Part I. Foundations of Data Systems

2019年6月26日　18:05

S01 Reliable, Scalable, and Maintainable Applications

2019年6月27日 11:28

1. 考虑数据系统Thinking About Data Systems

使用一些通用的组件来设计一个数据系统（架构），需要考虑

a. 如何在部分组件出错甚至不可用时保证数据的准确和完整？（可靠性Reliability）
b. 如何在部分组件劣化时依然保持高性能？（）
c. 如何应对增加的负载？（可扩展性Scalability）
d. 如何设计良好的API？（可维护性Maintainability）

2. 可靠性Reliability

a. 硬件故障

   例如硬盘平均故障时间MTTF是10-50年，因此对于有10000磁盘的系统，平均每天会损坏一块

b. 软件错误

   多个硬件故障往往是独立的，而软件错误可能会出现在所有节点（整个系统）中

c. 人类错误

3. 可扩展性Scalability

a. 描述负载

   以twitter为例，一个user发布的tweet要被n个follower见到

   i. 每个follower要看自己关注人的tweet时，就从每个关注者已发表的tweet中读取，合并
   ii. 每个user发布tweet时就推送到每个follower的缓存中

b. 描述性能

   使用percentile比median(p50)更好

c. 应对负载

   i. 垂直扩展scaling up：增加单机性能
   ii. 水平扩展scaling out：增加集群规模

   针对不同的具体情形设计不同的架构来应对负载

d. 可维护性

   i. 可操作性：让运维更简单
   ii. 简洁性：控制复杂度
   iii. 可演化性：容易修改

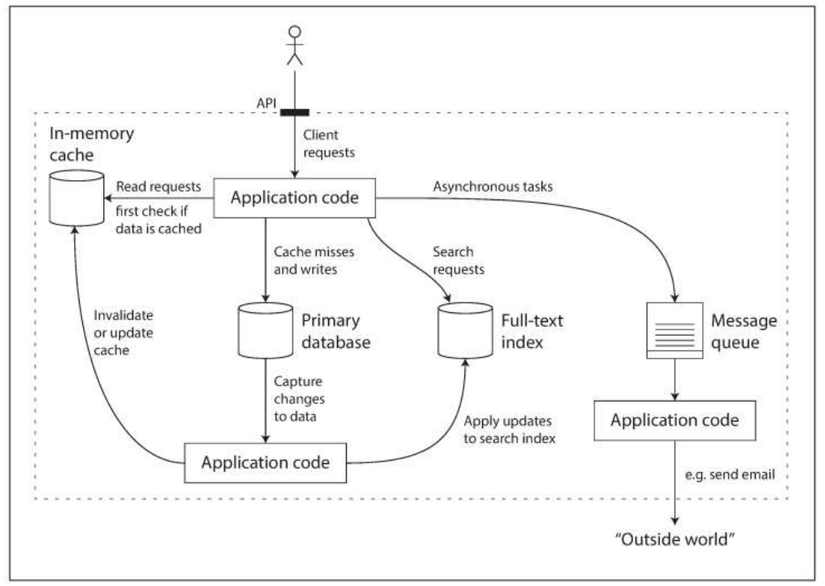


Figure 1-1. One possible architecture for a data system that combines several components.

S02 Data Model and Query Languages

2019年6月27日　　14:44

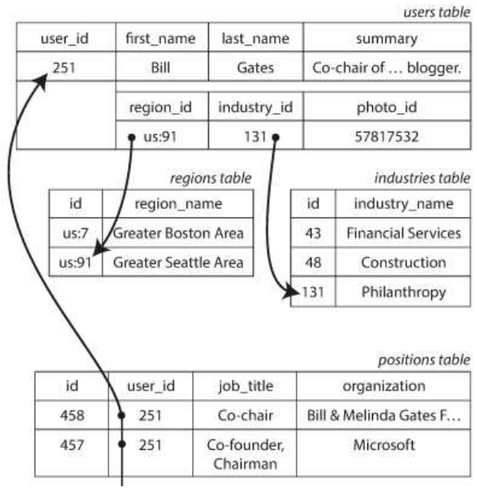
1. 关系模型对比文档模型Relational Model Versus Document Model

a. NoSQL的诞生

b. 对象和关系的不匹配，由此需要ORM（Objecy-Relational Mapping）
以简历为例，JSON格式的简历要比关系模型下的简历有更好的自明性，并且局部性也更好

c. 多对一和多对多关系
关系型数据库下的多对一关系可以利于减少数据重复，并且很容易join，而文档模型下join很弱或者没有，此使就需要应用层通过更多的查询然后模拟join，而例如JSON本质上是支持一对多关系

d. 如今的关系模型对比文档模型
   i. 文档模型schema更灵活，数据局部性更好
   ii. 在文档模型中不能根据某个值直接引用其他项目
   iii. 虽然文档模型join很弱，但是一般采用文档模型的应用并不需要join，具体取决于场景
   iv. 对于高度相关的数据（多对多），文档模型很不合适，关系模型可以适用，图模型是最适配的
   v. 对于schema不定且可能运行时改变的数据，关系模型就非常有害
   vi. 关系型数据库越来越支持JSON/XML，文档型数据库越来越支持join，融合互补是趋势




```
"user_id":     251,
"first_name":  "Bill",
"last_name":   "Gates",
"summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
"region_id":   "us:91",
"industry_id": 131,
"photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
```

2. 查询语言Query Languages for Data

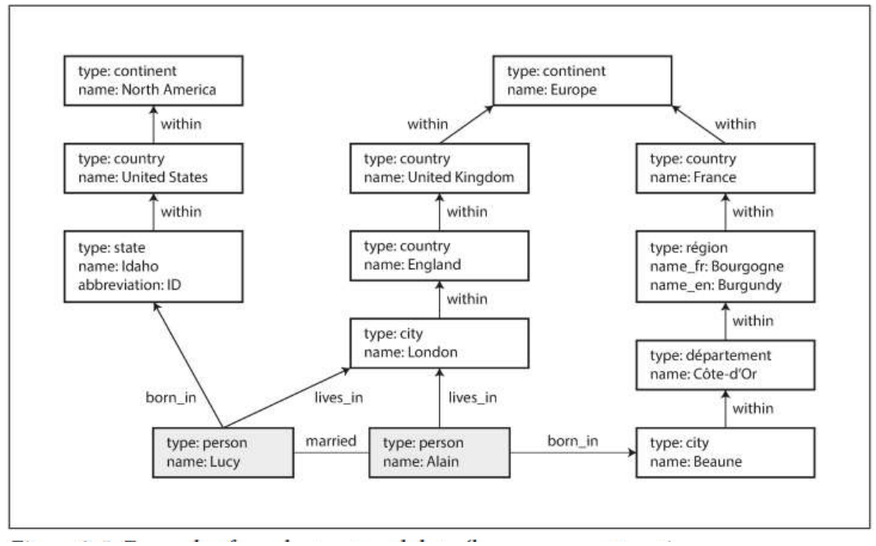
a. 声明式declarative和命令式imperative
命令式即编程语言的方式，逐行执行，而声明式即只声明需要的结果，过程由SQL optimizer/executor来决定，由此可以在不影响外部的情况下对SQL引擎进行修改提升，只要保证结果一致即可

b. Web上的声明式查询

c. MapReduce查询

3. 图数据模型Graph-Like Data Models

a. 属性图Property Graphs
   i. 节点vertex包括
      1) 唯一的ID
      2) 向外的边（出度）
      3) 向内的边（入度）
      4) 一组属性值（K-V对）
   ii. 边edge包括
      1) 唯一的ID
      2) 起始节点
      3) 终止节点
      4) 描述起始和终止节点之间的关系
      5) 一组属性值（K-V对）
   iii. 重要特点
      1) 任意节点可以通过边连接任意节点，没有schema来限制连接关系



iii. 重要特点

1) 任意节点可以通过边连接任意节点，没有schema来限制连接关系
2) 根据给定节点可以高效获取该节点的所有向外的边和向内的边（度）
3) 通过各种不同的关系，一个简单的图上可以存放各种复杂的信息

b. Cypher查询语言

c. SQL中的图查询语言

4. 小结Summary

a. NoSQL相比于传统的关系模型

i. 文档数据库DocumentDB着重单个文档的数据都是自我包含的，不同的文档之间存在关系非常罕见

ii. 图数据库GraphDB则着重任何数据都可能与其他任何数据发生任何关系

Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

S03 Storage and Retrieval

2019年6月29日　　21:54

1. 驱动数据库的数据结构Data Structures That Power Your Database

a. 散列索引Hash Indexes

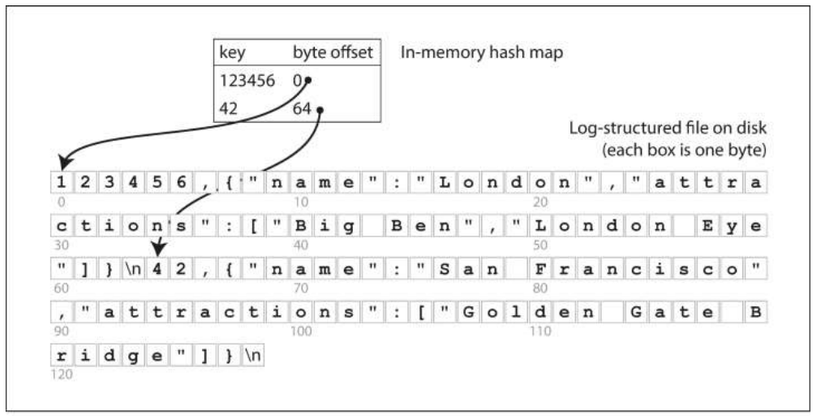


Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

对于不停append only的log，通过设置阈值并且进行压缩compaction生成新的文件来垃圾回收

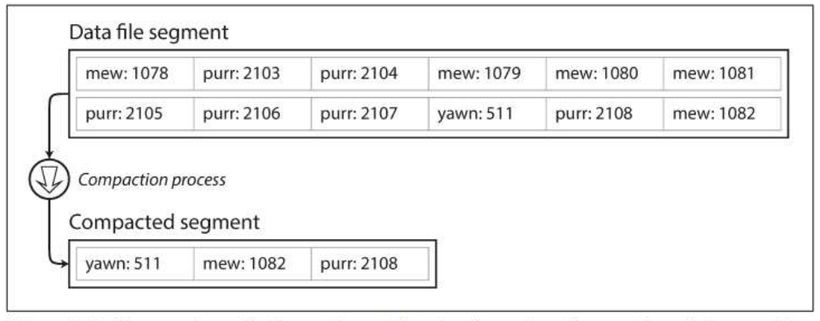


Figure 3-2. Compaction of a key-value update log (counting the number of times each cat video was played), retaining only the most recent value for each key.

对于真实可行的索引，还需要考虑

i. 格式

   直接使用二进制格式，并且通过size后raw string的格式来高效存储

ii. 删除记录

   通过特殊的标记值来表示这个key已被删除，则随后在compaction中就会跳过

iii. 崩溃恢复

   数据库重启时，内存索引（散列表）会丢失，需要根据真实数据情况或是索引本身的持久化快照进行重建

iv. 不完整的记录

   在任何时候数据库崩溃都可能导致写入了不完整的log，使用checksum等机制来确认数据完整性

v. 并发控制

   由于文件的append依赖顺序，因此可以通过多个读线程，单个写线程进行并发控制

优点：

i. 由于append是顺序写入，比in-place修改（随机写入）效率要高很多（HDD&SSD），因此使用看似浪费空间的顺序写入可以获得更理想的性能，空间通过compaction来保持
ii. 并发以及崩溃恢复在顺序写入的情况下更简单，前面完整写入的记录是immutable的
iii. 通过compaction合并旧文件的过程可以避免in-place导致的碎片化

局限:

i. 散列索引必须足够小能够被完整容纳在内存当中
ii. 对任意key的查找非常快O(1)，但是无法高效实现范围range查找

b. SSTables（Sorted String Table）和LSM-Trees（Log Structured Merge Trees）

根据key排序，并且每个log segment文件内包含的key没有重复，由此会有如下优点

i. 合并过程非常简单（归并排序），并且新的segment内的某一key会覆盖任何旧segment的同一key
ii. 由于key是有序的，不再需要在内存中维护所有key的index，只需要维护segment内稀疏的一部分key的offset，具体搜索某个key时先根据内存搜索树确定segment内的offset，随后再扫描这个offset开始segment很小的一部分即可

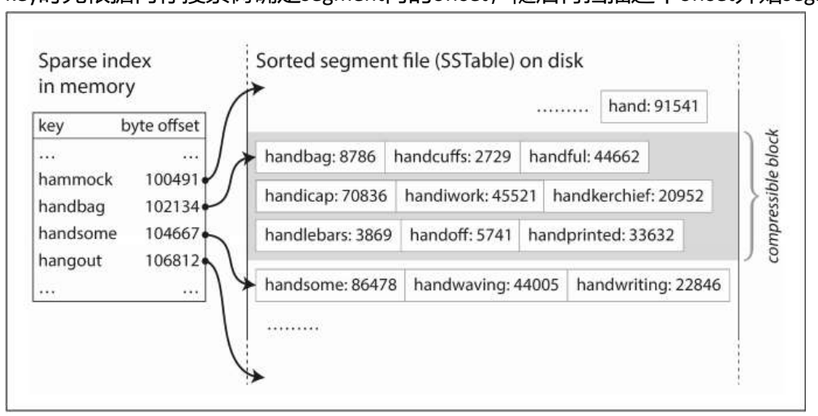


Figure 3-5. An SSTable with an in-memory index.

iii. 另外还可以通过块压缩，将稀疏key对应的segment内的每一部分key进行压缩，以提高磁盘带宽利用率

构建和维护SSTables

i. 内存维护搜索树作为memtable
ii. 当memtable大小达到阈值时直接写入磁盘，作为最新的segment，同时创建新的内存memtable
iii. 有搜索需求时，首先在内存memtable中查找，若没有则根据索引在一系列segments中查找
iv. 定期执行compaction过程，将segments合并并剔除过时或被删除的数据
v. 使用WAL log来防止正在内存而未写入磁盘嗯带memtable的丢失

c. B树

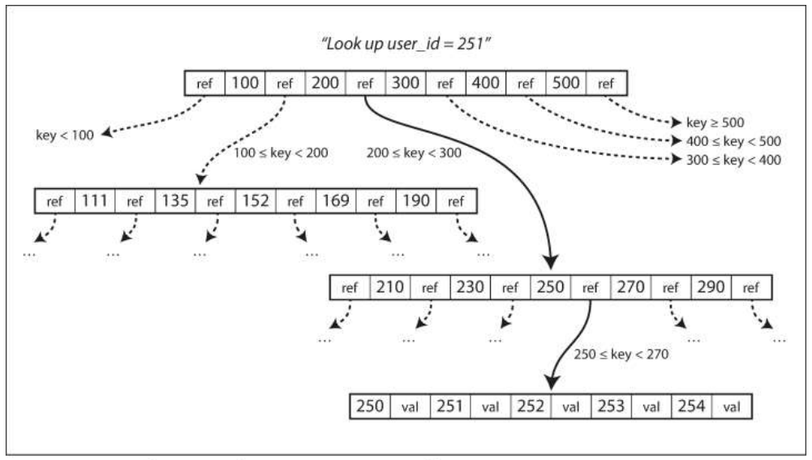


Figure 3-6. Looking up a key using a B-tree index.

一些B树的优化

i. 不通过WAL来进行崩溃恢复，二是使用copy-on-write，每次修改都构建好修改后的page，进行原子替换，这对并发也有利
ii. 储存缩写的key来减少page空间占用，由此来提供更高的分支因子，减少层数
iii. 每个叶子页通过存储兄弟叶子节点页的指针，实现不必回退父节点就可以顺序扫描所有key

d. 对比B树和LSM树
   i. LSM树的优点
      1) B树每个数据必须写两次（WAL和B树），即使很少数据修改都必须整个page更新；而LSM树的一个数据多次修改也会导致写入多次（写放大write amplification），可能对SSD的寿命有影响
      2) LSM树是顺序写入的，即使写放大，往往也比B树的随即写性能要高很多
      3) LSM由于compaction，数据紧凑，很容易压缩提高磁盘利用率，而B树由于分裂叶子节点，往往存在一些占位空间导致碎片化
   ii. LSM树的缺点
      1) LSM后台的compaction过程可能会影响到正在进行的读写，虽然平均响应时间和吞吐量受影响不大，但是可能随机出现某个读写延迟高，而B树的读写延迟更加稳定可预测
      2) 磁盘的带宽被compaction和正常写memtable到磁盘所共享，由此会互相影响性能，而B树的数据只存在于某一页中，不像LSM可能存在于多个segment中，因此B树能更好的支持事务transaction
e. 其他索引结构
   i. 在索引中存储真实数据
      一般索引是K-V对，K就是key，而V可以是真实数据的位置，或直接存放真实数据来减少多次跳跃，存储真实数据的索引称为聚集索引clustered index，而存储真实数据位置的索引就是非聚集索引nonclustered index，在两者之间的索引（存储真实数据位置以及一部分真实数据）称为覆盖索引covering index或包含列的索引index with included columns
   ii. 多列索引
      最简单的多列索引就是连接索引concatenated index，即将多个列连接成为一个索引的key；另外更通用的方式是多维索引multi-dimensional indexes，使用R树
   iii. 全文检索和模糊索引
   iv. 在内存中保存全部数据，内存数据库
      内存数据库的优势在于可以避免磁盘数据库数据结构序列化和反序列化的开销，同时内存中可以构建更复杂的数据模型

2. 事务还是分析Transaction Processing or Analytics?

Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

a. 数据仓库Data Warehousing

b. 星型和雪花型：分析的schemas

   i. 星型又叫维度模型，由一个核心的事实表fact table和周边的维度表dimensional tables构成，事实表往往是巨大的事件集合，维度表则是辅助描述事件的各个属性

   ii. 雪花型就是进一步分割的模型，更加规范化，但是星型易于使用，也更被推荐

   数据仓库中由于事实表往往包含整个事件的信息，因此有非常多的列

c. 列存Column-Oriented Storage

   由于分析型查询往往只涉及小部分列，而往往有非常多上十亿行，因此按行存储Row-Oriented Storage是低效不可接受的，按列存储要求所有列里的数据都以相同的顺序存储，这样一个索引就可以访问每列的对应数据，构成一个

   i. 列压缩

      1) 由于列存中，每个列的数据都是同类型的，因此非常容易压缩以提高IO利用率

      2) 可以是用位图索引bitmap indexes；这是因为列的行数可能很多，但是可能出现的值非常少，即重复值非常多，因此可以通过使用N个可能的值+N个位图（每个可能的值出现在一列的哪些行）来存储一个列

      3) 同时列存一次读取一个chunk，更易于保存在CPU缓存中，且可以充分利用SIMD等向量化方法来处理

   ii. 列存储的排序

      1) 一般列存储可以append-only，也可以通过类似SSTable的方式进行排序，并基于排序提供索引功能

      2) 由于列存储依靠位置来保证每个列的行能相互对应，因此排序时不能每个列单独自己排序，例如查询经常范围查colA，则根据colA进行排序，此后范围查时只需要扫描一小部分行就能获得所有数据

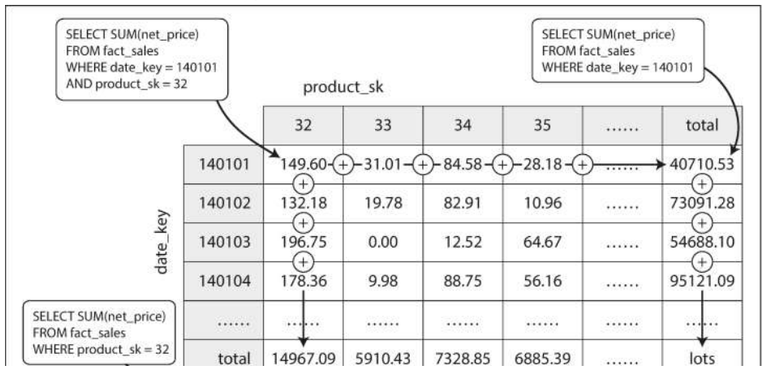
      3) 还可以存储多份根据不同列排序的数据，由此不同的查询自动选择最适配的排序的数据

   iii. 写入列存储

   通过SSTable的处理方式

   iv. 聚合：数据立方Data Cubes和物化视图Materialized Views

   例如sum/avg等聚合查询经常被使用到，因此可以通过物化视图（materialized views: a table-like object whose contents are the results of some query）和数据立方来缓存结果



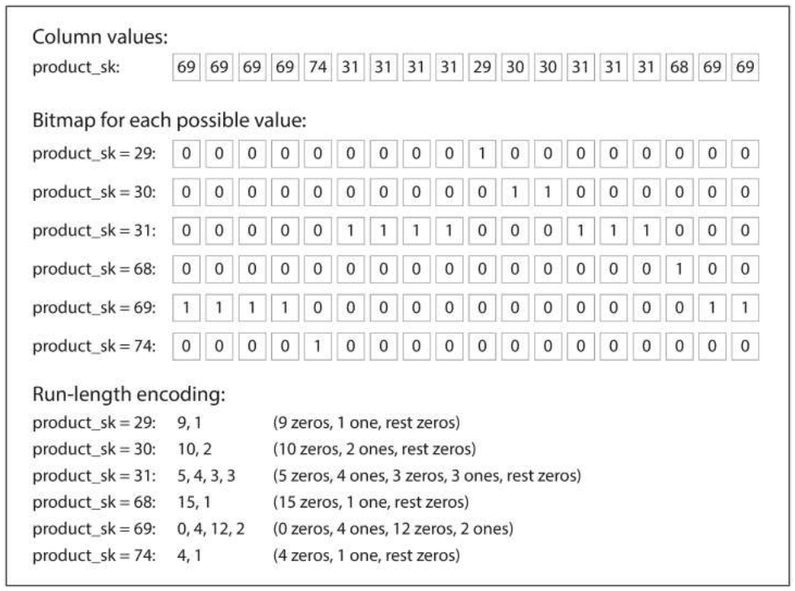


Figure 3-11. Compressed, bitmap-indexed storage of a single column.

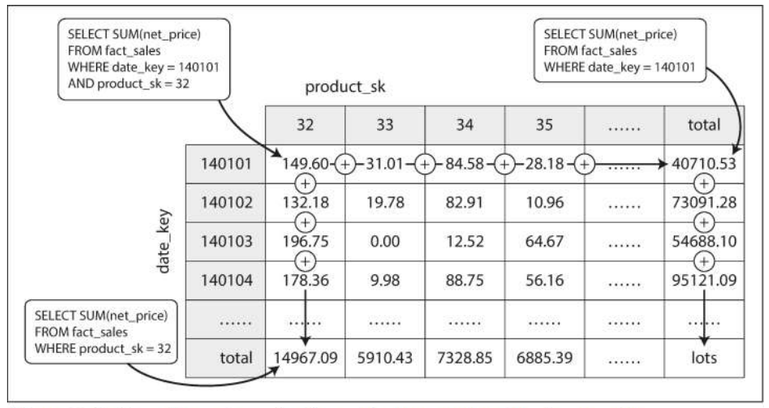


Figure 3-12. Two dimensions of a data cube, aggregating data by summing.

3. 小结Summary

a. 存储引擎两大类
   i. OLTP
      1) 面向用户，应对大量请求
      2) 每个请求根据key只涉及少量数据
      3) 瓶颈在于key的查询
   ii. OLAP
      1) 面向企业，应对少量请求
      2) 每个请求涉及海量数据
      3) 瓶颈在于磁盘带宽

b. OLTP侧的引擎
   i. log-structured（SSTables, LSM-trees）：只允许append和delete过时的文件，从不更新已经写的数据，核心在于把随机写转化成顺序写
   ii. update-in-place（B-trees）：就地修改，管理一组固定大小的页

S04 Encoding and Evolution

2019年6月30日 14:53

1. 数据编码格式Formats for Encoding Data

向后兼容backward compatibility：新代码能读取旧代码存储的数据

向前兼容forward compatibility：旧代码能读取新代码存储的数据

滚动升级rolling update：整个系统由多个分布的节点组成，每个节点可以单独升级，由此需要考虑新老节点间的数据交互

a. 语言限定的格式Language-Specific Formats，往往会有各种问题
   i. 语言限定，其他语言难以解码
   ii. 难以区分不同版本的数据
   iii. 效率
b. JSON, XML, 以及二进制格式

   JSON, XML, CSV等人类可读格式也存在问题
   i. 难以区分数字是数字还是字符串，以及难以区分数字的精度
   ii. JSON, XML支持Unicode的字符串，但是不支持二进制字节串
   iii. XML, JSON可选的支持schema，XML的schema使用广泛，但是JSON的schema并没有广泛使用
   iv. CSV不支持schema
c. Thrift和Protobuf

   Thrift和Protobuf都需要schema来描述数据，例如

   对于Protobuf：

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

   i. 每个字段后的数字tag标记传输的值对应第几个字段，因此传输时并不需要传输整个字段名，因此字段名可以修改，新字段可以添加，但是tag值不能随意使用
   ii. required和optional并不影响底层编码，而optional允许在运行时检查某个字段是否在一条message中，但新增加的字段不可以required
   iii. 向前兼容：旧代码可以直接忽略新增加的字段，根据字段类型跳过相应的字节数
   iv. 向后兼容：每个字段都有唯一的tag值，因此新代码也总能识别旧数据
   v. 对于repeated字段：允许将optional改为repeated，新代码看旧数据会看到一个含有0或者1个元素的列表，而旧代码看新数据会看到列表的最后1个元素
d. Avro

   跳过
e. schema的优点

2. 数据流模式Modes of Dataflow

a. 使用数据库的数据流Dataflow Through Databases

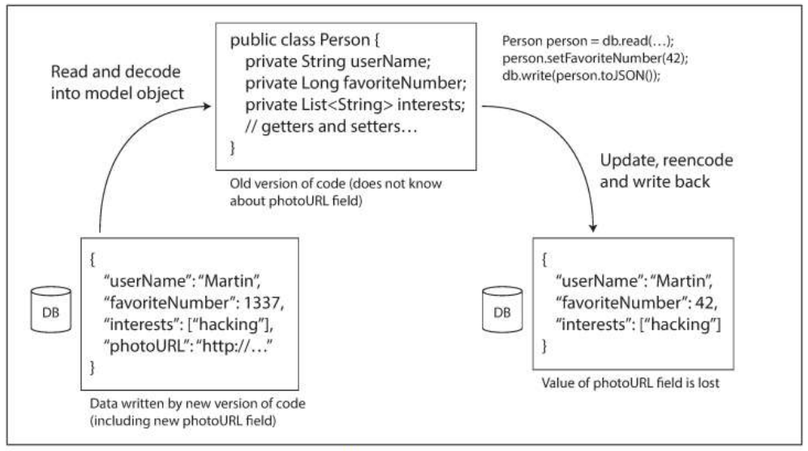


Figure 4-7. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you're not careful.

b. 使用服务的数据流：REST和RPC Dataflow Through Services: REST and RPC

例如在web浏览器里允许的JS程序可以使用XMLHttpRequest来变为HTTP客户端（Ajax技术），从而服务端返回编码的数据给JS程序处理，根据功能将一个整体划分成不同的服务，任意一个模块可以通过请求服务来获取数据，这样的架构称为称为SOA（service-oriented architecture）或微服务（microservices architecture），而如果一个模块通过中间平台再去获取另一个模块的服务，支撑这样微服务架构的中间平台就称为中间件（middleware）

RPC存在的问题

i. 不可预测：本地调用要么成功要么失败，而RPC由于通过网络，可能调用成功但是返回结果延迟或丢失，也可能调用本身延迟或丢失
ii. 超时：同不可预测，当超时之后，RPC发起方无从知晓是已经执行了但返回超时还是没有执行
iii. 幂等性：同上，如果超时后发起重试，如何确保这个操作不被重复执行，即幂等性
iv. 延迟：本地调用耗时可预测，而RPC的耗时则不可预测
v. 引用：本地调用可以使用引用或指针，而RPC传指针毫无意义
vi. 异构：RPC的发起方和执行方可能是不同的语言，而不同的语言之间的数据类型存在不兼容性

c. 使用消息传递的数据流Dataflow Through Message-Passing

使用消息队列（消息中间件）从发布者到订阅者持续低延迟推送数据流，相比于RPC，异步消息传递系统有如下特点

i. 当接收方过载或网络不稳定时，消息队列可以充当缓存，由此提高系统整体的可靠性
ii. 可以重发消息，以避免接收者宕机时的消息丢失
iii. 单发送者可以向多个接收者发送同一个消息
iv. 收发解耦，发送者只发送而不关心接收者，接收者只接收而不关心发送者
v. 但是消息系统相比于RPC，是单向的（即使在另一条通道中发送返回消息），发送者往往不会等待每一条消息的返回

消息系统Message Brokers例如RabbitMQ：一或多个生产者在消息系统上指定topic，并写入对应的数据流，一或多个消费者在消息系统上指定topic，并获取对应的数据流

分布式Actor系统例如Akka：每个actor之间不共享数据，一切都通过消息传递来执行，消息可能会丢失