Oceanus 场景约束

五八同城信息技术有限公司

目录

目录 4
1 引言 6
1.1 编写目的 6
1.2 Oceanus 设计之美 6
1.2.1 面向 JDBC 标准 6
1.2.2 无限可能 6
1.2.3 划定适用场景 6
2 聚集函数 7
2.1 函数 7
2.2 分库分表场景的实现思路 7
2.3 案例分析 7
2.3 结论 8
2.4 建议 8
3 多表操作 9
3.1 JOIN 9
3.1.1 查询 9
3.1.2 修改 9
3.2 SQL 嵌套 9
3.2.1 子查询 9
3.2.2 修改 10
3.3 结论 10
3.4 建议 10
4 分页 11
4.1 语法 11
4.2 分库分表场景的实现思路 11
4.3 案例分析 11
4.4 结论 12
4.5 建议 12
4.6 扩展 12
5 事务 14
5.1 CAP 理论 14
5.2 多阶段提交 14
5.2.1 两段式提交 14
5.2.2 三段式提交 14
5.3 产品实现 15
5.4 案例分析 15
5.4.1 案例 15
5.4.2 初步分析 16
5.4.1 业务改造方案 16
5.5 建议 16
6 路由 17
6.1 配置约定 17
6.2 建议 17

7 线程池 18
7.1 线程池用途 18
7.2 配置说明 18
8 埋点 19
8.1 埋点用途 19
8.2 注意事项 19
9 附加计算 20
9.1 Order by 语句 20
9.2 Group by 语句 20
9.3 Having 语句 20
9.4 别名 20
9.5 建议 20

1 引言

1.1 编写目的

数据在单据数据库存储，与分库分表场景的多库存储，在使用方法上有很大的不同。各类数据库中间件产品极力简化使用者的开发成本，希望做到操作单库场景一样操作多库。

Oceanus 通过标准配置，与业务方制定好路由规则(参见《Oceanus 使用文档》)，程序运行时提交简单 sql 即可自动路由到不同库中执行，合并结果后返回上层。

这样的用户视觉看似完美，实际上背后有着很大的隐性开销，本文将围绕着各类需求场景，阐述 Oceanus 的使用约束

1.2 Oceanus 设计之美

1.2.1 面向 JDBC 标准

基于 JDBC 标准完成的产品可以很好的集成其他产品，降低耦合成本，降低开发人员的学习成本，没有个性化 API。

1.2.2 无限可能

系统层次清晰，任何一点都可以切换更优的实现，集成外部系统，将 DB 中间件打造成自己的平台产品

1.2.3 划定适用场景

我们不做大而全的产品，不做完美无缺的方案。在一定范围内把事做好，提供可靠的技术支持。

2 聚集函数

业务中时常会有 count 条数、计算平均值之类的需求，单机场景与分库分表场景在实现上会有不同：

2.1 函数

min、max、avg、sum、count

2.2 分库分表场景的实现思路

- 步骤 1：

  拆解成另一种sql

- 步骤 2：

  路由到指定节点(全部节点)执行

- 步骤 3：

  按照原先语义进行结果集合并，封装成一个ResultSet返回上层

2.3 案例分析

例：有10个班，学生信息表(user)，存储学生信息，要求计算平均年龄

单库时可以执行 SQL：select avg(age) from user

分N个库时在所有库上执行上述 SQL 进行汇总，然后相加再除N

显然这是不对的，实际结果与正确答案不符

正确的应该拆解成：select sum(age), count(age) from user，再计算总和，除以学生人数。

2.3 结论

分库分表后，数据已经变得分散，单库时的语义已经不能得出准确结果，满足需求时所增加的开销远不止是N倍，甚至是超乎想象的。

2.4 建议

根据业务需求，借助其他途径解决。

如：建立异构表，用于存储年龄和人数

3 多表操作

当业务中一个实体操作对应多个表存储时，会经常写多表操作语句，这时有很多需要注意的点：

3.1 JOIN

3.1.1 查询

join 实现是 Nested Loop Join 算法，以小结果集驱动大结果集循环匹配，并且被 join 的字段如果没有索引的话，会进行全表检索。

单数据库场景中表结构的设计对执行效率影响非常大，分库分库场景需要把数据远程拉到本地进行关联，成本开销更是大得离谱。

3.1.2 修改

任何修改都会有锁在其中，当使用 join 条件作为连接条件进行修改时，如果编写不当，容易形成死锁。

3.2 SQL 嵌套

3.2.1 子查询

子查询是通过临时表的方式实现，能够避免事务和表锁死，但是效率也同样很低。

分库分库场景同样需要做数据迁移到本地进行关联，成本开销也是非常大

3.2.2 修改

子查询会锁住语句中的子表。默认事务隔离级别：REPEATABLE-READ，是不能保证这种条件下的原子表操作，会形成表的死锁。改为 REPEATABLE-READ 虽然可以解决问题，但是效率下降了很多，得不偿失。

3.3 结论

多表操作的实现在单机场景中的实现方案，不合适在分库分表的场景中使用，受网络因素影响，原本的问题甚至更加放大。

3.4 建议

根据业务需要，对多表操作的逻辑进行拆分，进行多次操作，简化单次的数据库执行复杂度。

4 分页

绝大多数系统中都会有分页查询的需求，使得这个问题足可以做成一个独立的重点，不同数据库对分页条件的支持也不同，Oceanus 支持 limit offset 语句，功能是满足的，但是也有其支持的条件范围

4.1 语法

limit N、 limit M,N、 limit N offset M

4.2 分库分表场景的实现思路

单库路由时，语句不变。重点在于多库(多表)路由时的实现方式。

- 步骤 1：

  分析 where 条件和 order by 条件，得到路由的指定分片

- 步骤 2：

  修改 limit 范围，在不同分片上分别执行

- 步骤 3：

  按照原先语义进行结果集合并，排序，截取原语句中指定的条目，封装成一个ResultSet返回上层

4.3 案例分析

例：有10个班，学生成绩表(score)，存储学生考试成绩，要求输出全部学生中，分数第11名到第20名

单库时可以执行 SQL：select score from user order by score desc limit 10, 10

分N个库时在所有库上执行上述 SQL 进行汇总，然后根据结果集排序得

出10条。

显然这又是不对的，实际结果与正确答案不符

正确做法：拆解SQL在N个库中执行：select score from user order by score desc limit 20，每个表取出20条，然后进行合并、排序，截取第11到20条，封装成ResultSet返回上层。

4.4 结论

分库分表场景中，数据分布策略是关键，如果能有绝对平均的分布，对某些条件查询会起到决定性作用，通用场景下的分页，由于数据分布不均，必须查所有的库(表)，并且尽可能多取，将数据拉到本地进行合并、排序，才能得到期望的正确结果。

4.5 建议

根据业务需要，建立索引系统，对要查询、排序的字段做外置索引，每次查询时首先经过索引，得到 shading 字段(例如主键 id)，在根据 shading 条件查询指定的分片。

4.6 扩展

Mysql 的 limit N 没有什么好说的，而 limit N offset M 实现就比较挫了，根据条件语句的不同，扫描行数也会有差别。详见 EXPLAIN 的结果。

下面是一个例子，表结构：

```
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | bigint(20)  | NO   | PRI | NULL    |       |
| uname | varchar(50) | YES  |     | NULL    |       |
| age   | int(11)     | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

语句 1：EXPLAIN SELECT * FROM t_user LIMIT 100, 10;
结果 1：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t_user	ALL	NULL	NULL	NULL	NULL	982	

结论 1：自然排序时进行全表扫描

语句 2：EXPLAIN SELECT * FROM t_user ORDER BY id LIMIT 100, 10;
结果 2：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t_user	index	NULL	PRIMARY	8	NULL	110	

结论 2：使用主键排序后，扫描 110 行，返回 10 行

语句 3：EXPLAIN SELECT * FROM t_user ORDER BY uname LIMIT 100, 10;
结果 3：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t_user	ALL	NULL	NULL	NULL	NULL	982	Using filesort

结论 3：使用无索引字段排序时，是借助外部排序机制进行全表扫描

由此可见，使用 limit 语句时为了避免全表扫描，还必须要带有索引的排序条件。即时这样，limit N offset M 时，还是会扫描 N+M 条记录，取最后的 N 条记录返回，翻页越大，效率越低。

5 事务

说到数据库都马上会想到 ACID，事务隔离级别…。

单库场景下的使用都已经很熟练了，数据库对分布式场景中的支持有不同的实现，如 mysql cluster。各类中间件也想做到 DB 操作的透明化。

理想归理想，现实环境中由于网络因素，总会产生这样那样的问题。例如网络延迟、闪断。所以分布式事务不仅是想象中的那么可控。

5.1 CAP 理论

著名的CAP理论不再赘述，一致性和可用性必须要有合理的权衡，做出合理的取舍。

5.2 多阶段提交

5.2.1 两段式提交

通过协调者调度事务流程，当参与者表示可以参与事务后，协调者进行加锁操作，成功加锁后再提交。

这是典型的悲观实现，其中任一节点 crash 都会导致事务失败，锁的范围极大，吞吐量无法提高。并且参与者如果在协调者提交之后突然 crash，还是会造成数据不一致的情况。

5.2.2 三段式提交

作为两段式提交的改良版，通过三步确认的方式执行一个分布式事务，只要前面两步确认成功了，即使最终提交时出现协调者 crash，还是会自动提交，弥补了两段式提交的缺陷。

这也是一个悲观实现，并没有缩小锁的范围，无法提高吞吐量。

5.3 产品实现

以 mysql cluster 为例，DB不要求使用者从代码上做过多调整，实现过程是很透明的。

很多开发者认为，产品提供出来的功能就应该是没问题的，是0开销的。这种想法本身就是一种错误的逻辑，做任何事都要有资源消耗，越是透明的东西就越不可控。

即时是Mysql官方也不推荐使用他的分布式事务，官方说明：

http://www.percona.com/live/mysql-conference-2013/sites/default/files/slides/XA_final.pdf

5.4 案例分析

有的需求并不是看上去那么需要强一致性，通过业务改造可以避免使用分布式事务。

5.4.1 案例

从 A 账户向 B 账户转账 100 元，新增转账记录

5.4.2 初步分析

AB 两个账户、转账记录，分别位于三个数据库中。分布式事务解决方案，给三个表加锁，A 账户减 100 元、B 账户加 100 元、写入 A 向 B 转账记录。当有转账记录可查询的时候，视为转账成功。

5.4.1 业务改造方案

业务需求中隐含的是处理逻辑要求强一致性，如果简单拆解成单机事务的话，执行到 B 账户增加 100 元时失败，A 账户已经减了 100 元，还要再做增加 100 元的补偿，而这种补偿又容易产生其他问题。
换一种思路，把账户余额增加一种冻结状态的话，可以进行逻辑扣款，当 B 成功增加 100 元，即可视为转账成功了

执行过程如下：
单机事务 1：A 账户冻结 100 元
单机事务 2：B 账户增加 100 元
异步操作：A 账户减 100 元
异步操作：增加转账记录

不难看出这样的处理方式中，流程进行到事务 2 时就可响应业务。后面的异步操作完全可以由系统保证，只要业务能够接受最终一致即可。当事务 1 成功而事务 2 失败时，进行账户 A 资金的解冻，即可达到业务上的回滚。

（PS:流程控制和系统间解耦，可通过可靠消息传输服务实现）

5.5 建议

大多数业务的分布式事务需求都可以改造成单机事务执行，制定方案之前多想想怎么规避，而不是把麻烦留给后人。
生产环境不要踩大坑：我不入地狱，谁爱去谁去

6 路由

6.1 配置约定

Oceanus 通过对 table 配置约定用于分片的字段。在 sql 解析后，如果命中配置的字段，会调用对应的路由函数，得到期望的唯一 namenode。否则会在所有 namenode 上都执行一遍，详见《Oceanus 使用文档》

6.2 建议

所有的操作尽量携带 sharding 字段，减少不必要的全表路由

7 线程池

7.1 线程池用途

Oceanus 在异步执行 sql 时会使用线程池来并发执行 N 个节点，线程池作为执行容器而存在。

7.2 配置说明

系统初始化时，根据配置实例化多个线程池实例，table 节点中指定由哪个线程池作为自己的执行容器。

未配置线程池的 table，会使用内部默认线程池。

关于线程池的配置项：线程个数、任务队列大小、超时时间，详见《Oceanus 使用文档》

8 埋点

8.1 埋点用途

为了给使用者更好的监控体验，Oceanus 定义了几个类型的埋点接口，当对应的位置执行时间超出配置预期，就会回调对应的埋点实现类。详见《Oceanus 使用文档》

8.2 注意事项

埋点回调虽然是异步执行，但是也要注意时间阈值的设置，如果设置不当，加上回调逻辑过于复杂，也可能造成堆积。尽量简单处理，比如输出一行日志。

9 附加计算

不仅是聚集函数，在单库场景中的一些附加运算语句，放在分库分表场景中，也会增加使用成本。

9.1 Order by 语句

排序需求是很常见的，分库分表中的排序会把各个库(表)的结果集放在一起，进行合并排序，有内存开销和 CPU 开销

9.2 Group by 语句

分组经常伴随着一些聚集函数，比如按照商品类型分组取平均价格。这使得在分库分表场景下，进行结果集合并的计算复杂度更大

9.3 Having 语句

当聚合函数作为筛选条件时，having 起到了过滤作用。分库分表场景中，同样还是结果集合并时，又进行一次 having 条件的遍历，增加了计算复杂度

9.4 别名

编写复杂 sql 时经常会用到别名， 执行过程中如果有多表带别名的复杂操作、又有聚合函数，就无法 100% 满足结果集字段的兼容性

9.5 建议

通过业务改造和功能拆分，借助其他系统(索引系统，或者异构数据表)，只运行简单 sql，避免不必要的开销，提高执行效率。