# 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。

目前已经明确采用行式的结果集，但问题是结果集合是一次生成一批行结果，还是每次生成一条？

初步的方案是一次生成一批，这样可能会浪费一些拷贝动作；

但是性能上不见得会比一次一条的低；

Group的中间结果因为需要上层的表达式可见，所以需要特殊的表达式节点，并且需要产生一个特殊的表定义结构，便于表达式节点访问具体的数据

## 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；

## 关于字符串的设计

字符串分为两种，一个是定长字符串，另外一种则是变长字符串；

目前对于定义小于16字节（暂定）的字符串，设计成类似于整型的形式，占用固定的16字节长度，其他小于16字节的，则找一个2整数幂的长度存储；这个是最简单一种形式；

对于变长字符串，则无法使用这样的方式存储，比较浪费空间，另外对于对齐的设计也比较困难。因此变长字符串采用另外一种设计，即列的结构使用定长的指针（也可以是类似句柄的定长数据类型），通过指针可以找到数据的真正位置，而变长字符串的数据，使用连续堆的方式紧凑存放。

则当出线空洞时，需要标记删除行，需要增加《删除列》，对于变长字符串列，需要设计单独的内存管理，继承自BUDDY的内存管理算法；使用最大的页面申请内存，采用页内双向链表进行管理；

## 关于后台进程

进程模型目前计划是采取多线程+多进程的模型，基于hyper的方法，更新是在主进程内完成，查询后续可以fork进程+线程；当前阶段，为了论文的方便，现在都做到一个进程里面；

通讯协议，计划用PostgreSQL的，因为再单独设计一个协议意义不大，而且使用PostgreSQL协议的另外一个好处是暂时先用PostgreSQL的客户端，避免重复开发，可以快速出原型，等比较成熟了，可以再使用PostgreSQL的代码进行改进；

为什么不用SQLite的？因为SQLite是嵌入式的，C/S的代码不是官方的，且功能不全，意义不大。

为什么不用MySQL的？因为MySQL的许可协议时GPL，虽然MySQL的使用更广，但因为许可的问题，后续不方便扩展功能。

## 关于基础数据库代码选择

开源数据库，以C/C++为源代码的，主要有以下几个：

MySQL, PostgreSQL, Firebird, SQLite

MySQL功能强大，性能高，目前是使用的最广的开源数据库了，但许可是GPL的，就这个就直接PASS了；

PostgreSQL 是目前功能最全的开源数据库，使用的BSD协议，greenplum即基于PostgreSQL改造的，其他比如早期的heroku也是用的PostgreSQL，功能强大，但是代码量较大，之前没有基础，所以也暂时没考虑；

Firebird是目前第三的开源的数据库，是borland开源的，协议类似Mozilla协议，需要开放员代码，所以也不考虑了

SQLite是嵌入式中使用最广的数据库，使用Public domain方式，即无开源协议；是几个开源协议中最友好的，功能全，而且代码精简。但SQLite不足的地方，DDL是全局锁，并发性有效，且无CS模式；

因此使用的SQLite作为基础代码，后续看发展情况，也可考虑切换到PostgreSQL；

## 异步，同步？线程模型

传统数据库多采用的同步模型，即一个线程或一个进程对应一个连接，这样简单可靠，但问题是连接数多的时候，性能会影响较大；

Thor的处理模型中本身就会涉及到多线程分解运行一个SQL，所以多线程协作是免不了的；

但多节点协作中的客户端，即需要用到异步模型；

计划采用异步模型，网络通信采用libev库封装；采用reactor模型，其中主线程负责监听，每个session的会话始终在一个线程，每个线程可以设置多于一个SESSION(初期先设置为一个session一个线程)，其他的操作均在这个线程中完成，包括和其他节点的连接，具体执行；

如果操作涉及到多线程，则IO线程为SESSION的主线程；

## 关于游标的设计

增加两种游标，实际使用时，根据参数来决定：

1. 均分范围型游标，相当于将一个表拆分成多个表
2. 资源竞争型游标，公用一个表，但都全部遍历，只是每一个段只被遍历一次

目前先使用资源竞争型游标做DEMO

## 关于HASH JOIN

HASH JOIN 并行需要先在两表中选取小表来做HASH TABLE

做HASH TABLE有两种模型，一种是本身就是基于HASH分区的，则每个JOIN节点自己做HASH TABLE（也可以运行时生成）

另一种则是通用的场景，需要整个表可间HASH TABLE，这个时候需要支持一个并行插入生成的HASH 表，这个时候效率是很高的，详细论文可以参考一下（之前看过，忘了是哪篇了）

**增加一个类似 prepare的阶段，并且这个阶段可以设置线程屏障，需要同步后再一起执行**

## 关于MERGE JOIN

Hash join 比较好做并行，主要是因为HASH JOIN适合大小表的情况，如果两个都是大表，则不太好处理。这个时候使用MERGE JOIN会比较好。

之前MONETDB的论文中，提到MERGE JOIN，但方式是将MERGE JOIN替换成HASH JOIN，因为他们觉得：

MERGE JOIN有几个缺点：

第一个问题是计算的消耗比较大

第二个问题是需要序列化整个表，这个内存消耗太大；

对于第一个问题，HYPER上经过实践，SORT的性能并行后续的优势会比较大，如果使用多种排序组合，特别是整形数据，使用RADIX SORT性能是比较好的，另外，有可能数据本身就是排序的；

对于第二个问题，内存问题，目前没有很好的办法，但如果基于分布式的场景，未尝不能考虑

综合来看，MERGE JOIN目前可以先不做，但后续需要考虑

## 关于STAR JOIN

星型连接即多个表（通常是小表）和一个大表连接，更复杂的连接则是雪花表；

星型连接在实际的数据仓库中非常常见，因此需要后续考虑进行合理优化。

## 关于MC JOIN

目前的HASH JOIN性能和MC JOIN很接近，暂时不考虑引入MC JOIN

## 关于LOCK FREE (免锁)设计

性能关键的地方可以考虑使用免锁设计，目前使用的是liblfds-6.1.1

## 关于多级并行的问题

当出现多个耗时计算节点时，目前的设计考虑是采用多级并行方式；

Select a.f1, count(a.f2) from test1 a, test2 b where a.f1=b.f1 group by a.f1;

解决并行的方式是通过多级并行方式，在group和join节点增加 exchange(n>m)节点

## 关于线程池设计

现在计划是采用TASK驱动多线程，线程池设计有两种模式：PUSH & PULL

PUSH的方式相对来说比较优雅，每个线程缺省进入睡眠，当有任务时，有监控线程监控状态，并指定工作线程唤醒；需要使用PTHREAD的CONDITION

POLL模式比较简单，每个线程共享一个队列，队列可采用免锁设计，当线程竞争获得任务后，进行工作状态，当队列没有内容，则进入SPIN状态，如果长期无内容，则进入睡眠；

POLL在线程数较多的时候有一定的影响，不过目前SQL的线程都是计算型任务，不是IO线程，没有阻塞型操作，因此线程太多没有意义，并且唤醒任务时通常是多个任务同时启动，所以建议缺省使用POLL模型，后续可以考虑优化；

## 关于多线程执行框架

语法解析完成后，先生成单线程的执行计划

根据单线程的执行计划，根据成本原则生成多线程的执行计划

* 使用多线程的规则，可根据成本或者HINT方式决定
  + 本身执行节点是否支持多线程
  + 当前环境的CPU个数
  + 表规模大小
  + 是否使用了HINT

主线程启动后，根据执行的并行度，申请线程资源；