解决方案

SYBASE
技术服务园地
连载（63）
SYBASE TECHNOLOGY SERVICE FIELD

开发人员升级至ASE 15.0的10大理由（四）

（接上期）

5 SQL UDF与替代触发器（Instead of Triggers）

5.1 SQL UDF

5.1.1 UDF简介

每个开发人员都会为不断编写相同的SQL表达式来实现业务计算而感到沮丧，当由于税率代码或其它考虑因素而产生公式变化时，定位并修改数十个有关的表达式简直就是一场梦魇。

通过将SQLJ的类方法暴露为SQLJ UDF，ASE 12.5迈出了解决该问题的第1步。虽然它支持开发人员将Java语言函数暴露为TSQL函数，但它有一些缺陷：首先，源函数必须用java编码（也许是开发人员并不熟悉的语言）。如果函数需要访问数据的话，问题就尤其严重了，因为函数需要调用JDBC方法；其次，随着对函数的每次调用，ASE必须进行上下文切换来激活嵌入的JRE来执行函数的Java字节码；再次，Java选项是单独的许可证选项；最后，嵌入的JRE是1.2.2，很快就过时了，并且功能和性能都有局限。

尽管存在局限，SQLJ UDF仍然给ASE新增了吸引人的功能。例如，TSQL的普通功能可轻松通过本地Java方法调用扩展成ASE没有的同等功能，如java.lang.String支持的常见字符串操作函数远超出了ASE的局限（字符串连接等）。

ASE 15.0.2新增了用标准TSQL命令创建标量用户自定义函数的能力。关键是标量，只能返回一个值。基本的语法为：

```
create function [ owner _name. ] function _name
( [ { @parameter _name [ as] parameter _datatype [ = default ] } [ ,... n
] ] ) returns return _datatype
[ with recompile] ]
as
[begin]
function _body
return scalar _expression
[end]
```

例如：

```
if exists (select 1 from sysobjects where name='elapsed_hms'
and type='SF' and uid=user_id())
drop function elapsed _hms
go
print "...creating UDF function 'elapsed _hms'"
go
create function elapsed _hms (@date1 datetime, @date2 datetime)
returns char(12) as begin
return convert(char(12) ,dateadd(ms,datediff (ms, @date1, @date2),
'Jan 1 1900') ,20)
end
go
```

该小巧的实用工具函数在基准测试时相当有用，它将不同测试场景的输出变为更人性化的可读格式（03：48：04），而非13684s。

5.1.2 UDF特点

SQL UDF有3个特点，与性能和其它实现因素相关，例如结果的可预测性。

（1）相关与无关UDF

根据查询的上下文，一个SQL UDF可被认为是相关或者无关的。相关UDF是其一个或多个参数参照到了查询中使用的表列；无关UDF是所有参数仅参照到变量(@vars和@@global)、系统函数或静态表达式。

请注意，是使用的查询决定了函数的性质而非函数本身。一个单独的UDF在不同的调用中可能是相关的与无关的。例如：

```
-- uncorrelated example for elapsed _hms: declare @oldest _sample datetime
select @oldest _sample=min (culture_date) from lab_tests
where test _status!=' reviewed'
select sample_age=elapsed_hms(@oldest_sample,getdate())
-- correlated example for elapsed _hms
select sample_age=elapsed_hms(culture_date,getdate()) from lab_tests
where test _status!=' reviewed'
```

以上两段的区别在于前一个仅使用了本地变量和系统函数作为参数，而后一个却参照了lab_tests表的culture_date列。

本特性之所以重要的原因是它对性能的影响。与getdate()等系统函数极为类似，无关SQL UDF每个语句仅计算一次，且其值在语句真正执行前已被规范化至查询表达式中。相关SQL UDF在查询执行的每行都要被计算。

（2）静态表达式与数据访问UDF

静态表达式用户自定义函数并不包括引用任何表数据的查询，而数据访问函数则包含了引用自数据库表中的数据查询。这里主要的考虑因素依然是性能。对于静态表达式UDF(例如上述的elapsed_hms函数)，UDF的执行速度与UDF中包含的语句数量直接成正比。对于数据访问UDF来说，嵌入至函数的查询则要服从与其它普通查询一样会遇到的问题：阻止、IO延迟、查询优化。结果，数据访问函数的执行速度很大程度取决于函数中查询的执行速度，这就常常超出了UDF源文件中仅对语句数量的考虑。幸好，与存储过程类似，SQL UDF在首次执行时就被预编译并优化，除非指定了“with recompile”选项(极度不推荐)。在后续的讨论中，这将相当重要，如果不注意，UDF可能会造成极大的性能问题。

（3）确定性的与非确定性的UDF

目前，对确定性的函数有两个稍有区别的定义：相同输入相同返回值；相同输入、相同数据库状态与相同的当前上下文环境，函数返回相同值。

非确定性的函数即是那些结果不能用上述两点中的任一点预测的函数。两个定义之间的不同显得晦涩难懂，请考虑以下内容：“相同的数据库状态”，暗指依赖于数据的函数如果在数据库内容是相同的情况下(相同行、相同列值)返回相同的值；“相同的上下文环境”，暗指在给定的相同执行上下文环境中函数返回相同值，例如同一用户、相同隔离级别、相同嵌套级别等。

两个定义之所以不同的原因是很重要的。如果计划在基于函数的索引中使用SQL UDF，则它必须遵从前一个定义，因为如果用户执行的查询或数据内容改变了计算表达式时，索引就不能生效了。反之，行级访问规则通常遵照后一个定义，因为其结果确实依赖于用户执行的查询或许可表的内容等。

5.1.3 UDF用例

SQL UDF有几个常见的用例，分别是实用工具函数、业务函数、访问规则函数和结果集状态函数。

（1）实用工具函数。通常是用来展现数据格式的函数，为改善开发过程中的易用性而创建。

（2）业务函数。通常是静态表达式函数，执行复杂的业务计算以便简化维护。它们也通常是基于函数的索引候选。

（3）访问规则函数。用来执行数据访问安全性检查的函数，通常用来实施细粒度访问控制（FGAC，也称为行级访问控制—RLAC）。它们最有可能是数据访问函数，用在政府应用程序中，访问规则函数可用来在用户访问每行数据时比较证书和数据敏感性标签。

（4）结果集状态函数。用在结果集中的函数，可提供优先依赖于结果集的数据。

最后一个用例很有趣。开发人员常常喜欢结果集中的行是顺序编号或是上卷聚合的。虽然它可以也可能必须通过应用程序逻辑实现，一些简单的脚本语言并不能支持对它的轻松实现。基于以下事实，诸如此类的函数可以实现：SELECT子句中的SQL UDF在结果集中的每一行都会被执行。虽然在SQL UDF中不支持DML操作，但ACF函数支持跨上下文环境的数据值持久化。考虑以下实现：

```
create function row_num (@data_value varchar(30)) returns int as begin
declare @ret_code1 int, @ret_code2 int, @rownum int, @cached_idle bigint,
@cur_idle bigint
-- try to fetch the last row_num and @@idle value from cache.... select
@rownum=convert (int,get_appcontext ('row_num','cur_row')),
@cached_idle=convert (bigint,get_appcontext( 'row_num', 'idle')),
@cur_idle=@@idle
-- if the current @@idle value is not the same as the cached on, we
-- were "idle" at some point, which means this is a new query and
-- we need to reset the cache and start over.
if (@cached_idle != @cur_idle) --or (@rownum is null)
select @rownum=1
else
select @rownum=@rownum+1
-- we can't overwrite an ACF - you have to rm it and then re-set it.... select @ret
_code1=rm_appcontext ('row_num','cur_row'),
@ret_code2=rm_appcontext('row_num', 'idle')
-- now set the cache to the new row_num and @@idle values
select @ret_code1=set_appcontext('row_num','cur_row' ,convert(varchar(10)
,@rownum)), @ret_code2=set_appcontext ('row_num','idle', convert (varchar
(20), @cur_idle))
-- return @rownum return @rownum
end
```

简单的改变将其变成了上卷求和的函数：

```
create function rolling_sum (@cur_sum bigint) returns bigint as begin
declare @ret_code1 int, @ret_code2 int, @new_sum bigint,
@cached_idle bigint, @cur_idle bigint
-- try to fetch the last row_num and @@idle value from cache.... select
@new_sum=convert(int,get_appcontext('rolling_sum','cur_sum')),
@cached_idle=convert(bigint,get_appcontext('rolling_sum', 'idle')),
@cur_idle=@@idle
-- if the current @@idle value is not the same as the cached on, we
-- were "idle" at some point, which means this is a new query and
-- we need to reset the cache and start over.
if (@cached_idle != @cur_idle) --or (@new_sum is null)
select @new_sum=@cur_sum
else
select @new_sum=@new_sum+@cur_sum
-- we can't overwrite an ACF - you have to rm it and then re-set it.... select @ret
_code1=rm_appcontext ('rolling_sum',' cur_sum'), @ret_code2=rm_appcon-
text('rolling_sum', 'idle')
-- now set the cache to the new row_num and @@idle values
```

```
select @ret_code1=set_appcontext('rolling_sum','cur_sum' ,convert(varchar
(20) ,@new _sum)), @ret_code2=set_appcontext('rolling_sum','idle' ,convert
(varchar(20) ,@cur _idle))
-- return @new _sum return @new _sum end
```

通过ACF来“缓存”值的能力对FGAC访问规则尤其重要。对于使用FGAC和ACF函数的参考资料在ASE系统管理指南（卷1）中的第17章：管理用户权限中的“行级访问控制”中讨论。因为可能大量执行行级访问函数，在判定权限规则并不简单的情况下，基于数据访问的UDF可能造成性能伤害。少量的值（<100或之类的）可使用ACF函数缓存，该缓存执行简单检查替代否则可能会需要发起的一系列查询。该函数的模板类似于：

```
create function check_permissions (@data_keyvalue varchar(30)) returns int
as begin
...
-- check to see if cache initialized - assuming this is connection from
-- a connection pool, specifically, we check to see if the cache is for
-- the current user
select @cached _user=get _appcontext( 'chk_sales_perm','cached_user') if
@cached_user!=suser_name()
begin
-- clear the cache and initialize to current login
select @acf _ret=rm_appcontext('chk_sales_perm', '*')
select @acf _ret=set_appcontext( 'chk_sales_perm','cached_user' ,suser _name
())
-- series of queries to set permissions here. . . each query that
-- sets a permission would have syntax similar to:
select @acf _ret=set_appcontext('chk_sales_perm' ,<colname>,<colname>)
from...
end
-- check to see if permission in cache
select @acf _value=get_appcontext( 'chk _sales_perm' , @data_keyvaiue)--
Tricky !!!
if @acf_value=@data_keyvalue return 1
else return 0
end
```

注意，最后一次ACF函数调用中把数据值作为键来使用它是可能都不会考虑到的方面，因为很多人都仅考虑使用静态的文字参数来调用ACF。

5.1.4　UDF实施考虑

尽管如此，这给UDF带来了一个很关键的考虑因素--对性能的影响。UDF在以下情况调用：

（1）无关UDF。出现一次就执行一次。对性能的影响是最小的，仅与函数语句的执行速度相关。

（2）SELECT中的相关UDF。结果集中的每行都执行一次。如果结果集中的行数很多或函数的语句有明显的额外开销的话，值得考虑对性能的影响。例如，一个函数的执行时间是10 ms，则对返回1 000行数据的语句来说会增加10 s的执行时间。

（3）用作谓词的相关UDF。表的每个物化行在计算时，其列将引用到函数，这将十分影响通常会计算大量行的查询，例如聚合查询。另外，如果谓词被应用到连接子句的外表以限制结果集，且谓词必须在连接前计算的话，它的影响将更为显著。类似地，如果是主细目关系（master-detail）的连接，其增加了物化行且谓词在连接之后才进行运算，则也可对性能产生巨大影响。

对于最后的这点举个例子，一个在where子句中的UDF将计算82 500行，总体执行时间从32 ms增加至33 ms，仅仅增加了1 ms。但是，当对135万行数据进行扫描时，执行时间从5s激增至差不多9min，远远超出按照推断预期的16ms。结果就得千方百计降低SQL UDF的执行时间，还有由于结果集大小或where子句计算而带来的执行次数。由于最终用于对响应时间的需求，每毫秒都应认真考虑。所以，需要对UDF认真进行性能优化。

5.2　替代触发器

替代触发器实际上是视图上的触发器，它支持开发人员根据对视图的插入、更新或删除动作来进行不同的编码。注意，只有依赖于单一表的视图才可进行插入、更新和删除，并且“create view … with check option”强制用户的任何DML操作在该操作后依然对用户可见。问题是在源于多个表的视图时，包含了连接子句或“group by”矢量聚合。在这些情况下，DML操作通常都会失败。它的功能与表上的触发器差不多，它提供DML操作后的后处理自定义逻辑。普通的触发器在表上操作，而替代触发器则严格在视图上操作。开发人员现在能在原先不支持的有连接、聚合或其它情况的视图上，在触发器中使用自定义的逻辑来决定对基表的适当操作或执行用来替代原始DML请求的动作。

5.2.1　对象-关系抽取

以前的版本在应用程序与数据库模式之间的对象—关系映射中是一个大问题。在开发面向对象的应用程序时，常用的技术是从较高级别的对象处继承父类的属性来创建子类，也可包含子类特定的属性。例如，医院的病人可能是住院病人或门诊病人。对于病人来说，有一些通用的属性，如全名、地址、保险信息、血型、性别。但是，门诊病人和住院病人有一些不同的属性，例如病房号、住院医生、住院原因等。与其使用一张很多null域的表，还不如用常用的方法，即一张表存储病人信息，然后不同的表分别存储住院/门诊病人的数据。

文/赛贝斯软件（中国）有限公司

（未完待续）