SYBASE
技术服务园地
连载（68）
SYBASE TECHNOLOGY SERVICE FIELD

解决方案

开发人员升级至ASE 15.0的10大理由（九）

（接上期）

10 语义分区

本文的目的并非要详细介绍应该使用可种分区模式，而仅仅是如何更好地使用分区，等一因为它已经远远超出了意向的范围。在此将讨论为何要考虑使用分区，语义分区与轮循如何区别一与为何语义分区要优越于联合视图（前文曾提及）。

在ASE中有很多原因需要对表进行分区，包括：

（1）降低数据库维护时间（超出范围）。

（2）对批处理提高性能。

（3）改善并行查询效率。

（4）提升插入或查询性能。

（5）与ASE/CE横向集成（超出范围）。

第一点和最后一点原因超出了本文的范围。表15总结了不同类型分区在上述情况和其余考虑下的有效性：

表15 不同类型分区及有效性

分区类型	分区数量*	查询谓词	批量处理	并行查询	插入/查询性能	横向扩展	优势和劣势
轮循	数10个*	等于	否	否	仅插入(APL)	否	优势：高速并发插入。11.x/12.x形式。 劣势：查询性能最差、无本地索引、对DBA维护无益处。
列表	低于10个*	等于	可能	可能	否	可能	优势：精确值替换 劣势：基数小、等于谓词只适用于查询、分区数据量大时插入速度慢
范围	10~100*	范围或等于	是	是	是	是	优势：对日期、算术序列、有序数字等的查询最有效。 劣势：分区数量大时插入速度慢
哈希	10~1000*	等于	否	可能	可能(APL)	否	优势：大量分区、非序列化访问键(客户编号) 劣势：等于谓词只对查询有效

注：并无最大限制，但作者推荐优化考虑了分区检索次数和可管理性。

在讨论ASE 15.0.x中的分区为何对开发人员重要之前，以下是几个有关分区必须遵守的规则：

（1）了解分区的目标一然后选择支持该目标的分区类型，不丢失任何重要内容。注意交换条件。如果需要关注插入性能，但也需要不错的并行查询支持一轮循分区可能不是最佳方案。虽然它对插入的性能最好，但对查询速度的影响可能会有问题。

（2）令分区粒度与查询范围匹配。在查询中在“天”的分区上用“周”的范围并不好。太多分区=太多痛苦。不仅是创建分区会变成DBA的噩梦，而且分区数量将会影响查询或DML的性能。通常，分区越多，性能影响越大。

（3）所有语义（哈希、列表、范围）分区将会影响模式。索引至少要被声明为本地或全局的，如果分区键或全局索引的一部分不是主键，则主键可能必须是应用程序强制的。另外，为了达到最佳的并行查询性能，也需要对模式进行非规范化，以便确保每个表中的相同分区键。可能需要创建一些视图来自动包含遗留应用的分区键。

（4）尽量将分区键加入where子句。例如，如果通过时间范围进行分区，并且查询将要检索最后10次交互一添加日期范围— 诸如在过去的一年(或两年)内一以避免检查所有数据。

（5）若DBA是以维护数据归档为目标的情况下应避免全局索引。基本原因是因为本地索引可被快速截断而全局索引必须用普通方法移除一可能是数小时vs.数秒。

在了解了这些规则之后，我们就来看看为何开发人员应该考虑使用表分区除了DBA维护方面和与集群的横向扩展方面以外。

10.1 提高批处理的性能

考虑在大容量事务型系统中数据老化性质有关

的典型性批处理—或甚至是OLTP处理。本质上存在：

非活动的数据—典型的历史数据是已经完成了对每个字节处理的数据。保留了历史报表功能，且基本是只读的。

事务后—已经达到了最终OLTP和后OLTP状态的数据(例如，完成了交易，填完了订单等)，但还需要最后的统计计算、数据移动的块抽取等最终的批处理。

事务中—已达到最终OLTP状态的数据，但未完成的批处理可能仍要对数据进行修改。它包含了诸如交易结算、计费状态、支付接收状态等。

事务—仍有用户更改的数据或运行于单独业务对象(域块数据项相对)的自动化工作流程。

让我们通过一个假想的web订单业务来举例。当订单创建后，可能变为预订了存货的“未决”状态或只支持在后续的会话中恢复它(例如，保存了购物车)。最终，订单变为“已下”。在信用卡付款成功后，该订单变为“已接受”(可用email解释的方式“拒绝”订单—例如信用卡拒付)。一旦“已接受”订单，仓库人员开始处理订单，且每个订单项便为“已包装”或“缺货”。当所有项“已发送”，订单状态变为“完成”。

从处理的角度出发，手头有货的特定商品数量应该在“已包装”后立即进行更新。但是，来考虑一下缺货的批处理—它每天晚上运行，并基于过去的一周订单情况决定是否需要预定更多产品防止缺货(因为与“有库存”状态比较，大部分web用户都讨厌看到“缺货”—难道不是么？)。在货运到达某个仓库后会开始一个批处理，用来在较早的核最近的客户订单中修改缺货项。通常，因为线路条数的硬件局限或与信用卡公司签订的合约所限制的连接，我们对同时能处理的信用卡数有限制。因此，当订单“已接受”时，还不能立即发起验证信用卡的过程，因为它很可能造成系统困难或出错。结果就是，甚至是一些OLTP的过程都基于工作流队列，其中一些自动流程会不断检测特定的订单状态并对外部合作伙伴发送信息。当消息被发送或被告知，该流程即进行下一订单—而其他流程都完成相同的任务。考虑以下订单状态和订单项状态查询表的例子：见表16，表17。

当然，我们大获成功了—因为DBA准备通过表分区来减少维护时间—它对业务非常重要，因为它24×7的性质会令维护对性能吞吐量造成影响。

表16 订单状态

Status	Order Status
-1	过期/可归档
0	购物车
1	已下
2	已接受
3	拒绝
4	取消
5	部分完成
6	全部完成

表17 订单项状态

Status	Order Status
-1	过期/可归档
0	购物车
1	新订单
2	已包装
3	缺货
4	取消
5	返回
6	递送

DBA的提议类似于：

```
create table orders (
        order_num           bigint          not null,
        customer_id         bigint          not null,
        order_status        tinyint         not null,
        order_date          datetime        not null,
...
                primary key (order_num)
) lock datarows
partition by range (order _num) (
        pending     values <= (0), -- orders in shopping cart not yet
placed
        p10M        values <= (10000000),
        p20M        values <= (20000000),
        ...
        p100M       values <= (100000000)
)
go
create table order_items    (
        order_num           bigint          not null,
        item_num            smallint        not null,
        product_sku         varchar(30)     not null,
        quantity            int             not null,
        item_status         tinyint         not null,
        status_date         datetime        not null,
        ...
                primary key (order_num, item_num)
) lock datarows
partition by range (order_num) (
        pending     values <= (0), -- orders in shopping cart not yet
placed
        p10M        values <= (10000000),
        p20M        values <= (20000000),
...
        p100M       values <= (100000000)
)
go
```

DBA辩解道：

(1) 因为订单编号是有序列的—订单日期也相同—效果会与在日期上分区相同—而且可以保留使用主键强制的本地索引。

(2) 订单项将全部在一个分区中，然而在order_date和order_item上对orders分区将导致在检索议长订单时需要检索多个分区，如果同一订单中的两项递送时间不同的话。

（3）因为order_num在两个表中都有，则基于查询优化的分区消除连接会更有效(与在order表上根据order_date分区，在order_item上根据order_num分区)，因其将降低连接处理的IO（在下面的查询性能部分讨论）。

（4）最早的情况是周订单的查询，如果order_num在一周的中间被分区的话，查询需要访问两个分区。

（5）超过7年时间的订单(由于税务的原因而保留)能被轻松快速地在几秒钟内被截断，而非使用一次慢速删除几个订单的大规模（日志记录）的删除。

都是非常正确且有深刻见解的决定。但是，除了获得连接性能外（仅对影响大量定单号的报表时才会引起注意），应用程序整体的处理并不能受益于该分区模式。开发人员Joe Guru提出了以下另一种模式：

```
create table orders (
        order_num           bigint          not null,
        customer_id         bigint          not null,
                primary key         (order_num)
                --
) lock datarows     (or der_status,     status_date)    (
partition by range
        stale           values  <=  ( - 1,  'Dec  31  9999'),
        cart            values  <=  (0,  'Dec  31  9999'),
        placed          values  <=  (1,  'Dec  31  9999'),
        accepted        values  <=  (2,  'Dec  31  9999'),
        rejected        values  <=  (3,  'Dec  31  9999'),
        cancelled       values  <=  (4,  'Dec  31  9999'),
        partial         values  <=  (5,  'Dec  31  9999'),
        Q1_2002           values  <=  (6,'Mar  31  2002  11:59:59pm'),
        Q2_2002           values  <=  (6,'Jun  30  2002  11:59:59pm'),
        ...
        Q4_2008           values  <=  (6,'Dec  31  2008  11:59:59pm')
        Q1_2009           values  <=  (6,'Mar  31  2009  11:59:59pm'),
        Q2_2009           values  <=  (6,'Jun  30  2009  11:59:59pm'),
        order_date          datetime        not null,
        order_status        smallint        not null,
        status_date         datetime        not null,
)
go
create unique index orders _PK on orders (order_num)    -- global
index
go
create view new_orders as
        select * from orders
          where order _status=1
            and status_date < 'Dec 31 9999'
go
create view placed_orders as
        select * from orders
          where order_status=2
            and status_date < 'Dec 31 9999'
go
create view cust_orders as
        select * from orders
          where order _status=6
go
create table order_items (
        order_num           bigint          not null,
        item_num            smallint        not null,
        product_sku         varchar(30)     not null,
        quantity            int             not null,
        item_status         smallint        not null,
        status_date         datetime        not null,
        stale           values  <=  ( - 1,  'Dec  31  9999'),
        cart            values  <=  (0,  'Dec  31  9999'),
        new_order       values  <=  (1,  'Dec  31  9999'),
        picked          values  <=  (2,  'Dec  31  9999'),
        back_order      values  <=  (3,  'Dec  31  9999'),
        cancelled       values  <=  (4,  'Dec  31  9999'),
        returned        values  <=  (5,  'Dec  31  9999'),
        Q1_2002           values  <=  (6,'Mar  31  2002  11:59:59pm'),
        Q2_2002           values  <=  (6,'Jun  30  2002  11:59:59pm'),
        ...
        Q4_2008           values  <=  (6,'Dec  31  2008  11:59:59pm')
        Q1_2009           values  <=  (6,'Mar  31  2009  11:59:59pm'),
        Q2_2009           values  <=  (6,'Jun  30  2009  11:59:59pm'),
...             -- primary key(order_num, item_num
)
partition by range(item_status,status_date)    (
)
go
create unique index orderitems _PK on order_items (order_num,
item_num) -- global index go
create view new_orderitems as
        select * from orderitems
          where order _status=1
            and status_date < 'Dec 31 9999'
go
create view back_orders as
        select * from orders
          where order_status=3
            and status_date < 'Dec 31 9999'
go

create view cust _orderitems as
        select * from orders
          where order _status=6
go
```

他对该提议的理由也同样有趣：

（1）新订单将进入最小的分区中—使用最小的索引大小等 —因此能获得最快的插入速度，因为最小化了遍历所有索引来插入索引键的逻辑IO。

（2）许多自动化过程现在不得不在查询新订单等常见查询中使用队列表，因此导致表扫描（如set rowcount 1; select * from orders where order _status=1)。结果，新的订单和一些流程导致对这些队列表的额外插入—随即导致工作流连接回工作表—或者总在改变状态前读取行。通过对这些工作流队列状态进行单独分区，可避免额外插入，并且该过程将不用进行分区扫描—就像使用了队列表一样。改变订单状态会导致延迟更新（当然），因为行会在分区间移动，但是由于数据行锁定，该删除将是逻辑删除（行被标注）。如果工作流持续不断，则分区将非常小，而无需重新组织来清理空间—或在数秒内就能完成。

（3）季度销售报表可在单一分区上使用范围扫描完成。

（4）DBA维护也最小化了，因为未递送的订单会在非常小的分区中，而且相当短暂，以致于运行更新语句大多没有意义。已递送的订单只会对最近的一两个分区进行维护—正如order_num分区模式的一样。

（5）在主键上的全局索引意味着归档非活动数据将是快速截断，删除过程无需干预，它的性能对于接收新订单（信用卡支付）及包装/递送来说是次要的。

Joe Guru是明智的。如果注意的话，分区语句先指定了order_status，并对每项进行了排序。这是因为Joe明白他想针对一片数据进行范围分区—而因为只能使用一种分区类型和分区键的计算序列，他选择了order_status这样重复性高的数据值，而不是之前在该列上进行的列表分区。结果就是对完成的订单，总是会落到该日期下—而对于新订单，它将总是能到伪列表分区。最终的效果就是他创建了一些分区，有的行为像列表，有的行为像范围分区。

10.2 改善并行查询效率

15.0中针对DSS做的改善是更好地实现了并行查询。它构建于感知分区的优化器上，支持动态的分区消除。结果是ASE 15.0能在两个类似的分区表中执行“矢量”或“导向”连接。考虑上述DBA的提议，在“orders”和“order_item”表上的order_num上分了10个区。使用ASE 12.5的轮循分区，可以抑制并行查询的开销。正如很多人都知道的，当在12.5中激活并行查询时，连接的内部表常常使用（n*m）线程。这是因为在外部表中扫描每个分区的线程需要为每个内部表分区发起一个线程因为不能假定数据位置。结果如图21。

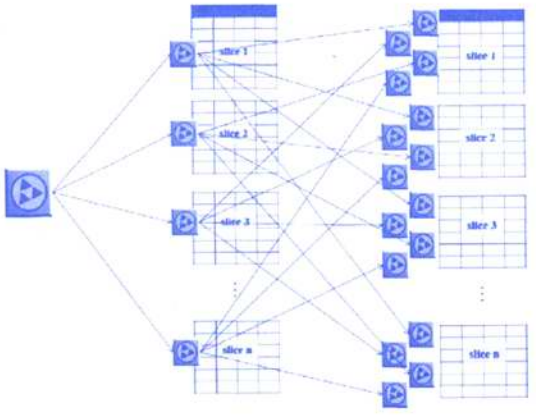

图21 ASE 12.5分片平行查询连接(老分区)

即使是小数量的分区—例如每个表4—都会在一个查询中产生16个工作线程（4 × 4=16—加上父线程）。少量的并发连接和ASE服务器将是CPU限制的，因为只有非常少量的用户会使用每个可用的CPU时间分片。例如，10个并发查询将需要160个工作线程，并行查询 — 尤其是那些包含连接的—常常不被DBA允许。

ASE 15.0对其进行了改变。通过使用语义分区—范围、哈希或列表—优化器知道哪个分区包含了哪些值，并可通过简单地使用分区键执行矢量连接，将连接“导向”包含了相同分区键值的分区。结果如图22。

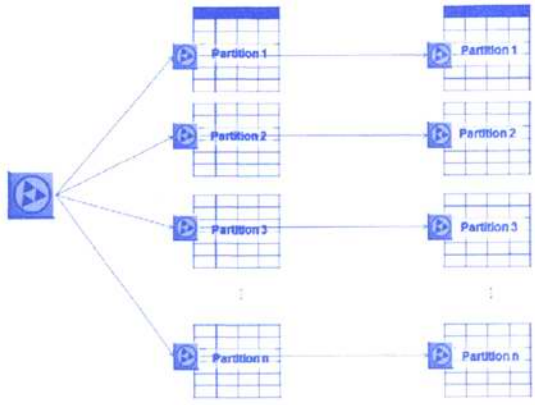

图22 ASE 15.0矢量连接并行查询

结果不仅大大地减少了CPU的需求—而且提升了查询性能，因为不同的工作线程不再竞争资源了！在此之上，在通常的分区消除情况下，优化器可能决定对分区1和2的访问甚至是不需要的—这样使用的整体资源将减少。在范围分区的情况下如果查询仅仅访问一部分范围的话，这将很容易发生。

10.3 插入和查询性能

在第一个例子中，Joe Guru对分区做出了建议，其中间接指出了一些分区将会很小的事实—它将有助于大部分活动的分区性能。它的常见理由是基于索引树的高度和性能聚合推进。分区对检索几行的单点查询或单独插入帮助不大。但是，它有助于整体性能。

文/赛贝斯软件（中国）有限公司

（未完待续）