SYBASE
技术服务园地
连载（62）
SYBASE TECHNOLOGY SERVICE FIELD

解决方案

开发人员升级至ASE 15.0的10大理由（三）

（接上期）

3 基于函数的索引

ASE 15.0中基于函数的索引简直是对性能的免费大提升，无需更改应用程序立即可用。已经证明以下类型的搜索谓词会引起性能问题，因为查询处理引擎不能将其视为“SARG”：

```
select ... where upper(last_name) = upper('McDonald')
select ... where datediff(dd,due_date,rcv_date) > 90
select ... where substring(ssn,8,4) = '1234'
```

事实上，在性能调整和优化的部分讲述了“在谓词操作符左边的公式和表达式将不被视为SARG”，这将不再正确。在ASE 15.0中，可为每个索引键使用表达式创建索引。例如，考虑：

```
CREATE INDEX total_idx ON order (unit*price*1.1,type) go
    SELECT unit*price*1.1 FROM order
      WHERE unit*price*1.1 > 200 AND
      type = 'Produce'
    go
```

该查询是一个完全覆盖查询，SELECT子句的投影列和WHERE子句的条件列都被同一索引“total_idx”覆盖了，结果甚至都不用读取基表。但一个重要的注意事项是，该函数表达式需要确定有用处。考虑以下表达式：

```
CREATE INDEX over_due_idx ON billing (datediff(dd,due_date,
getdate())
```

按照语法来说是没错的，但是如果考虑在不同的日期上运行查询将会发生什么。和其它索引一样，索引键被存储在索引树上。对于表达式键，当索引创建计算表达式时，表达式结果被存储在索引键中。对于上例的表达式，over_due_idx的结果是基于索引创建时或者索引行创建时(例如新行被插入表中)的datediff()表达式计算的，会产生类似结果的其它系统函数包括suser_name()、db_name()等。

但是，除了该方面以外，可考虑到系统立即获得的优势，在此之前不能使用表达式来降低被检索的行数，尤其是对XML文档。例如，使用早先的XML片断，考虑以下查询：

```
SELECT ... FROM book_orders_xml
    WHERE xmlextract ( '/BookOrder/Store/StoreID/text ()',
        order_xml returns char(4)) = '5023'
    AND xmlextract ( '/BookOrder/OrderDate/text ()',
        order_xml returns datetime) > 'Oct 1, 2008'
```

没有基于函数的索引帮助，以上查询即被翻译为表扫描，扫描的每行都会包含动态的XML解析和查询。但请考虑以下影响：

```
Create index store_orders_idx on book_orders_xml (
    xmlextract( '/BookOrder/Store/StoreID/text()' ,order_xml returns
char(4)), xmlextract( '/BookOrder/OrderDate/text()' , order_xml returns
datetime)
    )
```

表8 使用基于函数的索引的优势

522行的测试场景	结果用时（ms）
未使用索引和谓词的简单表扫描（select *）	60 ms
上例查询（store_id与order_date谓词），未使用索引	12, 423 ms
同一查询使用了上述基于函数的索引	30 ms

这是仅针对500行数据获得的400多倍提升！当结合了计算列自动将键数据项解析至关系型的列，同时在文档内提供了针对常用谓词的搜索功能，这尤其有趣。

4 计算列

我们都学过最重要的数据库设计规则是“键、都是键、没有别的还是键，Codd救救我吧”。然而事与愿违，所有人都不得不在模式中添加聚合值，并存储表达式结果以获取更爽快的性能或减轻应用程序开发人员维护复杂表达式的痛苦。以前的问题是，这些表达式通常被嵌入数据库触发器或不够直接的逻辑中，要么是造成日志扫描用以构建inserted/deleted表，或者在原子性的插入中为了调用触发器而带来的额外开销等。

幸运的是，ASE 15.0通过计算列新特性找到了解决途径。该特性支持一列被定义为一个表达式，要么物化（与其它数据属性存储在一起），要么虚拟化（非物化：执行时动态计算）。考虑以下少许改变过的pubs2模式：

```
create table titles _2 (
        title _id    tid              not null,
        title        varchar(80)      not null,
        type         char(12)         not null,
        pub_id       char(4)          null,
        price        money            null,
        advance      money            null,
        total_sales  int              null,
        notes        varchar(200)     null,
        pubdate      datetime         not null,
        contract     bit              not null,
        revenue      as price* total_sales   materialized  null,
        review           as (case
                  when total_sales > 10000 then 'Best Seller' when
                      total_sales between 1000 and 10000 then
                                  'Money Maker'
              when total_sales < 1000 then 'Flop' when
              total_sales < 100 then 'Fire Wood' else null
          end) not materialized
)
```

“revenue”列被物化了，而“review”列因未被物化而需在执行时动态计算。正如所展示的，表达式可以是简单的公式，也可以是复杂的case语句。它还可包含SQL UDF（后续讨论）或ASE自带的函数。

该特性可用于多个原因。还记得我们在简单审计问题中用来追踪最后修改表的用户的例子？广泛使用的“last_username”和“last_datetime”列看起来在每个表中都出现了？对该问题的一部分挑战是不能使用缺省值。虽然列的缺省值在插入时能生效，但如果提供值的话则很容易蒙骗过关，它们并不在更新时被触发。结果就是这些列常常都通过触发器实现，这不仅增加了应用程序的复杂度，并且影响性能，因为设置这些列需要与inserted/deleted表进行连接。它也加倍了数据库复制的负载，因为原始插入和触发器更新都将被复制。

ASE 15.0中的计算列代表了更为优雅的解决方案。还记得前面用来展现哈希函数的trade_hist例子么？它部分展现了表进行自审计的方法。更完整的例子类似于：

```
create table trade_hist (
        transaction_id    unsigned bigint     not null,
        customer_id       unsigned bigint     not null,
        trade_date        datetime            not null,
        symbol            char(6)             not null,
        ...
        changed_flag      timestamp           not null,
        ...
        inserted_by       as suser_name() materialized not null,
        inserted_on       as getdate() materialized not null,
        updated_by        as (case when changed_flag is not null
                              then suser_name() else null end)
                                  materialized      null,
        updated_on        as (case when changed_flag is not null
                              then getdate() else null end)
                                  materialized      null
)
go
```

本例展现了一些非常有趣的内容。因为物化计算列在任何相关列改变时会重新计算，inserted_by和inserted_on列的值将永远不会改变，因为它们与表中的任何列都互不相关。另一方面，时间戳（timestamp）列会在每次插入和更新时改变，仅需简单地与该列参照，则在每个DML操作引起时间戳列改变时，updated_by和updated_on列的表达式即会重新计算。因此，updated_by和updated_on列将会在每次插入或更新时被重新计算。这样，自审计表的简单审计需求则无需求助于触发器或存储过程代码。

另一例子是基于前面用来讨论基于函数索引的XML表：

```
create table book_orders (
  order_xml       text       null,
    order_num  as xmlextract ( '/BookOrder/OrderNum/text () ',
order_xml returns varchar (30))
                materialized   not null,
    store_id   as xmlextract('/BookOrder/Store/StoreID/text() ',
order_xml returns char(4))
                materialized   not null,
    order_date   as xmlextract('/BookOrder/OrderDate/text()',
order_xml returns datetime)
                materialized   not null,
                primary key (order_num,store_id)
)
go
create index store_order_idx on book_orders (store_id, order_date) go
```

注意，定义的计算列在相关列之后。同样，现在索引创建在了真实值上，而非派生表达式，结果就是更容易实现。结果是自解析的表，在每个插入表的XML消息时其键值被自动解析。与前面的查询结果相比较：

表9　ASE15计算列的应用效果

522行的测试场景	结果用时（ms）
未使用索引和谓词的简单表扫描（select *）	60 ms
上例查询（store_id 与 order_date 谓词），未使用索引	12, 423 ms
同一查询使用了上述基于函数的索引	30 ms
同一查询使用了带有索引的计算列的表	<10 ms

更为重要的是，不支持XML处理的遗留应用可基于普通的SQL表达式检索数据，而无需更改为复杂的XPATH表达式。

文/赛贝斯软件（中国）有限公司

（未完待续）