NoSQL 数据建模技术(1)

2012-05-15 10:28 陈皓 酷壳 我要评论(0) 字号：T | T

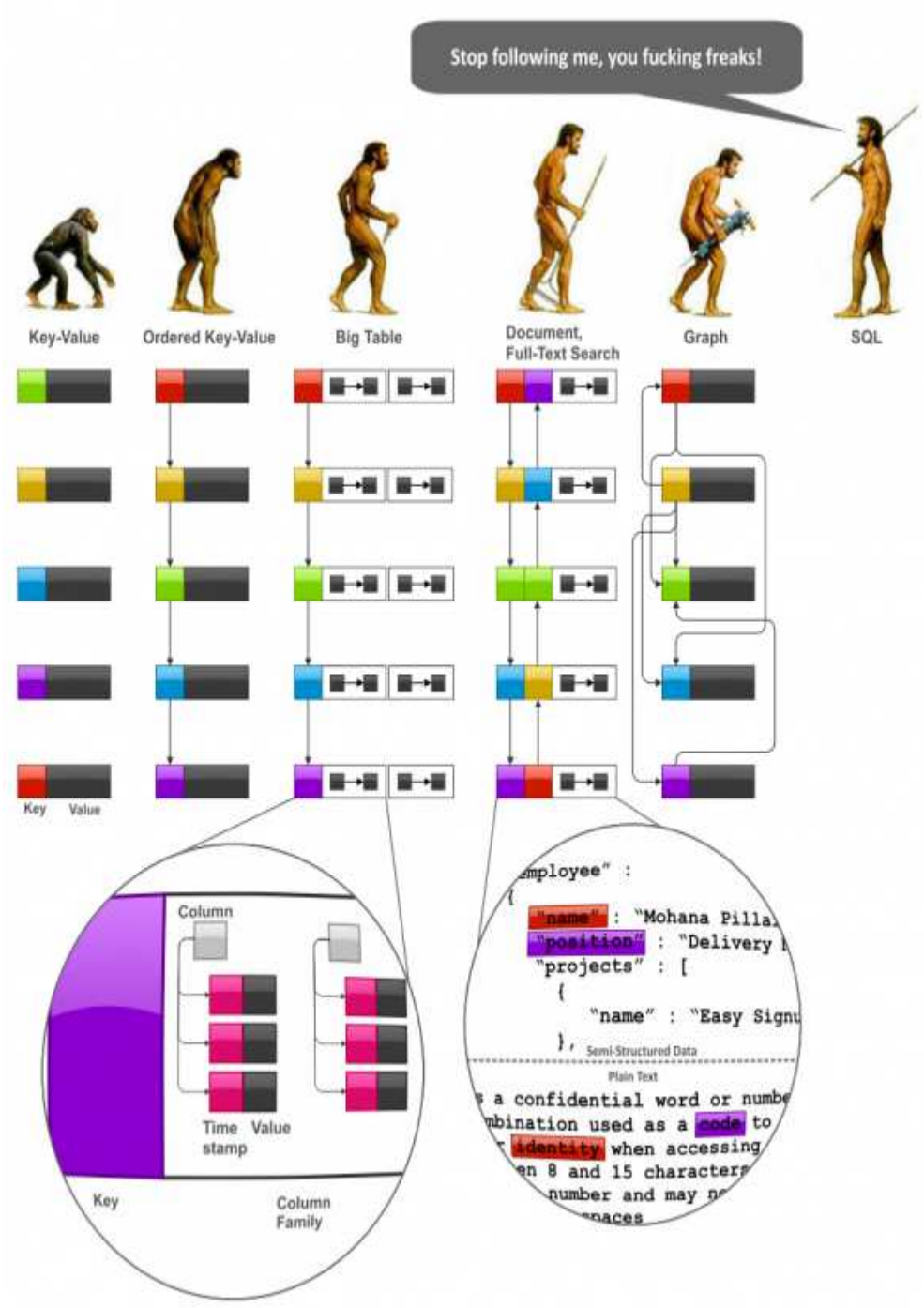
收藏

全文译自墙外文章“NoSQL Data Modeling Techniques”，译得不好，还请见谅。这篇文章看完之后，你可能会对 NoSQL 的数据结构会有些感觉。我的感觉是，关系型数据库想把一致性，完整性，索引，CRUD 都干好，NoSQL 只干某一种事，但是牺牲了很多别的东西。总体来说，我觉得 NoSQL 更适合做 Cache。

全文译自墙外文章“NoSQL Data Modeling Techniques”，译得不好，还请见谅。这篇文章看完之后，你可能会对 NoSQL 的数据结构会有些感觉。我的感觉是，关系型数据库想把一致性，完整性，索引，CRUD 都干好，NoSQL 只干某一种事，但是牺牲了很多别的东西。总体来说，我觉得 NoSQL 更适合做 Cache。下面是正文——

NoSQL 数据库经常被用作很多非功能性的地方，如，扩展性，性能和一致性的地方。这些 NoSQL 的特性在理论和实践中都正在被大众广泛地研究着，研究的热点正是那些和性能分布式相关的非功能性的东西，我们都知道 CAP 理论被很好地应用于了 NoSQL 系统中（陈皓注：CAP 即，一致性(Consistency)， 可用性(Availability)， 分区容忍性(Partition tolerance)，在分布式系统中，这三个要素最多只能同时实现两个，而 NoSQL 一般放弃的是一致性）。但在另一方面，NoSQL 的数据建模技术却因为缺乏像关系型数据库那样的基础理论没有被世人很好地研究。这篇文章从数据建模方面对 NoSQL 家族进行了比较，并讨论几个常见的数据建模技术。

要开始讨论数据建模技术，我们不得不或多或少地先系统地看一下 NoSQL 数据模型的成长的趋势，以此我们可以了解一些他们内在的联系。下图是 NoSQL 家族的进化图，我们可以看到这样的进化：Key-Value 时代，BigTable 时代，Document 时代，全文搜索时代，和 Graph 数据库时代：（陈皓注：注意图中 SQL 说的那句话，NoSQL 再这样发展下去就是 SQL 了，哈哈。）



NoSQL Data Models

首先，我们需要注意的是 SQL 和关系型数据模型已存在了很长的时间，这种面向用户的自然性意味着：

- 最终用户一般更感兴趣于数据的聚合显示，而不是分离的数据，这主要通过 SQL 来完成。

- 我们无法通过人手工控制数据的并发性，完整性，一致性，或是数据类型校验这些东西的。这就是为什么 SQL 需要在事务，二维表结构（schema）和外表联合上做很多事。

另一方面，SQL 可以让软件应用程序在很多情况下不需要关心数据库的数据聚合，和数据完整性和有效性进行控制。而如果我们去除了数据一致性，完整性这些东西，会对性能和分布存储有着重的帮助。正因为如此，我们才有数据模型的进化：

- **Key-Value 键值对存储**是非常简单而强大的。下面的很多技术基本上都是基于这个技术开始发展的。但是，Key-Value 有一个非常致命的问题，那就是如果我们需要查找一段范围内的 key。（陈皓注：学过 hash-table 数据结构的人都应该知道，hash-table 是非序列容器，其并不像数组，链接，队列这些有序容器，我们可以控制数据存储的顺序）。于是，有序键值 （Ordered Key-Value） 数据模型被设计出来解决这一限制，来从根本上提高数据集的问题。
- **Ordered Key-Value 有序键值**模型也非常强大，但是，其也没有对 Value 提供某种数据模型。通常来说，Value 的模型可以由应用负责解析和存取。这种很不方便，于是出现了 BigTable 类型的数据库，这个数据模型其实就是 map 里有 map，map 里再套 map，一层一层套下去，也就是层层嵌套的 key- value（value 里又是一个 key-value），这种数据库的 Value 主要通过“列族”（column families），列，和时间戳来控制版本。（陈皓注：关于时间戳来对数据的版本控制主要是解决数据存储并发问题，也就是所谓的乐观锁，详见《多版本并发控制(MVCC)在分布式系统中的应用》）
- **Document databases 文档数据库** 改进了 BigTable 模型，并提供了两个有意义的改善。第一个是允许 Value 中有主观的模式（scheme），而不是 map 套 map。第二个是索引。**Full Text Search Engines 全文搜索引擎**可以被看作是文档数据库的一个变种，他们可以提供灵活的可变的数据模式（scheme）以及自动索引。他们之间的不同点主要是，文档数据库用字段名做索引，而全文搜索引擎用字段值做索引。
- **Graph data models 图式数据库** 可以被认为是这个进化过程中从 Ordered Key-Value 数据库发展过来的一个分支。图式数据库允许构建议图结构的数据模型。它和文档数据库有关系的原因是，它的很多实现允许 value 可以是一个 map 或是一个 document。

NoSQL 数据模型摘要

本文剩下的章节将向你介绍数据建模的技术实现和相关模式。但是，在介绍这些技术之前，先来一段序言：

- NoSQL 数据模型设计一般从业务应用的具体数据查询入手，而不是数据间的关系：

- 关系型的数据模型基本上是分析数据间的结构和关系。其设计理念是： **”What answers do I have?”**
- NoSQL 数据模型基本上是从应用对数据的存取方式入手，如：我需要支持某种数据查询。其设计理念是 **”What questions do I have?”**

- NoSQL 数据模型设计比关系型数据库需要对数据结构和算法的更深的了解。在这篇文章中我会和大家说那些尽人皆知的数据结构，这些数据结构并不只是被 NoSQL 使用，但是对于 NoSQL 的数据模型却非常有帮助。
- 数据冗余和反规格化是一等公民。
- 关系型数据库对于处理层级数据和图式数据非常的不方便。NoSQL 用来解决图式数据明显是一个非常好的解决方案，几乎所有的 NoSQL 数据库可以很强地解决此类问题。这就是为什么这篇文章专门拿出一章来说明层级数据模型。

下面是 NoSQL 的分类表，也是我用来写这篇文章时做实践的产品：

- Key-Value 存储: Oracle Coherence, Redis, Kyoto Cabinet
- 类 BigTable 存储: Apache HBase, Apache Cassandra
- 文档数据库: MongoDB, CouchDB
- 全文索引: Apache Lucene, Apache Solr
- 图数据库: neo4j, FlockDB

概念技术 Conceptual Techniques

这一节主要介绍 NoSQL 数据模型的基本原则。

(1) 反规格化 Denormalization

反规格化 Denormalization 可以被认为是把相同的数据拷贝到不同的文档或是表中，这样就可以简化和优化查询，或是正好适合用户的某中特别的数据模型。这篇文章中所说的绝大多数技术都或多或少地导向了这一技术。

总体来说，反规格化需要权衡下面这些东西：

- ***查询数据量 /查询 IO*** VS ***总数据量***。使用反规格化，一方面可以把一条查询语句所需要的所有数据组合起来放到一个地方存储。这意味着，其它不同不同查询所需要的相同的数据，需要放在别不同的地方。因此，这产生了很多冗余的数据，从而导致了数据量的增大。

- ***处理复杂度*** VS ***总数据量***. 在符合范式的数据模式上进行表连接的查询，很显然会增加了查询处理的复杂度，尤其对于分布式系统来说更是。反规格化的数据模型允许我们以方便查询的方式来存构造数据结构以简化查询复杂度。

适用性: Key-Value Store 键值对数据库， Document Databases 文档数据库， BigTable 风格的数据库。

(2) 聚合 Aggregates

所有类型的 NoSQL 数据库都会提供灵活的 Schema（数据结构，对数据格式的限制）：

- Key-Value Stores 和 Graph Databases 基本上来说不会 Value 的形式，所以 Value 可以是任意格式。这样一来，这使得我们可以任意组合一个业务实体的 keys。比如，我们有一个用户帐号的业务实体，其可以被如下这些 key 组合起来： *UserID_name, UserID_email, UserID_messages* 等等。如果一个用户没有 email 或 message，那么相应也不会有这样的记录。
- BigTable 模型通过列集合来支持灵活的 Schema，我们称之为列族（*column family*）。BigTable 还可以在同一记录上出现不同的版本（通过时间戳）。
- Document databases 文档数据库是一种层级式的“去 Schema”的存储，虽然有些这样的数据库允许检验需要保存的数据是否满足某种 Schema。

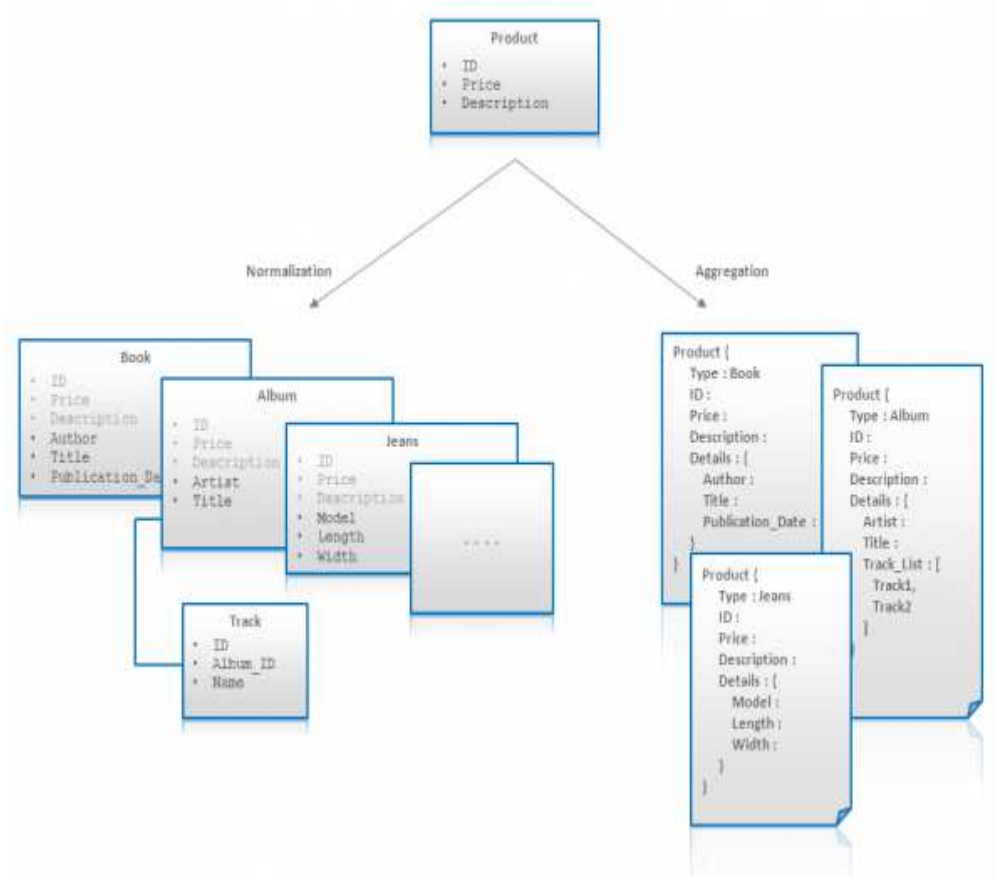
灵活的 Schema 允许你可以用一种嵌套式的内部数据方式来存储一组有关联的业务实体（陈皓注：类似于 JSON 这样的数据封装格式）。这样可以为我们带来两个好处。

- 最小化“一对多”关系——可以通过嵌套式的方式来存储实体，这样可以少一些表联结。
- 可以让内部技术上的数据存储更接近于业务实体，特别是那种混合式的业务实体。可能存于一个文档集或是一张表中。

下图示意了这两种好处。图中描给了电子商务中的商品模型（陈皓注：我记得我在“挑战无处不在”一文中说到过电商中产品分类数据库设计的挑战）

- 首先，所有的商品 Product 都会有一个 ID，Price 和 Description。
- 然后，我们可以知道不同的类型的商品会有不同的属性。比如，作者是书的属性，长度是牛仔裤的属性。其些属性可能是“一对多”或是“多对多”的关系，如：唱片中的曲目。
- 接下来，我们知道，某些业务实体不可能使用固定的类型。如：牛仔裤的属性并不是所有的牌子都有的，而且，有些名牌还会搞非常特别的属性。

对于关系型数据库来说，要设计这样的数据模型并不简单，而且设计出来的绝对离优雅很远很远。而我们 NoSQL 中灵活的 Schema 允许你使用一个聚合 Aggregate（product）可以建出所有不同种类的商品和他们的不同的属性：



Entity Aggregation

上图中我们可以比较关系型数据库和 NoSQL 的差别。**但是我们可以看到在数据更新上，非规格化的数据存储在性能和一致性上会有很大的影响，这就是我们需要重点注意和不得不牺牲的地方**。

适用性： Key-Value Store 键值对数据库， Document Databases 文档数据库， BigTable 风格的数据库。

(3) 应用层联结 Application Side Joins

表联结基本上不被 NoSQL 支持。正如我们前面所说的，NoSQL 是“面向问题”而不是“面向答案”的，不支持表联结就是“面向问题”的后果。表的联结是在设计时被构造出来的，而不是在执行时建造出来的。所以，表联结在运行时是有很大开销的（陈皓注：搞过 SQL 表联结的都知道笛卡尔积是什么东西，大可以在参看以前酷壳的“图解数据库表 Joins”），但是在使用了 Denormalization 和

Aggregates 技术后，我们基本不用进行表联结，如：你们使用嵌套式的数据实体。当然，如果你需要联结数据，你需要在应用层完成这个事。下面是几个主要的 Use Case：

- 多对多的数据实体关系——经常需要被连接或联结。
- 聚合 Aggregates 并不适用于数据字段经常被改变的情况。对此，我们需要把那些经常被改变的字段分到另外的表中，而在查询时我们需要联结数据。例如，我们有个 Message 系统可以有一个 User 实体，其包括了一个内嵌的 Message 实体。但是，如果用户不断在附加 message，那么，最好把 message 拆分到另一个独立的实体，但在查询时联结这 User 和 Message 这两个实体。如下图：

适用性: Key-Value Store 键值对数据库， Document Databases 文档数据库， BigTable 风格的数据库， Graph Databases 图数据库。

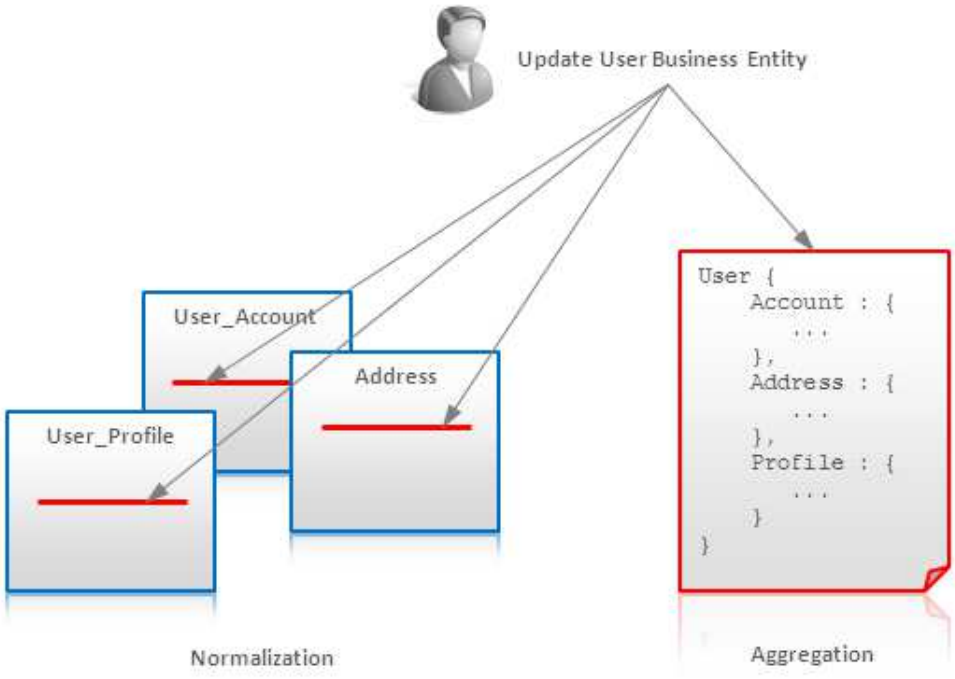
通用建模技术 General Modeling Techniques

在本书中，我们将讨论 NoSQL 中各种不同的通用的数据建模技术。

(4) 原子聚合 Atomic Aggregates

很多 NoSQL 的数据库（并不是所有）在事务处理上都是短板。在某些情况下，他们可以通过分布式锁技术或是应用层管理的 MVCC 技术来实现其事务性（陈皓注：可参看本站的“多版本并发控制(MVCC)在分布式系统中的应用”）但是，通常来说只能使用聚合 Aggregates 技术来保证一些 ACID 原则。

这就是为什么我们的关系型数据库需要有强大的事务处理机制——因为关系型数据库的数据是被规格化存放在了不同的地方。所以，Aggregates 聚合允许我们把一个业务实体存成一个文档、存成一行，存成一个 key-value，这样就可以原子式的更新了：



Atomic Aggregates

当然，原子聚合 Atomic Aggregates 这种数据模型并不能实现完全意义上的事务处理，但是如果支持原子性，锁，或 test-and-set 指令，那么， Atomic Aggregates 是可以适用的。

适用性： Key-Value Store 键值对数据库， Document Databases 文档数据库， BigTable 风格的数据库。

(5) 可枚举键 Enumerable Keys

也许，对于无顺序的 Key-Value 最大的好处是业务实体可以被容易地 hash 以分区在多个服务器上。而排序了的 key 会把事情搞复杂，但是有些时候，一个应用能从排序 key 中获得很多好处，就算是数据库本身不提供这个功能。让我们来思考下 email 消息的数据模型：

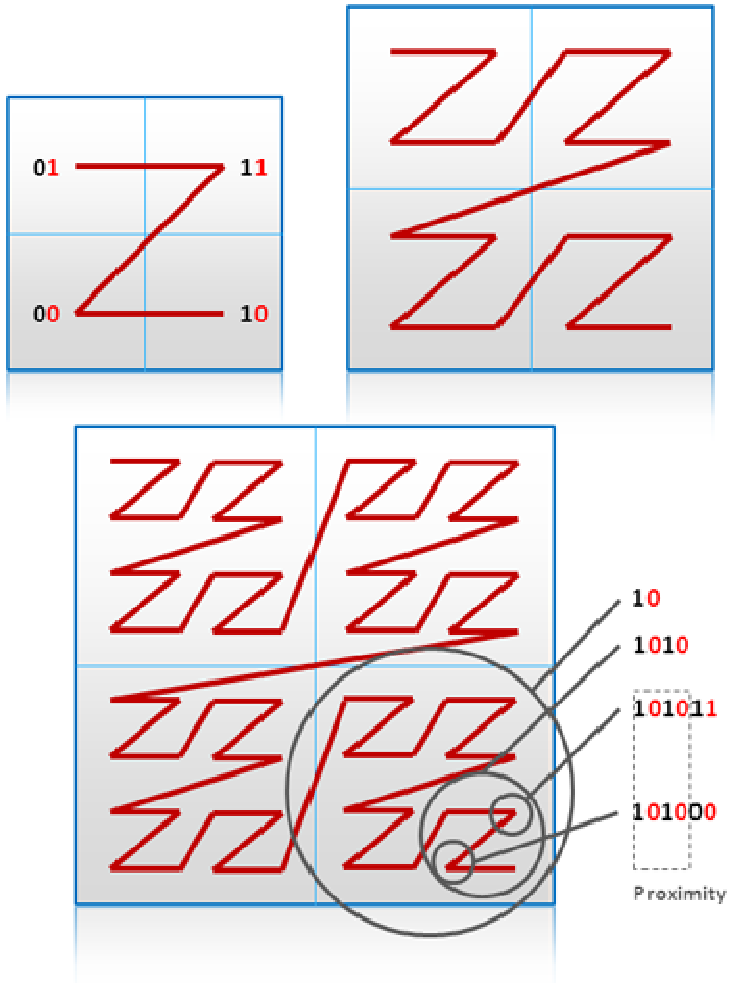
1. 一些 NoSQL 的数据库提供原子计数器以允许生一些连续的 ID。在这种情况下，我们可以使用 *userID_messageID* 来做为一个组合 key。如果我们知道最新的 message ID，就可以知道前一个 message，也可能知道再前面和后面的 Message。
2. Messages 可以被打包。比如，每天的邮件包。这样，我们就可以对邮件按指定的时间段来遍历。

适用性： Key-Value Store 键值对数据库。

(6) 降维 Dimensionality Reduction

Dimensionality Reduction 降维是一种技术可以允许把一个多维的数据映射成一个 Key-Value 或是其它非多给的数据模型。

传统的地理位置信息系统使用一些如“四分树 QuadTree” 或 “R-Tree” 来做地理位置索引。这些数据结构的内容需要被在适当的位置更新，并且，如果数据量很大的话，操作成本会很高。另一个方法是我们可以遍历一个二维的数据结构并把其扁平化成一个列表。一个众所周知的例子是 Geohash（地理哈希）。一个 Geohash 使用“之字形”的路线扫描一个 2 维的空间，而且遍历中的移动可以被简单地用 0 和 1 来表示其方向，然后在移动的过程中产生 0/1 串。下图展示了这一算法：（陈皓注：先把地图分成四份，经度为第一位，纬度为第二位，于是左边的经度是 0，右边的是 1，纬度也一样，上面是为 1，下面的为 0，这样，经纬度就可以组合成 01，11，00，10 这四个值，其标识了四块区域，我们可以如此不断的递归地对每个区域进行四分，然后可以得到一串 1 和 0 组成的字串，然后使用 0-9，b-z 去掉（去掉 a，i，l，o）这 32 个字母进行 base32 编码得到一个 8 个长度的编码，这就是 Geohash 的算法）



Geohash Index

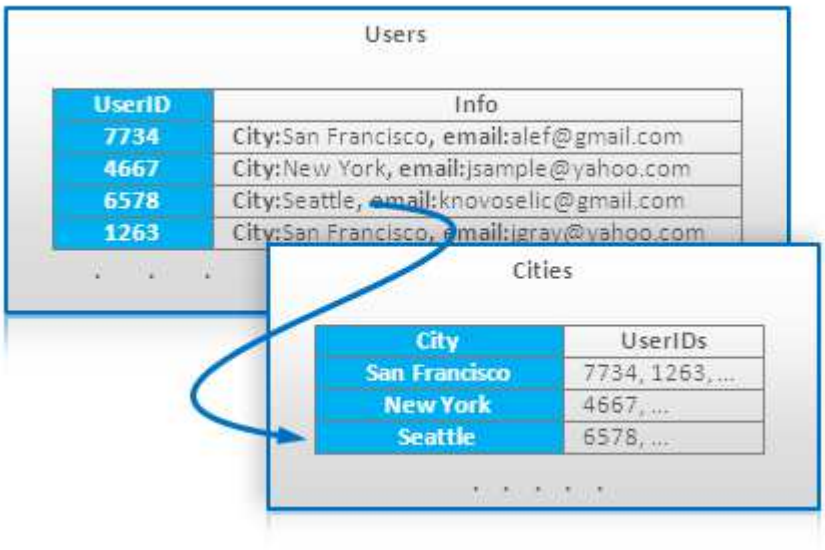
Geohash 的最强大的功能是使用简单的位操作就可以知道两个区域间的距离，就像图中所示（陈皓：proximity 框着的那两个，这个很像 IP 地址了）。Geohash 把一个二维的坐标生生地变成了一个一维的数据模型，这就是降维技术。BigTable 的降维技术参看到文章后面的 [6.1]。更多的关于 Geohash 和其它技术可以参看 [6.2] 和 [6.3]。

适用性： Key-Value Store 键值对数据库， Document Databases 文档数据库， BigTable 风格的数据库。

(7) 索引表 Index Table

Index Table 索引表是一个非常直白的技术，其可以你在不支持索引的数据库中得到索引的好处。BigTable 是这类最重要的数据库。这需要我们维护一个有相应存取模式的特别表。例如，我们有一个

主表存着用户帐号，其可以被 UserID 存取。某查询需要查出某个城市里所有的用户，于是我们可以加入一张表，这张表用城市做主键，所有和这个城市相关的 UserID 是其 Value，如下所示：



Index Table Example

可见，城市索引表的需要和对主表用户表保持一致性，因此，主表的每一个更新可能需要对索引表进行更新，不然就是一个批处理更新。无论哪个方式，这都会损伤一些性能，因为需要保持一致性。

Index Table 索引表可以被认为是关系型数据库中的视图的等价物。

适用性: BigTable 数据库。

(8) 键组合索引 Composite Key Index

Composite key 键组合是一个很常用的技术，对此，当我们的数据库支持键排序时能得到极大的好处。Composite key 组合键的拼接成为第二排序字段可以让你构建出一种多维索引，这很像我们之前说过的 Dimensionality Reduction 降维技术。例如，我们需要存取用户统计。如果我们需要根据不同的地区来统计用户的分布情况，我们可以把 Key 设计成这样的格式 *(State:City:UserID)*，这样一来，就使得我们可以通过 State 到 City 来按组遍历用户，特别是我们的 NoSQL 数据库支持在 key 上按区查询（如：BigTable 类的系统）：

```
1SELECT Values WHERE state="CA:*"

2SELECT Values WHERE city="CA:San Francisco*"
```

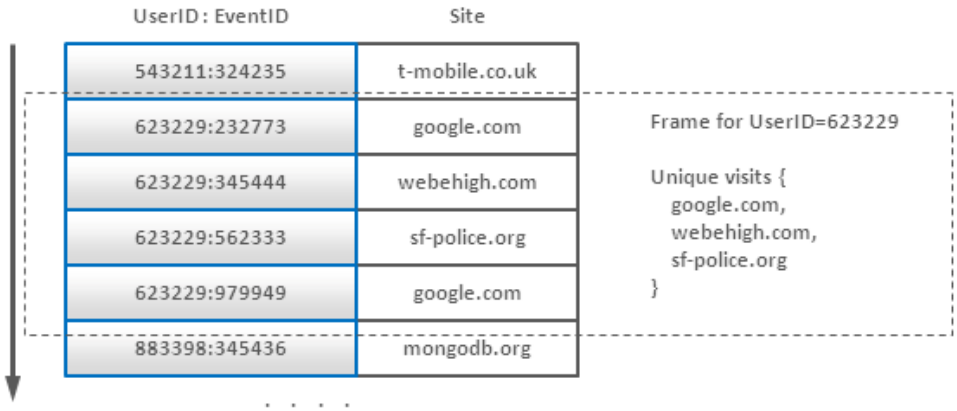
适用性： BigTable 数据库。

(9) 键组合聚合 Aggregation with Composite Keys

Composite keys 键组合技术并不仅仅可以用来做索引，同样可以用来区分不用的类型的数据以支持数据分组。考虑一个例子，我们有一个海量的日志数组，这个日志记录了互联网上的用户的访问来源。我们需要计算从某一网站过来的独立访客的数量，在关系型数据库中，我们可能需要下面这样的 SQL 查询语句：

```
1SELECT count(distinct(user_id)) FROM clicks GROUP BY site
```

我们可以在 NoSQL 中建立如下的数据模型：



Counting Unique Users using Composite Keys

这样，我们就可以把数据按 UserID 来排序，我们就可以很容易把同一个用户的数据（一个用户并不会产生太多的 event）进行处理，去掉那些重复的站点（使用 hash table 或是别的什么）。另一个可选的技术是，我们可以对每一个用户建立一个数据实体，然后把其站点来源追加到这个数据实体中，当然，这样一来，数据的更新在性能相比之下会有一定损失。

适用性： Ordered Key-Value Store 排序键值对数据库， BigTable 风格的数据库。

(10) 反转搜索 Inverted Search - 直接聚合 Direct Aggregation

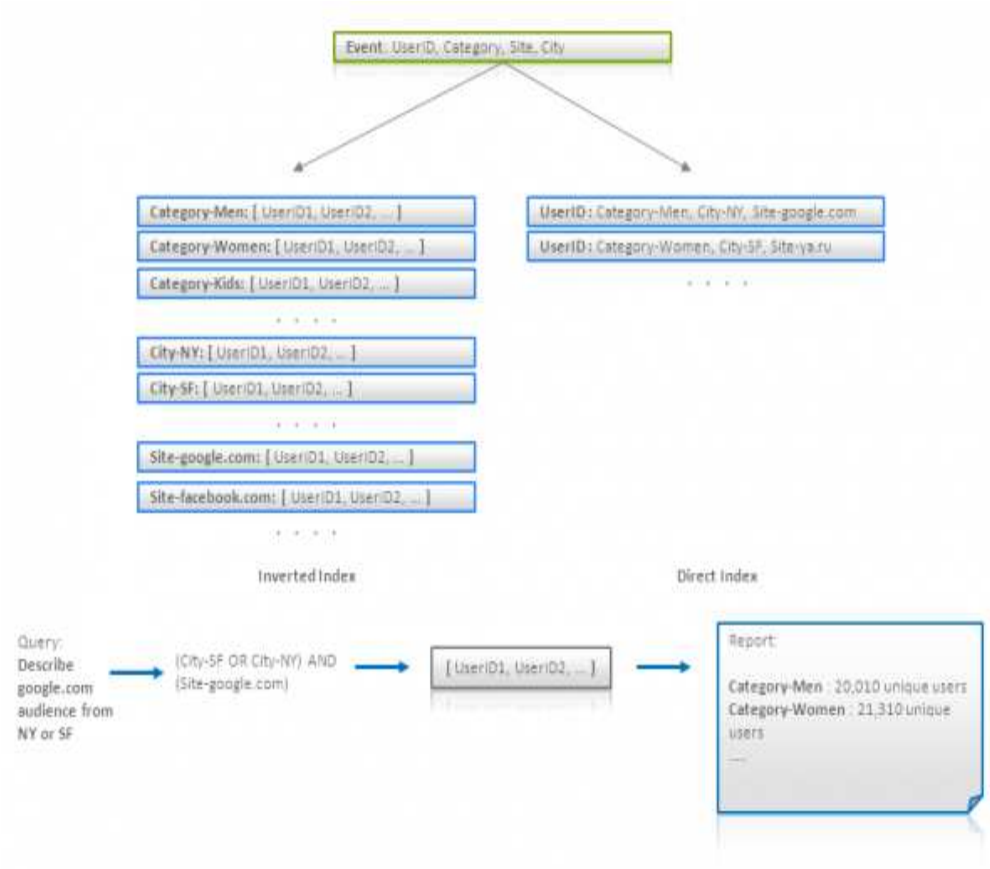
这个技术更多的是数据处理技术，而不是数据建模技术。尽管如此，这个技术还是会影响数据模型。这个技术最主要的想法是使用一个索引来找到满足某条件的数据，但是把数据聚合起需要使用全文搜索。还是让我们来说一个示例。还是用上面那个例子，我们有很多的日志，其中包括互联网用户和他们的访问来源。让我们假定每条记录都有一个 UserID，还有用户的种类 (Men, Women,

Bloggers, 等)，以及用户所在的城市，和访问过的站点。我们要干的事是，为每个用户种类找到满足某些条件（访问源，所在城市，等）的的独立用户。

很明显，我们需要搜索那些满足条件的用户，如果我们使用反转搜索，这会让我们把这事干得很容易，如： *{Category -> [user IDs]}* 或 *{Site -> [user IDs]}*。使用这样的索引， 我们可以取两个或多个 UserID 要的交集或并集（这个事很容易干，而且可以干得很快，如果这些 UserID 是排好序的）。但是，我们要按用户种类来生成报表会变得有点麻烦，因为我们用语句可能会像下面这样

```
1SELECT count(distinct(user_id)) ... GROUP BY category
```

但这样的 SQL 很没有效率，因为 category 数据太多了。为了应对这个问题，我们可以建立一个直接索引 *{UserID -> [Categories]}* 然后我们用它来生成报表：



Counting Unique Users using Inverse and Direct Indexes

最后，我们需要明白，对每个 UserID 的随机查询是很没有效率的。我们可以通过批查询处理来解决这个问题。这意味着，对于一些用户集，我们可以进行预处理（不同的查询条件）。

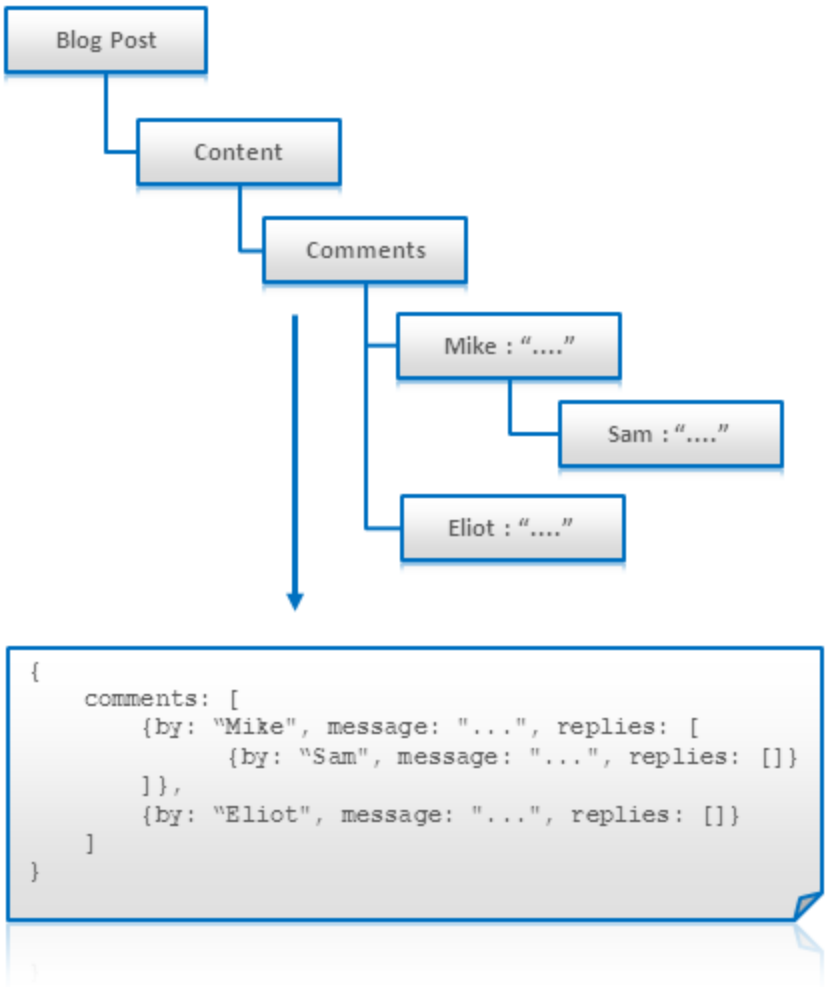
适用性： Key-Value Store 键值对数据库， Document Databases 文档数据库， BigTable 风格的数据库。

层级式模型 Hierarchy Modeling Techniques

(11) 树形聚合 Tree Aggregation

树形或是任意的图（需反规格化）可以被直接打成一条记录或文档存放。

- 当树形结构被一次性取出时这会非常有效率（如：我们需要展示一个 blog 的树形评论）
- 搜索和任何存取这个实体都会存在问题。
- 对于大多数 NoSQL 的实现来说，更新数据都是很不经济的（相比起独立结点来说）



Tree Aggregation

适用性: Key-Value 键值对数据库, Document Databases 文档数据库

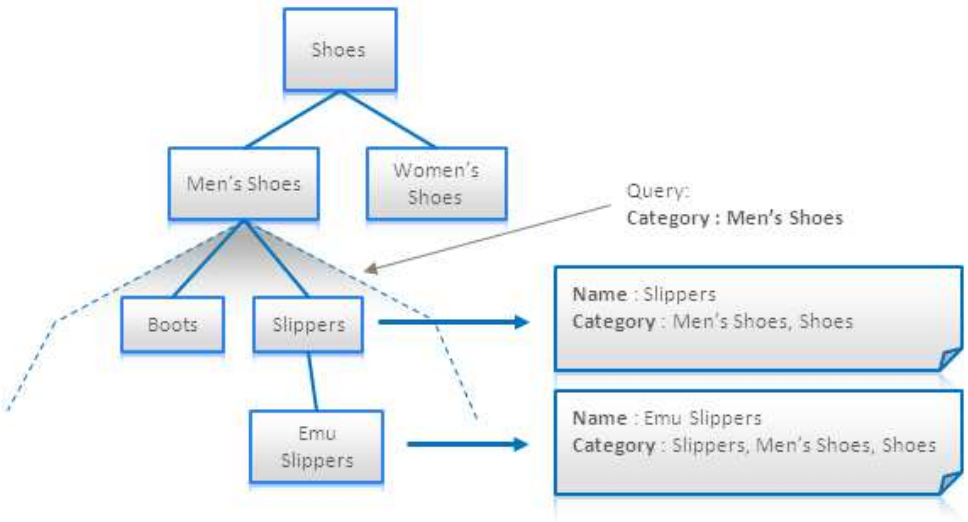
(12) 邻接列表 Adjacency Lists

Adjacency Lists 邻接列表是一种图 - 每一个结点都是一个独立的记录，其包含了 所有的父结点或子结点。这样，我们就可以通过给定的父或子结点来进行搜索。当然，我们需要通过 hop 查询遍历图。这个技术在广度和深度查询，以及得到某个结点的子树上没有效率。

适用性: Key-Value 键值对数据库, Document Databases 文档数据库

(13) Materialized Paths

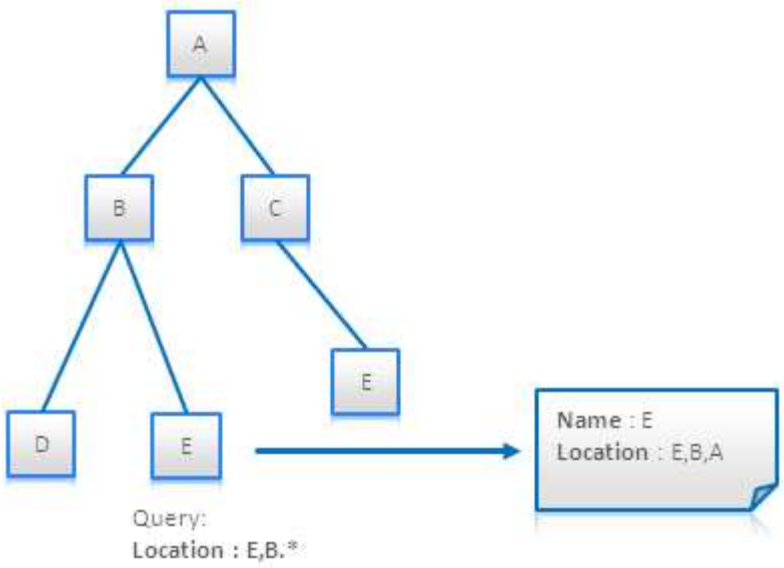
Materialized Paths 可以帮助避免递归遍历（如：树形结构）。这个技术也可以被认为是反规格化的一种变种。其想法是为每个结点加上父结点或子结点的标识属性，这样就可以不需要遍历就知道所有的后裔结点和祖先结点了：



Materialized Paths for eShop Category Hierarchy

这个技术对于全文搜索引擎来说非常有帮助，因为其可以允许把一个层级结构转成一个文档。上面的示图中我们可以看到所有的商品或 *Men's Shoes* 下的子分类可以被一条很短的查询语句处理——只需要给定个分类名。

Materialized Paths 可以存储一个 ID 的集合，或是一堆 ID 拼出的字符串。后者允许你通过一个正则表达式来搜索一个特定的分支路径。下图展示了这个技术（分支的路径包括了结点本身）：

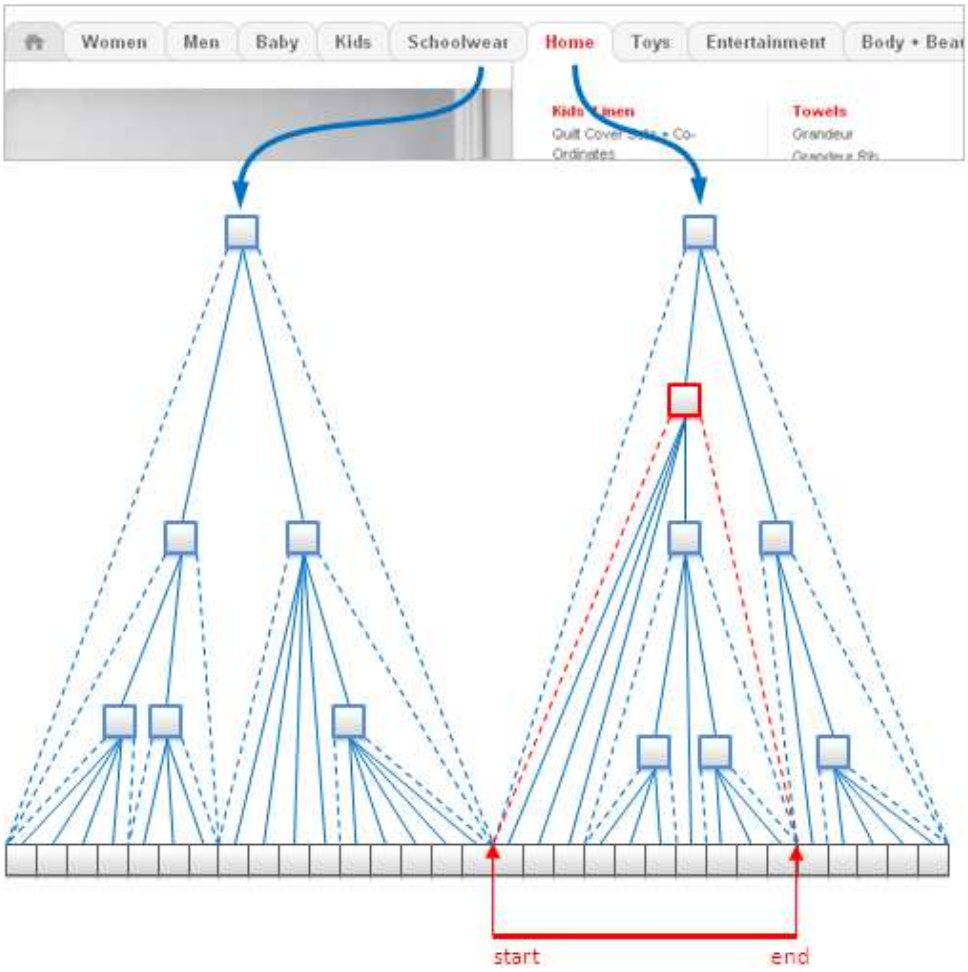


Query Materialized Paths using RegExp

适用性: Key-Value 键值对数据库, Document Databases 文档数据, Search Engines 搜索引擎

(14) 嵌套集 Nested Sets

Nested sets 嵌套集是树形结构的标准技术。它被广泛地用在了关系性数据库中，它完全地适用于 Key-Value 键值对数据库 和 Document Databases 文档数据库。这个技术的想法是把叶子结点存储成一个数组，并通过使用索引的开始和结束来映射每一个非叶子结点到一个叶子结点集，就如下图所示一样：



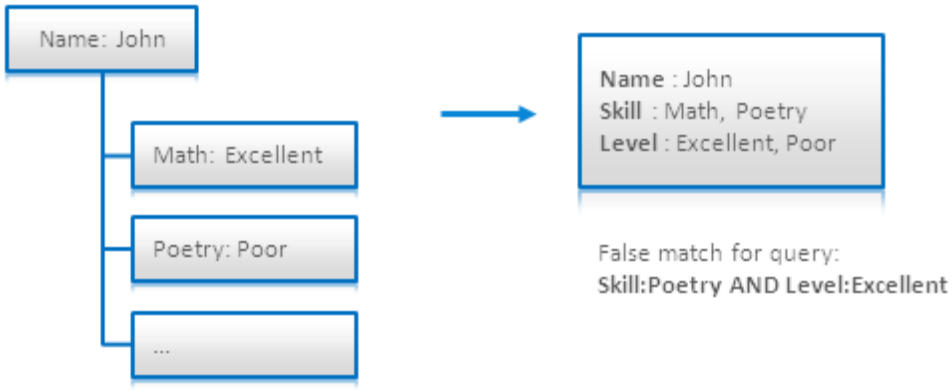
Modeling of eCommerce Catalog using Nested Sets

这样的数据结构对于 immutable data 不变的数据 有非常不错的效率，因为其点内存空间小，并且可以很快地找出所有的叶子结点而不需要树的遍历。尽管如此，在插入和更新上需要很高的性能成本，因为新的叶子结点需要大规模地更新索引。

适用性: Key-Value Stores 键值数据库, Document Databases 文档数据库

(15) 嵌套文档扁平化：有限的字段名 Nested Documents Flattening: Numbered Field Names

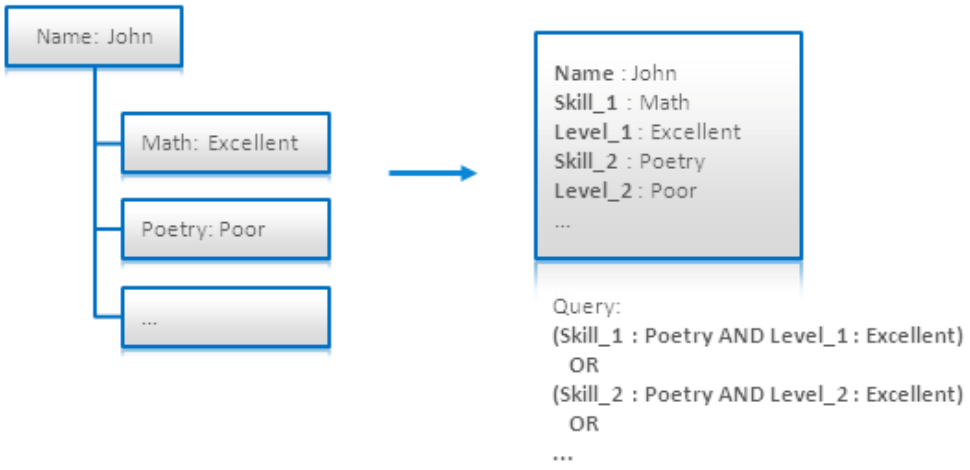
搜索引擎基本上来说和扁平文档一同工作，如：每一个文档是一个扁平的字段和值的例表。这种数据模型的用来把业务实体映射到一个文本文档上，如果你的业务实体有很复杂的内部结构，这可能会变得很有挑战。一个典型的挑战是把一个有层级的文档映映射出来。例如，文档中嵌套另一个文档。让我们看看下面的示例：



Nested Documents Problem

上面的每一个业务实体代码一种简历。其包括了人名和一个技能列表。我把这个层级文档映射成一个文本文档，一种方法是创建 *Skill* 和 *Level* 字段。这个模型可以通过技术或是等级来搜索一个人，而上图标注的那样的组合查询则会失败。（陈皓注：因为分不清 Excellent 是否是 Math 还是 Poetry 上的）

在引用中的［4.6］给出了一种解决方案。其为每个字段都标上数字 *Skill_i* 和 *Level_i*，这样就可以分开搜索每一个对（下图中使用了 OR 来遍历查找所有可能的字段）：



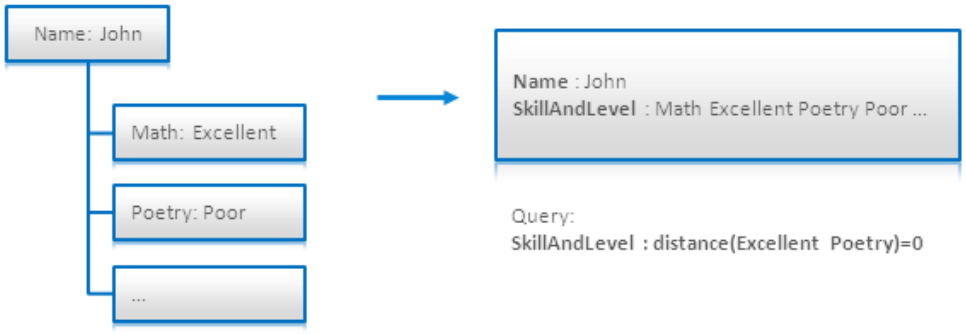
Nested Document Modeling using Numbered Field Names

这样的方式根本没有扩展性，对于一些复杂的问题来说只会让代码复杂度和维护工作变大。

适用性: Search Engines 全文搜索

(16)嵌套文档扁平化：邻近查询 Nested Documents Flattening: Proximity Queries

在附录［4.6]中给出了这个技术用来解决扁平层次文档。它用邻近的查询来限制可被查询的单词的范围。下图中，所有的技能和等级被放在一个字段中，叫 SkillAndLevel，查询中出现的“Excellent” 和 “Poetry” 必需一个紧跟另一个：



Nested Document Modeling using Proximity Queries

附录［4.3］中讲述了这个技术被用在 Solr 中的一个成功案例。

适用性: Search Engines 全文搜索

(17) 图结构批处理 Batch Graph Processing

Graph databases 图数据库，如 neo4j 是一个出众的图数据库，尤其是使用一个结点来探索邻居结点，或是探索两个或少量结点前的关系。但是处理大量的图数据是很没有效率的，因为图数据库的性能和扩展性并不是其目的。分布式的图数据处理可以被 MapReduce 和 Message Passing pattern 来处理。如： 在我前一篇的文章中的那个示例。这个方法可以让 Key-Value stores, Document databases, 和 BigTable-style databases 适合于处理大图。

Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

参考