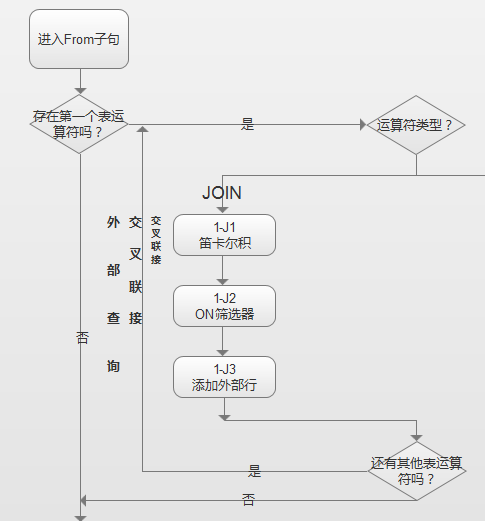
**逻辑查询处理**

SQL与其他编程语言不同的最明显特征是代码的查询顺序。在大多数语言中，代码是按照编写顺序来处理的；但在SQL中，第一个要处理的子句是FROM子句，尽管SELECT是第一个出现，但它几乎都是最后才处理。

每一步都会生成一个虚拟表，该虚拟表会称为下一步的输入，这些虚拟表对于用户是不可见的。如果在查询中没有指定某一个子句，则会跳过相应的步骤。

**步骤1：From阶段**

****

From阶段负责标识要查询的表，如果指定了表运算符，这个阶段还要按照从左到右的顺序对这些运算符进行处理。

**步骤1-J1：执行笛卡尔积（交叉联接）**

这个阶段对联接涉及的两个表执行笛卡尔积运算，生成虚拟表**VT1-J1**。这个虚拟表为左表行和有表行的每一个可能的组合包含一行数据。如果左表包含n行，右表包含m行，**VT1-J1**包含m \* n行。

**步骤1-J2：应用ON筛选器（联接条件）**

ON筛选器是查询中可以指定的三种筛选器（ON，Where，Having）中的第一个。ON筛选器作用于虚拟表（VT1-J1）中的所有行。只有使<on\_predicate>为True的那些行，才会包含在这一步返回的虚拟表（**VT1-J2**）。

**步骤1-J3：添加外部行（Outer Row）**

这一步只在外联接（outer join）中发生。对于外联接，通过为其指定一种外联接类型（LEFT ,RIGHT或FULL），就把一个或两个表标记为保留表（preserved table）。把表标记为**保留表**，即表示希望返回该表的所有行，即使ON筛选器过滤掉了这些行。

**左外联接**是把左表标记为保留表，**右外联接**是把右表标记为保留表，**完全外联接**是把两个表标记为保留表。

步骤1-J3返回VT1-J2中的行，以及保留表在1-J2中过滤掉的行。这些新行被称为**外部行**（outer row）。外部行中非保留表的列值被赋值为NULL，最后生成**VT1-J3**。

注意：*如果From子句中有多个表运算符，则按照从左到右的顺序进行处理。每个表运算符的结果作为下一个表运算符的左输入。最终生成的虚拟表将作为下一阶段的输入。*

**步骤2：Where阶段**

对上一步返回的虚拟表中返回的所有行应用Where筛选器。只有让<where\_predicate>逻辑条件为True的行，才会组成这一步要返回的虚拟表**VT2**。

注意：由于还有对表进行分组，所以在Where子句中不能使用聚合。

对于包含外联接的查询，有一个问题是让人困惑的，**何时使用ON筛选器？何时Where筛选器？**

两者的主要区别是：ON在添加外部行之前执行，而Where在添加外部行之后执行。ON中筛选掉的行不是最终的，因为步骤1-J3会将这些行再添加回来；相反Where筛选器对行的删除是最终的。

需要注意的是，只有在使用外联接时，ON和Where才有这种逻辑区别。当使用内连接时，它们是一致的。

**步骤3：GROUP BY阶段**

在Group By阶段，根据<group\_by\_specification>指定的列表，将上一步返回的虚拟表中的行分配到各个组，这一组列被称之为**分组集**。

*例如：GROUP BY C.customerid中C.customerid就是一个分组集。*

这个阶段将上一步返回的虚拟表中的行按组进行分组。由分组集中的所有属性值的每个唯一组合标识出一个组。最终得到的虚拟表VT3由两部分组成：分组后原来的行(原始信息)，组标识符（组信息）。

|  |  |  |  |
| --- | --- | --- | --- |
| 组信息  C.customerid | 原始信息  C.customerid | C.city | O.orderid |
| Jack | Jack  Jack | ShangHai  ShangHai | 1  2 |
| Lucy | Lucy  Lucy  Lucy | Chongqing  Chongqing  Null | 3  4  5 |
| Ajax | Ajax | Wuhan | 6 |

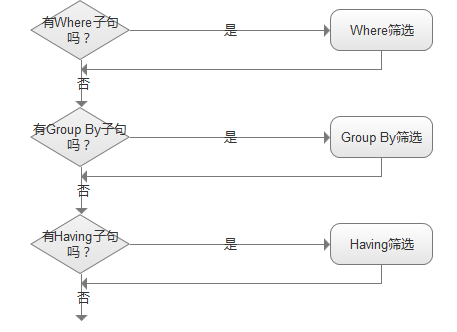
最终，Group By子句的查询将为每一个组生成一行。所以在查询中指定了Group By子句，则后面的所有步骤（Having，Select等）只能指定每个分组上的标量（单个）值表达式。这些表达式可以是Group By列表中的列，表达式（C.customerid），以及聚合行数（Count(C.customerid)）。

这个阶段会认为两个NULL值是相等的。也就是说，所有的NULL值将被分配到同一个组中。

**步骤4：HAVING阶段**

HAVING筛选器用于对上一步返回的虚拟表中的组进行筛选。只有使<having\_predicate>逻辑条件取值为True的组，才会称为这一步返回的虚拟表（VT4）的一部分。

注意：子查询不能作为聚合函数的输入。



**步骤5：SELECT阶段**

虽然SELECT子句出现在查询的最前面，但却放在第五步处理。

**步骤5-1：计算表达式**

SELECT列表中的表达式可以返回上一步得到的虚拟表的基础列，也可以是对这些基础列的操作。如果查询是一个聚合查询，就只能引用上一步返回的虚拟表中的组信息。如果要引用原始信息，就必须对它们进行聚合运算。这个阶段得到的是虚拟表**VT5**。

**步骤5-2：应用Distinct子句**

如果在查询中指定了Distinct子句，将从上一句返回的虚拟表中删除重复的行，并生成虚拟表**VT5-2**。

**步骤5-3：应用Top选项**

Top选项是T-SQL特有的一项功能，允许指定要返回的指定行数或百分比，根据查询的Order by子句来选择指定的数量的行。最终生成虚拟表**VT5-3**。

如果没有指定Order By子句或者未指定With Ties选项，或是根本没有指定Order By子句，那么Top查询就是非确定的。也就是说，返回的行只是SQL Server碰巧在物理上最先访问到的行，因此可能会产生不同的结果，但都可以认为是正确的。

**步骤5：ORDER BY阶段**

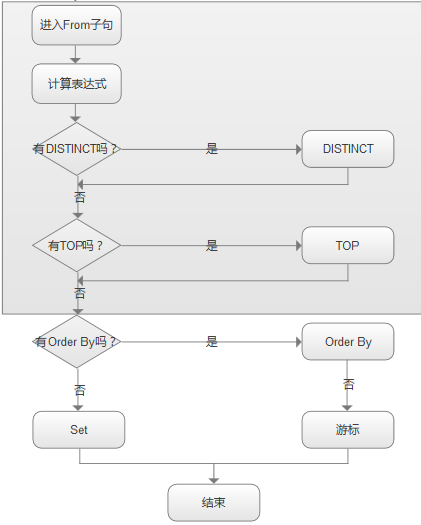
这一步按Order By子句中的列名列表对上一步返回的行进行排序，返回游标**VT6**。Order By子句是唯一可以重用SELECT列表中创建列名的步骤。

如果指定了Distinct，则Order By子句中的表达式只能访问上一步返回的虚拟表VT5。如果没有指定Distinct则可以指定任何在Select子句中的表达式。

如果查询中带有Order By子句，则不能用这样的查询来定义表表达式。表表达式包括：视图，内联表值函数，派生表和公用表表达式（CTE）。

除非真的需要有序行，否则不要指定Order By子句，因为排序需要耗费一定的资源。SQL Server需指定有序的索引扫描或sort运算符。

Order By子句认为两个NULL值是相等的。也就是说所有的NULL值是会排序到一起。T-SQL认为NULL值比已知值小。



查询优化

在处理性能问题时，数据库专家倾向于关注系统的技术层面，如资源队列、资源利用率等。而用户只把性能问题简单地认为是等待时间，他们发出一个请求，然后等待返回结果。用户通常认为 在交互请求后，超过三秒才得到响应，就算存在性能问题了。他们并不真关心平均每个磁盘转轴上 有多少个命令在等待，或者缓存命中率（cache hit ratio)是多少，也不关心阻塞，CPU利用率，缓 存中数据页的平均停留时间（page life expectancy)等。他们只关心等待时间，这正是优化性能应该着手的地方.

我推荐的一种使用自顶向下方法的优化论。这种方法首先分析实例级的等待时间，再通过一系 列步骤将其不断细化，直到找出系统中导致大量等待的进程组件。一旦找出这些令人讨厌的进程, 就可以集中优化它们了。以下是这种方法论的主要步骤：

1. 分析实例级别等待
2. 关联等待和队列
3. 确定方案
4. 细化到数据库/文件级
5. 细化到进程级
6. 优化索引/查询

实例级别等待

优化方法论的第一步是在实力级别上找出什么类型的等待占用了大部分的等待时间。这可以通过查询动态管理图(DMV,dynamic management view) sys.dm\_os\_wait\_stats来完成。该DMV包含了400多种等待类型，大多数类型至少在SQL Server联机丛书中至少有一段文字说明。可以将这个DMV作为优化的起点。而其他一些优化工具则不适合在开始使用。因为它们提供的信息量太大，让你无法入手，很容易迷失其中。

SELECT

wait\_type, --等待类型

waiting\_tasks\_count, --该类型等待的数量

wait\_time\_ms, --该类型总的等待时间

max\_wait\_time\_ms, --最大等待时间

signal\_wait\_time\_ms --从收到信号到开始运行之间的时间差

FROM sys.dm\_os\_wait\_stats

ORDER BY wait\_type;

这个DMV会在服务器最后一次重启时重置。

Signal wait time ms 属性就是表示从线程收到资源可用的信号开始，到线程得到CPU，开始使用资源的为止经历的时间。可以想到，这个数值很高，就说明CPU存在问题。

在各种等待类型中，可以找到与锁、闩锁（latch，一种轻型锁）、I/O (包括I/O闩锁）、并行查 询（parallelism)、事务口志、内存、编译、OLEDB (链接服务器和其他OLEDB组件）等相关的等待。

通常，你会忽略某些类型的等待，例如，睡眠等待类型 (SLEEP，当线程被挂起，未执行任何操作时发生)、队列等待类型（QUEUE，当工作线程空闲，等待分配任务时发生），以及在SQL Server联机丛书中特别描述的一 些并不代表出现问题的等待类型（如 CLR\_AUTO\_EVENT、REQUEST\_FOR\_DEADLOCK\_ SEARCH 等等）。要确保过滤掉不相关的等待.以免它们影响你的计算。

**等待的类型**

**资源等待** 当工作者请求访问由于某个其他工作人员正在使用该资源或尚未可用的资源而无法访问的资源时，会发生资源等待。资源等待的示例是锁，锁存器，网络和磁盘I / O等待。锁定和锁存等待是等待同步对象

**队列等待**  
队列等待工作空闲时等待分配工作。队列等待最常见于系统后台任务，例如死锁监视器和已删除的记录清理任务。这些任务将等待将工作请求放入工作队列。即使没有新的数据包放入队列，队列等待也可能周期性地变为活动状态。

**外部等待**  
当SQL Server工作程序等待外部事件（如扩展存储过程调用或链接服务器查询）完成时，会发生外部等待。当您诊断阻塞问题时，请记住外部等待并不总是暗示工作者是空闲的，因为工作人员可能会主动运行某些外部代码。

|  |  |  |
| --- | --- | --- |
| 类型 | 描述 | 影响 |
| LAZYWRITER\_SLEEP | 在惰性写入器任务被挂起时发生。这是等待的后台任务所花费的时间的度量。当您在寻找用户档位时，请不要考虑此状态。 | 无 |
| SQLTRACE\_INCREMENTAL\_FLUSH\_SLEEP | 此等待类型是线程在检查SQL跟踪信息以刷新到跟踪文件之间休眠的时间 | 无 |
| SLEEP\_TASK | 在等待发生通用事件时任务休眠时发生。 | 无 |
| SLEEP\_SYSTEMTASK | 在等待tempdb完成启动时在后台任务启动期间发生。 | 无 |
| FT\_IFTS\_SCHEDULER\_IDLE\_WAIT | 此等待类型是线程正在等待在全文搜索后台任务队列中输入某些工作的时间。 | 无 |
| SERVER\_IDLE\_CHECK | 在资源监视器尝试将SQL Server实例声明为空闲或尝试唤醒时，在SQL Server实例空闲状态同步期间发生。 | 无 |
| CLR\_AUTO\_EVENT | 当某任务当前正在执行公共语言运行时 (CLR) 执行并且正在等待特殊的自动事件启动时出现。 通常会出现长时间等待，这并不意味着出现问题。 | 无 |
| REQUEST\_FOR\_DEADLOCK\_SEARCH | 在死锁监视器等待开始下一次死锁搜索时出现。 在两次死锁检测之间可能出现该等待，长时间等待此资源并不指示出现问题。 | 无 |
| SQLTRACE\_BUFFER\_FLUSH | 当某任务正在等待后台任务将跟踪缓冲区每隔四秒刷新到磁盘时出现。  适用于: SQL Server 2008 R2仅。 | 无 |
| IOLATCH相关 | 在任务等待 I/O 请求中缓冲区的闩锁时发生。  长时间的等待可能指示磁盘子系统出现问题。 | 有 |
| ASYNC\_NETWORK\_IO | 当任务被阻止在网络之后时出现在网络写入中。 验证客户端是否正在处理来自服务器的数据。 | 有 |
| BROKER\_TO\_FLUSH | 此等待类型是指线程正在等待写入非活动传输对象（跟踪对话框的消息传输）到tempdb工作表以节省内存。  这种等待类型是我通常在进行等待统计分析时过滤掉的良性等待类型。此等待类型的等待时间将每1秒增加1秒。 | 无 |
| PAGEIOLATCH\_SH | 在任务等待 I/O 请求中缓冲区的闩锁时发生。 闩锁请求处于“共享”模式。 长时间的等待可能指示磁盘子系统出现问题。 | 有 |
| CMEMTHREAD | 当某任务正在等待线程安全内存对象时出现。 当多项任务尝试分配来自同一个内存对象的内存而导致出现争用时，便可能延长等待时间。 | 有 |
| CXPACKET | 当同步查询处理器交换迭代器，并生成和使用行时出现并行查询计划。 如果等待太久，无法通过优化查询（如添加索引）来减少等待时间，请考虑调整并行度的开销阈值或降低并行度。 | 有 |
| WRITELOG | 此等待类型是指线程正在等待异步I / O将日志块写入磁盘。  在以下情况下将日志块写入磁盘：   * 事务提交（除非在SQL Server 2014及更高版本中设置为延迟持久） * 日志块的最大大小为60Kb * 正在将数据文件页面写入磁盘，并且预写日志记录强制将当前日志块强制到磁盘（因为日志块包含描述正在写出的数据文件页面的更改的最新日志记录） * 该*sp\_flush\_log*  PROC执行（在SQL Server 2014以上，对于延迟耐久性日志管理）   将*WRITELOG*  视为服务器上最重要的等待之一是非常普遍的  ，但是你仍然可以做很多事情来尝试缓解它们。  您可以做很多事情来减少  *WRITELOG*等待和等待时间，包括：   * 减少生成的事务日志量 * 减少日志刷新的频率 * 减少/消除同步HA技术无意中听到的日志刷新 * 对于可用性组，远程日志副本会导致[*HADR\_SYNC\_COMMIT*](http://www.sqlskills.com/help/waits/hadr_sync_commit/)等待 * 升级到2012或更高版本，将max-outstanding-log-writes从32提高到112 * 将事务日志放在I / O子系统的最快部分 * 考虑实施延迟耐久性（在2014+）或内存中的OLTP / Hekaton   参考：<https://www.sqlskills.com/help/waits/writelog/> | 有 |
| OLEDB | 这种等待类型是一个线程正在等待来自OLE DB提供程序的数据，该数据库在内部用于DBCC CHECK \*命令和DMV之类的东西，外部用于链接服务器通信和某些SSIS包之类的东西。 | 有 |
| LCK\_M\_IS | 此等待类型是指线程正在等待获取资源上的Intent Shared锁，并且在资源上向不同线程授予不兼容模式的至少一个其他锁。  有关排除锁定等待的一般指导：   * 无法从*sys.dm\_os\_wait\_stats*输出中确定锁资源  。您可以从  *sys.dm\_os\_waiting\_tasks*（使用  [我的脚本](http://www.sqlskills.com/blogs/paul/updated-sys-dm_os_waiting_tasks-script-2/)）或查看*sys.dm\_tran\_locks*的  *resource\_description*字段中*查看资源*，其中  *request\_status*是'WAIT'。 * 您可以使用阻止的进程报告来获取有关等待指定阈值锁定的查询的更多详细信息（请参阅[此处](http://michaeljswart.com/2011/05/when-to-use-blocked-processes-reports/" \t "_blank)）。 * 看看阻塞链的头部（即持有锁定所有人的锁的线程）使用脚本（大量在线可用 - 我没有首选的）。那个帖子在等什么？修复等待可能有助于解决阻塞问题。例如，一个线程可能持有锁并提交事务，但是有一个带有慢速I / O子系统的同步镜像，因此镜像日志写入需要很长时间，使得事务提交需要更长时间，并且锁需要更长的时间释放，导致阻塞。 * 查找锁定升级，其中*UPDATE*事务已升级为表X锁定，从而导致广泛阻塞。 * 查找导致表锁定的索引操作，并考虑使用联机索引操作（或者如果已经使用它们，请考虑*2014*年以上的*WAIT\_AT\_LOW\_PRIORITY*功能）。 * 查找指定*TABLOCK*（导致表共享锁）或*TABLOCKX*（导致表独占锁）提示的代码。 * 查找将导致获取锁定然后等待用户输入或长时间未提交事务的应用程序代码。 * 请考虑创建非聚簇索引以从基础堆/聚簇索引中删除行锁。 * 考虑使用快照隔离或读取提交的快照隔离，以允许读者不采用S / IS锁定并减少阻塞。 * 检查正在使用的正确隔离级别为*REPEATABLE\_READ*，*SERIALIZABLE*将保持S / IS锁定，直到事务结束。 * 检查是否意外使用*SERIALIZABLE*隔离级别，使用分布式事务或错误作用域的.Net *TransactionScope*对象。   *LCK\_M\_IS*的具体指导等待：   * 对于Intent Shared锁，资源可以是页面，分区或表。 * 常见的阻止程序是发生锁升级的表X（独占）锁，或索引构建/重建的SCH\_M（模式修改）锁。 | 有 |
| LCK\_M\_IX | 当某任务正在等待获取意向排他 (IX) 锁时出现。  LCK\_M\_IX的具体指导 等待：   * 对于Intent Exclusive锁，资源可以是页面，分区或表。 * 常见的阻止程序是发生锁升级的表X（独占）锁，或索引构建/重建的SCH\_M（模式修改）锁。 * 如果阻止程序持有表S锁，请调查阻塞线程为何具有该锁（例如，在限制性隔离级别中使用TABLOCK提示或锁升级）。 | 有 |

**性能瓶颈**

在很多情况下，你会发现与**I/O相关**的等待是最常见的等待（例如，IOLATCH等待〉，其中有 几个原因。I/O通常是数据处理操作所涉及的最昂贵的资源。而且，当査询或索引没有经过良好地设计或优化时，结果一般会造成大量的I/O。此外，当客户在考虑计算机的性能时，他们通常只关注CPU 和内存，而不会对I/O子系统予以足够的重视。其实，数据库系统需要非常强健的I/O子系统。

对于与**网络相关**的等待（例如，ASYNC\_NETWORK\_IO),它们的值过高，则表明可能存在 网络问题。不过，这也可能表明客户端没有足够快地使用SQL Server发送给它的数据。

—些系统不需要访问数据的大部分.只需要非常频繁地访问数据的一少部分。联机事务处理(OLTP)环境通常就是这样的情况，它们的一些存储过程只访问数据的一小部分，但调用的次数非常频繁。在这样的环境中.代码的编译和重新编译可能就是产生瓶颈的主要原因，这时你可能会发现**信号等待（与CPU有关）**的值非常高。如果使用大量的特定査询，而不是用存储过程 和预先编译好的语句，就会因为这些特定的执行计划而消耗大最内存，这时你通常会发现 CMEMTHREAD等待类型的值非常高（当某任务正在等待线程安全的内存对象时出现这种等待）。

对于使用了太多**线程的并行查询**计划，也可能会出现问题。为了等待其他线程完成它们的工作 (CXPACKET等待），一些线程可能会等待很长时间，才可以继续工作；系统作为一个整体可能 无法提供域佳的吞吐量。这时可以考虑降低系统的最大并发度。不过，也要注意，CXPACKET等待类型有时也是其他原因造成的症状（例如，因为缺乏适当的索引而引起的过度的I/O),这时你 将会发现与I/O相关的等待存在较高的值。

OLTP系统还涉及大最小范围内的数据修改.在这种环境下事务日志志经常会成为一个瓶颈。当 SQL Server不能足够快地写完日志时，你通常会看到WRITELOG等待类型中有较高的值.

因为所有的临时表都在tempdb数据库中创建（无论是由执行计划隐式创建还是显式创建）, 所以tempdb数据库也吋能成为一个作常严重的瓶颈，SQL Server还利用tempdb的空间执行一些其他操作。tempdb数据库中的性能问题可能会导致与I/O相关的等待或其他等待出现较高的值。闩锁等待（例如.PAGE\_LATCH\_UP)出现较高的值，则表示在内部结构（如IAM、GAM、SGAM和 PFS页面）上出现了争用.原因可能是为临时表频繁地分配页面、向堆空间插入人置数据等。不正确的文件布局（layout）也可以导致这样的争用。

OLEDB等待类型代表与服务器调用、BULK INSERT、Full Text等相关的等待。不过，因为不 能生成OLEDB调用；因此，当调用开始时，就开始等待，当调用结束时，就结束等待。这意味着，这种等待类型中的较高的值不一定表示存在性能问题。

偶尔，你也会发现系统存在一呰与并发相关的（阻塞）问题.在这种情况下，锁等待（LCK) 的值将会比较高。

关联等待和队列

在实例级找出重量级等待后，就应该把它们和队列关联起来，以找出有问题的资源。这一步主要使用性能监视器来完成的。

SQL Server也提供一个名为sys.dm\_os\_performance\_counters的DVM，其中包含了性能监视器中所有与SQL Server实例对象相关的计数器。

以下是一些常用的参数指标：

1. **SQL Server Buffer: Buffer Cache Hit Ratio**

　　这是一个很重要查看内存是否不足的参数。SQL Server Buffer中的计数器Buffer Cache Hit Ratio用来指出SQLServer从缓存中而不是磁盘中获得数据的频率。sqlserver会将某些查询过的数据缓存在内存中用于以后再次查询使用。当一个查询A进来了以后数据库会编译这个sql看看需要哪些数据，然后执行计划首先去内存中找看是否有这次查询所需要的数据，如果这个同样的sql刚才已经执行过了或者该表的数据已经缓存在内存中，但是却没有在内存中找到数据，那就有可能是因为内存不足引起内存挤压将缓存数据写回硬盘或者释放掉来提供数据库其他请求来使用。一般来说oltp的系统，这个值最起码也应该在90%以上，理想值是99%。如果这个值低于90%，那建议你应该添加内存了。

1. **Memory: Pages/sec**

　　这个也是监控内存是否不足的一个比较重要的参数。这个计数器记录的是每秒钟内存和磁盘之间交换的页面数。频繁的交换页面就会消耗更多的io，这会影响到服务器的性能。打个比方，超市有一个货架上边摆满了新进的各种商品a、b、c，当你去超市想买a的时候直接去货架就能拿到a，方便的很，当顾客进超市逛一圈以后跟你说我怎么没有发现旧商品d呢，我就想买这个d，然后工作人员就会去仓库把商品d拿出来摆放到货架上供下次顾客来买。但是货架摆满了怎么办呢，只能将时间长没有人问津的a下架放到仓库然后空出来地方摆放d，但是下次另一个顾客来了又有想要购买a的意向，工作人员就得再次把a拿出来替换掉货架上的d。其实内存就是这个货架，硬盘就是仓库。因为货架太小了，导致只能频繁的更换货架上的商品来提供正常的运营，想减少反复来回搬运产生的io开销，只能换个更大的货架来满足需求。

　　如果服务器上只跑的sqlserver，那这个指标的理想范围应该是0-20之间，偶尔超过20的话影响不大，如果这个值频繁的超过20，那说明你的这台服务器可能需要加内存了。

当然这个指标要配合着上一个指标**Buffer Cache Hit Ratio**来看，如果上一个指标缓冲命中一直在99%或者更高，而这个期间内你的页交换一直在20以上，那意味着不仅仅是内存不足，而且其他的程序占用了系统内存。

1. **Memory: Available Bytes**

　　另一个监控内存情况的计数器就是这个。这个值最少最少也得大于5M，因为sqlserver需要始终维持5-10m的自由内存用于分配，当这个值低于5m的时候，那sqlserver可能会因为缺少内存而产生性能瓶颈。

1. **Physical Disk: % Disk Time**

　　这个计数器记录的是磁盘的繁忙程度（是整个磁盘阵列或者物理磁盘的繁忙程度）。理论上这个值应该低于55%，如果持续的高于55%，那说明这台服务器上可能有io瓶颈。

如果只是偶尔的出现几次，那不必担心，但是可以对应的找到这个时间点，数据库正在干嘛执行了哪些语句，对应的优化一下。

1. **Physical Disk: Avg. Disk Queue Length**

　　 这是一个比较重要的查看磁盘io情况的指标。理论上每个物理磁盘的值不应该超过2。当然这个值是需要计算的，比如用4块物理盘做了个raid10，此时在一个监控周期内磁盘队列的均值是10，那每块磁盘的队列值就是10/4=2.5，那么就可以说这个磁盘阵列存在i/o瓶颈了。这个跟之前的disktime指标一样，偶尔出现不必担心，如果长时间出现，那就得着手考虑解决磁盘的io性能问题了。

1. **Processor: % Processor Time**

　　这是监控cpu情况的一个指标（类似于disk time）。这个是观察cpu利用率的一个关键参数。如果Processor Time计数器的值持续超过80%，说明cpu存在瓶颈问题。如果只是偶尔出现，那说明可能是这个时间点有个特别消耗cpu的查询，可以在下一次这个时间点来临的时候尝试抓一下sql并且优化它。如果在某一个时间点以后cpu一直飙高，常见的情况就是：1.突然间的高并发2.索引重整3.突然一个经常使用的数据量特别大的索引失效了4.死锁5.其他好多好多。先找到问题所在，在处理掉它。

1. **System: Processor Queue Length**

　　这个指标类似于disk queue length，也是算单个cpu的。单个cpu不能超过2，比如你是2u的机器，那这个值不应该超过4，如果在一个监控周期内持续性的超过4，那就可能出现cpu瓶颈了。

确定行动方案

在找出主要等待类型和这些等待所涉及的资源之后，接下来的这一步是优化过程中承上启下的一步。根据目前掌握的信息，你将确定下一步研究的方向。在我们的例子中，就是要找出导致I/O，并行查询，与网络的等待，以及与事务日志等待的原因。

与I/O相关的等待需要我们进一步细化到数据库级别。并行等待(CXPACKET)，通过降低服务器的最大并行数，可以缓解这个问题。网络等待的值较高，表示网络带宽有问题，但可能表示其他问题

更多行动方案可以参考《Microsoft SQL Server 2008技术内幕：T-SQL查询》

细化到数据库/文件级别

优化处理的下一步是细化到数据库/文件级别。这一步要找出哪些数据库占用了大部分等待开销。在数据库内，还需要再细化到文件类型（数据/日志），因为文件的类型决定了要采取的行动方案。在数据库/文件级别分析I/O信息的一种工具是动态管理函数dm\_io\_virtual\_file\_stats。这个函数接收一个数据库ID和文件ID作为输入参数。返回与该文件相关的I/O信息。如果将这两个指定为NULL，则返回所有的信息。

-- 分析数据库I/O

WITH DBIO AS

(

SELECT

DB\_NAME(IVFS.database\_id) AS db,

MF.type\_desc,

SUM(IVFS.num\_of\_bytes\_read + IVFS.num\_of\_bytes\_written) AS io\_bytes,

SUM(IVFS.io\_stall) AS io\_stall\_ms

FROM sys.dm\_io\_virtual\_file\_stats(NULL, NULL) AS IVFS

JOIN sys.master\_files AS MF

ON IVFS.database\_id = MF.database\_id

AND IVFS.file\_id = MF.file\_id

GROUP BY DB\_NAME(IVFS.database\_id), MF.type\_desc

)

SELECT db, type\_desc,

CAST(1. \* io\_bytes / (1024 \* 1024) AS NUMERIC(12, 2)) AS io\_mb,

CAST(io\_stall\_ms / 1000. AS NUMERIC(12, 2)) AS io\_stall\_s,

CAST(100. \* io\_stall\_ms / SUM(io\_stall\_ms) OVER()

AS NUMERIC(10, 2)) AS io\_stall\_pct,

ROW\_NUMBER() OVER(ORDER BY io\_stall\_ms DESC) AS rn

FROM DBIO

ORDER BY io\_stall\_ms DESC;

结果输出了数据库名称，文件类型，以兆字节为单位的总I/O，以秒为单位的I/O等待时间，每个I/O等待时间占整个系统总等待时间的百分比，以及按I/O等待时间排序的行号。

假如数据库日志出现了较高的等待，你首先应该检查日志的配置是否合理，也就是说，把它放到不受干扰的磁盘驱动器上。记住数据日志是按照顺序写入的，所以把它分到多个磁盘上并没有什么帮助。对于导致密集日志操作的进程，可以通过减少日志操作量来优化这些进程。

至于tempdb，有许多操作都可能会给它带来压力，让它成为系统的严重瓶颈。

细化到进程级别

使用SQL Server自带的工具SQL Server Profiler。但有几点需要特别注意：

1. 不要把数据写入到表中，这样会严重影响性能。把数据写入到磁盘文件中是最快的选择，加载时我们可以使用fn\_trace\_gettable函数，用SELECT…INTO语句批量导入。
2. 追踪会产生大量I/O操作。不要把追踪文件访问和数据库文件相同的文件的磁盘上，理想情况是为追踪文件单独配置一个硬盘。
3. 慎重选择事件类和数据列，只跟踪需要的信息，移除所有默认的和不必要的对象。
4. 使用跟踪筛选功能，只选择相关的事件。

至于事件类，如果系统中的大部分操作都是由调用存储过程而引起的，而且每个存储过程只调用较少或有限的操作，则可以跟踪SP:Completed的事件类。然后可以根据存储过程对数据进行聚合。类似地，如果大部分操作是由调用批处理而引起的，而且每个批处理只有少量的操作，则可以跟踪SQL:BatchCompleted事件类。然而，如果每个过程调用了许多操作，就应该跟踪SP:StmtComplete事件类，以捕获每个存储过程调用的独立语句。如果操作是作为即席批处理（ad-hoc batch)而提交的（就像我们的例子一样），则应该跟踪SQL:StmtComplete事件类。不过要记住，跟踪每个语句的事件类时，可能会产生大量跟踪信息。对被跟踪的SQL Server实例有影响，所以要尽可能将这样的跟踪限制在较短的周期内，只收集具有代表性的工作负荷。最后，如果操作是作为远程过程调用(remote procedure call)而提交的，则应该跟综RPC:Completed事件类。注意，跟踪的所有亊件类都 是Completed事件类，而不是Starting事件类。只有Completed事件类才会携带性能信息，如Duration、 CPU、Reads和Writes,因为这些值在每个事件开始时都是未知的。

至于数据列，主要需要包含实际T-SQL代码的TexData列，以及一些与性能有关的计数器（例如，Duration列）。用户通常把等待认为是性能问题，而Duration代表该事件持续的时间（elapsed < time)。如果正在专门定位与I/O 相关的问題，则可以分析Reads和Writes列，我还喜欢跟踪 RowCounts列，特别是当査找与网络相关的问题时。对于把结果集返回给客户端的杳询，如果该查询在这个计数器上的数值比较大，则表明可能存在网络压力。

SQL Server 2008提供了一个名为sys.dm\_exec\_query\_stats的DMV，返回有关缓存的查询计划中的聚合性能统计信息 SQL Server。 缓存计划中的每个查询语句在该视图中对应一行，并且行的生存期与计划本身相关联。 在从缓存删除计划时，也将从该视图中删除对应行。

-- 返回数据库中最高总时间的5个查询模式

SELECT TOP (5)

MAX(query) AS sample\_query,

SUM(execution\_count) AS cnt,

SUM(total\_worker\_time) AS cpu,

SUM(total\_physical\_reads) AS reads,

SUM(total\_logical\_reads) AS logical\_reads,

SUM(total\_elapsed\_time) AS duration

FROM (SELECT

QS.\*,

SUBSTRING(ST.text, (QS.statement\_start\_offset/2) + 1,

((CASE statement\_end\_offset

WHEN -1 THEN DATALENGTH(ST.text)

ELSE QS.statement\_end\_offset END

- QS.statement\_start\_offset)/2) + 1

) AS query

FROM sys.dm\_exec\_query\_stats AS QS

CROSS APPLY sys.dm\_exec\_sql\_text(QS.sql\_handle) AS ST

CROSS APPLY sys.dm\_exec\_plan\_attributes(QS.plan\_handle) AS PA

WHERE PA.attribute = 'dbid'

AND PA.value = DB\_ID('Performance')) AS D

GROUP BY query\_hash

ORDER BY duration DESC;

优化索引和查询

现在已经知道了哪些查询模式需要进行优化，可以着手更具体的查询优化步骤了。这一步会设计到索引优化和查询代码优化。

SQL Server的数据库引擎顾问是一种工具。它可对输入的工作负荷进行分析，在此基础上为数据库优化提供建立。

索引是一种用于排序和搜索的结构。在查找数据库时，索引可以减少对I/O的消耗。

**页和区**

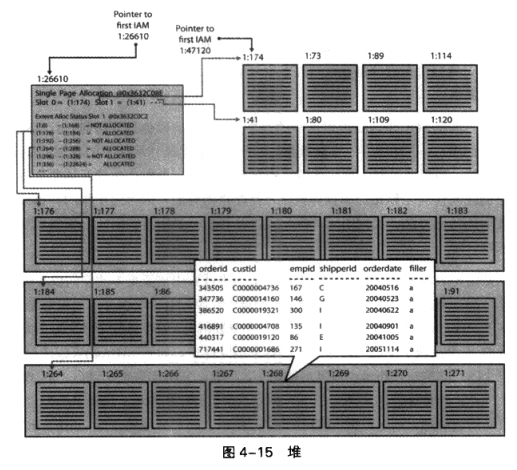
页是SQL Server存储数据的基本单元，大小为8KB。它可以包含表，索引，分配位图，可用空间信息等。页是SQL Server可以读写的最小I/O单元。即使只需访问一行， SQL Server也要把整个页加载到缓存，再从缓存中读取数据。对于基本数据操作所涉及的查询，其开销主要是I/O开销。

区是由8个物理上连续的页组成的单元。当表或索引需要更多的空间存储数据时，SQL Server会为对象分配一个完整的区。对于包含少量数据的对象，有一个例外：如果对象不足64KB，则当需要更多的空间时，SQL Server通常分配一个单独的页，而不是整个区。

**表的组织方式**

表有两种组织方式：**堆**或**B树**。从技术上来说，当在表上创建一个聚集索引时，表就组织成一个B树；否则就组织成一个堆。所以表的组织方式也称为HOBT(Heap Or B Tree)。无论如何组织，表都可以定义0个或多个非聚集索引，而非聚集索引又会组织成B树。

**堆(heap)**是不含聚集索引的表。它被称之为堆是由于它存储的数据不按照任何顺序组织，而是按分区对数据进行组织。在一个堆中，用于保存数据之间关系的唯一结构是一个索引分区映射(IAM,Index Allocation Map)的位图页。

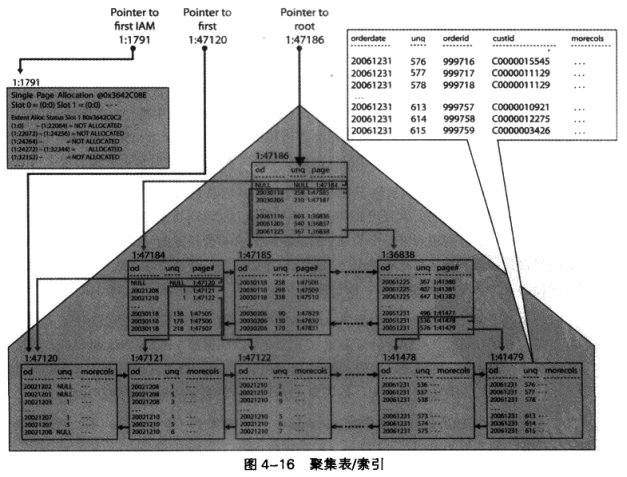


**聚集索引**

SQL Server中所有的索引都组织为**B树**结构，B树是平衡树的一种特例。**平衡树**的定义是“不存在叶子节点比其他叶子节点到根的距离要远的多的树”。它在叶子节点中维护整个表的所有数据。聚集索引不是数据的副本，而是数据本身。

当SQL Server需要对索引的页层执行无需扫描时，可以使用IAM页，这种扫描称为**分配顺序扫描(allocation order scan)**。而按照索引顺序完成的扫描称为**索引顺序扫描(index order scan)**。如果索引的碎片级别较高，则顺序扫描要慢得多。

当SQL Server需要导航到位于叶级的特定键时，它总是从根页开始，使用一种称为**索引查找(index seek)**的访问方法。这些读操作的I/O称为随机I/O，其读取次数和索引级别是一样多的。



**非聚集索引**

非聚集索引也结构化为一颗B树，而且在许多方面都和聚集索引类似。唯一的区别是非聚集索引的叶级行只包含索引键和特定的行定位符(row locator)。行定位符它是一个8字节的物理指针(RID)，它由数据库中文件号，文件中的目标页号，目标页的行号组成。因此SQL Server必须在查找操作之后，执行一个RID的查找操作，对于大量的数据而言，开销会非常高。

