Assignment 3 Report

Task (a.2)

For JobSchedulingLatency and MaxTaskCompletionTimeFromKafka, I use ValueState. Because this two class want to calculate the latency, which means to compute the gap that the first time event type transfer from SUBMIT to SCHEDULE, so I store the whole event object in ValueState and to check when the event type changes, if not just update the state. We can use MapState for this two classes. The key is jobid and value is event type. The logic is almost same with ValueState: Update map if not match. if matched, compute the latency. The advantage is that it's easy to store and calculate. We only have one object which needed to observe and compute. The disadvantage is that it is stored as a object but not a primitive type, which can cost more space and overhead, and the job and task event object indeed has many variables.

For BusyMachines, I use ListState. The logic is that after we keyby windowEnd, we want to use all previous objects to calculate the busiest machines, so we need to store all previous objects in the window. That's why ListState comes. For LongestSessionPerJob, the logic is almost the same: in order to compute max session per job, after we compute session per job, we need to store all these sessionJob object so that we can compare these object to calculate max session per job by connecting another key stream. I think we can use ReduceState instead of ListState. ReduceState can aggregate and compute on all previous objects every time instead of compute it at one time using ListState. The advantage for ListState is that it's quite straightforward. The disadvantage is it wastes many spaces when compared to ReduceState and lower the performance since it needs store all objects in the window.

Task (b.1)

For the data I collected, I take the median of all latency data in the log files each histogram.

Task	Parallelism	State Backend	Min	Max	p50	p99
JobSchedulingLatency	1	filesystem	0	104	47.5	103.43
	1	rocksdb	1	101	45.5	100.71
	2	filesystem	8.5	150.5	98.5	150.5
	2	rocksdb	6	174	73	174
	4	filesystem	10	173	91.25	172.5
	4	rocksdb	13	160	86.5	160
BusyMachines	1	filesystem	4	165	45	165
	1	rocksdb	13.5	1023	36.5	1023
	2	filesystem	5	136	47	136
	2	rocksdb	4	448	45	448
	4	filesystem	3	105	24.5	105
	4	rocksdb	1	125	3	125

LongestSessionPerJob	1	filesystem	0	66	2	66
	1	rocksdb	2	261	21.5	261
	2	filesystem	2	131	40	142
	2	rocksdb	4	254	18.5	254
	4	filesystem	7.5	186	51	186
	4	rocksdb	9	252	42	252
PerMachineTaskStatistics	1	filesystem	0	59	2	57.7
	1	rocksdb	1	216	12	215.1
	2	filesystem	1	103.5	6	103.5
	2	rocksdb	2	145	20.5	145
	4	filesystem	2	63	22.5	63
	4	rocksdb	3	75	29	75

Task (b.2)

1. Question: Does the latency decrease when you increase the parallelism? Why (not)?

In the view of my data, I think latency increase when you increase the parallelism.
This was a surprise for me, as we expected that higher parallelism will derive better latency. But why this happens is reasonable. It is because:
(1) The size of the data we compute is too small, 90MB data in job events and 1.8GB data in task events are a small figure. We compute the data normally between 30s to 1min. So it's hard to make difference on latency in such a short time. We can receive convincing result only on computing the large size of the data.
(2) Since the size of the data is really small, so the resources which we use to generate thread can make big effect. The higher parallelism we generated, the more resources we need to use. Hence, the resources affect the latency performance so that it cause higher latency when we increase the parallelism.

2. Question: Is there an overhead when using RocksDB as the state backend as compared to using in-memory state? What could be the reason (if yes/no)?

I think it often provides a overhead as compared to using in-memory state. The data we collected in the experiment proves my guess. When we run same task with same parallelism, the latency when we use RocksDB is often higher than we using in-memory state.
The reason is simple. When you choose RocksDB as your state backend, your state lives as a serialized byte-string in either the off-heap memory or the local disk. RocksDB is a Key-Value store, every time you register a keyed state, the key-value pairs are stored as serialized bytes within RocksDB. This means that data has to be deserialized or serialized with every read or write operation, which can compromise performance when compared to in-memory state backends in Flink.