分布式数据库灵活存储机制与实现

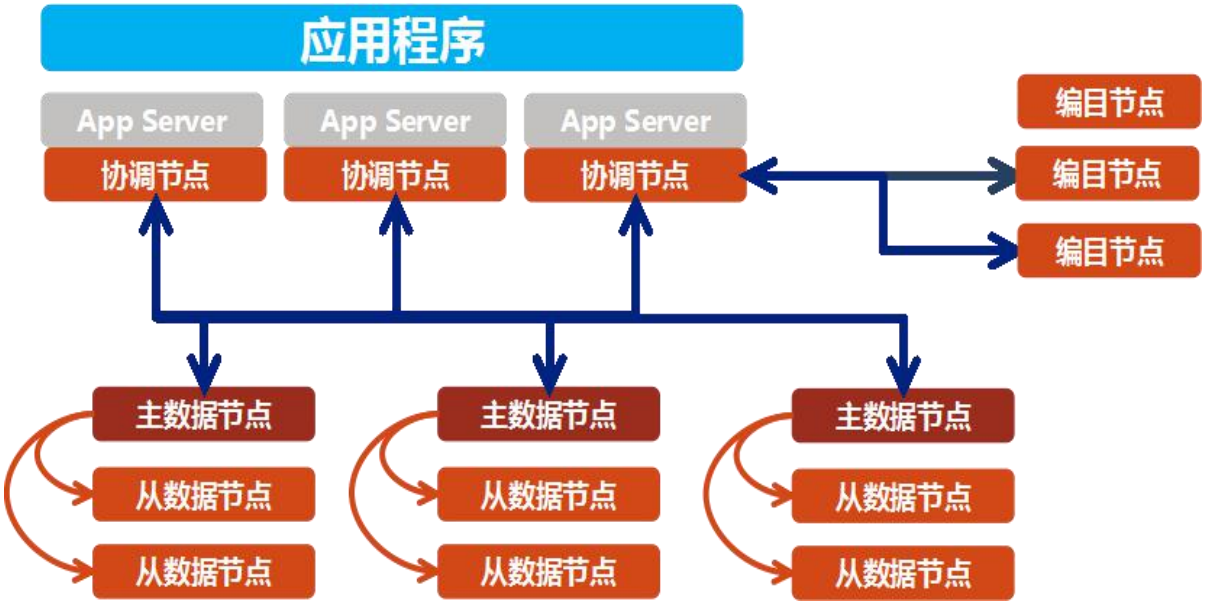
SequoiaDB
巨杉数据库

CONTENTS
目录

- 分布式架构与分区机制
- JSON/BSON 记录存储机制
- 文件存储/LOB大对象机制
- 查询性能的优化机制

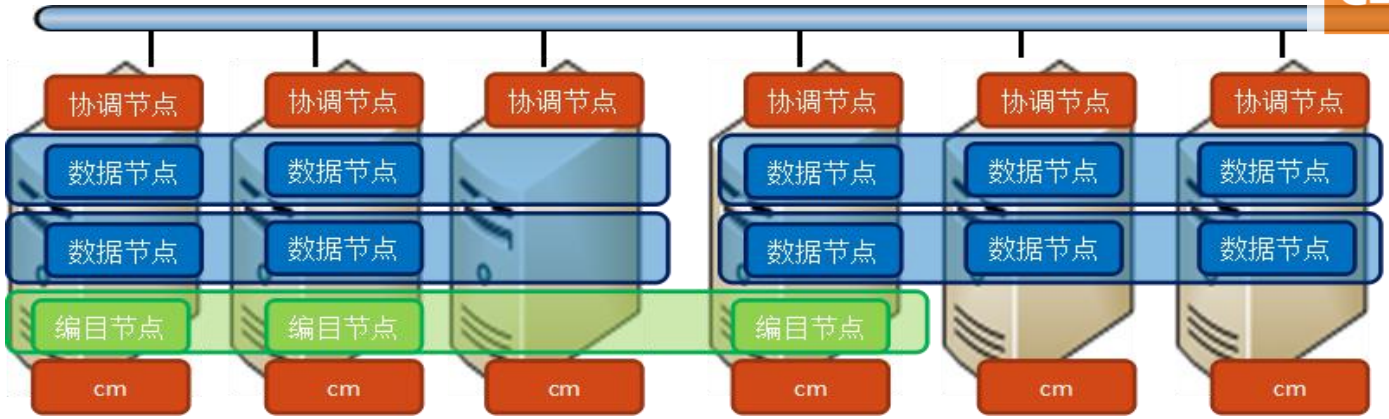
分布式架构与数据分区机制

分布式数据库架构



通过动态增加复制组数量达到水平扩张的目的

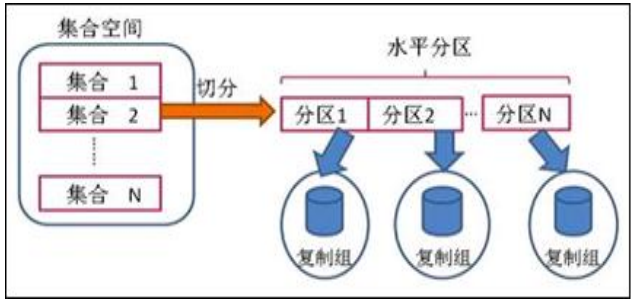
物理架构

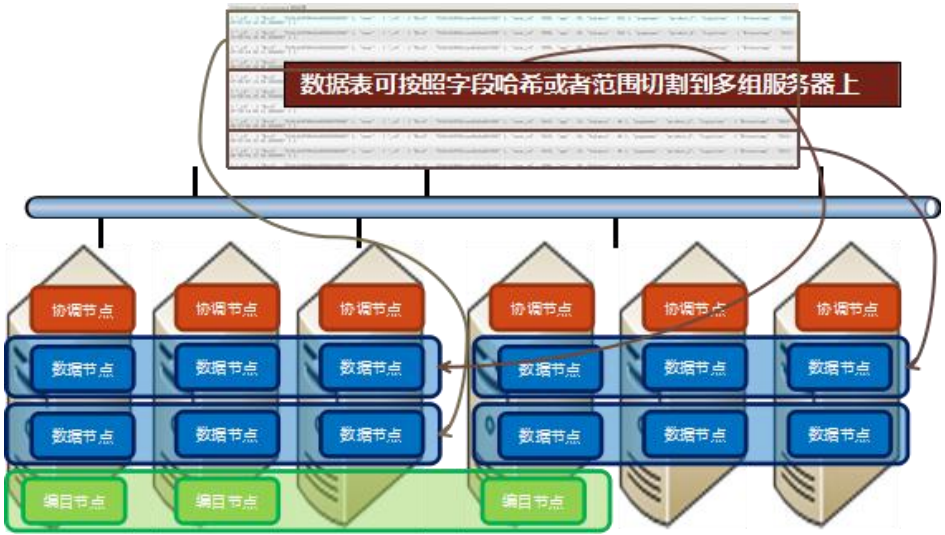


角色	功能
协调节点	胖客户层，从编目读取数据分布信息，从数据节点读取数据
编目节点	负责元数据信息存储，包括组信息、表切割信息
数据节点	负责数据表存储，提供查询、聚集、数据复制功能
CM节点	负责集群管理，包括watchdog, 节点增删启停

数据水平分区

SequoiaDB支持水平分区和垂直分区。水平分区尽可能选择唯一性较高的字段





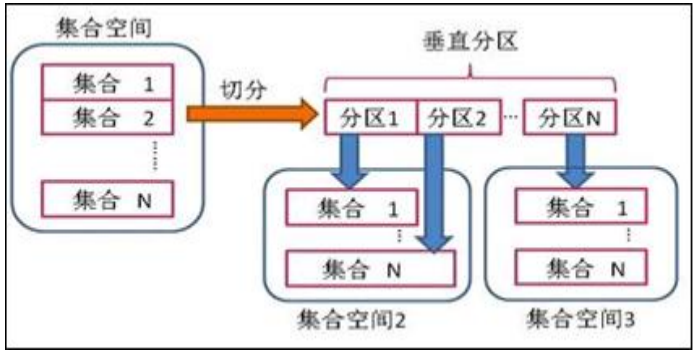
优势： 容量和性能可线性扩展

数据垂直分区

垂直分区选择记录生成的时间戳作为分区字段

-保障最新的一段时间数据尽可能驻留内存

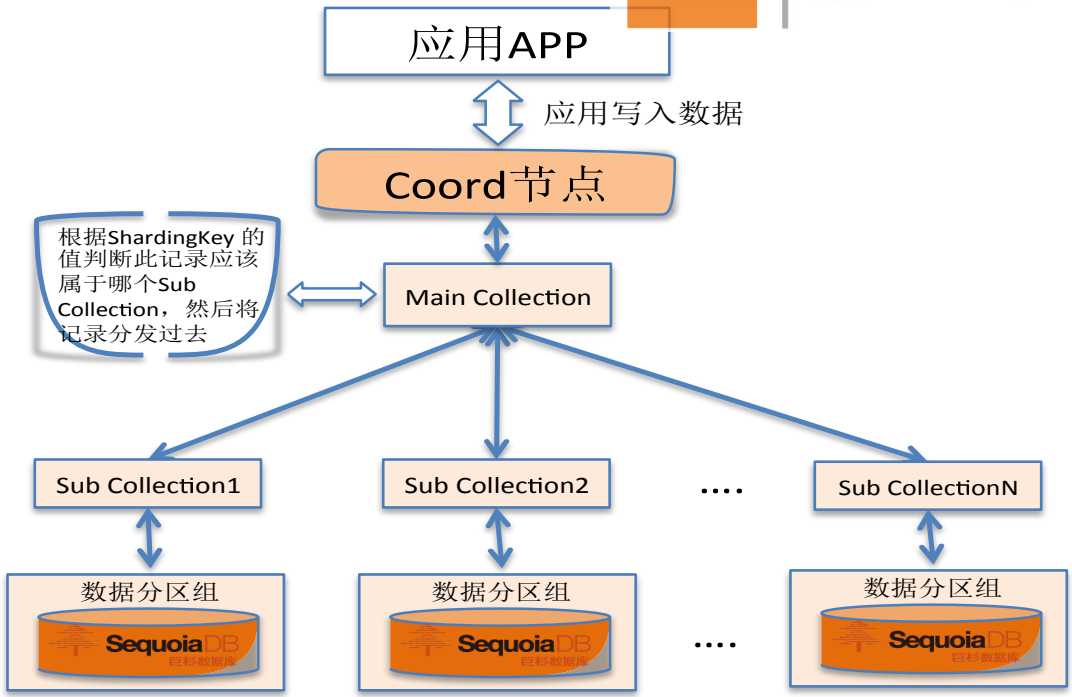
-可以按照时间范围快速删除数据



优势： 容量和性能可线性扩展

举例：Partition 分区机制

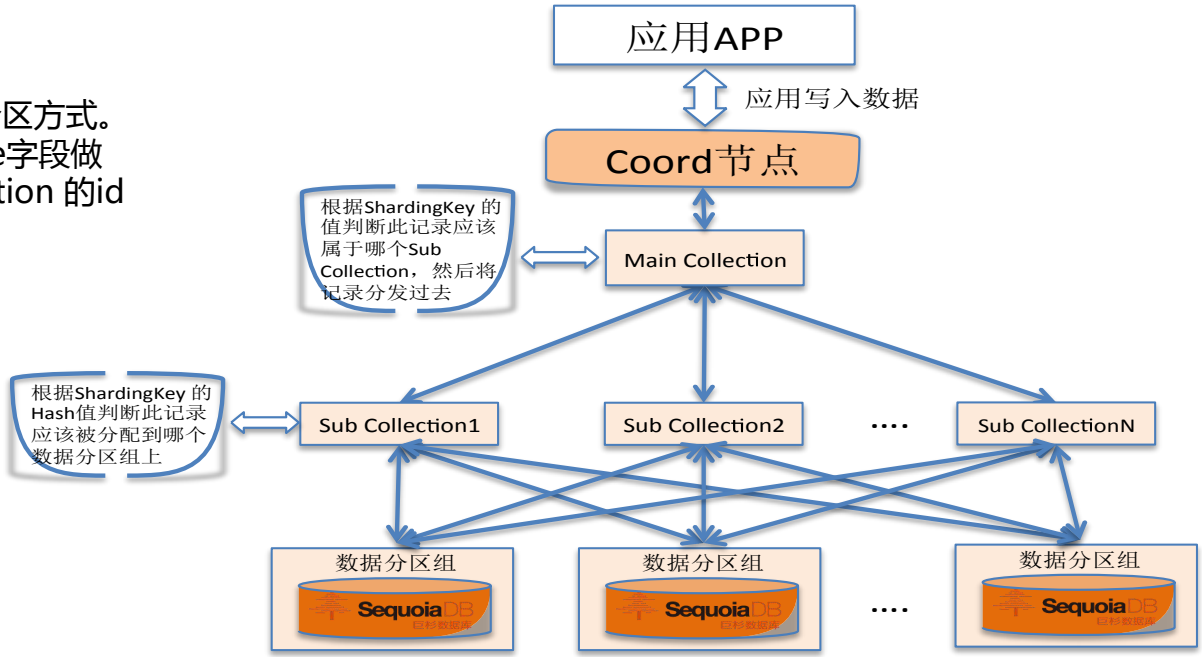
这个功能与部分关系型数据库的Partition功能类似，都是在数据库中建立一张逻辑的总视图，然后将多个Partition通过某个字段的范围限定挂载到总视图上。
main collection为逻辑视图，只在编目节点中保留一些数据范围信息，而sub collection 则是数据库中普通建立的集合，只是在配合main collection 一起使用时，才被称呼为sub collection。



举例：多维分区机制

在多种分布式分区原理上，最为复杂和最为高效的方法，就是数据多维分区。多维分区，顾名思义，就是在对集合做数据分区时，多种分区方式同时作用在一个集合上。

如图所示
多维分区可以为一个集合同时提供两种分区方式。一般情况下，用户可以在主子表中对time字段做（垂直）范围切分，然后再对sub collection 的id字段做（水平）Hash切分。

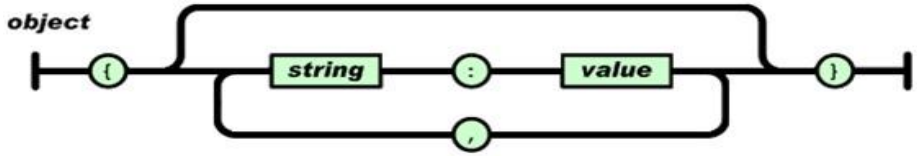


JSON/BSON 记录存储

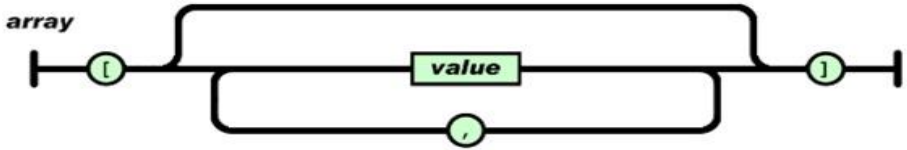
JSON 数据结构

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，它基于ECMAScript的一个子集，为纯文本格式，支持嵌套结构与数组。

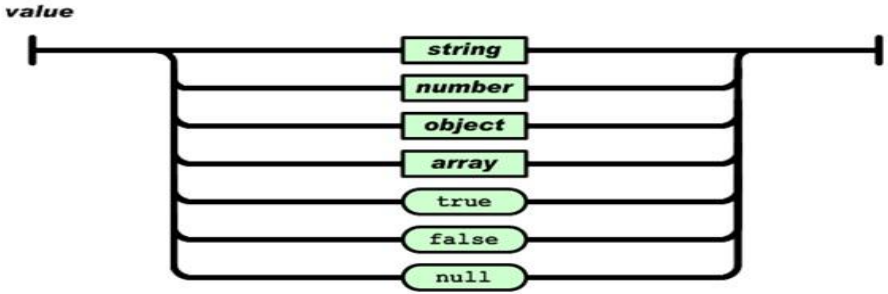
1、对象是一个无序的“键值对”集合，以“{”（左大括号）开始，“}”（右大括号）结束。每一个元素名后跟一个“:”（冒号）；而元素之间使用“,”（逗号）分隔；



2、数组是值的有序集合，以“[”（左中括号）开始，“]”（右中括号）结束。值之间使用“,”（逗号）分隔；

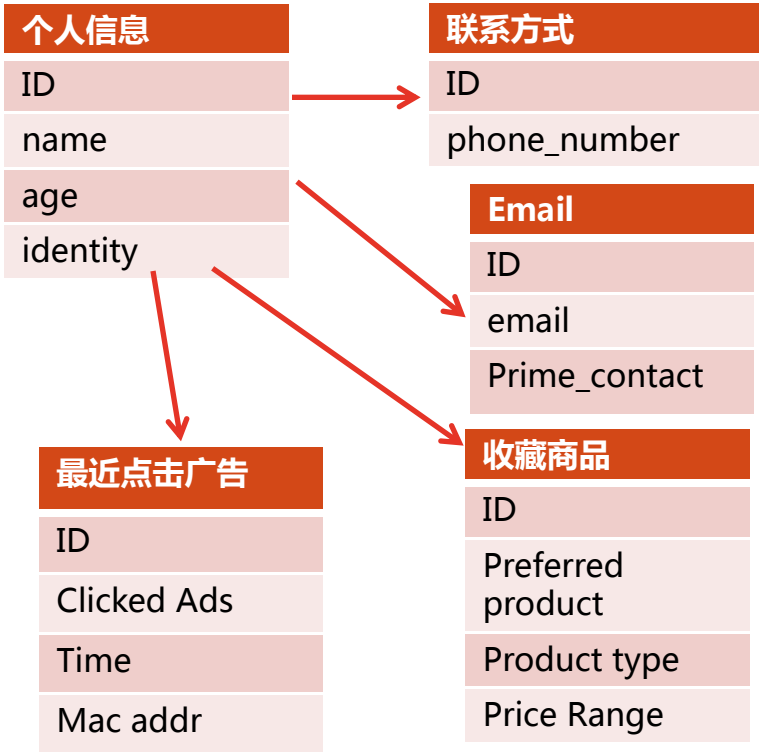


3、值可以为由双引号包裹的字符串，数值，对象，数组，true，false，null，以及特有的数据结构（例如日期，时间等）组成。



去范式的场景：对象存储

层次化嵌套形式实现一对多模型转换



个人信息	
ID	
name	
age	
identity	
电话列表[…]	phone_number
Email列表[…]	Email [email1, email2, …]
	Prime_contact
收藏商品列表 […]	Preferred product
	Product type […]
	Price Range
最近点击广告列表[…]	Clicked Ads
	Time
	Mac Addr

简化关系模型中的关联关系

Customer ID	First Name	Last Name	City
0	John	Doe	New York
1	Mark	Smith	San Francisco
2	Jay	Black	Newark
3	Meagan	White	London
4	Edward	Daniels	Boston

Account Number	Branch ID	Account Type	Customer ID
10	100	Checking	0
11	101	Savings	0
12	101	IRA	0
13	200	Checking	1
14	200	Savings	1
15	201	IRA	2

```
{     customer_id : 1,
      first_name : "Mark",
      last_name : "Smith",
      city : "San Francisco",
      accounts : [    {
            account_number : 13,
            branch_ID : 200,
            account_type :
"Checking"
      },
      {     account_number : 14,
            branch_ID : 200,

      account_type : "Savings",
            beneficiaries: [...]
      } ]
}
```

关系型数据结构

文档型数据结构

历史数据中同一个表的不同年份数据需要放在一起

例子：社保+商业保险的关联汇总

每年的数据表结构都会变化，历史贴源数据要包容变化的数据结构。

时间	IDh	年收入（元）	累计标保和	缴付保费合计	关联帐户	寿险缴付金额	两全缴付金额	年金缴付金额	医疗缴付金额	意外缴付金额
时间	IDh	年收入（元）	累计标保和	缴付保费合计	关联帐户	寿险缴付金额	两全缴付金额	年金缴付金额	医疗缴付金额	意外缴付金额
时间	IDh	年收入（元）	累计标保和	缴付保费合计	关联帐户	寿险缴付金额	两全缴付金额	年金缴付金额		

需要每年动态增加

时间	IDh	年收入（元）	累计标保和	缴付保费合计	关联帐户	寿险缴付金额	两全缴付金额	年金缴付金额	医疗缴付金额	意外缴付金额	交通缴付金额
2012	XX47	48200	26900	80400	A1212	500	5000	0	Null	Null	Null
2012	XX47	48200	26900	80400	A1213	0	0	76000	Null	Null	Null
2012	XX51	29400	15000	96600	A2039	500	90400	5800	Null	Null	Null
2013	XX40	180600	10800	21300	A3359	1100	3800	12400	100	800	Null
2013	XX37	29400	7400	71100	A6596	600	5100	1500	100	500	Null
2014	XX33	28200	6500	30700	A8767	200	800	29300	100	100	300
2014	XX53	33600	5800	80500	A9785	200	5300	2500	100	200	200
2015	XX42	97400	5300	9800	A9078	400	3300	3000	100	300	400

历史数据的动态存储方式

时间	IDh	年收入（元）	累计标保和	缴付保费合计	关联帐户	寿险缴付金额	两全缴付金额	年金缴付金额
2012	XX47	48200	26900	80400	A1212	500	5000	0
2012	XX47	48200	26900	80400	A1213	0	0	76000

```
{     Time : 2012,
      IDh : “xx47”,
      Income : 48,200,
      Acc_Sum : 26,900,
      Invoice_Sum : 80,400,
      accounts : [    {
            account_number : A1212,
            life_insurance: 500,
            labor_insurance: 5000,
            pension:0,
      },
      {     account_number : A1213,
            life_insurance: 0,
            labor_insurance: 0,
            pension:76,000,
      } ]
}
```

时间	IDh	年收入（元）	累计标保和	缴付保费合计	关联帐户	寿险缴付金额	两全缴付金额	年金缴付金额	医疗缴付金额	意外缴付金额	交通缴付金额
2014	XX33	28200	6500	30700	A8767	200	800	29300	100	100	300

```
{     Time : 2014,
      IDh : “xx33”,
      Income : 28,200,
      Acc_Sum : 6,500,
      Invoice_Sum : 30,700,
      accounts : [    {
            account_number :
A8767,
            life_insurance: 200,
            labor_insurance: 800,
            pension:29300,
            health_care:100,
            accident: 100
            transportation:300
      }]
}
```

文件存储 与 LOB机制

BSON和LOB按需搭配使用

缩略图 - BSON Binary

•{id:”001”,time:”2015120
1”,...photo:”aGVsbG8gd29ybGQh”}

•Binary只是BSON中的一个字段类型，以行的方式直接存储。对于数据库来讲，这就是一条普通记录。所以访问时不会产生额外的IO。

高清图 - LOB

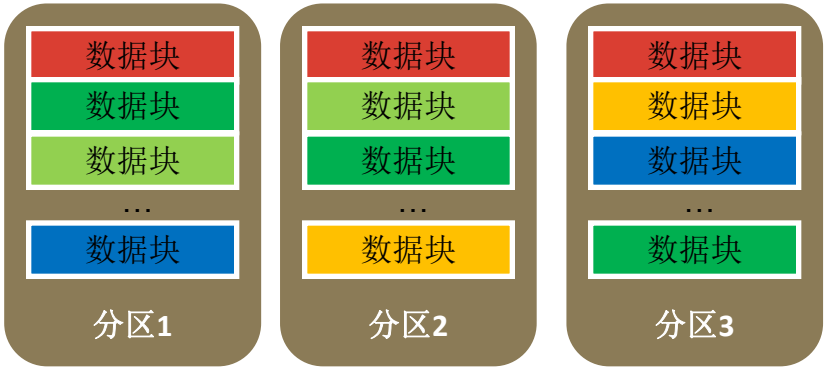
•单独的分布式块存储模式

•文件按照LOB处理

•自动按照64/128KB的数据块进行切分，放在不同分区存储

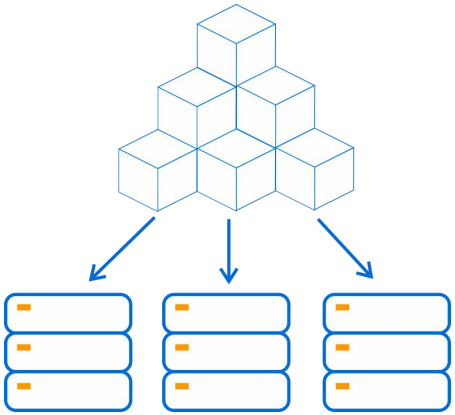
•使用DIO避免二进制数据占用文件系统缓存

•并行处理



LOB 块存储机制

由于不存在独立中控元数据节点，LOB存储机制理论上可以存放近乎无限数量的对象文件，并且不会由于元数据堆积而造成性能下降。同时，由于数据块被散列分布到所有数据节点，整个系统的吞吐量随集群磁盘数量的增加近乎线性提升。



文件按照LOB处理

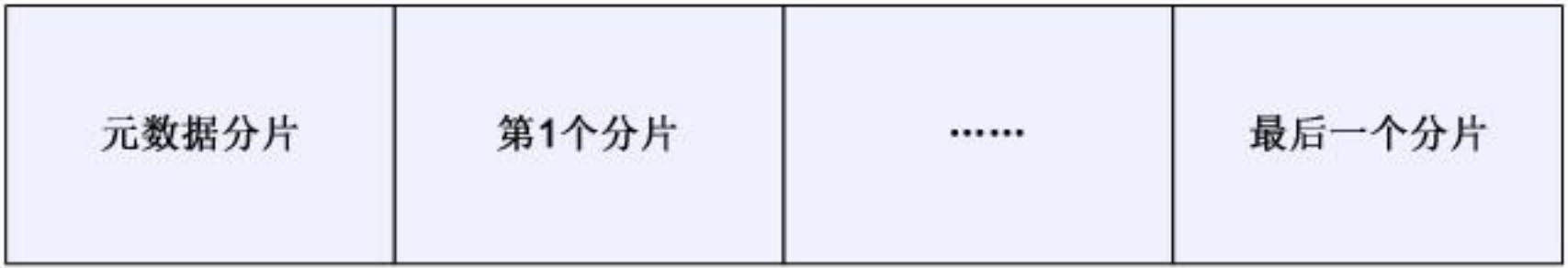
- 自动按照64/128KB的数据块进行切分，放在不同分区存储
- 使用DIO避免二进制数据占用文件系统缓存
- 并行处理

- 与GridFS相比不占用内存
- 与HDFS相比不存在Namenode限制

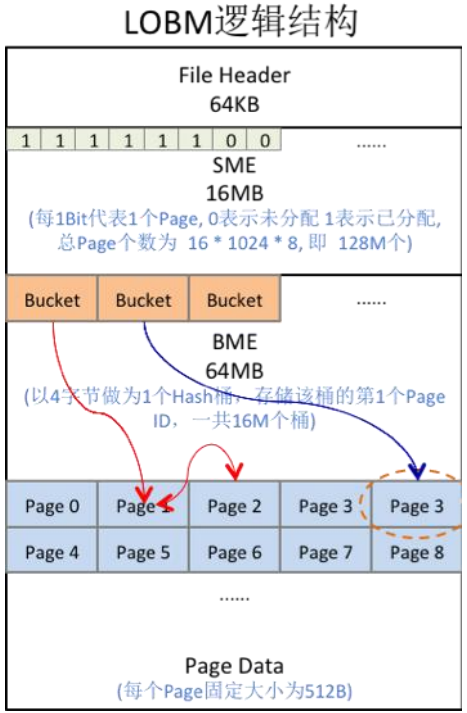
LOB 块存储机制

大对象（LOB）功能旨在突破普通JSON数据库，单条数据最大长度为 16MB 的限制，为用户写入和读取更大型记录提供便利。LOB 记录的大小目前不受限制。

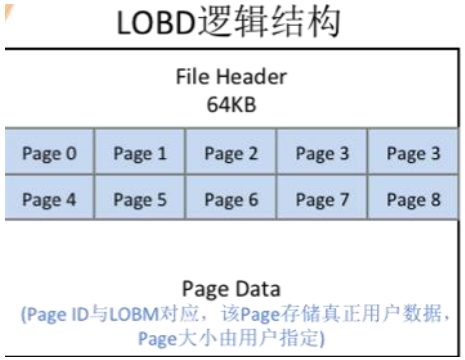
每一个 LOB 记录拥有一个 OID，通过指定集合及 OID 可以访问一条 LOB 记录。在非分区集合及哈希分区集合中均可使用 LOB 功能。集合间不共享 LOB 记录。当一个集合被删除时，其拥有的 LOB 记录自动删除。
LOB 记录的存储格式：



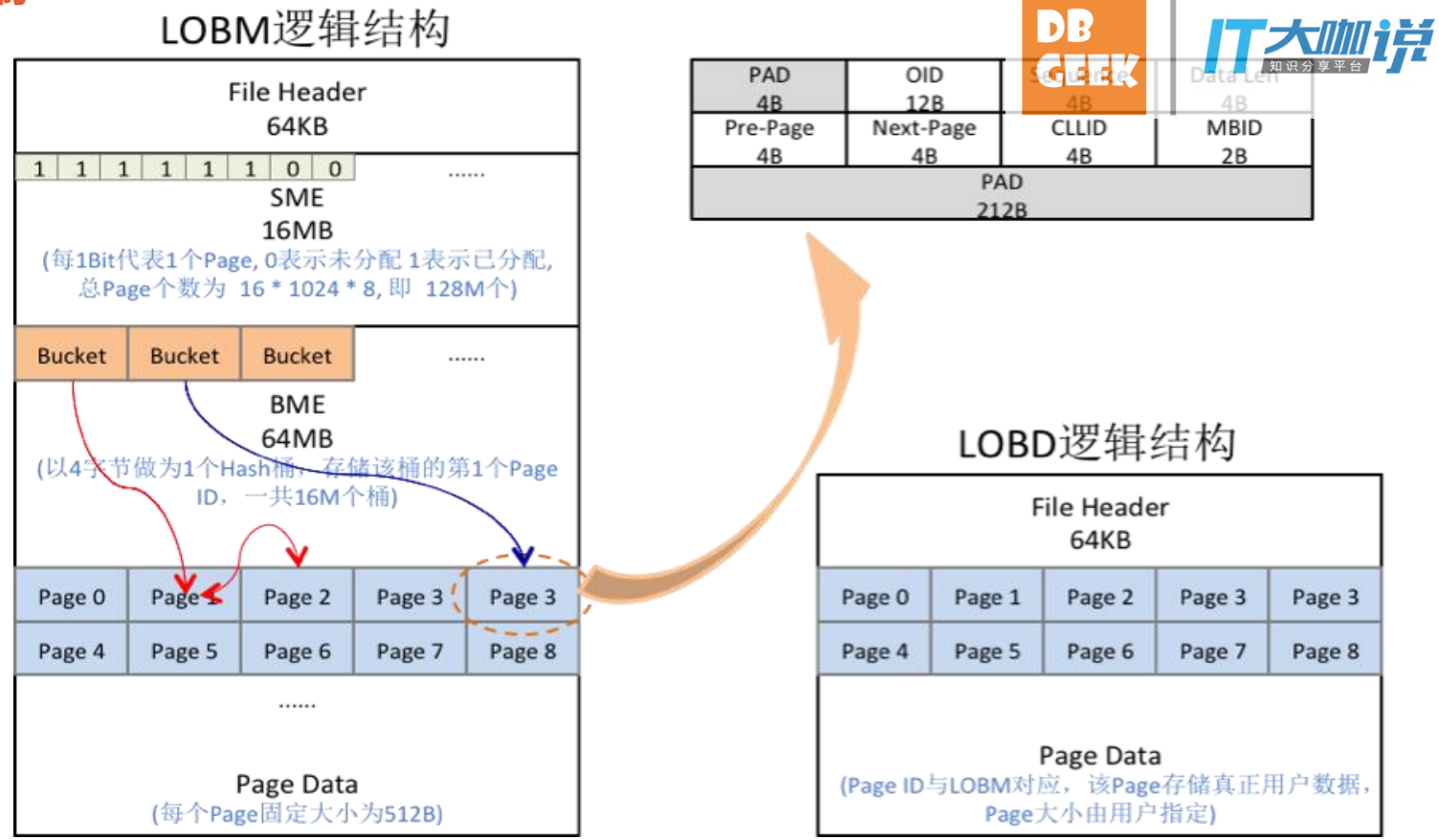
LOB 块存储机制



LOB存储结构分为元数据文件（LOBM）与数据文件（LOBD）。其中，元数据文件存储整个LOB数据文件的元数据模型，包括每个页的空闲状况、散列桶、以及数据映射表等一系列数据结构。而数据文件则存储用户真实数据，数据头之后所有数据页按照page size进行切分，每个数据页不包含任何元数据信息。在建立集合的过程当中，大对象存储必须依附于普通集合存在，一个集合中的大对象仅归属于该集合，不能被另外一个集合管理。



LOB 映射机制



对象存储机制：兼顾海量小文件和大文件的存储特性

- 其他实现方式
- 关系数据库+文件系统地址

问题：文件条目受关系型数据库的性能限制，比如超过3亿条后性能急剧下降

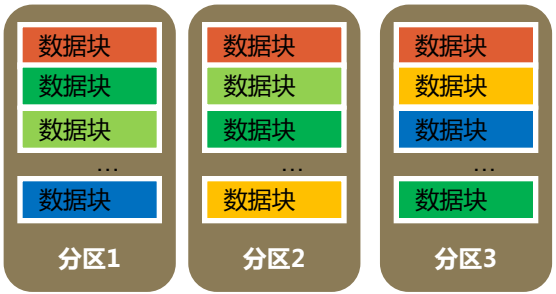
- HDFS

问题：受 Namenode限制无法处理大量的小文件，分配64MB存储浪费空间；小文件不定期后台自动聚合，影响系统的使用稳定性。

- HBase

问题：做Merge的过程造成I/O飙高，无法满足在线ECM服务场景

- 分布式对象存储
- 文件按照数据块处理
  -自动按照64/128KB的数据块进行
  -切分，放在不同分区存储
- 使用DIO避免二进制数据占用文件系统缓存并行处理
  -与GridFS相比不占用内存
  -与HDFS相比不存在Namenode限制



DB GEEK

IT大咖说 知识分享平台

查询性能优化

SequoiaDB

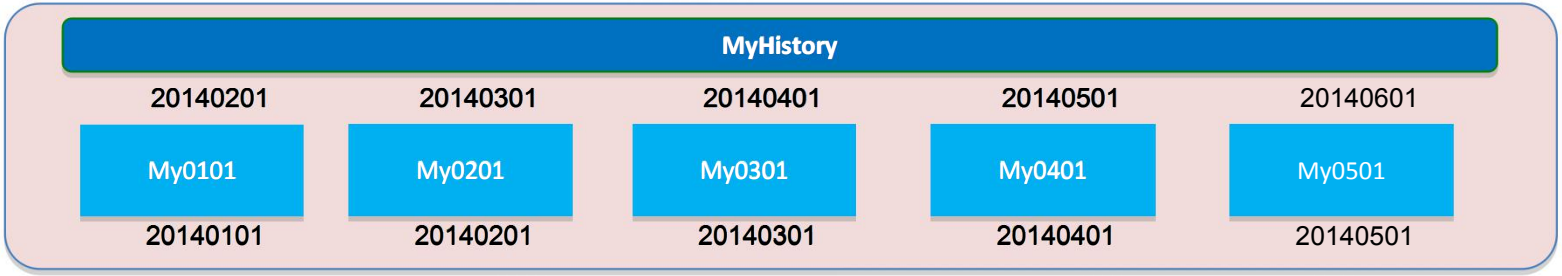
主子集合/时间序列机制

- 时间序列数据的处理是常见需求，很多应用中海量数据的时间特性以递增为主，且数据的热度随着时间的推移递减
- 集合分区（主子集合）机制可以轻松应对时间序数据
  - 避免单一集合数据量膨胀时索引树过大而导致的写入性能雪崩
  - 按时间序能直观反映数据访问热点，保障热点数据集合的性能
  - 直观的分配资源给不同集合，直观的备份、归档规则

```
db.cs.createCL( “My0101” )
db.cs.MyHistory.attachCL(
    “cs.My0101” ,
   {UpBound:{date:” 20140201” },
    LowBound:{date:” 20140101” }}
)
```

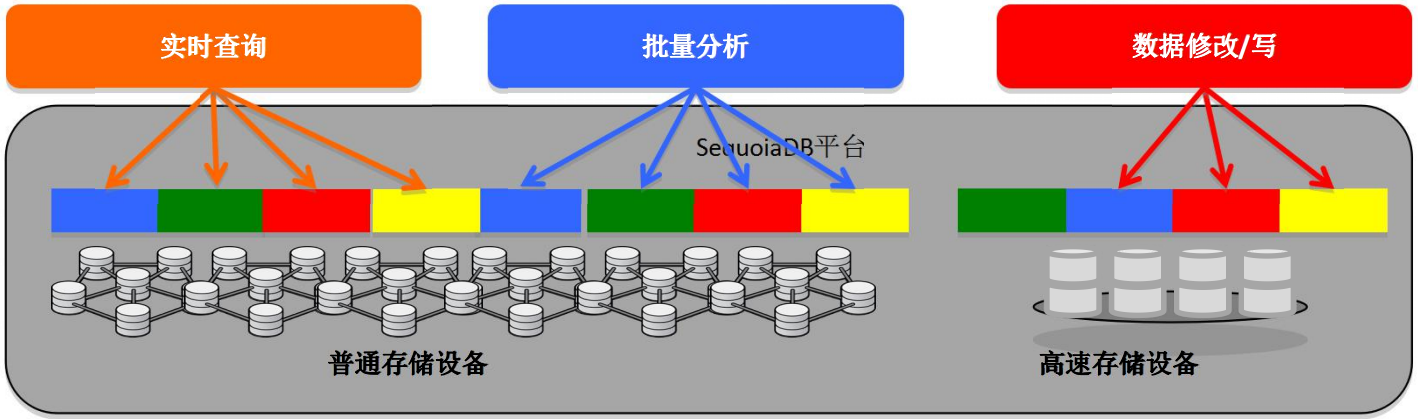
```
db.cs.createCL( “My0201” )
db.cs.MyHistory.attachCL(
    “cs.My0201” ,
   {UpBound:{date:” 20140301” },
    LowBound:{date:” 20140201” }}
)
```

```
db.cs.createCL( “My0301” )
db.cs.MyHistory.attachCL(
    “cs.My0301” ,
   {UpBound:{date:” 20140401” },
    LowBound:{date:” 20140301” }}
)
```



读写分离及自定制数据分布策略

- 数据在多个分布节点内自动复制，并实现写请求和读请求的自动分离，避免读请求对数据写入的影响。
- 此外，可进一步定制数据分布策略，保证不同类型业务可以运行在同一平台上，但同时又不会互相干扰，比如：
  - 冷/热数据区分离
  - 写交易的“强一致性”和“弱一致性”分离
  - 查询/批量分离



SequoiaDB 2.8 即将发布
关注微信号获取最新推送

SequoiaDB 巨杉数据库
www.sequoiadb.com
Sales_support@sequoiadb.com
hr@sequoiadb.com
400-8038-339