

1 引言

1.1 编写目的

一个系统的性能的提高，不单单是试运行或者维护阶段的性能调优的任务，也不单单是开发阶段的事情，而是在整个软件生命周期都需要注意，进行有效工作才能达到的。所以我希望按照软件生命周期的不同阶段来总结数据库性能优化相关的注意事项。

1.2 分析阶段

一般来说，在系统分析阶段往往有太多需要关注的地方，系统各种功能性、可用性、可靠性、安全性需求往往吸引了我们大部分的注意力，但是，我们必须注意，性能是很重要的非功能性需求，必须根据系统的特点确定其实时性需求、响应时间的需求、硬件的配置等。最好能有各种需求的量化的指标。

另一方面，在分析阶段应该根据各种需求区分出系统的类型，大的方面，区分是 OLTP（联机事务处理系统）和 OLAP（联机分析处理系统）。

1.3 设计阶段

设计阶段可以说是以后系统性能的关键阶段，在这个阶段，有一个关系到以后几乎所有性能调优的过程—数据库设计。

在数据库设计完成后，可以进行初步的索引设计，好的索引设计可以指导编码阶段写出高效率的代码，为整个系统的性能打下良好的基础。

以下是性能要求设计阶段需要注意的：

1.3.1 数据库逻辑设计的规范化

数据库逻辑设计的规范化就是我们一般所说的范式，我们可以这样来简单理解范式：

- 第 1 规范：没有重复的组或多值的列，这是数据库设计的最低要求。
- 第 2 规范 每个非关键字段必须依赖于主关键字，不能依赖于一个组合式主关键字的某些组成部分。消除部分依赖，大部分情况下，数据库设计都应该达到第二范式。
- 第 3 规范 一个非关键字段不能依赖于另一个非关键字段。消除传递依赖，达到第三范式应该是系统中大部分表的要求，除非一些特殊作用的表。

更高的范式要求这里就不再作介绍了，个人认为，如果全部达到第二范式，大部分达到第三范式，系统会产生较少的列和较多的表，因而减少了数据冗余，也利于性能的提高。

1.3.2 合理的冗余

- 完全按照规范化设计的系统几乎是不可能的，除非系统特别的小，在规范化设计后，有计划地加入冗余是必要的。
- 冗余可以是冗余数据库、冗余表或者冗余字段，不同粒度的冗余可以起到不同的作用。
- 冗余可以是为了编程方便而增加，也可以是为了性能的提高而增加。从性能角度来说，冗余数据库可以分散数据库压力，冗余表可以分散数据量大的表的并发压力，也可以加快特殊查询的速度，冗余字段可以有效减少数据库表的连接，提高效率。

1.3.3 主键的设计

- 主键是必要的，SQL SERVER 的主键同时是一个唯一索引，而且在实际应用中，我们往往选择最小的键组合作为主键，

所以主键往往适合作为表的聚集索引。聚集索引对查询的影响是比较大的，这个在下面索引的叙述。

- 在有多个键的表，主键的选择也比较重要，一般选择总的长度小的键，小的键的比较速度快，同时小的键可以使主键的 B 树结构的层次更少。
- 主键的选择还要注意组合主键的字段次序，对于组合主键来说，不同的字段次序的主键的性能差别可能会很大，一般应该选择重复率低、单独或者组合查询可能性大的字段放在前面。

1.3.4 外键的设计

- 外键作为数据库对象，很多人认为麻烦而不用，实际上，外键在大部分情况下是很有用的，理由是：
- 外键是最高效的一致性维护方法，数据库的一致性要求，依次可以用外键、CHECK 约束、规则约束、触发器、客户端程序，一般认为，离数据越近的方法效率越高。
- 谨慎使用级联删除和级联更新，级联删除和级联更新作为 SQL SERVER 2000 当年的新功能，在 2005 作了保留，应该有其可用之处。我这里说的谨慎，是因为级联删除和级联更新有些突破了传统的关于外键的定义，功能有点太过强大，使用前必须确定自己已经把握好 其功能范围，否则，级联删除和级联更新可能让你的数据莫名其妙的被修改或者丢失。从性能看级联删除和级联更新是比其他方法更高效的方法。

1.3.5 字段的设计

字段是数据库最基本的单位，其设计对性能的影响是很大的。需要注意如下：

- 数据类型尽量用数字型，数字型的比较比字符型的快很多。
- 数据类型尽量小，这里的尽量小是指在满足可以预见的未来需求的前提下的。
- 尽量不要允许 NULL，除非必要，可以用 NOT NULL+DEFAULT 代替。
- 少用 TEXT 和 IMAGE，二进制字段的读写是比较慢的，而且，读取的方法也不多，大部分情况下最好不用。
- 自增字段要慎用，不利于数据迁移。

1.3.6 数据库物理存储和环境的设计

- 在设计阶段，可以对数据库的物理存储、操作系统环境、网络环境进行必要的设计，使得我们的系统在将来能适应比较多的用户并发和比较大的数据量。
- 这里需要注意文件组的作用，适用文件组可以有效把 IO 操作分散到不同的物理硬盘，提高并发能力。

1.3.7 系统设计

- 整个系统的设计特别是系统结构设计对性能是有很大影响的，对于一般的 OLTP 系统，可以选择 CS 结构、三层的 CS 结构等，不同的系统结构其性能的关键也有所不同。
- 系统设计阶段应该归纳一些业务逻辑放在数据库编程实现，数据库编程包括数据库存储过程、触发器和函数。用数据库编程实现业务逻辑的好处是减少网络流量并可更充分利用数据库的预编译和缓存功能。

1.3.8 索引的设计

在设计阶段，可以根据功能和性能的需求进行初步的索引设计，这里需要根据预计的数据量和查询来设计索引，可能与将来实际使用的时候会有所区别。

关于索引的选择，应改主意：

- 根据数据量决定哪些表需要增加索引，数据量小的可以只有主键。
- 根据使用频率决定哪些字段需要建立索引，选择经常作为连接条件、筛选条件、聚合查询、排序的字段作为索引的候选字段。
- 把经常一起出现的字段组合在一起，组成组合索引，组合索引的字段顺序与主键一样，也需要把最常用的字段放在前面，把重复率低的字段放在前面。
- 一个表不要加太多索引，因为索引影响插入和更新的速度。

1.4 编码阶段

编码阶段是本文的重点，因为在设计确定的情况下，编码的质量几乎决定了整个系统的质量。

编码阶段首先是需要所有程序员有性能意识，也就是在实现功能同时有考虑性能的思想，数据库是能进行集合运算的工具，我们应该尽量的利用这个工具，所谓集合运算实际是批量运算，就是尽量减少在客户端进行大数据量的循环操作，而用 SQL 语句或者存储过程代替。关于思想和意识，很难说得很清楚，需要在编程过程中来体会。

下面罗列一些编程阶段需要注意的事项：

1.4.1 只返回需要的数据

返回数据到客户端至少需要数据库提取数据、网络传输数据、客户端接收数据以及客户端处理数据等环节，如果返回不需要的数据，就会增加服务器、网络 and 客户端的无效劳动，其害处是显而易见的，避免这类事件需要注意：

- 横向来看，不要写 SELECT 的语句，而是选择你需要的字段。
- 纵向来看，合理写 WHERE 子句，不要写没有 WHERE 的 SQL 语句。
- 注意 SELECT INTO 后的 WHERE 子句，因为 SELECT INTO 把数据插入到临时表，这个过程会锁定一些系统表，如果这个 WHERE 子句返回的数据过多或者速度太慢，会造成系统表长期锁定，堵塞其他进程。
- 对于聚合查询，可以用 HAVING 子句进一步限定返回的行。

1.4.2 尽量少做重复的工作

这一点和上一点的目的是一样的，就是尽量减少无效工作，但是这一点的侧重点在客户端程序，需要注意的如下：

- 控制同一语句的多次执行，特别是一些基础数据的多次执行是很多程序员很少注意的。
- 减少多次的数据转换，也许需要数据转换是设计的问题，但是减少次数是程序员可以做到的。
- 杜绝不必要的子查询和连接表，子查询在执行计划一般解释成外连接，多余的连接表带来额外的开销。
- 合并对同一表同一条件的多次 UPDATE，比如
`UPDATE EMPLOYEE SET FNAME='HAIWER' WHERE EMP_ID=' VPA30890F' UPDATE EMPLOYEE SET LNAME='YANG' WHERE EMP_ID=' VPA30890F'` 这两个语句应该合并成以下一个语句 `UPDATE EMPLOYEE SET FNAME='HAIWER',LNAME='YANG' WHERE EMP_ID=' VPA30890F'`
- UPDATE 操作不要拆成 DELETE 操作+INSERT 操作的形式，虽然功能相同，但是性能差别是很大的。
- 不要写一些没有意义的查询，比如 `SELECT FROM EMPLOYEE WHERE 1=2`

1.4.3 注意事务和锁

事务是数据库应用中和重要的工具，它有原子性、一致性、隔离性、持久性这四个属性，很多操作我们都需要利用事务来保证数据的正确性。在使用事务中我们需要做到尽量避免死锁、尽量减少阻塞。具体以下方面需要特别注意：

- 事务操作过程要尽量小，能拆分的事务要拆分开来。
- 事务操作过程不应该有交互，因为交互等待的时候，事务并未结束，可能锁定了很多资源。
- 事务操作过程要按同一顺序访问对象。
- 提高事务中每个语句的效率，利用索引和其他方法提高每个语句的效率可以有效地减少整个事务的执行时间。
- 尽量不要指定锁类型和索引，SQL SERVER 允许我们自己指定语句使用的锁类型和索引，但是一般情况下，SQL SERVER 优化器选择的锁类型和索引是在当前数据量和查询条件下是最优的，我们指定的可能只是在目前情况下更有，但是数据量和数据分布在将来是会变化的。
- 查询时可以用较低的隔离级别，特别是报表查询的时候，可以选择最低的隔离级别（未提交读）。

1.4.4 注意临时表和表变量的用法

在复杂系统中，临时表和表变量很难避免，关于临时表和表变量的用法，需要注意：

- 如果语句很复杂，连接太多，可以考虑用临时表和表变量分步完成。
- 如果需要多次用到一个大表的同一部分数据，考虑用临时表和表变量暂存这部分数据。

- 如果需要综合多个表的数据，形成一个结果，可以考虑用临时表和表变量分步汇总这多个表的数据。
- 其他情况下，应该控制临时表和表变量的使用。
- 关于临时表和表变量的选择，很多说法是表变量在内存，速度快，应该首选表变量，但是在实际使用中发现，这个选择主要考虑需要放在临时表的数据量，在数据量较多的情况下，临时表的速度反而更快。
- 关于临时表产生使用 SELECT INTO 和 CREATE TABLE + INSERT INTO 的选择，我们做过测试，一般情况下，SELECT INTO 会比 CREATE TABLE + INSERT INTO 的方法快很多，但是 SELECT INTO 会锁定 TEMPDB 的系统表 SYSOBJECTS、SYSINDEXES、SYSCOLUMNS，在多用户并发环境下，容易阻塞其他进程，所以我的建议是，在并发系统中，尽量使用 CREATE TABLE + INSERT INTO，而大数据量的单个语句使用中，使用 SELECT INTO。
- 注意排序规则，用 CREATE TABLE 建立的临时表，如果不指定字段的排序规则，会选择 TEMPDB 的默认排序规则，而不是当前数据库的排序规则。如果当前数据库的排序规则和 TEMPDB 的排序规则不同，连接的时候就会出现排序规则的冲突错误。一般可以在 CREATE TABLE 建立临时表时指定字段的排序规则为 DATABASE_DEFAULT 来避免上述问题。

1.4.5 子查询的用法

子查询是一个 SELECT 查询，它嵌套在 SELECT、INSERT、UPDATE、DELETE 语句或其它子查询中。任何允许使用表达式的地方都可以使用子查询。

子查询可以使我们的编程灵活多样，可以用来实现一些特殊的功能。但是在性能上，往往一个不合适的子查询用法会形成一个性能瓶颈。

如果子查询的条件中使用了其外层的表的字段，这种子查询就叫作相关子查询。相关子查询可以用 IN、NOT IN、EXISTS、NOT EXISTS 引入。

关于相关子查询，应该注意：

- NOT IN、NOT EXISTS 的相关子查询可以改用 LEFT JOIN 代替写法。比如：
- 如果保证子查询没有重复，IN、EXISTS 的相关子查询可以用 INNER JOIN 代替。比如：
- IN 的相关子查询用 EXISTS 代替，比如
- 不要用 COUNT() 的子查询判断是否存在记录，最好用 LEFT JOIN 或者 EXISTS，比如有人写这样的语句：

1.4.6 慎用游标

数据库一般的操作是集合操作，也就是对由 WHERE 子句和选择列确定的结果集作集合操作，游标是提供的一个非集合操作的途径。一般情况下，游标实现的功能往往相当于客户端的一个循环实现的功能，所以，大部分情况下，我们把游标功能搬到客户端。

游标是把结果集放在服务器内存，并通过循环一条一条处理记录，对数据库资源（特别是内存和锁资源）的消耗是非

常大的，所以，我们应该只有在没有其他方法的情况下才使用游标。

另外，我们可以用 SQL SERVER 的一些特性来代替游标，达到提高速度的目的。

➤ 字符串连接的例子

这是论坛经常有的例子，就是把一个表符合条件的记录的某个字符串字段连接成一个变量。比如需要把 JOB_ID=10 的 EMPLOYEE 的 FNAME 连接在一起，用逗号连接，可能最容易想到的是用游标：

➤ 用 CASE WHEN 实现转换的例子

很多使用游标的原因是因为有些处理需要根据记录的各种情况需要作不同的处理，实际上这种情况，我们可以用 CASE WHEN 语句进行必要的判断处理，而且 CASE WHEN 是可以嵌套的。

变量参与的 UPDATE 语句的例子

SQL SERVER 的语句比较灵活，变量参与的 UPDATE 语句可以实现一些游标一样的功能，比如：

在 SELECT A,B,C,CAST(NULL AS INT) AS 序号

INTO #T

FROM 表

ORDER BY A ,NEWID()

产生临时表后，已经按照 A 字段排序，但是在 A 相同的情况下是乱序的，这时如果需要更改序号字段为按照 A 字段分组的记录序号，就只有游标和变量参与的 UPDATE 语句可以实现了，这个变量参与的 UPDATE 语句如下：

```
DECLARE @A INT
```

```
DECLARE @序号 INT
```

```
UPDATE #T SET
```

```
@序号=CASE WHEN A=@A THEN @序号+1 ELSE 1 END,
```

```
@A=A,
```

```
序号=@序号
```

➤ 如果必须使用游标，注意选择游标的类型，如果只是循环取数据，那就应该用只进游标（选项 FAST_FORWARD），一般只需要静态游标（选项 STATIC）。

➤ 注意动态游标的不确定性，动态游标查询的记录集数据如果被修改，会自动刷新游标，这样使得动态游标有了不确定性，因为在多用户环境下，如果其他进程或者本身更改了纪录，就可能刷新游标的记录集。

1.4.7 尽量使用索引

建立索引后，并不是每个查询都会使用索引，在使用索引的情况下，索引的使用效率也会有很大的差别。只要我们在查询语句中没有强制指定索引，索引的选择和使用方法是 SQLSERVER 的优化器自动作的选择，而它选择的根据是查询语句的条件以及相关表的统计信息，这就要求我们在写 SQL 语句的时候尽量使得优化器可以使用索引。

为了使得优化器能高效使用索引，写语句的时候应该注意：

➤ 不要对索引字段进行运算，而要想办法做变换，比如

➤ 不要对索引字段进行格式转换

➤ 不要对索引字段使用函数

WHERE LEFT(NAME, 3)='ABC' 或者 WHERE SUBSTRING(NAME, 1, 3)='ABC'

➤ 不要对索引字段进行多字段连接

1.4.8 注意连接条件的写法

多表连接的连接条件对索引的选择有着重要的意义，所以我们在写连接条件的时候需要特别的注意。

➤ 多表连接的时候，连接条件必须写全，宁可重复，不要缺漏。

➤ 连接条件尽量使用聚集索引

➤ 注意 ON 部分条件和 WHERE 部分条件的区别

1.4.9 其他需要注意的地方

经验表明，问题发现的越早解决的成本越低，很多性能问题可以在编码阶段就发现，为了提早发现性能问题，需要注意：

➤ 程序员注意、关心各表的数据量。

➤ 编码过程和单元测试过程尽量用数据量较大的数据库测试，最好能用实际数据测试。

➤ 每个 SQL 语句尽量简单

➤ 不要频繁更新有触发器的表的数据

➤ 注意数据库函数的限制以及其性能

1.4.10 学会分辨 SQL 语句的优劣

自己分辨 SQL 语句的优劣非常重要，只有自己能分辨优劣才能写出高效的语句。

➤ 查看 SQL 语句的执行计划，可以在查询分析器使用 CTRL+L 图形化的显示执行计划，一般应该注意百分比最大的几个图形的属性，把鼠标移动到其上面会显示这个图形的属性，需要注意预计成本的数据，也要注意其标题，一般都是 CLUSTERED INDEX SEEK 、 INDEX SEEK 、 CLUSTERED INDEX SCAN 、 INDEX SCAN 、 TABLE SCAN 等，其中出现 SCAN 说

明语句有优化的余地。也可以用语句 `SET SHOWPLAN_ALL ON` 要执行的语句 `SET SHOWPLAN_ALL OFF` 查看执行计划的文本详细信息。

- 用事件探查器跟踪系统的运行，可疑跟踪到执行的语句，以及所用的时间，CPU 用量以及 IO 数据，从而分析语句的效率。
- 可以用 WINDOWS 的系统性能检测器，关注 CPU、IO 参数

1.5 测试、运行、维护阶段

测试的主要任务是发现并修改系统的问题，其中性能问题也是一个重要的方面。重点应该放在发现有性能问题的地方，并进行必要的优化。主要进行语句优化、索引优化等。

试运行和维护阶段是在实际的环境下运行系统，发现的问题范围更广，可能涉及操作系统、网络以及多用户并发环境出现的问题，其优化也扩展到操作系统、网络以及数据库物理存储的优化。

这个阶段的优化方法在这里不再展开，只说明下索引维护的方法：

- 可以用 `DBCC DBREINDEX` 语句或者 SQL SERVER 维护计划设定定时进行索引重建，索引重建的目的是提高索引的效能。
- 可以用语句 `UPDATE STATISTICS` 或者 SQL SERVER 维护计划设定定时进行索引统计信息的更新，其目的是使得统计信息更能反映实际情况，从而使得优化器选择更合适的索引。
- 可以用 `DBCC CHECKDB` 或者 `DBCC CHECKTABLE` 语句检查数据库表和索引是否有问题，这两个语句也能修复一般的问题。

2 设计技巧

2.1 分类拆分数据量大的表

对于经常使用的表（如某些参数表或代码对照表），由于其使用频率很高，要尽量减少表中的记录数量。

2.2 索引设计

在索引设计中，索引字段应挑选重复值较少的字段；在对建有复合索引的字段进行检索时，应注意按照复合索引字段建立的顺序进行。

2.3 数据操作的优化

2.4 数据库参数的调整

数据库参数的调整是一个经验不断积累的过程，应由有经验的系统管理员完成。

2.5 必要的工具

2.6 避免长事务。

2.7 通俗地理解三个范式

通俗地理解三个范式，对于数据库设计大有好处。在数据库设计中，为了更好地应用三个范式，就必须通俗地理解三个范式（通俗地理解是够用的理解，并不是最科学最准确的理解）：

第一范式：1NF 是对属性的原子性约束，要求属性具有原子性，不可再分解；

第二范式：2NF 是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性；

第三范式：3NF 是对字段冗余性的约束，即任何字段不能由其他字段派生出来，它要求字段没有冗余。

没有冗余的数据库设计可以做到。但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。具体做法是：在概念数据模型设计时遵守第三范式，降低范式标准的工作放到物理数据模型设计时考虑。降低范式就是增加字段，允许冗余。基本表及其字段之间的关系，应尽量满足第三范式。但是，满足第三范式的数据库设计，往往不是最好的设计。为了提高数据库的运行效率，常常需要降低范式标准：适当增加冗余，达到以空间换时间

2.8 提高数据库运行效率的办法

在给定的系统硬件和系统软件条件下，提高数据库系统的运行效率的办法是：

- (1) 在数据库物理设计时，降低范式，增加冗余，少用触发器，多用存储过程。
 - (2) 当计算非常复杂、而且记录条数非常巨大时(例如一千万条)，复杂计算要先在数据库外面。
 - (3) 发现某个表的记录太多，例如超过一千万条，则要对表进行水平分割。水平分割的做法是，以该表主键 PK 的某个值为界线，将该表的记录水平分割为两个表。若发现某个表的字段太多，例如超过八十个，则垂直分割该表，将原来的一个表分解为两个表。
 - (4) 对数据库管理系统 DBMS 进行系统优化，即优化各种系统参数，如缓冲区个数。
 - (5) 在使用面向数据的 SQL 语言进行程序设计时，尽量采取优化算法。
- 总之，要提高数据库的运行效率，必须从数据库系统级优化、数据库设计级优化、程序实现级优化，这三个层次上同时下功夫。

3 大数量性能优化设计

数据库优化包含以下三部分，数据库自身的优化，数据库表优化，程序操作优化

3.1 数据库自身的优化

3.1.1 增加次数据文件，设置文件自动增长（粗略数据分区）

➤ 增加次数据文件

从 SQL SERVER 2005 开始，数据库不默认生成 NDF 数据文件，一般情况下有一个主数据文件（MDF）就够了，但是有些大型的数据库，由于信息很多，而且查询频繁，所以为了提高查询速度，可以把一些表或者一些表中的部分记录分开存储在不同的数据文件里

由于 CPU 和内存的速度远大于硬盘的读写速度，所以可以把不同的数据文件放在不同的物理硬盘里，这样执行查询的时候，就可以让多个硬盘同时进行查询，以充分利用 CPU 和内存的性能，提高查询速度。 在这里详细介绍一下其写入的原理，数据文件（MDF、NDF）和日志文件（LDF）的写入方式是不一样的：

数据文件：SQL Server 按照同一个文件组里面的所有文件现有空闲空间的大小，按这个比例把新的数据分布到所有有空间的数据文件里，如果有三个数据文件 A. MDF, B. NDF, C. NDF, 空闲大小分别为 200mb, 100mb, 和 50mb, 那么写入一个 70mb 的东西，他就会向 ABC 三个文件中一次写入 40、20、10 的数据，如果某个日志文件已满，就不会向其写入

日志文件：日志文件是按照顺序写入的，一个写满，才会写入另外一个

由上可见，如果能增加其数据文件 NDF, 有利于大数据量的查询速度，但是增加日志文件却没什么用处。

➤ 设置文件自动增长（大数据量，小数据量无需设置）

在 SQL Server 2005 中，默认 MDF 文件初始大小为 5MB，自增为 1MB，不限增长，LDF 初始为 1MB，增长为 10%，限制文件增长到一定的数目，一般设计中，使用 SQL 自带的设计即可，但是大型数据库设计中，最好亲自去设计其增长和初始大小，如果初始值太小，那么很快数据库就会写满，如果写满，在进行插入会是什么情况呢？当数据文件写满，进行某些操作时，SQL Server 会让操作等待，直到文件自动增长结束了，原先的那个操作才能继续进行。如果自增长用了很长时间，原先的操作会等不及就超时取消了（一般默认的阈值是 15 秒），不但这个操作会回滚，文件自动增长也会被取消。也就是说，这一

次文件没有得到任何增大，增长的时间根据自动增长的大小确定的，如果太小，可能一次操作需要连续几次增长才能满足，如果太大，就需要等待很长时间，所以设置自动增长要注意以下几点：

- 1) 要设置成按固定大小增长，而不能按比例。这样就能避免一次增长太多或者太少所带来的不必要的麻烦。建议对比较小的数据库，设置一次增长 50 MB 到 100 MB。对大的数据库，设置一次增长 100 MB 到 200 MB。
- 2) 要定期监测各个数据文件的使用情况，尽量保证每个文件剩余的空间一样大，或者是期望的比例。
- 3) 设置文件最大值，以免 SQL Server 文件自增长用尽磁盘空间，影响操作系统。
- 4) 发生自增长后，要及时检查新的数据文件空间分配情况。避免 SQL Server 总是往个别文件写数据。

因此，对于一个比较繁忙的数据库，推荐的设置是开启数据库自动增长选项，以防数据库空间用尽导致应用程序失败，但是要严格避免自动增长的发生。同时，尽量不要使用自动收缩功能。

➤ 数据和日志文件分开存放在不同磁盘上

数据文件和日志文件的操作会产生大量的 I/O。在可能的条件下，日志文件应该存放在一个与数据和索引所在的数据文件不同的硬盘上以分散 I/O，同时还有利于数据库的灾难恢复。

3.1.2 表分区，索引分区（优化①粗略的进行了表分区，优化②为精确数据分区）

➤ 为什么要表分区？

当一个表的数据量太大的时候，我们最想做的一件事是什么？将这个表一分为二或者更多分，但是表还是这个表，只是将其内容存储分开，这样读取就快了 N 倍了

原理：表数据是无法放在文件中的，但是文件组可以放在文件中，表可以放在文件组中，这样就间接实现了表数据存放在不同的文件中。能分区存储的还有：表、索引和大型对象数据。

SQL SERVER 2005 中，引入了表分区概念，当表中的数据量不断增大，查询数据的速度就会变慢，应用程序的性能就会下降，这时就应该考虑对表进行分区，当一个表里的数据很多时，可以将其分拆到多个的表里，因为要扫描的数据变得更少，查询可以更快地运行，这样操作大大提高了性能，表进行分区后，逻辑上表仍然是一张完整的表，只是将表中的数据在物理上存放多个表空间（物理文件上），这样查询数据时，不至于每次都扫描整张表

➤ 什么时候使用分区表：

- 1、表的大小超过 2GB。
- 2、表中包含历史数据，新的数据被增加到新的分区中。

➤ 表分区的优缺点

表分区有以下优点：

- 1、改善查询性能：对分区对象的查询可以仅搜索自己关心的分区，提高检索速度。
- 2、增强可用性：如果表的某个分区出现故障，表在其他分区的数据仍然可用；
- 3、维护方便：如果表的某个分区出现故障，需要修复数据，只修复该分区即可；
- 4、均衡 I/O：可以把不同的分区映射到磁盘以平衡 I/O，改善整个系统性能。

缺点：

分区表相关：已经存在的表没有方法可以直接转化为分区表。不过 Oracle 提供了在线重定义表的功能。

➤ 表分区操作三步走

3.1.2.1 创建分区函数

■ CREATE PARTITION FUNCTION xx1(int) AS RANGE LEFT FOR VALUES (10000, 20000);

注释：创建分区函数：myRangePF2，以 INT 类型分区，分三个区间，10000 以内为 A 区，1W-2W 在 B 区，2W 以上在 C 区。

3.1.2.2 创建分区架构

■ CREATE PARTITION SCHEME myRangePS2 AS PARTITION xx1 TO (a, b, c);

注释：在分区函数 XX1 上创建分区架构：myRangePS2，分别为 A,B,C 三个区间 A,B,C 分别为三个文件组的名称，而且必须三个 NDF 隶属于这三个组，文件所属文件组一旦创建就不能修改

3.1.2.3 对表进行分区

■ 常用数据规范--数据空间类型修改为：分区方案，然后选择分区方案名称和分区列列表，结果如图所示：

也可以用 sql 语句生成 CREATE TABLE [dbo].[AvCache](

[AVNote] [varchar](300) NULL,

[bb] [int] IDENTITY(1,1)

) ON [myRangePS2] (bb); --注意这里使用[myRangePS2]架构，根据 bb 分区

3.1.2.4 查询表分区

■ SELECT *, \$PARTITION.[myRangePF2](bb) FROM dbo.AVCache

	FlightNo	FlightDate	AVNote	bb	(无列名)
1	1	2012-01-01 00:00:00.000	1	1	1
2	1	2012-01-01 00:00:00.000	1	2	1
3	1	2012-01-01 00:00:00.000	1	3	1
4	1	2012-01-01 00:00:00.000	1	4	1
5	1	2012-01-01 00:00:00.000	1	5	2
6	1	2012-01-01 00:00:00.000	1	6	2
7	1	2012-01-01 00:00:00.000	1	7	2
8	1	2012-01-01 00:00:00.000	1	8	2
9	1	2012-01-01 00:00:00.000	1	9	2

这样就可以清楚的看到表数据是如何分区的了

3.1.2.5 创建索引分区

下面我们就可以通过语句将创建的表或者索引加入到新的文件组中了：

```
1 --向文件组 Index_Store 中添加索引
2 CREATE NONCLUSTERED INDEX idx_on_other_fileGroup ON Person.Address(PostalCode) ON Index_Store
3 --如果不指定文件组名，则添加到默认的 PRIMARY 文件组中
4 CREATE NONCLUSTERED INDEX idx_on_other_fileGroup ON Person.Address(PostalCode)
```

添加到一个专门放 Index 的文件组

3.1.3 分布式数据库设计

分布式数据库系统是在集中式数据库系统的基础上发展起来的，理解起来也很简单，就是将整体的数据库分开，分布到各个地方，就其本质而言，分布式数据库系统分为两种：1. 数据在逻辑上是统一的，而在物理上却是分散的，一个分布式数据库在逻辑上是一个统一的整体，在物理上则是分别存储在不同的物理节点上，我们通常说的分布式数据库都是这种 2. 逻辑是分布的，物理上也是分布的，这种也成联邦式分布数据库，由于组成联邦的各个子数据库系统是相对“自治”的，这种系统可以容纳多种不同用途的、差异较大的数据库，比较适宜于大范围内数据库的集成。

分布式数据库较为复杂，在此不作详细的使用和说明，只是举例说明一下，现在分布式数据库多用于用户分区性较强的系统中，如果一个全国连锁店，一般设计为每个分店都有自己的销售和库存等信息，总部则需要有员工，供应商，分店信息等数据库，这类型的分店数据库可以完全一致，很多系统也可能导致不一致，这样，各个连锁店数据存储在本地，从而提高了影响速度，降低了通信费用，而且数据分布在不同场地，且存有多个副本，即使个别场地发生故障，不致引起整个系统的瘫痪。但是他也带来很多问题，如：数据一致性问题、数据远程传递的实现、通信开销的降低等，这使得分布式数据库系统的开发变得较为复杂，只是让大家明白其原理，具体的使用方式就不做详细的介绍了。

3.1.4 整理数据库碎片

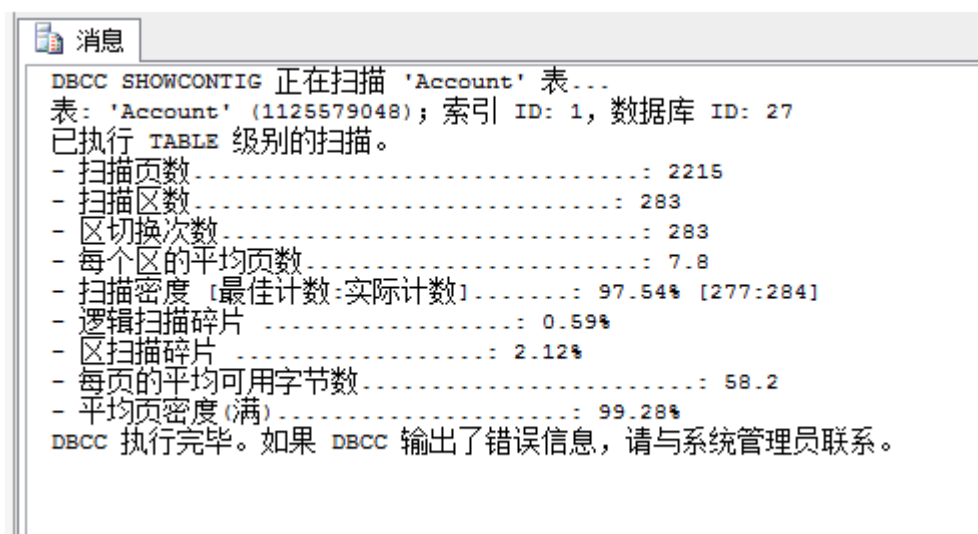
➤ 如果你的表已经创建好了索引，但性能却仍然不好，那很可能是产生了索引碎片，你需要进行索引碎片整理。

什么是索引碎片？

由于表上有过度地插入、修改和删除操作，索引页被分成多块就形成了索引碎片，如果索引碎片严重，那扫描索引的时间就会变长，甚至导致索引不可用，因此数据检索操作就慢下来了。

如何知道是否发生了索引碎片？

在 SQLServer 数据库，通过 DBCC SHOWCONTIG 或 DBCC SHOWCONTIG(表名)检查索引碎片情况，指导我们对其进行定时重建整理。



```
消息
DBCC SHOWCONTIG 正在扫描 'Account' 表...
表: 'Account' (1125579048); 索引 ID: 1, 数据库 ID: 27
已执行 TABLE 级别的扫描。
- 扫描页数.....: 2215
- 扫描区数.....: 283
- 区切换次数.....: 283
- 每个区的平均页数.....: 7.8
- 扫描密度 [最佳计数:实际计数].....: 97.54% [277:284]
- 逻辑扫描碎片.....: 0.59%
- 区扫描碎片.....: 2.12%
- 每页的平均可用字节数.....: 58.2
- 平均页密度(满).....: 99.28%
DBCC 执行完毕。如果 DBCC 输出了错误信息，请与系统管理员联系。
```

通过对扫描密度（过低），扫描碎片（过高）的结果分析，判定是否需要索引重建，主要看如下两个：
Scan Density [Best Count:Actual Count]-扫描密度 [最佳值:实际值]：DBCC SHOWCONTIG 返回最有用的一个百分比。这是扩展盘区的最佳值和实际值的比率。该百分比应该尽可能靠近 100%。低了则说明有外部碎片。

Logical Scan Fragmentation-逻辑扫描碎片：无序页的百分比。该百分比应该在 0%到 10%之间，高了则说明有外部碎片。

解决方式：

一是利用 DBCC INDEXDEFRAG 整理索引碎片

二是利用 DBCC DBREINDEX 重建索引。

两者区别调用微软的原话如下：

DBCC INDEXDEFRAG 命令是联机操作，所以索引只有在该命令正在运行时才可用，而且可以在不丢失已完成工作的情况下中断该操作。这种方法的缺点是在重新组织数据方面没有聚集索引的除去/重新创建操作有效。

重新创建聚集索引将对数据进行重新组织，其结果是使数据页填满。填满程度可以使用 FILLFACTOR 选项进行配置。这种方法的缺点是索引在除去/重新创建周期内为脱机状态，并且操作属原子级。如果中断索引创建，则不会重新创建该索引。也就是说，要想获得好的效果，还是得用重建索引，所以决定重建索引。

3.2 数据库表优化

3.2.1 设计规范化表，消除数据冗余

数据库范式是确保数据库结构合理，满足各种查询需要、避免数据库操作异常的数据库设计方式。满足范式要求的表，称为规范化表，范式产生于 20 世纪 70 年代初，一般表设计满足前三范式就可以，在这里简单介绍一下前三范式

先给大家看一下百度百科给出的定义：

第一范式（1NF）无重复的列

所谓第一范式（1NF）是指在关系模型中，对域添加的一个规范要求，所有的域都应该是原子性的，即数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。

第二范式（2NF）属性

在 1NF 的基础上，非码属性必须完全依赖于码[在 1NF 基础上消除非主属性对主码的部分函数依赖]

第三范式（3NF）属性

在 1NF 基础上，任何非主属性不依赖于其它非主属性[在 2NF 基础上消除传递依赖]

通俗的给大家解释一下（可能不是最科学、最准确的理解）

第一范式：属性(字段)的原子性约束，要求属性具有原子性，不可再分割；

第二范式：记录的惟一性约束，要求记录有惟一标识，每条记录需要有一个属性来做为实体的唯一标识。

第三范式：属性(字段)冗余性的约束，即任何字段不能由其他字段派生出来，在通俗点就是：主键没有直接关系的数据列必须消除(消除的办法就是再创建一个表来存放他们，当然外键除外)

如果数据库设计达到了完全的标准化，则把所有的表通过关键字连接在一起时，不会出现任何数据的复本(repetition)。

标准化的优点是明显的，它避免了数据冗余，自然就节省了空间，也对数据的一致性(consistency)提供了根本的保障，杜绝了数据不一致的现象，同时也提高了效率。

3.2.2 适当的冗余，增加计算列

数据库设计的实用原则是：在数据冗余和处理速度之间找到合适的平衡点

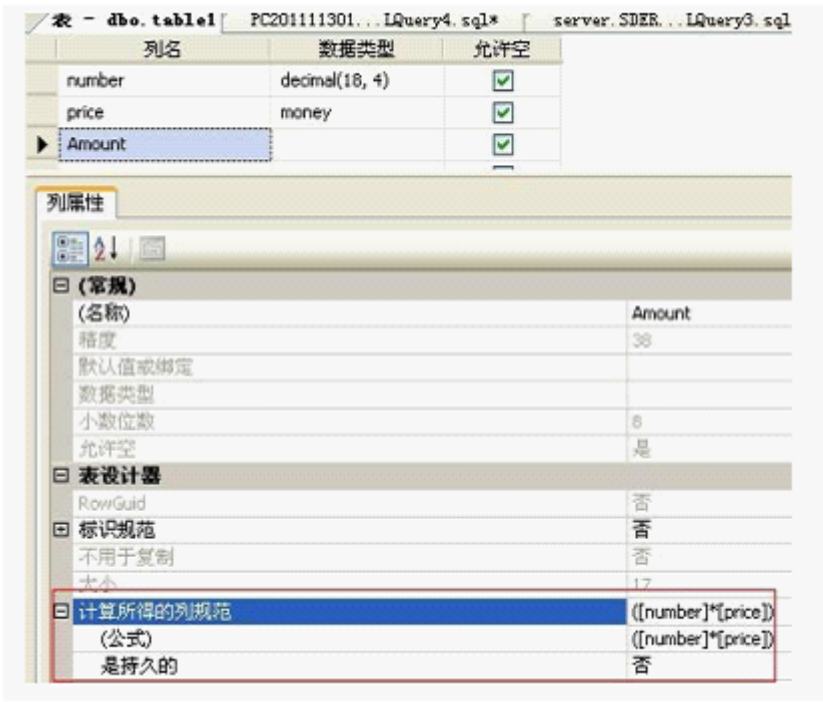
满足范式的表一定是规范化的表，但不一定是最佳的设计。很多情况下会为了提高数据库的运行效率，常常需要降低范式标准：适当增加冗余，达到以空间换时间的目的。比如我们有一个表，产品名称，单价，库存量，总价值。这个表是不满足第三范式的，因为“总价值”可以由“单价”乘以“数量”得到，说明“金额”是冗余字段。但是，增加“总价值”这个冗余字段，可以提高查询统计的速度，这就是以空间换时间的作法。合理的冗余可以分散数据量大的表的并发压力，也可以加快特殊查询的速度，冗余字段可以有效减少数据库表的连接，提高效率。

其中“总价值”就是一个计算列，在数据库中有两种类型：数据列和计算列，数据列就是需要我们手动或者程序给予赋值的列，计算列是源于表中其他的数据计算得来，比如这里的“总价值”

在 SQL 中创建计算列：

```
create table table1
(
    number decimal(18,4),
    price money,
    Amount as number*price --这里就是计算列
)
```

也可以再表设计中，直接手动添加或修改列属性即可：如下图



是否持久性，我们也要注意：

如果是'否'，说明这列是虚拟列，每次查询的时候计算一次，而且那么它是不可用来做 check, foreign key 或 not null 约束。

如果是'是'，就是真实的列，不需要每次都计算，可以再此列上创建索引等等。

3.2.3 索引

索引是一个表优化的重要指标，在表优化中占有极其重要的成分，所以将单独写一章”SQL 索引一步到位“去告诉大家如何建立和优化索引

3.2.4 主键和外键的必要性

主键与外键的设计，在全局数据库的设计中，占有重要地位。因为：主键是实体的抽象，主键与外键的配对，表示实体之间的连接。

主键：根据第二范式，需要有一个字段去标识这条记录，主键无疑是最好的标识，但是很多表也不一定需要主键，但是对于数据量大，查询频繁的数据库表，一定要有主键，主键可以增加效率、防止重复等优点。

主键的选择也比较重要，一般选择总的长度小的键，小的键的比较速度快，同时小的键可以使主键的 B 树结构的层次更少。主键的选择还要注意组合主键的字段次序，对于组合主键来说，不同的字段次序的主键的性能差别可能会很大，一般应该选择重复率低、单独或者组合查询可能性大的字段放在前面。

外键：外键作为数据库对象，很多人认为麻烦而不用，实际上，外键在大部分情况下是很有用的，理由是：外键是最高效的一致性维护方法

数据库的一致性要求，依次可以用外键、CHECK 约束、规则约束、触发器、客户端程序，一般认为，离数据越近的方法效率越高。

谨慎使用级联删除和级联更新，级联删除和级联更新作为 SQL SERVER 2000 当年的新功能，在 2005 作了保留，应该有其可用之处。我这里说的谨慎，是因为级联删除和级联更新有些突破了传统的关于外键的定义，功能有点太过强大，使用前必须确定自己已经把握好其功能范围，否则，级联删除和级联更新可能让你的数据莫名其妙的被修改或者丢失。从性能看级联删除和级联更新是比其他方法更高效的方法。

3.2.5 存储过程、视图、函数的适当使用

很多人习惯将复杂操作都放在应用程序层，但如果你要优化数据访问性能，将 SQL 代码移植到数据库上(使用存储过程，视图，函数和触发器)也是一个很大的改进原因如下：

1. 存储过程减少了网络传输、处理及存储的工作量，且经过编译和优化，执行速度快，易于维护，且表的结构改变时，不影响客户端的应用程序
- 2、使用存储过程，视图，函数有助于减少应用程序中 SQL 复制的弊端，因为现在只在一个地方集中处理 SQL
- 3、使用数据库对象实现所有的 TSQL 有助于分析 TSQL 的性能问题，同时有助于你集中管理 TSQL 代码，更好的重构 TSQL 代码

3.2.6 传说中的‘三少原则’

①：数据库的表越少越好

②：表的字段越少越好

③：字段中的组合主键、组合索引越少越好

当然这里的少是相对的，是减少数据冗余的重要设计理念。

3.2.7 分割你的表，减小表尺寸

如果你发现某个表的记录太多，例如超过一千万条，则应对该表进行水平分割。水平分割的做法是，以该表主键的某个值为界线，将该表的记录水平分割为两个表。

如果你发现某个表的字段太多，例如超过八十个，则垂直分割该表，将原来的一个表分解为两个表

3.2.8 字段设计原则

字段是数据库最基本的单位，其设计对性能的影响是很大的。需要注意如下：

- A、数据类型尽量用数字型，数字型的比较比字符型的快很多。
- B、数据类型尽量小，这里的尽量小是指在满足可以预见的未来需求的前提下的。
- C、尽量不要允许 NULL，除非必要，可以用 NOT NULL+DEFAULT 代替。
- D、少用 TEXT 和 IMAGE，二进制字段的读写是比较慢的，而且，读取的方法也不多，大部分情况下最好不用。
- E、自增字段要慎用，不利于数据迁移

3.3 程序操作优化

概述：程序访问优化也可以认为是访问 SQL 语句的优化，一个好的 SQL 语句是可以减少非常多的程序性能的，下面列出常用错误习惯，并且提出相应的解决方案

3.3.1 操作符优化

➤ IN、NOT IN 操作符

IN 和 EXISTS 性能有外表和内表区分的，但是在大数据量的表中推荐用 EXISTS 代替 IN 。

Not IN 不走索引的是绝对不能用的，可以用 NOT EXISTS 代替

➤ IS NULL 或 IS NOT NULL 操作

索引是不索引空值的，所以这样的操作不能使用索引，可以用其他的办法处理，例如：数字类型，判断大于 0，字符串类型设置一个默认值，判断是否等于默认值即可

➤ <> 操作符（不等于）

不等于操作符是永远不会用到索引的，因此对它的处理只会产生全表扫描。用其它相同功能的操作运算代替，如 `a<>0` 改为 `a>0 or a<0` `a<>''` 改为 `a>''`

➤ 用全文搜索搜索文本数据，取代 like 搜索

全文搜索始终优于 like 搜索：

(1) 全文搜索让你可以实现 like 不能完成的复杂搜索，如搜索一个单词或一个短语，搜索一个与另一个单词或短语相近的单词或短语，或者是搜索同义词；

(2) 实现全文搜索比实现 like 搜索更容易 (特别是复杂的搜索)；

3.3.2 SQL 语句优化

➤ 在查询中不要使用 `select *`

为什么不能使用，地球人都知道，但是很多人都习惯这样用，要明白能省就省，而且这样查询数据库不能利用“覆盖索引”了

➤ 尽量写 WHERE 子句

尽量不要写没有 WHERE 的 SQL 语句

➤ 注意 SELECT INTO 后的 WHERE 子句

因为 SELECT INTO 把数据插入到临时表，这个过程会锁定一些系统表，如果这个 WHERE 子句返回的数据过多或者速度太慢，会造成系统表长期锁定，堵塞其他进程。

➤ 对于聚合查询，可以用 HAVING 子句进一步限定返回的行

➤ 避免使用临时表

(1) 除非却有需要，否则应尽量避免使用临时表，相反，可以使用表变量代替；

(2) 大多数时候 (99%)，表变量驻扎在内存中，因此速度比临时表更快，临时表驻扎在 TempDb 数据库中，因此临时表上的操作需要跨数据库通信，速度自然慢。

➤ 减少访问数据库的次数：

程序设计中最好将一些常用的全局变量表放在内存中或者用其他方式减少数据库的访问次数

➤ 尽量少做重复的工作

尽量减少无效工作，但是这一点的侧重点在客户端程序，需要注意的如下：

A、控制同一语句的多次执行，特别是一些基础数据的多次执行是很多程序员很少注意的

B、减少多次的数据转换，也许需要数据转换是设计的问题，但是减少次数是程序员可以做到的。

C、杜绝不必要的子查询和连接表，子查询在执行计划一般解释成外连接，多余的连接表带来额外的开销。

D、合并对同一表同一条件的多次 UPDATE，比如

```
UPDATE EMPLOYEE SET FNAME=' HAIWER' WHERE EMP_ID=' VPA30890F'
```

```
UPDATE EMPLOYEE SET LNAME=' YANG' WHERE EMP_ID=' VPA30890F'
```

这两个语句应该合并成以下一个语句

```
UPDATE EMPLOYEE SET FNAME=' HAIWER' ,LNAME=' YANG'
```

```
WHERE EMP_ID=' VPA30890F'
```

E、UPDATE 操作不要拆成 DELETE 操作+INSERT 操作的形式，虽然功能相同，但是性能差别是很大的。

F、不要写一些没有意义的查询，比如

```
SELECT * FROM EMPLOYEE WHERE 1=2
```

3.3.3 WHERE 使用原则

1)在下面两条 select 语句中：

```
select * from table1 where field1<=10000 and field1>=0;
```

```
select * from table1 where field1>=0 and field1<=10000;
```

如果数据表中的数据 field1 都>=0，则第一条 select 语句要比第二条 select 语句效率高得多，因为第二条 select 语句的第一个条件耗费了大量的系统资源。

第一个原则：在 where 子句中应把最具限制性的条件放在最前面。

2)在下面的 select 语句中：

```
select * from tab where a=... and b=... and c=...;
```

若有索引 index(a,b,c)，则 where 子句中字段的顺序应和索引中字段顺序一致。

第二个原则：where 子句中字段的顺序应和索引中字段顺序一致。

以下假设在 field1 上有唯一索引 I1，在 field2 上有非唯一索引 I2。

3) select field3,field4 from tb where field1='sdf' 快

```
select * from tb where field1='sdf' 慢，
```

因为后者在索引扫描后要多一步 ROWID 表访问。

```
select field3,field4 from tb where field1>='sdf' 快
```

```
select field3,field4 from tb where field1>'sdf' 慢
```

因为前者可以迅速定位索引。

```
select field3,field4 from tb where field2 like 'R%' 快
```

```
select field3,field4 from tb where field2 like '%R' 慢，
```

因为后者不使用索引。

卧槽，还有 10T 计算机和运营资料、以及各种红包和实物抽奖！同时加入程序员交流群，绝对没有广告只有交流和学习，毫无套路！详情：<https://t.1yb.co/3WEI> 也可以加我微信：tsdabear 快速入群！

4) 使用函数如：

```
select field3,field4 from tb where upper(field2)='RMN' 不使用索引。
```

如果一个表有两万条记录，建议不使用函数；如果一个表有五万条以上记录，严格禁止使用函数！两万条记录以下没有限制。