第一章：简介

像 MongoDB,CouchDB,Riak, Cassandra,或者 HBase 这样的 NoSQL 数据库系统的主要优势在于它们的对现代应用的后台存储的方便性。这些数据库系统让应用能够更加灵活地存储和读取，无需事先定义一种模型就可以获取数据。因为移除了事先数据管理的操作，开发者们能够更加容易的获取并运行他们的应用程序而不用担心由于他们数据集合的范围、类型和依赖性可能带来的错误。

有人质疑这种“立即满足”型的应用开发方式可能会长期导致在代码管理、分享、性能等方面的问题。但是 NoSQL 方法的支持者们认为在发展响应迅速地独立数据集的过程中，仅仅是维持模型的开销的成本就太高。无论这两种方法孰优孰劣，在实践中我们发现使用键值和半结构化的数据形式的数量正在增长。而有策略的分析研究这些海量数据的重要性也在不断增加。

有一些数据库系统一开始就支持对原始的数据进行分析。例如 MongoDB 现在提供了一系列的数据分析集成工具；MapReduce 也有分析数据的功能；而其他的 NoSQL 数据库系统则通过连接能够进行数据分析的开源计算框架（如 :Hadoop ,MapReduce ,Apache Tez ）来进行数据分析。

但是现在在使用 NoSQL 数据库系统的功能时存在非常明显的缺点。本地原始数据有时会和 SQL 的标准差很远，从而导致了很多第三方智能分析工具（SAP Business Objects, IBM Cognos, Microstrategy ,Tableau ）不能使用。同时，由于 Hadoop 提供了 SQL 接口用来存储数据（如 Hadapt，Hive，Impala）而这些功能要求用户在数据分析前首先通过 SQL

创建表。这也是 NoSQL 一开始就排除掉的主要原因。在有些实例中 ,模型可以容易的添加，但是其他情况中，数据必须首先要抽取-转换-加载 才能导入到有用的模型中。

传统数据库系统能够对多结构的数据(如 :关系、键值或者其他半结构数据)进行标准的 SQL 查询。而这篇论文讨论的是如何设计一层关系使得用户能够在任何情况下都不需要建立数据模型。基本的想法就是告诉读者一个普遍的逻辑关系【13，15】, 这个关系中对于存在独立的键列的逻辑表都在与之对应的独立数据集。而嵌套数据会被分解到不同的列中 , 因此可能导致逻辑表可能具有成百上千的列。

由于用物理方式存储这种数据是不切实际的 ,所以物理表示方法因逻辑表示方法的不同而改变。数据虽然存在关系数据库中 , 但是只有其中的一个子集真正的被定义成关系数据库中的一个列，而其他的列则是以连续的二进制列的方式存在与数据库系统中。当用户要使用时，这些数据通过内部解压的方式用这些连续的二进制列中提取出来。

我们主要的工作就是完成一个能够合理地完成上述操作的数据库系统的结构。这个结构包含以下内容：一个关系存储层；目录；表分析器；列具体化功能（column materializer）, 加载器； 查询重载器；转化数据索引。这篇论文描述了它的设计，启动还有各个元素之间的交互。

我们也设计了一个我们心目中的系统的原型 ,并把它和其他几种有多种存储方式和分析方式的系统进行了比较。（1. mongoDB 2. Postgre 的 JSON 拓展 3. 能够存储键值对的 EAV 模型的关系数据库）我们发现我们的原型在上述功能中表现更好而且还能提供更多的标准

SQL 接口。

第二章： 相关工作

分析性的系统并不要求用户定义一个表因为数据直接被系统以两种类型装载：

1.专门针对无联系数据模型的数据库系统，它没有专门的 SQL 接口

2.一般提供 SQL 或者类似 SQL 接口的数据库系统，它要求用户在查询前先定义潜在的数据的类型。

为了追求编程时的方便和灵活 ,第一种系统通常会要求使用者放弃一部分子集或者学习一种查询的语言。例如 :JAQL 使用迭代函数来提供了单元化数据的处理平台。不管它的拓展性，它不能使关系型的操作更加优化； MongoDB 虽然能够接受像 JSON 这样的数据形式，它还是要求用户学习它的基于 JS 的查询语言，但是这种语言不支持类似“关系连接”（relational joins）这样的操作。

第二种系统允许用户任意的定义一个模型（schema）。大多数数据库系统通过外部的表支持这样的操作。从而把它读到的无联系的数据转化成有联系的数据。这些外部表还可以以索引的方式提高数据库系统的性能或者加载到数据库系统中。很多使用了 Hadoop 开源环境的 SQL 项目（Hive 和 Impala）都用了这个相似的概念，也就是提供一个没有独立存储模型的关系查询引擎。数据存在 HDFS 中 ,而用户通过 SQL-on Hadoop 的方法访问他们的数据。

登陆了 schema 之后，用户可能输入查询命令，然后读取在 HDFS 上的数据然后利用 SOH 的方法完成他们的操作。

虽然这些系统比原来那些要在加载的时候预先定义一个模型的系统更加灵活 ,他们仍然有一定的限制。要完成 SQL 查询的命令，还是需要预定义一个目标模型来执行这些查询。而 Sinew 则不需要这些要求 ,自动的展示数据的逻辑视图通过数据本身而不需要用户的输入。

Google Tenzing, Google Dremel 和 Apache Drill 提供了无需定义 schema 就直接用 SQL 查询的方法。

Tenzing， 类似 Hive 的 SQL-on-MapReduce 系统通过潜在的数据推测关系模型 当且仅当 单调的数据类型能够匹配存在的关系模型。相对的，Dremel 和 Drill 还有 Sinew 支持嵌套数据。与 Sinew 不同的是 ,Dremel 和 Drill 只是查询执行引擎 ,只能单纯的提供分析。而 Sinew 被设计为传统数据库系统的拓展，能够支持半结构化数据和支持其他存在于关系数据库中的键值数据。Sinew 能够支持事务的更新，综合存储的接入控制，读写并发控制。Sinew 还可以继承原来关系数据库的优点：数据的统计和最优的查询能力。这个令 Sinew 与 Pathfinder 类似，但是 Sinew 比它更加广泛的支持所有多结构数据，为数据提供更加清晰准确的 SQL 接口。

为了能够清楚的为多结构数据提供一个全特征的关系接口，Sinew 使用了一种新的方法来在 RDBMS 存储多结构数据。一种普遍的方法是（曾被 XML 数据库使用过）是创建一个“shredder”来把文件传输进入 RDBMS。通常依赖于一些边际模型的改变来把文件装入关系元组中。但是，即使是潜在的关系 schema 对于预测和选择等操作是最优的，系统的性

能仍然被限制，因为即使重建一个最简单的记录也需要多次拼表。

最近，电子商务和其他网络应用使用一个 row-per-object 的数据映射方法。这检验了独立数据集对大数据的形成和存储的性能。这种方法要求对象能够整合进入每一个可能的键所在的元祖并且用具有很多列的宽表来存数据。一般来说，这种数据具有大量稀疏的键值，所以要求 RDBMS 的那些 NULL 的空数据不能对查询产生过度的影响或者造成性能的下降。面向列的 RDBMS 能够满足上述要求，但是它们在重建嵌套数据时存在困难。以为嵌套数据不是清楚的表示，所以只有用它们内部出现的键的集合来表示它们。我们将在 3.1.1 更加深入的讨论这些问题。

3.系统结构

从根本上说,Sinew 与之前的工作不同在于，这是一个可以查询和操作多结构数据而不要求用户在查询过程任何点定义模式的系统。 此外，它通过利用现有的数据库功能和委托查询优化和执行对一个潜在的 RDBMS 尽可能提供高性能的读和写。

虽然我们在设计中严重依赖 RDBMS，我们限制设计不要求改变 RDBMS 代码。 这大大扩展了 Sinew 的适用性，允许它被用来和大部分现有的关系数据库系统（最好是对象关系系统）合作而不要求转换到一个新的数据库系统。Sinew 因此应被认为是一个上面放着一个数据库系统的层。

Sinew 的结构如图 1 所示。 在最高层，系统包含以下内容，这部分将在本章剩下部分详述：

•有“混合”物理模式的 RDBMS(3.1.1 节)

•一个目录(3.1.2 节)

•一个数据库模式分析器(3.1.3 节)

•一个列实现器(3.1.4 节)

•一个加载器(3.2.1 节)

•一个查询重写器(3.2.2 节)

•一个(可选)原文索引(4.3 节)

为方便讨论我们将假设数据以 JSON 格式输入 Sinew。 通常，任何数据可以支持被表示为一个需要的，可选的，嵌套的和重复的字段，即使相同名字中的种类不完全相同。

3.1 存储层

3.1.1 混合模式

给定一个输入数据由文档形式包含一些潜在的嵌套键值对的集合，Sinew 自动生成一个逻辑的，面向用户的视图和独立于 RDBMS 关系数据的底层的模式来维护它。 我们先讨论逻辑视图(一个基于加载目前为止的数据创造的发展模式，不利于用户发出 SQL 查询)然后讨论它的物理实现。

在逻辑视图，Sinew 提供了一个普遍的关系，每个文档对应一行。 任何文档中的每对独一的顶层键会在通常模式中用 SQL 提供的传统获得模式作为一个列开给客户。 因此，对文档中的每一对键值对，值会被(逻辑上)储存在文档对应的行和(逻辑上)键对应的列。 如果文档包含一个嵌套对象 ,它的子键也被引用为以父对象的键之前的子键的点号分隔的名字不同的列。 与其他数据库原语如字符串和整数，嵌套对象保持可被最初的键引用。 (数组没有这么直接且将在 4.2 节讨论)

例如，给定图 2 给定的数据集，用户将有图 3 展示的视图，和下列查询：

```
SELECT url FROM webrequests WHERE hits > 20;
```

将返回所有在多于 20“hit”的对象中与键“url”的存在有联系的值。

有两种方式储存 RDBMS 逻辑模式的属性，我们将用物理和虚拟来讨论两种方式。 一个物理列是既在逻辑视图的列也在数据系统中储存为物理列。 相比之下，一个虚拟列是逻辑视图的列而在数据库系统中不存在物理列。 相反，它可以从原始键值对的序列化表示中(在 4.1 节介绍)被运行时取出。

因此，有两种极端从逻辑模式映射到物理模式： (1)将所有列储存为物理列或(2)将所有列储存为虚拟的。 在(1)中，我们有一个与上面介绍的逻辑模式相同的物理数据库模式(即一个包含对于每一个数据集合中的唯一键的列的宽表)，而在(2)，我们对每一个对象序列化(文本或二进制)有一个单独列的表并储存在该列(每行一个对象)。

所有物理列选项提供一个更简单的系统设计但能对许多、潜在的稀疏属性或那些大型嵌套对象运行问题解决数据集。 例如，行获得的 RDBMS 对每表中每行的每一个模式属性分配至少 1 位储存空间。 这个空间存留在元组头或者与其他数据在元组的身体。 这种对每个属性预分配的空间(是否属性的非空值存在)会导致储存稀疏数据变得膨胀，会显著的降低性能。

为更好理解全物理获得的问题，考虑现有版本的推特 API。这 API 规定推文有 13 个可为空的、顶层属性，当完全展开时将扩展为 23 个键。加入嵌套用户对象(可选包含推文)、标签、符号、url、用户评论和媒体后，原始推文展开后的版本会包含至多 150 个可选属性。如果我们试图在 InnoDb，一个 MySQL 受欢迎的储存引擎，保存这些表现，每条记录将达到至少 300 个字节的额外储存开销(InnoDB 标题的每项属性包含 2 个字节)。对一个最小推文(只有文字而没有额外的项目或元数据)，这个标题开销会比数据本身还大。 甚至在有高效 NULL 表达的 RDBMS(例如 Postgres，一个使用位图来表示 NULL 的存在)，有个不可忽略

的系统消耗来对稀疏系统保持追踪。 因此，作为一个实际的限制，和简化目录的设计，大部分行获得 RDBMS 占据了一个模式可能声明的列中的一个硬限制。

列获得数据库系统对宽的、稀疏的数据没有相同的储存膨胀问题且能扩展数据至比行储存更大的程度。 然而，一个完全展开的物理表示通常不是最佳的。 再次提到推特的 API，一个普遍的查询模式可能检索发送了一个给定推文的“用户”(推特嵌套一个完全的用户对象作为推文的一个属性)。 在一个完全展开的表达式，嵌套键的父对象不在明确存在。 相反，他们是合并所有属性名是所需父对象扩展的列的结果。 为了返回父对象，系统必须首先计算列的恰当集然后支付合并他们的消耗。对于频繁访问的嵌套键，最后把他们储存为单独的集合而不是独立的元素。

下面给出“全物理列”的稀疏开销方法，一种可能会去另一个极端，上述“全虚拟列”的方法。 虽然从列中取出键值的消耗包括序列化数据会保持较小(见 4.1 节和附录 B)，用一个单独列保存所有的序列化属性会降低 RDBMS 优化器生成高效查询计划的能力。 这是因为，我们规定不修改底层数据库代码，优化器不能维护属性层的数据。对优化器而言，虚拟列并不存在。

为了演示查询计划潜在的不同，我们列出了表 1 来自 1000 万条来自推特推文的查询列表。 每条推文有上述介绍的属性，稀疏度的范围从 1%到 100%。 (见第 6 节我们的实验设置)。 这个查询计划通过满足表 2 展示条件的优化器来优化。 这些计划包含不同的用于 UNIQUE 和 GROUP BY 查询的操作器，也和 JOIN 里的合取查询不同。 这些差异可以归结于这样的事实：优化器为虚拟列的查询假设一个固定的选择度(实验中 1000 万数据中的 200 行)。 因此当选择度事实上远远更低的时候()，得到的查询计划会是次优的，这可能带来巨大性能影响，取决于数据集和系统配置。 例如，当查询一个物理模式对比一个虚拟的时自合取看到一个数量级的提高，最初的 50 分钟查询时间仅在 4 分钟内完成。

为了发挥每属性一列映射的性能优势和空间高效和单列序列化的可扩展性 ,我们合并两种映射。在我们的混合模式 ,我们为一些属性创造列并将剩余储存在一个特别的序列化列，我们称为列储存所。 这允许我们在物理模式更有效的时候实现在数据库系统利用他们的好处，同时保持稀疏和至少频繁获得的键为稀疏列。

3.1.2 目录

为了保持一个逻辑模式和物理模式之间的正确的映射,促进系统的其余部分的优化,Sinew 仔细记录属性名称、类型和存储的方法(物理或虚拟列)。这个元数据保存在一个目录,信息记录如下:

•被存储的键

•已经从数据导出的键类型信息

•每个键出现的次数

•是否物理或虚拟列

• “脏”标识(3.1.4 节讨论)

事实上,这目录分为两个部分。第一部分,如图 4 中的示例所示，包含一个全局列表，其属性为出现在任何文件中的 id、名称、类型三元组。这个全局列表辅助数据序列化(在 4.1 节中描述) ,作为字典,把每个属性映射到一个 ID,从而 ,在一个特定的属性需要从存储层引用的时候，提供一个紧凑的键的表示这个属性。第二部分的目录(参见图 4(b))对每个表维持一个目录(而不是全局范围地跨表),并包含上面提到的其他信息。

根据这些信息,Sinew 能够识别的逻辑模式和当前的物理模式,使查询能够有处理指向虚拟列的引用转化为能与底层 RDBMS 模式相匹配的语句。统计(唯一的键的数目和每个键的出现次数),由模式分析程序使用(在第 3.1.3 部分中进行了描述),动态地决定对哪些列实现化

或留在列库。下面我们将讨论这种交互。

3.1.3 模式分析程序

为了适应不断变化的数据模型和查询模式,模式分析程序定期评估当前目录中定义的存储模式,以决定适当的物理和虚拟列的分布。选择哪些列进行实例化的主要目标是使得实例化的整体系统花销和相关的系统的维护表的开销最小化,使系统的性能最大化。

经常出现的属性和,带有明显不同于 RDBMS 优化器的默认假设的基数的属性是好的实例化的候选项。一个简单的阈值用于这两种情况下。出现频率高于第一阈值的属性，或带有差异高于第二阈值的基数的属性，被实例化为实体列,其余属性作为虚拟列。

作为新数据加载到系统,密度和基数的特征列可能会改变。因此,模式分析程序也会检查已经实例化的列是否仍高于阈值。如果不是,他们会被标记为去实例化。

3.1.4 列实例化程序

列实例化程序负责维护动态的物理模式，通过从列库到物理列移动数据（或者相反)。我们的目标是其作为一个后台运行进程，只有当系统中有闲置资源可用时才运行。实现这一目标关键需求是列实例化程序的渐进进程,其他查询运行时可以停止,当查询完成和资源空闲时可以恢复。因此,我们的设计不要求整体的实例化,一些键的值可能存在于列库，而另一些存在于相应的物理列。我们称这样的一个脏列,在这种情况下,在目录下的脏标识必须被设置。已经设置了脏位,查询访问键时必须检查物理列和库列(这是通过合并函数实现的，见 3.2.2 节)。

列实例化程序按如下工作。每当模式分析器决定把一个虚拟列成一个物理列，反之亦然,在数据移动之前，它把对应的脏标识设置为真。实例化程序周期性地轮询目录中标识为脏的列,对这样的列,它检查目录是否列现在应该是物理或虚拟(这决定了数据移动的方向)。然后,它逐行循环遍历和任何行,它发现列库中的数据,它应该是在一个物理列(或者相反),它对那一

行(并且只有那一行)执行一个原子更新，把值移动到正确的位置。实例化程序和加载程序不允许同时运行(我们通过目录中的锁存器实现),所以当迭代结束,它可以保证现在所有的数据都是在正确的位置。然后把脏标识设置为假,这个过程就完成了。

正如上面提到的,这个设计的重要特征是,虽然每个行更新是原子性的,整个实例化过程不是。在任何时候,实例化程序被停止并进行查询处理。由于需要添加合并函数查询处理，这些查询会略慢于对非脏列的查询。精确的减缓程度依赖于底层的数据库系统是如何实现合并的。在我们基于 PostgreSQL 的实现(见第五节),我们观察到合并查询 10%的最大减速。对于受限于磁盘带宽的工作负载，我们没有观察到减缓。

3.2 用户层

3.2.1 加载器

批量加载是在两个步骤完成,序列化和插入。在第一步中,加载程序解析每个文档,以确保它的语法有效,然后序列化成 4.1 节中描述的格式。序列化发生的时候,加载器聚合出现在数据集的关于键的形式、类型和稀疏度的信息并将这些信息添加到目录中。更确切地说,每一个要加载的键-值对,加载程序推断数据类型并在目录中查找键和类型(我们称之为一个属性的组合)，获取其属性 ID。如果属性不存在于目录,序列化器将它插入目录,获得一个新生成的 ID，并序列化键-值对，与其余的数据一起存入列库。因此,虚拟列在加载时自动创建新的键，在序列化过程中。将一个新属性添加到模式的成本只是在序列化中它第一次出现在数据集时，将新属性插入目录的成本(用户的感知不到的成本)。

在插入中,所有的序列化数据放入列库。然后 Sinew 在目录中将所有受影响的列的脏标识设置为真,当列实例化程序发现脏标识并实列化新加载的数据，它们的数据最终被转移到适当的物理列。这个设计决定是出于希望保持尽可能的模块化系统组件。总是加载到列库,

加载程序不需要知道物理模式,并且不需要与模式分析程序和系统的列实现组件交互。

3.2.2 查询重写器

Sinew 的混合存储方案需要,把面向用户、逻辑模式的查询转变与底层物理模式相匹配的查询。因此,Sinew 的查询重写器将查询发送到存储层执行前修改查询。具体地说,将给定的查询转换成一个抽象语法树,重写器使所有引用目录中的信息的列有效化。任何无效的引用列,无论是因为其引用了虚拟列或是引用了“脏”的实体列，都会被重写。例如,给定的查询:

```
SELECT url, owner
FROM webrequests
WHERE ip IS NOT NULL;
```

对虚拟列的引用,“owner,”将由一个函数转换,这个函数基于系统选择的序列化格式，从列库提取键(在 4.1 节我们提供实例实现一个这样的功能):

```
SELECT url, extract_key_txt(data, ‘owner’)
FROM webrequests
WHERE ip IS NOT NULL;
```

在“owner”是脏的(即没有完全实例化)情况下,列引用将被转换为物理列和键提取的 SQL 合并:

```
SELECT url, COALESCE(owner, extract_key_txt(data, 'owner'))
FROM webrequests
WHERE ip IS NOT NULL;
```

除了所需的键,提取函数包含了类型参数(上面是把“text”作为参数的语法糖衣),它是基于呈现在原始的语义的类型约束，由查询重写器动态地确定的查询。然后提取函数只作用于那些正确的类型的值。这种行为使 Sinew 优雅地处理情况相同键值对应于多个类型而不是抛出类型不匹配的异常(例如 :如果该值是一个预计为整数的函数参数),它将有选择性地提取整数值,对字符串、布尔值、或其他类型的值则返回 NULL。另一方面,在通常情况下,预期属性的类型不能由查询语义确定(比如 :投影的情况下),函数将返回向下转换为字符串类型的值。

第四章：改进方法

4.1

有许多种存储序列化对象数据的方法，但是大多数方法并不合适于处理通用关系数据库管理系统的操作。一种方法是使数据保持为原来的字符串格式并且存储在单个文本域。这让装载的过程变得简单（因为装载之前并不需要转换）。但是操作这些数据变的开销更大因为系统必须在执行计算前将数据转换为逻辑表达式。

另一种可选的方法是用一个像 Apache Avro 或 Google Protocol Buffers 这样使用二进制块而不是用文本来表示数据对象的序列化格式。通常情况下，两种格式都消除了句法上人类可阅读文本表示的冗余，并且提供了比在单个键之间迭代速度更快的技术方法，尤其是

通过记录存储序列号数据的模式。然而，两种格式，像 JSON，是“顺序的”，这也就意味着在这些原始数据上不支持随机读取。为了从给定的数据中取得单个一个键，应用必须要么将整个给定数据解序列化使之成为逻辑形式并然后将适当的属性解引用；要么每次读取整个序列化数据的一个属性直到获得所需的属性或者到达给定数据的尽头（如果所需的属性不存在）。

因为数据的获取（关系代数中投影）是一个对于 SQL 查询来说非常常用的操作，我们决定使用一个常规的序列化格式而不是上面说的更关注于数据转换跟平台独立性而不是分析性的两种方法。特别是当我们的格式通过允许对数据的随机读取来减少获得存储在一个序列化对象里的属性时。我们在附录 Azhong 探索了 Sinew’s format,Avro,跟 Protocol Buffers 在获取操作时的性能差别。

与一个标准的关系型数据库管理系统的元组非常类似，我们的序列化格式有一个包括对象元数据的头部跟一个包含实际数据的主体。特别地，这个头部，也就是在图 5 种所展示的结构，由一个表示记录中属性属性个数的整数，跟上一串对应于出现的键的属性 ID（用于区分类别）的有序整数组成。在属性 ID 序列之后是第二串表示每个属性在数据中的字节偏移值的整数序列。这题包含用二进制数表示的实际数据。

我们的序列化格式让 Sinew 能够通过从它的数据中提取文件结构的方式来快速确定一个键的位置或者无需读取整个序列化数据就能确定某个键不存在。当寻找一个键时，Sinew 仅仅需要确认属性 ID 的列表。当最终没有搜索到时，就可以确定这个键不存在于这个文档中。如果它找到了这个键，则会在偏移列表中找出偏移值，然后跳转到数据中正确的位置获得所需类型的值。为了将在头部中属性 ID 的二进制查找的缓存位置最大化，我们选择从偏移列表中分离键的列表（而不是讲偏移信息放在键的后面）。

键是直接获取的。每当给出一个目标键的时候，查找模式会获取对应的属性 ID 并从目

录的字典中取得类型信息。对于每一个记录，它稍后会对位于头部的属性 ID 列表进行二进制查找。并且如果 ID 存在，它会从跟在属性 ID 后的列表中获取偏移量。模式可以通过这个属性与下一个的偏移信息来计算属性的长度并且以正确的类型获取属性的值。在一个给定的数据项中进行查找的开销是 O（logN），因为它为了偏移的引用跟目标值的长度在头部进行了二进制搜索。因为这个，它会明显地笔其他前面提到过的所需开销为 O（n）的序列化格式表现出的性能要好。得益于连续存储在缓存的属性 ID，属性查找在常数系数上也有更好的表现。

值得注意的是，面向列存储的序列化格式如 RCFiles 或者 Parquet 允许随机访问并且可以在以列为对象的数据池中序列化数据。然而，当给定一个将我们的存储解决方案混合设计而成的将一些属性存储到物理列中而其他的序列化在一个列存储的数据池的设计方案时，数据池需要与物理列所取的方向适配。因此，如果 Sinew 支撑的数据库系统时列存储的，RCFiles 或者 Parquet 可能被用来取代我们常用的序列化格式。然而，如果支撑数据库系统的是行存储，一个列存储的数据格式不能与这样的数据池兼容。并且，按照我们的习惯，应该使用以对象为中心的序列化格式。

4.2

在我们的综合存储模式中，就算他们被具体化，嵌套对象跟数组还是会因为内容集合让优化控制器执行缓慢而引发执行时的阻塞。在这一节，我们演接触一些技术来提升嵌套集在物理上的表现。

对于嵌套的对象，由多种选择。尽管 Sinew 将会把所有的嵌套对象的子属性分类，也就是将他们具体化并且在需要的情况下标记他们已经被具体化，但是完全平整的数据不能达到最优的结果（在 Sec。. 3.1.1 中讨论）。因此，当 Sinew 缺省地选择单个的每行包含一个文件的表的时候（使用一个通用关系模式），系统会让模式上产生这种极端情况下的缓和。

如果有可以在在文件集中组成的逻辑组（比如嵌套对象这个例子），用户可以特别地制定这些组放入特殊的表中并且在查询时间中将他们链接起来。

在嵌套数组的情况下，用户可以根据数组句法的重要性挑选一种方法（比如无序集，有序集等等）。缺省情况下，系统将数组作为一个关系型数据库管理系统数组的数据格式来存储。但是如果数组元素的个数是有限的（并且小），系统会将每个位置单独存储为一个列。当数组中的对象是齐次时，可以让 Sinew 构成一个更加理想的无力模式，从而大大提升性能。

4.3

因为大多数文本索引的实现都包含了可以提供高性能范围查询、部分匹配跟模糊匹配的机制。所以通过对个性数据库管理系统中存储的数据增加一个外部索引，我们可以更加增强 Sinew 的性能与查询的表达性。

反向索引对虚拟列的查询十分有用。从大的方面来讲，反向文本索引表示输入数据，并将检索项编制在一起，通过使用一组包含某个项的记录相对应的 ID 值。此外，他能通过强制类型字段来提供编制其中检索组的选择。Sinew 利用了这个功能，它将字段同目录中每一个属性结合，并将虚拟列断言重写为文本索引查询。这个结果（匹配的 ID 记录的集合）可以在之后作为对原始关系的过滤。

尽管我们收录反向索引的主要目的是为了加快虚拟列上含有标准 SQL WHERE 子句断言的查询的聘雇，但是 Sinew 将反向索引用来支持整个数据集合的文本查询。用户可以搜索出现在所有列（物理或虚拟）的文本，Sinew 将会用反向索引来查找含有目标文本行的集合。

全文本查询性能不仅仅让 Sinew 可以提供一个可以更加易于表达对半结构化和数据关系断言集合，也可以让 Sinew 可以处理与此数据一起出现的完全非结构化的数据，这是通

过实验简单滴讲非结构化数据存储到一个普通的文本列中病提供全文本索引来存取。不过由于非结构话数据与关系属性无类似之处（与键和属性对应的半结构化数据不同），与这种非结构化数据相连的不是“存粹的”SQL，而是 Sinew 有一个特殊的函数，这个函数可以在 SQL 查询的 WHERE 语句中被调用，同时传入两个参数：查询够格键和字符串

一条简单的查询语句如下：

```
SELECT *
FROM webrequests
WHERE matches( ‘*’ , "full text query or regex");
```

分组工作：

汤子扬： 第 1、2 章

苏逸民：第 3 章 3.1.2 及以后

李炜铭：第 3 章 剩余

周基源：第 4 章

周晓斌：Protobuf

Google Protobuf编码技术

May 05, 2015

Google Protocol Buffer是Google於2008年開源的一款非常優秀的**序列化反序列化工具**，它最突出的特點是輕便簡介，而且有很多語言的接口（官方的支持C++，Java，Python，C，以及第三方的Erlang, Perl等）。

关于序列化的定义，摘自WikiPedia：

> *In computer science, in the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.*

个人理解：序列化是把以某种数据结构储存的信息转换成基于某种便于传输的格式组织的数据，目的是为了便于传输。反序列化是按照某种格式读取数据，再将其转换为原来的数据结构格式。既然是用于传输目的的格式，Protocol必然会对其进行数据压缩。

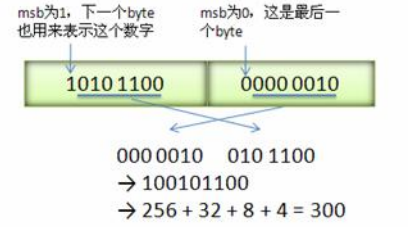
本文以int32和string为例子,探讨了Protobuf将特定结构体序列化，变为二进制数据的技术。

Varint

Varint是一种紧凑的数字表示方法，它可以用一个或多个字节表示一个数字。数字的值越小，表示需要的字节也就越少。

Varint中每个byte的最高位bit有特殊含义，0代表此字节是数据最后byte，1代表后面的byte仍是数据的一部分。每个byte剩余的7bits用于表示数据的大小。

以一个例子为例：300，以varint表示的话是两个字节： 1010 1100 0000 0010



整个过程如下：

1. 首先将300以二进制表示出来：0001 0010 1100。
2. 从后往前，每7位加上标识是否最后字节的最高bit(以红色标出)组成新的序列：0000 0010 1010 1100。标识位的值可能感到与直觉相反，但请考虑到数据的存储是依据**小端规则(little endian)**进行的，即最低字节存储在低地址单元。从数据存储的角度来看，最低字节反而是数据开始的首字节。

由此，整形300的话我们以两个字节就能够表示，无需用到四个字节或八个字节来存储。

从这里我们也可以看出，**压缩的原理是用标识位来取代了用于高字节占位的若干个0**，数据的值越小，表示需要的字节也就越少,因为值越小，用于占位的0也就越多，小于128的整数只需一个字节就能表示。很容易想到，这种压缩技术应用在值非常大的整形或者负数会有反效果，反而增加了需要表示的字节数，**因为省去的高位占位的0数目小于增加的标志位数目**,在这种情况下Protobuf有别的处理方法，这里不详述。

Protobuf正是用这种编码技术对整型数据进行压缩，附上Protocal Buffer对于int32的Varint示例代码和简单分析。

```
inline uint8* CodedOutputStream::WriteVarint32FallbackToArrayInline(
    uint32 value, uint8* target) {
  target[0] = static_cast<uint8>(value | 0x80);
  if (value >= (1 << 7)) {
    target[1] = static_cast<uint8>((value >>  7) | 0x80);
    if (value >= (1 << 14)) {
      target[2] = static_cast<uint8>((value >> 14) | 0x80);
      if (value >= (1 << 21)) {
        target[3] = static_cast<uint8>((value >> 21) | 0x80);
        if (value >= (1 << 28)) {
          target[4] = static_cast<uint8>(value >> 28);
          return target + 5;
        } else {
          target[3] &= 0x7F;
          return target + 4;
        }
      } else {
        target[2] &= 0x7F;
        return target + 3;
      }
    } else {
      target[1] &= 0x7F;
      return target + 2;
    }
  } else {
    target[0] &= 0x7F;
    return target + 1;
  }
}
```

具体的实现是以一个 `uint8` 的数组 `target` 存储每一个字节，每次使适当右移后(使特定的某7位成为低7位)的值的低七位与1000 0000按位或(最高位1表示默认不是最后一个字节)，存储结果到 `target` ，最后使最后一个字节与0111 1111按位与，置最高位为0表结束，最后返回指向最后一个字节的指针 `target` 。

Protocal Buffer Message

在PB中，有一种proto类型的文件，要求用户自定义信息逻辑单元*Message*，即一个结构体。下面举例说明一个简单的Hello.proto文件，。

```
message Hello {
    required int32 id = 1;
    required float flt = 2;
    required string str = 3;
    optional int32 opt = 4;
}
```

首先每个结构体需要用关键字*Message*描述，其中每个字段的修饰符有 `required` ， `repeated` ， `optional` 三种，required表示該字段是必須的，repeated表示該字段可以重复出現，它描述的字段可以看做C語言中的數組，optional表示該字段可有可無。

同时，必须人为地赋予每一个字段一个标号**field_number**，如图中的1,2,3,4所示。

Protocol Buffer的数据类型

Protocol Buffer需要用户自定义自己的数据体，而且结构体的定义要符合google制定的规则。结构体中每个字段都需要一个数据类型，Protocol Buffer支持的数据类型在wire_format_lite.h中定义：

```
 enum WireType {
    WIRETYPE_VARINT           = 0,
    WIRETYPE_FIXED64          = 1,
    WIRETYPE_LENGTH_DELIMITED = 2,
    WIRETYPE_START_GROUP      = 3,
    WIRETYPE_END_GROUP        = 4,
    WIRETYPE_FIXED32          = 5,
  };
```

其中：

VARINT类数据表示要用variant编码对所传入的数据做压缩存储，variant编码细节见上文。

FIXED32和FIXED64类数据不对用户传入的数据做variant压缩存储，只存储原始数据。

LENGTH_DELIMITED类数据主要针对string类型、repeated类型和嵌套类型，对这些类型编码时需要存储他们的长度信息。

START_GROUP是一个组（该组可以是嵌套类型，也可以是repeated类型）的开始标志。

END_GROUP是一个组（该组可以是嵌套类型，也可以是repeated类型）的结束标志。

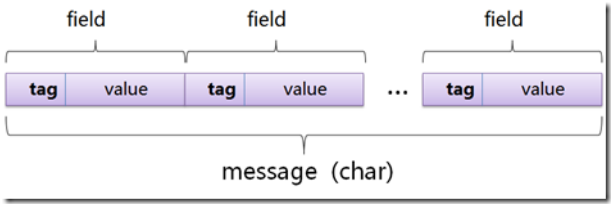
每类数据包含的具体数据类型如下表所示：

WireType	表示类型
VARINT	int32,int64,uint32,uint64,sint32,sint64,bool,enum
FIXED64	fixed64,sfixed64,double
LENGTH_DELIMITED	string,bytes,embedded messages, packed repeadted field
START_GROUP	group的开始标志
END_GROUP	group的结束标志
FIXED32	fixed32,sfixed32,float

Protocal Buffer的编码

有了上述预备知识后，我们开始正式研究PB的编码。**Protobuf的编码尽其所能地将字段的元信息和字段的值压缩存储，并且字段的元信息中含有对这个字段描述的所有信息。**

整個结构体序列化后，如下图：



可以看到，整個消息是以二进制流的方式存储，在這個二进制流中，逐個字段以定义的顺序紧紧相邻。每個字段中由标签tag和字段的值value組成。

其中tag是這样编码的：

1. field_number << 3 | wire_type(wire_type共有6种，因此需要低三位来表示，tag描述了字段的元信息)
2. 对上面得到的无符号类型整數做varint编码

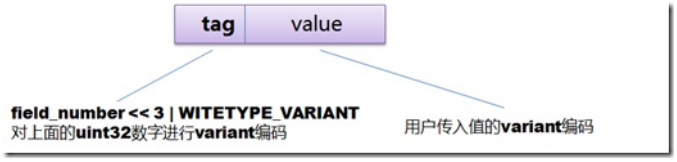
其中field_number是定义*Message*时，赋予每个字段的标号，wire_type是字段的数据类型。

下面以int32和string为例，看看PB压缩编码的过程。

---

int32类型的编码

1. 计算字段长度 1 + Int32Size(value)。
2. 调用WriteString()写入标签与压缩后的值。



计算字段长度 `1 + Int32Size(value)`;

以下代码片截取自 `Hello.pb.cc`

```
  // required int32 id = 1;
   if (has_id()) {
     total_size += 1 +
       ::google::protobuf::internal::WireFormatLite::Int32Size(
         this->id());
   }
```

调用了 `Int32Size()`,定义如下：

```
inline int WireFormatLite::Int32Size(int32 value) {
  return io::CodedOutputStream::VarintSize32SignExtended(value);
}
```

里面嵌套调用了 `VarintSize32SignExtended(value)` ，声明如下：

```
  // If negative, 10 bytes.  Otherwise, same as VarintSize32().
  static int VarintSize32SignExtended(int32 value);
```

`VarintSize32()`声明如下：

```
  // Returns the number of bytes needed to encode the given value as a varint.
  static int VarintSize32(uint32 value);
```

函数功能已很清晰，返回varint压缩value后的字节数(实现代码已在上文贴出)。

调用 `WriteInt32()`将该字段的标签和字段值写入新空间中：

```
// @@protoc_insertion_point(serialize_to_array_start:Hello)
  // required int32 id = 1;
  if (has_id()) {
    target = ::google::protobuf::internal::WireFormatLite::WriteInt32ToArray(1, this->id(), target);
  }
```

`WriteInt32ToArray()`定义如下

```
inline uint8* WireFormatLite::WriteInt32ToArray(int field_number,
                                                int32 value,
                                                uint8* target) {
  target = WriteTagToArray(field_number, WIRETYPE_VARINT, target);
  return WriteInt32NoTagToArray(value, target);
}
```

```
inline uint8* WireFormatLite::WriteTagToArray(int field_number,
                                              WireType type,
                                              uint8* target) {
  return io::CodedOutputStream::WriteTagToArray(MakeTag(field_number, type),
                                                target);
}
```

里面`MakeTag`实际调用了一个Macro，相关代码如下

```
#define GOOGLE_PROTOBUF_WIRE_FORMAT_MAKE_TAG(FIELD_NUMBER, TYPE)                \
  static_cast<uint32>(                                                          \
    ((FIELD_NUMBER) << ::google::protobuf::internal::WireFormatLite::kTagTypeBits) \
      | (TYPE))
```

```
  // Number of bits in a tag which identify the wire type.
  static const int kTagTypeBits = 3;
```

然后我们得到了`field\_number \<\< 3 \| wire\_type`这个标签，接着调用`WriteTagToArray()`把标签写入`target`指针指向的空间中。

```
inline uint8* CodedOutputStream::WriteTagToArray(
    uint32 value, uint8* target) {
  if (value < (1 << 7)) {
    target[0] = value;
    return target + 1;
  } else if (value < (1 << 14)) {
    target[0] = static_cast<uint8>(value | 0x80);
    target[1] = static_cast<uint8>(value >> 7);
    return target + 2;
  } else {
    return WriteVarint32FallbackToArray(value, target);
  }
}
```

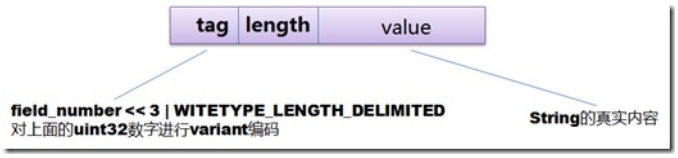
最后调用`WriteInt32NoTagToArray()`把varint(value)也写入`target`指向的空间中，整个编码工作就完结了。

```
void CodedOutputStream::WriteVarint32(uint32 value) {
  if (buffer_size_ >= kMaxVarint32Bytes) {
    // Fast path:  We have enough bytes left in the buffer to guarantee that
    // this write won't cross the end, so we can skip the checks.
    uint8* target = buffer_;
    uint8* end = WriteVarint32FallbackToArrayInline(value, target);
    int size = end - target;
    Advance(size);
  } else {
    // Slow path:  This write might cross the end of the buffer, so we
    // compose the bytes first then use WriteRaw().
    uint8 bytes[kMaxVarint32Bytes];
    int size = 0;
    while (value > 0x7F) {
      bytes[size++] = (static_cast<uint8>(value) & 0x7F) | 0x80;
      value >>= 7;
    }
    bytes[size++] = static_cast<uint8>(value) & 0x7F;
    WriteRaw(bytes, size);
  }
}
```

---

string类型

1. 计算长度 1 + varint(stringLength) + stringLength。
2. 调用WriteInt32()将标签、长度和原串写入内存。



与int32一样，先计算长度，`1 + varint(stringLength) + stringLength`从这里可以看出，PB对String的值不作压缩，因此最终字符串长度比原串要更长。

```
  // required string str = 3;
  if (has_str()) {
    total_size += 1 +
      ::google::protobuf::internal::WireFormatLite::StringSize(
        this->str());
  }
```

计算Varint(字符串长度)与字符串长度。与Int32相比，代码简单不少。

```
inline int WireFormatLite::StringSize(const string& value) {
  return io::CodedOutputStream::VarintSize32(value.size()) +
         value.size();
}
```

```
  // Returns the number of bytes needed to encode the given value as a varint.
  static int VarintSize32(uint32 value);
```

调用`WriteString()`将该字段的标签、长度和值写入内存中，完成string的编码。

```
void WireFormatLite::WriteString(int field_number, const string& value,
                                 io::CodedOutputStream* output) {
  // String is for UTF-8 text only
  WriteTag(field_number, WIRETYPE_LENGTH_DELIMITED, output);
  GOOGLE_CHECK(value.size() <= kint32max);
  output->WriteVarint32(value.size());
  output->WriteString(value);
}
```

这三个函数调用过于冗长，这里不做展开，原理与Int32相近。

```
inline void WireFormatLite::WriteTag(int field_number, WireType type,
                                     io::CodedOutputStream* output) {
  output->WriteTag(MakeTag(field_number, type));
}
```

```
void CodedOutputStream::WriteVarint32(uint32 value) {
  if (buffer_size_ >= kMaxVarint32Bytes) {
    // Fast path:  We have enough bytes left in the buffer to guarantee that
    // this write won't cross the end, so we can skip the checks.
    uint8* target = buffer_;
    uint8* end = WriteVarint32FallbackToArrayInline(value, target);
    int size = end - target;
    Advance(size);
  } else {
    // Slow path:  This write might cross the end of the buffer, so we
    // compose the bytes first then use WriteRaw().
    uint8 bytes[kMaxVarint32Bytes];
    int size = 0;
    while (value > 0x7F) {
      bytes[size++] = (static_cast<uint8>(value) & 0x7F) | 0x80;
      value >>= 7;
    }
    bytes[size++] = static_cast<uint8>(value) & 0x7F;
    WriteRaw(bytes, size);
  }
}
```

```
inline void CodedOutputStream::WriteString(const string& str) {
  WriteRaw(str.data(), static_cast<int>(str.size()));
}
```

Reference

1. https://developers.google.com/protocol-buffers/?hl=zh-cn—Protobuf首页
2. http://www.cnblogs.com/cobbliu/archive/2013/03/02/2940074.html —博客園CobbLiu