

# 复习

考试内容

大数据管理

关系型系统

大数据存储系统

大数据运算系统

大数据分析

DR, 数据降维

LSH, 局部敏感哈希

流数据处理

蒙特卡洛

## 考试内容

### 大数据分析，30分

9道题，打勾

LSH, SVD要弄清楚

贝叶斯公式，简单的概率

知道每个算法是干什么的

概率转移矩阵的行和等于1

### 大数据管理，70分

全是打勾的，比如选择，判断，七选五

## 大数据管理

### 关系型系统

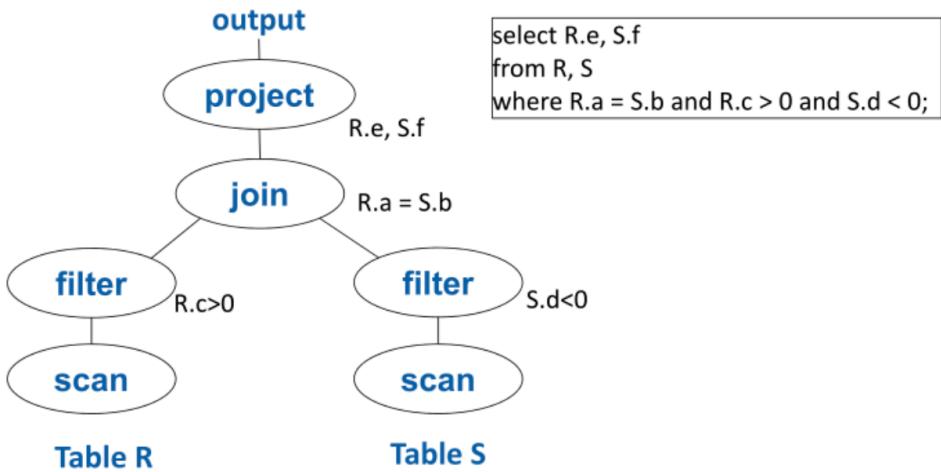
#### 关系模型和关系运算

# SQL Select

<b>select</b> 列名,...,列名	投影
<b>from</b> 表,..., 表	选择, 连接
<b>where</b> 条件	选择, 连接
<b>group by</b> 列名,...,列名	分组统计
<b>having</b> 条件	分组后选择
<b>order by</b> 列名,...,列名	结果排序

## 数据库的内部实现

- 数据库的表是如何在磁盘上存储的
- 两种索引，哈希索引和B+树索引，B+树索引新节点的插入和旧节点的删除
- 范围查询时，根据范围大小，查询优化器决定是直接顺序访问还是走索引
- 缓冲的四种替换策略
  - 随机替换
  - 先进先出
  - LRU（使用哈希链表）
  - CLOCK，就是缓存命中置1，扫描到置0，如果一圈下来还是0，说明一圈的时间都没有命中，直接将其选为Victim
- 查询优化器最终生成一颗运算树



- join的三种思路
  - 嵌套循环，可以理解为两重for循环
  - 哈希，扫描R时走S的索引，或者扫描R创建索引然后扫描S走索引，或者分别先分片，对分片后的部分分别join
  - 排序，R和S分别排序，然后归并，如果放不进内存，则分别按照M个页排序组成一个个run，然后对这些run进行归并

## 事务处理和数据仓库

### 事务处理

## 数据冲突引起的问题

- Read uncommitted data (读脏数据) (写读)
  - 在T2 commit之前，T1读了T2已经修改了的数据
- Unrepeatable reads(不可重复读) (读写)
  - 在T2 commit之前，T1写了T2已经读的数据
  - 如果T2再次读同一个数据，那么将发现不同的值
- Overwrite uncommitted data (更新丢失) (写写)
  - 在T2 commit之前，T1重写了T2已经修改了的数据

### 1. 读未提交 (Read Uncommitted)

- 含义：一个事务可以读取到其他事务尚未提交的数据。这是最低的隔离级别，它允许事务之间的最大并发度，但牺牲了数据的一致性。

- **解决的问题：**几乎不解决任何一致性问题，反而会引发多种问题。比如脏读（Dirty Read），即一个事务读取到了另一个事务未提交的数据；还可能导致不可重复读（Non - Repeatable Read）和幻读（Phantom Read）。例如，事务 A 修改了数据但未提交，事务 B 此时读取到了事务 A 未提交的数据，这就是脏读。

## 2. 读提交（Read Committed）

- **含义：**事务只能读取已经提交的事务所做的修改。许多数据库系统将其作为默认隔离级别。
- **解决的问题：**解决了脏读问题，确保事务不会读取到其他事务未提交的数据。但仍存在不可重复读问题，即一个事务在执行过程中，同一查询可能返回不同结果，因为在两次读取之间，其他事务可能已经提交并修改了数据；也可能出现幻读问题，即同一查询可能返回不同数量的行。

## 3. 可重复读（Repeatable Read）

- **含义：**在一个事务执行期间，对同一数据的读操作始终返回相同结果，不管其他事务是否对该数据进行了修改并提交。
- **解决的问题：**解决了脏读和不可重复读问题。通过在读取数据时对其加锁或使用多版本并发控制（MVCC）机制来实现。不过，在某些数据库实现中，可能仍存在幻读问题，即事务在执行期间，同一范围查询可能返回不同数量的行。但像 MySQL 的 InnoDB 存储引擎在可重复读隔离级别下，使用 Next - Key Lock 算法避免了幻读的产生。

## 4. 串行化（Serializable）

- **含义：**最高的隔离级别，事务会按照一定顺序串行执行，确保在任何时刻只有一个事务可以访问数据。
- **解决的问题：**解决了所有常见的并发一致性问题，包括脏读、不可重复读和幻读。因为事务是串行执行的，不存在并发干扰，但这导致了最低的并发性能，大量事务可能需要等待其他事务释放锁。
- 并发控制的方法
  - 悲观，2PL，先集中加锁，然后执行，然后集中解锁，隔离级别是可串行化
  - 乐观，快照隔离，读快照数据，记录写结果，检查有无冲突，无冲突 commit，有冲突 abort，隔离级别是可重复读
- 持久性通过WAL（写前写日志）实现

## 数据仓库

- OLAP，在线分析处理，基本数据模型是多维矩阵

## 分布式数据库

- 三种架构
  - shared memory，适合并行处理，频繁交换
  - shared disk，适合数据一致性要求高
  - shared nothing，适合大数据，高并发
- 分布式事务，2PC
  - 第一阶段，协调者向每个参与者发送query to commit消息，每个参与者回复yes or no
  - 第二阶段，当第一阶段所有回答都是yes，协调者向每个参与者发送commit消息，每个参与者提交事务回复acknowledgment
  - 第三阶段，当第一阶段有一个回答是no，协调者向每个参与者发送abort消息，每个参与者回滚事务然后回复acknowledge

## 大数据存储系统

### 分布式文件系统

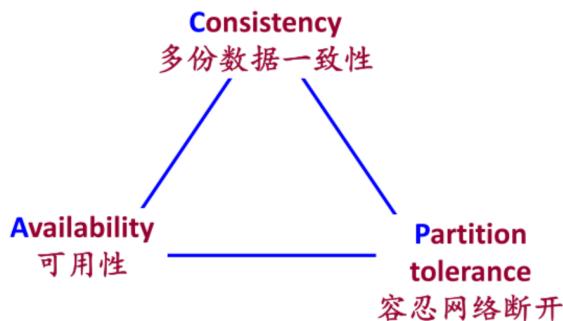
## 分布式系统类型

- Client / Server
  - 客户端发送请求，服务器完成操作，发回响应
  - 例如：3-tier web architecture
    - Presentation: web server
    - Business Logic: application server
    - Data: database server
- P2P (Peer-to-peer)
  - 分布式系统中每个节点都执行相似的功能
  - 整个系统功能完全是分布式完成的
  - 没有中心控制节点
- Master / workers
  - 有一个/一组节点为主，进行中心控制协调
  - 其它多个节点为workers，完成具体工作

## 故障模型(Failure Model)

- Fail stop
  - 当出现故障时，进程停止/崩溃
- Fail slow
  - 当出现故障时，运行速度变得很慢
- Byzantine failure
  - 包含恶意攻击

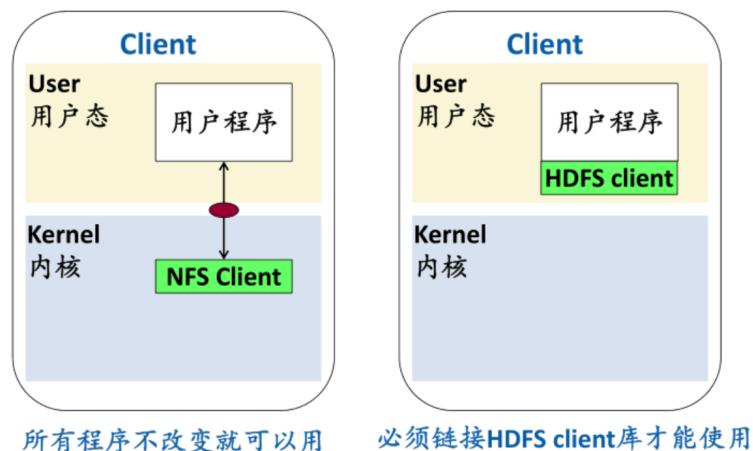
## CAP定理



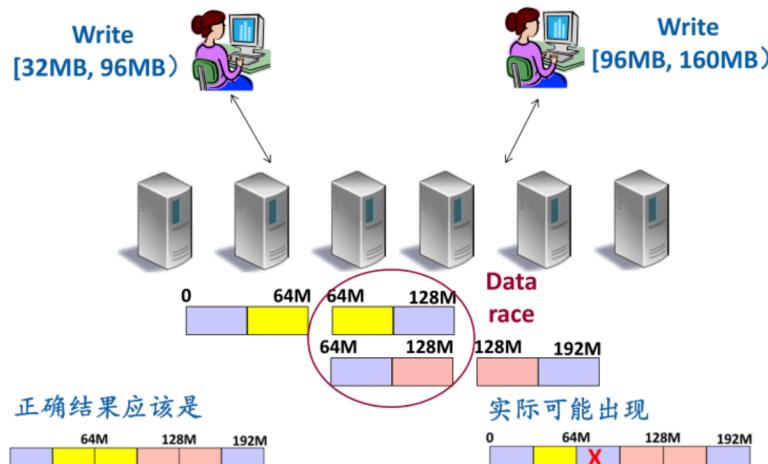
三者不可得兼

- NFS，C/S架构，无状态的，读写都是幂等的，以数据页为单位，多个client读取同一文件，本地缓存（在内存中）文件，每次关闭文件时把修改写入server，每次打开时get最新的文件属性并比较来判断本地缓存是否过期
- AFS，相较于NFS，使用了观察者模式，当server更新时，向所有登记的client发送callback，以文件为单位，缓存是在本地硬盘，所以可以缓存大文件，有统一命名空间和详细的权限管理
- GFS
- HDFS，GFS的开源实现，很好的顺序读性能，不支持并发写，但支持并发追加（不保证追加的顺序）

## POSIX文件系统 vs. 应用层文件系统



## HDFS/GFS文件操作：不支持并发写



## 键值存储、zookeeper

### Dynamo

quorum机制，数据存放在N个节点，每次必须成功写W个节点，每次必须读R个节点，这样就能保证读到最新的数据

## Dynamo小结

- 最简单的<key,value>模型，get/put操作
- 单节点上存储由外部存储系统实现
- 多节点间的数据分布
  - Consistent hashing
  - Quorum (N, W, R)
  - Eventual consistency

## BigTable/Hbase

## LSM-Tree外存层次的组织

- **Leveling:** 基本设计

- 每层（逻辑上）仅有一个大的排序文件
- 实际上可分成多段，每段对应一个区间（参照B+Tree叶子）



- **Tiering:** compaction总I/O少，但排序文件更多，读性能下降

- 每层有多个排序文件，它们之间是有重叠的
- Compaction归并上层多个排序文件，形成一个新的大文件

compaction



## Bigtable / Hbase小结

- Key包含了row key, column key的结构
- 除了Get/Put, 还提供Scan(范围扫描操作)
  - 按照row key有序存储
- 底层存储采用了分布式文件系统
- Master与Tablet Server
- Tablet Server的内部结构
  - LSM-Tree
  - MemTable, SSTable, 和log

## Cassandra

dynamo和bigtable的结合体

## Cassandra与Dynamo和Bigtable

- Cassandra可以看作是Dynamo和Bigtable的结合体

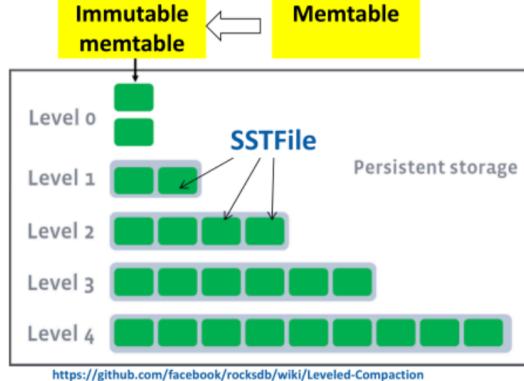
	Dynamo	Bigtable	Cassandra
数据模型中的key	key	row key, column key	row key, column key, <b>super column key</b>
数据存储	Berkeley DB, MySQL	LSM-Tree 内存: MemTable 分布式文件系统: SSTable, Log	LSM-Tree 内存: MemTable <b>本地文件:</b> SSTable, Log
备份冗余	Consistent hashing	分布式文件系统	Consistent hashing

## RocksDB

## RocksDB



- 2012年Facebook基于Google LevelDB开发RocksDB
- C/C++实现，库而非单独系统，有序（类似BigTable），单机



- Memtable是 $C_0$
- $L_0$ 是MemTable直接排序写成的文件(相当于 $C_1$ )，采用tiering
- $L_1..L_k$ 是标准LSM-tree
  - 采用leveling
  - 每个SSTFile对应一个key range

## ZooKeeper

- ZAB广播，2PC的优化
  - leader把txn写入本地log，并propose这个txn，每个follower收到propose后把txn写入本地log并返回ack
  - leader收到至少f个ack后（一共 $2f+1$ 个节点），写commit到log并广播commit，每个follower收到commit之后把commit写入本地log并修改zookeeper树
- ZAB恢复

## ZAB Recovery

- 竞选Leader
  - 每个节点察看自己看到的最大Txn ID
  - 选择Leader为看到max(TxnID)为最大的节点
  - 可以最大限度地保护Client写操作
- TxnID共64位：高32位代表epoch，低32位为in-epoch id
  - 每次选Leader, epoch ++
  - 在一个Leader内部，新的txn增计低32位
  - 于是，每次Recovery后，一定使用了更高的txnid
- 新的Leader
  - 把所有正确执行的Txn都确保正确执行（idempotent，再广播一次）
  - 其它已经提交但是还没有执行的Client操作，都丢弃
  - Client会重试

## 文档数据库

格式

- JSON
- Google Protocol Buffer
- XML

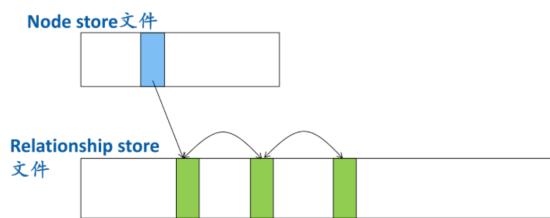
MongoDB

- 要求能看懂mongodb的相关语法

## 图数据库

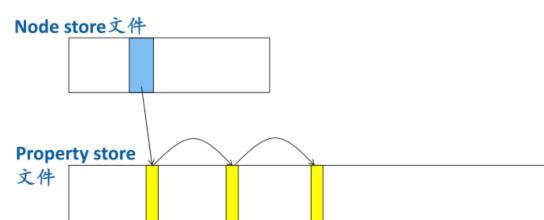
- Neo4j
  - 要求能看懂cypher语法

### Node和Relationship



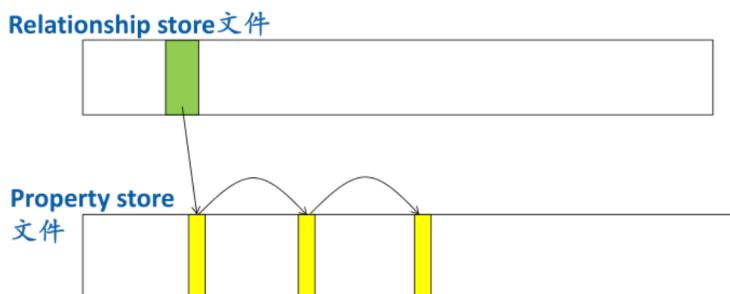
- 同一个node的relationship形成一个双向链表，链接指针为relationship id
- 相应的node中记录链表的第一个relation id

### Node和Property



- 每个顶点node可能有多个property
- 同一个node的所有property形成一个单向链表，指针为next property id

### Relationship和Property

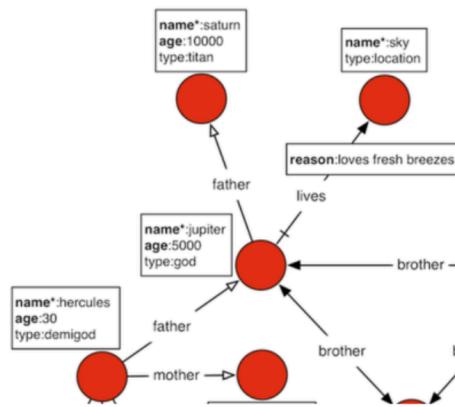


- 每个边relationship可能有多个property
- 同一个relationship的所有property形成一个单向链表，指针为next property id

- JanusGraph
  - 使用Gremlin查询语言

## Gremlin查询举例

```
gremlin> g.V().has('name', 'hercules').out('father').out('father').values('name')
=>saturn
```



- RDF
  - 语言是SPARQL

## SPARQL

- SPARQL（读作sparkle）：RDF的查询语言

- 类似SQL Select语句
- where语句规定一个子图模板
- ?前缀代表变量，注意“.”

```
SELECT ?s
WHERE { ?s Takecourse 体系结构 .
        ?s ISA Student . }
```

Select结果：张飞



## 两种不同的图表示

### • 属性图

- 前面看到的图
- 顶点、边、属性
- Neo4j、JanusGraph/Gremlin支持的图

### • RDF图

- 继续介绍
- 属性都表示成为顶点，所以没有从属于顶点/边的属性
- 等价于属性图

## RDF

### • RDF(Resource Description Framework)

- W3C标准，广泛用于语义网络

### • 每个RDF记录是三元组(subject 主, predicate 谓, object 宾)

- 例如，

(张飞, Takecourse, 体系结构)

(张飞, ISA, Student)

(体系结构, ISA, Course)



- subject和object都是图的顶点

- predicate表达了边的类型

- 多个RDF三元组表达一张图

## SPARQL

### • SPARQL (读作sparkle) : RDF的查询语言

- 类似SQL Select语句
- where语句规定一个子图模板
- ?前缀代表变量，注意“.”

```
SELECT ?s
WHERE { ?s Takecourse 体系结构 .
        ?s ISA Student . }
```

Select结果: 张飞



## 大数据运算系统

### • MapReduce

# 比较

## MapReduce

- Map
- Shuffle
- Reduce
- 选择的功能更加丰富
  - 程序实现的
  - 类似最简单的SQL select
  - 但不直接支持join



## SQL Select

- Selection/projection
- Group by
- Aggregation, Having
- 功能由数据类型和SQL语言标准定义
  - 有UDF: user defined function
  - 但支持得不好

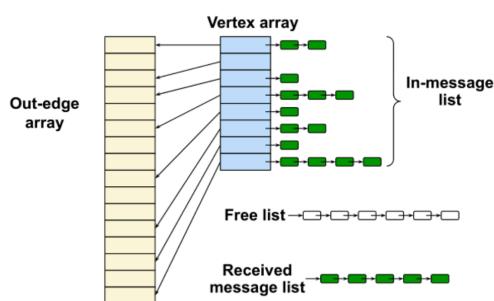
## 同步图计算系统

- PageRank算法

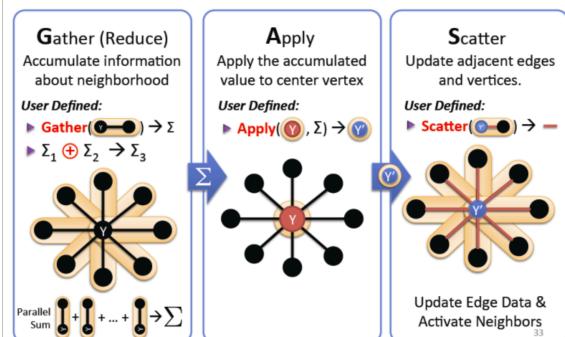
## 图计算系统

- Pregel模型, GraphLite工具
- GAS模型

## GraphLite Worker



## GAS Decomposition



## GraphLite (Pregel模型) 的实现

## GAS抽象

```

double sum= 0.0;
for ( ; !msgs->done (); msgs->next ()) {
    sum += msgs->getValue ();
}
val = 0.15 + 0.85 * sum;
*mutableValue () = val;
int64_t n = getOutEdgeIterator ().size ();
sendMessageToAllNeighbors (val / n);

```

接收消息  
更新状态  
发送消息

```

double sum= 0.0;
for ( ; !msgs->done (); msgs->next ()) {
    sum += msgs->getValue ();
}
val = 0.15 + 0.85 * sum;
*mutableValue () = val;
int64_t n = getOutEdgeIterator ().size ();
sendMessageToAllNeighbors (val / n);

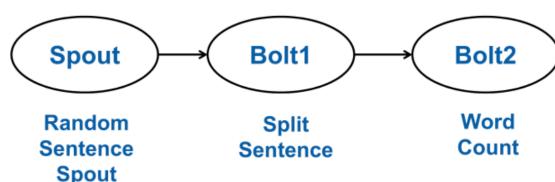
```

改为 Gather  
改为 Apply  
改为 Scatter

## 数据流系统

- Storm

### 举例:一个简单的Topology和程序



- Topology表示数据流上定义的一组运算

### 举例：主程序

```

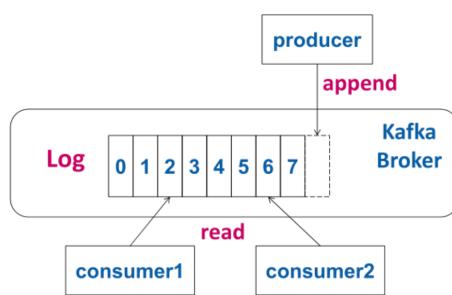
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("spout", new RandomSentenceSpout(), 5);
    builder.setBolt("split", new SplitSentence(), 8)
        .shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("split", new Fields("word"));
    Config conf = new Config();
    conf.setMaxTaskParallelism(3);
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("word-count", conf, builder.createTopology());
    Thread.sleep(10000);
    cluster.shutdown();
}

```

建立 Topology  
运行

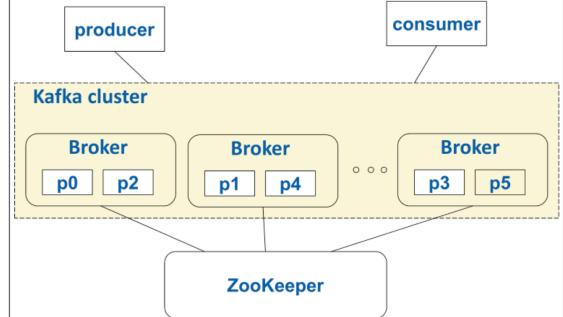
- Kafka

### Kafka基本模型



消息保留的时间：给定时间/磁盘空间设置，超出就删除

### Kafka系统结构



## 内存数据库

- MonetDB

- SPA HAMA

对比维度	SAP HANA	MonetDB
研发背景	SAP公司研发，是企业级解决方案核心部分，集成于各类企业应用	荷兰CWI研究所研制，开源，兼顾学术研究和商业应用
存储方式	内存列式存储，也支持行式存储；分热存储（存放常用数据）、冷存储（存放不常用数据），热存储含主存储和增量存储	内存列式存储，数据存于类似数组结构，采用独特二级存储模型和ColumnBM列式数据组织方式
查询处理	支持SQL，能同时处理OLTP和OLAP业务，避免传统BI的ETL过程，实现实时分析；体系架构为share - nothing，资源配置灵活	支持几乎所有SQL，SQL被翻译为中间语言MAL运算，MAL运算针对BAT (Binary Association Table)，存储一列数据且数据映射到内存) 数据；有向量化查询处理和“Vectorized in - cache execution model”执行模型，优化cache和磁盘读写性能
功能特性	支持ACID事务语义，几乎完整整合R功能，内置大量与SAP业务相关功能	实现成熟的列式存储和向量计算能力，支持多种数据类型（结构化、半结构化、非结构化）
应用场景	聚焦大型企业复杂业务流程管理与数据分析，如财务、供应链、制造等领域	应用场景广泛，从学术研究到中小企业数据分析均可适用
市场定位	企业级商业产品，需搭配特定硬件，软件价格较高	开源数据库，对成本敏感用户和开发者有吸引力

## 内存键值系统

- Memcached
- Redis

# 内存key-value系统

## • Memcached

- 用户：Facebook, twitter, flickr, youtube, ...
- 本质上是单机内存键值对系统
- 数据在内存中以hash table的形式存储
- 支持最基础的<key, value>数据模型
- 通常被用于前端的cache
- 可以使用多个memcached建立一个分布式系统

## • Redis：与memcached相比

- 分布式内存键值对系统
- 提供更加丰富的类型，例如hashes, lists, sets 和sorted sets
- 支持副本和集群

# 内存大数据运算系统

- spark
  - 了解RDD的概念
  - 能看懂scala语法
  - 看懂spark例子

## Transformation (1)

Transformation	涵义
<code>map(func)</code>	func是某种转换：一个输入元素→一个结果元素。对RDD的每个元素调用func生成结果RDD。
<code>filter(func)</code>	func是一个过滤条件：一个输入元素→True/False。对RDD的每个元素调用func，丢弃为False的元素，形成结果RDD。
<code>flatMap(func)</code>	func: 一个输入元素→一组元素（0个或多个）对RDD的每个元素调用func生成结果RDD。 （类似MapReduce中的Map）
<code>union(otherDataset)</code>	集合并
<code>intersection(otherDataset)</code>	集合交
<code>distinct()</code>	去重

## Transformation (2)

Transformation	涵义
<code>groupByKey()</code>	输入RDD(K, V)，返回结果RDD(K, Iterable<V>)。相同K的V放入了一个列表。
<code>reduceByKey(func)</code>	func把两个同类的值归并为一个值：(V, V)→V。输入RDD(K, V)，返回结果RDD(K, V)，其中对于相同K的所有V都调用了func，归并为一个V。 （类似MapReduce中的Reduce）
<code>sortByKey([asc])</code>	对输入RDD(K, V)按照K进行排序，可选参数asc为True则从小到大，False则从大到小排序。
<code>join(otherDataset)</code>	输入RDD1(K, V)和RDD2(K, W)，结果RDD为(K, (V, W))，相同K进行等值连接。
<code>cogroup(otherDataset)</code>	输入RDD1(K, V)和RDD2(K, W)，结果RDD为(K, (Iterable<V>, Iterable<W>))。把两个数据集中相同K的值放入两个表。
<code>repartition(numPartitions)</code>	Reshuffle RDD数据产生numPartitions个划分。

Action: 返回结果给main driver

Action	涵义
reduce(func)	func把两个同类元素归并为一个元素 (T,T)->T 在输入RDD上调用func，最后形成一个结果元素。
collect()	把输入RDD转换为一个数组，返回给main driver
count()	返回RDD中元素个数
first()	返回RDD中第一个元素 (基本等同于take(1))。
take(n)	把RDD中前n个元素作为数组返回
saveAsTextFile(path)	把RDD写成为一个文本文件
saveAsSequenceFile(path)	把RDD写成一个Hadoop SequenceFile

## RDD的两类运算

### • Transformation

- 输入是RDD
- 输出是RDD

### • Action

- 输入是RDD
- 输出是可以返回给driver程序的结果
- 输出不是分布式的数据集

## 基础数据结构： RDD

### • Resilient Distributed Data sets

- 一个数据集
- 只读，整个数据集创建后不能修改
- 通常进行整个数据集的运算

### • 优点

- 并发控制被简化了
- 可以记录lineage（数据集上的运算序列），可以重新计算
  - 并不需要把RDD存储在stable storage上

## Java RDD的两种类型

- Class JavaRDD<T>                   元素类型为T的RDD
- Class JavaPairRDD<K,V>           元素包含一个K和一个V

### • 互相转换

- JavaRDD<Tuple2<K,V>> → JavaPairRDD<K,V>
  - JavaPairRDD.fromJavaRDD(rdd)
- JavaPairRDD<K,V> → JavaRDD<K>
  - JavaPairRDD.keys()
- JavaPairRDD<K,V> → JavaRDD<V>
  - JavaPairRDD.values()

# 大数据分析

## DR, 数据降维

### SVD, 奇异值分解

#### 1. 定义与公式

对于任意实矩阵  $A \in \mathbb{R}^{m \times n}$ , 存在唯一分解:  $A = U\Sigma V^\top$

其中:

- $U \in \mathbb{R}^{m \times m}$  和  $V \in \mathbb{R}^{n \times n}$  为正交矩阵 ( $U^\top U = I_m$ ,  $V^\top V = I_n$ )。
- $\Sigma \in \mathbb{R}^{m \times n}$  为对角矩阵, 对角线元素为非负奇异值  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \geq 0$  ( $r = \text{rank}(A)$ ) , 其余元素为 0。

#### 2. 核心性质

- 奇异值与特征值关联:

    奇异值

$\sigma_i$  是  $A^\top A$  或  $AA^\top$  的特征值的平方根，即  $\sigma_i = \sqrt{\lambda_i(A^\top A)} = \sqrt{\lambda_i(AA^\top)}$ 。

- **低秩近似：**

保留前

$k$  个最大奇异值，得到秩为  $k$  的近似矩阵  $A_k = U_k \Sigma_k V_k^\top$ ，其在 Frobenius 范数下是原矩阵的最佳低秩近似。

### 3. 缺点

- **计算复杂度高：**精确计算 SVD 的时间复杂度为  $O(mn^2)$  或  $O(m^2n)$ ，对大规模矩阵不友好（文档提到 " $O(nm^2)$  or  $O(n^2m)$ "）。
- **缺乏稀疏性：**奇异向量通常是稠密的，难以解释物理意义（文档指出 "Singular vectors are dense!"）。
- **依赖全局信息：**分解结果受矩阵所有元素影响，对局部特征不敏感。
- **非适应性：**无法处理动态更新的数据，需重新计算全矩阵分解。

## 矩阵范数及其应用

### 1. 谱范数 (2 - 范数)

- **定义：**矩阵  $A$  的最大奇异值，即  $\|A\|_2 = \sigma_{\max}$ 。
- **性质：**
  - 诱导范数，满足  $\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2$ 。
  - 对正交变换不变，即  $\|UAV\|_2 = \|A\|_2$  ( $U, V$  为正交矩阵)。

- **示例：**

文档题目中矩阵

$A = \begin{pmatrix} 2 & 3 & i \\ -1 & 2 & -i \end{pmatrix}$  的谱范数为最大奇异值  $\sqrt{15}$  (计算见下文)。

### 2. Frobenius 范数 (F - 范数)

- **定义：**矩阵元素平方和的平方根，即  $\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2} = \sqrt{\sum_{i=1}^r \sigma_i^2}$
- **性质：**
  - 酉不变性，即  $\|UAV\|_F = \|A\|_F$ 。
  - 可通过奇异值直接计算（文档隐含公式）。

- **示例：**

题目中矩阵的 F - 范数为

$\sqrt{2^2 + 3^2 + 1^2 + (-1)^2 + 2^2 + 1^2} = \sqrt{20} = 2\sqrt{5}$ , 与奇异值平方和结果一致。

### 3. 核范数（迹范数）

- **定义：**矩阵奇异值之和，即  $\|A\|_N = \sum_{i=1}^r \sigma_i$ 。

- **性质：**

- 衡量矩阵的低秩程度，常用于矩阵补全问题。

- **示例：**

题目中矩阵的核范数为

$\sqrt{15} + \sqrt{5}$ , 反映矩阵的“能量”总和。

### 4. 向量范数与矩阵范数的关联

- **1-范数：**向量  $x$  的 1-范数为  $\|x\|_1 = \sum_i |x_i|$ , 矩阵 1-范数为列和最大值。
- **无穷范数：**向量  $x$  的无穷范数为  $\|x\|_\infty = \max_i |x_i|$ , 矩阵无穷范数为行和最大值。

## CUR

将矩阵  $A \in \mathbb{R}^{m \times n}$  近似分解为  $A \approx CUR$ , 其中:

- $C \in \mathbb{R}^{m \times k}$ : 从 A 中选取的 k 列构成的矩阵 (列子集)。
- $R \in \mathbb{R}^{k \times n}$ : 从 A 中选取的 k 行构成的矩阵 (行子集)。
- $U \in \mathbb{R}^{k \times k}$ : 中间校准矩阵, 通过伪逆计算 ( $U = (C^\top A R)^\dagger$ )。

## LSH, 局部敏感哈希

### 1. 数据预处理

- 文档等集合数据：通过 k-shingling 生成集合，再用 Min-Hashing 生成签名向量。
- 向量数据 (如图像特征)：使用随机投影生成哈希签名。

### 2. 分带与哈希分桶

- 将签名矩阵划分为 b 个带，每带 r 行 ( $b \times r = \text{总哈希函数数}$ )。
- 对每带内的子签名用哈希函数分桶，相同桶内的数据作为候选相似对。

## 流数据处理

## 采样

### 按固定比例采样

采用**固定比例采样** (1/10)：对每个查询生成一个  $[0, 9]$  的随机整数，若结果为 0 则保留该查询，否则丢弃。

- 简单随机采样对**单次出现的唯一查询和多次出现的重复查询赋予相同的采样概率**，但重复查询因出现次数更多，其被采样的总概率更高，导致样本中重复查询的“存在感”被稀释。
- 具体而言，重复查询需两次均被采样才会被识别为“重复”，但单次被采样时会被误判为“唯一查询”，从而低估重复比例（段落至）。

### 维持固定大小样本

•

**目标：**始终维持一个大小为  $s$  的随机样本，确保每个元素被包含的概率相等。

•

**算法：**

◦

**蓄水池采样 (Reservoir Sampling) :**

1. 存储前

$s$  个元素作为初始样本。

2. 对于第

$n$  个元素 ( $n > s$ )，以概率  $\frac{s}{n}$  替换样本中的随机一个元素。

◦

**证明：**通过数学归纳法可证，每个元素被保留的概率为  $\frac{s}{n}$  ( $n$  为已处理元素总数)。

■  
**基例：**  $n = s$  时，每个元素概率为 1。

■  
**归纳步骤：**假设  $n$  时概率为  $\frac{s}{n}$ ，则  $n + 1$  时，旧元素保留概率为  $\frac{s}{n} \times \frac{n}{n+1} = \frac{s}{n+1}$ ，新元素被选中概率为  $\frac{1}{n+1}$ ，满足等概率性。

### 滑动窗口查询

#### DGIM

##### 1. 算法目标

解决**数据流滑动窗口内的近似计数问题**，即在长度为  $N$  的滑动窗口中，快速估算最近  $k$  ( $k \leq N$ ) 个元素内 1 的个数，允许误差不超过 50%。适用于无法存储全量数据的实时场景，如网络流量监控、日志分析等。

##### 2. 核心思想

- **桶 (Bucket) 结构:**
  - 每个桶存储两个信息:
    - **结束时间戳** (模  $N$ , 占  $O(\log N)$  位): 表示桶内最后一个 1 的位置。
    - **1 的个数** (必须为 2 的幂, 如  $2^r$ , 占  $O(\log \log N)$  位)。
  - **桶的性质:**
    - 同大小的桶最多有 2 个 (如最多 2 个大小为 4 的桶)。
    - 按大小降序排列, 时间戳不重叠, 过期桶 (时间戳超过当前时间 -  $N$ ) 自动淘汰。
- **合并策略:**  
当出现 3 个同大小的桶时, 合并最早的 2 个为一个更大的桶 (如 3 个大小为 1 的桶合并为 1 个大小为 2 的桶), 确保桶数量维持在  $O(\log N)$  级别。

### 3. 关键步骤

1. **数据更新:**
  - 收到新位为 1 时, 创建大小为 1 的新桶, 触发合并规则 (若同大小桶数  $\geq 3$ )。
  - 收到新位为 0 时, 仅检查并删除过期桶。

2. **查询处理:**

- 累加所有完全包含在查询窗口内的桶的大小。
- 对最后一个部分包含的桶, 取其大小的一半作为近似值 (误差来源)。
- 误差上限为 50%, 源于对部分桶的“半量假设”。

### 4. 空间与时间复杂度

- **空间复杂度:** 每个流存储  $O(\log^2 N)$  位 (桶数为  $O(\log N)$ , 每个桶占  $O(\log N)$  位), 远低于原始数据规模。
- **时间复杂度:** 单次更新或查询均为  $O(\log N)$ , 支持实时处理。

### 5. 优势与局限性

- **优势:**
  - 空间高效, 适用于大规模数据流。
  - 误差可控 ( $\leq 50\%$ ), 满足近似分析需求。
  - 单遍处理, 实时性强。

- 局限性：

- 仅适用于二进制数据流（0/1），多值数据需扩展。
- 固定 50% 误差上限，高精度场景需改进（如增加桶数降低误差）。

## 布隆过滤器

从数据流中选择具有属性  $x$  的元素

### 1. 关键组件

- 位数组（Bit Array）：长度为  $n$  的二进制数组，初始全为 0。
- 哈希函数组： $k$  个独立的哈希函数  $\{h_1, h_2, \dots, h_k\}$ ，每个函数将元素映射到  $[0, n - 1]$  的范围内。

### 2. 插入元素

对于集合中的元素  $s$ ，执行以下操作：

1. 对每个哈希函数  $h_i$ ，计算  $h_i(s)$ ，得到  $k$  个哈希值。
2. 将位数组中对应的  $k$  个位置设置为 1（即使已为 1，仍保持为 1）。

### 3. 查询元素

判断元素  $x$  是否属于集合：

1. 对每个哈希函数  $h_i$ ，计算  $h_i(x)$ ，检查对应的  $k$  个位置是否全为 1。
2. 若是：则认为  $x$  可能属于集合（存在假阳性）。
3. 若否：则  $x$  必定不属于集合（无假阴性）。

### 1. 误判概率公式

假设集合大小为  $m$ ，位数组长度为  $n$ ，哈希函数个数为  $k$ ，则误判概率为：

$$P_{\text{false positive}} = (1 - e^{-km/n})^k$$

推导过程如下：

- 单个位在插入  $m$  个元素后仍为 0 的概率： $(1 - \frac{1}{n})^{km} \approx e^{-km/n}$ 。
- 所有  $k$  个哈希位均为 1 的概率（即误判概率）： $(1 - e^{-km/n})^k$ 。

### 2. 最优参数选择

- 哈希函数个数：最优  $k = \frac{n}{m} \ln 2$ ，此时误判概率最小。
- 位数组长度：若已知集合大小  $m$  和目标误判率  $p$ ，则  $n = -\frac{m \ln p}{(\ln 2)^2}$ 。

## 示例：

- 若  $m = 1$  亿,  $n = 8$  亿 (1GB 内存),  $k = 6$  (最优值), 则误判概率约为 1.4% (低于简单哈希函数的 12.5%)。

## FM算法

数据流最后  $k$  个元素中的不同元素数量

### 1. 初始化

- 初始化

$R = 0$  (记录最大尾部零位数)。

### 2. 处理每个元素

对于数据流中的元素  $a$ :

#### 1. 计算哈希值

$h(a)$ , 转换为二进制字符串。

#### 2. 计算

$r(a)$ : 从哈希值末尾开始统计连续零位的个数。

#### 3. 更新

$R = \max(R, r(a))$ 。

### 3. 估计基数

最终估计的不同元素数量为  $\hat{m} = 2^R$ 。

## AMS算法

估计最后  $k$  个元素的平均值/标准差

- 

**初始化**: 选择一些随机时间点  $t$  ( $t < n$ ,  $n$  为数据流长度, 实际应用中数据流长度可能未知,  $n$  为当前已处理的元素数) 作为起始点, 设定用于估计的变量  $X$ 。

- 

**计数维护**: 从选定的时间  $t$  开始, 对出现的元素进行计数, 记录在相应变量  $X$  的  $X.val$  中。

- 

**矩估计**: 对于 2 阶矩, 估计公式为  $S = n(2c - 1)$ , 其中  $n$  是数据流长度,  $c$  是  $X.val$  (元素的计数)。最终估计值是多个这样的估计值的平均值, 即  $S = \frac{1}{k} \sum_j f(X_j)$ ,  $k$  为变量  $X$  的个数。假设数据流为  $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$ , 长度  $n = 15$ , 随机选取 3 个变量  $X1, X2, X3$ , 分别从第 3、8、13 个位置开始。若  $X1.el = c$ ,  $X1.val$  最终为 3;  $X2.el = d$ ,  $X2.val$  最终为 2;  $X3.el = a$ ,  $X3.val$  最终为 2。则对应的估计值分别为  $15 * (2 * 3 - 1) = 75$ ,  $15 * (2 * 2 - 1) = 45$ ,  $15 * (2 * 2 - 1) = 45$ , 平均值为  $(75 + 45 + 45)/3 = 55$ , 接近

真实的 2 阶矩值 59 (a 出现 5 次, b 出现 4 次, c 出现 3 次, d 出现 3 次,  $5^2 + 4^2 + 3^2 + 3^2 = 59$ )。

## 蒙特卡洛

蒙特卡洛方法是一类利用重复随机抽样来获取复杂计算问题近似解的算法。其应用包括模拟、积分、优化和抽样。

### 式子含义解释

1.

$$\mathbb{E}(X) = \mathbb{E}_Y(\mathbb{E}(X|Y))$$

。从含义上看，等式左边

$\mathbb{E}(X|Y)$  是随机变量  $X$  的总体期望。等式右边  $\mathbb{E}(X|Y)$  是在给定  $Y$  条件下  $X$  的条件期望，它是关于  $Y$  的函数，再对这个函数关于  $Y$  求期望  $\mathbb{E}_Y(\mathbb{E}(X|Y))$ 。

。举个例子，假设

$X$  表示一个城市居民的收入， $Y$  表示居民的职业。 $\mathbb{E}(X|Y)$  就是给定职业  $Y$  时居民收入的平均水平（比如给定职业是教师时，教师群体的平均收入）。而  $\mathbb{E}_Y(\mathbb{E}(X|Y))$  就是先求出每个职业对应的平均收入，再根据不同职业在城市中的分布情况（即  $Y$  的概率分布）对这些平均收入进行加权平均，最终得到的结果就是整个城市居民的平均收入，也就是  $\mathbb{E}(X|Y)$ 。

2.

$$\mathbb{E}(g(X)) = \mathbb{E}_Y(\mathbb{E}(g(X)|Y)) \quad (\text{对于任意函数 } g)$$

。这里

$g(X)$  是随机变量  $X$  的函数。等式左边  $\mathbb{E}(g(X))$  是  $g(X)$  的总体期望。等式右边  $\mathbb{E}_Y(\mathbb{E}(g(X)|Y))$  是在给定  $Y$  条件下  $g(X)$  的条件期望，是关于  $Y$  的函数，再对其关于  $Y$  求期望  $\mathbb{E}_Y(\mathbb{E}(g(X)|Y))$ 。

。例如，设

$X$  还是居民收入， $g(x) = x^2$ （表示收入的平方）， $Y$  为职业。 $\mathbb{E}(g(X)|Y)$  就是给定职业  $Y$  时居民收入平方的平均水平， $\mathbb{E}_Y(\mathbb{E}(g(X)|Y))$  是根据不同职业的分布对这些收入平方的平均水平进行加权平均，得到的就是整个城市居民收入平方的平均水平，即  $\mathbb{E}(g(X))$ 。这两个式子都体现了全期望公式的思想，通过条件期望来计算总体期望。

### 逆变换方法

通过生成一组均匀分布的随机数，然后根据指定分布的分布函数  $F(x)$  的反函数，获取服从指定分布的随机数

简而言之，将均匀分布的随机数转换成其他分布的随机数

逆变换方法是许多高级采样技术的基石

## 拒绝采样

想要采样概率密度为 $q(x)$ 的分布

选取一个方便采样的分布，比如均匀分布 $p(x)$ ，进行采样（也就是生成一组随机数），对采样的结果乘以一个缩放因子 $M$

对于每个采样结果，以概率  $\frac{q(x)}{Mp(x)}$  接受（生成一个随机数，小于等于则接受，大于则拒绝）

所有被接受的数，服从概率密度为 $q(x)$ 的分布

缩放因子 $M$ 可以理解为，每生成 $M$ 个服从 $p(x)$ 分布的样本，就有一个满足 $q(x)$ 分布

## 蒙特卡洛积分

解决高纬或解析不可积的积分计算问题

根据大数定理，把原积分转换为期望估计问题

蒙特卡洛积分主要有均匀采样法和重要性采样法两种

## 均匀采样法

假设积分区间为  $[a, b]$ ，令  $p(x) = \frac{1}{b-a}$ （均匀分布），则： $I = \int_a^b h(x)dx = (b-a) \int_a^b h(x)\frac{1}{b-a}dx = (b-a)\mathbb{E}[h(X)]$

步骤：

1. 生成  $n$  个均匀样本  $x_i \sim \text{Unif}(a, b)$ ；
2. 计算样本均值  $\bar{h} = \frac{1}{n} \sum h(x_i)$ ；
3. 积分估计值为  $\hat{I} = (b-a)\bar{h}$

## 重要性采样法

当  $h(x)$  在大部分区域值接近零，仅在小区域贡献显著时，可选择与  $h(x)$  形态匹配的**重要性分布**  $g(x)$ ，减少无效采样： $I = \int h(x)dx = \int h(x)\frac{g(x)}{g(x)}dx = \int \frac{h(x)}{g(x)}g(x)dx = \mathbb{E}_g\left[\frac{h(x)}{g(x)}\right]$

步骤：

1. 生成  $x_i \sim g(x)$ ；
2. 计算加权均值  $\hat{I} = \frac{1}{n} \sum \frac{h(x_i)}{g(x_i)}$

$$\text{均匀采样方差: } \text{Var}(\hat{I}_{\text{uni}}) \propto \frac{V^2}{N} \cdot \left( \frac{1}{V} \int f^2 dx - \left( \frac{I}{V} \right)^2 \right)$$

$$\text{重要性采样方差: } \text{Var}(\hat{I}_{\text{imp}}) \propto \frac{1}{N} \cdot \left( \int \frac{f^2}{p} dx - I^2 \right)$$

拒绝采样用于**生成复杂分布样本**，而蒙特卡洛积分用于**估计复杂函数积分**。两者均基于随机采样，共同构成蒙特卡洛方法的核心工具链

### 马尔可夫链蒙特卡洛 (MCMC)

#### MH采样

**1. 核心类比：**随机漫步中的“探索 - 接受”机制可以将 MH 算法想象成一个在目标分布“地形”上的随机漫步过程：

- **当前位置：**表示马尔可夫链的当前状态  $x$ ，对应目标分布  $p(x)$  中的一个点。
- **提议移动：**从当前位置随机“迈出一步”(生成候选状态  $x^*$ )，这一步的大小和方向由**提议分布**（如高斯分布）决定。
- 

#### 判断是否接受移动：

如果新位置  $x^*$  的“海拔”(即  $p(x^*)$ ) 比当前位置高，**一定接受移动**，因为高海拔区域对应目标分布的高概率区域。

如果新位置海拔较低，**以一定概率接受移动**，概率与两地海拔的比值成正比，避免链被困在局部高概率区域，确保能探索整个分布。

#### 2. 拒绝采样的“动态平衡”升级

- **与拒绝采样的联系：**MH 算法类似拒绝采样，但不再完全丢弃拒绝的样本，而是通过**重复当前状态**维持马尔可夫链的连续性。
- **关键改进：**拒绝采样需要独立生成样本，而 MH 算法利用马尔可夫链的**历史状态**，使样本间存在相关性，从而在高维空间中更高效地“游走”。

#### 3. 详细平衡条件的直观意义

-

**微观可逆性**: 在长时间运行后, 链在任意两个状态  $x$  和  $x^*$  之间的来回移动次数达到平衡, 即从  $x$  到  $x^*$  的次数乘以  $p(x)$ , 等于从  $x^*$  到  $x$  的次数乘以  $p(x^*)$ 。

•

**宏观结果**: 这种平衡确保链的分布最终收敛到目标分布  $p(x)$ , 就像一滴墨水在水中扩散后达到均匀分布。

### Gibbs采样

1.

**初始化**: 设定初始状态  $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \dots, x_d^{(0)})$ 。

2.

**迭代更新**: 按顺序或随机顺序更新每个变量  $x_i$ , 每次从条件分布  $p(x_i|\mathbf{x}_{-i})$  中采样 ( $\mathbf{x}_{-i}$  表示除  $x_i$  外的其他变量)。

◦ 例如, 更新

$x_i$  时, 固定  $\mathbf{x}_{-i}$  的当前值, 采样得到  $x_i^{(t)} \sim p(x_i|\mathbf{x}_{-i}^{(t-1)})$ 。

3.

**样本收集**: 经过 burn-in 期后, 保留后续样本作为目标分布的近似。