第 14 章　可扩展性设计之数据切分

14.0 引言

通过 MySQL Replication 功能所实现的扩展总会受到数据库大小的限制，一旦数据库过于庞大，尤其是当写入过于频繁，很难由一台主机支撑的时候，我们还是会面临扩展瓶颈。此时，就必须找其他技术手段来解决这个瓶颈，即本章所要介绍的数据切分技术。

14.1 何谓数据切分

可能很多读者朋友在网上或杂志上都已经多次见到关于数据切分的相关文章了，只不过在有些文章中称之为数据的 Sharding。其实无论称之为数据的 Sharding 还是数据的切分，其实质都是一样的。简单来说，就是指通过某种特定的条件，将存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。数据的切分同时还可以提高系统的总体可用性，因为单台设备 Crash 之后，只有总体数据的某部分不可用，而不是所有的数据。

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。一种是按照不同的表（或者 Schema）来切分到不同的数据库（主机）之上，这种切分可以称之为数据的垂直（纵向）切分；另外一种则是根据表中数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上，这种切分称之为数据的水平（横向）切分。

垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低、相互影响很小、业务逻辑非常清晰的系统。在这种系统中，可以很容易做到将不同业务模块所使用的表分拆到不同的数据库中。根据不同的表来进行拆分，对应用程序的影响也更小，拆分规则也会比较简单清晰。

水平切分与垂直切分相比，稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中，对于应用程序来说，拆分规则本身就较根据表名来拆分更为复杂，后期的数据维护也会更复杂。

当某个（或者某些）表的数据量和访问量特别大，通过垂直切分将其放在独立的设备上后仍然无法满足性能要求时，就必须将垂直切分和水平切分相结合，先垂直切分，然后再水平切分，这样才能解决这种超大型表的性能问题。

下面就针对垂直、水平及组合切分这三种数据切分方式的架构实现及切分后数据的整合进行相应的分析。

14.2 数据的垂直切分

我们先来看一下，数据的垂直切分到底是如何切分的。数据的垂直切分，也可以称为纵向切分。将数据库想象成由很多个一大块一大块的“数据块”（表）组成，垂直地将这些“数据块”切开，然后把它们分散到多台数据库主机上面。这样的切分方法就是垂直（纵向）的数据切分。

一个架构设计较好的应用系统，其总体功能肯定是由很多个功能模块所组成的，而每一个功能模块所需要的数据对应到数据库中就是一个或多个表。而在架构设计中，各个功能模块相互之间的交互点越统一、越少，系统的耦合度就越低，系统各个模块的维护性及扩展性也就越好。这样的系统，实现数据的垂直切分也就越容易。

功能模块越清晰，耦合度越低，数据垂直切分的规则定义也就越容易。完全可以根据功能模块来进行数据的切分，不同功能模块的数据存放于不同的数据库主机中，可以很容易就避免跨数据库的 Join 存在，同时系统架构也非常清晰。

当然，很难有系统能够做到所有功能模块使用的表完全独立，根本不须要访问对方的表，或者须要将两个模块的表进行 Join 操作。这种情况下，就必须根据实际的应用场景进行评估权衡。决定是迁就应用程序将需要 Join 的表的相关模块都存放在同一个数据库中，还是让应用程序做更多的事情——完全通过模块接口取得不同数据库中的数据，然后在程序中完成 Join 操作。

一般来说，如果是一个负载相对不大，而且表关联又非常频繁的系统，那可能数据库让步，将几个相关模块合并在一起，减少应用程序工作的方案可以减少较多的工作量，是一个可行的方案。

当然，通过数据库的让步，让多个模块集中共用数据源，实际上也是间接默许了各模块架构耦合度增大的发展，可能会恶化以后的架构。尤其是当发展到一定阶段，发现数据库实在无法承担这些表所带来的压力，不得不面临再次切分时，所带来的架构改造成本可能远远大于最初就使用切分的架构设计。

所以，在数据库进行垂直切分的时候，如何切分、切分到什么样的程度，是一个比较考验人的难题。这只能在实际的应用场景中通过平衡各方面的成本和收益，才能分析出一个真正适合自己的拆分方案。

比如在本书所使用的示例系统的 example 数据库中，我们简单分析一下，然后设计一个简单的切分规则，进行一次垂直拆分。

系统功能基本可以分为 4 个功能模块：用户、群组消息、相册以及事件，分别对应为如下这些表：

（1）用户模块表：user,user_profile,user_group,user_photo_album

（2）群组讨论表：groups,group_message,group_message_content,top_message

（3）相册相关表：photo,photo_album,photo_album_relation,photo_comment

（4）事件信息表：event

初略一看，没有哪个模块可以脱离其他模块独立存在，模块与模块之间都存在着关系，莫非无法切分？

当然不是，再稍微深入分析一下，可以发现，虽然各个模块所使用的表之间都有关联，但是关联关系还算清晰，也比较简单。

- 群组讨论模块和用户模块之间主要存在通过用户或群组关系来进行关联。一般都会通过用户的 id 或 nick_name 及 group 的 id 来进行关联，通过模块之间的接口实现不会带来太多麻烦。
- 相册模块仅仅与用户模块存在用户的关联。这两个模块之间的关联基本只有通过用户 id 关联的内容，简单清晰，接口明确。
- 事件模块与各个模块可能都有关联，但是都只关注其各个模块中对象的 ID 信息，同样比较容易分拆。

所以，第一步可以将数据库按照功能模块相关的表进行一次垂直拆分，每个模块所涉及的表单独分到一个数据库中，模块与模块之间的表关联在应用系统端都通过接口来处理。如数据垂直切分示意图（图 14-1）所示：

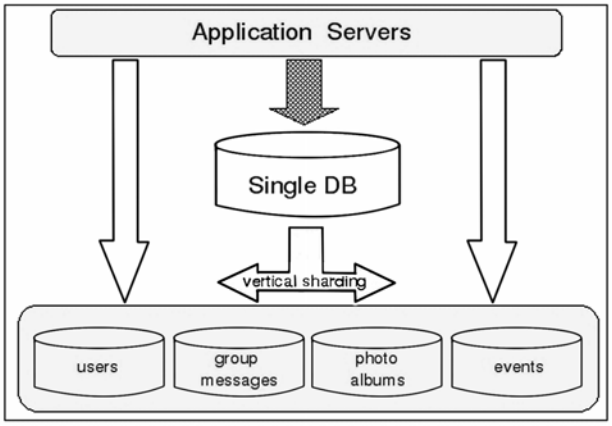

图 14-1 垂直切分示意

通过这样的垂直切分之后，之前只能通过一个数据库来提供的服务，就被分拆成 4 个数据库来提供服务，服务能力自然是增加几倍了。

垂直切分的优点：

- 数据库的拆分简单明了，拆分规则明确；
- 应用程序模块清晰明确，整合容易；
- 数据维护方便易行，容易定位。

垂直切分的缺点：

- 部分表关联无法在数据库级别完成，要在程序中完成；
- 对于访问极其频繁且数据量超大的表仍然存在性能瓶颈，不一定能满足要求；
- 事务处理相对复杂；
- 切分达到一定程度之后，扩展性会受到限制；
- 过度切分可能会带来系统过于复杂而难以维护。

针对于垂直切分可能遇到数据切分及事务问题，在数据库层面实在是很难找到一个较好的处理方案。实际应用案例中，数据库的垂直切分大多是与应用系统的模块相对应的，同一个模块的数据源存放于同一个数据库中，可以解决模块内部的数据关联问题。而模块与模块之间，则通过应用程序以服务接口的方式来相互提供所需要的数据。虽然这样做在数据库的总体操作次数方面确实会有所增加，但是在系统整体扩展性及架构模块化方面，都是有益的。可能某些操作的单次响应的时间会稍有增加，但是系统的整体性能很可能反而会有一定的提升。而扩展瓶颈问题，就只能依靠下一节将要介绍的数据水平切分架构来解决了。

14.3 数据的水平切分

上面一节分析介绍了数据的垂直切分，本节分析数据的水平切分。数据的垂直切分基本上可以简单地理解为按照表或模块来切分数据，而水平切分则不同。一般来说，简单的水平切分主要是将某个访问极其平凡的表再按照某个字段的某种规则分散到多个表中，每个表包含一部分数据。

简单来说，可以将数据的水平切分理解为按照数据行的切分，就是将表中的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中。当然，为了能够比较容易地判定各行数据被切分到哪个数据库中了，切分总是须要按照某种特定的规则来进行的：如根据某个数字类型字段基于特定数目取模，某个时间类型字段的范围，或者某个字符类型字段的 hash 值。如果整个系统中大部分核心表都可以通过某个字段来进行关联，那这个字段自然是一个进行水平分区的上上之选了，当然，非常特殊无法使用的情况除外。

一般来说，像现在非常火爆的 Web 2.0 类型网站，基本上大部分数据都能够通过会员用户

信息关联上，可能很多核心表都非常适合通过会员 ID 来进行数据的水平切分。而像论坛社区讨论系统，就更容易切分了，可以按照论坛编号来进行水平切分。切分之后基本上不会出现各个库之间的交互。

如果示例系统的所有数据都是和用户关联的，那么就可以根据用户来进行水平拆分，将不同用户的数据切分到不同的数据库中。当然，唯一区别是用户模块中的 groups 表和用户没有直接关系，所以 groups 不能根据用户来进行水平拆分。对于这种特殊情况下的表，完全可以独立出来，放在一个独立的数据库中。其实这个做法可以说是利用了前面一节所介绍的“数据的垂直切分”方法，将在下一节中更为详细地介绍这种垂直切分与水平切分同时使用的联合切分方法。

所以，对于示例数据库来说，大部分的表都可以根据用户 ID 来进行水平切分。不同用户相关的数据进行切分之后存放在不同的数据库中。如将所有用户 ID 通过被 2 取模然后分别存放于两个不同的数据库中。每个和用户 ID 关联上的表都可以这样切分。这样，基本上每个用户相关的数据，都在同一个数据库中，即使须要关联，也非常容易实现。

可以通过水平切分示意图（图 14-2）更为直观地展示水平切分相关信息：

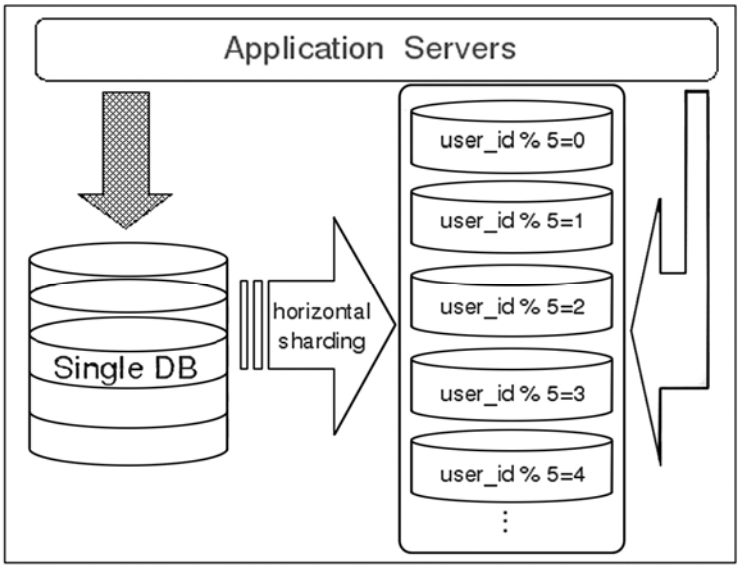

图 14-2

水平切分的优点：

- 表关联基本能够在数据库端全部完成；
- 不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；
- 应用程序端整体架构改动相对较少；

- 事务处理相对简单；
- 只要切分规则能够定义好，基本上较难遇到扩展性限制。

水平切分的缺点：

- 切分规则相对复杂，很难抽象出一个能够满足整个数据库的切分规则；
- 后期数据的维护难度有所增加，人为手工定位数据更困难；
- 应用系统各模块耦合度较高，可能会对后面数据的迁移拆分造成一定的困难。

14.4 垂直与水平联合切分的使用

前面两节内容中，分别了解了“垂直”和“水平”这两种切分方式的实现和切分之后的架构信息，以及两种架构各自的优缺点。但是在实际的应用场景中，除了那些负载并不是太大、业务逻辑也相对简单的系统可以通过上面两种切分方法之一来解决扩展性问题之外，恐怕其他大部分业务逻辑复杂、系统负载大的系统，都无法通过上面任何一种数据的切分方法来实现较好的扩展性，这就需要将上述两种切分方法结合使用，不同的场景使用不同的切分方法。

本节将结合垂直切分和水平切分各自的优缺点，进一步完善整体架构，并提高系统的扩展性。

一般来说，数据库中的所有表很难通过某一个（或少数几个）字段全部关联起来，所以仅仅通过数据的水平切分无法解决所有问题。而垂直切分也只能解决部分问题，对于那些负载非常高的系统，即使只是单个表都无法通过单台数据库主机来承担其负载。必须结合“垂直”和“水平”两种切分方式，充分利用两者的优点，避开其缺点。

每一个应用系统的负载都是一步一步增长上来的，在开始遇到性能瓶颈的时候，大多数架构师和 DBA 都会选择先进行数据的垂直拆分，因为这样的成本最低，最符合这个时期所追求的最大投入产出比。然而，随着业务的不断扩张，系统负载的持续增长，在系统稳定一段时期之后，经过了垂直拆分之后的数据库集群可能再次不堪重负，遇到了性能瓶颈。

此时该如何抉择？是再次进一步细分模块，还是寻求其他的解决办法？如果我们再像最开始那样继续细分模块，进行数据的垂直切分，那可能在不久的将来，又会遇到现在所面临的同样问题。而且随着模块的不断细化，应用系统的架构也会越来越复杂，整个系统很可能会出现失控的局面。

这时候就必须要利用数据水平切分的优势来解决遇到的问题。而且，完全不必在使用数据水平切分时，推倒之前进行数据垂直切分的成果，而是在其基础上利用水平切分的优势来避开垂直切分的弊端，解决系统复杂性不断扩大的问题。而水平拆分的弊端（规则难以统一）也已经被之前的垂直切分解决掉了，让水平切分可以进行得得心应手。

对于示例数据库，假设在最开始进行了数据的垂直切分，然而随着业务的不断增长，数据库系统遇到了瓶颈，我们选择重构数据库集群的架构。如何重构？考虑到之前已经做好了数据的垂直切分，而且模块结构清晰明确，而业务增长的势头越来越猛，即使现在再次拆分模块，也坚持不了太久。所以选择了在垂直切分的基础上再进行水平切分。

经历过垂直切分后的数据库集群中的各个数据库都只有一个功能模块，而每个功能模块中的所有表基本上都会与某个字段进行关联。如用户模块全部都可以通过用户 ID 进行切分，群组讨论模块则都通过群组 ID 来切分，相册模块则根据相册 ID 来进切分，最后的事件通知信息表考虑到数据的时限性（仅仅访问最近某个事件段的信息），则按时间来切分。

组合切分示意图（图 14-3）展示了切分后的整个架构：

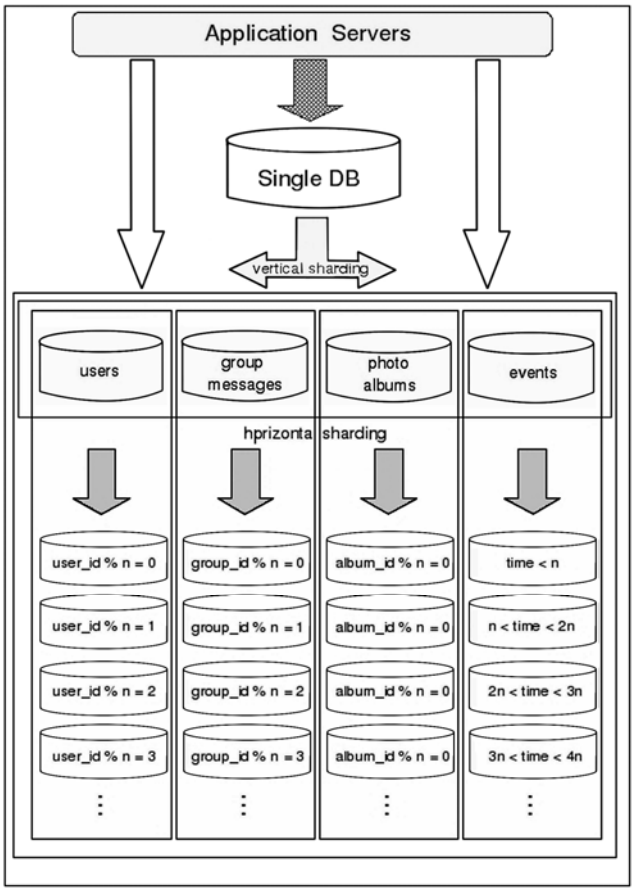

图 14-3 组合切分示意图

实际上，在很多大型的应用系统中，垂直切分和水平切分基本上是并存的，而且经常在不断

地交替进行，以增加系统的扩展能力。我们在应对不同的应用场景时，也须要充分考虑到这两种切分方法的局限及优势，在不同的时期（负载压力）使用不同的方式。

联合切分的优点：

- 可以充分利用垂直切分和水平切分各自的优势而避免各自的缺陷；
- 让系统扩展性得到最大化提升。

联合切分的缺点：

- 数据库系统架构比较复杂，维护难度更大；
- 应用程序架构也更复杂。

14.5 数据切分及整合方案

通过前面的章节，已经清楚了通过数据库的数据切分可以极大地提高系统的扩展性。但是，数据库中的数据经过垂直和（或）水平切分被存放在不同的数据库主机之后，应用系统面临的最大问题就是如何让这些数据源得到较好的整合，可能这也是很多读者非常关心的一个问题。本节主要的内容就是分析各种可以帮助我们实现数据切分及数据整合的整体解决方案。

数据的整合很难依靠数据库本身来达到，虽然 MySQL 存在 Federated 存储引擎，可以解决部分类似的问题，但是在实际应用场景中却很难较好地运用。那该如何来整合这些分散在各个 MySQL 主机上的数据源呢？

总的来说，存在两种解决思路：

（1）在每个应用程序模块中配置管理自己需要的一个（或者多个）数据源，直接访问各个数据库，在模块内完成数据的整合；

（2）通过中间代理层来统一管理所有的数据源，后端数据库集群对前端应用程序透明。

可能 90%以上的人在面对这两种解决思路时都会倾向于选择第二种，尤其是系统不断庞大复杂的时候。确实，这是一个非常正确的选择，虽然短期内须要付出的成本可能会相对大一些，但对整个系统的扩展性来说，是非常有帮助的。

所以，对于第一种解决思路就不过多分析了，下面重点分析第二种思路中的一些解决方案。

1. 自行开发中间代理层

在决定选择通过数据库的中间代理层来解决数据源整合的架构方向之后，有不少公司（或者企业）自行开发了符合自身应用特定场景的代理层应用程序。

自行开发中间代理层可以最大程度地应对自身应用的特点，最大化定制个性化需求，在面对

变化的时候也可以灵活应对。这应该是自行开发代理层最大的优势了。

当然，选择自行开发，享受个性化定制最大化乐趣的同时，自然也需要投入更多的成本来进行前期研发及后期的持续升级改进工作，而且本身的技术门槛可能也比简单的 Web 应用更高。所以，在决定选择自行开发之前，仍须要进行比较全面的评估。

由于自行开发更多时候考虑的是如何更好地适应自身应用系统，应对自身的业务场景，所以这里也不好分析太多。下面将主要分析当前比较流行的几种数据源整合解决方案。

2. 利用 MySQL Proxy 实现数据切分及整合

MySQL Proxy 是 MySQL 官方提供的一个数据库代理层产品，和 MySQL Server 一样，它也是一个基于 GPL 开源协议的开源产品。可用来监视、分析或传输它们之间的通讯信息。它的灵活性允许最大限度地使用它，目前具备的功能主要有连接路由、Query 分析、Query 过滤和修改、负载均衡，以及基本的 HA 机制等。

实际上，MySQL Proxy 本身并不具有上述所有的功能，而是提供了实现上述功能的基础。要实现这些功能，还须要我们自行编写 LUA 脚本。

MySQL Proxy 实际上是在客户端请求与 MySQL Server 之间建立了一个连接池。所有客户端请求都发向 MySQL Proxy，然后经由 MySQL Proxy 进行相应的分析，判断出是读操作还是写操作，分发至对应的 MySQL Server 上。对于多节点 Slave 集群，也可以起到负载均衡的效果。如 MySQL Proxy 基本架构图（图 14-4）：

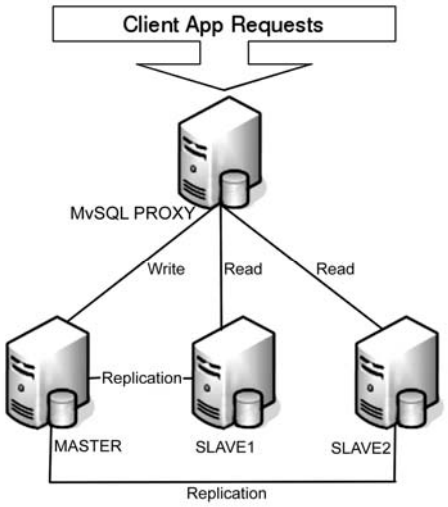

图 14-4 MySQL Proxy 架构

通过上面的架构简图，可以清晰地看到 MySQL Proxy 在实际应用中所处的位置，以及能做

的基本事情。MySQL Proxy 详细的实施细则在 MySQL 官方文档中有非常详细的介绍和示例，感兴趣的读者朋友可以直接从 MySQL 官方网站免费下载或者在线阅读，这里就不赘述。

3. 利用 Amoeba 实现数据切分及整合

Amoeba 是一个基于 Java 开发的，专注于解决分布式数据库数据源整合 Proxy 程序的开源框架，基于 GPL3 开源协议。目前，Amoeba 已经具有 Query 路由、Query 过滤、读写分离、负载均衡及 HA 机制等相关内容，如图 14-5 所示。

Amoeba 主要解决以下几个问题：

（1）数据切分后复杂数据源整合；

（2）提供数据切分规则并降低数据切分规则给数据库带来的影响；

（3）降低数据库与客户端的连接数；

（4）读写分离路由。

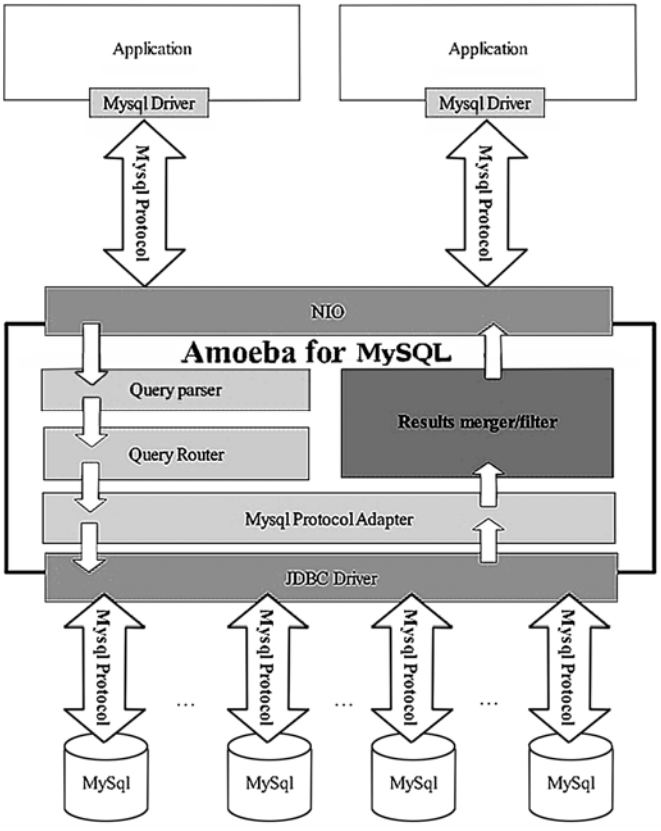

图 14-5 Amoeba For MySQL 架构

可以看出，Amoeba 所做的事情，正好就是通过数据切分来提升数据库的扩展性所需要的。

Amoeba 并不是一个代理层的 Proxy 程序，而是一个开发数据库代理层 Proxy 程序的框架，目前基于 Amoeba 所开发的 Proxy 程序有 Amoeba For MySQL 和 Amoeba For Aladin 两个。

Amoeba For MySQL 是专门针对 MySQL 数据库的解决方案，前端应用程序请求的协议及后端连接的数据源数据库都必须是 MySQL。对于客户端的任何应用程序来说，Amoeba For MySQL 和一个 MySQL 数据库没有什么区别，任何使用 MySQL 协议的客户端请求，都可以被 Amoeba For MySQL 解析并进行相应的处理。Amoeba For 可以告诉我们 Amoeba For MySQL 的架构信息（出自 Amoeba 开发者博客）：

Amoeba For Aladin 则是一个适用更为广泛、功能更为强大的 Proxy 程序。它可以同时连接不同数据库的数据源为前端应用程序提供服务，但是仅仅接受符合 MySQL 协议的客户端应用程序请求。也就是说，只要前端应用程序通过 MySQL 协议连接上来，Amoeba For Aladin 会自动分析 Query 语句，根据 Query 语句中所请求的数据来自动识别出该 Query 的数据源是在什么类型数据库的哪一个物理主机上。Amoeba For Aladdin 架构图（图 14-6）展示了 Amoeba For Aladin 的架构细节（出自 Amoeba 开发者博客）。

乍一看，两者好像完全一样嘛。细看才会发现两者主要的区别仅在于通过 MySQL Protocal Adapter 处理之后，根据分析结果判断出数据源数据库，然后选择特定的 JDBC 驱动和相应协议连接后端数据库。

其实通过上面两个架构图大家可能已经发现了 Amoeba 的特点，它只是一个开发框架，我们除了选择它已经提供的 For MySQL 和 For Aladin 这两款产品之外，还可以基于自身的需求进行二次开发，得到更适合自己应用特点的 Proxy 程序。

但对于使用 MySQL 数据库来说，不论是 Amoeba For MySQL 还是 Amoeba For Aladin 都可以很好地使用。当然，考虑到任何一个系统越是复杂，其性能肯定就会有一定的损失，维护成本自然也会更高一些。所以，在仅仅须要使用 MySQL 数据库的时候，还是建议使用 Amoeba For MySQL。

Amoeba For MySQL 的使用非常简单，所有的配置文件都是标准的 XML 文件，总共有 4 个，分别如下：

- amoeba.xml——主配置文件，配置所有数据源及 Amoeba 自身的参数；
- rule.xml——配置所有 Query 路由规则的信息；
- functionMap.xml——配置用于解析 Query 中的函数所对应的 Java 实现类；
- rullFunctionMap.xml——配置路由规则中需要使用到的特定函数的实现类。

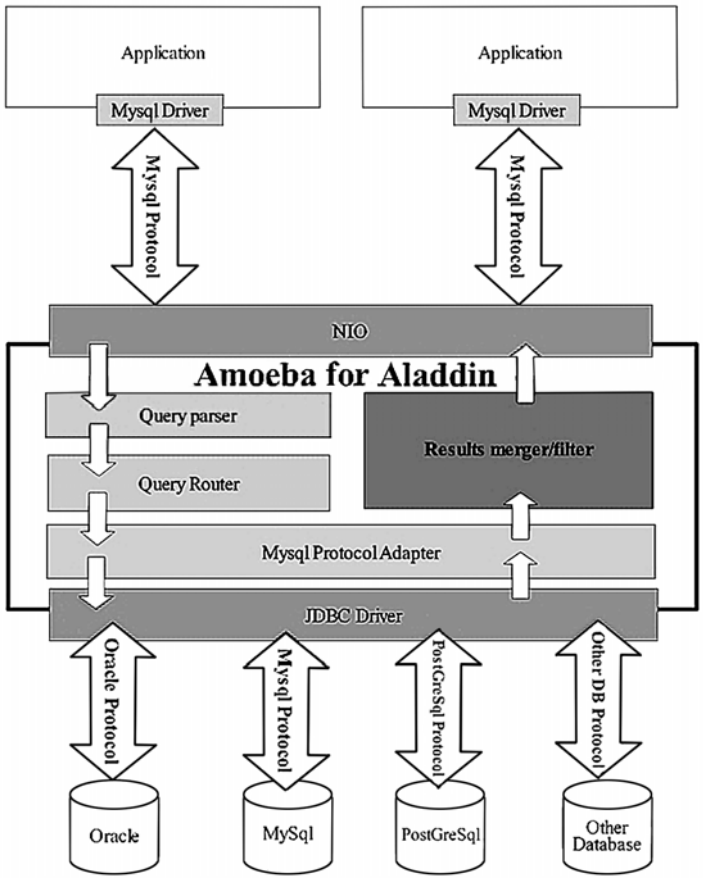

图 14-6 Amoeba For Aladdin 架构

如果您的规则不是太复杂，基本上仅使用上面4个配置文件中的前面两个就可完成所有工作。Proxy 程序常用的功能如读写分离、负载均衡等配置都在 amoeba.xml 中进行。此外，Amoeba 已经支持了实现数据的垂直切分和水平切分的自动路由，路由规则可以在 rule.xml 进行设置。

目前 Amoeba 稍有欠缺的主要就是其在线管理功能及对事务的支持方面了，曾经在与相关开发者的沟通过程中提出过这方面的建议，希望能够提供一个可以进行在线维护管理的命令行管理工具，方便在线维护使用，得到的反馈是管理专门的管理模块已经纳入开发日程了。另外在事务支持方面 Amoeba 暂时还无法做到，即使客户端应用在提交给 Amoeba 的请求时包含事务信息的，Amoeba 也会忽略事务相关信息。当然，在经过不断完善之后，我相信事务支持肯定是 Amoeba 重点考虑的功能。

关于 Amoeba 更为详细的使用方法读者可以通过 Amoeba 开发者博客（*http://amoeba. sf.net*）上面提供的使用手册获取，这里就不再细述了。

4. 利用 HiveDB 实现数据切分及整合

和前面的 MySQL Proxy 及 Amoeba 一样，HiveDB 同样是一个基于 Java 针对 MySQL 数据库的提供数据切分及整合的开源框架，只是目前的 HiveDB 仅仅支持数据的水平切分。主要解决大数据量下数据库的扩展性及数据的高性能访问问题，同时支持数据的冗余及基本的 HA 机制。

HiveDB 的实现机制与 MySQL Proxy 和 Amoeba 有一定的差异，它并不是借助 MySQL 的 Replication 功能来实现数据的冗余，而是自行实现了数据冗余机制，而其底层主要是基于 Hibernate Shards 来实现数据切分工作。

在 HiveDB 中，通过用户自定义的各种 Partition keys（即制定数据切分规则），将数据分散到多个 MySQL Server 中。访问时运行 Query 请求，则会自动分析过滤条件，并行从多个 MySQL Server 中读取数据，并合并结果集返回给客户端应用程序。

单纯从功能方面来讲，HiveDB 可能并不如 MySQL Proxy 和 Amoeba 那样强大，但是其数据切分的思路与前面二者并无本质差异。此外，HiveDB 并不只是一个开源爱好者所共享的内容，而是存在商业公司支持的开源项目。

HiveDB 官方网站上的 HiveDB 架构示意图（图 14-7），描述了 HiveDB 如何来组织数据的基本信息，虽然不能详细地表现出架构方面的信息，但是也基本可以展示其在数据切分上独特的一面了。

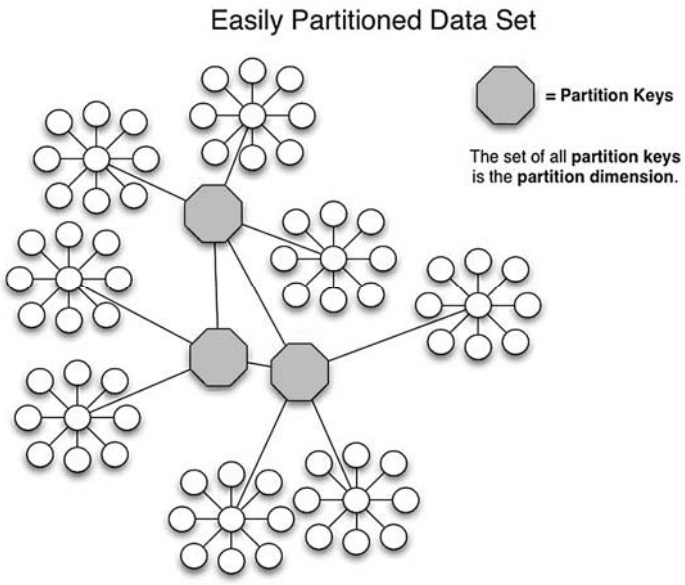

图 14-7 HiveDB 架构示意

5. 其他实现数据切分及整合的解决方案

除了上面介绍的几个数据切分及整合的整体解决方案之外，还存在很多其他的解决方案、如在 MySQL Proxy 的基础上做进一步扩展的 HSCALE，通过 Rails 构建的 Spock Proxy，以及基于 Pathon 的 Pyshards，等等。

不管大家选择使用哪一种解决方案，总体设计思路基本上都不应该有任何变化，即通过数据的垂直和水平切分，增强数据库的整体服务能力，让应用系统的整体扩展能力尽量得到提升，扩展方式尽可能便捷。

只要通过中间层 Proxy 应用程序较好地解决了数据切分和数据源整合问题，那么数据库的线性扩展能力将像应用程序一样方便：只要通过添加廉价的 PC Server 服务器，即可线性增加数据库集群的整体服务能力，让数据库不再轻易成为应用系统的性能瓶颈。

14.6 数据切分与整合中可能存在的问题

这里，大家应该对数据切分与整合的实施有一定的认识了，或许很多读者都已经根据各种解决方案的优劣基本选定了适合于自己应用场景的方案，后面的工作主要就是实施准备了。

在实施数据切分方案之前，仍要分析一些可能存在的问题。一般来说，可能遇到的问题主要有以下几点：

- 引入分布式事务的问题；
- 跨节点 Join 的问题；
- 跨节点合并排序分页问题。

1. 引入分布式事务的问题

一旦数据进行切分被分别存放在多个 MySQL Server 中，不管切分规则设计得多么完美（实际上并不存在完美的切分规则），都可能造成之前某些事务所涉及的数据已经不在同一个 MySQL Server 中了。

在这样的场景下，如果应用程序仍然按照老的方案，那么势必须要引入分布式事务来解决。而在 MySQL 各个版本中，只有从 MySQL 5.0 开始以后的各个版本才对分布式事务提供支持，而且目前仅有 Innodb 提供分布式事务支持。不过，即使我们刚好使用了支持分布式事务的 MySQL 版本，同时也使用 Innodb 存储引擎，分布式事务本身对于系统资源的消耗就很大，性能也并不太高，引入分布式事务在异常处理方面会带来很多比较难控制的问题。

怎么办？其实可以通过一个变通的方法来解决这种问题，首先须要考虑的是：数据库是否是唯一一个能够解决事务的地方？其实并不是这样的，完全可以结合数据库及应用程序来共同解

决。各个数据库解决自身的事务，然后通过应用程序来控制多个数据库上的事务。

也就是说，只要我们愿意，完全可以将一个跨多个数据库的分布式事务分拆成多个仅处于单个数据库上的小事务，并通过应用程序来总控各个小事务。当然，这样做要求应用程序必须要有足够的健壮性，当然也会给应用程序带来一些技术难度。

2. 跨节点 Join 的问题

上面介绍了可能引入分布式事务的问题，现在再看看需要跨节点 Join 的问题。数据切分之后，也许有些老的 Join 语句无法继续使用，因为 Join 使用的数据源可能被切分到多个 MySQL Server 中了。

怎么办？这个问题从 MySQL 数据库角度来看，如果非得在数据库端直接解决的话，恐怕只能通过 MySQL 一种特殊的存储引擎 Federated 处理了。Federated 存储引擎是 MySQL 解决类似于 Oracle 的 DB Link 之类问题的方案。和 Oracle DB Link 的主要区别在于，Federated 会保存一份远端表结构的定义信息在本地。乍一看，Federated 确实是解决跨节点 Join 非常好的方案。但是我们还应该清楚一点，那就是如果远端的表结构发生了变更，本地的表定义信息是不会跟着发生变化的。如果在更新远端表结构的时候并没有更新本地的 Federated 表定义信息，Query 运行很可能出错，无法得到正确的结果。

对待这类问题，还是推荐通过应用程序来处理，先在驱动表所在的 MySQL Server 中取出驱动结果集，然后根据驱动结果集再到被驱动表所在的 MySQL Server 中取出相应的数据。可能很多读者朋友会认为这样做将对性能产生一定的影响，是的，确实会有一定的负面影响，但除此之外，基本上没有太多其他更好的解决办法了。而且，由于数据库通过较好的扩展之后，每台 MySQL Server 的负载就可以得到较好的控制，单纯针对单条 Query 来说，其响应时间可能比不切分之前要提高一些，所以性能方面带来的负面影响也并不是太大。更何况，类似于这种跨节点 Join 的需求也并不是太多，相对于总体性能而言，可能也只是很小一部分而已。所以为了整体性能，偶尔牺牲一点点，其实是值得的，毕竟系统优化本身就是很多取舍和平衡的过程。

3. 跨节点合并排序分页问题

一旦进行了数据的水平切分之后，可能就并不只有跨节点 Join 无法正常运行，有些排序分页的 Query 语句的数据源可能也会被切分到多个节点，其直接后果就是这些排序分页 Query 无法继续正常运行。其实这和跨节点 Join 是一个道理，数据源存在于多个节点上，要通过一个 Query 来解决，就是一个跨节点 Join 操作。同样 Federated 也可以部分解决，但存在的风险也一样。但是有一点不同：Join 很多时候都有一个驱动与被驱动的关系，所以它涉及的多个表之间的数据读取一般会存在一个顺序关系。但是排序分页就不同了，排序分页的数据源基本上可以说是一个表（或者一个结果集），并不存在顺序关系，所以从多个数据源取数据的过程是完全可以并行的。这样，排序分页数据的取数效率可以比跨库 Join 更高，所以带来的性能损失相对要小，

在有些情况下可能比在原来未进行数据切分的数据库中效率更高了。当然，不论是跨节点 Join 还是跨节点排序分页，都会使应用服务器消耗更多的资源，尤其是内存资源，因为在读取访问及合并结果集的这个过程须要比不处理合并处理更多的数据。

分析到这里，可能很多读者朋友会发现，上面所有的这些问题，我的建议基本上都是通过应用程序来解决的。大家可能心里开始犯嘀咕了，是不是因为我是 DBA，所以就把很多事情都扔给应用架构师和开发人员了？

其实完全不是这样，首先应用程序由于其特殊性，可以非常容易做到很好的扩展性，但是数据库就不一样，必须借助很多其他的方式才能做到扩展，而且在扩展过程中，很难避免带来有些原来在集中式数据库中可以解决但被切分开成一个数据库集群之后就成为一个难题的情况。要想让系统整体得到最大限度的扩展，只能让应用程序做更多的事情，来解决数据库集群无法较好解决的问题。

14.7 小结

通过数据切分技术将一个大的 MySQL Server 切分成多个小的 MySQL Server，既解决了写入性能瓶颈问题，同时也再一次提升了整个数据库集群的扩展性。不论是通过垂直切分，还是水平切分，都能够让系统遇到瓶颈的可能性更小。尤其是在使用垂直和水平相结合的切分方法之后，理论上将不再遇到扩展瓶颈了。