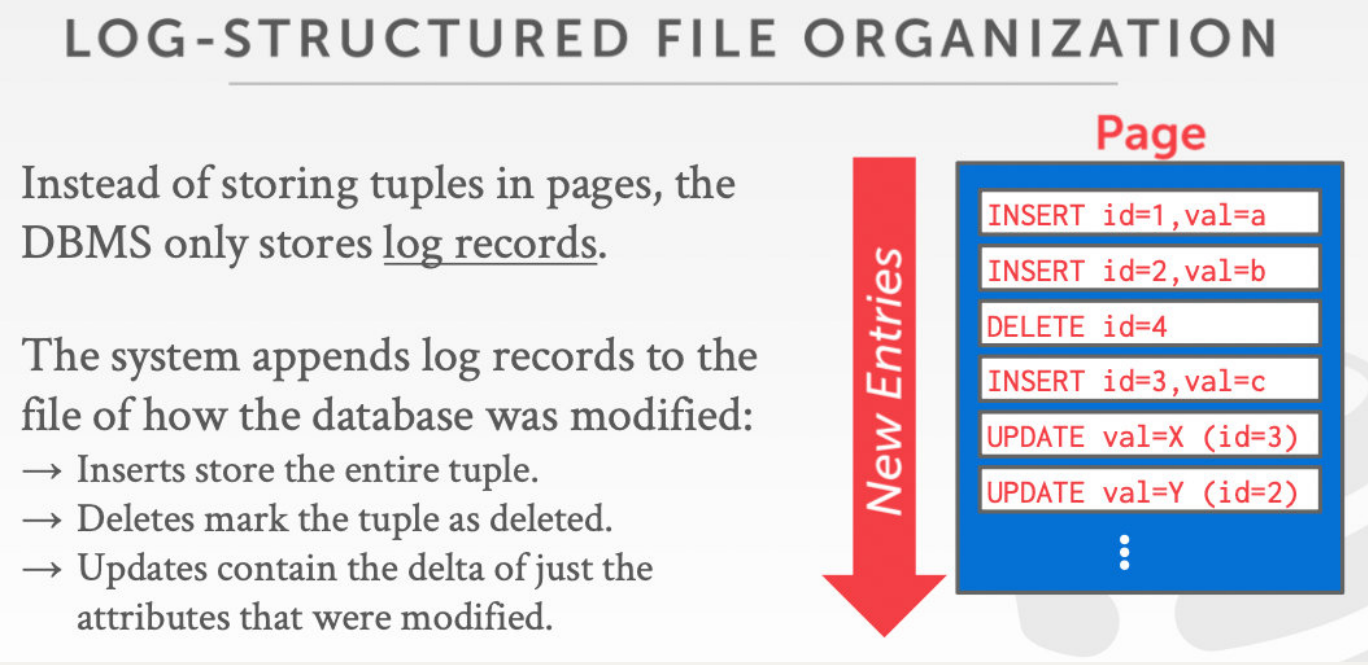
Log-structured的page layout

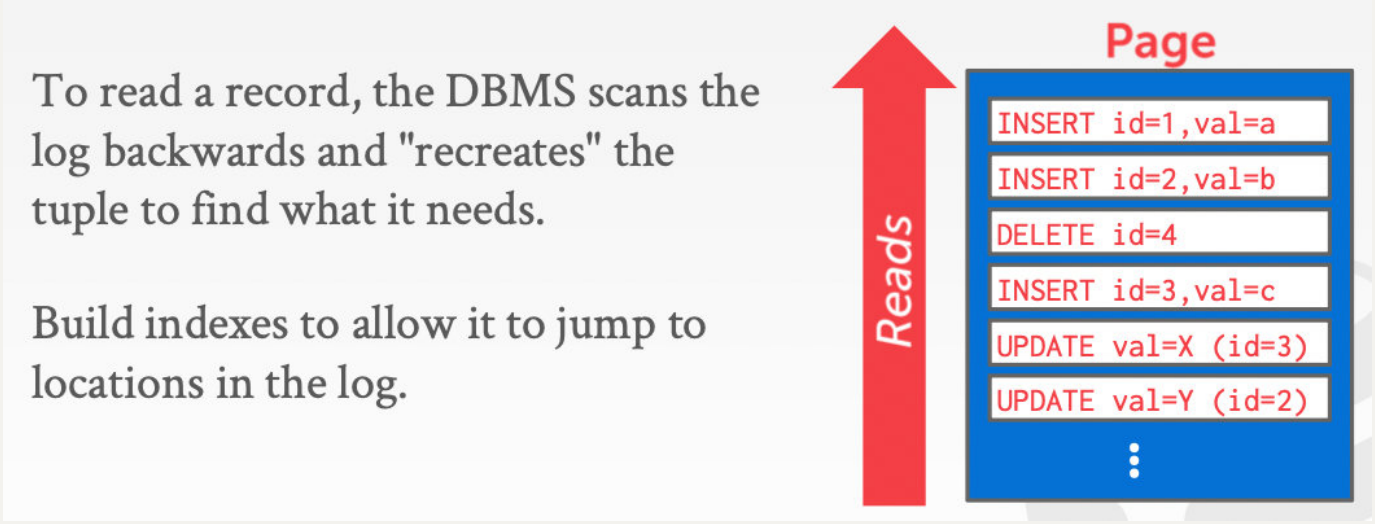
上一篇Blog中介绍的是Tuple-oriented的page layout，接下来介绍Log-structured的page layout，在这样一种数据库文件组织方式下，tuple中存储的不是数据本身，而是数据的log，即数据的变化：

- 在向数据库中插入数据时，会存入一个tuple，tuple中存储的不是数据本身，而是记录了“我插入了一个什么样的数据”
- 删除一个tuple时，不需要把这个tuple从数据库文件里除去，而是要新增一条数据，新增数据的内容是“我把xxx数据给删了”，也就是说，在删数据时，不是真正的删去数据，而是新加了一条log
- 在修改数据库文件的数据时，不是真正的去修改文件里对应位置的数据，而是新写入一条log，log记录着“那条数据被我改掉了”

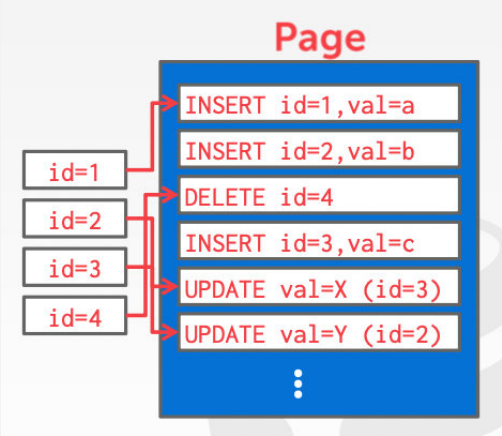
上述这些概念对应的实例如下图右侧所示：



在这样的数据库中，我们想要读某个数据的话，就需要从下往上的回放log，因为我们想要的数据有很大的概率是从log的中部及以下的位置第一次被插入的，如果从上往下回放log，那么开销太大，刚刚说的过程如下所示



如果这样还觉得慢的话，可以再做一个索引/日志，把和id=x有关的操作都汇集起来，存在某个特定的数据结构里，当我们查询的时候直接基于这个特定数据结构的某个实例来回放，就像下图一样

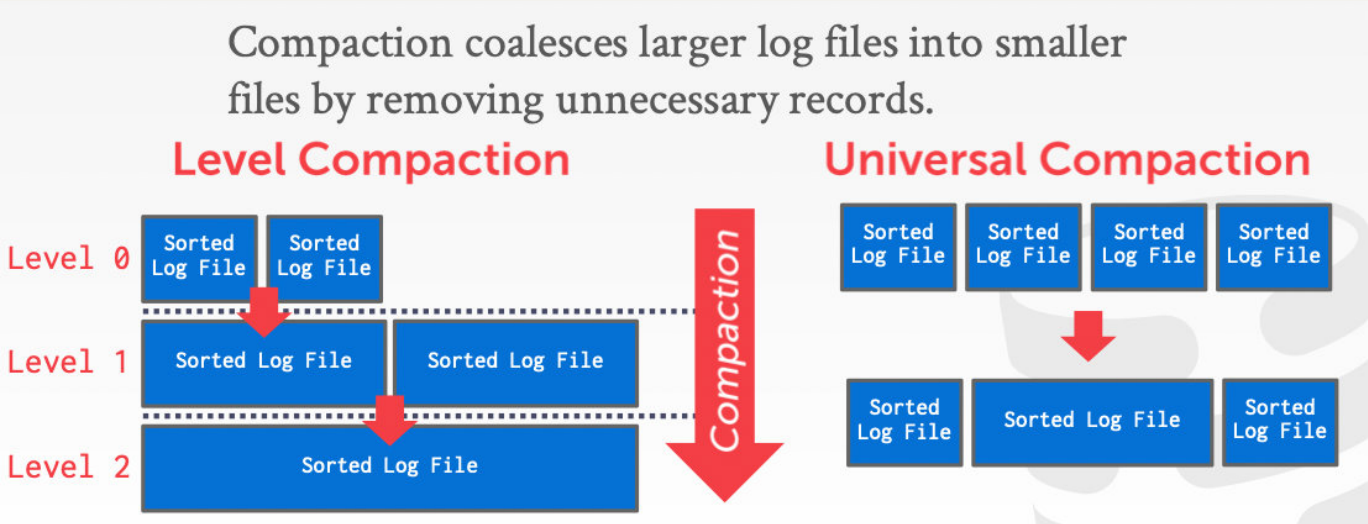


但这种基于log的数据库有一个严重的问题，数据库占用的存储空间会很庞大且冗余（比如说我们对某条数据的频繁修改就会制造很多的log写入数据库文件，之后对这条数据的每次查询都要把这些相关的log全都回放一遍，效率很低），因此这种数据库一般会周期性的压缩日志（比如说某个tuple的某个字段先被修改成1，又被修改成2，那么就可以把前面那次修改成1的日志删去，因为这样不影响后面的任何操作），这种压缩日志的思想的实例如下图所示，下图就是上面的图的log经压缩后的结果

id为1，2，3的tuple被插入时的log都是写在这个page里的，因此DBMS可以基于log推断出它们最终被修改成了什么值，因此只需留下这一条信息即可，id=4的tuple被插入时的log不在这个page里，因此只能留下一个它被删除的log

这种基于log的思路与机制更多的被用在KV数据库当中，因为KV数据库是一个键对应一个值，不像关系型数据库的一个tuple里面有好几个字段，这样的话，KV数据库每次tuple被更新之后，可以直接基于更新的log来得出K对应的V当前是什么值，反观关系型数据库，某一个字段被更新后，在log回放/压缩时，我们还要查看其他字段的log，检查其他的字段此前有没有更新，效率比KV数据库低的多

压缩日志的方法也不止一种，首先是层级压缩（Level Compaction），对于像前面的例子里id为4的tuple，记录它被插入和后续更改的log page不是同一个page，我们就做不到在单个的page里完成彻底的日志压缩，如果我们能同时读取这前面提到的这两个page，把它们里面关于那条tuple的日志整合一下，就可以完成最终彻底的压缩，即有关于这个tuple的日志最后只有一条，如果两两page合二为一还是无法彻底压缩，那就再度合二为一，一层一层向下合并，有点像Linux的Buddy System，如下图左侧所示



Rocks DB也是基于这个原理实现了日志的压缩，而且它最多能压缩到第七层

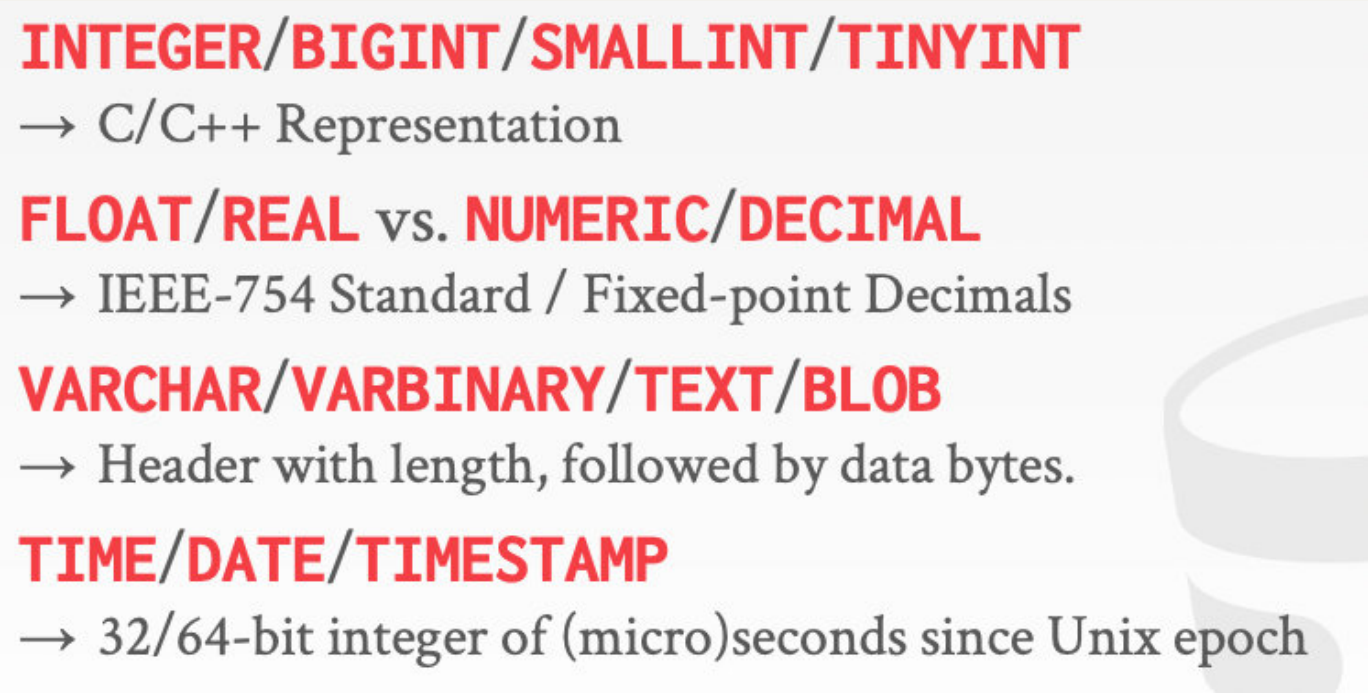
压缩完了之后，我们再从数据库读数据的时候，从第0层开始读，如果没有找到，那么有可能就是想要的数据经过了层间的压缩合并，被存到了下面的第一层，如果还没找到，那就有可能在第二层...

上图右侧展示的是Universal Compaction，它没有层的概念，压缩的过程就是两个存有log的page被合成了一个更大的log page，跨page记录的log也被压缩，这种方法更通用

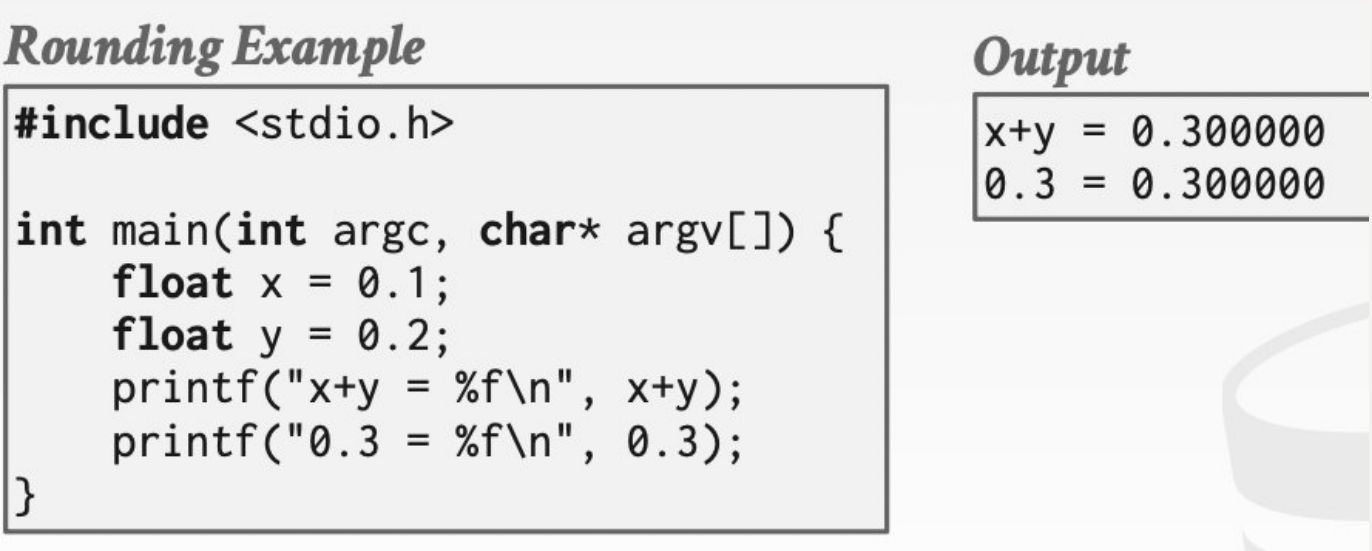
前面有提到这种基于log的存储引擎的一些坏处，但它也有好处，它可以在向数据库插入数据时把本该向存储设备随机的写转换成顺序的写，如果是tuple-oriented的存储引擎的话，我们想修改Key=xxx在数据库里对应的tuple的话，就得先在数据库文件里找到相应的页，然后读入内存，完成数据的修改，然后写回磁盘，但倘若换成这种基于log的存储引擎，我们只需在修改数据库内的数据时，新建一个数据库文件的块，或者使用当前还没写满的块，直接向里面记日志就行，在大量持续更新数据库的时候效率很高，但前面也提到了，在读数据库的时候，效率一般，因为要有日志的回放，还有就是压缩的时候比较麻烦，因为每次修改都有日志记录会不可避免的导致数据量非常大，压缩操作本身也是有开销的

Data Representation

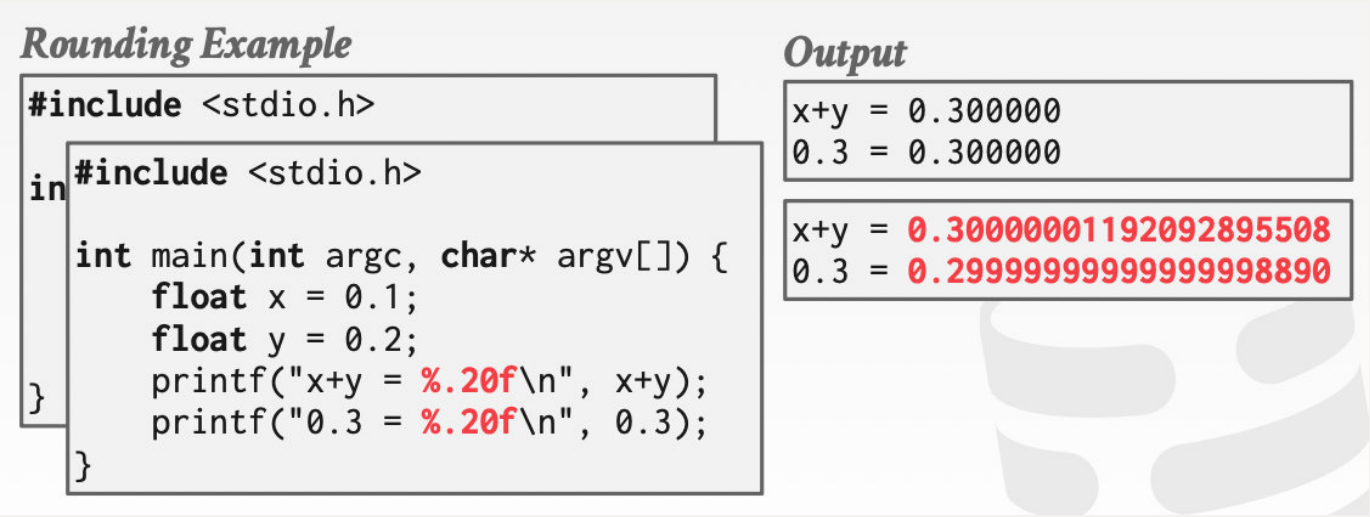
接下来我们回到tuple-oriented的存储引擎，讨论DBMS存储的数据在磁盘上的存储格式，数据库的tuple里面的数据类型一般都是下面这样，和我们平常使用的高级编程语言提供的数据类型不完全一样



首先是关于实数/小数，我们直接把float/double类型的数据直接按照C/C++的编码方式存到磁盘上会有一些问题，以下面的使用浮点类型变量的C程序为例，默认情况下也许我们看不到什么异常的地方

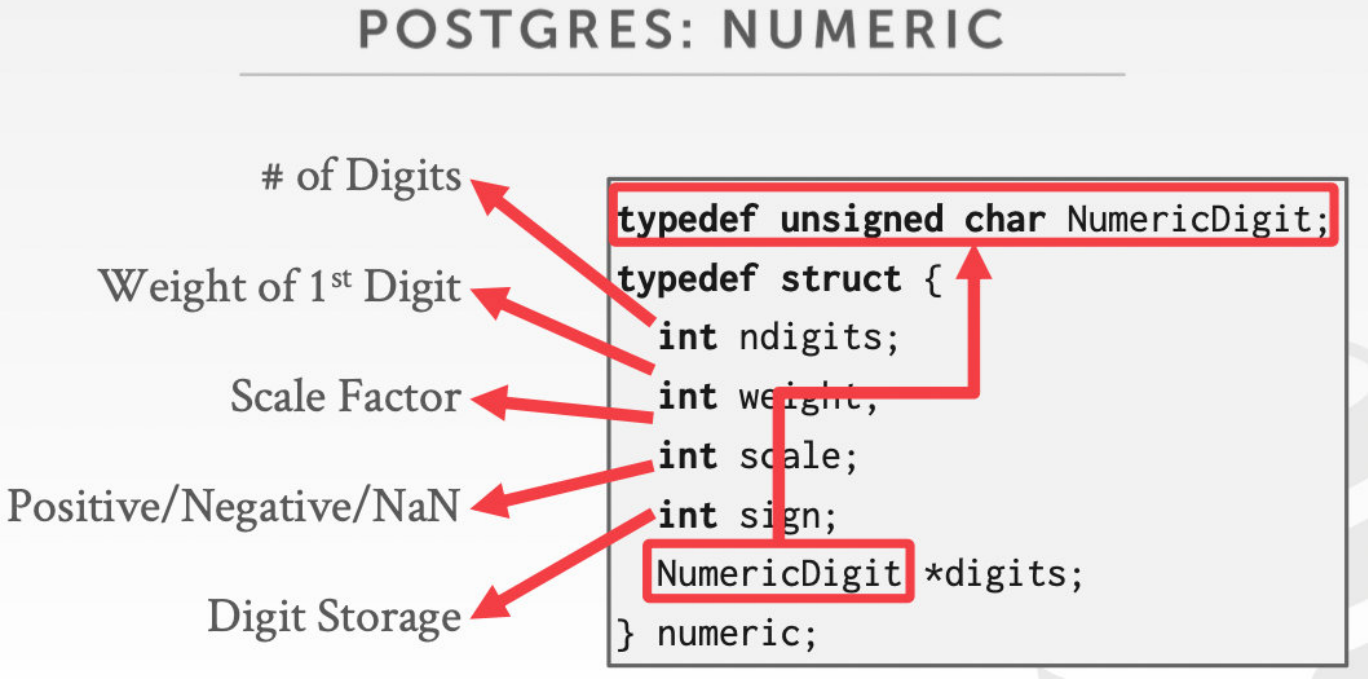


但如果我们将C程序中的格式串设定成“打印小数点后20位”



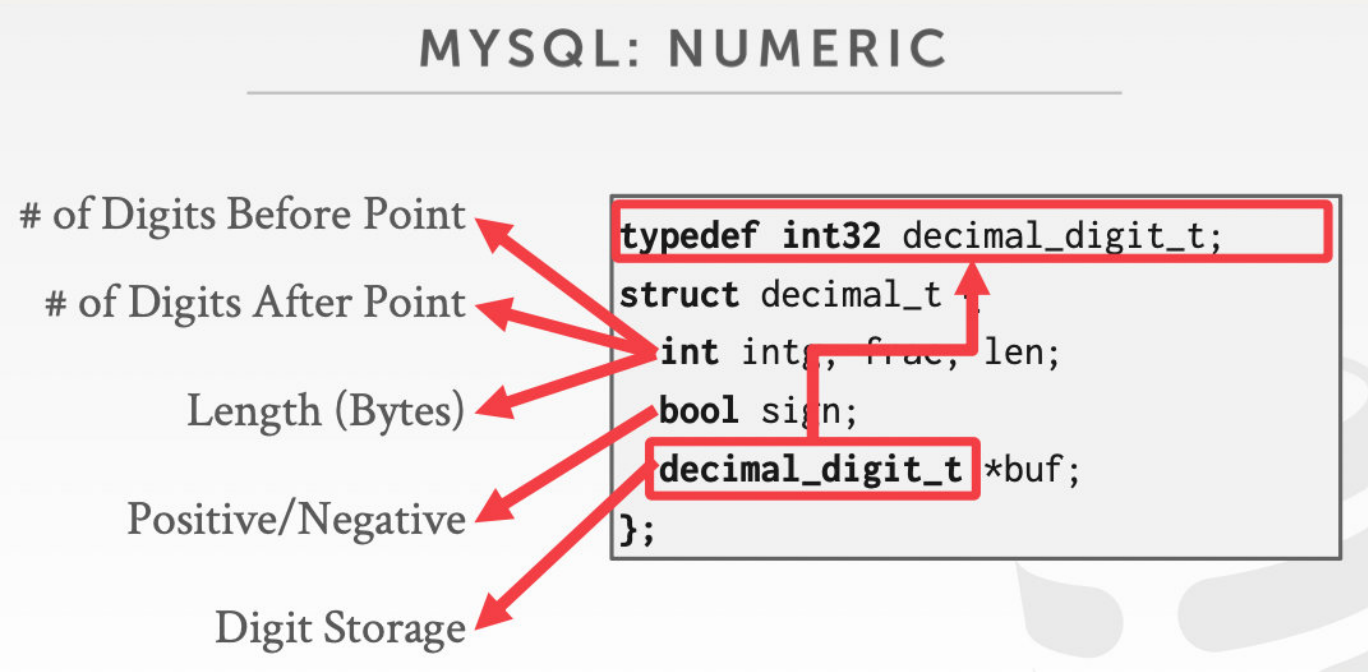
我们就会发现，我们自己以为程序中的数值是精确的0.3，实际上在计算机的内存中存储的并不是我们想象的内容，当小数点后面的位数不多时，打印时会做四舍五入，但如果要求打印小数点后n位这种精确数值时，它就不是我们想象的那样，在实际的软件工程业务开发时，和钱有关的数据，也是严禁使用浮点型的

数据库为了准确的存储，采用的策略是“在字面上存数据”，把数据变成字符串，那么它就是绝对标准的了，我们以Postgres数据库内部的实现为例来分析，如下所示：



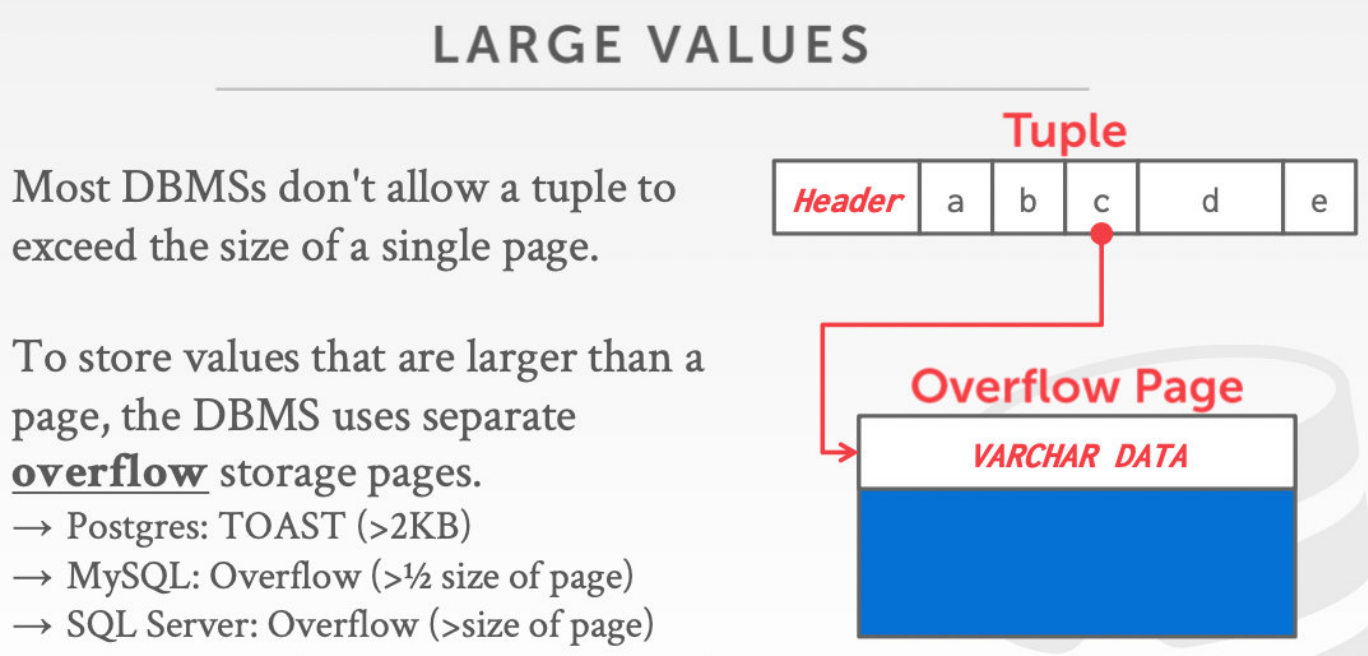
`ndigits`记录的是数据有多少位，`weight`记录的是“是几进制”，`scale`记录的是类似于科学计数法中的指数幂次，`sign`是符号位，`digit`是指向字符串的指针，比如说要存0.35，那么`ndigits`就是2，`scale`就是-2，`digit`就指向"35"，最终读取对应的`numeric`类型的结构体，就能从这个结构体“还原”出我们存储的数值是多少

MySQL相关的内部实现如下：

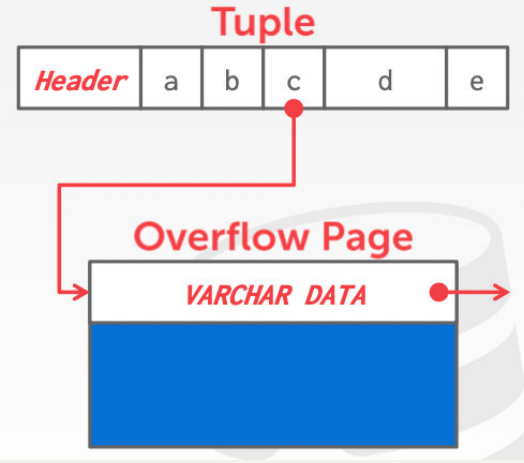


`intg`记录的是小数点之前有多少位，`frac`记录的是小数点之后有多少位，`len`记录的是“该数据有多少字节长”，`sign`是记录正负用的

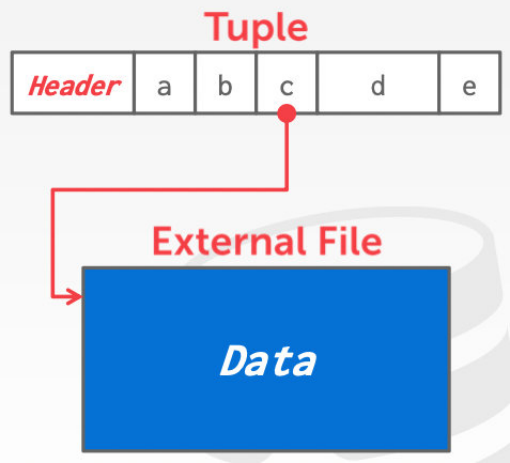
这就解决了有关于小数/实数在磁盘上的存储问题，接下来讨论"large values"的问题，即“超长字段”，举个例子，如果数据库的tuple里面有一个字段是字符串，我们用这个字符串存了一个小说，这个字符串会相当的长，甚至比存储引擎所管理的page还要长，存储引擎会使用Overflow Page，即“溢出页”来解决这个问题，如下图，c是tuple里的超长字符串，我们把c的内容存到溢出页里，并且构建起这个tuple对溢出页的“引用”关系（记录溢出页的地址之类的操作）



上图也列出了各个数据库中触发溢出页的条件，要是“超长字段”比溢出页的大小还要大，那就再多用几个，它们之间像链表一样连接起来，如下图：



这样做的话很浪费存储空间，因此就有了外部文件的构想，我们可以在其他的设备，诸如网盘上存储超长字段，我们只需在tuple里记录外部文件的地址，如下所示



这样提升了磁盘的利用率，并且操作系统的磁盘I/O也是以页这个量级为单位的，要是一次磁盘I/O只能处理一个tuple的数据，也是十分的低效，外部文件的存在也改善了这一点，此外，外部文件也有一些问题，如果其他的软件/进程也可以访问并修改这些外部文件的话，就有些麻烦，我们需要额外管理

System Catalogs

System Catalogs，即DBMS的目录，用于存储DBMS的元数据，比如说表结构，列的结构，索引，视图，用户，权限，内部的统计信息，数据库的元数据都是以表的形式存在的

A DBMS stores meta-data about databases in its internal catalogs.
→ Tables, columns, indexes, views
→ Users, permissions
→ Internal statistics

Almost every DBMS stores the database's catalog inside itself (i.e., as tables).
→ Wrap object abstraction around tuples.
→ Specialized code for "bootstrapping" catalog tables.

Database Workloads

Database workloads，即数据库的工作负载，分为如下三种

On-Line Transaction Processing (OLTP)
→ Fast operations that only read/update a small amount of data each time.

On-Line Analytical Processing (OLAP)
→ Complex queries that read a lot of data to compute aggregates.

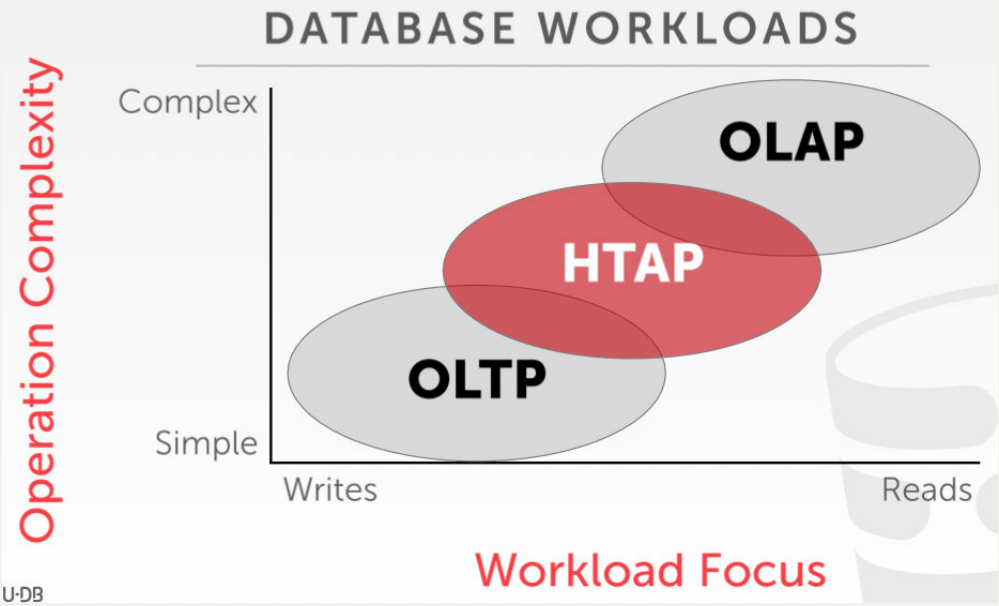
Hybrid Transaction + Analytical Processing
→ OLTP + OLAP together on the same database instance

OLTP，在线事务/交易处理，支持快速的操作，读写数据量很小的tuple，而且一般是高并发的，像微信支付这样的场景，同时发生很多交易，但每笔交易涉及的数据量都很小

OLAP，在线分析处理，OLTP这种workload一般来自用户，OLAP则来自于提供相应软件服务的公司本身，比如说微信支付要统计当天有多少笔交易，交易的平均金额是多少，交易总额是多少，这一般对应着复杂的SQL语句，要读大量的数据

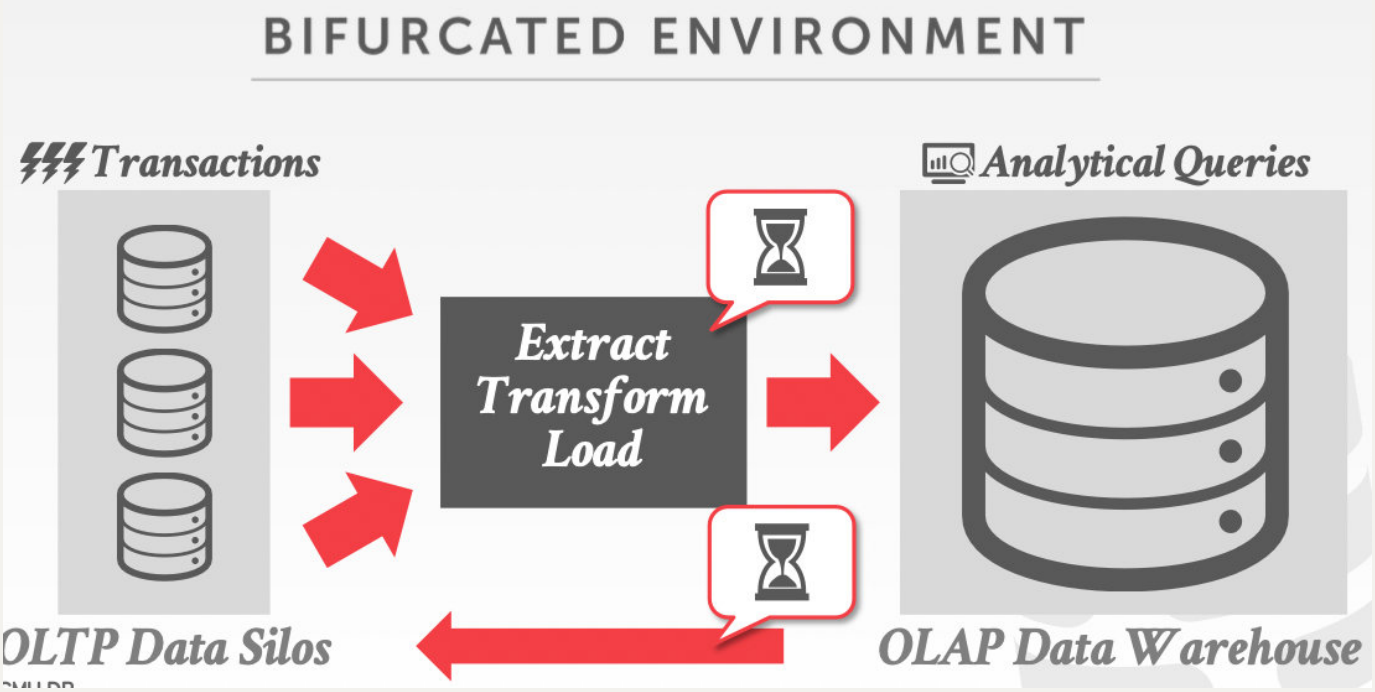
第三种，HTAP，则是前两者的混合，意味着可以同时应对这两种类型的workload的数据库架构

OLTP更倾向于写，OLAP更倾向于读

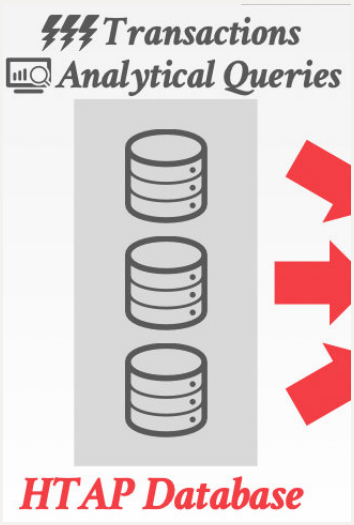


在不考虑HTAP的情况下（目前工业界HTAP的实践并不多），软件公司会将OLTP和OLAP这两类工作负载分开处理，用小数据库的集群处理OLTP，用分库分表的策略将工作负载分开，等到需要统计分析数据（即OLAP对应的场景）时，对这个集群执行ETL操作（提取数据->做变换->加载），把得到的数据转存到大的数据仓库里，然后让数据仓库应对OLAP的

负载，这样将两种工作负载的处理分离开来的好处是，应对软件公司内部统计分析时的OLAP负载时，即便是执行复杂的SQL语句，也是在数据仓库上执行，不会影响集群，进而不会影响处理OLTP的效率以及用户体验，同时，小规模数据集群的架构也往往不适合运行复杂的SQL语句，可能光执行这些复杂的SQL就足以让它们挂掉，在数据仓库应对完OLAP的负载，即完成数据统计后，可以将统计后的结果写回小数据库集群，我们喜闻乐见的“用户年终总结”就是这么来的，刚刚说的这些概念形象一点的展示就是下图：



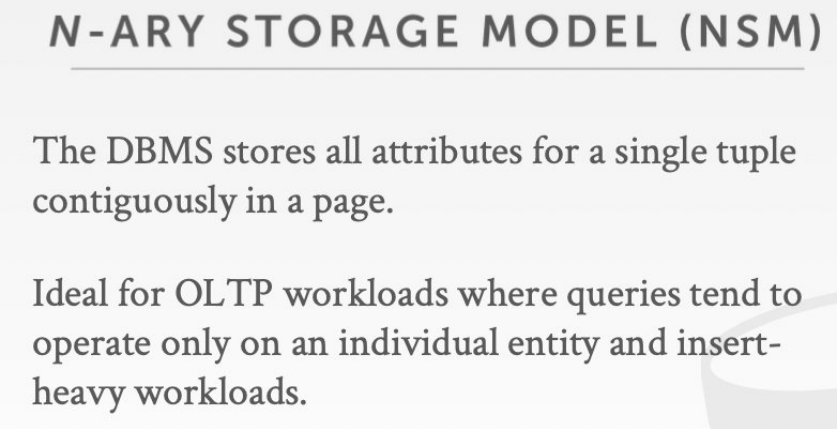
如果我们有足够好的HTAP型数据库的话，只需要它一个就可以了，节约了资源，也少了很多数据处理交换的过程，不过工业界HTAP做的好的并不多



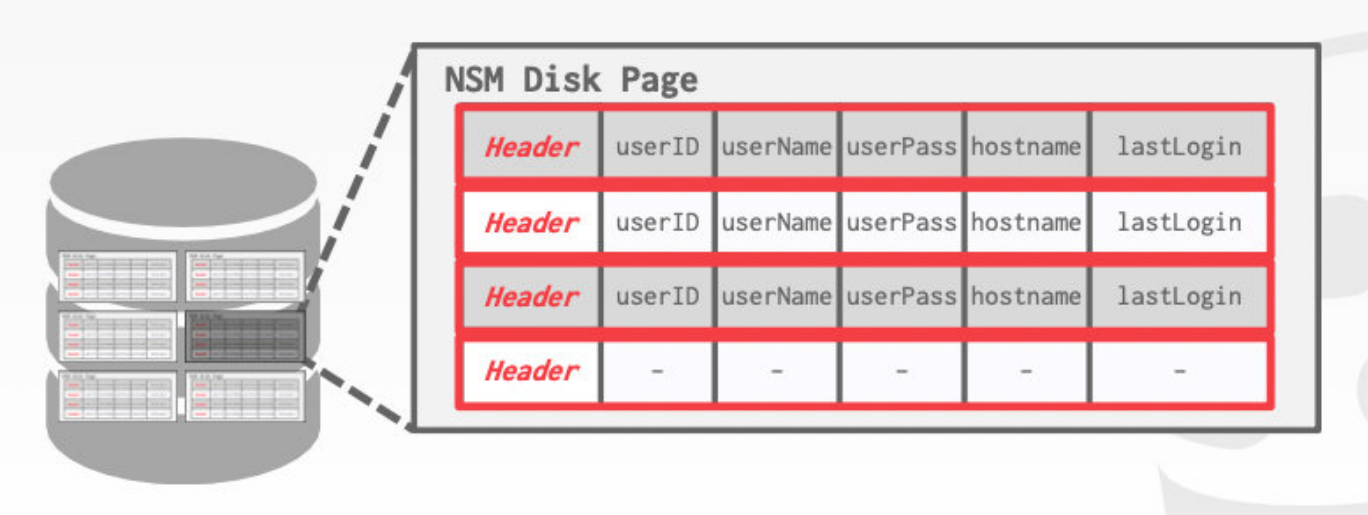
行/列存储

数据库的关系模型和SQL语句并没有要求或者限制数据库底层的数据存储方式，只要通过层层的向上封装能满足它们想提出的抽象即可，即“SQL想查出来一行一行的数据，但数据库不一定在磁盘上一行一行的存”，我们可以修改底层的存储方式，进而更好的应对OLTP/OLAP

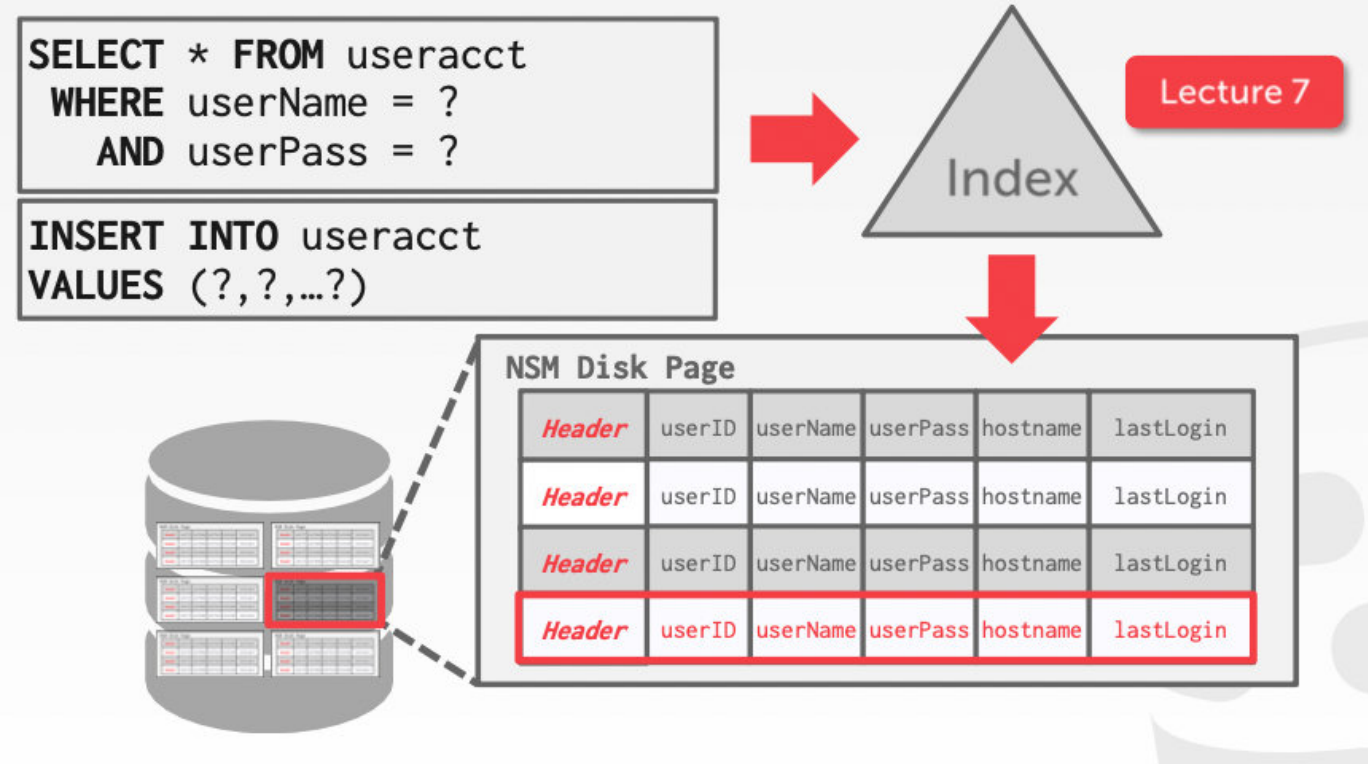
此前一直谈论的存储引擎采用的存储方式都是行存储，行存储的概念如下，它明显更适合应对OLTP



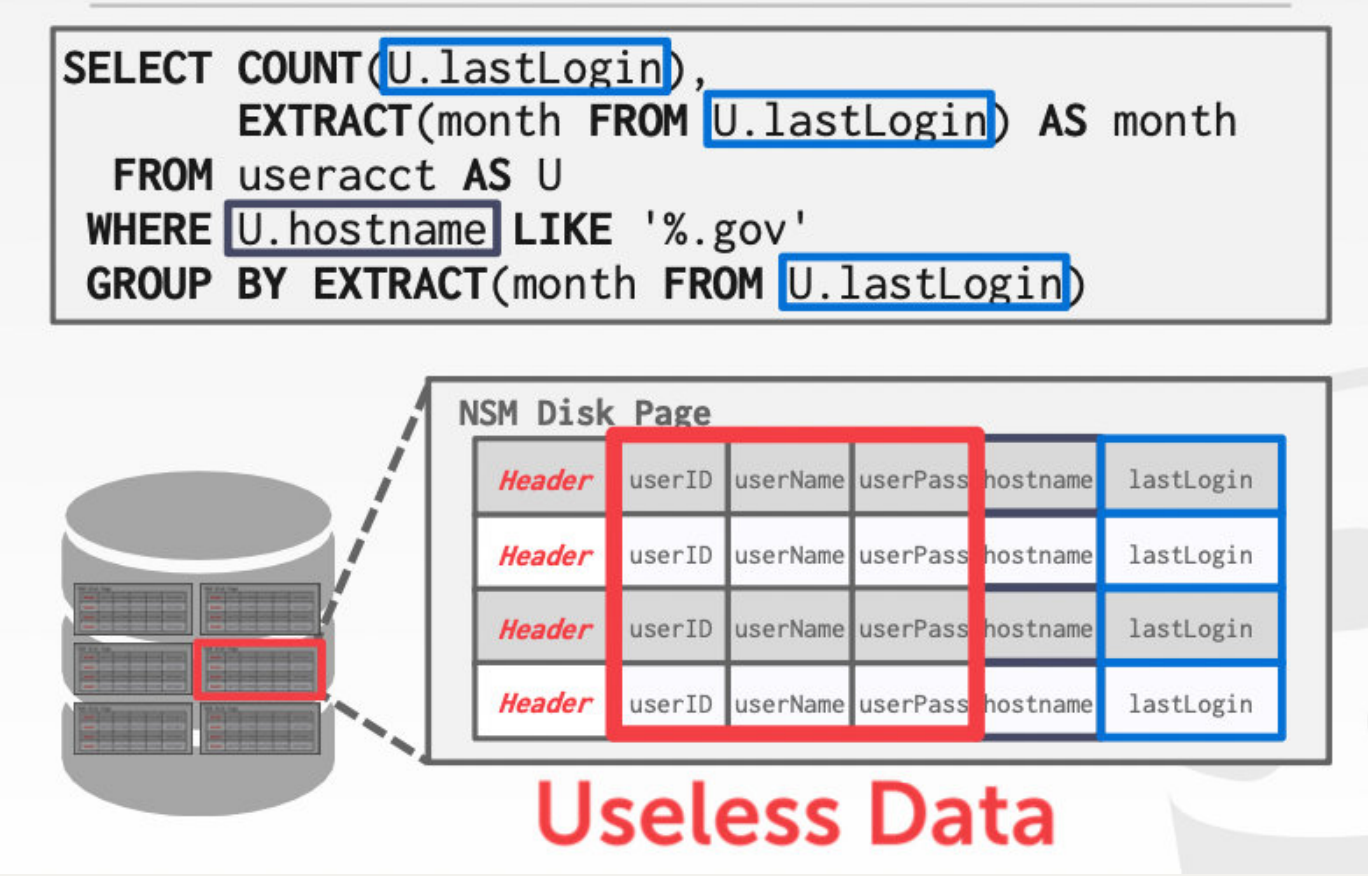
用画图理解行存储大概就是下面这样



应对OLTP负载时的工作流程如下所示，效率很高



但如果我们有OLAP的负载，比如说维基百科想统计所有登陆后缀是.gov，即政府用户的登录情况，如果是行存储的数据库，那就需要扫描数据库文件里所有的页，流程如下图：



在这个过程中，我们并不关注每个行的前几个字段，它们都是无用的数据，可是由于数据库底层是行存储的，我们还是把这些无用的数据从磁盘里读了出来，并且解析了（比如说要读tuple header里的bitmap，还要根据小数存储采用的数据结构来解析小数，只有把前方的无用字段解析完了才能解析我们想要的字段），总之就是行存储在这种情况下会做很多无用功

行存储的优势和劣势总结如下：

Advantages

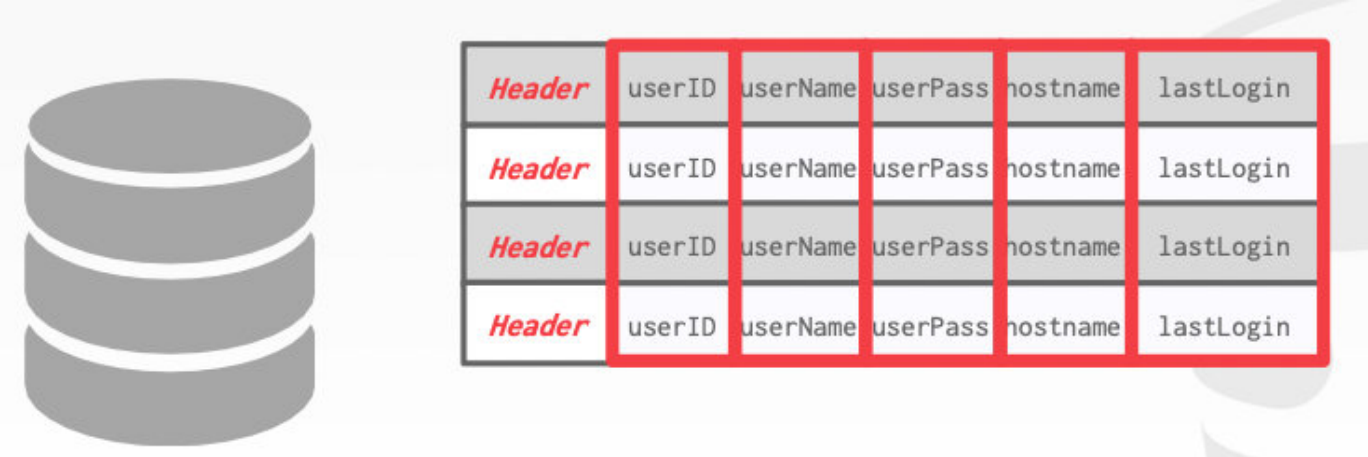
→ Fast inserts, updates, and deletes.
→ Good for queries that need the entire tuple.

Disadvantages

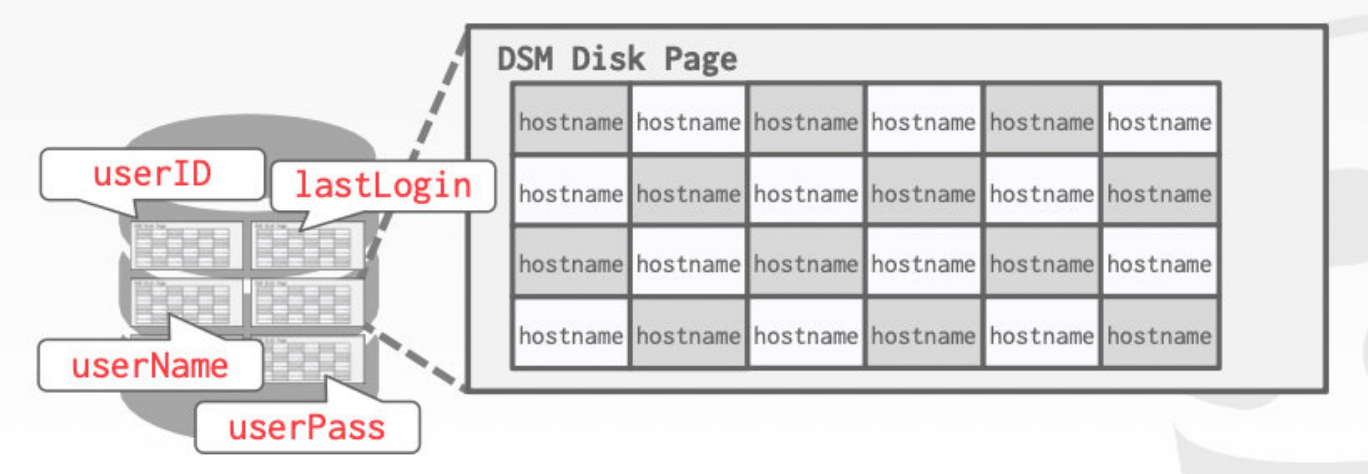
→ Not good for scanning large portions of the table and/or a subset of the attributes.

因此就有了列存储，一个tuple存的是之前概念里的一列数据，比如说在一个记录员工信息的数据库里，一个tuple用来存储所有员工的身高，一个tuple用于存储所有员工的性别...这很有利于数据分析时OLAP负载，图解如下，

数据库的表逻辑上还是这样

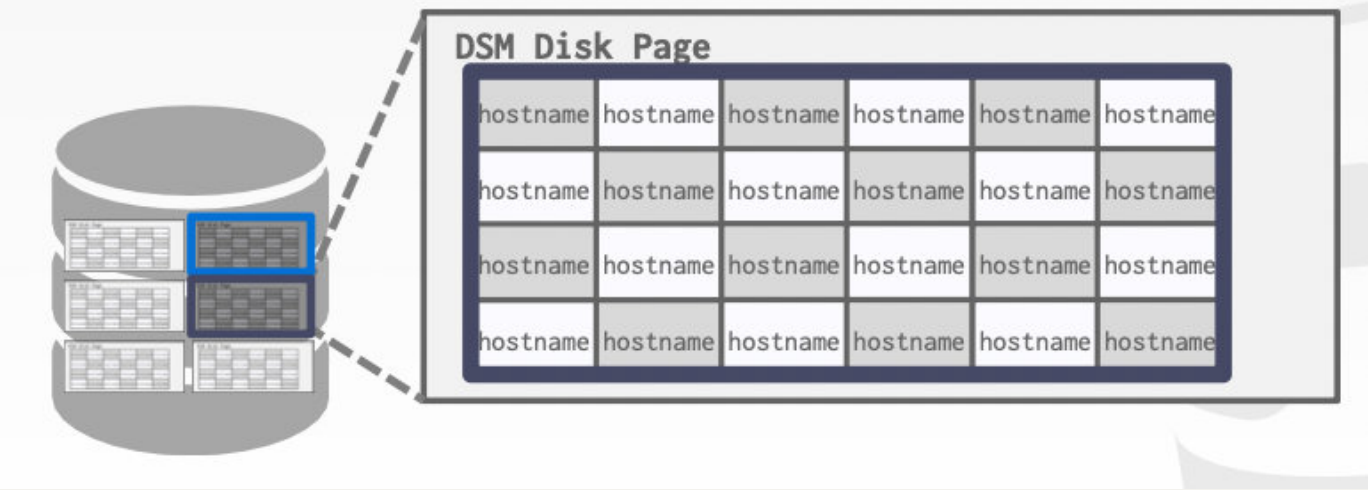


实际上的存储形式如下，不同的页用来存不同的字段/列

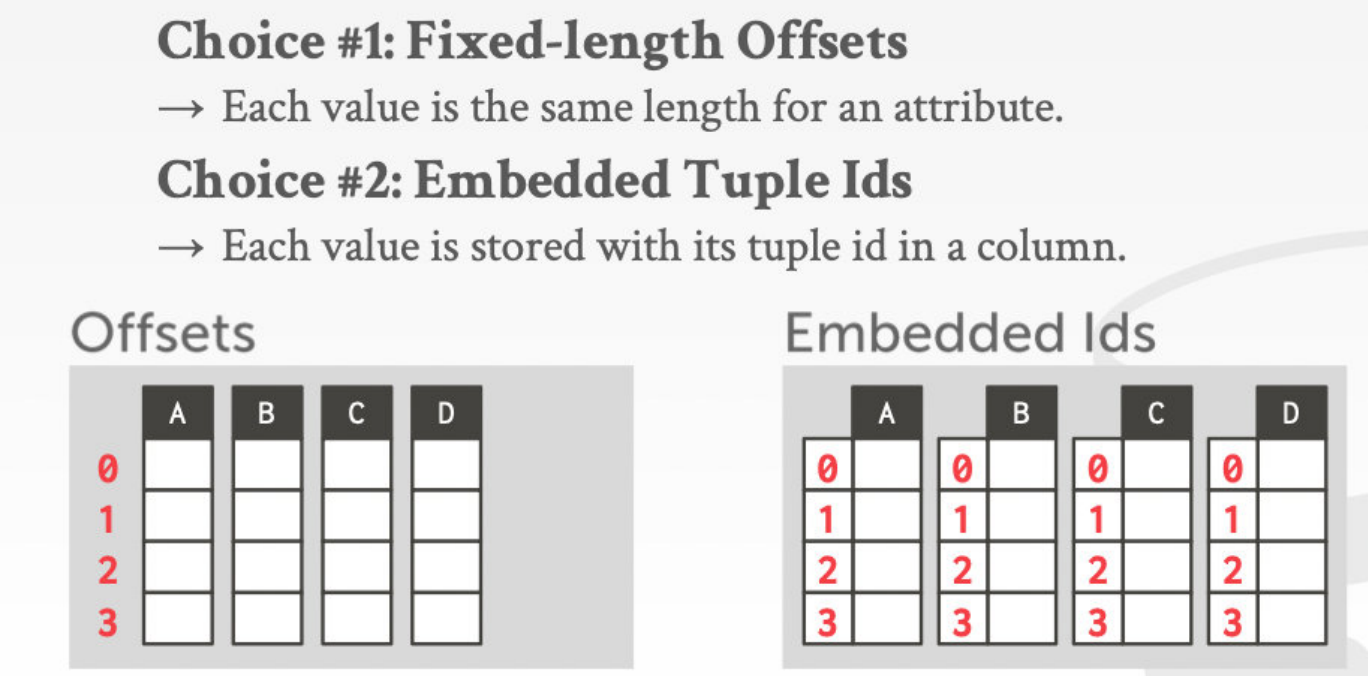


回到刚才的OLAP的例子中，列存储的情况下，便可以应对自如：

```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

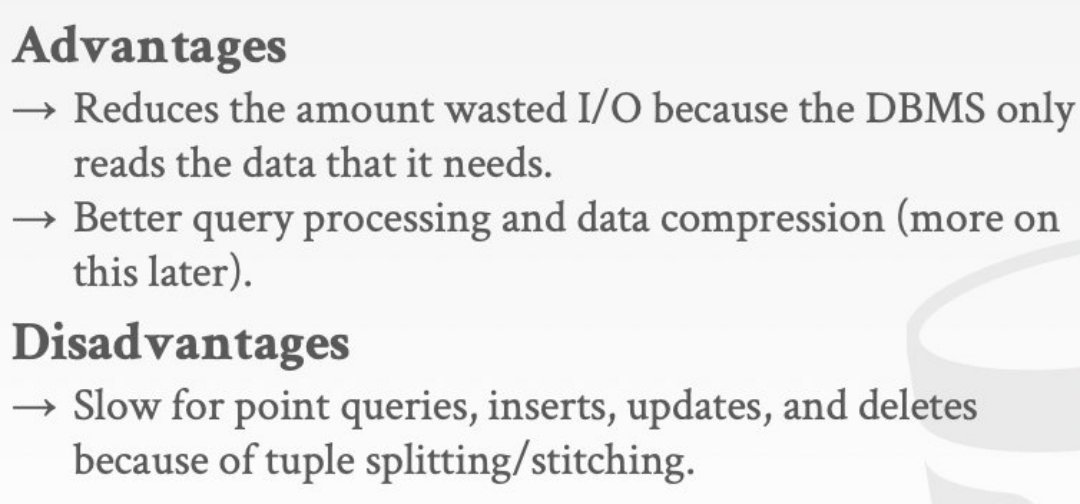


那么，在列存储的情况下，我们如果想像以前一样，读整行的数据，该怎么办?
有如下两种解决方法：



1. 我们基于offset来找到逻辑上的某一行的各个字段，然后把它们拼到一起
2. 第二种方法比前一种更浪费空间，我们连续存储的每一列数据，不仅要存储该字段对应的信息，还要额外记录它的ID作为索引，但这么做的话在找的时候比较好找，不用像前一种方法需要计算偏移量

总结一下，列存储的优势/劣势如下：



最终，本Lecture总结一下就是，存储引擎的架构和数据库的上层设计很紧的耦合在一起，我们需要根据不同的工作负载选择不同的存储模型

Reference:
https://www.bilibili.com/video/BV1sP4y1H7uB/?spm_id_from=333.788