大型数据库系统应用设计方法

可扩展性、高可用性及负载均衡

基本概念

可扩展性（Scalability|伸缩性）：

在一些大的系统中，预测最终用户的数量和行为是非常困难的，**可扩展性**是指系统适应不断增长的用户数的能力。提高这种并发会话能力的一种最直观的方式就增加资源（CPU，内存，硬盘等），集群是解决这个问题的另一种方式，它允许一组服务器组在一起，像单个服务器一样分担处理一个繁重的任务。

高可用性（High availability）：

单一服务器的解决方案并不是一个健壮方式，因为容易出现单点失效。像银行、账单处理这样一些关键的应用程序是不能容忍哪怕是几分钟的死机。它们需要这样一些服务在任何时间都可以访问并在可预期的合理的时间周期内有响应。集群方案通过在集群中增加的冗余的服务器，使得在其中一台服务器失效后仍能提供服务，从而获得高的可用性。

负载均衡（Load balancing）：

负载均衡是集群的一项关键技术，通过把请求分发给不同的服务器，从而获得高可用性和较好的性能。一个负载均衡器可以是从一个简单的Servlet或Plug-Ins（例如一个Linux box利用ipchains来实现），到昂贵的内置SSL加速器的硬件。除此之外，负载均衡器还需执行一些其他的重要任务，如“会话胶粘”让一个用户会话始终存在一个服务器上，“健康检查”用于防止将请求分发到已失效的服务器上。有些负载均衡器也会参与我们下面将要谈到“失效转移”过程。

基本概念

容错（Fault tolerance）：

高可用性意味着对数据正确性的要求不那么高。在J2EE集群中，当一个服务器实例失效后，服务仍然是有效的，这是因为新的请求将被冗余服务器处理。但是，当一个请求在一个正在失效的服务器中处理时，可能得到不正确的结果。不管有多少个错误，容错的服务应当能确保有严格的正确的行为。

失效转移（Failover）：

失效转移是集群中用来获取容错能力的另一项关键的技术。当一个结点失效后，通过选择集群中的另一个结点，处理将会继续而不会终止。转移到另一个结点可以被显式的编码，或是通过底层平台自动地透明地路由到另一个服务器。

等幂方法（Idempotent methods）：

等幂方法是指这样一些方法：重复用相同的参数调用都能得到相同的结果。这些方法不会影响系统状态，可以重复调用而不用担心改变系统。例如：getUsername()就是等幂的，而deleteFile就不是。当我们讨论HTTP Session失效转移和EJB失效转移时，它是一个重要的概念。

讨论的背景

主题

数据库基本问题调查

关系数据库的基本背景

ACID基本概念解析

范式问题解析（Normalization）

数据库基本问题调查

大家都使用过哪些数据库?

哪些内容是数据库系统的关键点?

常见的数据存储

传统的数据库系统

- Oracle
- DB2、SQL Server
- MySQL、PosgreSQL

分布式数据库

- Google Spanner & BigTable & MegaStore
- OceanBase、Hbase

缓存服务器 & KeyValue Store

- Tair
- Memcached
- Redis

关系数据库的主要业务场景

Billing （记账类业务，电信、银行）

Booking （订票类业务，航空）

Inventory （库存管理，零售）

这些业务的共同特征是什么?

数据库的主要特性

ACID

- 原子性（Atomicity）
- 完整性（Consistency）
- 隔离性 （Isolation）
- 持久性 （Durability）

Relation & SQL

- Structured Query Language （即SQL）
- A Relational Model of Data for Large Shared Data Banks （By **Edgar Codd**）

ACID的基础概念

Transaction的概念借自Contract Law

- 一手交钱、一手交货（Atomicity）
- 不会出现库存为负，也不会出现资金为负的情况（Consistency）
- 可同时与多人进行交易（Isolation）
- 离柜概不负责（Durability）

Atomicity

- 要么全部成功，要么全不成功

Consistency

- 写入数据库的数据必须满足所有定义的约束规则（主键、唯一键、外键等约束）

Isolation

- 确保并发执行的事务就如同串行执行的事务一样，保证系统状态（state）的一致性。

Durability

- 一旦提交，哪怕出现掉电、Crash也不会丢数据

ACID

- **原子性**
  - 整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- **一致性**
  - 在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。
- **隔离性**
  - 隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。
- **持久性**
  - 在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。
- **由于一项操作通常会包含许多子操作，而这些子操作可能会因为硬件的损坏或其他因素产生问题，要正确实现ACID并不容易。**

几个基础概念

Write-Ahead Log

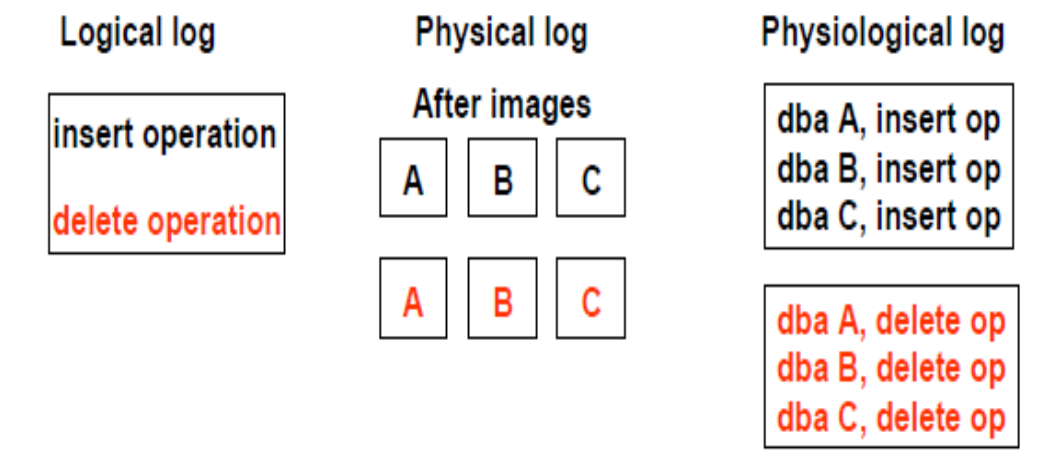
Redo

- Logical
- Physical
- Physiological

Undo

事务槽 - 事务标识

SCN – 系统变更统一时间戳（逻辑时钟）



如何实现原子性

事务槽为变更入口，单一入口（原子）

每个变更的记录都包含事务槽信息

数据库中如何保证C

通过Read Dirty与锁来解决PK/UK

通过Ref检查来解决FK的问题（需要Index）

通过PreCommit trigger来做Null以及Check

数据库中如何保证

锁控制

- 不同粒度的锁（表级、块级、记录级）
- 不同维度的锁（数据相关锁，内存相关锁）

MVCC

- Snapshot Isolation
- Block Image + SCN + Undo Image 判断

差别在于读取哪个时间点的Snapshot

数据库中如何保证D

Log before Data

LGWR before DBWn

Flush Log on Commit

- **Durability On Commit**

Checkpoint Before Redo Log File Reuse

ACID的代价

不同的Isolation对应不同的代价

- Serialiazability
- Read Committed (Through Snapshot)
- Read Dirty？ (没有并发控制)

不同的Durability级别

- Flush on Commit
- Flush on Timeout (Time Range)
- Flush on Batch (commits count?)

NORMALIZATION解决的问题

- Basically, normalization is the **process of efficiently organising data** in a database.
- There are two main objectives of the normalization process: **eliminate redundant data** (storing the same data in more than one table) and **ensure data dependencies make sense** (only storing related data in a table).
- Both of these are valuable goals as they **reduce the amount of space a database consumes** and **ensure that data is logically stored**.

NORMALIZATION带来的问题

表之间的依赖（关系依赖，耦合）

表关联的成本（关联开销，可能的IO开销）

系统扩展的复杂度（解耦合）

如何权衡NORMALIZATION

尽量不要对静态数据做Normalization

- 除非你希望节约存储空间

考虑范式化 Vs 反范式化的投入产出

为什么很多IT新人喜欢Normalization

- 那是因为他们的老师告诉他们需要

建议

- 适度的使用
- 关键在于判断业务之间的耦合性

数据库的扩展性问题

数据库架构、系统架构在于：

- 如何满足如下的要求

检索问题

- Relation

并发问题

- Isolation
- Consistency(UK)

一致性问题

- Isolation

速度问题

- Performance, Durability+Isolation

数据库检索问题

如何从班级的联系方式中找到XX的电话号码?

如何从公司的联系方式中找到XX的电话号码?

如何从移动公司的系统中找到XX的电话号码?

如何从移动、电信、联通的数据库找到XX的电话号码?

数据库的并发问题

同时有多个人要购买手机号?

如何保证大家购买的不是同一个手机号?

如何支持几百、几千、几万人同时购买手机号?

数据库的一致性问题

如何保证大家看到的库存有效?

如何保证读取的信息是准确的?

库存的变更如何实时的提供给每一个人看到?

数据库的性能问题?

如何快速的让1个人买到号码?

- 有多快?

如何快速的让10个人买到号码?

- 要不要排队?
- 一个服务员?
- 一个营业厅?

PERFORMANCE VS SCALABILITY

1.当只有一个人访问时，速度如何?

2.当有很多人访问时，速度如何?

- 大家都同样快?

如果满足1

- 表示Performance很好?

如何能较好的满足2

- 表示系统有较好的Scalability

一致性问题再探讨

- 新浪发的微博需要强一致吗?

ITPUB的论坛需要强一致吗?

当当的图书描述信息需要强一致吗?

12306的火车票库存信息需要强一致吗?

支付宝/财付通的账户余额需要强一致吗?

中行信用卡/招商银行卡的账户信息需要强一致吗?

深入讨论扩展性

数据库系统的扩展性

- **Scale（扩展）就是让我们的数据库能够提供更强的服务能力，更强的处理能力。**
- **Scalable（可扩展）则是表明数据库系统在通过相应升级（包括增加单机处理能力或者增加服务器数量）之后能够达到提供更强处理能力。在理论能上来说，任何数据库系统都是 Scalable 的，只不过是所需要的实现方式不一样而已。**
- **Scalability（扩展性）则是指一个数据库系统通过相应的升级之后所带来处理能力提升的难易程度。虽然理论上任何系统都可以通过相应的升级来达到处理能力的提升，但是不同的系统提升相同的处理能力所需要的升级成本（资金和人力）是不一样的，这也就是我们所说的各个数据库应用系统的 Scalability 存在很大的差异。**

数据库系统的扩展性

- **Scale Up 则是指纵向的扩展，向上扩展，也就是通过增加当前处理节点的处理能力来提高整体的处理能力，是通过升级现有服务器的配置，如增加内存，增加CPU，增加存储系统的硬件配置，或者是直接更换为处理能力更强的服务器和更为高端的存储系统。**

- **Scale Out 就是指横向的扩展，向外扩展，也就是通过增加处理节点的方式来提高整体处理能力，即通过增加机器来增加整体的处理能力。**

SCALE UP 优缺点

- **Scale Up 优点：**
  - 处理节点少，维护相对简单；
  - 所有数据都集中在一起，应用系统架构简单，开发相对容易；

- **Scale Up 缺点**
  - 高端设备成本高，且竞争少，容易受到厂家限制；
  - 受到硬件设备发展速度限制，单台主机的处理能力总是有极限的，容易遇到最终无法解决的性能瓶颈；
  - 设备和数据集中，发生故障后的影响较大；

SCALE OUT 优缺点

- **Scale Out 优点**
  - 成本低，很容易通过价格低廉的PC Server 搭建出一个处理能力非常强大的计算集群；
  - 不容易遇到瓶颈，因为很容易通过添加主机增加处理能力；
  - 单个节点故障对系统整体影响较小；也存在缺点，更多的计算节点，大部分时候都是服务器主机，这自然会带来整个系统维护复杂性的提高，在某些方面肯定会增加维护成本，而且对应用系统的架构要求也会比 Scale Up 更高，需要集群管理软件的配合。
- **Scale Out 缺点**
  - 处理节点多，造成系统架构整体复杂度提高，应用程序复杂度提高；
  - 集群维护难以程度更高，维护成本更大；

SCALABILITY很好的数据库应用系统遵循的原则

- 事务相关性最小化原则
- 数据一致性原则
- 高可用及数据安全原则

事务相关性最小化原则

- **分布式的架构带来分布式事务的问题**
  - 在传统的集中式数据库架构中，事务的问题非常好解决，可以完全依赖数据库本身非常成熟的事务机制来保证。但是一旦我们的数据库作为分布式的架构之后，很多原来在单一数据库中所完成的事务现在可能需要跨多个数据库主机，这样原来单机事务可能就需要引入分布式事务的概念。

- **分布式事务本身就是一个非常复杂的机制**
  - 不管是商业的大型数据库系统还是各开源数据库系统，虽然大多数数据库厂家基本上都实现了这个功能，但或多或少都存在各种各样的限制。而且也存在一些 Bug，可能造成某些事务并不能很好的保证，或者是不能顺利的完成。

一些解决方案

1. **进行 Scale Out 设计的时候合理设计切分规则，尽可能保证事务所需数据在同一个 MySQL Server 上，避免分布式事务。**
2. **大事务切分成多个小事务，数据库保证各个小事务的完整性，应用控制各个小事务之间的整体事务完整性。**
3. **结合上述两种解决方案，整合各自的优势，避免各自的弊端。**
    I. 比如我们可以在保证部分核心事务所需数据在同一个 MySQL Server 上，而其他并不是特别重要的事务，则通过分拆成小事务和应用系统结合来保证。
    II. 通过这样相互平衡设计的原则，我们既可以避免应用程序需要处理太多的小事务来保证其整体的完整性，同时也能够避免拆分规则太多复杂而带来后期维护难度的增加及扩展性受阻的情况

数据一致性原则

- **如何在 Scale Out 的同时又较好的保证数据一致性呢？**
  - BASE 模型。即：基本可用，柔性状态，基本一致和最终一致
  - 简单来说，应用系统通过相关的技术实现，让整个系统在满足用户使用的基础上，允许数据短时间内处于非一致状态，而通过后续技术来保证数据在最终保证处于一致状态。

- **高可用及数据安全原则**
  - 在支持可扩展性的同时，要注意高可用性和数据安全。

SHARDING\REPLICATION\CLUSTER

基本方法

SHARDING

SHARE NOTHING

- 并行数据库要求尽可能的去并行执行数据库操作，从而提高性能。在并行计算体系结构实现中有很多可选的体系结构。包括：

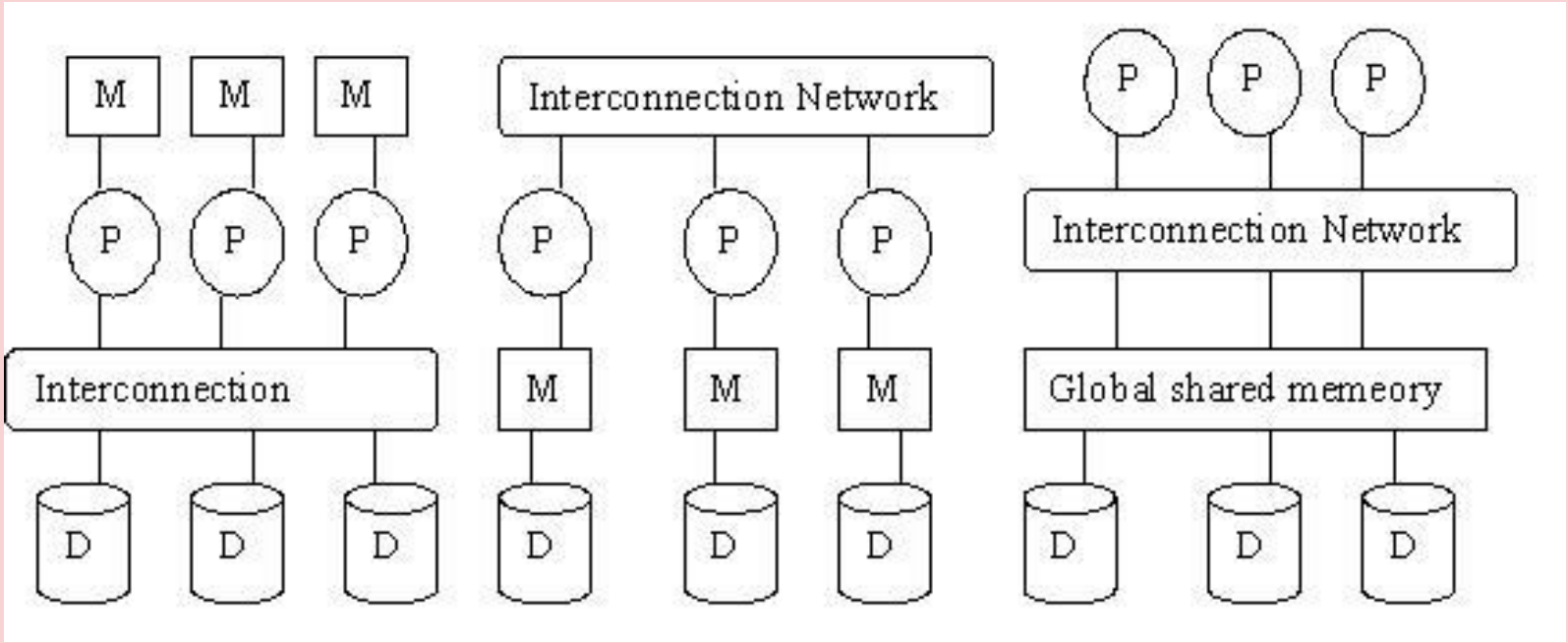
  - **Share-memory**：多个cpu共享同一片内存，cpu之间通过内部通讯机制（interconnection network）进行通讯；

  - **Share-disk**： 每一个cpu使用自己的私有内存区域，通过内部通讯机制直接访问所有磁盘系统。

  - **Share-nothing**： 每一个cpu都有私有内存区域和私有磁盘空间，而且2个cpu不能访问相同磁盘空间，cpu之间的通讯通过网络连接。

并行计算体系结构

Share disk　　share nothing　　share memory



并行计算体系结构

- Shared memory 体系结构的cpu之间通过主存进行通讯，具有很高的效率；但当更多的cpu被添加到主机上时，内存竞争contection就成为瓶颈，cpu越多，瓶颈越厉害。Shared disk也存在同样问题，因为磁盘系统由 Interconnection Network 连接在一起。
- Shared memory和shared disk的基本问题是interference：当添加更多的cpu，系统反而减慢，因为增加了对内存访问（memroy access）和网络带宽（network bandwidth）的竞争。这样shared nothing体系得到了广泛的推广。
- Shared nothing体系是数据库稳定增长，当随着事务数量不断增加，增加额外的cpu和主存就可以保证每个事务处理时间不变。

SHARED NOTHING ARCHITECTURE (SN)

A **shared nothing architecture** (SN) is a distributed computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system. More specifically, none of the nodes share memory or disk storage.

People typically contrast SN with systems that keep a large amount of centrally-stored state information, whether in a database, an application server, or any other similar single point of contention. While SN is best known in the context of web development, the concept predates the web: Michael Stonebraker at the University of California, Berkeley used the term in a 1986 database paper.[1] In it he mentions existing commercial implementations of the architecture (although none are named explicitly). Teradata, which delivered its first system in 1983, was probably one of those commercial implementations.[2] Tandem Computers officially released NonStop SQL, a shared nothing database, in 1984. [3]

SHARED NOTHING ARCHITECTURE (SN)

Shared nothing is popular for web development because of its scalability. As Google has demonstrated, a pure SN system can scale almost infinitely simply by adding nodes in the form of inexpensive computers, since there is no single bottleneck to slow the system down.[4] Google calls this *sharding*. A SN system typically partitions its data among many nodes on different databases (assigning different computers to deal with different users or queries), or may require every node to maintain its own copy of the application's data, using some kind of coordination protocol. This is often referred to as *database sharding*.

从 SHARD 到 SHARDING

- “Shard” 这个词英文的意思是”碎片”，而作为数据库相关的技术用语，似乎最早见于大型多人在线角色扮演游戏(MMORPG)中。“Sharding” 称之为“分片”。
- Sharding 不是一门新技术，而是一个相对简朴的软件理念。MySQL 5 之后才有了数据表分区功能，那么在此之前，很多 MySQL 的潜在用户都对 MySQL 的扩展性有所顾虑，而是否具备分区功能就成了衡量一个数据库可扩展性与否的一个关键指标(当然不是唯一指标)。数据库扩展性是一个永恒的话题，MySQL 的推广者经常会被问到：如在单一数据库上处理应用数据捉襟见肘而需要进行分区化之类的处理，是如何办到的呢？答案是：Sharding。
- Sharding 不是一个某个特定数据库软件附属的功能，而是在具体技术细节之上的抽象处理，是水平扩展(Scale Out，亦或横向扩展、向外扩展)的解决方案，其主要目的是为突破单节点数据库服务器的 I/O 能力限制，解决数据库扩展性问题。

数据库扩展性

目前的商业数据都有自己的扩展性解决方案，在过去相对来说比较成熟，但是随着互联网的高速发展，不可避免的会带来一些计算模式上的演变，这样很多主流商业系统也难免暴露出一些不足之处。比如 Oracle 的 RAC 是采用共享存储机制，对于 I/O 密集型的应用，瓶颈很容易落在存储上，这样的机制决定后续扩容只能是 Scale Up（向上扩展） 类型，对于硬件成本、开发人员的要求、维护成本都相对比较高。

Sharding 基本上是针对开源数据库的扩展性解决方案，很少有听说商业数据库进行 Sharding 的。目前业界的趋势基本上是拥抱 Scale Out，逐渐从 Scale Up 中解放出来。

SHARDING 的应用场景

- 任何技术都是在合适的场合下能发挥应有的作用。 Sharding 也一样。联机游戏、IM、BSP 都是比较适合 Sharding 的应用场景。其共性是抽象出来的数据对象之间的关联数据很小。
  - IM ，每个用户如果抽象成一个数据对象，完全可以独立存储在任何一个地方，数据对象是 Share Nothing 的；
  - Blog 服务提供商的站点内容，基本为用户生成内容(UGC)，完全可以把不同的用户隔离到不同的存储集合，而对用户来说是透明的。
- “Share Nothing” 是从数据库集群中借用的概念，有些类型的数据粒度之间就不是 “Share Nothing” 的，比如类似交易记录的历史表信息，如果一条记录中既包含卖家信息与买家信息，如果随着时间推移，买、卖家会分别与其它用户继续进行交易， Sharding会可能导致两个买卖家的信息会分布到不同的 Sharding DB 上，而这时如果针对买卖家查询，就会跨越更多的 Sharding ，开销就会比较大。
- Sharding 并不是数据库扩展方案的银弹，也有其不适合的场景，如处理事务型应用会非常复杂。对于跨不同DB的事务，很难保证完整性，得不偿失。所以，采用什么样的 Sharding 形式，不是生搬硬套的。

SHARDING与数据库分区(PARTITION)的区别

- 有的时候，Sharding 也被近似等同于水平分区(Horizontal Partitioning)，网上很多地方也用 水平分区来指代 Sharding，但二者之间实际上还是有区别的。Sharding 的思想是从分区的思想而来，但数据库分区基本上是数据对象级别的处理，比如表和索引的分区，每个子数据集上能够有不同的物理存储属性，还是单个数据库范围内的操作，而 Sharding 是能够跨数据库，甚至跨越物理机器的。

	Sharding	分区
存储依赖	可跨越DB 可跨越物理机器	可跨越表空间，不同的物理属性， 不能跨DB存储
数据划分	常见为时间、范围、面向服务等	范围、Hash、列表，混合分区等
存储方式	分布式	集中式
扩展性	Scale Out	Scale Up
可用性	无单点	存在单点(DB 本身)
价格	低廉	适中(DAS) 甚至昂贵(SAN)
应用场景	常见于 Web 2.0 网站	多数传统应用

SHARDING 策略

Sharding根据切分规则类型，可分为两种切分模式：

•数据的垂直（纵向）切分是按照不同的表（或者 Schema）来切分到不同的数据库（主机）之上；

•数据的水平（横向）切分是根据表中的数据的逻辑关系，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）上面。

数据的垂直切分

数据的垂直切分，也可以称之为纵向切分。将数据库想象成为由很多个一大块一大块的“数据块”（表）组成，我们垂直的将这些“数据块”切开，然后将他们分散到多台数据库主机上面。这样的切分方法就是一个垂直（纵向）的数据切分。

当我们的功能模块越清晰，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，可以很容易就避免掉跨数据库的 Join 存在，同时系统架构也非常的清晰。

EXAMPLE 数据库-垂直划分

系统功能可以基本分为四个功能模块：用户，群组消息，相册以及事件，分别对应为如下这些表：

•用户模块表：

- user, user_profile, user_group, user_photo_album

•群组讨论表：

- groups, group_message, group_message_content, top_message

•相册相关表：

- photo, photo_album, photo_album_relation, photo_comment

•事件信息表：

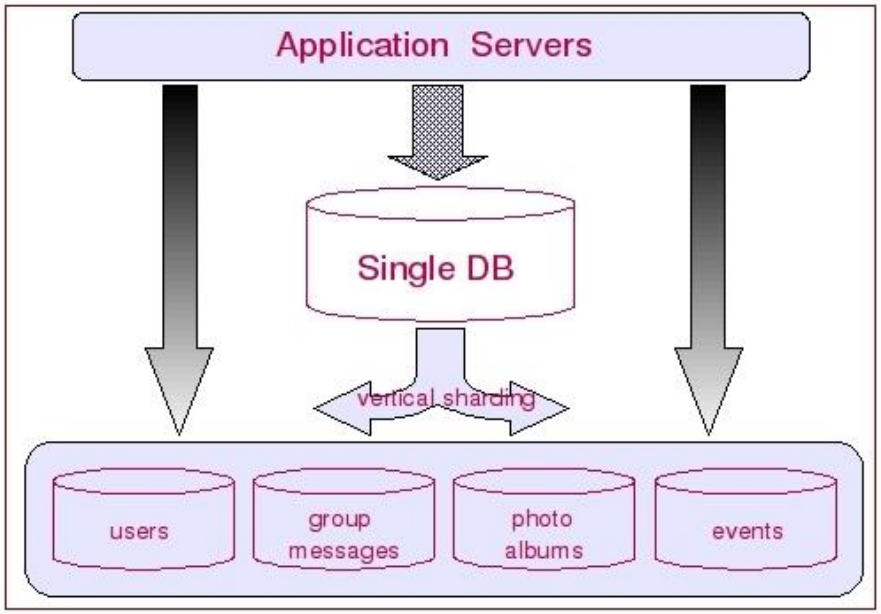
- event

EXAMPLE 数据库-垂直划分

- 群组讨论模块和用户模块之间主要存在通过用户或者是群组关系来进行关联。一般关联的时候都会是通过用户的 id 或者 nick_name 以及 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通过用户的关联。这两个模块之间的关联基本就有通过用户 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，同样可以做到很容易分拆。

EXAMPLE 数据库-垂直划分

我们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到一个数据库中，模块与模块之间的表关联都在应用系统端通过接口来处理。



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，就被分拆成四个数据库来提供服务，服务能力自然是增加几倍了。

垂直切分的优缺点

- **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，需要在程序中完成；
  - 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；
  - 事务处理相对更为复杂；
  - 切分达到一定程度之后，扩展性会遇到限制；
  - 过度切分可能会带来系统过渡复杂而难以维护。

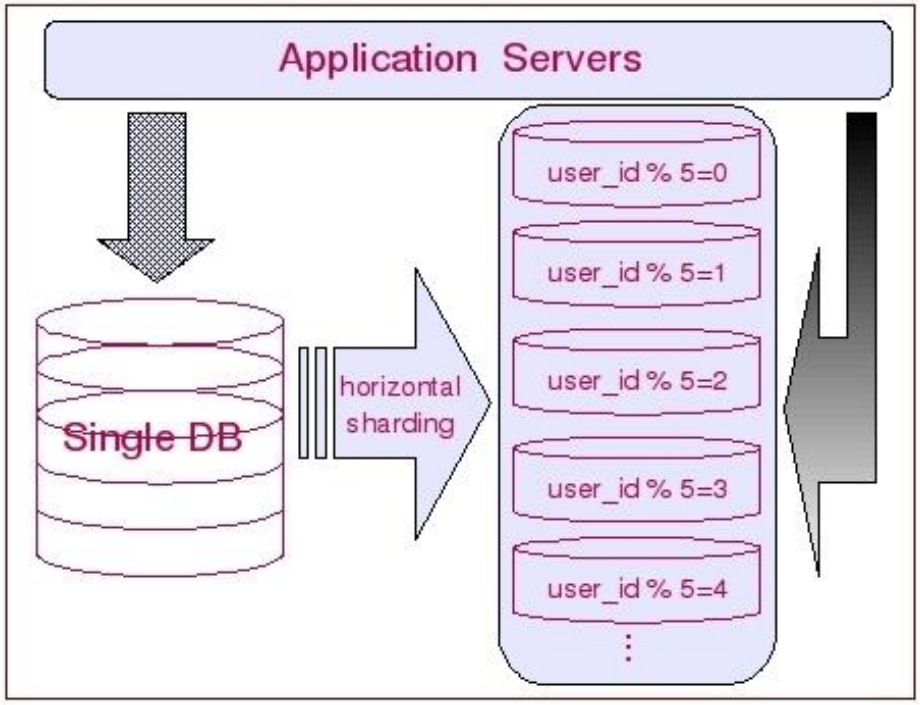
数据的水平切分

- **水平切分可理解为是按照数据行的切分，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。**
  - 为了能够比较容易的判定各行数据被切分到哪个数据库中了，切分总是都需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，或者是某个字符类型字段的 hash 值。

- **如EXAMPLE数据库，所有数据都是和用户关联的，那么我们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。**
- **唯一有点区别的是用户模块中的 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，则可以独立出来，单独放在一个数据库中。**

EXAMPLE数据库-水平划分

大部分的表都可以根据用户 ID 来进行水平的切分。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 通过取模（模5）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，即使是需要关联，也可以非常简单的关联上。



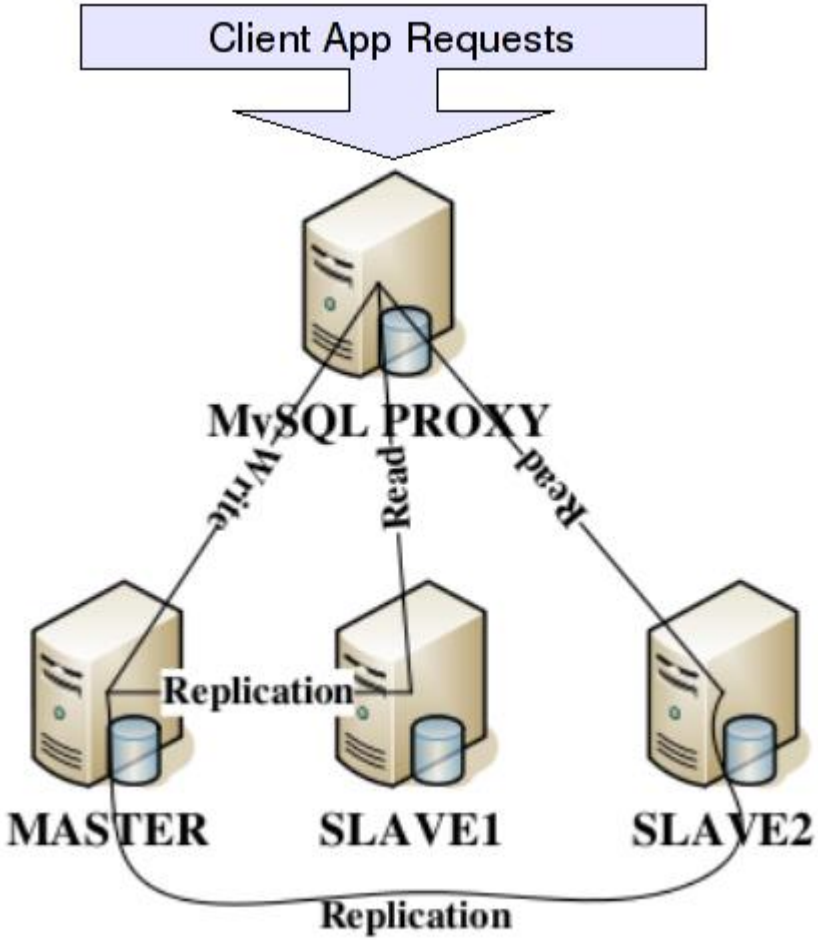
水平切分的优缺点

- **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**
  - 切分规则相对更为复杂，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。

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，同样是一个基于 GPL 开源协议的开源产品。可用来监视、分析或者传输他们之间的通讯信息。他的灵活性允许你最大限度的使用它，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这些功能，而是提供了实现上述功能的基础。要实现这些功能，还需要通过我们自行编写 LUA 脚本来实现。
- 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。所有客户端请求都是发向 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，也可以起做到负载均衡的效果。



高可用性
HIGH AVAILABILITY

SINGLE MYSQL SERVER

MySQL Client

SQL (SELECT, INSERT, UPDATE, DELETE)

MySQL Storage Engines (InnoDB, MyISAM, PBXT...)

Disk Storage (XFS, ReiserFS, JFS, ext3...)

WHY HA?

- **Something can and will fail**
  - Service Maintenance
  - Downtime is expensive
  - Adding HA to an existing system is complex

- **墨菲定律（Murphy's Law）**
  - Anything that can go wrong will go wrong

高可用性HA-IBM定义

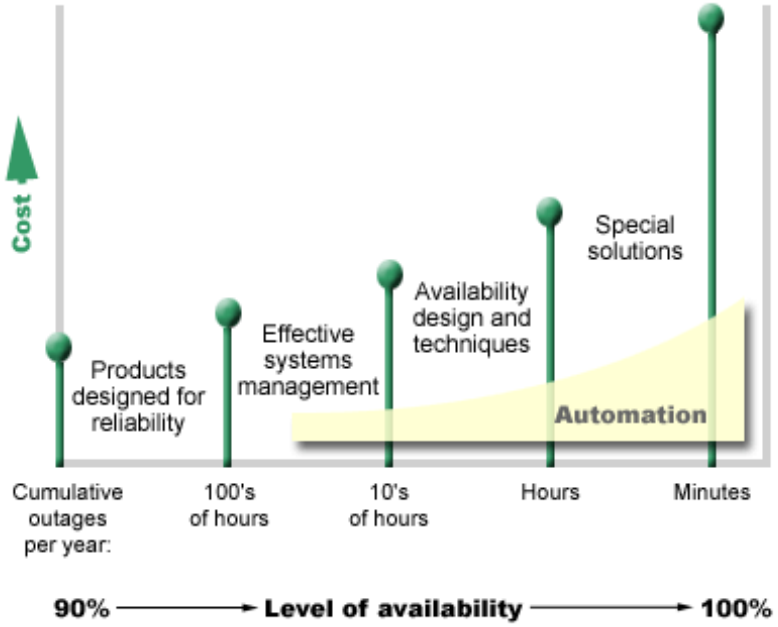
业务连续性是指企业的一种能力，有了此能力，企业能够抵御中断，并根据预定义的服务级别协议正常且连续不断地经营重要服务。要实现期望的给定级别业务连续性，必须选择一系列服务、软件、硬件和过程，用文档计划加以描述，付诸实现并定期实践。业务连续性解决方案必须解决有关数据、运营环境、应用程序、用于主管环境的应用程序以及最终用户接口的问题。所有这些都必须予以提供，才能交付完整的业务连续性解决方案。

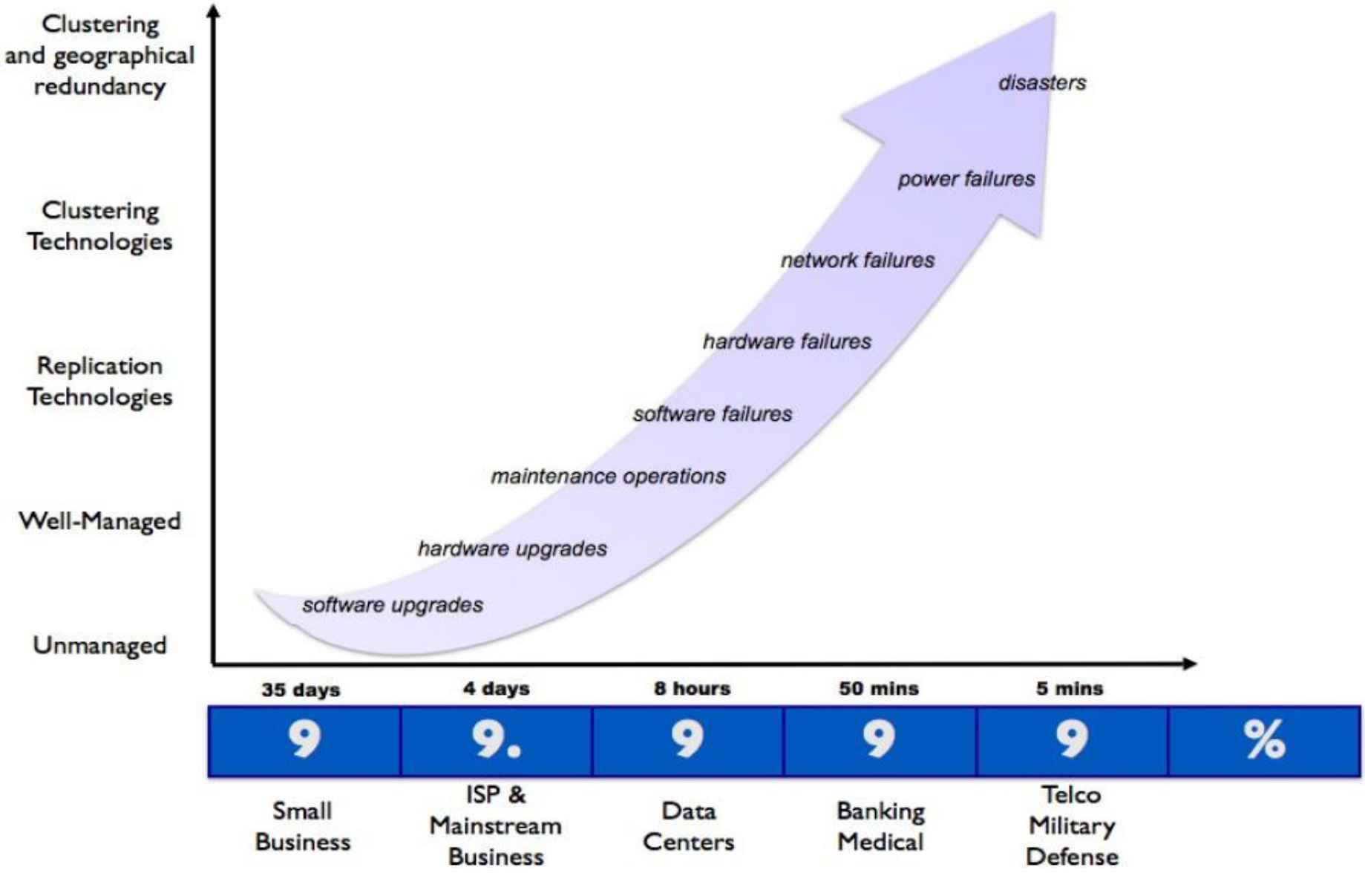
业务连续性包括灾难恢复 (DR) 和高可用性 (HA)，是指抵御所有中断（预期中断、意外中断以及灾难），并为所有重要应用程序提供连续处理的能力。最终目标是让中断时间少于总服务时间的 0.001%。与灾难恢复方案相比，高可用性环境通常包括要求更为苛刻的恢复时间目标（数秒到数分钟）和恢复点目标（零用户中断）。

可用性级别	每年的停机时间
90%	36.5 天
95%	18.25 天
99%	3.65 天
99.9%	8.76 小时
99.99%	50 分钟
99.999%	5 分钟

需要 100% 可用性的应用程序吗?

在大多数情况下，可以通过实施合理的处理和系统管理实务来实现高级别的可用性。当所需要的越接近连续可用性，则必须作出的投资就越大。在作出这种投资之前，应该确信需要该级别的可用性。下列数字显示不同技术所能提高可用性的程度，但是需要付出的成本可能会随之增加。

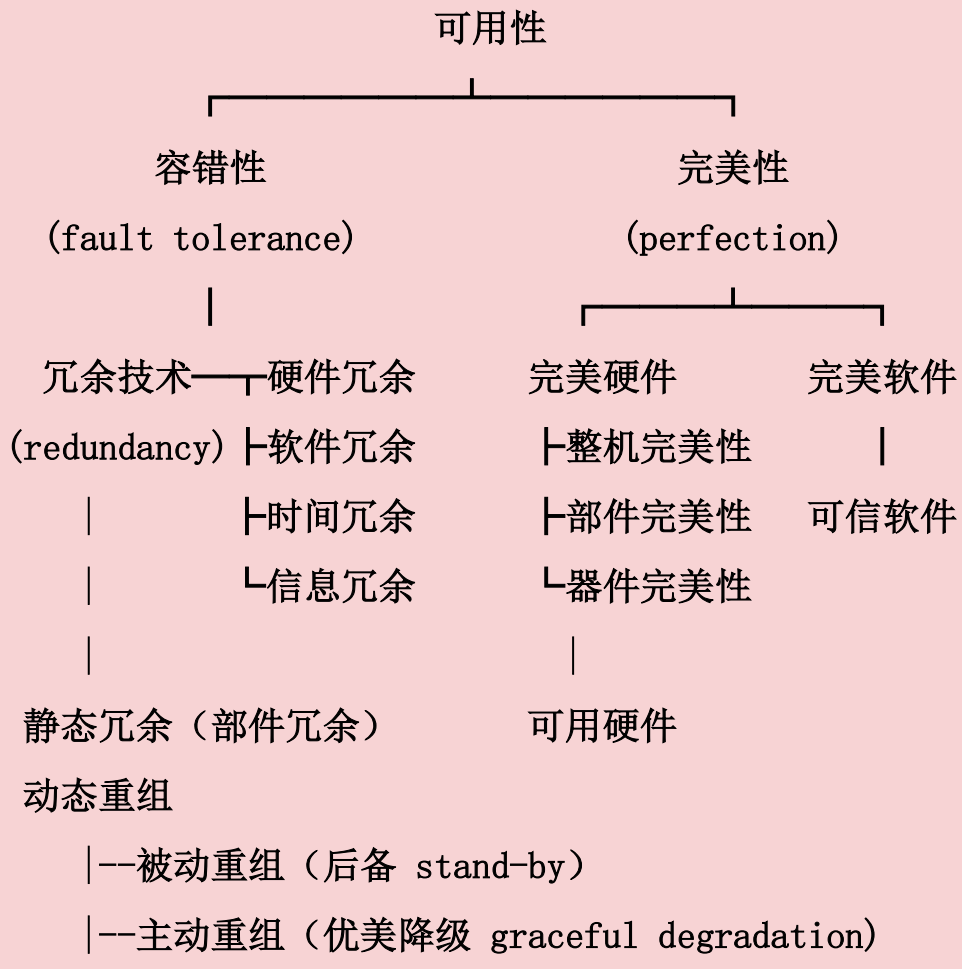



Clustering and geographical redundancy
Clustering Technologies
Replication Technologies
Well-Managed
Unmanaged
disasters
power failures
network failures
hardware failures
software failures
maintenance operations
hardware upgrades
software upgrades
35 days
4 days
8 hours
50 mins
5 mins
9
9.
9
9
9
%
Small Business
ISP & Mainstream Business
Data Centers
Banking Medical
Telco Military Defense

系统可用性的定义

- **在特定时间内和特定条件下系统正常工作的相应程度。**
- **可用性的测量方式：**
  - 系统的可用性(availability)，即利用率。
  - 可用性的平均值即平均利用率，其计算方法为：
    - A = MTBF / (MTBF + MTTR)
  - MTBF(MeanTime Between Failures)
    - 故障间隔平均时间
  - MTTR(MeanTime To Repair)
    - 系统平均修复时间

系统可用性的获得


可用性
容错性
(fault tolerance)
完美性
(perfection)
冗余技术
(redundancy)
硬件冗余
软件冗余
时间冗余
信息冗余
静态冗余（部件冗余）
动态重组
|--被动重组（后备 stand-by）
|--主动重组（优美降级 graceful degradation)
完美硬件
整机完美性
部件完美性
器件完美性
可用硬件
完美软件
可信软件

完美性与避错技术

完美性追求一种避错技术，即避免出错。要求组成系统的各个部件、器件具有高可用性，不允许出错，或者出错率降至最低。

•硬件的可用性与完美性

- 指元器件的完美性、部件的完美性、整机与系统的完美性

•软件的可用性与完美性

- 指软件的正确性、完美性、兼容性

容错性与容错技术

- **容错技术：在一定程度上容忍故障的技术**
- **容错系统：采用容错技术的系统**
  - 当系统因某种原因出错或者失效，系统能够继续工作，程序能够继续运行，不会因计算机故障而中止或被修改，执行结果也不包含系统中故障引起的差错。

- **容错技术也称为故障掩盖技术(fault masking)。**

容错性与容错技术

- 冗余技术是容错技术的重要结构，它以增加资源的办法换取可用性。由于资源的不同，冗余技术分为硬件冗余、软件冗余、时间冗余和信息冗余。资源与成本按线性增加，而故障概率则可按对数规律下降。

- 冗余要消耗资源，应当在可用性与资源消耗之间进行权衡和折衷。

双CPU容错系统

当一个 CPU板出现故障时，另一个 CPU保持继续运行。这个过程对用户是透明的，系统没有受到丝毫影响，更不会引起交易的丢失，充分保证数据的一致性和完整性。系统的容错结构能够提供系统连续运行的能力，任何单点故障不会引起系统停机，系统提供在线的维护诊断工具可在应用继续运转的情况下修复单点故障。

冗余类型

1.硬件冗余：

增加线路、设备、部件，形成备份。

2.软件冗余：

增加程序，一个程序分别用几种途径编写，

按一定方式执行，分段或多种表决。

3.时间冗余：

指令重复执行，程序回卷技术。

4.信息冗余：

增加信息数据位数，检错、纠错。

容错系统工作方式
自动侦测(AUTO-DETECT)

- **自动侦测(Auto-Detect)**

  通过专用的冗余侦测线路和软件判断系统运行情况，发现可能的错误和故障，进行严谨的判断与分析。确认主机出错后，启动后备系统。

  侦测程序需要检查主机硬件（处理器与外设部件）、主机网络、操作系统、数据库、重要应用程序、外部存储子系统（如磁盘阵列）等。

- **为了保证侦测的正确性，防止错误判断，系统可以设置安全侦测时间、侦测时间间隔、侦测次数等安全系数，通过冗余通信连线，收集并记录这些数据，作出分析处理。**
- **数据可信是切换的基础。**

自动切换(AUTO-SWITCH)

- 当确认某一主机出错时，正常主机除了保证自身原来的任务继续运行外，将根据各种不同的容错后备模式，接管预先设定的后备作业程序，进行后续程序及服务。
- 系统的接管工作包括文件系统、数据库、系统环境（操作系统平台）、网络地址和应用程序等。
- 如果不能确定系统出错，容错监控中心通过与管理者交互进行有效的处理。
- 决定切换基础、条件、时延、断点

自动恢复(AUTO-RECOVERY)

- 故障主机被替换后，离线进行故障修复。修复后通过冗余通信线与正常主机连线，继而将原来的工作程序和磁盘上的数据自动切换回修复完成的主机上。这个自动完成的恢复过程用户可以预先设置，也可以设置为半自动或不恢复。

常用方法

双机双工热备份(MUTUAL BACKUP)

- **两机同时运行，分不同作业，各自资源负载，故障、接管、修复、交还。**
  - 双主机通过一条TCP/IP网络线以及一条RS-232电缆线相联
  - 双主机各自通过一条SCSI电缆线与RAID磁盘阵列相联
  - 双主机各自运行不同的作业，彼此独立，并相互备援
  - 主机A故障后，主机B自动接管主机A运行。
  - 主机A的作业将在主机B上自动运行。
  - 主机A修复后，主机B将把A的作业自动交还主机A。
  - 主机B故障时，主机A接管主机B的作业和数据。
  - 主机B修复时,主机A再将原来接管的作业和数据交还主机B 。

主从热备份 (MASTER/SLAVE)

- **主从式（M/S），M运行，S后备，M故障，S接管并升级为M，原M修复后作为S。**
  - 双主机通过一条TCP/IP网络线以及一条RS-232电缆线相联。
  - 双主机各自通过一条SCSI电缆线与RAID相联。
  - 主机A为Master，主机B为Slave。
  - 主机A处理作业和数据，主机B作为热备份机。
  - 主机A故障后，主机B自动接管主机A的作业和数据。
  - 主机B同时接管A的主机名(Host)及网络地址(IP)。
  - 主机A的作业将在主机B上自动运行。
  - 主机A的客户(client)可继续运行，无需重新登录。
  - 主机B现为Master,主机A修复后作为Slave，作为热备份机。

RULES OF HIGH AVAILABILITY

- Prepare for failure
- Aim to ensure that no important data is lost
- Keep it simple, stupid (KISS)
- Complexity is the enemy of reliability
- Automate it
- Test your setup frequently!

高可用常用方法

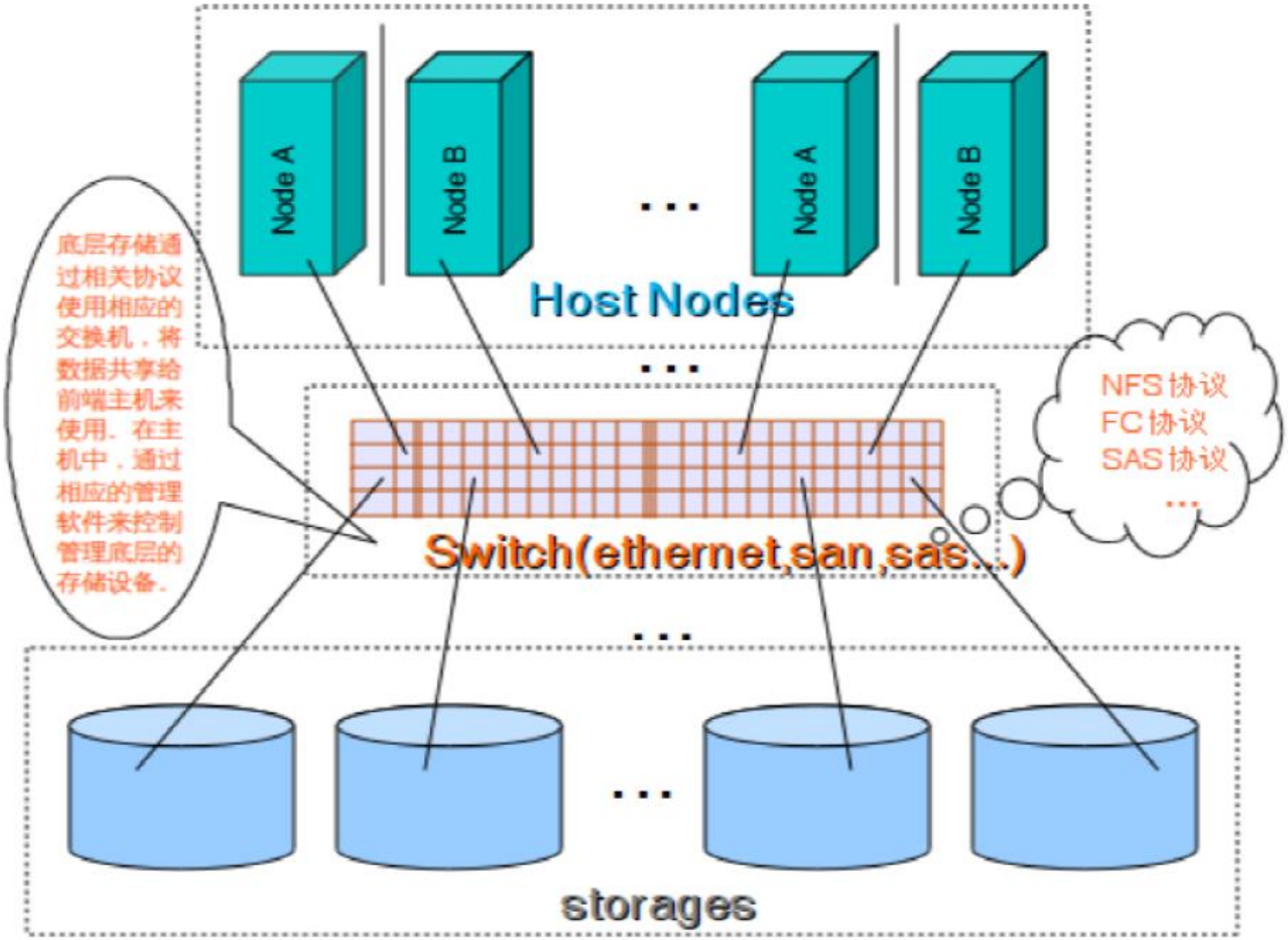
主机硬件高可用

- 硬件冗余（冷备/热备）
- 主机冗余、电源冗余、网络环境冗余...

数据高可用

- 基于共享数据存储的数据高可用
  - SAN、NAS、iScsi 、SAS…
- 基于数据库软件的数据复制冗余
  - MySQL Replication、Oracle Data Guard ...
- 基于第三方（或自行设计）的数据复制冗余
  - Tungeten、DBMoto、MMM ...

SHARE STORAGE



REPLICATION

REPLICATION

通过 MySQL 的 Replication，可以将一台 MySQL 中的数据完整的复制到多台主机上面的 MySQL 数据库中，并且正常情况下这种复制的延时并不是很长。当我们各台服务器上面都有同样的数据之后，应用访问就不再只能到一台数据库主机上面读取数据了，而是访问整个 MySQL 集群中的任何一台主机上面的数据库。

整个复制过程实际上就是Slave从Master端获取该日志然后再在自己身上完全顺序的执行日志中所记录的各种操作。

REPLICATION 机制的实现原理

- MySQL的 Replication 是一个异步的复制过程，从一个 MySQL instance（我们称之为 Master）复制到另一个 MySQL instance（我们称之 Slave）。在 Master 与 Slave 之间的实现整个复制过程主要由三个线程来完成，其中两个线程（SQL线程和IO线程）在 Slave 端，另外一个线程（IO线程）在 Master 端。

- 首先必须打开 Master 端的Binary Log（mysql-bin.xxxxxx）功能

MYSQL 复制的基本过程

1. Slave 上面的IO线程连接上 Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；
2. Master 接收到来自 Slave 的 IO 线程的请求后，通过负责复制的 IO 线程根据请求信息读取指定日志指定位置之后的日志信息，返回给 Slave 端的 IO 线程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息在 Master 端的 Binary Log 文件的名称以及在 Binary Log 中的位置；
3. Slave 的 IO 线程接收到信息后，将接收到的日志内容依次写入到 Slave 端的 Relay Log文件(mysql-relay-bin.xxxxxx)的最末端，并将读取到的Master端的 bin-log的文件名和位置记录到master-info文件中，以便在下一次读取的时候能够清楚的告诉Master “我需要从某个bin-log的哪个位置开始往后的日志内容，请发给我”
4. Slave 的 SQL 线程检测到 Relay Log 中新增加了内容后，会马上解析该 Log 文件中的内容成为在 Master 端真实执行时候的那些可执行的 Query 语句，并在自身执行这些 Query。这样，实际上就是在 Master 端和 Slave 端执行了同样的 Query，所以两端的数据是完全一样的。

复制实现级别

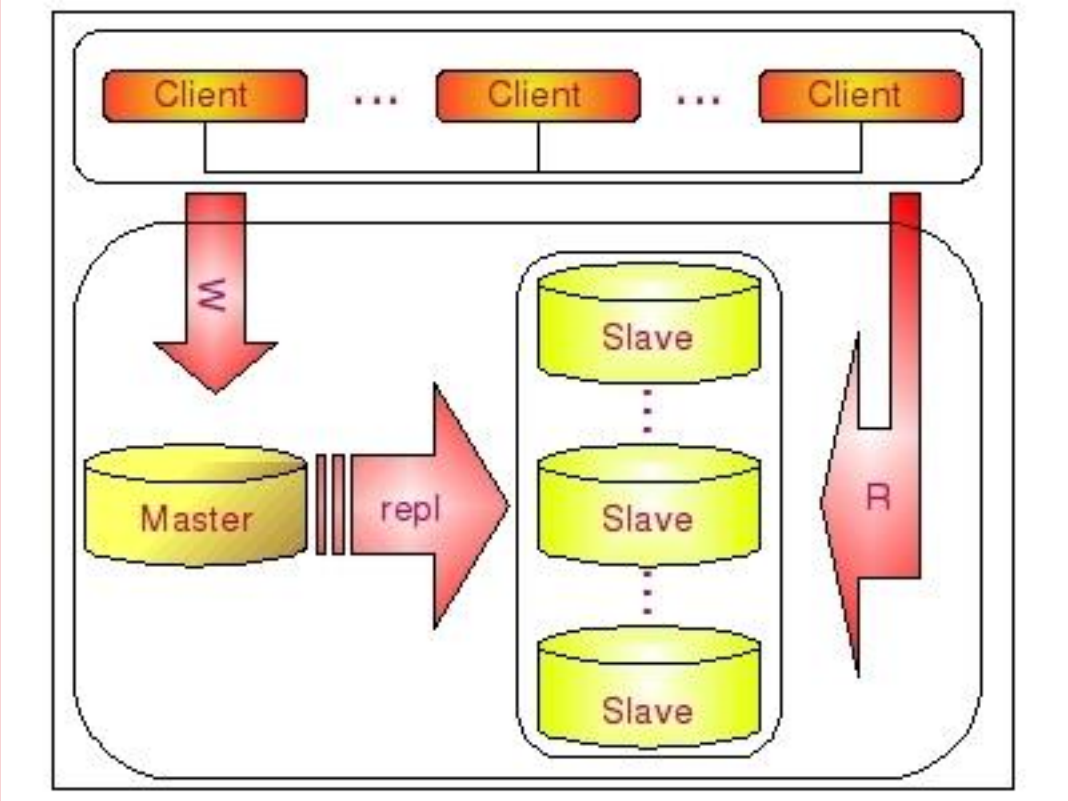
- **Row Level：Binary Log 中会记录成每一行数据被修改的形式，然后在 Slave 端再对相同的数据进行修改。**
  - 优点：在 Row Level 模式下，Binary Log 中可以不记录执行的sql语句的上下文相关的信息，仅仅只需要记录那一条记录被修改了，修改成什么样了。所以 Row Level 的日志内容会非常清楚的记录下每一行数据修改的细节，非常容易理解。而且不会出现某些特定情况下的存储过程，或function，以及trigger的调用和触发无法被正确复制的问题。
  - 缺点：Row Level下，所有的执行的语句当记录到 Binary Log 中的时候，都将以每行记录的修改来记录，这样可能会产生大量的日志内容，比如有这样一条update语句：UPDATE group_message SET group_id = 1 where group_id = 2，执行之后，日志中记录的不是这条update语句所对应的事件（MySQL以事件的形式来记录 Binary Log 日志），而是这条语句所更新的每一条记录的变化情况，这样就记录成很多条记录被更新的很多个事件。自然，Binary Log 日志的量就会很大。尤其是当执行ALTER TABLE 之类的语句的时候，产生的日志量是惊人的。因为MySQL对于 ALTER TABLE 之类的 DDL 变更语句的处理方式是重建整个表的所有数据，也就是说表中的每一条记录都需要变动，那么该表的每一条记录都会被记录到日志中。

复制实现级别

- **Statement Level:每一条会修改数据的 Query 都会记录到 Master的 Binary Log 中。Slave在复制的时候 SQL 线程会解析成和原来 Master 端执行过的相同的 Query 来再次执行。**
  - 优点：Statement Level下的优点首先就是解决了Row Level下的缺点，不需要记录每一行数据的变化，减少 Binary Log 日志量，节约了 IO 成本，提高了性能。因为他只需要记录在Master上所执行的语句的细节，以及执行语句时候的上下文的信息。
  - 缺点：由于他是记录的执行语句，所以，为了让这些语句在slave端也能正确执行，那么他还必须记录每条语句在执行的时候的一些相关信息，也就是上下文信息，以保证所有语句在slave端杯执行的时候能够得到和在master端执行时候相同的结果。另外就是，由于Mysql现在发展比较快，很多的新功能不断的加入，使mysql得复制遇到了不小的挑战，自然复制的时候涉及到越复杂的内容，bug也就越容易出现。在statement level下，目前已经发现的就有不少情况会造成mysql的复制出现问题，主要是修改数据的时候使用了某些特定的函数或者功能的时候会出现，比如：sleep()函数在有些版本中就不能准确复制，在存储过程中使用了last_insert_id()函数，可能会使slave和master上得到不一致的id等等。由于row level是基于每一行来记录的变化，所以不会出现类似的问题。

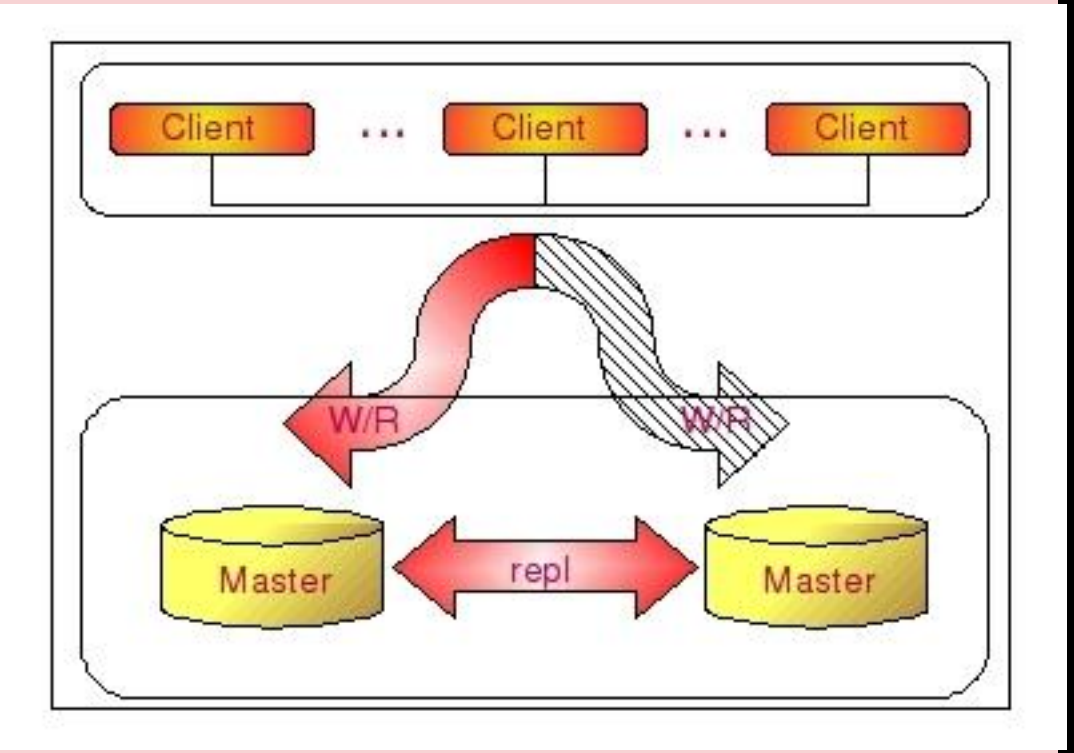
常规复制架构(MASTER - SLAVES)

对于数据实时性要求不是特别 Critical 的应用，只需要通过廉价的pc server来扩展 Slave 的数量，将读压力分散到多台 Slave 的机器上面，即可通过分散单台数据库服务器的读压力来解决数据库端的读性能瓶颈



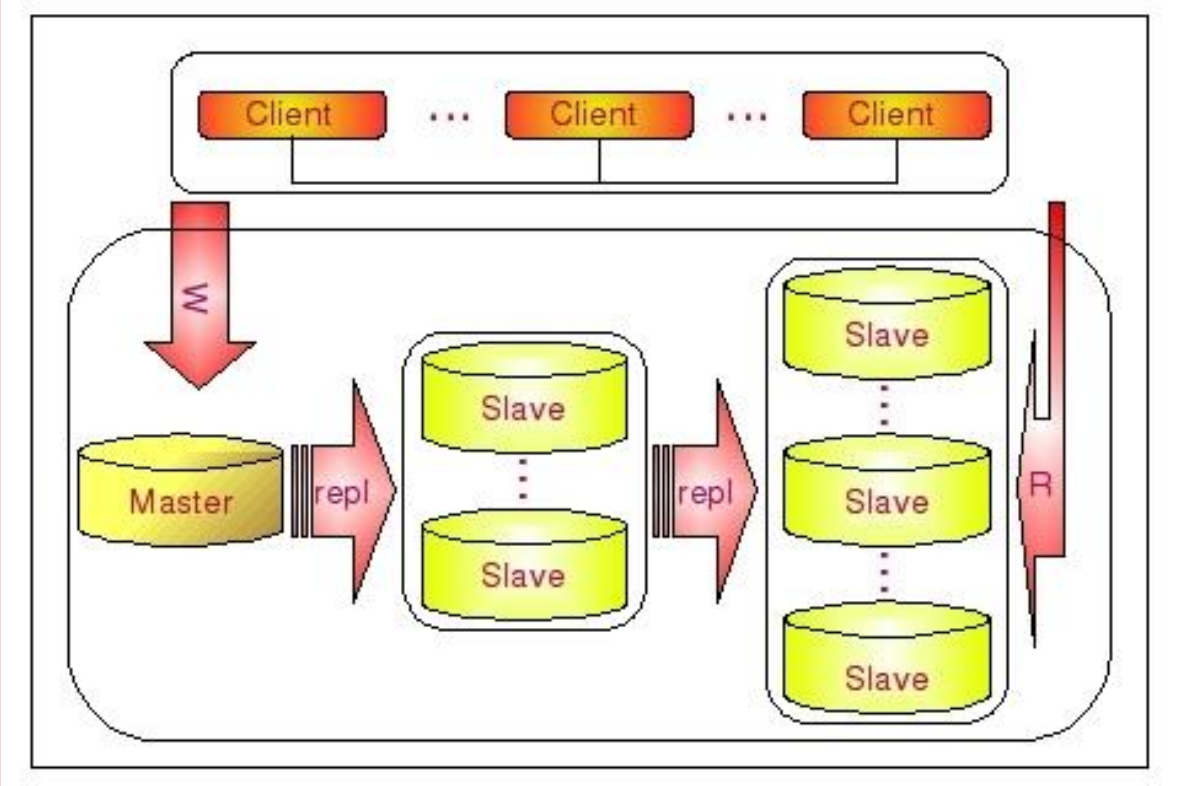
DUAL MASTER 复制架构(MASTER - MASTER)

Dual Master 环境就是两个 MySQL Server 互相将对方作为自己的 Master，自己作为对方的 Slave 来进行复制。这样，任何一方所做的变更，都会通过复制应用到另外一方的数据库中。



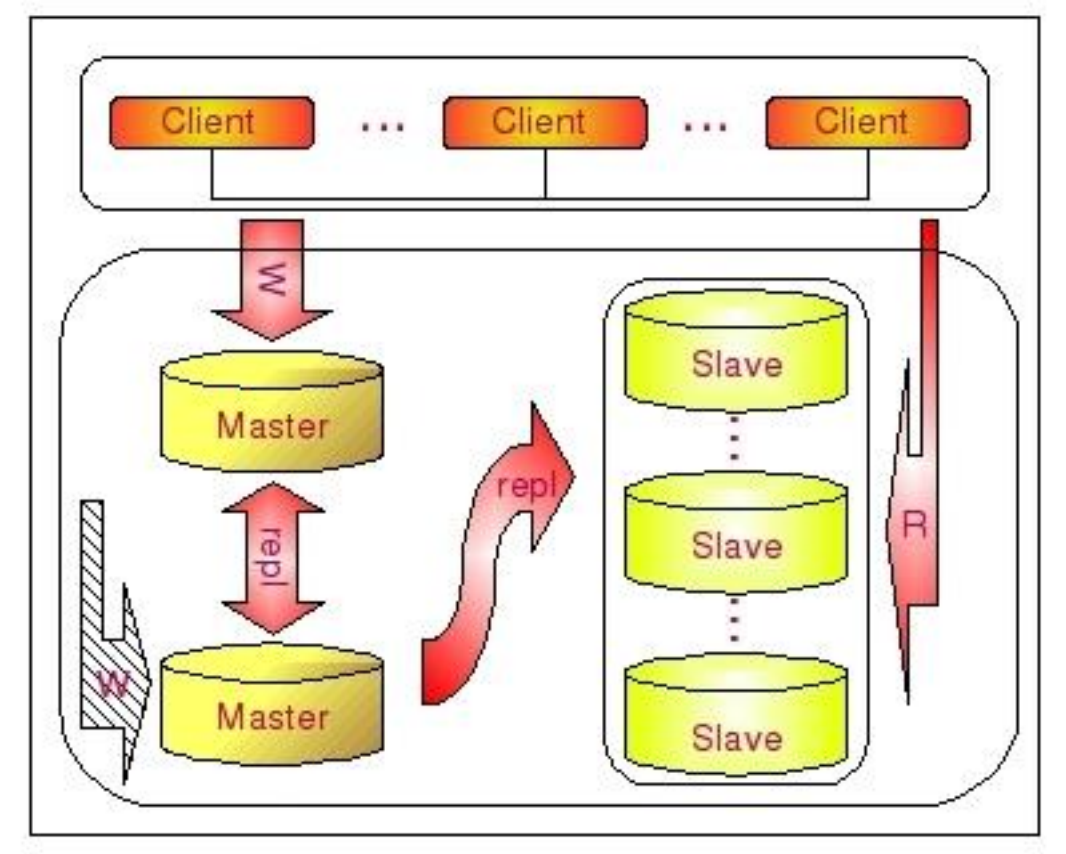
级联复制架构(MASTER - SLAVES - SLAVES ...)

有些应用场景中，可能读写压力差别比较大，读压力特别的大，一个 Master 可能需要上10台甚至更多的 Slave 才能够支撑注读的压力。复制就会消耗较多的资源，很容易造成复制的延时。



DUAL MASTER 与级联复制结合架构(MASTER - MASTER - SLAVES)

MMS同时解决 Master 因为所附属的 Slave 过多而成为瓶颈的问题，及解决人工维护和出现异常需要切换后可能存在重新搭建 Replication 的问题。



CLUSTER

MYSQL CLUSTER
SQL服务器节点

MySQL Cluster实际上是在无共享存储设备的情况下实现的一种完全分布式数据库系统，其主要通过NDB Cluster（简称NDB）存储引擎来实现。

MySQL Cluster的环境由以下三部分组成：

•SQL层的SQL服务器节点（后面简称为SQL节点），也就是我们常说的MySQL Server。

- 主要负责实现一个数据库在存储层之上的所有事情，比如连接管理，Query 优化和响应，Cache 管理等等，只有存储层的工作交给了 NDB 数据节点去处理了。也就是说，在纯粹的MySQL Cluster环境中的SQL节点，可以被认为是一个不需要提供任何存储引擎的MySQL服务器，因为他的存储引擎有Cluster环境中的 NDB 节点来担任。所以，SQL层各MySQL服务器的启动与普通的 MySQL Server 启动也有一定的区别，必须要添加ndbcluster参数选项才行。我们可以添加在my.cnf配置文件中，也可以通过启动命令行来指定。

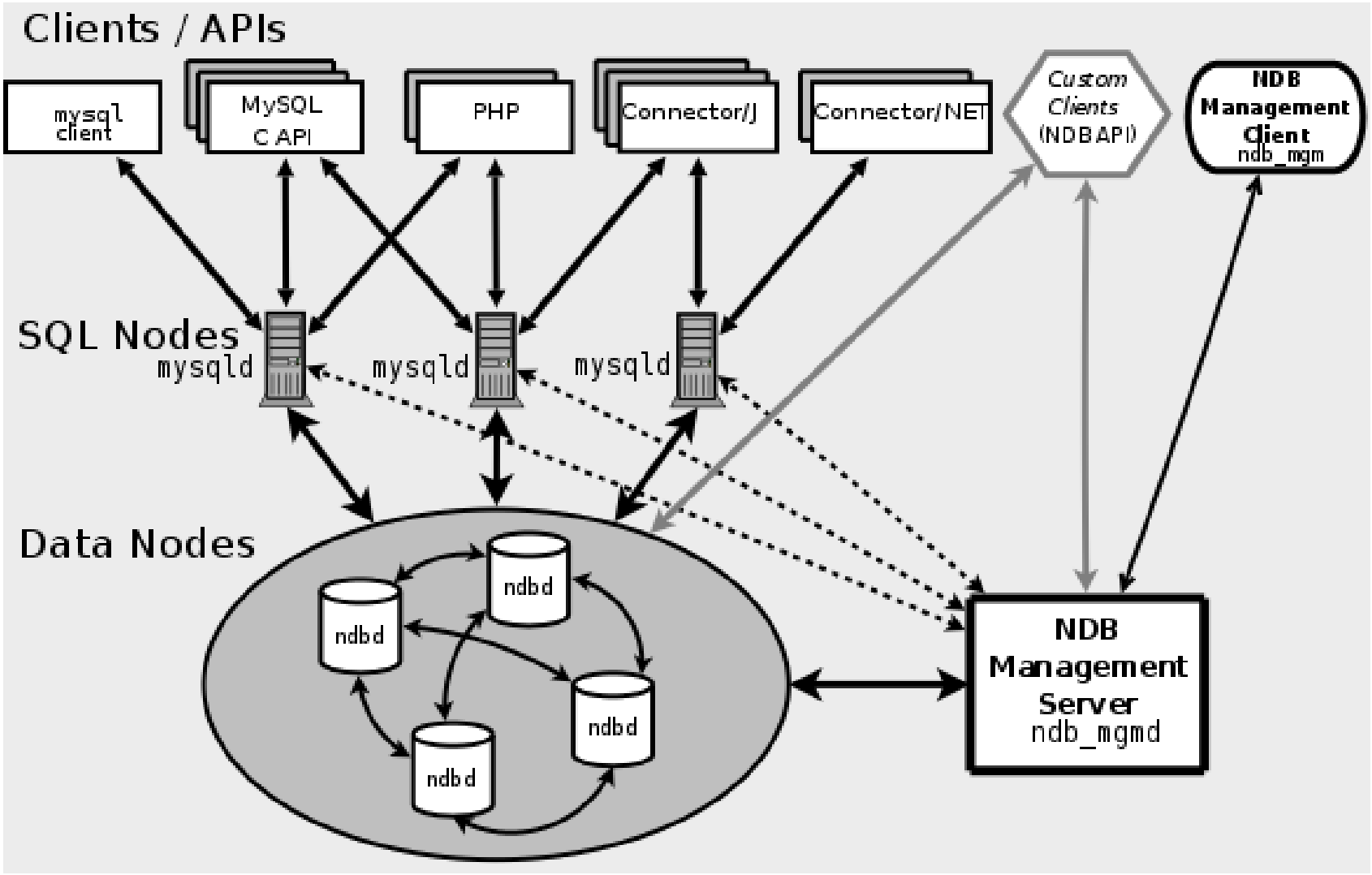
NDB 数据节点

- **Storage 层的 NDB 数据节点，也就是NDB Cluster。**
  - 最初 NDB 是一个内存式存储引擎，当然也会将数据持久化到存储设备上。最新的 NDB Cluster 存储引擎已经改进了这一点，可以选择数据是全部加载到内存中还是仅仅加载索引数据。
  - NDB 节点主要是实现底层数据存储功能，来保存Cluster的数据。每一个Cluster节点保存完整数据的一个fragment，也就是一个数据分片（或者一份完整的数据，视节点数目和配置而定），所以只要配置得当，MySQL Cluster在存储层不会出现单点的问题。
  - 一般来说，NDB 节点被组织成一个一个的NDB Group，一个NDB Group实际上就是一组存有完全相同的物理数据的NDB节点群。

MANAGE节点

- **负责管理各个节点的Manage节点主机：**
  - 管理节点负责整个Cluster集群中各个节点的管理工作，包括集群的配置，启动关闭各节点，对各个节点进行常规维护，以及实施数据的备份恢复等。
  - 管理节点会获取整个Cluster环境中各节点的状态和错误信息，并且将各Cluster集群中各个节点的信息反馈给整个集群中其他的所有节点。
  - 由于管理节点上保存了整个Cluster环境的配置，同时担任了集群中各节点的基本沟通工作，所以他必须是最先被启动的节点。

架构图


Clients / APIs
mysql client
MySQL C API
PHP
Connector/J
Connector/NET
Custom Clients (NDBAPI)
NDB Management Client ndb_mgm
SQL Nodes
mysqld
mysqld
mysqld
Data Nodes
ndbd
ndbd
ndbd
ndbd
NDB Management Server ndb_mgmd

可用性方案讨论

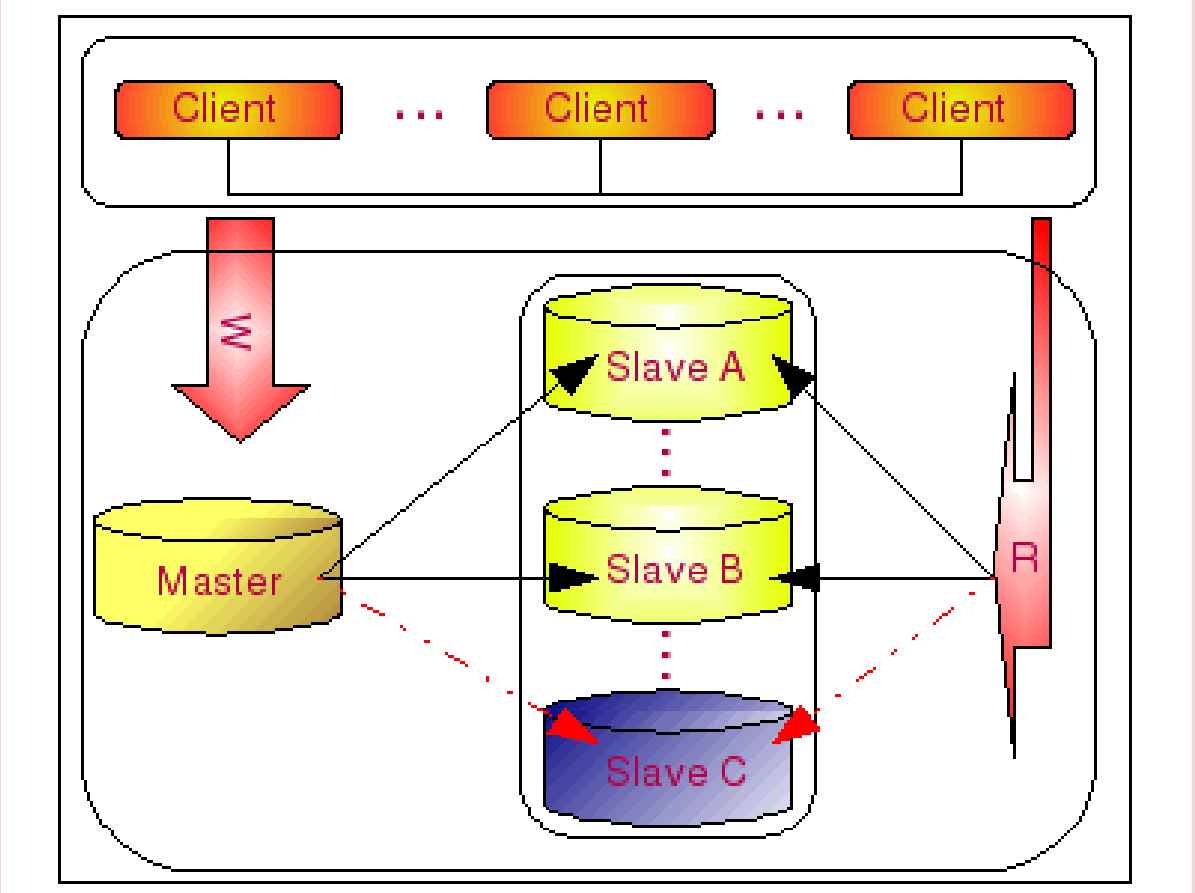
基于数据库复制的可用性分析

常规的 MASTER - SLAVE 解决基本的主备设计

- **普通的 Master - Slave 架构是目前很多系统中使用最为常见的一种架构方式。该架构设计不仅仅在很大程度上解决的系统的扩展性问题，带来性能的提升，同时在系统可用性方面也提供了一定的保证。**
- **在普通的一个 Master 后面复制一个或者多个 Slave 的架构设计中，当我们的某一台 Slave 出现故障不能提供服务之后，我们还有至少一台 MySQL 服务器（Master）可以提供服务，不至于所有和数据库相关的业务都不能运行下去。如果 Slave 超过一台，那么剩下的 Slave 也仍然能够不受任何干扰的继续提供服务。**
  - 我们的 MySQL 数据库集群整体的服务能力要至少能够保证当其缺少一台 MySQL Server 之后还能够支撑系统的负载，否则一切都是空谈。

M-S复制——S故障的情况

当我们的 Slave 集群中的一台 Slave C 出现故障 crash 之后，整个系统的变化仅仅只是从 Master 至 Slave C 的复制中断，客户端应用的 Read 请求也不能再访问 Slave C。当时其他所有的 MySQL Server 在不需要任何调整的情况下就能正常工作。客户端的请求 Read 请求全部由 Slave A 和 Slave B 来承担。



MASTER 单点问题的解决

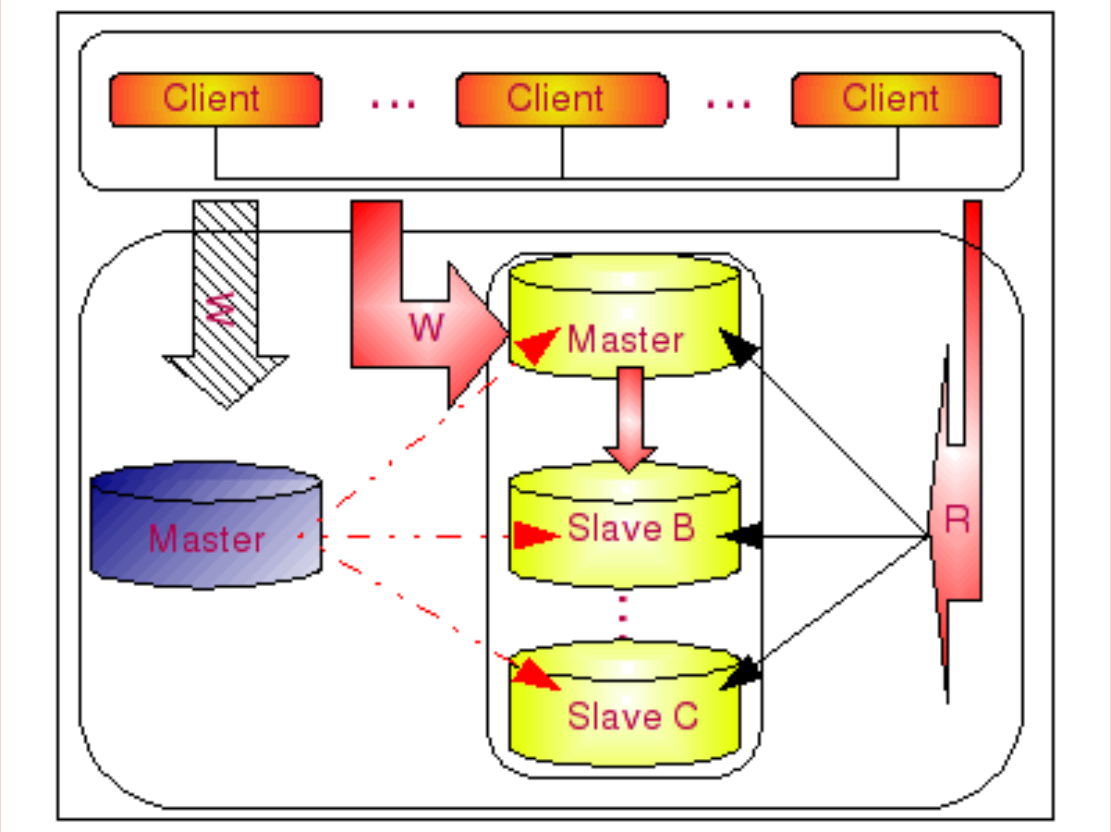
- **当 Master 出现问题后所有客户端的 Write 请求就无法处理**

- **这时可以有如下两种解决方案：**
  - 一个是将 Slave 中的某一台切换成 Master 对外提供服务，同时将其他所有的 Slave 都以通过 CHANGE MASTER 命令来将通过新的 Master 进行复制。
  - 另一个方案就是新增一台 Master，也就是 Dual Master 的解决方案。

M-S复制——M故障的情况

第一种解决方案，将一台 Slave 切换成 Master 来解决问题

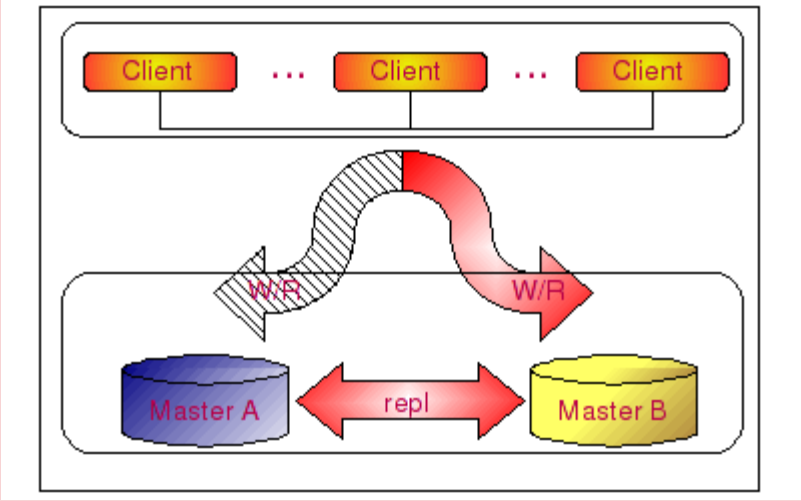
当 Master 出现故障 crash 之后，原客户端对 Master 的所有 Write 请求都会无法再继续进行下去了，所有原 Master 到 Slave 的复制也自然就断掉了。这时候，我们选择一台 Slave 将其切换成 Master。假设选择的是 Slave A，则我们将其他 Slave B 和 Slave C 都通过 CHANGE MASTER TO 命令更换其 Master，从新的 Master 也就是原Slave A 开始继续进行复制。同时将应用端所有的写入请求转向到新的 Master。对于 Read 请求，我们可以去掉对新 Master 的 Read 请求，也可以继续保留。



通过 DUAL MASTER 来解决 MASTER 的单点问题

我们通过两台 MySQL Server 搭建成 Dual Master 环境，正常情况下，所有客户端的 Write 请求都写往 Master A，然后通过 Replication 将 Master A 复制到 Master B。一旦 Master A 出现问题之后，所有的 Write 请求都转向 Master B。而在正常情况下，当 Master B 出现问题的时候，实际上不论是数据库还是客户端的请求，都不会受到实质性的影响。

特点就是在 Master 出现故障之后的处理比较简单，可控性比较大。而弊端就是需要增加一台 MySQL 服务器，在成本方面投入更大。

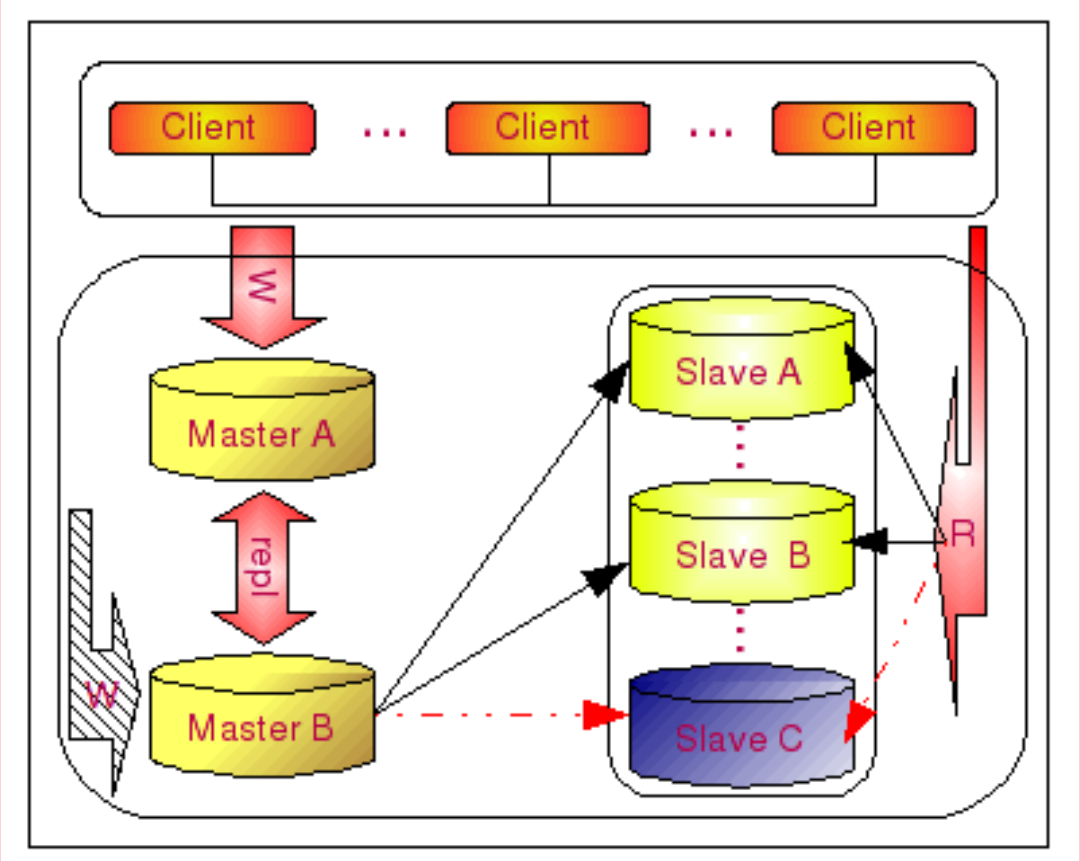


M-M-S复制——S故障的情况

考虑 Slave 出现异常的情况。

在这个架构中，Slave 出现异常后的处理情况和普通的 Master - Slave 架构的处理方式完全一样，仅仅需要在应用访问 Slave 集群的访问配置中去掉一个 Slave 节点即可解决

不论是通过应用程序自己判断，还是通过硬件解决方案，都可以很容易的实现。

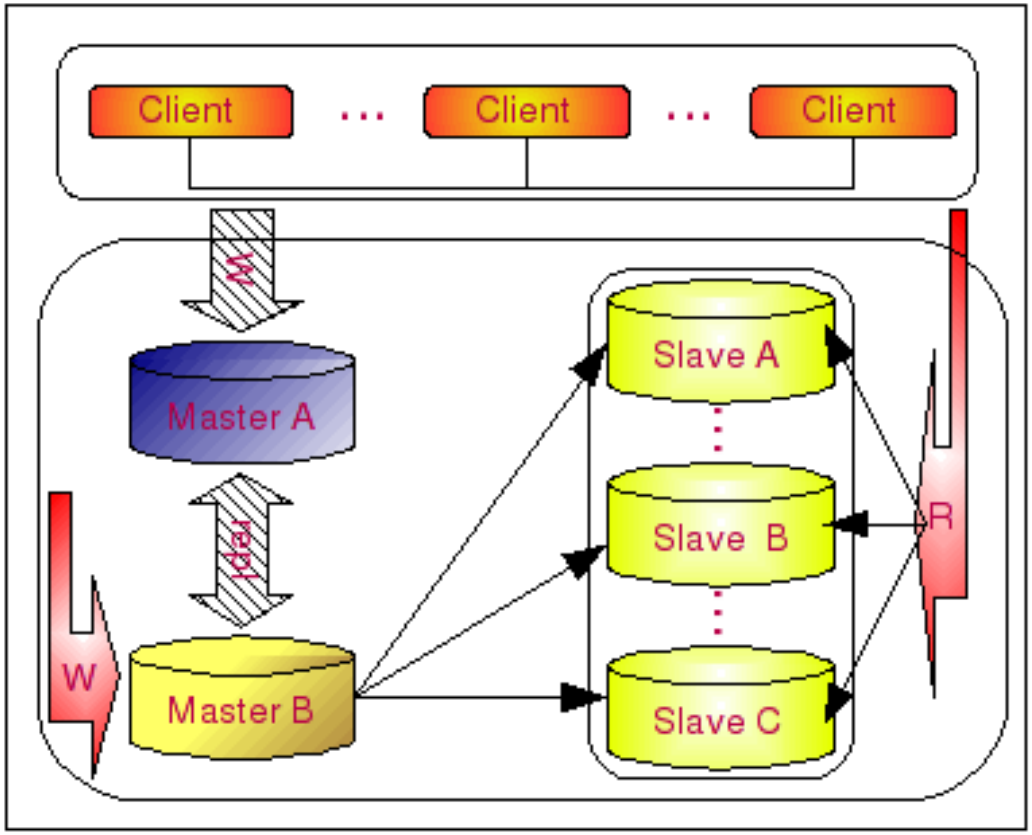


M-M-S复制——当 MASTER A 出现故障之后的处理方案

当 Master A 出现故障 crash 之后，Master A 与 Master B 之间的复制将中断，所有客户端向 Master A 的 Write 请求都必须转向 Master B。

这个转向动作的实现，可以通过VIP 的方式实现。

由于之前所有的 Slave 就都是从 Master B 来实现复制，所以 Slave 集群不会受到任何的影响，客户端的所有 Read 请求也就不会受到任何的影响，整个过程可以完全自动进行，不需要任何的人为干预。

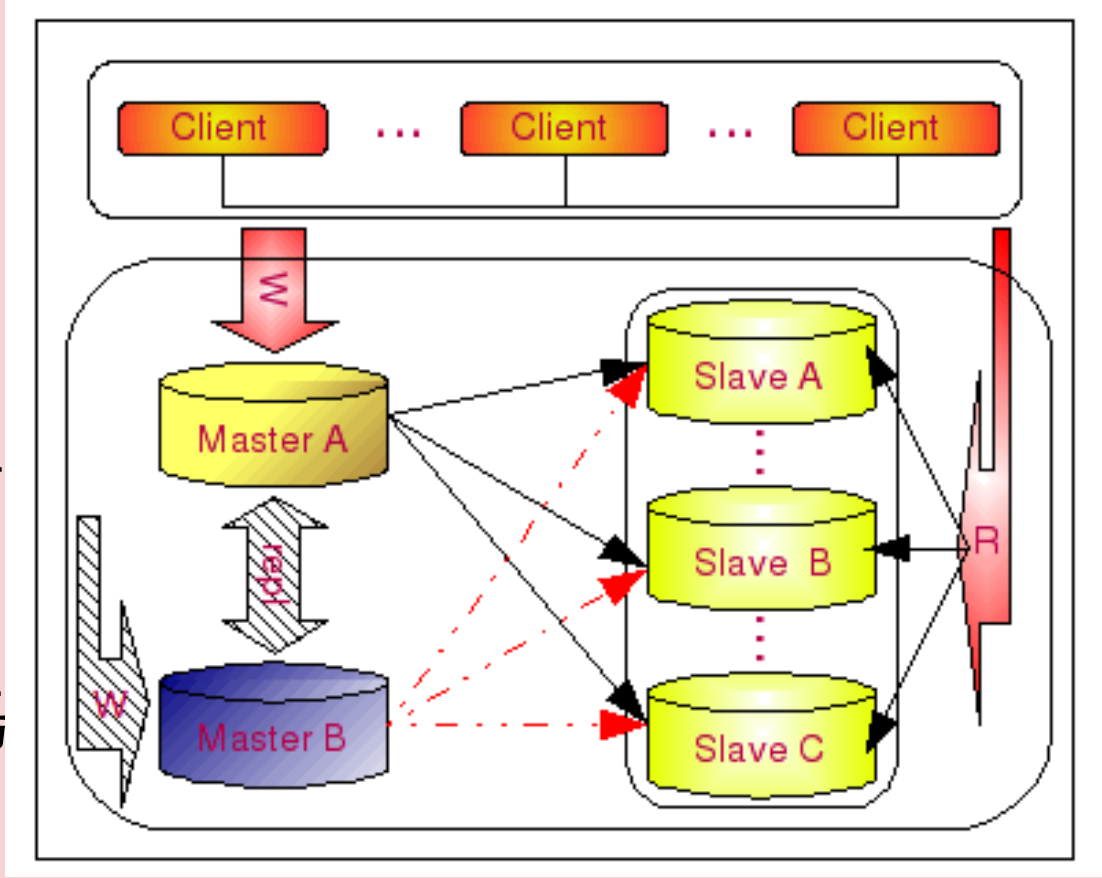


M-M-S复制——当 MASTER B 出现故障之后的情况又如何

首先可以确定的是我们的所有 Write 请求都不会受到任何影响，而且所有的 Read 请求也都还是能够正常访问。

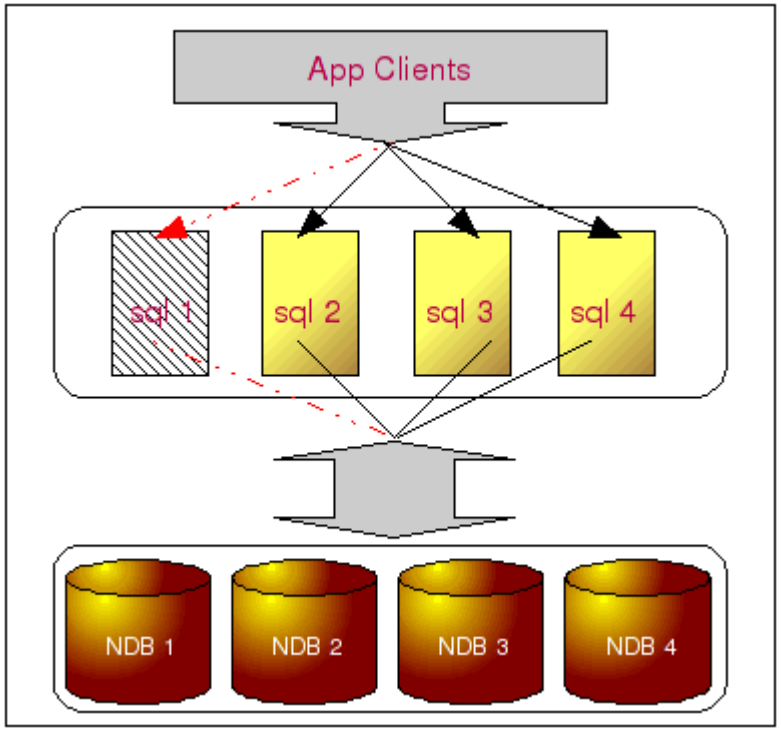
但所有 Slave 的复制都会中断，Slave 上面的数据会开始出现滞后的现象。

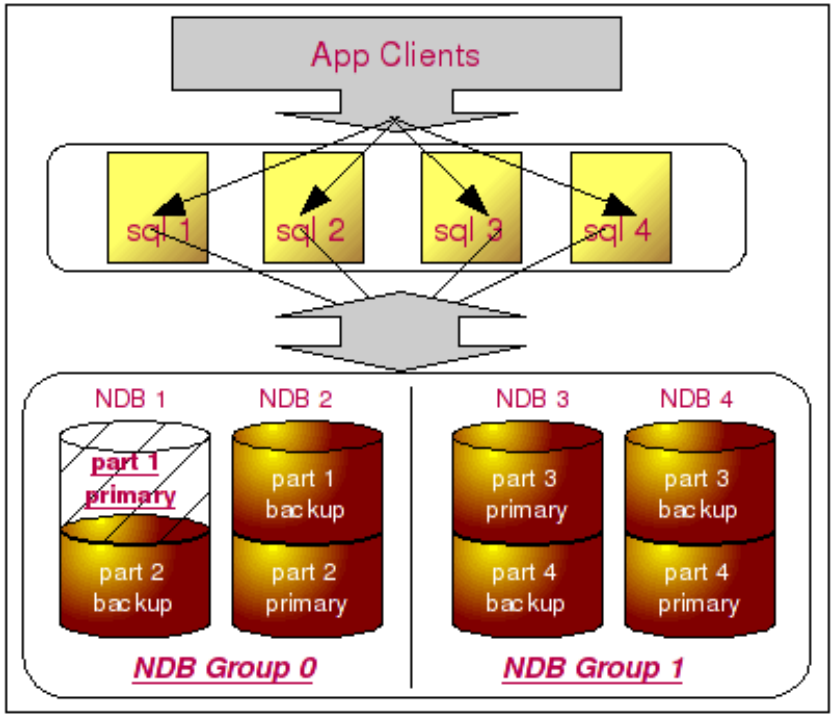
需要做的就是将所有的 Slave 进行 CHANGE MASTER TO 操作，改为从 Master A 进行复制。由于所有 Slave 的复制都不可能超前最初的数据源，所以可以根据 Slave 上面的 Relay Log 中的时间戳信息与 Master A 中的时间戳信息进行对照来找到准确的复制起始点，不会造成任何的数据丢失。



利用 MYSQL CLUSTER 实现整体高可用

- MySQL Cluster 是一个完整的分布式架构的系统，而且支持数据的多点冗余存放，数据实时同步等特性。
- 由于 MySQL Cluster 的架构主要由两个层次两组集群来组成，包括 SQL 节点（mysqld） 和NDB 节点（数据节点），所有两个层次都需要能够保证高可用性才能保证整体的可用性。





讨论-一致性的代价

- Synchronous replication
  - Transactions are not committed until data is replicated and applied
  - Provides consistency, but slower
  - Provided by MySQL Cluster
- Asynchronous replication
  - Transactions committed immediately and replicated
  - No consistency, but faster
  - Provided by MySQL Server

架构设计理论与原则

- 网站架构设计的精神食粮

架构设计理论与原则

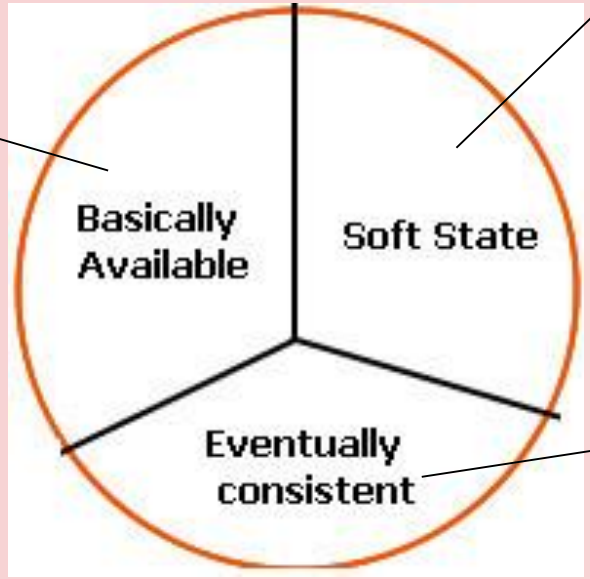
■关于数据一致性—ACID vs BASE

■ACID（Atomicity、 Consistency、 Isolation、 Durability）是关系型数据库的最基本原则，遵循ACID原则强调一致性，对成本要求很高，对性能影响很大。

■问题：ACID原则适用于互联网应用吗？可用性似乎比一致性重要些

■BASE（**B**asically **A**vailable、 **S**oft state、 **E**ventually consistent）策略

基本可用
数据能够保证80%一致性就够了，剩下20%就不要过于纠结了。可参考八二定律



软状态
可以理解为无连接的，而Hard state是面向连接的。Soft state可以有一段时间不同步，异步。

最终一致性
在某一段短时间内允许数据不一致，但经过一段较长时间，等所有节点上数据的拷贝都整合在一起的时候，数据会最终达到完全一致

BASE策略与ACID不同，其基本思想就是通过牺牲强一致性，以获得更好的可用性或可靠性

架构设计理论与原则

■关于分布式系统—CAP理论

一致性
分布式系统中，数据一般会存储在不同节点，一致性就是要保证对数据操作的原子性

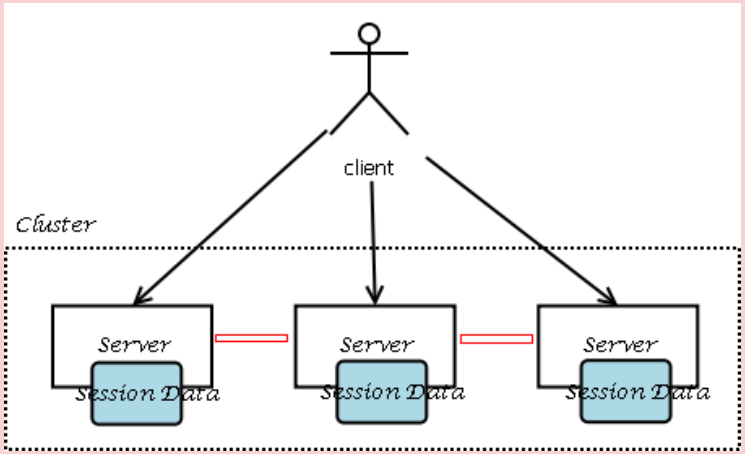
可用性
确保客户访问数据时可得到响应。不强调各个节点上数据要保持一致性。

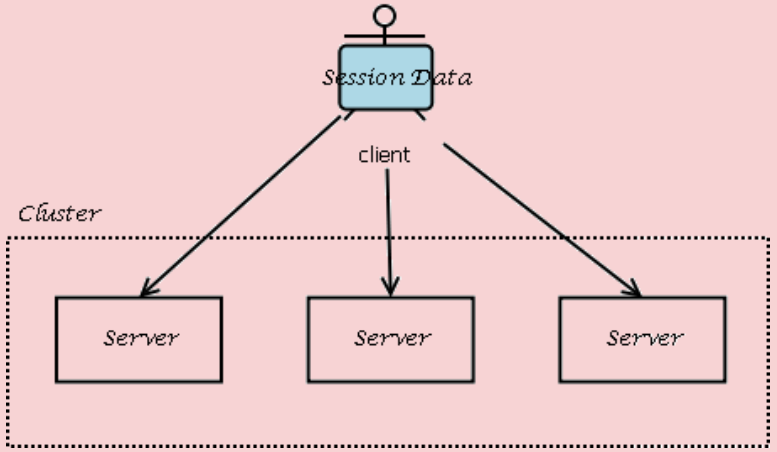
分区容忍性
数据分区存储后，即使部分分区组件不可用，其施加的操作也能够完成

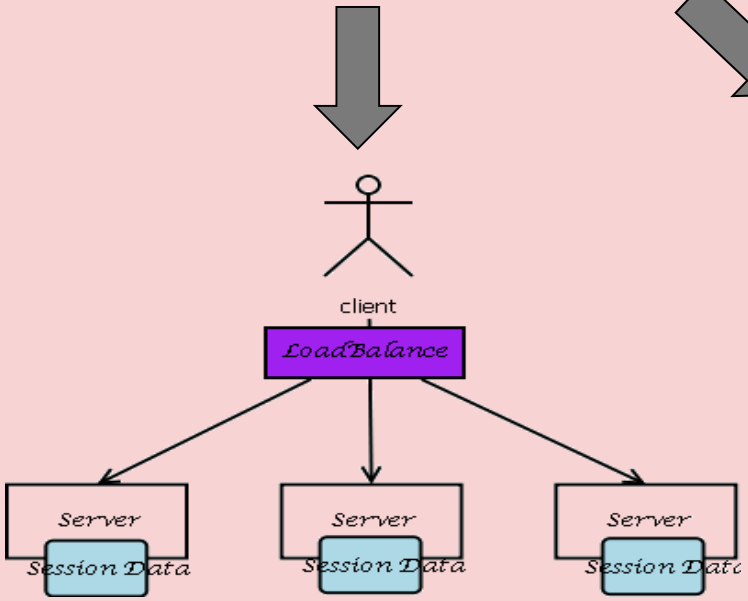
CAP理论指出：一个分布式系统不可能同时满足一致性、可用性和分区容忍性这三项需求，最多只能同时满足其中两个。

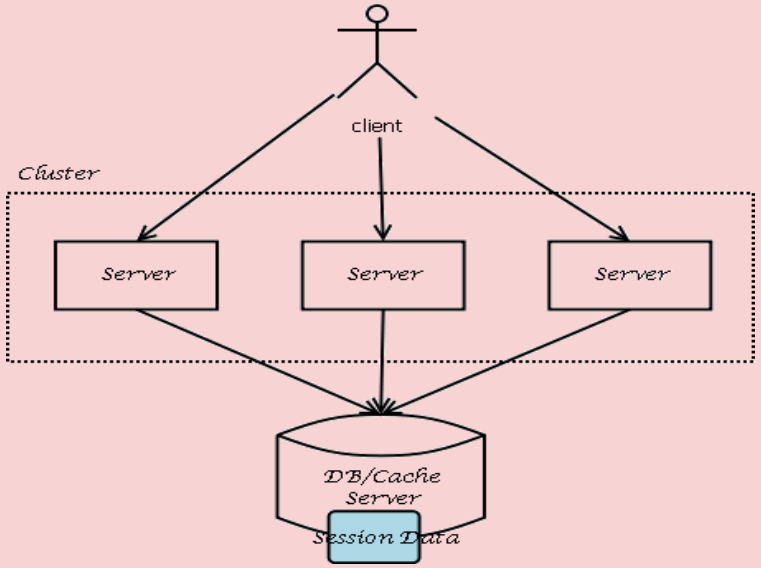
架构设计理论与原则

- 无共享架构（Share Nothing Architecture）



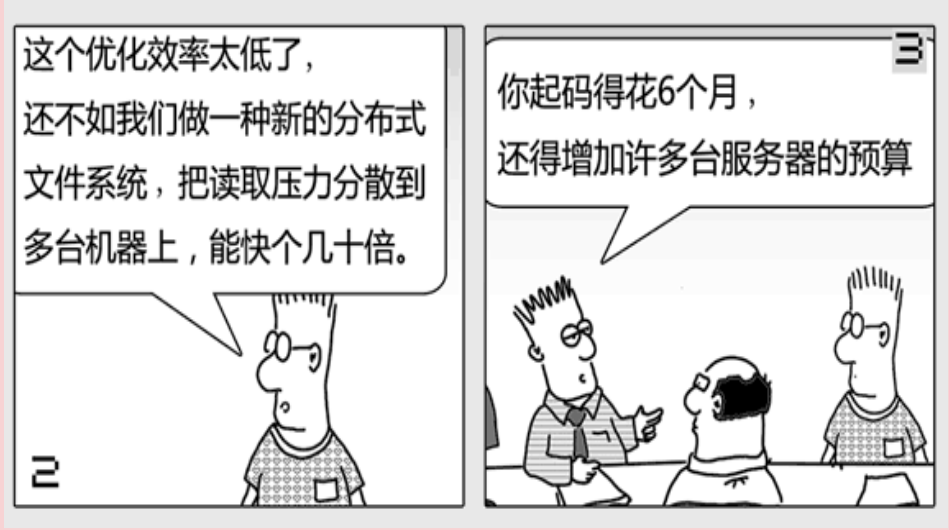






架构设计理论与原则

■**考量成本，先硬后软原则**



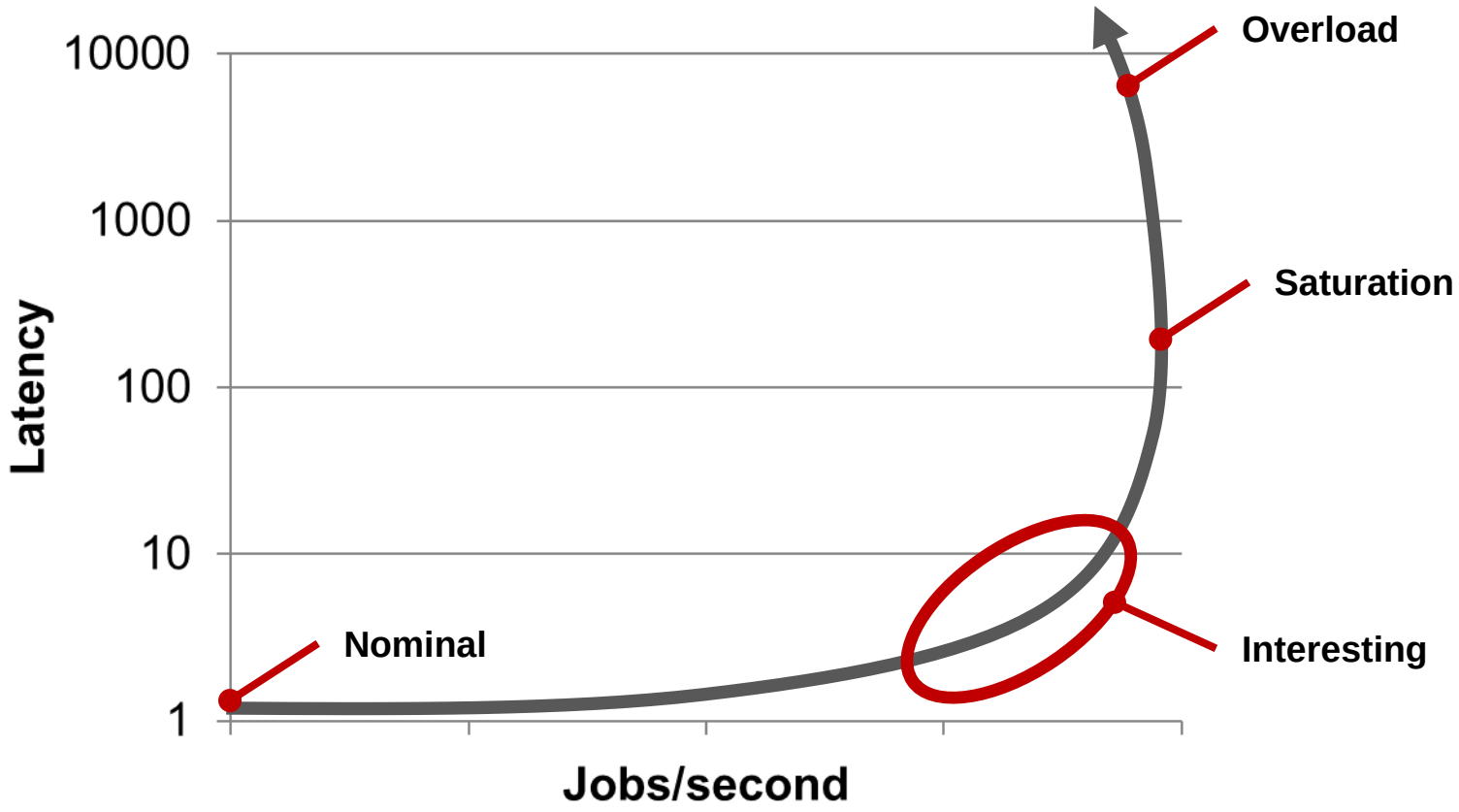
LOAD BALANCING

WHAT IS LOAD BALANCING?

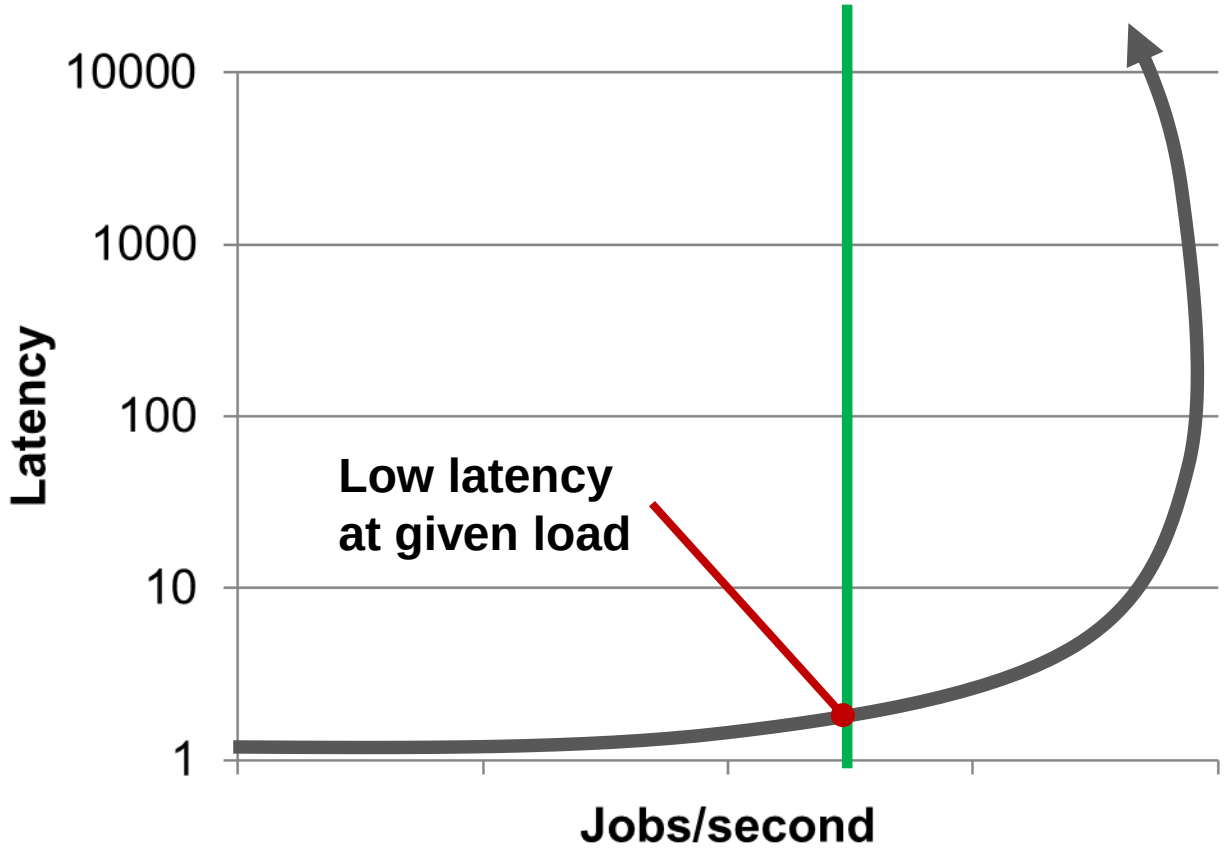
Wikipedia: "…methodology to distribute workload across multiple computers … to achieve optimal resource **utilization**, maximize **throughput**, minimize **response time**, and avoid **overload**"

All part of the latency curve

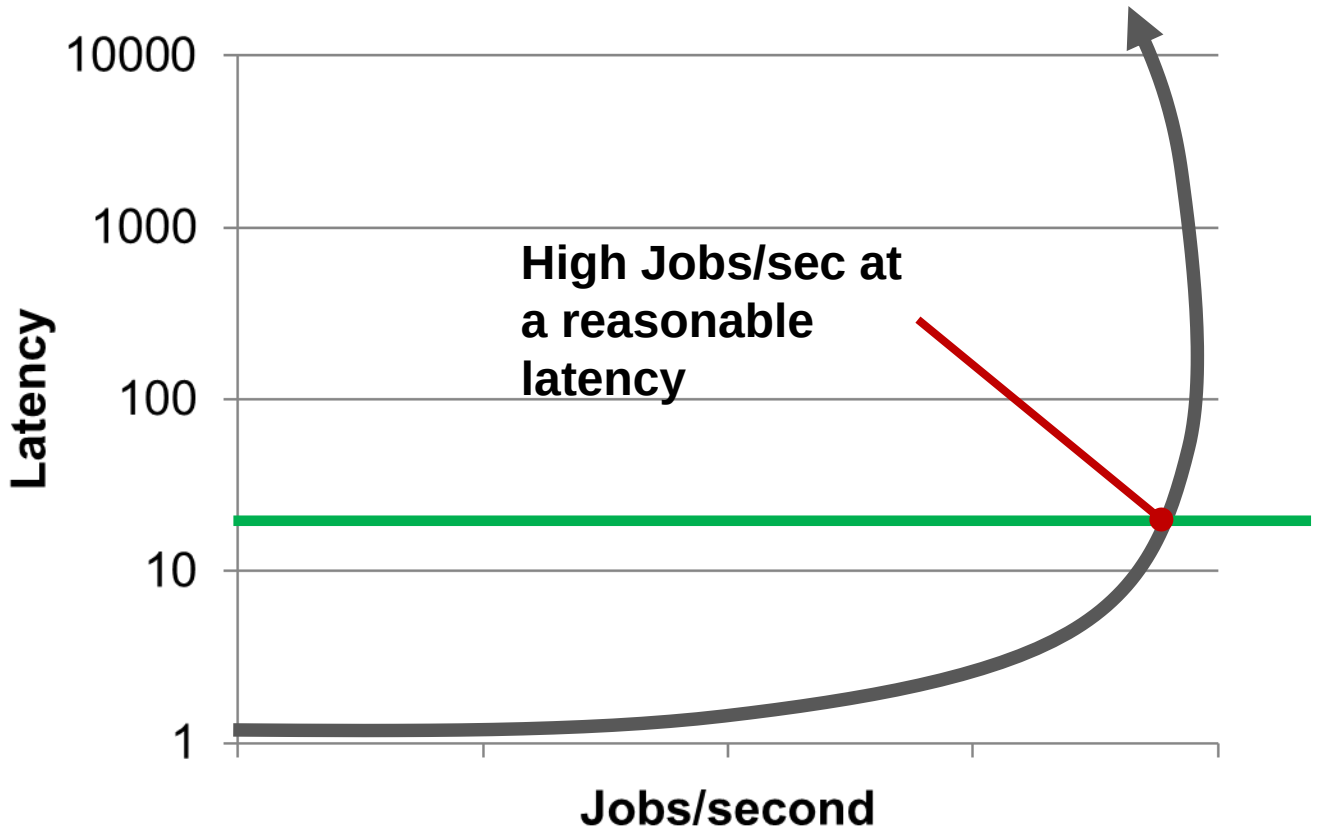
THE LATENCY CURVE



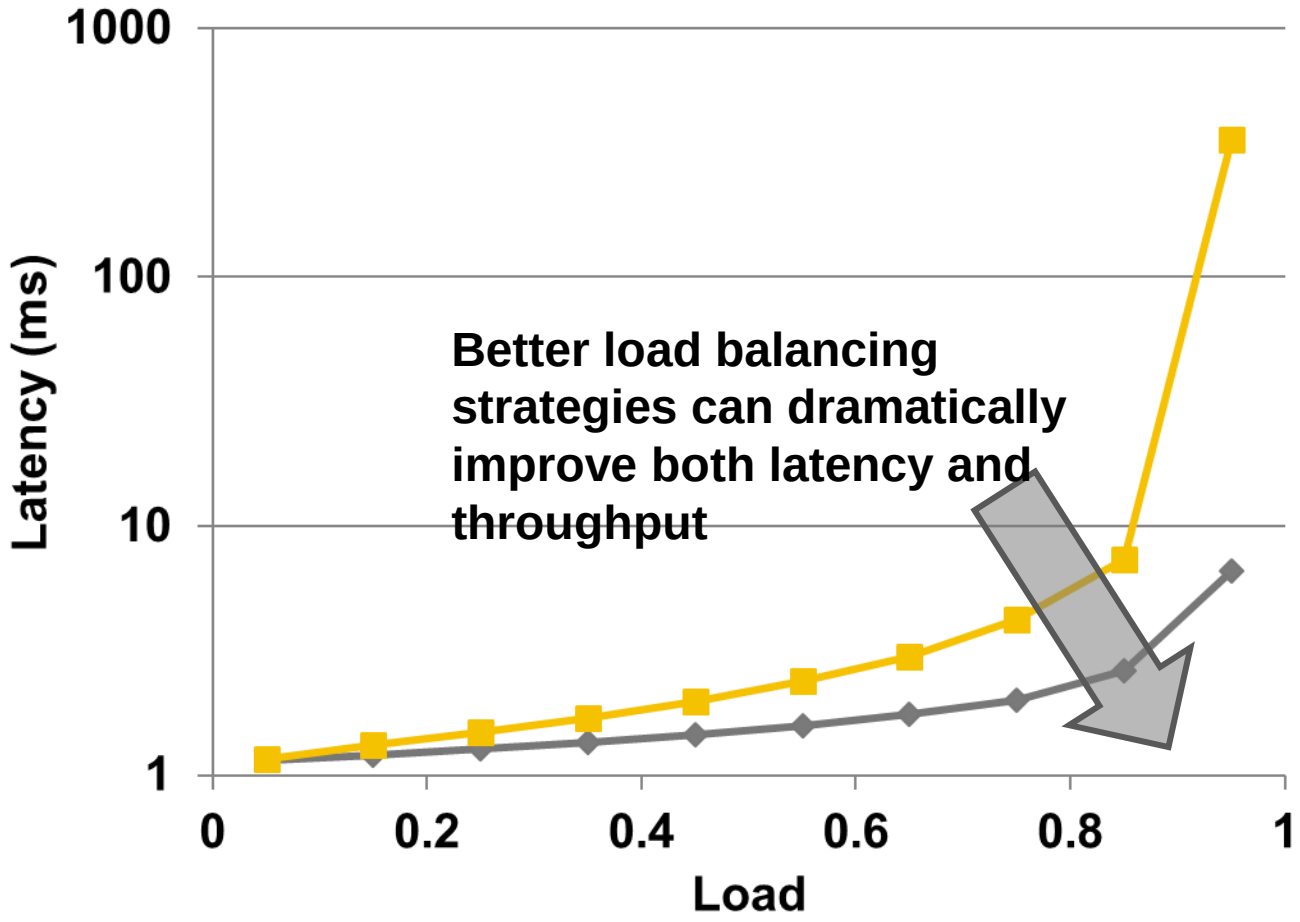
GOAL FOR REAL-TIME SYSTEMS



GOAL FOR BATCH SYSTEMS



THE LATENCY CURVE



LOAD BALANCING TENSIONS

- We want to reduce queue lengths in the system to yield better latency
- We want to lengthen queue lengths to keep a "buffer" of work to keep busy during irregular traffic and yield better throughput
- For distributed systems, equalizing queue lengths sounds good

INTRODUCTION

- Load Balancing is a technique to spread work between two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, and minimize response time.

- Load balancer is a tool that directs a client to the least busy or most appropriate Web server among several servers that contain mirrored contents.

WHY IS LOAD BALANCING OF SERVERS NEEDED?

- The web server may not be able to handle high volumes of incoming traffic.
- The users will have to wait until the web server is free to process their requests.
- There may be a situation where upgrading the server hardware will no longer be cost effective.

LOAD-BALANCING TECHNIQUES

- Three types of load balancers exist:
  - Hardware appliances
  - Network switches
  - Software

LOAD-BALANCING TECHNIQUES

- A hardware appliance-based load balancer is a closed box.
- A network switch-based load balancer uses a Layer2 or Layer3 switch to integrate the load-balancing service.
- A software load balancer is software which you can install on a dedicated server.

HOW TO ACHIVE LOAD BALANCING?

- More servers need to be added to distribute the load among the group of servers, which is also known as a server cluster.
- The load distribution among these servers is known as load balancing.
- Load balancing applies to all types of servers (application server, database server).

SOFTWARE LOAD BALANCERS

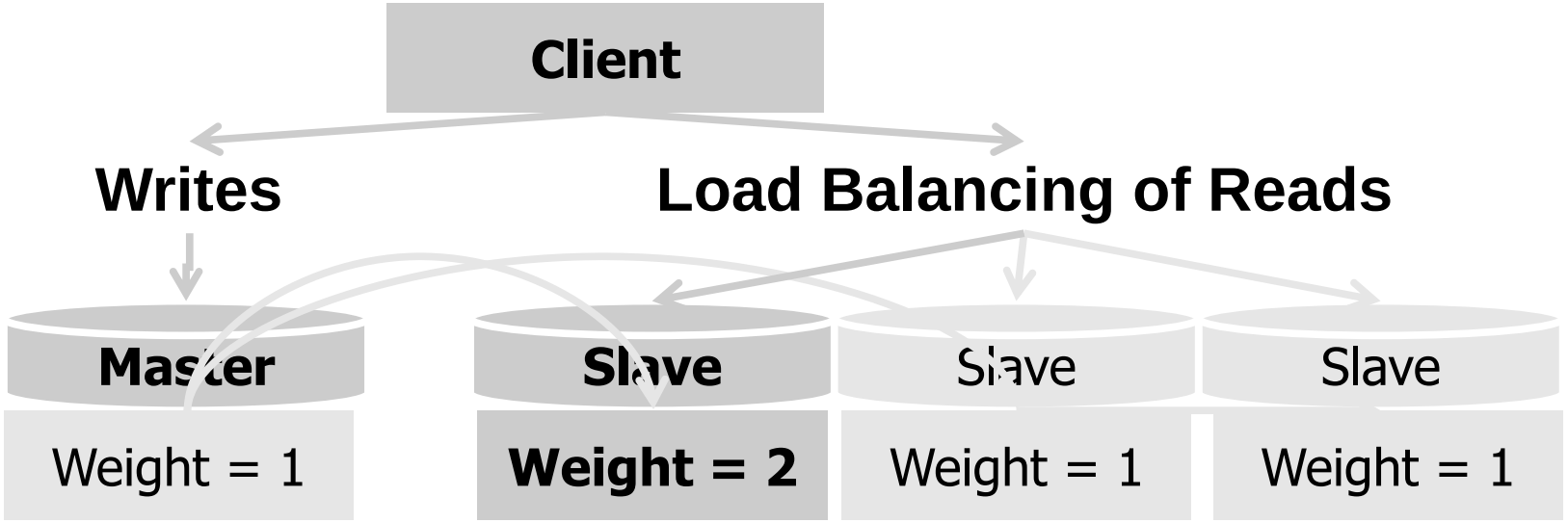
- Haproxy
- Pure Load Balancer (PLB)
- Perlbal
- Pound
- Pen

HAPROXY LOAD BALANCER

- HAProxy is a free, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications.

LOAD BALANCING

- Built-in or user-defined through callback
  - Round robin, Random, Random once
  - 1.3 – kind of adaptive by setting limits for slave lag
  - 1.4 – nodes can be assigned a weight/probability

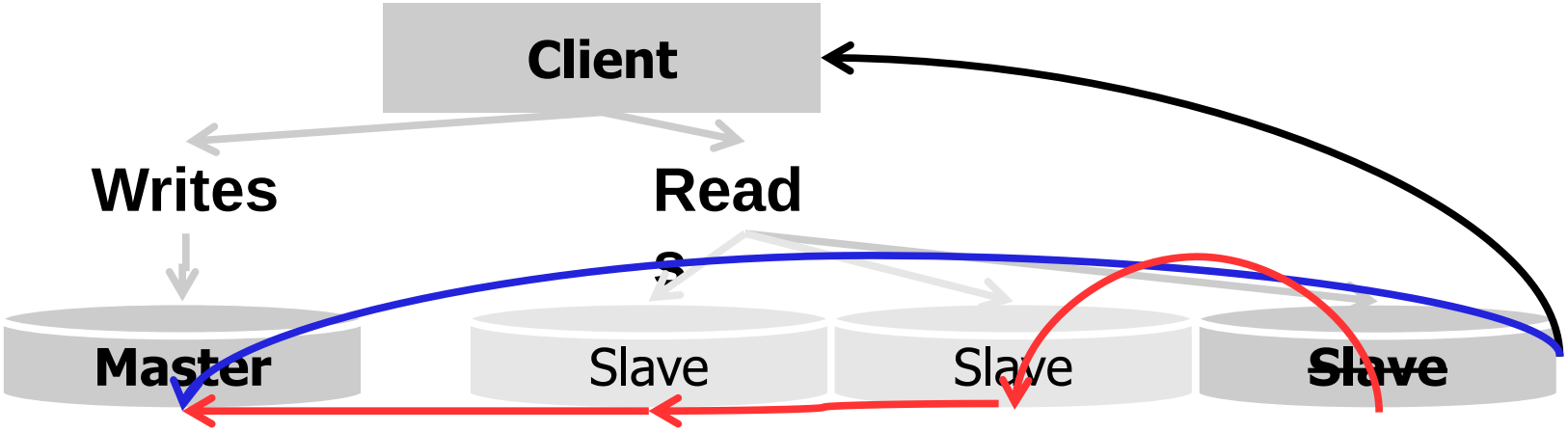


- The best and default load balancing stategy is the one with the lowest impact on the connection state: random once.
  - Minimize switches for the life-span of your PHP script. Pick a random slave during startup and use it for all reads.

- The upcoming 1.4 release will allow you to assign a weight to a node. Nodes with a weight of two will get twice as many requests as nodes with a weight of one. This is not only great if your nodes have different computing power but also to prefer local nodes over remote ones by setting a higher weight.

POOLING: CONNECTION FAILOVER

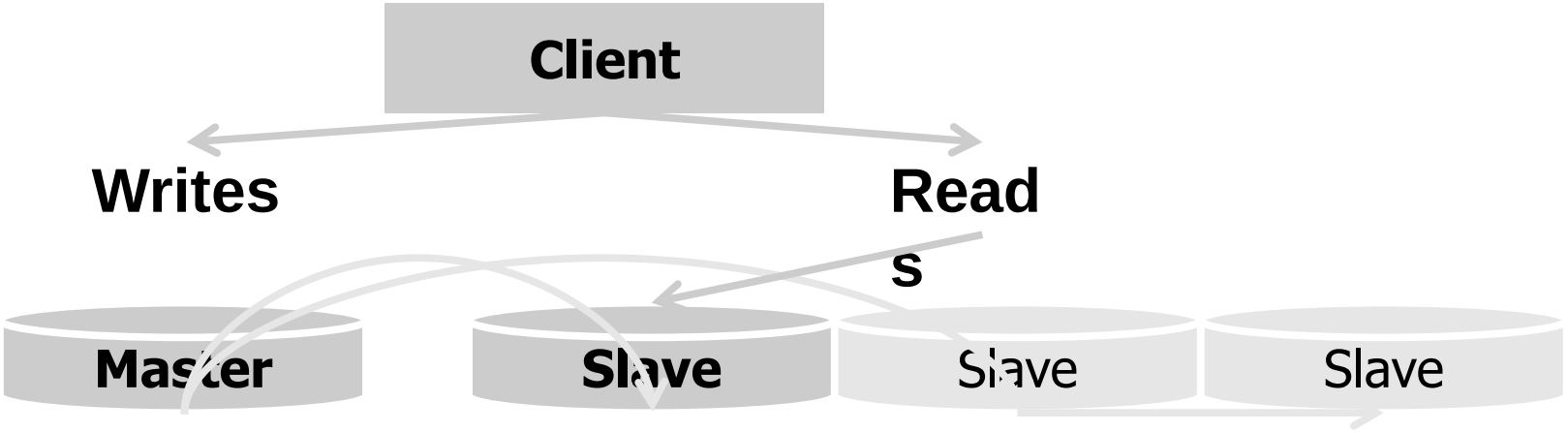
- Caution: connection state change
  - Default: disabled = raise exception
  - Optional: silently failover master
  - 1.4 – remember failed, try more slaves before master



- Connection state changes are what speaks against automatic, silent fail over once a connection has been established and fails. If your application is written in a way that automatic reconnect upon failure is allowed, you can enable it. Otherwise, handle the error, do not close the connection but run the next query to trigger a new load balancing attempt! Failed nodes can be remembered for the duration of a request.
- Automatic failover can either try to connect to the master or try to use another slave before failing over to the master.

POOLING: LAZY CONNECTIONS

- Do not open connection before needed
  - Delay open until statement execution
  - Reduce number of connections, reduce load
  - Great for legacy apps – even without load balancing



MYSQL常用HA和LB方案

常用的Mysql的HA+LB开源方案一般有以下几种

强调下mysql的架构特殊性，一般只有一个写入点，LB主要通过读写分离实现

- 主主复制
- 主从复制
- Mysql-proxy
- Mysql集群套件
- Mysql-mmm
- DRBD+heartbeat
- Lvs+keepalive
- Haproxy
- Amoeba
- 应用层软件自主开发（mysql_rw_php）