# THOR问题思考纪要

## 数据库中使用一些轻量级压缩

在数据库中很少使用重量级压缩，因为目前主要的瓶颈是在CPU，而不是磁盘，所以使用轻量级压缩要更合适一些，比如：数值压缩（Numeric Compression）, 字符串压缩（String Compression），字典压缩（Dictionary-based Compression）

数值压缩：整数类型存储的数据通常远远小于整数边界，数值压缩就是主要是信息熵，压缩数据的宽度，这种简单的压缩方式效率很高，而且解压的负担非常小。通常可以利用一段数据中的最大最小值来计算一个偏移，然后再对数据进行数值压缩

## 表达式运算溢出的问题

表达式运算时涉及到类型升级的问题，如果进行升级，则会加大计算的成本，但如果不升级数据类型，特别是乘法运算时，很容易出现数据溢出问题。

逻辑运算不会存在数据类型升级问题；

对于整型：

加法，减法，乘法：升级一个数据类型，最大支持64位

除法，不需要升级数据类型

对于浮点型：

计算时全部使用DOUBLE类型进行运算

数值类型提升 int8 > int16> int32 > int64 >? double

float > double

int64 和 float，则转换成double

## 常量在向量表达式运算时是否需要优化？

常量一般时和变量进行运算的，否则就可以再编译期进行优化，因此不需要单独针对常量向量做优化

## 是否需要针对NULLMAP做优化？

第一阶段不考虑NULL值，第二阶段考虑

## 表达式计算是否需要针对各种数据类型优化？

实际测试的结果表明，整形，浮点型的算术运算，不同精度类型的速度实际上差别不大（甚至INT和LONG要比SHORT，char的速度要快一些）。

差别主要是在存储空间上，主要是对CACHE更友好，但因为目前的CACHE是足够容纳最大数据类型的SEGMENT的，因此针对不同的数据类型优化算术运算的意义不大。

在SIMD的情况下会好些，但是需要做很多而外的优化，实际优化代码的工作量较大，价值不高。

因此算术运算，整形建议直接提升至BIGINT类型？，浮点直接升级为DOUBLE类型

逻辑运算不涉及类型升级；

## 关于SIMD指令

SIMD指令，目前最新的AUX2支持到256位，未来的AVX 512估计在2015年能够普及，目前主要的使用方式是使用 汇编，intrinsic，以及编译器自动优化等方式，目前用的较多的是使用intrinsic，intrinsic和汇编是一一对应的，但有些功能可能不完整， intrinsic指令使用的要求较高，太过于涉及底层细节，可移植性较差，编译器自动优化的方式，之前有论文说效果较差，后面需要实测一下。

和SIMD类似的技术，还有SIMT，SMT，这两个技术目前都是在GPU上的，目前CPU还不支持，后续CPU+GPU混合计算可能是个趋势，另外，现在内存数据库（内存计算）的技术发展实际上很多事可以参考图形处理的一些相关经验。

如何设计最后一段的数据扫描

通常扫描都是直接整段进行扫描的，这样是不需要任何参数，但是最后一段，因为不会填满一段，所以处理方式就会比较特殊；

有两种方式，一种就是生成两个表达式树，一个是针对SEGMENT满的情况，一个是针对没满的情况，运行时判断是否已经到达最后一段，如果没有则使用前面加速的表达式计算，最后一段则使用动态的计算方式；

另外一种就是直接使用动态方式，先判断段内的大小是否需要用SIMD加速

结论，先使用简单的方式，性能影响应该是很小的；

## 关于表达式的缓冲

一种方式就是在表达式节点中，设置固定的缓冲，比如成员变量，这样用起来比较方便，但问题是空间消耗大，如果是复杂SQL的话，表达式本身就会占用很大的空间；

另外一种就是计算的空间由外部给，表达式本身没有固定的缓冲，这样一个SQL，一个线程仅仅使用一个BUFFER就可以搞定；

但带来的问题是，对于常量来说，会存在反复的取值赋值运算，这个实际上是浪费的；

## 关于执行节点的缓冲

执行节点可能会在不同的线程中执行，因此执行节点的结果的内存是由外部提供的，以避免多余的拷贝

执行节点的缓冲和表达式计算的缓冲不太一样，执行节点设计上考虑了多线程执行的场景，而表达式计算则目前不考虑多线程执行，而考虑多线程会增加缓冲的复杂度，所以两者的缓冲设计上差别较大。

## 关于执行节点的输出

执行节点有两种输出方式：

一个是除了最后一个段，每次固定输出1024个，设计要复杂一下，但对SIMD指令会友好一些，但涉及需要移动缓冲，这个可能会影响部分性能。

另外一种是输出一个范围[0,1024]，每次数据的都不固定

我们来分析查询涉及的主要操作：

* WHERE，会减少数据集合
* ON ，会减少数据集合
* HAVING，会减少数据集合
* DISTINCE，会减少数据集合
* GROUP+AGGR，会产生新行，全部输出
* AGGR，会产生一个新行
* SORT，改变顺序，全部输出；
* JOIN 会重新排列组合，改变输出，可能多也可能少；

结论：虽然输出范围【0，1024】的方案比较简单，但输出的个数比较少，性能会有一些影响，而且实际上主要是WHERE和类似操作涉及到相关的问题，其他的不需要特殊处理；因此采用固定1024的方案；

## GROUP BY的中间结果

进行分组聚合的结果最好是行模式，这样性能是最好的，但输出结果如果行方式的，会和其他的表结构模型上有区别，会增加复杂度。

## 关于最大最小值HINT

在使用表达式做范围过滤时，目前不考虑B树索引的，因此，如果全表扫描，性能会比较差，

一种折中的方法就是在数据加载时，设置段的最大最小值的HINT，这样一方面可以比较快捷的实现范围查找，另外一方面可以比较方便数据压缩。

最大最小值的HINT可以统一存放，可以用一个小的辅助表来存放，这样在使用WHERE做表达式过滤时可以先扫描这个表，用最大最小值过滤一遍；只要最大最小中的任一条件满足，及需要扫描实际的数据段，否则可以跳过该段；

## 关于顺序取值和随机取值

表字段的顺序取值和随机取值的区别是比较大的，

如果是顺序取值，特别是正好是一段未压缩的数据时，直接取值就可以，如果是随机取值，则必须提供一个缓冲区，需要先将零散的数据拷贝到缓冲区才能进行后续的操作；

顺序取值的话可以做一些优化，通常随机取值时需要赋值所有的ROWID，而顺序取，则在第一个位置设置一个特殊值，当遇到这个特殊值，就表示是顺序取值；

## 关于内存管理

内存管理，目前有几个诉求:

* 用来管理列式的内存表的数据；

对于列存表的内存管理，内存的利用率是首要考虑的要素，由于列存表的数据都是以段的方式存储的，所以列存表的内存大小都是以段的整数倍，以1024为段的大小为例，则，内存分配的大小是以1024\*n，n取值范围[1,16]，16字节的NUMERIC, 另外考虑到数据压缩的情况，则最小的单位为1024 bits(128 bytes)。

对于定长数据，最长的就是16字节的NUMERIC，而最小的是BOOL类型，1BITS，

而对于变长数据类型，则采用定长指针+BUFFER的方式存储，目前只考虑VARCHAR类型，这个属于自行管理的一种类型，只需要考虑一次获取的内存块能够较大即可。

* 用来管理运行过程中的过程数据；

对于运行过程中的数据，要求申请释放的性能能够尽可能的最高，释放可以采用region-base方式，即多次申请一次释放；

在聚合，排序等场景下，需要考虑行存表，存储模型可以参考变长字符串的存储模型；

除了以上的内存大小的需求，内存管理还必须考虑以下几点：

* 内存边界对齐
* 能针对NUMA优化
* 动态扩展

目前比较常见的内存池算法，可以参考linux内核的内存管理算法，目前主要有buddy, SLAB(slab, slub, slob)；算法就不在这里一一描述了：

buddy算法比较适合2的整数幂的内存需求，如果是非2整数幂，则存在一定的内存碎片；另外效率上，因为存在递归，所以申请和释放速度上会慢一些；

SLAB算法，对于小内存数据的效率高，当出现大量动态对象时，申请释放效率高；是现在Linux内核的主要算法（现在已经升级为SLUB），但是碎片较多；

综合考虑，目前算法上，buddy比较适合列存的用法。

设计buddy内存的要点：

1. 需要一个bitmap来描述最小页面的占用情况，用于当页面释放时，快速合并页面
2. 每个层级的空闲内存都组织一个双向链表，用于快速从空闲内存中申请空间；双向链表的目的在于快速合并页面；
3. BITMAP可以动态管理，避免初始化时就生成最小页面的BITMAP

## 采用延时内存整理，还是立即释放

延时的话，只需要在整理时创建辅助内存，用于整理空间，因此延时的方法占用内存小，但是性能抖动更明显；

立即释放的话，则需要常驻的辅助空间，来表示每个PAGE的状态，但问题是如果单个页面反复申请释放的话，则会出现频繁的分页合并动作，性能较差；

综合两者，采用立即释放方法，但可以缓存一部分队列，即每个空闲链表的大小超过阈值时就立即释放，否则只是归还到空闲列表；