[存储和处理时间序列数据](https://segmentfault.com/a/1190000002797644)

就像我们在前一章提到的，一个时间序列是一系列数值，每个数值都伴随着一个时间值，代表数据被记录时的时间。时间序列数据存入后就很少再需要修改了，查询时经常是查询一个连续时间段的数据，也可能查询汇总或者聚合后的数据。时间序列数据库是一种储存多个时间序列的方式，在其中检索一个或几个时间序列的某一个特定时间段的数据是特别高效的。同样地，主要用来查询一个时间段数据的应用程序也适合使用时间序列数据库来实现。像之前所解释的，本书的主题是存储和处理大规模时间序列数据，为了实现这个目标，首选技术是非关系型NoSQL数据库，比如Apache HBase或MapR-DB。

为大规模时间序列数据库的实际实现提供务实的建议，是本书的目标，所以我们需要聚焦于一些可以简化和增强真实世界中应用程序发展进程的一些基本步骤。我们会简单看看适用于中小型数据集的方法，然后深入探究我们主要关注的问题：如何实现大规模TSDB。

为了得到一个扎实的实现，有几种可供选择的设计方法。如何选择取决于数据的属性。有多少种不同的时间序列？获得的数据是什么类型的？使用怎样的速度采集数据？需要存储多久的数据？这些问题的答案有助于我们确定最优的实现策略。

尽管我们已经提到处理时间序列数据的一些主要方面，这一章会比之前更深入地探讨存储和访问时间序列数据的基本方法。第四章会提供如何使用现有开源软件来最好地实现这些概念的建议。这两章有比较多的内容需要理解。然后你就可以记住如果将这些关键的想法集中到一起而不是迷失在细节中，这里是一个本章内容的一个简单路线图：

1) 平面文件

对时间序列数据来说是受限的工具，不适合快速增长的数据，查询起来也会效率低下

2) 真正的数据库：关系型数据库

扩展性不好，常见的星型模式（star schema）不适合处理时间序列数据

3) 真正的数据库：非关系型NoSQL数据库

首选方案，因为它可扩展型好、高效、能快速响应基于时间段的查询

基本设计：使用包含时间序列ID的唯一row key，列是不同时间偏移的数值；存储多于一个时间序列

可选设计：使用宽表逐点存储数据；混合宽表和blob类型的设计；将数据缓存到内存，然后blob直写

我们已经回顾了主要思想，现在我们更详细地重温一下并且解释它们的重要性。

最简单的数据存储：平面文件

你可以扩展非常简单的设计（比如简单的二维表），使用更聪明的文件格式来使其更先进，比如列存储的Parquet格式。Parquet是一个有效并且简单的现代化格式，可以存储时间和一些可选值。图3-1展示了两种记录时间序列数据的Parquet schema。左图中的schema适合你已经知道怎么合理使用时间序列数据的情况，它是一个特定场景的存储方案。例子中，只存了明确指定的4个时间序列的数据（一个存放时间的t和一个存放数据的tempIn组合起来，为一个时间序列。t和它对应的tempIn、pressureIn、tempOut、pressureOut即4个时间序列），如果需要增加新的，就需要修改schema。右图中的Parguet schema抽象程度更高，对你想要往文件里嵌入更多元数据的场景更适合。并且这种格式没有事先对时间序列的数量做任何限制。如果你想要构建一个给其他人使用的时间序列库，右边的格式会更合适一些。

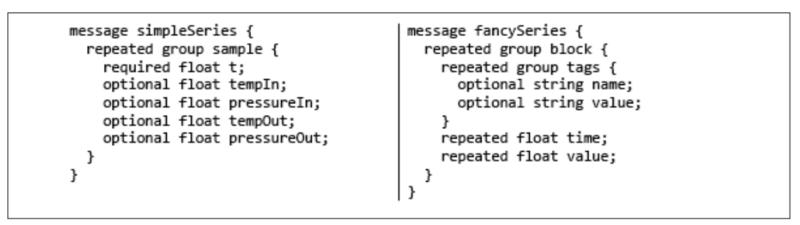


图3-1

使用Parquet格式来存储时间序列数据的两种可能的schema。左边的schema使用固定的类型名称将问题域确定了。在不改变schema的情况下，只可以存储4个时间序列。相反地，右边的schema更加灵活，你可以增加新的时间序列。另外它的抽象层次也更高，把几个单一的时间序列（一对time、value）按照tags分组，然后放到一个单独的block中。

这样的一种时间序列数据（特别是使用类似Parquet格式的情况）是非常有用的，但前提是你需要分析的时间序列数量相对较小，并且所感兴趣的时间范围相对于单个文件所存储数据的时间跨度很大（比如每个文件存放一个月的数据，你查的时候也应该每次查一个月的数据，而不是每次查一天的）。

系统最初使用平面文件来实现是一种非常普遍的情况，而且不久之后这种简单的实现不再适应快速增长的数据的情况也是很普遍的。基本问题是单一文件中的时间序列数量增加了，任何特定的查询中，真正有用的数据占所读取数据的比例就下降了，因为多数读取到的数据其实是属于其他时间序列的。

同样地，在文件中的时间跨度比平均查询的时间范围已经长很多的情况，真正有用的数据占所被读取数据的比例又下降了，因为文件中的大部分数据已经在你感兴趣的时间范围之外了（比如数据记录了1个月的数据，而查询时一般只查某一天的，那为了定位到这一天，需要先读大量前边的实际不需要的数据）。努力解决这些问题的同时一般又会引入其他的问题。使用大量的文件来确保每个文件中只有较少的时间序列，会使文件数量大幅增长。同样地，减少每个文件所存储数据的时间范围会使得文件数量翻倍增长。当在一个类似Apache Hadoop中HDFS的文件系统存储数据时，大量的文件会导致严重的稳定性问题。基于Hadoop的上层系统，如MapR可以轻松处理大量的文件，但检索和管理大量的小文件也是很低效的，因为需要增加很多搜索时间。

为了避免这些问题，很自然的一步是转而使用某些形式的真正的数据库来存储这些数据。选择合适的数据库的方法并不是显而易见的，但基于数据库的类型和它的设计方案，你有几个选项。我们会研究这些问题来帮助你作选择。

改用真正的数据库：RDBMS怎么样？

即使是经过良好分区的平面文件，在处理大规模时间序列数据时也会力不从心，所以你也行会考虑使用某些类型的真正的数据库。当第一次在数据库中存储时间序列数据时，使用所谓的星型模式（star schema）设计，并且将数据存放到RDBMS是个很诱人的选择。在这样一种数据库设计中，核心数据存放在事实表（fact table），就像图3-2展示的那样。

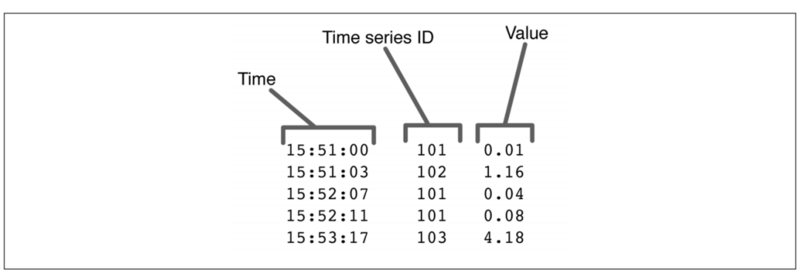


图3-2

将时间序列数据存放到RDBMS的一个事实表的设计。其中存放了时间（TIme列）、序列ID（Time series ID列）和数值（Value列）三列。序列的细节存放在维表（dimension table）中（这一对Time、Value是一个时间序列，但这个时间序列的细节，比如Value的含义是什么，存放在另一张表中，可以使用Time series ID去那个表查）。

在星型模式中，一个表存储主要的数据，并且会引用其他表（维表）。该设计一个核心假定是维表要相对小巧，而且不常变动。图3-2中的时间序列事实表里，唯一被引用的维表，就是存放这个时间序列详细信息的维表，它的内容是表中数据（Value列）的含义。比如，如果我们的时间序列数据是从一个工厂的泵或者其他设备从采集的，我们会希望在获取这个泵的多个维度的数据，如入口和出口的压强和温度、泵在不同频段的震动和泵自身的温度等。这其中的每个泵的每一个维度，都是一个单独的时间序列，每个时间序列会有类似泵的序列号、位置、商标、型号等信息，这些信息都存放在维表中。

实际上一些应用程序已经使用像这样的星型模式来存放时间序列数据了。我们在多数NoSQL数据库中也可以使用这样的设计。星型模式解决了有大量不同时间序列的问题，在数据点的规模达到数亿甚至数十亿的情况下也可以工作得很好。然而就像我们在第一章中看到的，即使是19世纪的航运数据也会产生上十亿的数据点。在2014年，纳斯达克证券交易所在过去三个月就会处理十亿规模的交易量。记录一个中型计算机集群的运行环境的话，一天会产生五亿的数据点。

并且简单地将这些数据存储起来是一回事，对其检索和处理就是另一回事了。现代的应用程序如机器学习系统甚至状态显示系统都需要每秒检索和处理上百万的数据点。

虽然RDBMS可以扩展到这些大小、速度需求的下限，但带来的消耗和引入的复杂性会急剧上升。随着数据规模的继续增长，基于RDBMS的应用程序越来越不适合处理这样规模的时间序列数据了。使用星型模式但转而使用NoSQL数据库的话，也没有特别的帮助，因为这个问题的核心是星型模式带来的，而不只是数据量。

使用宽表的NoSQL数据库

星型模式所触及的核心问题是每次测量都要使用一行。一个增加时间序列数据库中数据检索速度的技术是在每一行存储很多数值。在一些像Apache HBase或者MapR-DB的NoSQL数据库中，列的数量几乎是不受限制的，只要任何特定一行中有数据的列的数量在几十万之内。这种能力可以被用来在每行存放多个数值。这样做的话，数据点就可以被更高速地检索，因为扫描数据的最大速度部分取决于需要扫描的行的数量，部分取决于待检索数据点的总数，部分取决于待检索数据的总量。减少行的数量，就大幅减少了一部分检索开销，检索速度就提升了。图3-3展示了使用宽表来减少时间序列数据行数量的一种方式。这个技术和OpenTSDB（一个开源的数据库，我们会在第四章详细讲到）之中使用的默认表结构很相似。需要注意这样的表设计，和那些需要提前定义详细schema的系统的表设计是很不一样的。有一件事情，如果你想把schema写下来，那将异常庞大。

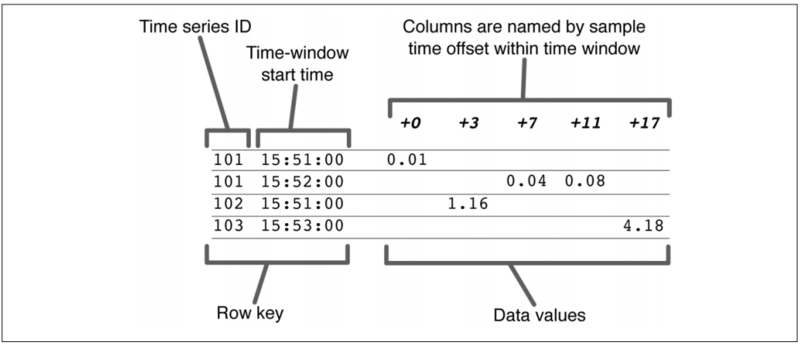


图3-3

在NoSQL时间序列数据库中一个宽表的使用。关键的结构是直观的，在真正的应用程序中，使用的会是一个二进制的格式，但这样顺序的属性是一样的。

因为HBase和MapR-DB都是按照主键的顺序来存储数据，图3-3中的键设计会导致每行包含一小段时间的数据在磁盘上是连续存储的（因为Row key是按时间顺序增长，HBase和MapR-DB是按列族存放数据的，Data values中的数据就会全部按照时间顺序存放在磁盘上）。这个设计意味着检索一个特定时间段的数据，涉及的主要是顺序磁盘操作，就会比数据按行分散开的情况快很多。为了从这个表结构获得性能优势，每个时间窗口的采样点要足够富裕，这样就可以减少行的数量，从而提升检索速度。典型情况，时间窗口会被调整成每一行包括100-1000采样点的样子。

混合模式设计的NoSQL数据库

图3-3中的表设计可以继续改进，通过将一行中的所有数据压缩成一个单一的被称作blob的数据结构。Blob可以高度压缩，所以需要从磁盘读取的数据量就更少了。并且，如果使用HBase来存储时间序列数据，每行只有一列的情况会减少了每列数据在HBase所使用的磁盘文件格式上的开销，这样又进一步提高了性能。图3-4的混合式表结构中，一些行的数据已经被压缩，另一些行没有。

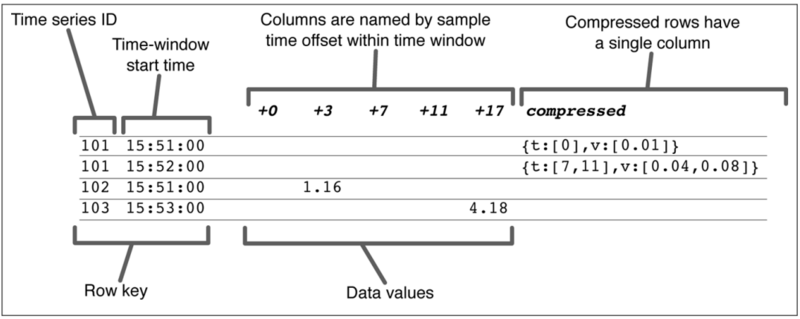


图3-4

在混合模式设计中，行中的数据可以被存储成一个单一的数据结构（blob）。注意实际压缩的数据更可能是二进制的格式。这里使用JSON格式显示是为了更容易理解。

图3-3中的宽表格式可以进化成图3-4的压缩格式（blob样式），只要确保那些被压缩的行对应的时间窗口不会或者很少再有新增的数据。一般地，一旦时间窗口结束后，新的数据就不属于这个时间窗口了，然后对这个时间窗口中数据的压缩就可以开始了。因为在同一行中，已压缩和未压缩的数据可以共存，如果在对行压缩之后，又有新数据过来了，可以再简单地重新压缩这一行，将新数据合并进来。

图3-5展示的是概念上的混合式时间序列数据库的数据流。

在后台将数据从旧格式转换成blob格式，会让renderer（图3-5中所显示的）检索数据并绘制出来的速度有质的提升。例如，在4个节点的MapR集群中，数据以压缩格式存放的话，3千万的数据点可以在大概20秒内被检索、聚合、绘制出来。

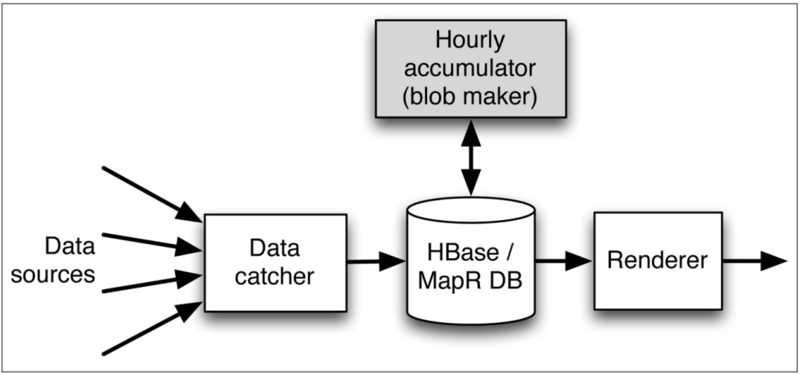


图3-5

混合式时间序列数据库的数据流。数据从数据源到达catcher，然后被插入到NoSQL数据库中。之后blob maker在后台定时将数据压缩成blob格式。数据由renderer检索和格式化。

再进一步：blob直写设计

压缩旧数据依然存在一个性能瓶颈。因为数据以未压缩的格式插入进来，每个数据点到来后都需要对行做一个更新来将数值插入到数据库中。对行的更新操作会限制数据的插入速度到每个集群中的每个节点上只有每秒2万个数据点。

另一方面，图3-6中的blob直写方式的数据流允许插入速度增加了大概1千倍。为什么blob直写方式会带来如此大的性能提升？基本的区别是blob maker被转移到catcher和NoSQL时间序列数据库之间了。使用这种方式，blob maker就可以从内存的数据缓存中直接读取输入的数据，而不是从存储层的宽表中提取之前已经被写入进去的数据。

基本的思想是数据到达后先被存放在内存中。这些数据同时也被写入到日志文件中。这些日志文件就是图3-6中的restart logs，它们是在Hadoop系统存放的平面文件，不是存储层的一部分。Restart logs允许内存中的数据缓存被重新导入，在数据管道必须被重建的时候。

在正常操作中，在时间窗口的末尾，新的内存中数据结构会被创建，现在旧的内存中数据结构就可以用来创建压缩的blob然后写入数据库了。一旦blob被写入了，日志文件就被删除了。这样就无需像之前的混合设计中将数据两次写入。在图3-5中的混合设计中，全部的输入数据流都会逐点写入到存储层，然后再被blob maker读取。读的情况和写大致一样。一旦数据被压缩成了blob，它又被写入到数据库中。相反地，在图3-6的blob直写设计的数据流中，完整的数据流只写入到内存中（这样速度很快），而不是写入到数据库中。数据在压缩成blob之前不会被写入到数据库，所以写入速度大幅提升。数据库操作的次数从之前数据点的数量变成了blob的数量，很容易将次数减少到之前几千分之一这样的量级。

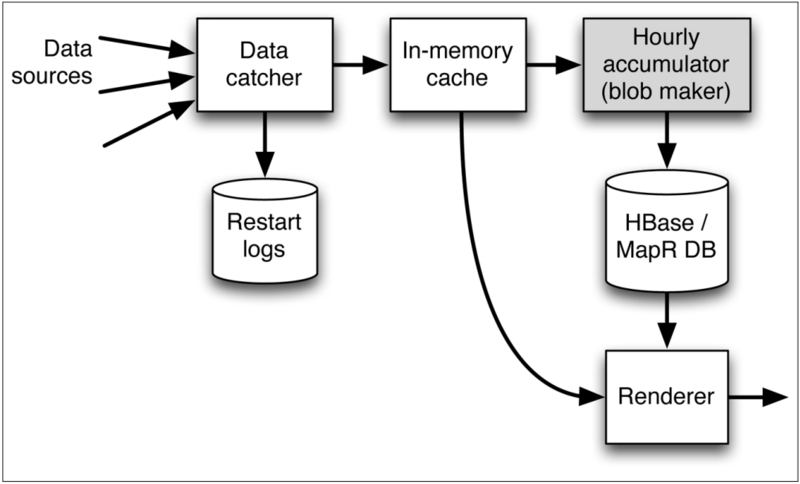


图3-6

Blob直写方式的数据流。Catcher在内存中暂存数据，并且将其写入到restart logs中。Blob maker周期地从缓存中读取数据，然后将压缩成的blob写入到数据库中。这个设计的性能提升来自于renderer可以同时从内存和数据库中获取数据。

blob直写方式的优势是什么？一个真实世界的例子展示了它可以做什么。使用了这个架构，仅使用了一个10节点的MapR集群中的4个节点，就可以实现每秒往MapR-DB的表中插入超过一亿的数据点。这些节点都有着很高的性能，其中每个节点有15个CPU核、大量内存和12块高配置磁盘，但你使用多数硬件都可以达到这个性能级别的1/5到1/2。

这个性能级别听起来是用来处理海量数据的，可能超出了我们所需要的处理能力，但是在第五章我们会展示为什么这样的性能是非常有用的，即使是对那些相对温和的应用程序。

为什么关系型数据库不是很合适

在这一点，询问为什么一个关系型数据库不能处理和使用混合模式的MapR-DB或者HBase所能承受的插入和分析数据的负载是公平的。当只有blob数据被插入而不使用宽表的情况，这个问题特别有趣，因为现代关系型数据库通常支持blob或者array类型。

这个问题的答案是，关系型数据库主要解决的问题不是提高插入和检索数据的速度，它现在这样运行是有其合理性的。使用关系型数据库的主要原因也不是因为它有更好的性能。如果使用关系系型数据库的blob格式存储数据，就意味着需要放弃大多数其他好处。此外，SQL没有提供一个好的抽象方法，来隐藏访问blob格式数据中的细节。SQL不能用任何合理的方式来访问这些数据，并且像多行事务等特性也完全派不上用场了。事务在这里还会成为问题，因为即使不使用，它也会成为一种消耗。一个关系型数据库需要满足多行事务的需求，这使它更难被扩展到多个节点上。尽管使用如Oracle的高成本数据库可以在单个节点实现很高的性能。而使用类似Apache Hbase或者MapR-DB的NoSQL数据库，你可以简单地通过加硬件的方式实现更高的性能。

为自己用不到的特性买单的模式在一些高性能系统中是存在的。为了可扩展而牺牲传统关系型数据库的一些固有特性也是常见的，但即使你这样做了，还是得不到自己想要的扩展性。在这种情况，使用类似HBase或者MapR-DB的替代方案是有实质上的好处的，因为你同时得到了性能和可扩展性。

混合模式设计：我可以从哪得到一个？

这些宽表、blob混合的表设计是非常诱人的。它们所许诺的巨大性能级别令人兴奋，而且它们能运行在有容错机制、基于Hadoop的系统（比如MapR），从运维的角度看也是很吸引人的。这些新方法都不是空想，它们已经被构建出来，并且被证明有着惊人的结果。然而我们在这里呈现的，很大程度都是概念上的东西。有真正已经实现的吗？下一章我们会讲到如何使用OpenTSDB（一个开源时间序列数据库工具）和几个开源的MapR扩展，来实现这些新的设计。结果是利用本章所描述的概念以达到高性能的时间序列数据库是现代使用场景所需要的。

# Storing and Processing Time Series Data

As we mentioned in previous chapters, a time series is a sequence of values, each with a time value indicating when the value was recorded. Time series data entries are rarely amended, and time series data is often retrieved by reading a contiguous sequence of samples, possibly after summarizing or aggregating the retrieved samples as they are retrieved. A time series database is a way to store multiple time series such that queries to retrieve data from one or a few time series for a particular time range are particularly efficient. As such, applications for which time range queries predominate are often good candidates for implementation using a time series database. As previously explained, the main topic of this book is the storage and processing of large-scale time series data, and for this purpose, the preferred technologies are NoSQL non-relational databases such as Apache HBase or MapR-DB.

Pragmatic advice for practical implementations of large-scale time series databases is the goal of this book, so we need to focus in on some basic steps that simplify and strengthen the process for realworld applications. We will look briefly at approaches that may be useful for small or medium-sized datasets and then delve more deeply into our main concern: how to implement large-scale TSDBs. To get to a solid implementation, there are a number of design decisions to make. The drivers for these decisions are the parameters that define the data. How many distinct time series are there? What kind of data is being acquired? At what rate is the data being acquired? For how long must the data be kept? The answers to these questions help determine the best implementation strategy.

Although we’ve already mentioned some central aspects to handling time series data, the current chapter goes into the most important ideas underlying methods to store and access time series in more detail and more deeply than previously. Chapter 4 then provides tips for how best to implement these concepts using existing open source software. There’s a lot to absorb in these two chapters. So that you can better keep in mind how the key ideas fit together without getting lost in the details, here’s a brief roadmap of this chapter:

1) Flat files Limited utility for time series; data will outgrow them, and access is inefficient.

2) True database: relational (RDBMS) Will not scale well; familiar star schema inappropriate.

3) True database: NoSQL non-relational database

Preferred because it scales well; efficient and rapid queries based on time range

Basic design Unique row keys with time series IDs; column is a time offset Stores more than one time series

Design choices Wide table stores data point-by-point Hybrid design mixes wide table and blob styles Direct blob insertion from memory cache

Now that we’ve walked through the main ideas, let’s revisit them in some detail to explain their significance.

## Simplest Data Store: Flat Files

You can extend this very simple design a bit to something slightly more advanced by using a more clever file format, such as the columnar file format Parquet, for organization. Parquet is an effective and simple, modern format that can store the time and a number of optional values. Figure 3-1 shows two possible Parquet schemas for recording time series. The schema on the left is suitable for special-purpose storage of time series data where you know what measurements are plausible. In the example on the left, only the four time series that are explicitly shown can be stored (tempIn, pressureIn, tempOut, pressureOut). Adding another time series would require changing the schema. The more abstract Parquet schema on the right in Figure 3-1 is much better for cases where you may want to embed more metadata about the time series into the data file itself. Also, there is no a priori limit on the number or names of different time series that can be stored in this format. The format on the right would be much more appropriate if you were building a time series library for use by other people.

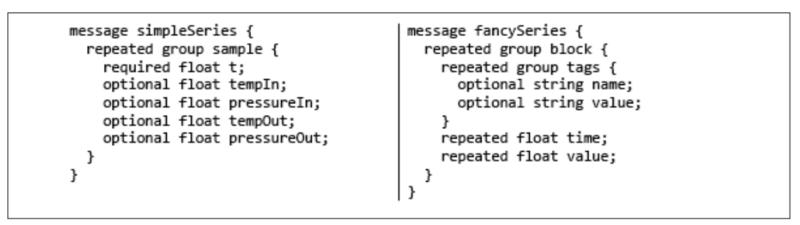


Figure 3-1

Such a simple implementation of a time series—especially if you use a file format like Parquet—can be remarkably serviceable as long as the number of time series being analyzed is relatively small and as long as the time ranges of interest are large with respect to the partitioning time for the flat files holding the data. While it is fairly common for systems to start out with a flat file implementation, it is also common for the system to outgrow such a simple implementation before long. The basic problem is that as the number of time series in a single file increases, the fraction of usable data for any particular query decreases, because most of the data being read belongs to other time series. Likewise, when the partition time is long with respect to the average query, the fraction of usable data decreases again since most of the data in a file is outside the time range of interest. Efforts to remedy these problems typically lead to other problems. Using lots of files to keep the number of series per file small multiplies the number of files. Likewise, shortening the partition time will multiply the number of files as well. When storing data on a system such as Apache Hadoop using HDFS, having a large number of files can cause serious stability problems. Advanced Hadoop-based systems like MapR can easily handle the number of files involved, but retrieving and managing large numbers of very small files can be inefficient due to the increased seek time required. To avoid these problems, a natural step is to move to some form of a real database to store the data. The best way to do this is not entirely obvious, however, as you have several choices about the type of database and its design. We will examine the issues to help you decide.

## Moving Up to a Real Database: But Will RDBMS Suffice?

Even well-partitioned flat files will fail you in handling your large-scale time series data, so you will want to consider some type of true database. When first storing time series data in a database, it is tempting to use a so-called star schema design and to store the data in a relational database (RDBMS). In such a database design, the core data is stored in a fact table that looks something like what is shown in Figure 3-2.

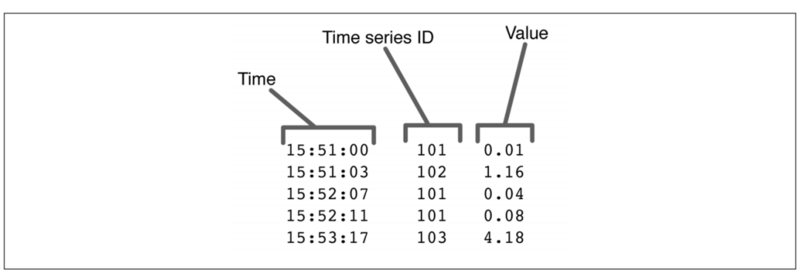


Figure 3-2

In a star schema, one table stores most of the data with references to other tables known as dimensions. A core design assumption is that the dimension tables are relatively small and unchanging. In the time series fact table shown in Figure 3-2, the only dimension being referenced is the one that gives the details about the time series themselves, including what measured the value being stored. For instance, if our time series is coming from a factory with pumps and other equipment, we might expect that several values would be measured on each pump such as inlet and outlet pressures and temperatures, pump vibration in different frequency bands, and pump temperature. Each of these measurements for each pump would constitute a separate time series, and each time series would have information such as the pump serial number, location, brand, model number, and so on stored in a dimension table. A star schema design like this is actually used to store time series in some applications. We can also use a design like this in most NoSQL databases as well. A star schema addresses the problem of having lots of different time series and can work reasonably well up to levels of hundreds of millions or billions of data points. As we saw in Chapter 1, however, even 19th century shipping data produced roughly a billion data points. As of 2014, the NASDAQ stock exchange handles a billion trades in just over three months. Recording the operating conditions on a moderate-sized cluster of computers can produce half a billion data points in a day. Moreover, simply storing the data is one thing; retrieving it and processing it is quite another. Modern applications such as machine learning systems or even status displays may need to retrieve and process as many as a million data points in a second or more. While relational systems can scale into the lower end of these size and speed ranges, the costs and complexity involved grows very fast. As data scales continue to grow, a larger and larger percentage of time series applications just don’t fit very well into relational databases. Using the star schema but changing to a NoSQL database doesn’t particularly help, either, because the core of the problem is in the use of a star schema in the first place, not just the amount of data.

## NoSQL Database with Wide Tables

The core problem with the star schema approach is that it uses one row per measurement. One technique for increasing the rate at which data can be retrieved from a time series database is to store many values in each row. With some NoSQL databases such as Apache HBase or MapR-DB, the number of columns in a database is nearly unbounded as long as the number of columns with active data in any particular row is kept to a few hundred thousand. This capability can be exploited to store multiple values per row. Doing this allows data points to be retrieved at a higher speed because the maximum rate at which data can be scanned is partially dependent on the number of rows scanned, partially on the total number of values retrieved, and partially on the total volume of data retrieved. By decreasing the number of rows, that part of the retrieval overhead is substantially cut down, and retrieval rate is increased. Figure 3-3 shows one way of using wide tables to decrease the number of rows used to store time series data. This technique is similar to the default table structure used in OpenTSDB, an open source database that will be described in more detail in Chapter 4. Note that such a table design is very different from one that you might expect to use in a system that requires a detailed schema be defined ahead of time. For one thing, the number of possible columns is absurdly large if you need to actually write down the schema.

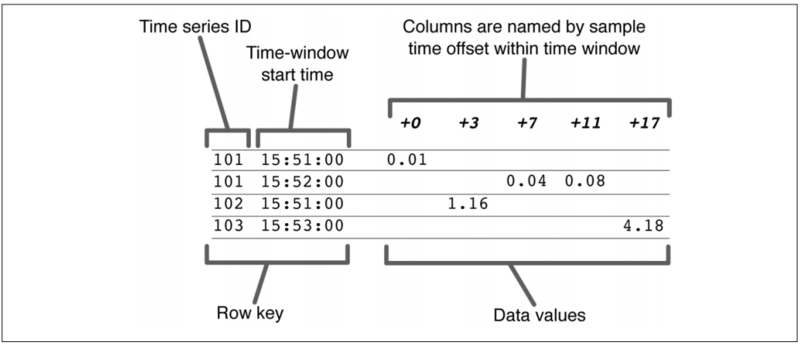


Figure 3-3

Because both HBase and MapR-DB store data ordered by the primary key, the key design shown in Figure 3-3 will cause rows containing data from a single time series to wind up near one another on disk. This design means that retrieving data from a particular time series for a time range will involve largely sequential disk operations and therefore will be much faster than would be the case if the rows were widely scattered. In order to gain the performance benefits of this table structure, the number of samples in each time window should be substantial enough to cause a significant decrease in the number of rows that need to be retrieved. Typically, the time window is adjusted so that 100– 1,000 samples are in each row.

## NoSQL Database with Hybrid Design

The table design shown in Figure 3-3 can be improved by collapsing all of the data for a row into a single data structure known as a blob. This blob can be highly compressed so that less data needs to be read from disk. Also, if HBase is used to store the time series, having a single column per row decreases the per-column overhead incurred by the on-disk format that HBase uses, which further increases performance. The hybrid-style table structure is shown in Figure 3-4, where some rows have been collapsed using blob structures and some have not.

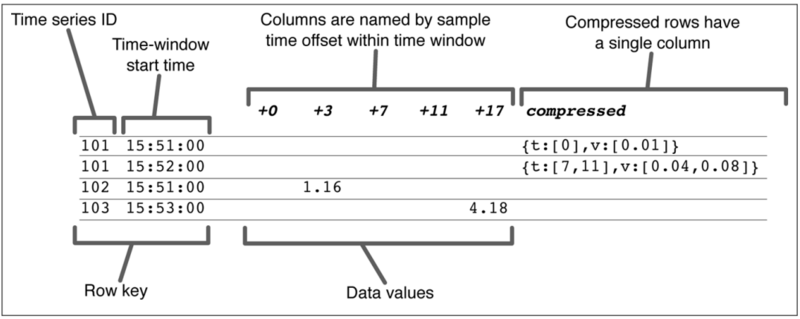


Figure 3-4

Data in the wide table format shown in Figure 3-3 can be progressively converted to the compressed format (blob style) shown in Figure 3-4 as soon as it is known that little or no new data is likely to arrive for that time series and time window. Commonly, once the time window ends, new data will only arrive for a few more seconds, and the compression of the data can begin. Since compressed and uncompressed data can coexist in the same row, if a few samples arrive after the row is compressed, the row can simply be compressed again to merge the blob and the late-arriving samples. The conceptual data flow for this hybrid-style time series database system is shown in Figure 3-5. Converting older data to blob format in the background allows a substantial increase in the rate at which the renderer depicted in Figure 3-5 can retrieve data for presentation. On a 4-node MapR cluster, for instance, 30 million data points can be retrieved, aggregated and plotted in about 20 seconds when data is in the compressed form.

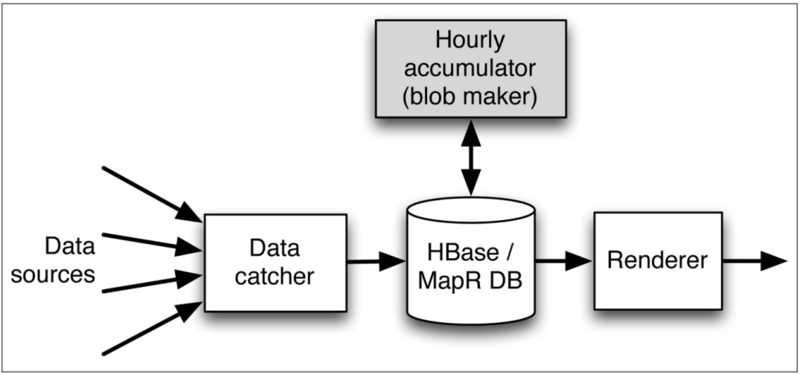


Figure 3-5

## Going One Step Further: The Direct Blob Insertion Design

Compression of old data still leaves one performance bottleneck in place. Since data is inserted in the uncompressed format, the arrival of each data point requires a row update operation to insert the value into the database. This row update can limit the insertion rate for data to as little as 20,000 data points per second per node in the cluster. On the other hand, the direct blob insertion data flow diagrammed in Figure 3-6 allows the insertion rate to be increased by as much as roughly 1,000-fold. How does the direct blob approach get this bump in performance? The essential difference is that the blob maker has been moved into the data flow between the catcher and the NoSQL time series database. This way, the blob maker can use incoming data from a memory cache rather than extracting its input from wide table rows already stored in the storage tier. The basic idea is that data is kept in memory as samples arrive. These samples are also written to log files. These log files are the “restart logs” shown in Figure 3-6 and are flat files that are stored on the Hadoop system but not as part of the storage tier itself. The restart logs allow the in-memory cache to be repopulated if the data ingestion pipeline has to be restarted. In normal operations, at the end of a time window, new in-memory structures are created, and the now static old in-memory structures are used to create compressed data blobs to write to the database. Once the data blobs have been written, the log files are discarded. Compare the point in the data flow at which writes occur in the two scenarios. In the hybrid approach shown in Figure 3-5, the entire incoming data stream is written point-by-point to the storage tier, then read again by the blob maker. Reads are approximately equal to writes. Once data is compressed to blobs, it is again written to the database. In contrast, in the main data flow of the direct blob insertion approach shown in Figure 3-6, the full data stream is only written to the memory cache, which is fast, rather than to the database. Data is not written to the storage tier until it’s compressed into blobs, so writing can be much faster. The number of database operations is decreased by the average number of data points in each of the compressed data blobs. This decrease can easily be a factor in the thousands.

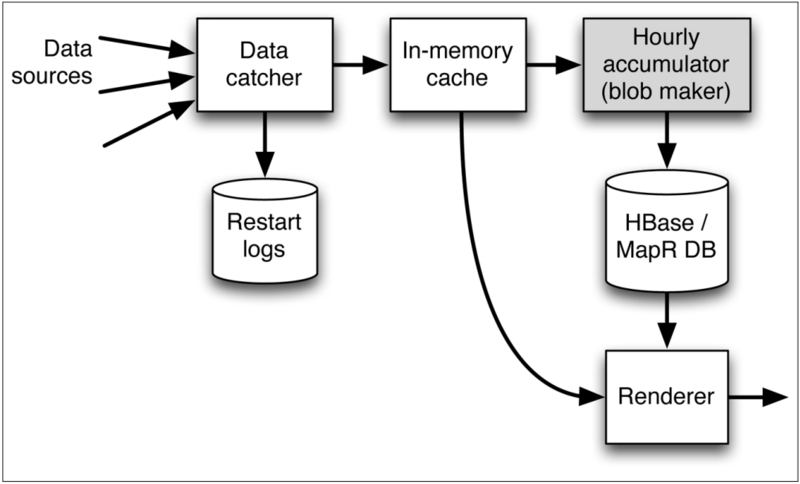


Figure 3-6

What are the advantages of this direct blobbing approach? A real-world example shows what it can do. This architecture has been used to insert in excess of 100 million data points per second into a MapR-DB table using just 4 active nodes in a 10-node MapR cluster. These nodes are fairly highperformance nodes, with 16 cores, lots of RAM, and 12 well-configured disk drives per node, but you should be able to achieve performance within a factor of 2–5 of this level using most hardware. This level of performance sounds like a lot of data, possibly more than most of us would need to handle, but in Chapter 5 we will show why ingest rates on that level can be very useful even for relatively modest applications.

## Why Relational Databases Aren’t Quite Right

At this point, it is fair to ask why a relational database couldn’t handle nearly the same ingest and analysis load as is possible by using a hybrid schema with MapR-DB or HBase. This question is of particular interest when only blob data is inserted and no wide table data is used, because modern relational databases often have blob or array types. The answer to this question is that a relational database running this way will provide reasonable, but not stellar, ingestion and retrieval rates. The real problem with using a relational database for a system like this is not performance, per se. Instead, the problem is that by moving to a blob style of data storage, you are giving up almost all of the virtues of a relational system. Additionally, SQL doesn’t provide a good abstraction method to hide the details of accessing of a blob-based storage format. SQL also won’t be able to process the data in any reasonable way, and special features like multirow transactions won’t be used at all. Transactions, in particular, are a problem here because even though they wouldn’t be used, this feature remains, at a cost. The requirement that a relational database support multirow transactions makes these databases much more difficult to scale to multinode configurations. Even getting really high performance out of a single node can require using a high-cost system like Oracle. With a NoSQL system like Apache HBase or MapR-DB instead, you can simply add additional hardware to get more performance. This pattern of paying a penalty for unused features that get in the way of scaling a system happens in a number of high-performance systems. It is common that the measures that must be taken to scale a system inherently negate the virtues of a conventional relational database, and if you attempt to apply them to a relational database, you still do not get the scaling you desire. In such cases, moving to an alternative database like HBase or MapR-DB can have substantial benefits because you gain both performance and scalability.

## Hybrid Design: Where Can I Get One?

These hybrid wide/blob table designs can be very alluring. Their promise of enormous performance levels is exciting, and the possibility that they can run on fault-tolerant, Hadoop-based systems such as the MapR distribution make them attractive from an operational point of view as well. These new approaches are not speculation; they have been built and they do provide stunning results. The description we’ve presented here so far, however, is largely conceptual. What about real implementations? The next chapter addresses exactly how you can realize these new designs by describing how you can use OpenTSDB, an open source time series database tool, along with special open source MapR extensions. The result is a practical implementation able to take advantage of the concepts described in this chapter to achieve high performance with a large-scale time series database as is needed for modern use cases.