本文介绍了在 CockroachDB中优化SQL性能的最佳实践。

相关技术演示，请查看[Performance Tuning](http://doc.cockroachchina.baidu.com/#deploy/performance-tuning/)

多行DML最佳实践

使用多行DML而不是多个单行DML

对于INSERT，UPSERT和DELETE语句，一个多行DML比多个单行DML执行更快。 如果有可能，请使用多行DML而不是多个单行DML。

更多信息查看：

* [Insert Multiple Rows](http://doc.cockroachchina.baidu.com/#develop/sql-statements/INSERT/#insert-multiple-rows-into-an-existing-table)
* [Upsert Multiple Rows](http://doc.cockroachchina.baidu.com/#develop/sql-statements/UPSERT/#upsert-multiple-rows)
* [Delete Multiple Rows](http://doc.cockroachchina.baidu.com/#develop/sql-statements/DELETE/#delete-specific-rows)
* [How to improve IoT application performance with multi-row DML](https://www.cockroachlabs.com/blog/multi-row-dml/)

使用TRUNCATE代替DELETE来删除表中的所有行

TRUNCATE通过drop掉表，然后重建的方式来删除所有行；它的性能比使用DELETE好，DELETE是通过执行多个事务来删除所有行。

批量插入最佳实践

使用多行INSERT语句批量插入现有表

要将数据批量插入到现有表中，请在一个多行INSERT语句中批处理多个行，并且不要在事务中包含INSERT语句。通过监控不同批量大小（10行，100行，1000行）的性能，确定应用程序的最佳批量大小。更多信息查阅 [Insert Multiple Rows](http://doc.cockroachchina.baidu.com/#develop/sql-statements/INSERT/#insert-multiple-rows-into-an-existing-table).

使用IMPORT而不是INSERT批量插入到新表中

要将数据批量插入到全新表中，IMPORT语句的性能优于INSERT。

并行执行语句

CockroachDB支持在单个事务中并行执行独立的INSERT，UPDATE，UPSERT和DELETE语句。 并行执行语句有助于减少聚合延迟并提高性能。 要并行执行语句，请将RETURNING NOTHING子句附加到事务对应语句中。更多请查看：[Parallel Statement Execution](https://www.cockroachlabs.com/docs/stable/parallel-statement-execution.html)

分配列族

列族是表中的一组列，它们作为单个kv对存储在基础kv存储中。

创建表时，所有列都存储为单个列族。

大多数情况，默认情况可以保证有效的kv存储和性能。然而，当频繁更新列和低频更新列组合在一起时，每次更新重写时都会更新到低频更新的列，特别是低频更新的列很大时，把它们分为不同的族是更高效的。更多信息查看[列族](https://www.cockroachlabs.com/docs/stable/column-families.html)。

交错表

交错表通过优化关联密切的表的kv结构来提高查询性能，如果可能一起读取和写入数据，则尝试将数据保持在相同的键值范围内。 如果表经常连接在由交错关系组成的列上将特别有用。

唯一ID最佳实践

以下是生成唯一ID的常用方法：

* 通过使用往返（roundtrip）SELECTs的事务单调增加INT ID
* 使用SERIAL变量生成随机唯一ID

第一种方法没有利用像CockroachDB这样的分布式数据库中可能的并行化。 第二种方法的瓶颈在于，在时间上彼此靠近的ID具有相似的值，并且在表格中彼此物理上靠近。 这可能会导致表中的读写热点问题。

CockroachDB中的最佳实践是使用UUID类型生成唯一ID，该类型并行生成随机唯一ID，从而提高性能。

使用UUID生成唯一ID

要自动生成唯一行ID，请使用带有gen\_random\_uuid()函数的UUID列作为默认值：

**CREATE** **TABLE** t1 (id **UUID** **PRIMARY** **KEY** **DEFAULT** gen\_random\_uuid(), name **STRING**);

**INSERT** **INTO** t1 (name) **VALUES** ('a'), ('b'), ('c');

**SELECT** \* **FROM** t1;

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

| id | name |

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

| 60853a85-681d-4620-9677-946bbfdc8fbc | c |

| 77c9bc2e-76a5-4ebc-80c3-7ad3159466a1 | b |

| bd3a56e1-c75e-476c-b221-0da9d74d66eb | a |

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

(3 rows)

或者，您可以使用带有uuid\_v4()函数的BYTES列作为默认值：

**CREATE** **TABLE** t2 (id BYTES **PRIMARY** **KEY** **DEFAULT** uuid\_v4(), name **STRING**);

**INSERT** **INTO** t2 (name) **VALUES** ('a'), ('b'), ('c');

**SELECT** \* **FROM** t2;

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

| id | name |

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

| "**\x**9b**\x**10**\xdc\x**11**\x**9a**\x**9cGB**\xbd\x**8d**\t\x**8c**\xf**6@vP" | a |

| "**\xd**9s**\xd**7**\x**13**\n**\_L\***\xb**0**\x**87c**\xb**6d**\xe**1**\xd**8@" | c |

| "**\uac**74**\x**1dd@B**\x**97**\xac\x**04N&**\x**9eBg**\x**86" | b |

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

(3 rows)

在任何一种情况下，生成的ID都是128位，足够大所以几乎没有机会生成重复值。 此外，一旦表增长超出单个键值范围（默认情况下超过64MB），新ID将分散在所有表的范围内，因此可能跨越不同的节点。 这意味着多个节点将共担负载。

如果生成的ID存储在相同的kv范围内很重要，则可以使用SERIAL数据类型，它是INT的别名，unique\_rowid()函数作为默认值：

**CREATE** **TABLE** t3 (id SERIAL **PRIMARY** **KEY**, name **STRING**);

**INSERT** **INTO** t3 (name) **VALUES** ('a'), ('b'), ('c');

**SELECT** \* **FROM** t3;

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

| id | name |

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

| 293807573840855041 | a |

| 293807573840887809 | b |

| 293807573840920577 | c |

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

(3 rows)

在insert上，unique\_rowid()函数从执行插入的节点的时间戳和ID生成默认值。 除非在每个节点每秒生成非常大量的ID（100,000+）的情况下，否则这种时间排序的值可能是全局唯一的。

使用INSERT和RETURNING子句生成唯一ID

如果有什么原因让你无法使用UUID来生成唯一的ID，你可能会使用带有SELECT的INSERT来返回ID。但是，使用[带有RETURNING子句的INSERT语句](http://doc.cockroachchina.baidu.com/#develop/constraints/insert/#insert-and-return-values)可以提高性能。

生成单调增加的唯一ID

假设表schema如下：

> **CREATE** **TABLE** X (

ID1 INT,

ID2 INT,

ID3 INT **DEFAULT** 1,

**PRIMARY** **KEY** (ID1,ID2)

);

常见的方法是使用带有INSERT然后是SELECT的事务：

> **BEGIN**;

> **INSERT** **INTO** X **VALUES** (1,1,1)

**ON** CONFLICT (ID1,ID2)

**DO** **UPDATE** **SET** ID3=X.ID3+1;

> **SELECT** \* **FROM** X **WHERE** ID1=1 **AND** ID2=1;

> **COMMIT**;

但是，最佳性能实践是使用带有INSERT的RETURNING子句而不是事务：

> **INSERT** **INTO** X **VALUES** (1,1,1),(2,2,2),(3,3,3)

**ON** CONFLICT (ID1,ID2)

**DO** **UPDATE** **SET** ID3=X.ID3 + 1

RETURNING ID1,ID2,ID3;

生成随机唯一ID

假设表schema如下：

> **CREATE** **TABLE** X (

ID1 INT,

ID2 INT,

ID3 SERIAL,

**PRIMARY** **KEY** (ID1,ID2)

);

生成随机唯一ID的常用方法是使用SELECT语句的事务：

> **BEGIN**;

> **INSERT** **INTO** X **VALUES** (1,1);

> **SELECT** \* **FROM** X **WHERE** ID1=1 **AND** ID2=1;

> **COMMIT**;

但是，最佳性能实践是使用带有INSERT的RETURNING子句而不是事务：

> **INSERT** **INTO** X **VALUES** (1,1),(2,2),(3,3)

RETURNING ID1,ID2,ID3;

索引最佳实践

使用二级索引

你可以使用二级索引来提高涉及列不在主键的查询的性能，你可以创建它们：

* 在CREATE TABLE建表时的INDEX子句中使用。除了明确定义的索引之外，CockroachDB还会自动为具有Unique约束的列创建二级索引。
* 对于已存在的表，使用CREATE INDEX
* 使用ALTER TABLE添加UNIQUE约束，它会为约束列自动创建索引

如何创建更有用的二级索引，你可以查看[best practices](http://doc.cockroachchina.baidu.com/#develop/performance-optimization/sql-best-practices/#best-practices).

使用索引加快join速度

CockroachDB支持merge joins和hash joins。 CockroachDB尽可能使用merge joins，因为它们在计算和内存方面比hash joins更高效。 但是，仅当正在连接的表在相关列上有索引时，才可能进行merge joins; 如果不满足此条件，CockroachDB将转向较慢的hash joins。

为什么merge joins比hash joins快?

Merge joins 在计算上代价更小，并且不需要额外的内存。 它们在两个表的索引列上执行，如下所示：

* CockroachDB从每个表中取一行并进行比较。
* 如果行相等，则CockroachDB返回行。
* 如果行不相等，CockroachDB会丢弃较低值的行并使用下一行重复该过程，直到处理完所有行。

相反，hash joins在计算上消耗更高并且需要额外的内存。 它们在两个表上执行如下：

* CockroachDB在较小的表上创建内存中的哈希表。
* 然后，CockroachDB使用哈希表并扫描较大的表以查找较小表中的匹配行。

为什么要创建索引来执行merge join？

merge join要求在合并列上对两个表建立索引。 如果不满足此条件，CockroachDB将转向较慢的hash join。 因此，在两个表上使用JOIN时，首先在表上创建索引，然后使用JOIN运算符。

请注意，merge join只能用于 [distributed query processing](https://www.cockroachlabs.com/blog/local-and-distributed-processing-in-cockroachdb/).

删除未使用的索引

虽然索引可以提高读取性能，但每次写入都会产生开销。 在某些情况下，与上面讨论的用例一样，是需要权衡的。 但是，如果索引未使用，则会降低DML操作的速度。 因此，尽可能删除未使用的索引。

Join最佳实践

查看 [Join Performance Best Practices](http://doc.cockroachchina.baidu.com/#develop/sql-syntax/join-expressions/#performance-best-practices).

Subquery最佳实践

查看 [Subquery Performance Best Practices](http://doc.cockroachchina.baidu.com/#develop/sql-syntax/subqueries/#performance-best-practices).

Table Scans 最佳实践

避免对大表是用SELECT \*

对于大型表，尽可能避免表扫描（即读取整个表数据）。 而是在SELECT语句中定义必需的字段。

示例

假设表schema如下：

> **CREATE** **TABLE** accounts (

id INT,

customer **STRING**,

address **STRING**,

balance INT

nominee **STRING**

);

现在，如果我们想要查找所有客户的帐户余额，效率低下的表扫描：

> **SELECT** \* **FROM** ACCOUNTS;

此查询检索表中存储的所有数据。 更有效的查询：

> **SELECT** CUSTOMER, BALANCE **FROM** ACCOUNTS;

此查询返回客户的帐户余额。

避免在大表使用 SELECT DISTINCT

SELECT DISTINCT允许你通过删除重复条目从查询中获取唯一条目。 但是，SELECT DISTINCT在计算上消耗高。可以使用 [SELECT with the WHERE clause](http://doc.cockroachchina.baidu.com/#develop/sql-statements/SELECT/#filter-rows) 代替.

使用“AS OF SYSTEM TIME”减少与长时间运行查询的冲突

如果你有长时间运行的查询（例如执行全表扫描的分析查询），并且可以容忍稍微过时的读取，请考虑使用... AS OF SYSTEM TIME子句。 使用此选项，你的查询将返回过去在不同位置出现的数据，并且不会导致与其他并发事务冲突，从而提高应用程序的性能。

但是，因为AS OF SYSTEM TIME返回历史数据，所以你读取的数据可能是旧的。

理解并避免事务争用

满足以下三个条件时发生事务争用：

* 有多个并发事务或语句（由同时连接到单个CockroachDB集群的多个客户端发送）。
* 它们对相同的数据进行操作，特别是对具有相同索引键值的表行（在主键或二级索引上，或通过交错）或使用较接近的索引kv，因而索引数据存在相同range。
* 存在事务写入或修改数据

所有竞争相同key的一组事务将在性能上受限于单个节点的最大处理速度（有限的水平可伸缩性）。 非竞争事务不会受到这种影响。

有两个级别的争用：

* 在相同range但不同索引kv上操作的事务将受到单个节点（范围租约持有者）的总体硬件容量的限制。
* 对相同索引kv进行操作的事务（具体地说，对给定索引键操作在相同列族上的事务）将更严格地序列化以遵守事务隔离语义。

事务争用还会提高事务重启率，从而正确实现客户端事务重试很关键。

为避免争用，有多种策略：

* 使用具有更随机的值分布的索引键值，以便不同行上的事务更可能在不同的数据range上操作。 有关建议，请参阅行ID的[SQL常见问题解答](https://www.cockroachlabs.com/docs/stable/sql-faqs.html)。
* 使事务更小，以便每个事务的工作量更少。 特别是，避免每个事务在多个客户端 - 服务器交换。 例如，使用通用表达式在单个SQL语句中将多个SELECT和INSERT / UPDATE / DELETE / UPSERT子句组合在一起。
* 替换行中的值时，请使用UPSERT并为插入的行中的所有列指定值。 与SELECT，INSERT和UPDATE的组合相比，这通常在争用下具有最佳性能。
* 提高数据的规范化，以避免由不同表中的不同事务修改的相同记录的部分。 但请注意，这是一把双刃剑，因为非规范化还可以通过在不同range内创建经常引用数据的多个副本来提高性能。
* 如果应用程序严格要求在非常少的不同索引键值上操作，请考虑使用ALTER ... SPLIT AT，以便每个索引键值可以由群集中的单独节点组提供。

最好通过架构和应用程序的设计尽可能避免争用。 但是，有时候争论是不可避免的。 为了在存在争用时最大限度地提高性能，你需要最大化单个range的性能。 为实现这一目标，可以应用多种策略：

* 最小化range副本之间的网络距离，可能使用zone配置和分区。
* 使用最快的存储设备。
* 如果竞争事务在相同range内的不同key上运行，则为每个节点提高CPU功率（更多核心）。 但请注意，如果所有事务都在同一个key上运行，则不太可能继续改进。