关系数据库作为生产环境中的OLTP，承载着重要的使命，说它是整个业务系统的最重要的部分都不为过。AWS RDS MySQL作为当前很受欢迎并且使用量很大的托管服务，在一定程度下降低了运维和DBA的负担。但是我们仍然需要面对线上出现的各种复杂的RDS MySQL性能问题，由于MySQL的性能问题是一个全方位立体的问题，上到整个业务系统的架构，读写节点的拓扑，节流策略的设置，业务侧分库分表，下到表的关系建立，表结构设计，SQL语句优化，MySQL服务器的参数调优，实时的监控MySQL的指标并反馈等等。这篇文章根据自己遇到的各种客户问题从SQL latency的角度切入来谈一下latency问题的定位，RDS MySQL侧的监控以及一些优化的方法。

我们只有确定了问题很可能出在哪里，然后采取对应的措施才可能解决。否则一顿RDS实例升级操作可能根本没有解决问题还白花钱。但是一般线上分析可能对业务有影响，而且可能捕捉不到问题现场，所以多采用事后分析；而出现RDS MySQL性能恶化的情况下需要一种应急措施来维持线上业务，那么建议在业务侧那边采取限流策略或者对RDS MySQL来降级(比如从读写变为只读)，极端情况下，业务侧那边要采用熔断机制暂时不把新的请求发给RDS MySQL，不要让一个已经很糟糕的server变得更糟糕。升级RDS实例做为可选的最后一步应急措施！

接下来我们按照先分析问题，然后解决问题的顺序来讲，最后还给出一些如何监控SQL层面以及收集信息的建议。

1. 分析问题：
   1. 从业务侧看到SQL比较慢，latency不外乎由如下几部分组成（通过枚举请求处理的各个路径即可获知，全链路监控的重要性就体现在这里了）：

业务侧收发请求带入的latency；

业务侧的runtime环境带入的latency（比如业务侧跑在EC2虚机上，那么该虚机所在的物理主机当前的load可能引起的抖动会对latency有贡献），

业务侧与RDS MySQL实例之间的网络带入的latency；

RDS MySQL server本身，它所在的物理主机的load，以及RDS MySQL的拓扑结构带入的latency；

RDS MySQL用到的EBS卷带入的IO latency。

* 1. Latency有两种：

一种是CPU消耗性的即所谓的CPU bound（比如没有使用index的SQL会进行全表扫描一般会有比较高的CPU使用率，尤其是数据绝大部分都在innodb的buffer pool的话）；

一种是非CPU消耗性的，主要有IO消耗性的所谓的IO bound包括disk IO与network IO（比如有频繁的写操作而且需要尽快落盘）和对共享资源的争用导致的延迟（比如数据库级别的锁和OS级别的锁，并发进入innodb的线程数超过设置的阈值，这些都可以让当前线程挂起并放弃CPU）。

可以查看RDS MySQL CPU使用率以及可用物理内存的监控图，如果CPU使用率比较高，打开RDS enhanced monitoring（因为这个要收费，所以只是在出现问题的时候在enable。另外，它只是收集系统的一些统计信息，所以它对mysqld进程的性能影响几乎很小可以忽略），把能选上的选项都勾选，目的是查看CPU高是不是真的是mysqld进程占用了绝大部分。

根据出问题的时间段，查看该时间段的RDS的disk write/read latency平均值和最大值监控图，以及这个时间段的read/write IOPS监控图。如果该时间段RDS的disk average latency比平时高很多，而IOPS也很大接近了该EBS卷的IOPS capacity，可以考虑在线提升EBS的IOPS capacity。

* 1. 如果排除了其他的原因比如业务侧，网络和EBS IO的问题，那么问题就narrow down到MySQL层面了。

如果是mysqld占用的CPU高，常见的有如下几种情况：

SQL没有使用索引导致的全表扫描；

大表的排序操作或者group by操作；

多个表的Join操作；

复杂的嵌套子查询操作；

* + 1. 对于已经执行完的事务，可以通过查看slow query日志来分析：

慢查询日志的分析有很多开源工具比如pt-query-digest（percona有很多工具很好用比如pt-mysql-summary），MySQL自带的mysqldumpslow也可以分析。

通过慢查询分析工具找到慢查询的SQL以后可以通过explain或explain extended来看该SQL的执行计划来分析一下该SQL的索引使用情况，进而可以推断。

举几个简单例子来看执行计划：

explain select \* from test;

+----+-------------+-------+------+---------------+------+---------+------+------+-------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+------+-------+

| 1 | SIMPLE | test | ALL | NULL | NULL | NULL | NULL | 10000 | NULL |

+----+-------------+-------+------+---------------+------+---------+------+------+-------+

上面输出表示这个SQL要执行全表扫描（type为ALL），扫描的行数大概是10000（rows为10000），显然这个SQL执行的慢。

explain select t1 from test\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: test

type: index

possible\_keys: NULL

key: idx\_t1

key\_len: 8

ref: NULL

rows: 10000

Extra: Using index

上面的输出中尽管使用了索引（Extra字段显示Using index），但是注意这里的type是index表示的是基于索引的全扫描方式返回数据。同样从扫描的行数rows仍然为10000看到该SQL依然执行很慢。

explain select \* from test where id =1 ;

+----+-------------+------------+-------+---------------+---------+---------+-------+------+-------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+------------+-------+---------------+---------+---------+-------+------+-------+

| 1 | SIMPLE | test | const | PRIMARY | PRIMARY | 8 | const | 1 | NULL |

+----+-------------+------------+-------+---------------+---------+---------+-------+------+-------+

上面的输出中的type为const，表示最多只有一行匹配，用于主键或者唯一索引比较时。同样从扫描的行数rows也能看出只扫描了一行。

分析慢查询SQL的时候，通过explain来查看执行计划是非常重要的一环，而且不同版本的MySQL对同一个SQL的执行计划可能不同，因此一定要查看执行计划。explain的输出中每个字段的含义具体请参考MySQL的官网：<https://dev.mysql.com/doc/refman/5.7/en/explain-output.html>

* + 1. 还可以把这个找到的SQL利用MySQL的profiling工具来分析一下：

Set profiling =1；（Profiling is enabled per session）

执行这些SQL；

Show profiles；-----------显示有哪些query id；

Show profile for query xx；-----显示某个query的具体执行细节

举个例子，我们要查看query id为2的SQL的执行细节如下:

show profile for query 2;

+----------------------+----------+

| Status | Duration |

+----------------------+----------+

| starting | 0.000147 |

| checking permissions | 0.000023 |

| Opening tables | 0.000047 |

| init | 0.000081 |

| System lock | 0.000031 |

| optimizing | 0.000034 |

| statistics | 0.001650 |

| preparing | 0.000046 |

| executing | 0.000018 |

| Sending data | 2.460588 |

| end | 0.000041 |

| query end | 0.000019 |

| closing tables | 0.000022 |

| freeing items | 0.000055 |

| cleaning up | 0.000085 |

+----------------------+----------+

很明显上面的输出中“Sending data”这个状态占了2秒多，这个状态表示mysql 线程对select语句执行read，process rows和send data to client整个过程。一般这个过程会涉及到大量的硬盘读（如果数据不再memory的话）。

在MySQL 5.7版本以后，profiling会逐渐废弃，推荐使用performance schema。

关于profiling参考官网：<https://dev.mysql.com/doc/refman/5.7/en/show-profile.html>；关于thread status参考官网：<https://dev.mysql.com/doc/refman/5.7/en/general-thread-states.html>

* + 1. 还可以通过Performance\_schema系统库中的统计信息来查看：

Performance\_schema是默认关闭的并且是静态参数，可以考虑在自定义参数组中把它enable并重启。由于Performance\_schema中收集信息的table在开启后会对MySQL的性能有一些影响，所以一般平时不要enable那些table，在需要troubleshooting的时候在enable具体的table，然后收集一小段时间后，确认收集到需要的信息后在把这些table disable。

举个例子来看如何查看SQL语句的细节：

* + - 1. 首先enable events\_statements\_\*和events\_stages\_\*这些consumers：

UPDATE performance\_schema.setup\_consumers

SET ENABLED = 'YES'

WHERE NAME LIKE '%events\_statements\_%';

UPDATE performance\_schema.setup\_consumers

SET ENABLED = 'YES'

WHERE NAME LIKE '%events\_stages\_%';

* + - 1. 然后执行一个SQL查询：

SELECT \* FROM employees.employees WHERE emp\_no = 10001;

+--------+------------+------------+-----------+--------+------------+

| emp\_no | birth\_date | first\_name | last\_name | gender | hire\_date |

+--------+------------+------------+-----------+--------+------------+

| 10001 | 1953-09-02 | Georgi | Facello | M | 1986-06-26 |

+--------+------------+------------+-----------+--------+------------+

* + - 1. 接着通过查询events\_statements\_history\_long 表来找到上面那个SQL查询的EVENT\_ID（这里通过表的emp\_no字段的值10001来找EVENT\_ID）。

SELECT EVENT\_ID, TRUNCATE(TIMER\_WAIT/1000000000000,6) as Duration, SQL\_TEXT

FROM performance\_schema.events\_statements\_history\_long WHERE SQL\_TEXT like '%10001%';

+----------+----------+--------------------------------------------------------+

| event\_id | duration | sql\_text |

+----------+----------+--------------------------------------------------------+

| 31 | 0.028310 | SELECT \* FROM employees.employees WHERE emp\_no = 10001 |

+----------+----------+--------------------------------------------------------+

* + - 1. 最后通过EVENT\_ID从events\_stages\_history\_long中查看对应SQL对应的各个子步骤的执行时长。

SELECT event\_name AS Stage, TRUNCATE(TIMER\_WAIT/1000000000000,6) AS Duration

FROM performance\_schema.events\_stages\_history\_long WHERE NESTING\_EVENT\_ID=31;

+--------------------------------+----------+

| Stage | Duration |

+--------------------------------+----------+

| stage/sql/starting | 0.000080 |

| stage/sql/checking permissions | 0.000005 |

| stage/sql/Opening tables | 0.027759 |

| stage/sql/init | 0.000052 |

| stage/sql/System lock | 0.000009 |

| stage/sql/optimizing | 0.000006 |

| stage/sql/statistics | 0.000082 |

| stage/sql/preparing | 0.000008 |

| stage/sql/executing | 0.000000 |

| stage/sql/Sending data | 0.000017 |

| stage/sql/end | 0.000001 |

| stage/sql/query end | 0.000004 |

| stage/sql/closing tables | 0.000006 |

| stage/sql/freeing items | 0.000272 |

| stage/sql/cleaning up | 0.000001 |

+--------------------------------+----------+

具体可以参考官网：<https://dev.mysql.com/doc/refman/5.7/en/performance-schema-query-profiling.html>，<https://dev.mysql.com/doc/refman/5.7/en/performance-schema-examples.html>

慢查询日志分析和performance schema同样也适合分析CPU不高的场景。

* + 1. 对于正在执行中的事务，可以通过查看show full processlist看具体的线程当前正在做什么：

比如对于查询时间长、运行状态（State 列）是"Sending data"，"Copying to tmp table"、"Copying to tmp table on disk"、"Sorting result"、"Using filesort" 等都是可能有性能问题的查询（SQL）。

另外，需要利用show global status来计算QPS，在 QPS 高导致 CPU 使用率高的场景中，查询执行时间通常比较短，show full processlist可能不容易捕捉到当前执行的查询。

1. 解决问题：

通过上面的思路和分析很可能就找到了Root cause，那么接下来就是去fix或者optimization的时候了。下面的前两条主要是根据分析的结果来解决当前SQL慢的方法，剩下的是那几条则是更多的从日常优化的角度。

* 1. 如果是SQL语句本身导致的，那么可以考虑从SQL层面，索引层面，表结构层面甚至是表的关系建立方面来着手。
     1. 不同的关系数据库对同一个复杂的SQL的执行计划可能都不同，所以可能在Oracle上执行很快的SQL语句直接放在MySQL上执行就会很慢，这个时候需要对该SQL语句进行变形甚至拆分多几个SQL语句来迎合MySQL。
     2. 如果是因为执行过程没有使用索引导致，那么考虑建立合适的索引来加速。
     3. 如果是因为表太大，可以考虑进行分区或者分表的方式。
     4. 如果是因为太多表join比如10个表join操作导致的，那么需要考虑是否应该把这样的join语句放在MySQL上跑。一般这种情况下，可以考虑把对应的数据抽取到一个OLAP上去做分析。
  2. 如果是因为高QPS或者高并发导致的，那么可以考虑从数据库可调参数，读写拓扑，负载均衡，业务侧限流，cache前置，分库分表，分布式sharding，甚至业务侧划分为多个独立的集群以及建模范式来着手。
     1. 你的业务场景是否用NoSQL来建模，NoSQL是去关系的，天然的并发比关系数据库好。
     2. 随着你的业务的增长，可能QPS会更高，那么就要考虑是否应该从业务侧分库分表或者部署分布式sharding集群，或者直接从业务侧划分多个独立的集群来拆分QPS。
     3. 单个关系数据库实例本身能承载的并发都是有限的，由于具体实现的复杂性以及并发模型BIO/NIO的选择，共享数据的争用从而不可能很大。

这里做个广告，AWS RDS Aurora for MySQL同等配置下要比RDS MySQL的性能好，区别于MySQL的BIO模型，Aurora采用基于epoll的NIO的基于事件驱动的React方式来提升并发性能。Aurora的优点远不止这些，有兴趣可以参考AWS Aurora相关博客和文档。

这个时候，考虑是读多还是写多，读多的话是否可以容忍弱一致性，考虑建立更多的一级或者多级read replica来着手（只所以会用到级联复制，是因为同一个master的slave越多，那么master上的多个事务提交线程（需要binlog文件写锁）与多个slave对应的master上的多个dump thread（需要binlog文件读锁）对binlog文件锁的争用就越剧烈，从而影响master和slave的性能。这个也是不建议主库有很多slave的原因。），同时对多个read replica前置对MySQL协议感知的中间件比如Mycat或者对集群拓扑感知的client library来做负载均衡读请求；写多的话，可能需要分区，分库分表，sharding甚至划分独立集群。

* + 1. 前置的cache比如redis是关系数据库的最佳搭档，尽量要部署，让更多的弱一致性的读请求不要到达关系数据库。
    2. 业务侧的限流机制非常重要，可以缓解数据库的负载，另外，客户端库的指数退让重试机制非常适合和有状态的服务比如关系数据库来交互的。目的就是让数据库不要进一步恶化直到崩溃。
    3. RDS MySQL有很多参数可以用来调整：

比如一般建议把MySQL的query cache禁用（现在一般生产环境不怎么使用query cache，query cache本身全局锁容易成为瓶颈，每次DML commit还要干掉对应的query cache中的旧的内容，还有如果query cache的hit不高更要命）。

如果可以容忍很少的数据丢失和/或者主从数据不一致，可以不用把sync\_binlog和innodb\_flush\_log\_at\_trx\_commit设置为1即每次事务commit都需要落盘。

比如设置合理的wait\_timeout不让淘气的客户端不干事还占用资源。

MySQL很多参数之间有着错综复杂的关系，在确认你对这些关系很清楚前，不要轻易修改默认的参数。

在这个场景下，升级RDS实例的机型可能对QPS和高并发有帮助。

* 1. 利用SHOW ENGINE INNODB STATUS查看innodb的一些metric：

Semphore section有metric “Spin rounds per wait”，这个metric如果比较大，表示当前很多线程需要等待一个OS级别的锁，在这样的情况下最好降低innodb\_thread\_concurrency（这个参数默认是0，建议可以修改为128试试）。

* 1. Information\_schema.INNODB\_TRX是当前运行的所有事务的信息：

SELECT trx.trx\_id, trx.trx\_started, trx.trx\_mysql\_thread\_id FROM INFORMATION\_SCHEMA.INNODB\_TRX trx WHERE trx.trx\_started < CURRENT\_TIMESTAMP - INTERVAL 1 SECOND\G

上面是查看在1秒前开始运行还没有结束的事务。如果发现某个事务很久还没有结束并且不是期望的行为，那么可以考虑kill掉该事务。

应该及时提交长时间的事务，如果一直有事务没有提交可以考虑把这个事务给kill了。

* 1. 对于table经常修改（增删改）的情况，最好定期比如1周用Analyze table 命令来更新表的统计信息（统计信息对执行计划评估有很大影响，执行计划不好就会导致SQL性能下降）。
  2. 硬盘碎片空间可以被MySQL重复利用，但会引发查询性能不可控地下降。

对于MySQL的硬盘空间的回收只能对独立的表空间有用（每个表的数据保存为一个文件，因此就可以通过拷贝数据删掉旧的文件来把硬盘空间归还给OS），共享的表空间是所有的表共同保存在几个文件中，没有办法进行类似的操作。

对于基于innodb存储引擎的MySQL数据库，要回收磁盘空间，推荐使用重置表存储引擎的方法：alter table table\_name engine=innodb（重置期间所有写会话会被阻塞，读会话不受影响），重置消耗时间与已删除数据无关，仅与剩下数据呈线性相关。

Optimize table对于innodb引擎来说，也会被mapping到alter table来处理（创建一个新表，拷贝数据，原子rename新表为旧表表名）

这些命令对在线业务有一些影响，最好在业务低峰做。

* 1. 设置autocommit=1，尤其应该检查一些GUI客户端（比如通过慢查询日志或者general日志）例如MySQL Workbench和Navicat，它们极有可能在连接MySQL时采用autocommit=0的模式，应该调整过来，否则就可能会有事务忘记提交。

1. 如何更好的从SQL层面监控和收集有用的信息，给事后分析准备充足的“食材”：

MySQL的性能问题很复杂，不是一句话就能解决的，需要收集到足够的信息来分析。出现SQL慢的时候，需要收集如下的信息：

* 1. 慢查询日志中出现的SQL；

慢查询日志相对general日志来说（general日志没有什么特殊目的尽量不要打开，对性能还是有一些影响的），对MySQL的影响比较小，在出问题的时候可以开启（这个时候开启只能记录新的慢SQL）。如果想一直记录慢查询的SQL也可以考虑一直开启，这样也方便做事后分析。

注意：有时候慢查询日志里面没有记录实际执行很慢的SQL，但是通过审计日志或者general日志却能发现这些慢的SQL，是因为这些SQL都没有执行成功（比如客户端在发起的SQL还没有执行完就关闭了TCP连接），所以它是不会记录到慢日志中的，但是都会记录到审计日志或者general日志。

* 1. 可以在业务端的监控系统监控业务端发起的SQL，如果SQL持续比较慢比如连续30秒SQL响应平均超过2s的时候则触发调用一个脚本来收集以下的信息：
     1. 执行一次show global variables；
     2. 每间隔一秒执行一次show full processlist， show global status，SHOW ENGINE INNODB STATUS， 一共执行比如5次。
     3. 执行一次SELECT trx.trx\_id, trx.trx\_started, trx.trx\_mysql\_thread\_id FROM INFORMATION\_SCHEMA.INNODB\_TRX trx WHERE trx.trx\_started < CURRENT\_TIMESTAMP - INTERVAL 1 SECOND\G
     4. 执行一次 SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_LOCKS\G
     5. 执行一次SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_LOCK\_WAITS\G

收集上面的所有的这些的输出，如果可能的话最好在出问题的时候把慢查询日志也收集，收集完毕就可以关闭慢查询日志（慢查询日志记得一定要把output修改为FILE，同时要enable log\_queries\_not\_using\_indexes来记录所有不使用索引的SQL）

* 1. 如果问题持续时间比较长，可以利用performance schema来收集更详细的信息。
  2. 最好在出问题的时候，查看RDS console的如下的监控图：

出问题的时间段的RDS的DISK write/read latency平均值和最大值，这个时间段的read/write IOPS； CPU使用的平均值和最大值以及可用内存； RDS实例的网络network-in和network-out。

把这些信息都收集完毕后，你可以自己来做分析，也可以让AWS来帮你分析。你选用AWS RDS MySQL的一个附加效果就是AWS就是你的DBA。

总的来说，关系数据库RDS MySQL的优化是一个基于技术的艺术，千变万化还可能随时间变化，但是不管如何变化，套路总是不怎么变的（这些套路对原生的MySQL也是适用的），希望这篇文章对你有帮助。