaz-db.r5.xlarge-64-write-run-5

```
transactions: 829095 (1381.33 per sec.)
queries: 16583368 (27629.11 per sec.)
```

B.281 mysql-multiaz-db.r5.4xlarge-64-write-run-1

```
transactions: 2174491 (3623.69 per sec.)
queries: 43492938 (72479.03 per sec.)
```

B.282 mysql-multiaz-db.r5.4xlarge-64-write-run-2

```
transactions: 2159299 (3598.38 per sec.)
queries: 43189200 (71972.88 per sec.)
```

B.283 mysql-multiaz-db.r5.4xlarge-64-write-run-3

```
transactions: 2159394 (3598.53 per sec.)
queries: 43190604 (71975.05 per sec.)
```

B.284 mysql-multiaz-db.r5.4xlarge-64-write-run-4

```
transactions: 2184641 (3640.59 per sec.)
queries: 43695636 (72816.54 per sec.)
```

B.285 mysql-multiaz-db.r5.4xlarge-64-write-run-5

```
transactions: 2167996 (3612.85 per sec.)
queries: 43362731 (72261.61 per sec.)
```

B.286 aurora-db.r5.large-16-write-run-1

```
transactions: 325650 (542.70 per sec.)
queries: 6513292 (10854.56 per sec.)
```

B.287 aurora-db.r5.large-16-write-run-2

```
transactions: 324600 (540.97 per sec.)
queries: 6492257 (10819.75 per sec.)
```

B.288 aurora-db.r5.large-16-write-run-3

```
transactions: 328289 (547.12 per sec.)
queries: 6566133 (10943.06 per sec.)
```

B.289 aurora-db.r5.large-16-write-run-4

```
transactions: 327796 (546.30 per sec.)
queries: 6556206 (10926.38 per sec.)
```

B.290 aurora-db.r5.large-16-write-run-5

```
transactions: 328637 (547.70 per sec.)
queries: 6573031 (10954.49 per sec.)
```

B.291 aurora-db.r5.xlarge-16-write-run-1

```
transactions: 502008 (836.62 per sec.)
queries: 10041003 (16733.82 per sec.)
```

B.292 aurora-db.r5.xlarge-16-write-run-2

```
transactions: 502198 (836.95 per sec.)
queries: 10044565 (16739.99 per sec.)
```

B.293 aurora-db.r5.xlarge-16-write-run-3

```
transactions: 506690 (844.44 per sec.)
queries: 10134658 (16890.22 per sec.)
```

B.294 aurora-db.r5.xlarge-16-write-run-4

```
transactions: 507430 (845.66 per sec.)
queries: 10149407 (16914.53 per sec.)
```

B.295 aurora-db.r5.xlarge-16-write-run-5

```
transactions: 508919 (848.14 per sec.)
queries: 10179177 (16964.14 per sec.)
```

B.296 aurora-db.r5.2xlarge-16-write-run-1

```
transactions: 561237 (935.36 per sec.)
queries: 11225462 (18708.44 per sec.)
```

B.297 aurora-db.r5.2xlarge-16-write-run-2

```
transactions: 562713 (937.81 per sec.)
queries: 11255045 (18757.52 per sec.)
```

B.298 aurora-db.r5.2xlarge-16-write-run-3

```
transactions: 563264 (938.74 per sec.)
queries: 11266139 (18776.15 per sec.)
```

B.299 aurora-db.r5.2xlarge-16-write-run-4

```
transactions: 564229 (940.34 per sec.)
queries: 11285422 (18808.28 per sec.)
```

B.300 aurora-db.r5.2xlarge-16-write-run-5

```
transactions: 565302 (942.12 per sec.)
queries: 11306764 (18843.56 per sec.)
```

B.301 aurora-db.r5.4xlarge-16-write-run-1

```
transactions: 956765 (1594.55 per sec.)
queries: 19136691 (31893.23 per sec.)
```

B.302 aurora-db.r5.4xlarge-16-write-run-2

```
transactions: 958215 (1596.96 per sec.)
queries: 19165665 (31941.43 per sec.)
```

B.303 aurora-db.r5.4xlarge-16-write-run-3

```
transactions: 963499 (1605.78 per sec.)
queries: 19271327 (32117.75 per sec.)
```

B.304 aurora-db.r5.4xlarge-16-write-run-4

```
transactions: 962242 (1603.67 per sec.)
queries: 19246097 (32075.49 per sec.)
```

B.305 aurora-db.r5.4xlarge-16-write-run-5

```
transactions: 971385 (1618.91 per sec.)
queries: 19429186 (32380.66 per sec.)
```

B.306 aurora-db.r5.large-32-write-run-1

```
transactions: 385512 (642.38 per sec.)
queries: 7711138 (12849.19 per sec.)
```

B.307 aurora-db.r5.large-32-write-run-2

```
transactions: 382939 (638.15 per sec.)
queries: 7659722 (12764.52 per sec.)
```

B.308 aurora-db.r5.large-32-write-run-3

```
transactions: 387972 (646.55 per sec.)
queries: 7760371 (12932.60 per sec.)
```

B.309 aurora-db.r5.large-32-write-run-4

```
transactions: 387111 (645.10 per sec.)
queries: 7742994 (12903.37 per sec.)
```

B.310 aurora-db.r5.large-32-write-run-5

```
transactions: 391468 (652.38 per sec.)
queries: 7830413 (13049.30 per sec.)
```

B.311 aurora-db.r5.xlarge-32-write-run-1

```
transactions: 655644 (1092.64 per sec.)
queries: 13115382 (21856.89 per sec.)
```

B.312 aurora-db.r5.xlarge-32-write-run-2

```
transactions: 659793 (1099.51 per sec.)
queries: 13197667 (21993.30 per sec.)
```

B.313 aurora-db.r5.xlarge-32-write-run-3

```
transactions: 659003 (1098.20 per sec.)
queries: 13181935 (21967.13 per sec.)
```

B.314 aurora-db.r5.xlarge-32-write-run-4

```
transactions: 652358 (1087.15 per sec.)
queries: 13049395 (21746.77 per sec.)
```

B.315 aurora-db.r5.xlarge-32-write-run-5

```
transactions: 650250 (1083.63 per sec.)
queries: 13006688 (21675.41 per sec.)
```

B.316 aurora-db.r5.2xlarge-32-write-run-1

```
transactions: 987055 (1644.97 per sec.)
queries: 19743810 (32904.00 per sec.)
```

B.317 aurora-db.r5.2xlarge-32-write-run-2

```
transactions: 992079 (1653.32 per sec.)
queries: 19844522 (33071.35 per sec.)
```

B.318 aurora-db.r5.2xlarge-32-write-run-3

```
transactions: 991867 (1652.97 per sec.)
queries: 19840099 (33064.03 per sec.)
```

B.319 aurora-db.r5.2xlarge-32-write-run-4

```
transactions: 992474 (1653.99 per sec.)
queries: 19852014 (33084.01 per sec.)
```

B.320 aurora-db.r5.2xlarge-32-write-run-5

```
transactions: 985782 (1642.84 per sec.)
queries: 19718679 (32861.87 per sec.)
```

B.321 aurora-db.r5.4xlarge-32-write-run-1

```
transactions: 1539589 (2565.78 per sec.)
queries: 30795602 (51321.92 per sec.)
```

B.322 aurora-db.r5.4xlarge-32-write-run-2

```
transactions: 1541574 (2569.13 per sec.)
queries: 30835278 (51388.91 per sec.)
```

B.323 aurora-db.r5.4xlarge-32-write-run-3

```
transactions: 1541662 (2569.23 per sec.)
queries: 30836465 (51390.00 per sec.)
```

B.324 aurora-db.r5.4xlarge-32-write-run-4

```
transactions: 1541631 (2569.20 per sec.)
queries: 30836371 (51390.21 per sec.)
```

B.325 aurora-db.r5.4xlarge-32-write-run-5

```
transactions: 1539110 (2565.01 per sec.)
queries: 30785683 (51305.95 per sec.)
```

B.326 aurora-db.r5.large-64-write-run-1

```
transactions: 400639 (667.55 per sec.)
queries: 8014499 (13353.86 per sec.)
```

B.327 aurora-db.r5.large-64-write-run-2

```
transactions: 404125 (673.41 per sec.)
queries: 8084224 (13471.06 per sec.)
```

B.328 aurora-db.r5.large-64-write-run-3

```
transactions: 398920 (664.72 per sec.)
queries: 7980207 (13297.40 per sec.)
```

B.329 aurora-db.r5.large-64-write-run-4

```
transactions: 402554 (670.72 per sec.)
queries: 8052828 (13417.24 per sec.)
```

B.330 aurora-db.r5.large-64-write-run-5

```
transactions: 403022 (671.57 per sec.)
queries: 8062129 (13434.12 per sec.)
```

B.331 aurora-db.r5.xlarge-64-write-run-1

```
transactions: 774612 (1290.71 per sec.)
queries: 15495699 (25819.88 per sec.)
```

B.332 aurora-db.r5.xlarge-64-write-run-2

```
transactions: 779653 (1299.13 per sec.)
queries: 15596175 (25987.86 per sec.)
```

B.333 aurora-db.r5.xlarge-64-write-run-3

```
transactions: 781060 (1301.51 per sec.)
queries: 15624965 (26036.51 per sec.)
```

B.334 aurora-db.r5.xlarge-64-write-run-4

```
transactions: 780848 (1301.17 per sec.)
queries: 15620424 (26029.23 per sec.)
```

B.335 aurora-db.r5.xlarge-64-write-run-5

```
transactions: 781239 (1301.81 per sec.)
queries: 15628292 (26042.03 per sec.)
```

B.336 aurora-db.r5.2xlarge-64-write-run-1

```
transactions: 1307094 (2178.20 per sec.)
queries: 26144516 (43568.46 per sec.)
```

B.337 aurora-db.r5.2xlarge-64-write-run-2

```
transactions: 1307479 (2178.83 per sec.)
queries: 26152289 (43581.02 per sec.)
```

B.338 aurora-db.r5.2xlarge-64-write-run-3

```
transactions: 1310803 (2184.34 per sec.)
queries: 26219279 (43692.24 per sec.)
```

B.339 aurora-db.r5.2xlarge-64-write-run-4

```
transactions: 1309273 (2181.84 per sec.)
queries: 26188020 (43640.99 per sec.)
```

B.340 aurora-db.r5.2xlarge-64-write-run-5

```
transactions: 1311042 (2184.81 per sec.)
queries: 26223572 (43700.66 per sec.)
```

B.341 aurora-db.r5.4xlarge-64-write-run-1

```
transactions: 2153272 (3588.26 per sec.)
queries: 43072058 (71776.30 per sec.)
```

B.342 aurora-db.r5.4xlarge-64-write-run-2

```
transactions: 2147448 (3578.62 per sec.)
queries: 42956003 (71584.19 per sec.)
```

B.343 aurora-db.r5.4xlarge-64-write-run-3

```
transactions: 2150415 (3583.55 per sec.)
queries: 43015569 (71683.18 per sec.)
```

B.344 aurora-db.r5.4xlarge-64-write-run-4

```
transactions: 2149513 (3582.04 per sec.)
queries: 42997084 (71652.10 per sec.)
```

B.345 aurora-db.r5.4xlarge-64-write-run-5

```
transactions: 2140786 (3567.48 per sec.)
queries: 42822844 (71361.53 per sec.)
```

B.346 polardb-manythreads-2-write-run-1

```
transactions: 451419 (752.35 per sec.)
queries: 2708514 (4514.12 per sec.)
```

B.347 polardb-manythreads-2-write-run-2

```
transactions: 454478 (757.46 per sec.)
queries: 2726868 (4544.74 per sec.)
```

B.348 polardb-manythreads-2-write-run-3

```
transactions: 446212 (743.68 per sec.)
queries: 2677272 (4462.08 per sec.)
```

B.349 polardb-manythreads-2-write-run-4

```
transactions: 448781 (747.96 per sec.)
queries: 2692686 (4487.75 per sec.)
```

B.350 polardb-manythreads-2-write-run-5

```
transactions: 452924 (754.86 per sec.)
queries: 2717544 (4529.16 per sec.)
```

B.351 polardb-manythreads-4-write-run-1

```
transactions: 804149 (1340.23 per sec.)
queries: 4824894 (8041.40 per sec.)
```

B.352 polardb-manythreads-4-write-run-2

```
transactions: 806172 (1343.60 per sec.)
queries: 4837032 (8061.62 per sec.)
```

B.353 polardb-manythreads-4-write-run-3

```
transactions: 805903 (1343.16 per sec.)
queries: 4835418 (8058.93 per sec.)
```

B.354 polardb-manythreads-4-write-run-4

```
transactions: 797450 (1329.06 per sec.)
queries: 4784700 (7974.38 per sec.)
```

B.355 polardb-manythreads-4-write-run-5

```
transactions: 804696 (1341.14 per sec.)
queries: 4828176 (8046.86 per sec.)
```

B.356 polardb-manythreads-8-write-run-1

```
transactions: 1496709 (2494.47 per sec.)
queries: 8980254 (14966.84 per sec.)
```

B.357 polardb-manythreads-8-write-run-2

```
transactions: 1512707 (2521.12 per sec.)
queries: 9076242 (15126.71 per sec.)
```

B.358 polardb-manythreads-8-write-run-3

```
transactions: 1493245 (2488.69 per sec.)
queries: 8959470 (14932.15 per sec.)
```

B.359 polardb-manythreads-8-write-run-4

```
transactions: 1500650 (2501.02 per sec.)
queries: 9003900 (15006.13 per sec.)
```

B.360 polardb-manythreads-8-write-run-5

```
transactions: 1508033 (2513.33 per sec.)
queries: 9048198 (15079.95 per sec.)
```

B.361 polardb-manythreads-16-write-run-1

```
transactions: 2764963 (4608.12 per sec.)
queries: 16589778 (27648.71 per sec.)
```

B.362 polardb-manythreads-16-write-run-2

```
transactions: 2772253 (4620.25 per sec.)
queries: 16633518 (27721.53 per sec.)
```

B.363 polardb-manythreads-16-write-run-3

```
transactions: 2820359 (4700.43 per sec.)
queries: 16922154 (28202.55 per sec.)
```

B.364 polardb-manythreads-16-write-run-4

```
transactions: 2848370 (4747.10 per sec.)
queries: 17090220 (28482.62 per sec.)
```

B.365 polardb-manythreads-16-write-run-5

```
transactions: 2850690 (4750.99 per sec.)
queries: 17104140 (28505.94 per sec.)
```

B.366 polardb-manythreads-32-write-run-1

```
transactions: 4845779 (8075.82 per sec.)
queries: 29074674 (48454.91 per sec.)
```

B.367 polardb-manythreads-32-write-run-2

```
transactions: 4835186 (8058.10 per sec.)
queries: 29011116 (48348.62 per sec.)
```

B.368 polardb-manythreads-32-write-run-3

```
transactions: 4845852 (8075.95 per sec.)
queries: 29075112 (48455.70 per sec.)
```

B.369 polardb-manythreads-32-write-run-4

```
transactions: 4853880 (8088.68 per sec.)
queries: 29123280 (48532.10 per sec.)
```

B.370 polardb-manythreads-32-write-run-5

```
transactions: 4834847 (8057.55 per sec.)
queries: 29009082 (48345.31 per sec.)
```

B.371 polardb-manythreads-64-write-run-1

```
transactions: 7097935 (11828.52 per sec.)
queries: 42587610 (70971.10 per sec.)
```

B.372 polardb-manythreads-64-write-run-2

```
transactions: 7050196 (11748.90 per sec.)
queries: 42301176 (70493.41 per sec.)
```

B.373 polardb-manythreads-64-write-run-3

```
transactions: 7062840 (11769.97 per sec.)
queries: 42377040 (70619.83 per sec.)
```

B.374 polardb-manythreads-64-write-run-4

```
transactions: 7116718 (11859.75 per sec.)
queries: 42700308 (71158.49 per sec.)
```

B.375 polardb-manythreads-64-write-run-5

```
transactions: 7107383 (11844.21 per sec.)
queries: 42644300 (71065.24 per sec.)
```

B.376 polardb-150GB-2-write-run-1

```
transactions: 381864 (636.43 per sec.)
queries: 2291184 (3818.60 per sec.)
```

B.377 polardb-150GB-2-write-run-2

```
transactions: 371589 (619.31 per sec.)
queries: 2229534 (3715.83 per sec.)
```

B.378 polardb-150GB-2-write-run-3

```
transactions: 377432 (629.04 per sec.)
queries: 2264592 (3774.27 per sec.)
```

B.379 polardb-150GB-2-write-run-4

```
transactions: 387651 (646.08 per sec.)
queries: 2325906 (3876.48 per sec.)
```

B.380 polardb-150GB-2-write-run-5

```
transactions: 394853 (658.08 per sec.)
queries: 2369118 (3948.49 per sec.)
```

B.381 polardb-150GB-4-write-run-1

```
transactions: 702833 (1171.37 per sec.)
queries: 4216998 (7028.24 per sec.)
```

B.382 polardb-150GB-4-write-run-2

```
transactions: 708522 (1180.85 per sec.)
queries: 4251132 (7085.11 per sec.)
```

B.383 polardb-150GB-4-write-run-3

```
transactions: 720585 (1200.96 per sec.)
queries: 4323510 (7205.74 per sec.)
```

B.384 polardb-150GB-4-write-run-4

```
transactions: 670219 (1117.01 per sec.)
queries: 4021314 (6702.09 per sec.)
```

B.385 polardb-150GB-4-write-run-5

```
transactions: 673649 (1122.74 per sec.)
queries: 4041894 (6736.42 per sec.)
```

B.386 polardb-150GB-8-write-run-1

```
transactions: 1250839 (2084.68 per sec.)
queries: 7505034 (12508.09 per sec.)
```

B.387 polardb-150GB-8-write-run-2

```
transactions: 1211925 (2019.84 per sec.)
queries: 7271550 (12119.02 per sec.)
```

B.388 polardb-150GB-8-write-run-3

```
transactions: 1372472 (2287.41 per sec.)
queries: 8234832 (13724.47 per sec.)
```

B.389 polardb-150GB-8-write-run-4

```
transactions: 1293754 (2156.21 per sec.)
queries: 7762524 (12937.25 per sec.)
```

B.390 polardb-150GB-8-write-run-5

```
transactions: 1231165 (2051.75 per sec.)
queries: 7386990 (12310.51 per sec.)
```

B.391 polardb-150GB-16-write-run-1

```
transactions: 2065902 (3443.05 per sec.)
queries: 12395412 (20658.32 per sec.)
```

B.392 polardb-150GB-16-write-run-2

```
transactions: 2096504 (3492.65 per sec.)
queries: 12579024 (20955.87 per sec.)
```

B.393 polardb-150GB-16-write-run-3

```
transactions: 2067163 (3444.85 per sec.)
queries: 12402978 (20669.09 per sec.)
```

B.394 polardb-150GB-16-write-run-4

```
transactions: 2105095 (3508.36 per sec.)
queries: 12630570 (21050.14 per sec.)
```

B.395 polardb-150GB-16-write-run-5

```
transactions: 2114775 (3524.51 per sec.)
queries: 12688650 (21147.07 per sec.)
```

B.396 polardb-150GB-32-write-run-1

```
transactions: 2797575 (4658.04 per sec.)
queries: 16785450 (27948.25 per sec.)
```

B.397 polardb-150GB-32-write-run-2

```
transactions: 2774162 (4622.19 per sec.)
queries: 16644972 (27733.16 per sec.)
```

B.398 polardb-150GB-32-write-run-3

```
transactions: 2835099 (4715.84 per sec.)
queries: 17010594 (28295.06 per sec.)
```

B.399 polardb-150GB-32-write-run-4

```
transactions: 2863422 (4772.07 per sec.)
queries: 17180532 (28632.43 per sec.)
```

B.400 polardb-150GB-32-write-run-5

```
transactions: 2843491 (4738.87 per sec.)
queries: 17060946 (28433.22 per sec.)
```

B.401 polardb-150GB-64-write-run-1

```
transactions: 3914662 (6521.90 per sec.)
queries: 23487972 (39131.40 per sec.)
```

B.402 polardb-150GB-64-write-run-2

```
transactions: 4259803 (7098.78 per sec.)
queries: 25558818 (42592.70 per sec.)
```

B.403 polardb-150GB-64-write-run-3

```
transactions: 4681078 (7797.84 per sec.)
queries: 28086468 (46787.06 per sec.)
```

B.404 polardb-150GB-64-write-run-4

```
transactions: 4937870 (8227.82 per sec.)
queries: 29627220 (49366.93 per sec.)
```

B.405 polardb-150GB-64-write-run-5

```
transactions: 4973430 (8280.02 per sec.)
queries: 29840580 (49680.10 per sec.)
```

B.406 polardb-manythreads-1GB-500-write-run-1

```
transactions: 7898510 (13151.63 per sec.)
queries: 47391062 (78909.81 per sec.)
```

B.407 polardb-manythreads-1GB-500-write-run-2

```
transactions: 7990975 (13305.60 per sec.)
queries: 47945850 (79833.59 per sec.)
```

B.408 polardb-manythreads-1GB-500-write-run-3

```
transactions: 7972243 (13274.37 per sec.)
queries: 47833458 (79646.23 per sec.)
```

B.409 polardb-manythreads-1GB-500-write-run-4

```
transactions: 8007720 (13333.30 per sec.)
queries: 48046320 (79999.80 per sec.)
```

B.410 polardb-manythreads-1GB-500-write-run-5

```
transactions: 8002117 (13323.77 per sec.)
queries: 48012705 (79942.60 per sec.)
```

References

[1] Alexandre Verbitski et. al. *Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases*. SIGMOD, 2017.

[2] Alexandre Verbitski et. al. *Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes*. SIGMOD, 2018.

[3] George Coulouris et. al. *Distributed Systems - Concepts and Design 5th ed.* Pearson, 2011.

[4] Hau Du et. al. *Multi-Paxos: An Implementation and Evaluation*. University of Washington,

[5] James C. Corbett et. al. *Spanner: Google's Globally-Distributed Database*. OSDI, 2012.

[6] Sapna Jain et. al. *Comparative Study of Traditional Database and Cloud Computing Database*. International Journal of Advanced Research in Computer Science, 2017.

[7] Wei Cao et. al. *PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database*. VLDB, 2018.

[8] Alexey Kopytov. *sysbench - Scriptable database and system performance benchmark*. May 2019. URL: `https://github.com/akopytov/sysbench`.

[9] Alibaba Cloud Docs. *Test methods - concept_ ebd_ 5gv_ tdb .concept*. June 2018. URL: `https://github.com/AlibabaCloudDocs/polardb/blob/master/cn.zh-CN/%5C%E6%5C%80%5C%A7%5C%E8%5C%83%5C%BD%5C%E7%5C%99%5C%BD%5C%E7%5C%9A%5C%AE%5C%E4%5C%B9%5C%A6/%5C%E6%5C%B5%5C%8B%5C%E8%5C%AF%5C%95%5C%E6%5C%96%5C%B9%5C%E6%5C%B3%5C%95.md`.

[10] Alibaba Clouder. *PolarDB: Deep Dive on Alibaba Cloud's Next-Generation Database*. Apr. 2018. URL: `https://www.alibabacloud.com/blog/deep-dive-on-alibaba-clouds-next-generation-database_578138`.

[11] Alibaba Tech. *Alibaba Unveils PolarFS Distributed File System for Cloud Computing*. Aug. 2018. URL: `https://hackernoon.com/alibaba-unveils-new-distributed-file-system-6bade3ad0413`.

[12] Amazon AWS. *Sign Up for the Preview of Amazon Aurora Multi-Master*. Dec. 2018. URL: `https://aws.amazon.com/about-aws/whats-new/2017/11/sign-up-for-the-preview-of-amazon-aurora-multi-master/`.

[13] Amazon AWS FAQ. *Amazon Aurora Storage and Reliability*. Dec. 2018. URL: `https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.StorageReliability.html`.

[14] Amazon AWS FAQ. *Amazon AWS: Database Engine Versions*. Dec. 2018. URL: `https://aws.amazon.com/rds/faqs/`.

[15] Amazon AWS FAQ. *Replicating Amazon Aurora MySQL DB Clusters Across AWS Regions*. Dec. 2018. URL: `https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Replication.CrossRegion.html`.

[16] Amazon AWS Team. *Amazon RDS DB Instance Storage*. June 2019. URL: `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html`.

[17] Amazon AWS Team. *Amazon RDS DB Instance Storage*. May 2019. URL: `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html#Concepts.Storage.GeneralSSD`.

[18] Amazon AWS Team. *Amazon RDS Multi-AZ Deployments*. June 2019. URL: `https://aws.amazon.com/rds/details/multi-az/`.

[19] Amazon AWS Team. *Amazon RDS Read Replicas Now Support Multi-AZ Deployments*. Jan. 2018. URL: `https://aws.amazon.com/about-aws/whats-new/2018/01/amazon-rds-read-replicas-now-support-multi-az-deployments/`.

[20] Amazon AWS Team. *Amazon Relational Database Service (RDS)*. June 2019. URL: `https://aws.amazon.com/rds/`.

[21] Amazon AWS Team. *Regions and Availability Zones*. May 2019. URL: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`.

[22] Amazon AWS Team. *What Is Amazon Relational Database Service (Amazon RDS)?* June 2019. URL: `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html`.

[23] Ben Stopford. *Shared Nothing v.s. Shared Disk Architectures: An Independent View*. Nov. 2009. URL: `http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/`.

[24] BMC. *SaaS vs PaaS vs IaaS: What's The Difference and How To Choose*. Dec. 2018. URL: `https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/`.

[25] Eric Brewer. *CAP Twelve Years Later: How the "Rules" Have Changed*. IEEE Computer, 2012.

[26] Brian Storti. *Raft: Consensus made simple(r)*. Oct. 2017. URL: `https://www.brianstorti.com/raft/`.

[27] Daniel Abadi. *NewSQL database systems are failing to guarantee consistency and I blame Spanner*. Sept. 2018. URL: `http://dbmsmusings.blogspot.com/2018/09/newsql-database-systems-are-failing-to.html`.

[28] Diego Ongaro. *The Raft Consensus Algorithm*. Oct. 2014. URL: `https://ramcloud.stanford.edu/~ongaro/cs244b.pdf`.

[29] John Ousterhout Diego Ongaro. *In Search of an Understandable Consensus Algorithm (Extended Version)*. Stanford University, 2014.

[30] HashiCorp. *HashiCorp Terraform - Write, Plan, and Create Infrastructure as Code.* May 2019. URL: `https://www.terraform.io/`.

[31] Trond Humborstad. *Database and storage layer integration for cloud platforms.* NTNU, 2018.

[32] IBM. *What is the cloud?* Dec. 2018. URL: `https://www.ibm.com/cloud/learn/what-is-cloud-computing`.

[33] M. Stonebraker J. Hellerstein. *Architecture of a Database System.* NOW, 2007.

[34] John Gemignani. *Amazon RDS Under the Hood: Multi-AZ.* Dec. 2017. URL: `https://aws.amazon.com/blogs/database/amazon-rds-under-the-hood-multi-az/`.

[35] John Gemignani. *Amazon RDS Under the Hood: Multi-AZ.* Dec. 2017. URL: `https://aws.amazon.com/blogs/database/amazon-rds-under-the-hood-multi-az/`.

[36] John White. *Private vs. Public Cloud: What's the Difference?* Oct. 2018. URL: `https://www.expedient.com/blog/private-vs-public-cloud-whats-difference/`.

[37] Leslie Lamport. *Paxos made simple).* Stanford University, 2001.

[38] Lixun Peng, Inaam Rana. *POLARDB: InnoDB based shared-everything storage solution.* Apr. 2018. URL: `https://www.percona.com/live/18/sessions/polardb-innodb-based-shared-everything-storage-solution`.

[39] Lixun Peng, Inaam Rana. *YouTube - POLARDB: InnoDB based shared-everything storage solution.* Apr. 2018. URL: `https://www.youtube.com/watch?v=bjKwFmqGv7U`.

[40] Mark Callaghan. *Transaction Processing in NewSQL.* Oct. 2018. URL: `http://smalldatum.blogspot.com/2018/10/transaction-processing-in-newsql.html`.

[41] MinervaDB Corporation. *Benchmarking MySQL 5.7 using Sysbench 1.1.* May 2019. URL: `https://minervadb.com/index.php/2018/03/13/benchmarking-mysql-using-sysbench-1-1/`.

[42] Elmasri & Navathe. *Fundamentals of Database Systems, 7th ed.* Pearson, 2016.

[43] Nicole Hemsoth. *Alibaba rolls own distributed file system for cloud database performance.* Aug. 2018. URL: `https://www.nextplatform.com/2018/08/21/alibaba-rolls-own-distributed-file-system-for-cloud-database-performance/`.

[44] Phil Intihar. *Understanding Burst vs. Baseline Performance with Amazon RDS and GP2.* July 2017. URL: `https://aws.amazon.com/blogs/database/understanding-burst-vs-baseline-performance-with-amazon-rds-and-gp2/`.

[45] Dale Skeen. *A quorumbased commit protocol.* Cornell University, 1982.

[46] Tom Cocagne. *Understanding Paxos.* Dec. 2018. URL: `https://understandingpaxos.wordpress.com/`.

[47] Wesley Wilk, David Gardner. *Amazon RDS Under the Hood: Single-AZ instance recovery.* Dec. 2018. URL: `https://aws.amazon.com/blogs/database/amazon-rds-under-the-hood-single-az-instance-recovery/`.