04 B+树结构的物理实现

Physical Implementation

南京大学软件学院

数据文件格式（File Formats）面对的挑战

- 对开发者来说，内存的访问几乎是透明的
  - 虚拟内存机制，不需要手动管理偏移量
- 而磁盘的访问，是通过系统调用实现的
  - 需要指定目标文件内的偏移量，把数据从磁盘上的形式解析成主存形式
  - 类似于非托管内存模型的语言中构建数据结构（unmanaged memory model）
  - 本质上，设计一个数据结构，自己处理垃圾收集和碎片问题
  - 二进制格式，布局（layout），不能使用malloc，free，而不得不用read和write

理解这个差异，将B树的基本语义实现在磁盘上

二进制编码（复习）

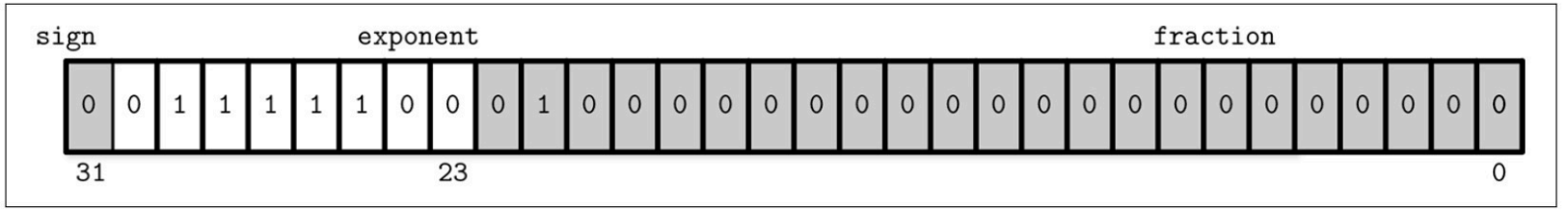
- 原始类型的实现（integer、date、string……）
  - 固定大小，编解码使用相同字节序（endianness）

第一步：用二进制形式表示键和值

十六进制32-bit integer 0xAABBCCDD，大端，小端两种不同的字节序编码
通常得了解目标平台的字节序（byte-order或endianness），再做处理

二进制编码（复习）

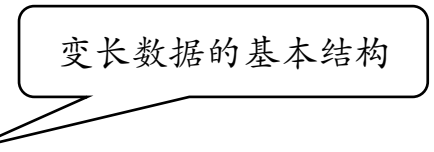
- 1 byte = 8 bits
  - 短整型（short）2byte（16bits），整型（int）4byte（32bits），长整型（long）8byte（64bits）
  - 单精度浮点数（float）双精度浮点数（double）由符号sign、小数fraction、指数exponent构成
  - IEEE二进制浮点数算术标准（IEEE 754）
  - 例子：32位单精度浮点数，0.15652的二进制如下



低位23位表示小数值，随后8位表示指数值，最后的一位表示符号位（是否为负数）
浮点数是通过分数计算出来的，所以只是近似值，具体不展开，我们课程只介绍存储的表现形式

二进制编码（复习）

- 字符串和变长数据
  - 所有原始数据类型都有固定的大小，构造更复杂的类似C的struct
  - 原始值组合到结构体，并使用固定长度的数组或指针指向其他区域
  - 字符串可以序列化成一个表示长度的数值字段size + size个字节
  - UCSD字符串或Pascal字符串




```
String
{
    size    uint_16
    data    byte[size]
}
```

二进制编码

- 按位打包的数据：布尔、枚举、标志
  - 布尔值——单个字节（也可以true/false编码1/0），一般8个布尔值和在一起，一个占一位
  - 枚举值——设计成证书，用于二进制格式和通信协议，应用于重复多，基数少的值
  - 标志——打包的布尔值和枚举的组合，表示多个非互斥的布尔值参数
    - 页是否包含值单元、值是定长还是变长、是否存在于当前节点相关联的溢出页

枚举值表示B树的节点类型

```
enum NodeType {
   ROOT,     // 0x00h
   INTERNAL, // 0x01h
   LEAF      // 0x02h
};
```

```
int IS_LEAF_MASK         = 0x01h; // bit #1
int VARIABLE_SIZE_VALUES = 0x02h; // bit #2
int HAS_OVERFLOW_PAGES   = 0x04h; // bit #3
```

每个比特位代表一个标志值，只能将2的幂用作掩码，因为二进制下2的幂总是只有一位为1，例如：23==8==1000b、24==16==0001 000b 等）

查询所用数据库及服务器的编码配置

```
# 查询数据库及服务器的编码配置
show variables like '%character%'

# 查询数据表的编码格式
show create table <表名>

# 查询各个字段编码格式
show full columns from <表名>

```

```
+--------------------------+--------------------------------------------------------+
| Variable_name            | Value                                                  |
+--------------------------+--------------------------------------------------------+
| character_set_client     | utf8mb4                                                |
| character_set_connection | utf8mb4                                                |
| character_set_database   | utf8mb4                                                |
| character_set_filesystem | binary                                                 |
| character_set_results    | utf8mb4                                                |
| character_set_server     | utf8mb4                                                |
| character_set_system     | utf8                                                   |
| character_sets_dir       | /usr/local/Cellar/mysql/8.0.23/share/mysql/charsets/ |
+--------------------------+--------------------------------------------------------+
```

```
CREATE TABLE `test` (
  `id` int unsigned NOT NULL AUTO_INCREMENT COMMENT '自增id',
  `device_code` varchar(128) DEFAULT NULL COMMENT '设备code',
  `feature_text` text COMMENT '特征',
  `feature_varchar` varchar(2048) DEFAULT NULL COMMENT '特征',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='算法特征表'
```

```
+-----------------+---------------+-----------------+------+-----+---------+----------------+-------
| Field           | Type          | Collation       | Null | Key | Default | Extra          | Privi
+-----------------+---------------+-----------------+------+-----+---------+----------------+-------
| id              | int unsigned  | NULL            | NO   | PRI | NULL    | auto_increment | selec
| device_code     | varchar(128)  | utf8_general_ci | YES  |     | NULL    |                | selec
| feature_text    | text          | utf8_general_ci | YES  |     | NULL    |                | selec
| feature_varchar | varchar(2048) | utf8_general_ci | YES  |     | NULL    |                | selec
+-----------------+---------------+-----------------+------+-----+---------+----------------+-------
```

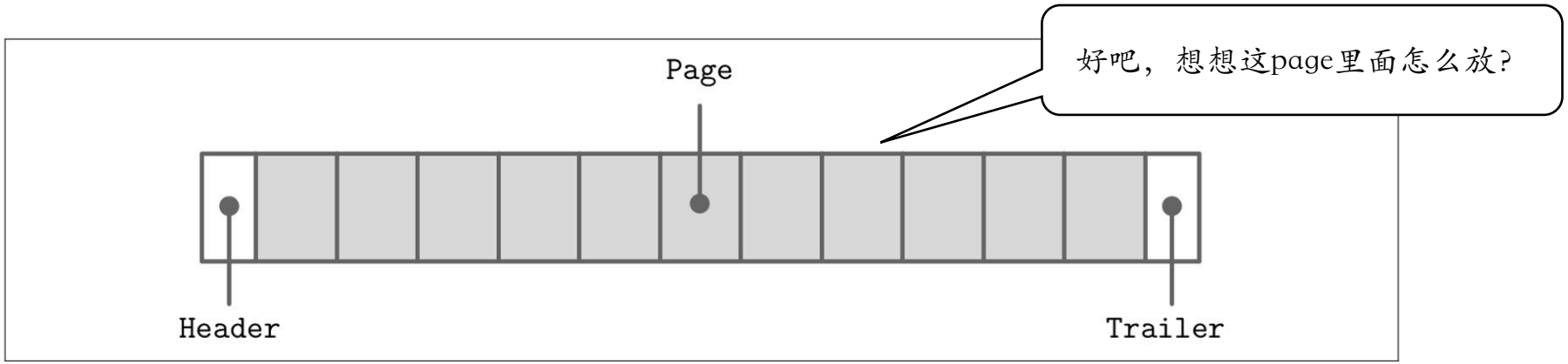
```
# 修改表的默认配置
ALTER TABLE `table` DEFAULT CHARACTER SET utf8mb4;

# 修改某一字段的编码格式
ALTER TABLE `test` CHANGE `device_code` `device_code` VARCHAR(36) CHARACTER SET utf8 NOT NULL

修改表的全部字段
alter table `recall_sku` convert to character set utf8mb4 COLLATE utf8mb4_unicode_ci
```

数据库文件物理组织形式通用原理

- 通常，设计文件格式，第一步是确定寻址方式

数据存储结构一般分成两类——原地更新、仅追加（append-only）通常都是按照页进行组织

  - 文件拆成相同大小的页（page），单个块（block）或者连续块（multiple block）组成
  - 相同大小的page，目的是简化读取和写入访问，按页写入，从内存写到磁盘



数据库文件物理组织形式通用原理

- 数据库的表结构（schema）一般是固定的，指定了字段的数量、顺序和类型
  - 固定的好处是减少存储数量，使用位置标识符，而不需要记录加上字段名

```
Fixed-size fields:
| (4 bytes) employee_id                 |
| (4 bytes) tax_number                  |
| (3 bytes) date                        |
| (1 byte)  gender                      |
| (2 bytes) first_name_length           |
| (2 bytes) last_name_length            |

Variable-size fields:
| (first_name_length bytes) first_name |
| (last_name_length bytes) last_name   |
```

整个数据表文件，需要构建一个“查找表”吗？

如果要读name，

1）先读出last_name_length，计算出变长字段的偏移量，再去读可变长度的内容；

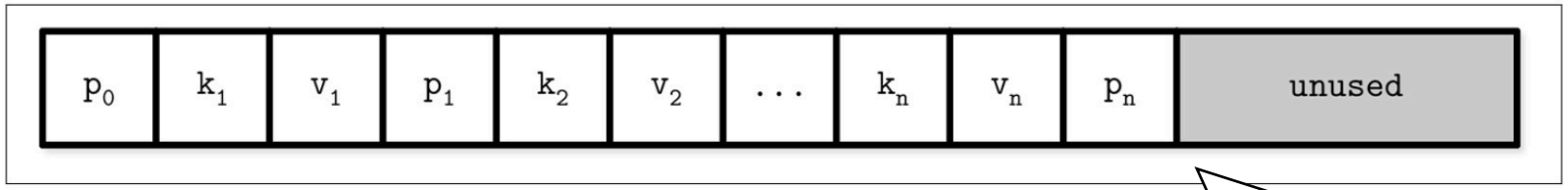
2）避免计算，将偏移量和长度都编入定长区域，这样可以独立定位任何一个变长字段

3）复杂的结构，比如聚簇，在结构体前再增加层次结构

所以，数据文件往往在头部、尾部、或单独文件包含“查找表”，记录起始偏移量

定长：页的结构（Page Structure）

- Page的大小一般是 4-16k
  - 例子：B树索引的叶节点（包含key和数据记录对）和非叶节点（key和指向其他节点的指针），[BAYER72]设计了简单的，用于定长数据记录的page组织方式，每个page都是一连串的三元组，k：key；v：关联值；p：指针

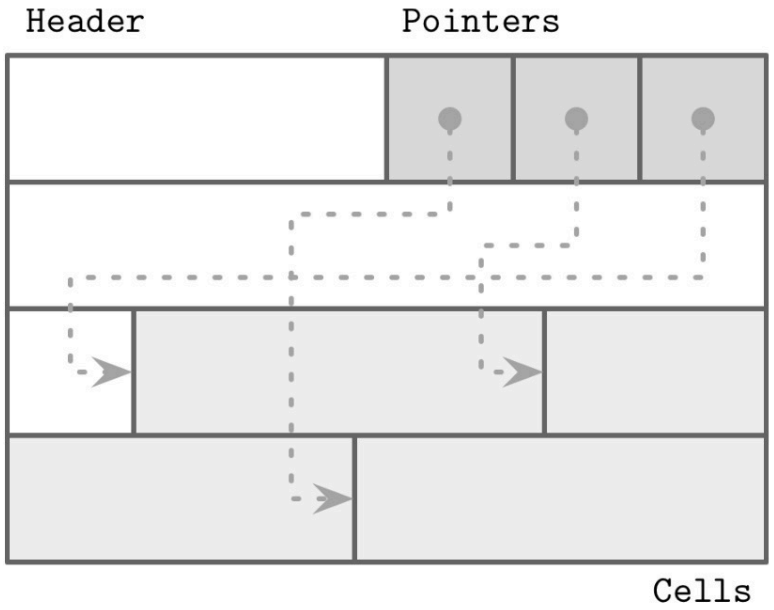


缺点：1）插入……2）变长……

好吧，这么简单美好的设计的问题是什么？

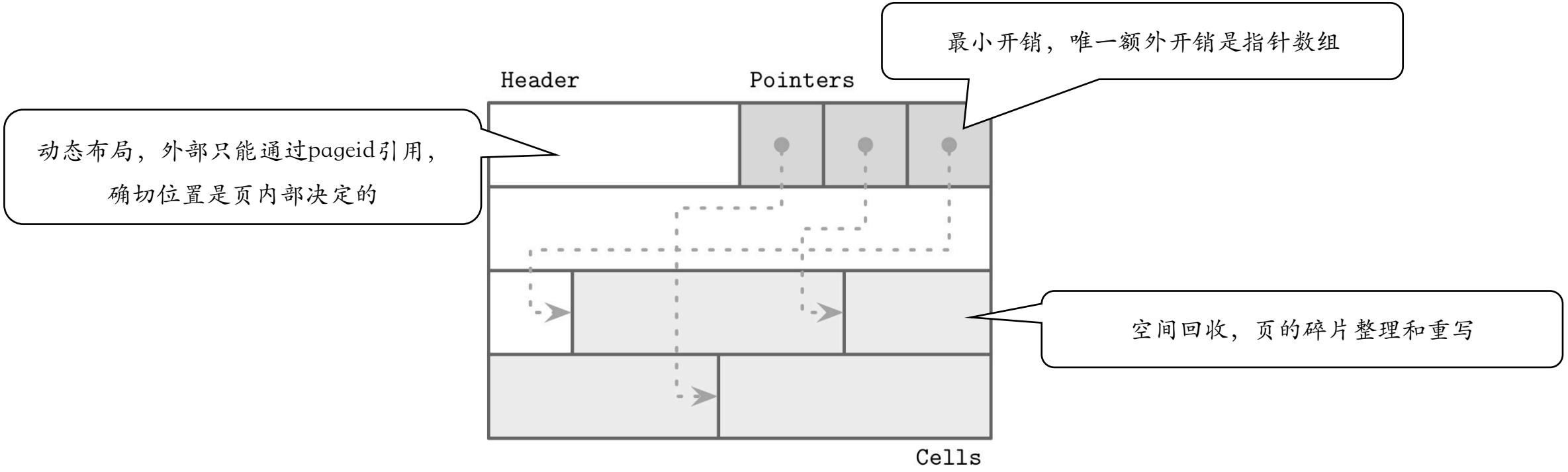
变长：分槽页（slotted page）

- 变长记录最大的问题是如何管理可用空间，删除记录需要回收的……
  - 把大小为n的记录放进大小m的记录先前占用的空间，要么空余要么放不进
  - 把page继续拆分，依旧存在上述问题
- 我们需要满足
  - 最小开销的变长存储需求
  - 回收已删除记录的空间
  - 引用页中的记录，无论它在哪



PostgreSQL

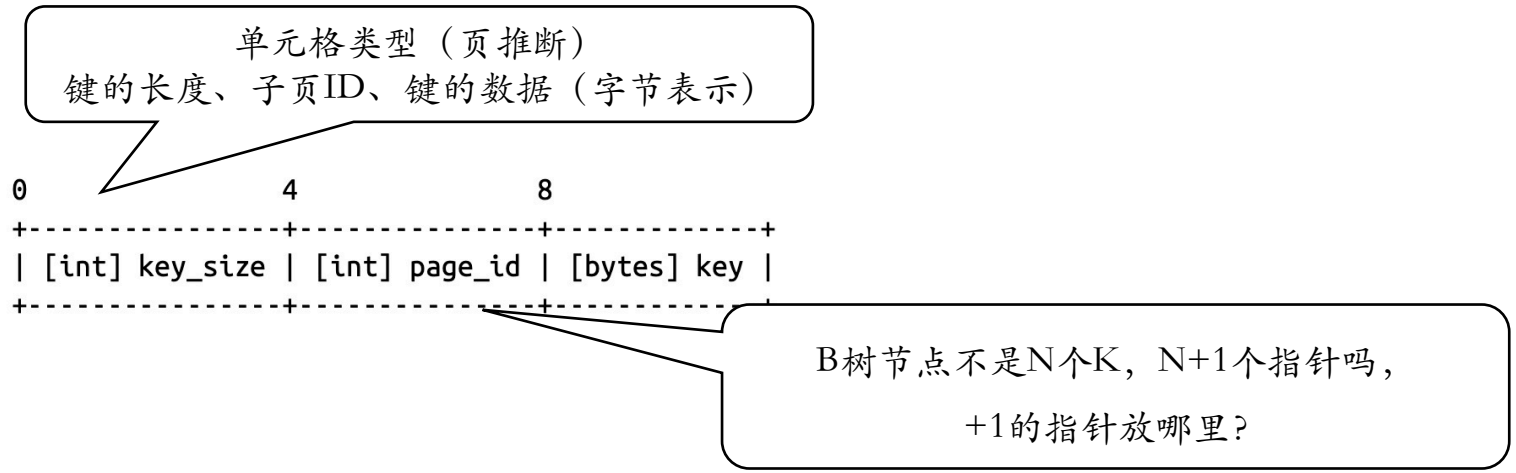
变长：分槽页（slotted page）



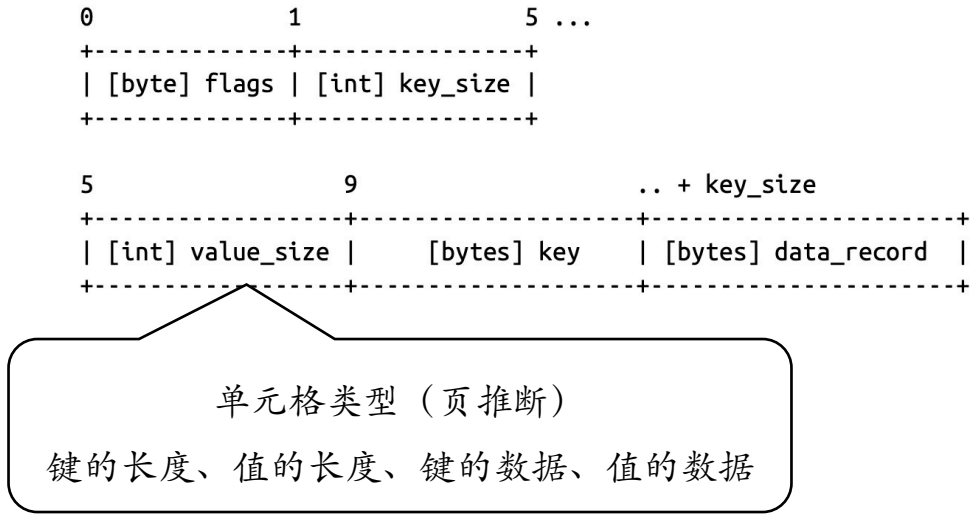
PostgreSQL

变长：分槽页（slotted page）的单元格布局（Cell Layout）

K Cell（键单元格）

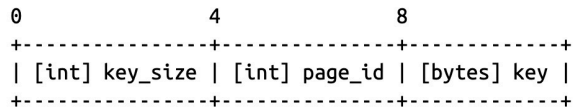


KV Cell（键值单元格）

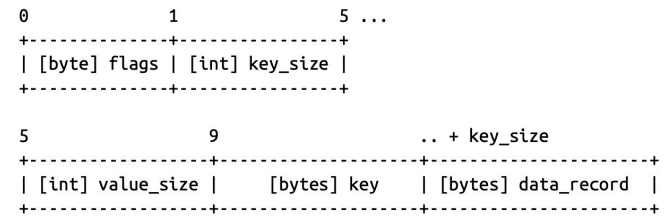


变长：分槽页（slotted page）的单元格布局（Cell Layout）

K Cell（键单元格）



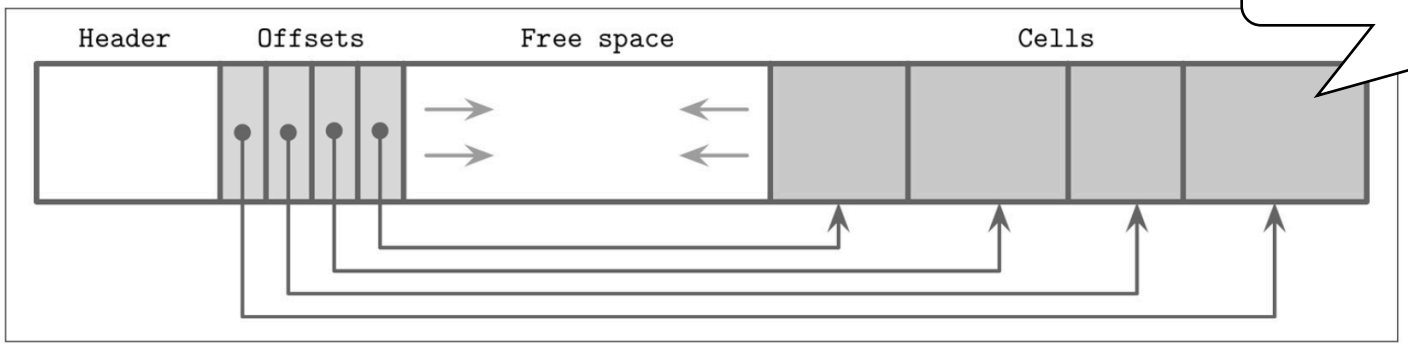
KV Cell（键值单元格）



单元格的键和值不需要定长了，可以变长，位置读取可以通过偏移量从定长的单元格头部计算得到

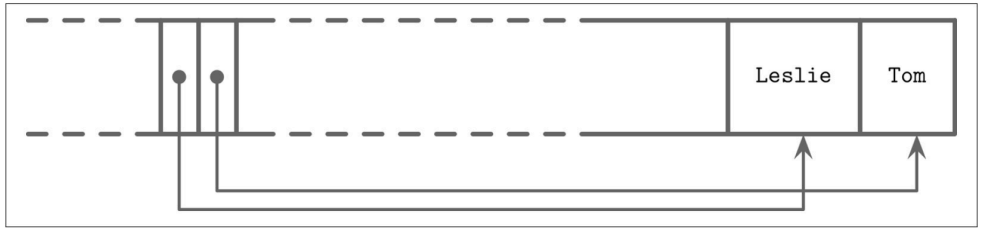
要找一个键，跳过单元格头部，读取key_size个字节。要找一个值，跳过单元格头部再加上key_size个字节，然后读取value_size

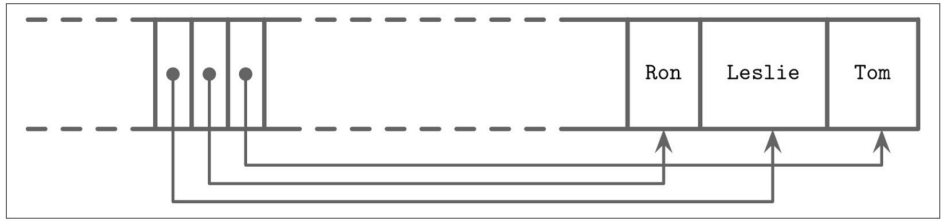
将单元格放入分槽页



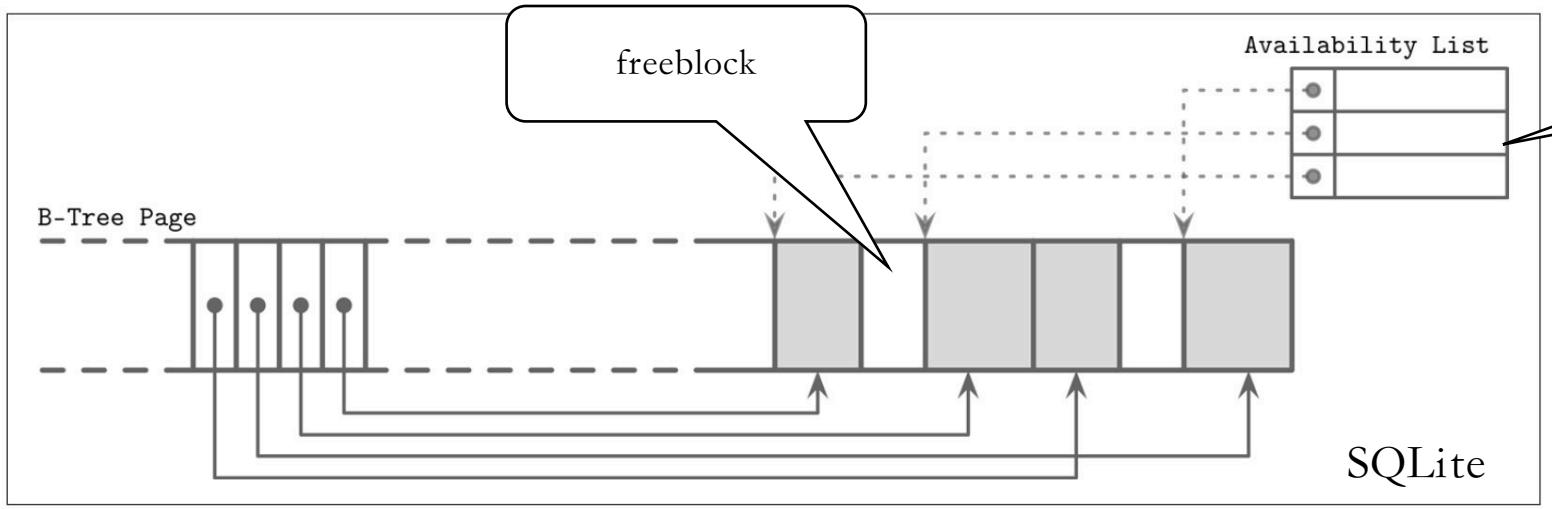
如果写入B树节点非顺序呢?

注意：字典顺序和写入数据的随机顺序





管理变长数据（del/modify）



一般存放页头，保留第一个空白指针和可用空间大小

构建freeblock，并指向第一个空闲块的指针保存在页头部，并保存可用字节数（确定是否能在碎片整理后被放入该页）

使用空闲块的策略

1）首次适配优先：找第一个适配的空闲块，会带来额外开销

2）最佳适配优先：找一个剩余段最小的空间

额外的小问题

- 版本
  - 文件名带版本前缀，Apache Cassandra，na，ma
  - 版本使用专门文件存储，PostgreSQL将版本存在PG_VERSION
  - 直接存在每一个具体文件（索引）的头部，头部按照不变格式编码
- Freeblock，留点空间……70%/30%原则
- 校验和：checksum/XOR，循环冗余校验CRC（检测连续比特位的损坏）

Folk, Michael J., Greg Riccardi, and Bill Zoellick. 1997. *File Structures: An Object- Oriented Approach with C++ (3rd Ed.)*. Boston: Addison-Wesley Longman.
Giampaolo, Dominic. 1998. *Practical File System Design with the Be File System (1st Ed.)*. San Francisco: Morgan Kaufmann.
Vitter, Jeffrey Scott. 2008. “Algorithms and data structures for external memory.” *Foundations and Trends in Theoretical Computer Science* 2, no. 4 (January): 305-474. *https://doi.org/10.1561/0400000014*.

页头（Page Header）

- 页头保存有关可用于页定位、维护和优化的信息
  - 包括——页内容和布局的标志位、单元格数量、空闲空间的上界下界的偏移量以及其他
  - PostgreSQL——页大小和布局版本
  - MySQL InnoDB——记录总数、层数和其他一些与实现相关的值
  - SQLite ——单元格的数量和最右指针

页头（Page Header）——Magic Numbers

- 一个多字节常数值
  - 标志：表示后面是一个 page
  - 标识：指定页的类型或者标识其版本
  - 意义：验证页加载和对齐是否正确

例如：Magic Number 为
50 61 47 45（Page）

你知道什么 Magic Number 的例子吗?

i = 0x5f3759df - (i >> 1);

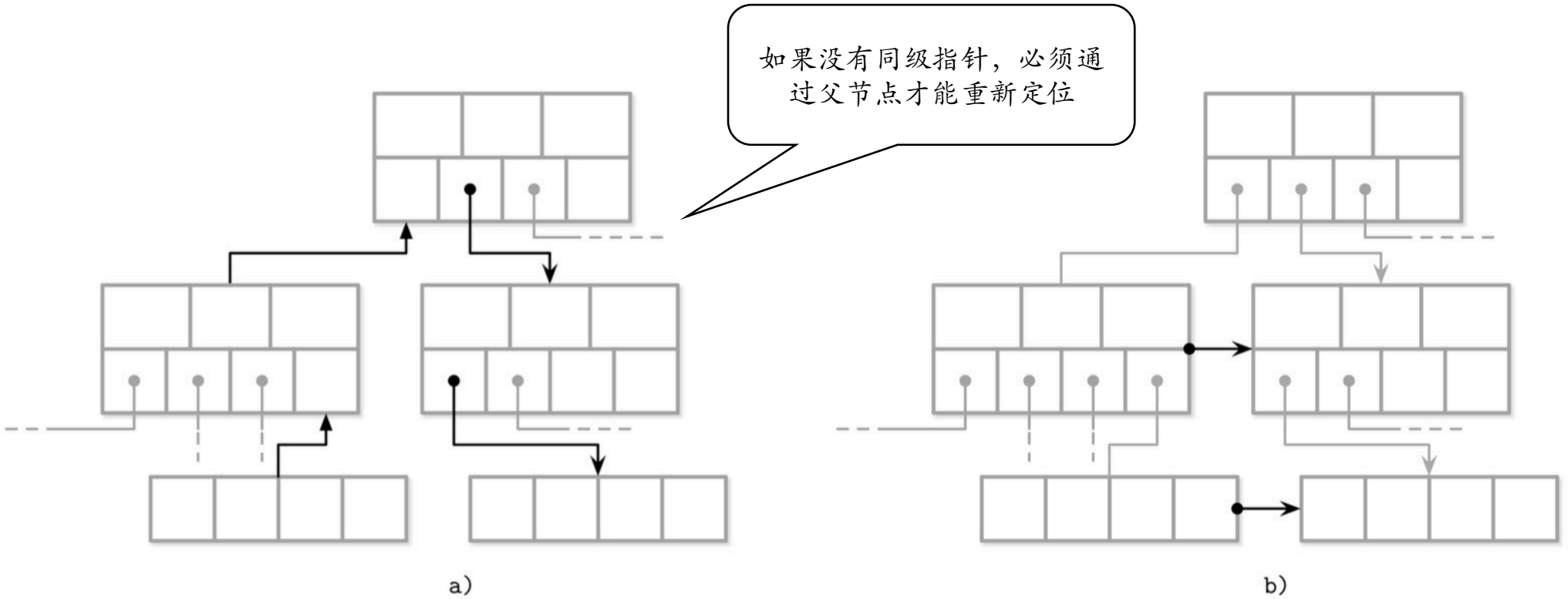
1.JPEG (jpg)，文件头：FFD8FF

2.PNG (png)，文件头：89504E47

3.GIF (gif)，文件头：47494638

4.Windows Bitmap (bmp)，文件头：424D

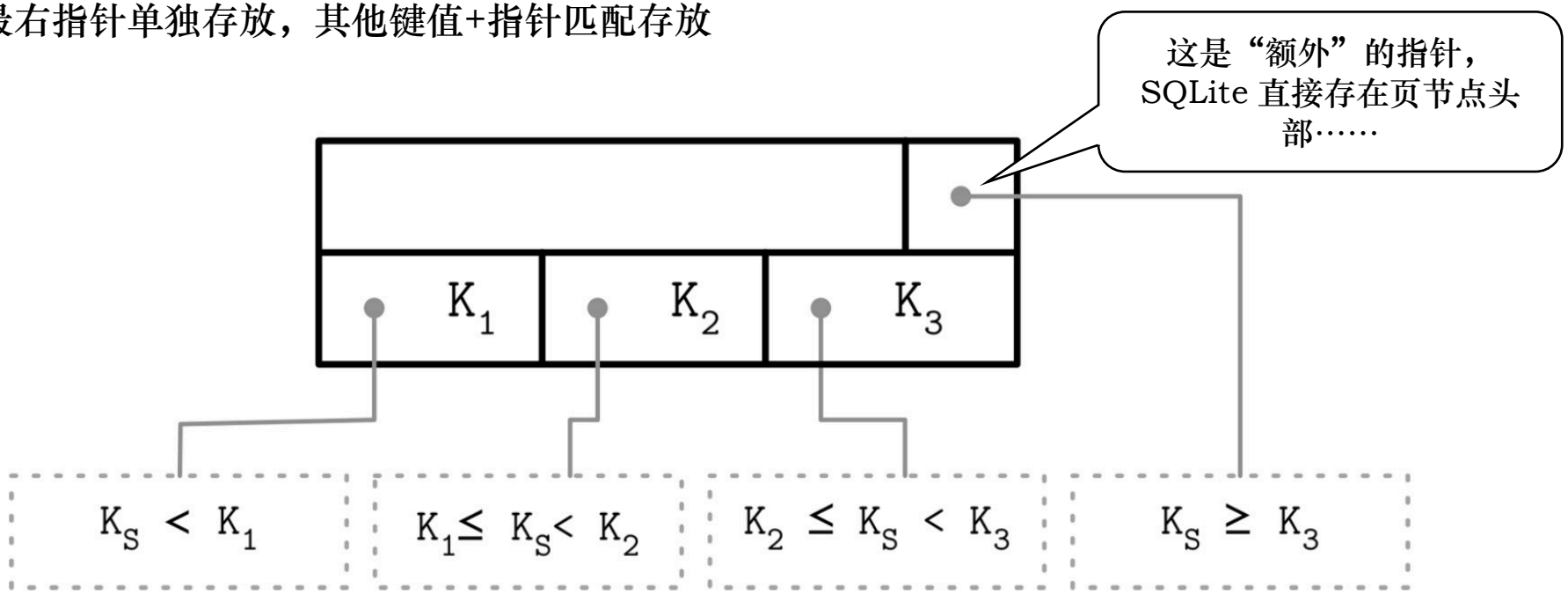
同级指针（Sibling Links）



同级指针的优点和缺点各是什么?

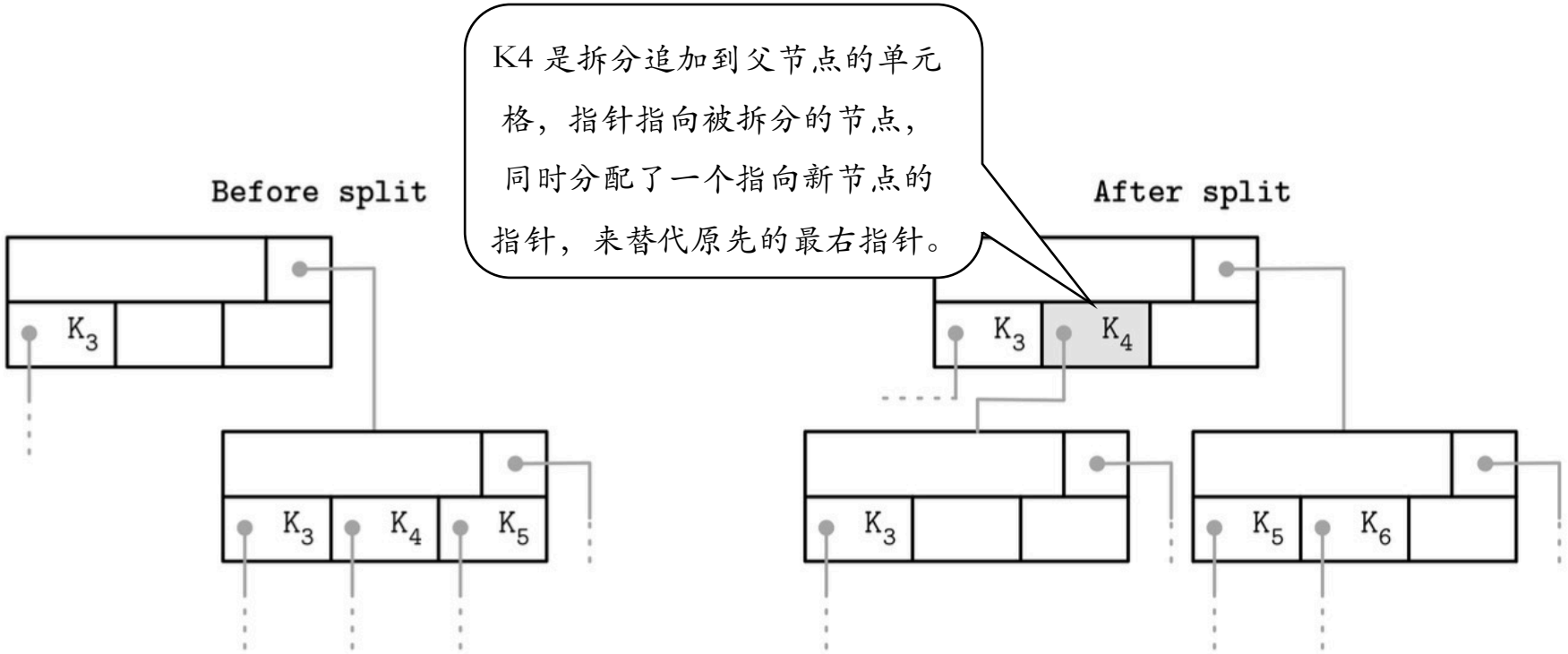
最右指针（Rightmost Pointers）

- B 数拆分子树并进行遍历，指向子页的指针总比键的个数多一个（指针 N+1）
  - 最右指针单独存放，其他键值+指针匹配存放



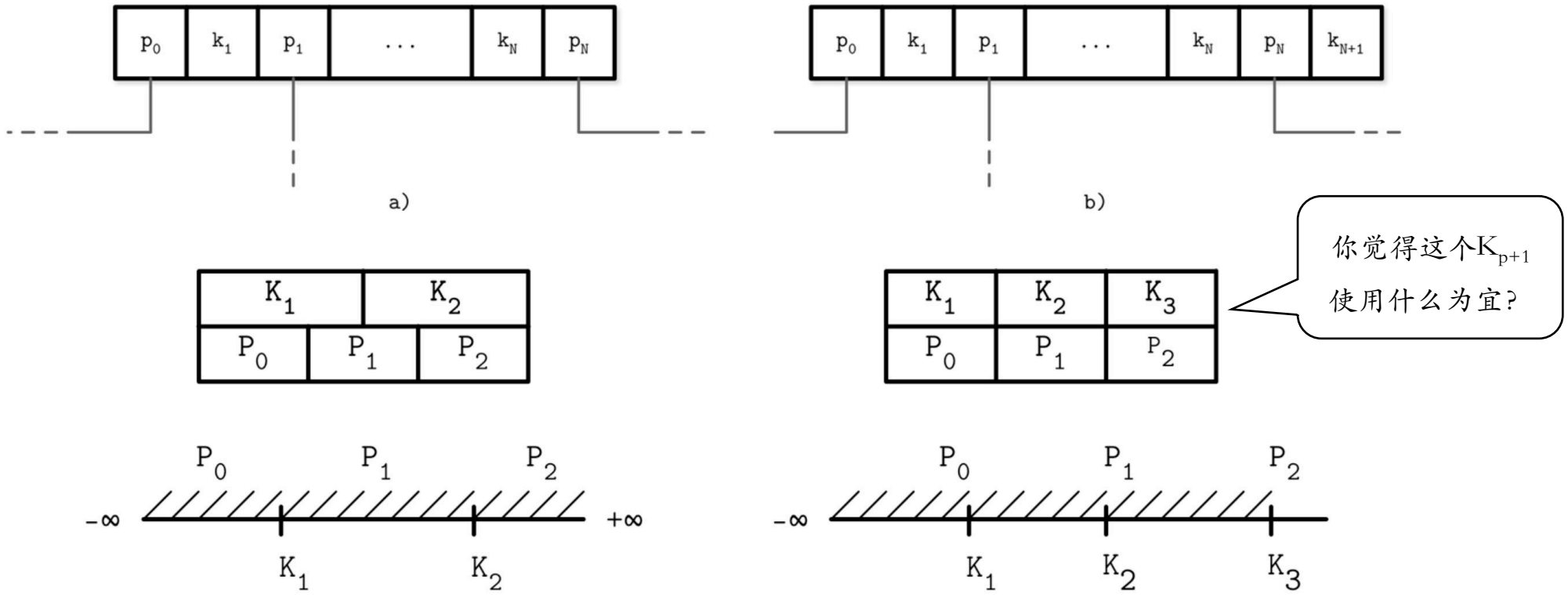
最右指针（Rightmost Pointers）

- 拆分后最佳到其父节点上，则必须对最右侧的子指针重新赋值



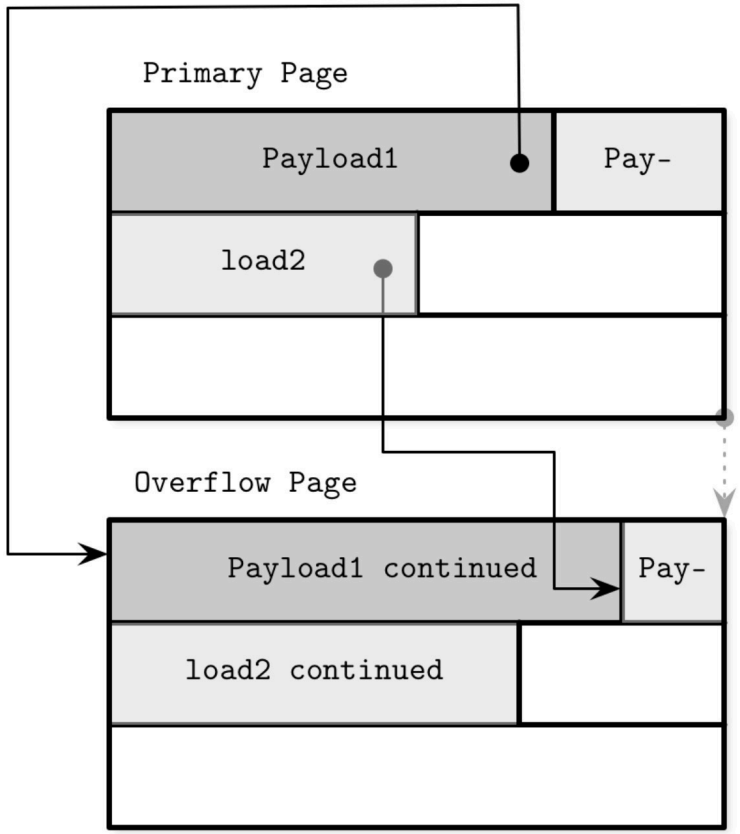
节点高键（ High Keys）

- 想一想？N+1这个单独节点，这样组织增加了复杂性，有没有可能“统一”？



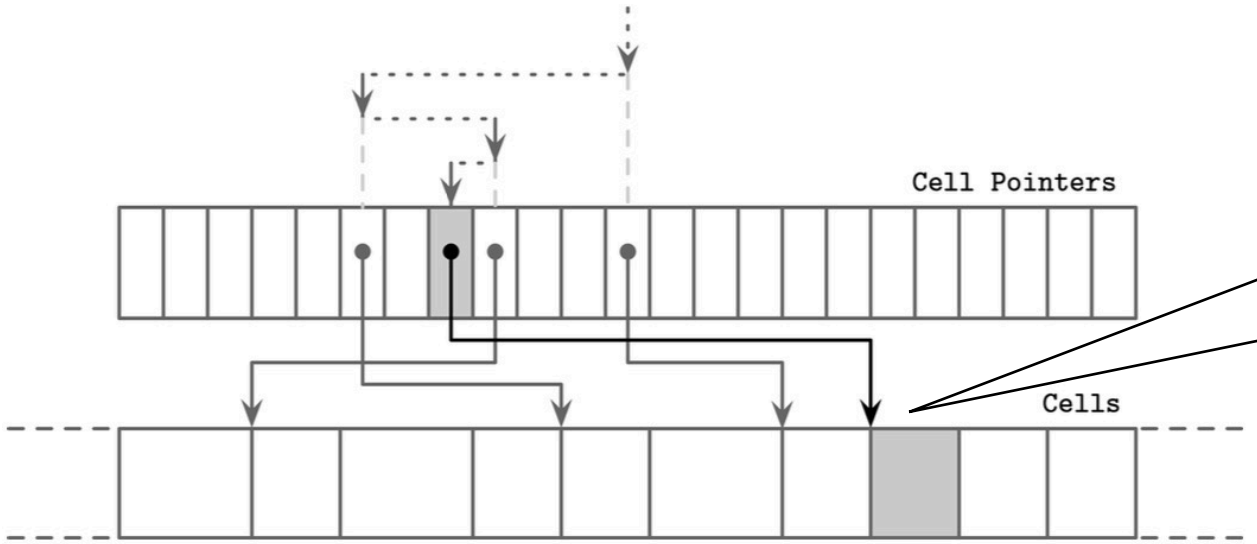
溢出页（Overflow Pages）

- 节点大小和树扇出是固定且不会动态改变的
- 我们也很难找到一个普遍最优的值
  - 树中存在变长值，并且它们足够大，那么页中只能放下少数几个值
  - 如果值很小，会浪费保留的空间
- B 树算法规定每个节点持有特定数量的元素
  - 所以，不得不面对一种方法增加或者扩展页大小



二分搜索（Binary Search）

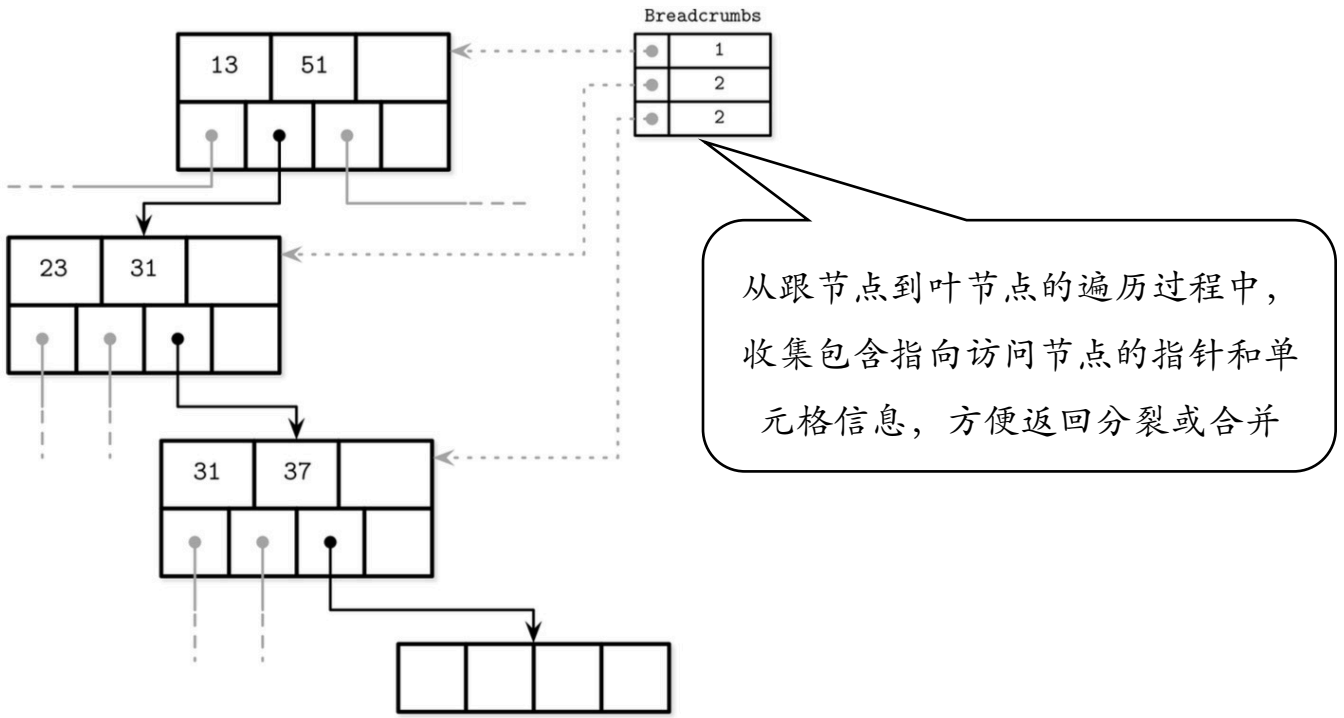
- 二分搜索算法的基本逻辑——返回是啥?
  - 输入已排序元素的数组和一个搜索键，
    - 返回一个整数，指定输入数组的位置，返回负数，表示没搜索到，并给出一个插入点
  - 插入点是第一个大雨给定键的元素下标。



带间接指针的二分搜索（灰色为要搜索的数据，虚线是通过指针进行二分搜索，实线是跟随指针的访问动作

分裂与合并

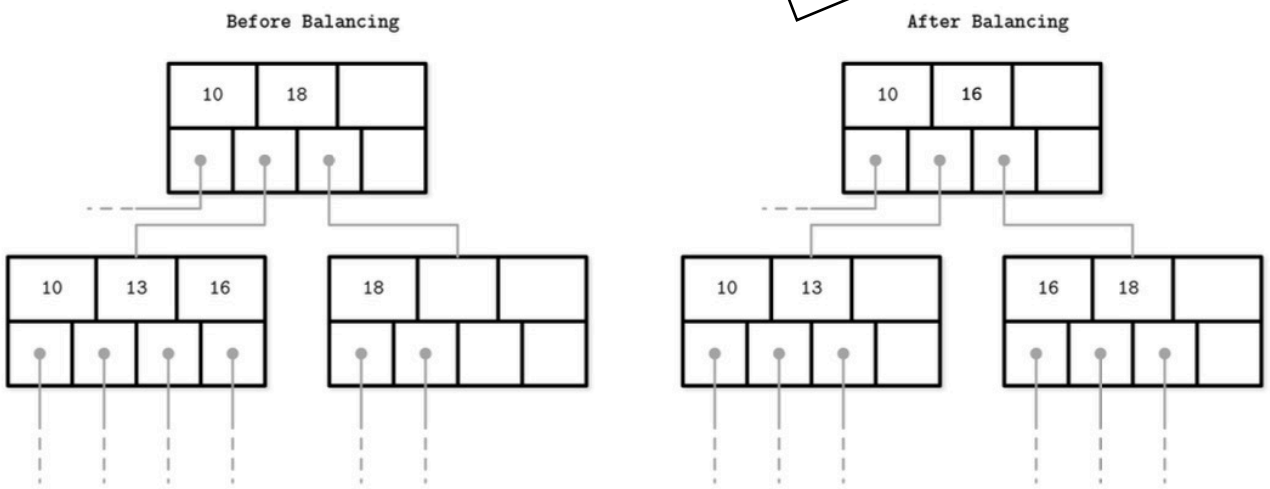
- B+树分裂和合并都是从叶节点发起，向上传递的（想一想，这样会出什么问题？）
  - 需要一条从将要分裂或合并的叶节点遍历回跟节点的路径
  - 用什么数据结构实现?
- 有没有其他方案?
  - 好处和不好分别是什么?



再平衡（Rebalancing）

- 频繁分裂和合并必然造成一些问题
  - 性能、节点利用率、树层数的问题（特别是部分节点的频繁分裂合并）
- 有没有推迟分裂的方案？——再平衡
  - 提高平均占用率
  - 但需要额外的跟踪和平衡
  - 更高的利用率意味更高效的搜索

SQLite实现同级节点的平衡算法，很多其他数据库将此算法作为优化手段进行整理时实现。

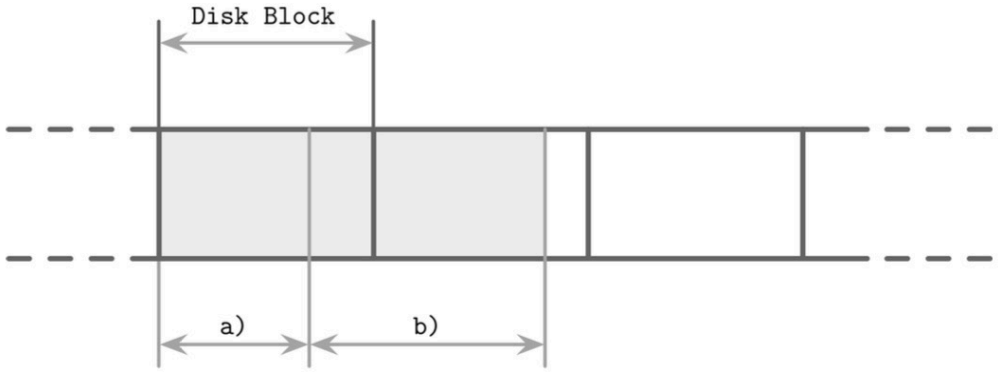


右侧追加（Right-Only Appends）

- 自增数值作为主索引的优势在优化方面
  - 所有的插入都在索引的末尾（最右侧的叶子），大量的分裂集中在每层的最右边）
  - PostgreSQL叫fastpath，新条目的插入可以跳过整个读取路径
  - SQLite叫quickbalance，右节点满的时候不再平衡或拆分，直接新分配节点
- 这种特征，怎么在非自增数值做主索引中应用?
  - 批量加载，重新构建（自下而上的构建树，逐层写入）
  - 不可变B+树

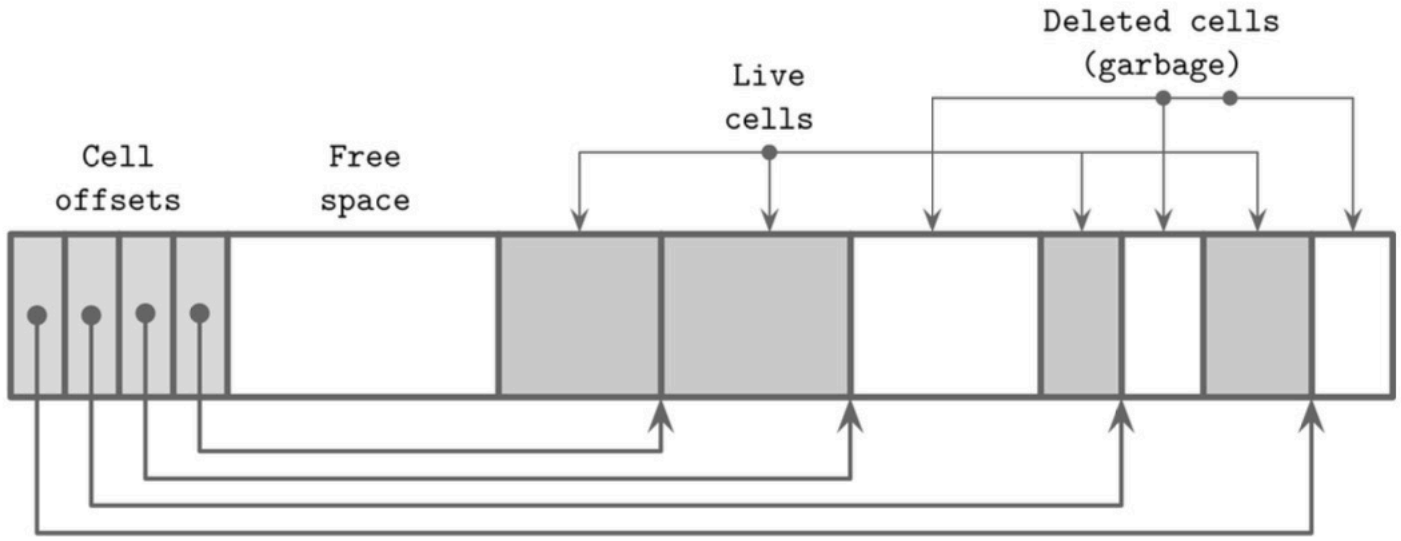
压缩

- 压缩本质上也是一种平衡，访问速度vs.压缩率（空间利用率），时空平衡
- 不同粒度下的压缩
  - 文件压缩（好处、坏处、局限），应用在较小文件
  - 按页压缩数据（易于理解，问题是什么？）
  - 按记录行/列压缩（Snappy，zLib，lz4）
- 压缩算法选择的基本逻辑
  - 压缩比、性能或内存开销
  - 指标：内存开销、压缩性能、解压性能、压缩比



维护（Vacuum and Maintenance）

- 一个软件系统，除了面向用户的操作之外，还有面向系统本身的操作
  - 维护存储的完整性、回收空间、减少开销和维持Page的有序
  - 后台执行，批处理



删除的基本操作：填空/无法寻址

维护—更新和删除碎片

- 填空（Null）或删除偏移量
  - 删除偏移量是最常使用的方式，体会一下Page为什么要设置偏移量（分离）
  - 有些数据库又专门的垃圾收集功能，将删除和更新的单元格留在原处，以便进行多版本控制（后续再详说）
  - 事务完成后，ghost record的数据结构被回收
- 碎片化单元格（fragmented）需要被碎片整理
  - 重写页面，更新操作对叶节点页面也可以更新偏移量，不重写页面
  - 碎片整理（compaction、vacuum……）
  - 更新偏移量、移动新位置、可用磁盘页id进入空闲页列表（freelist）

B+树物理实现的总结

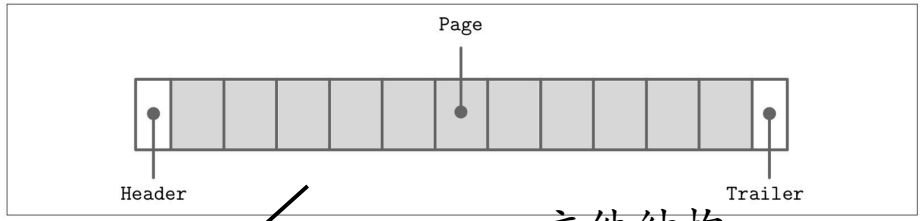
- 系统、软件工程等所有工程学科都一样——魔鬼都在细节中
- **静态**：编码、页结构、页头、最右指针、高键、溢出页
- **动态**：遍历、间接指针二分搜索、父指针或导航结构
- **维护**：再平衡、右侧追加、批量加载、垃圾收集

基于磁盘的B树

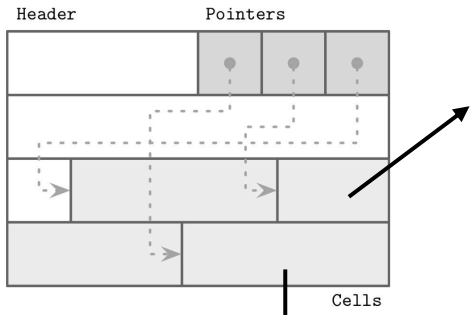
Graefe, Goetz. 201. " Modern B-Tree Techniques." Foundations and Trends ni Databases 3, no. 4(April): 203-402. https://doi.org/10.1561/1900000028.

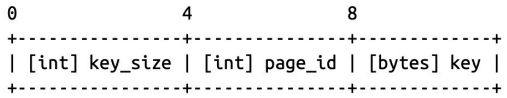
Healey, Christopher G. 2016. Disk-Based Algorithms for Bgi Data (Ist Ed.). Boca Raton: CRC Press.

静态分析

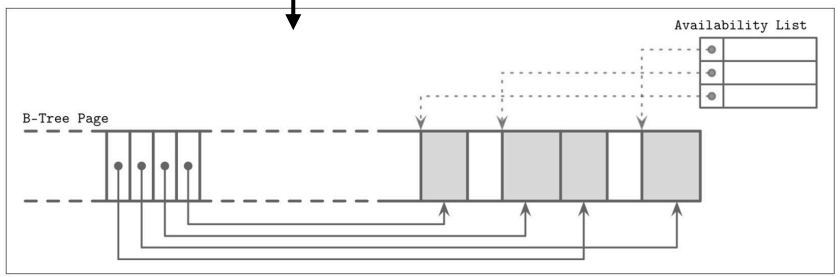


文件结构

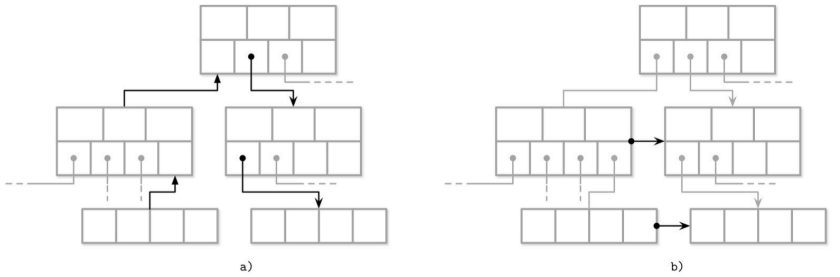




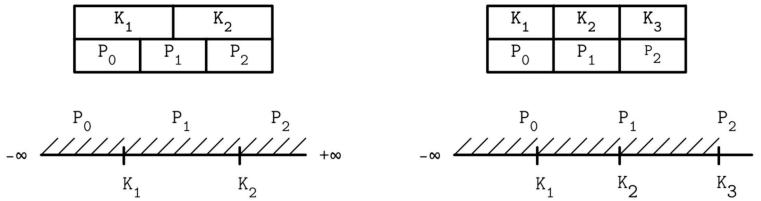
页结构



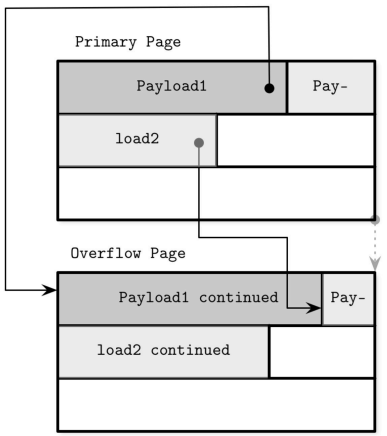
指针和数据分离



同级指针

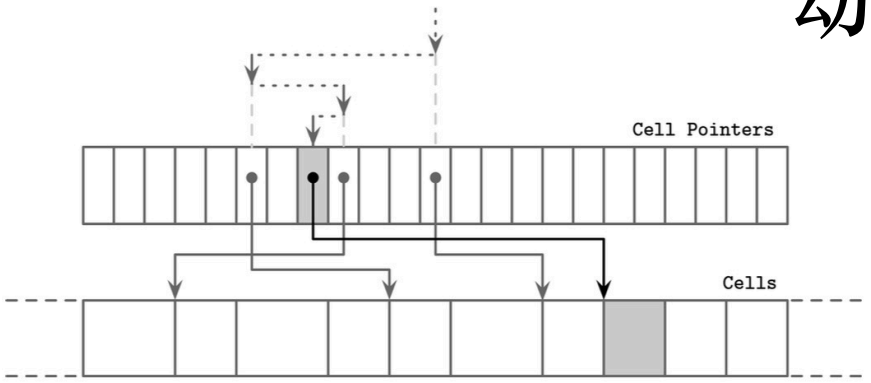


高键

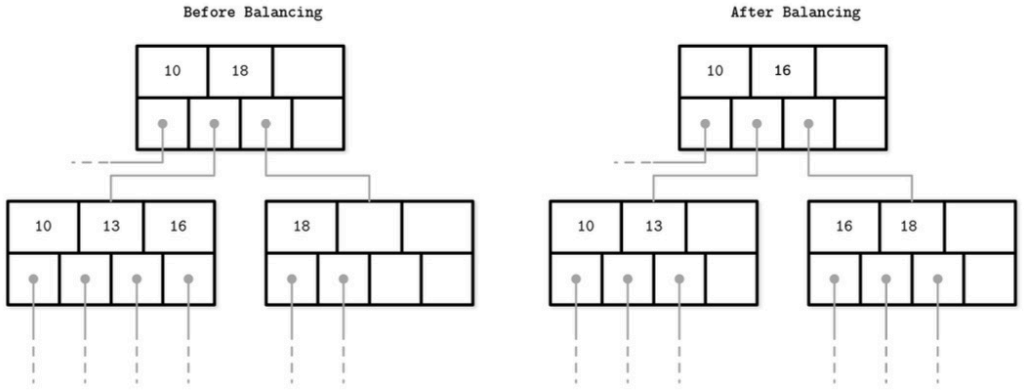


溢出页结构

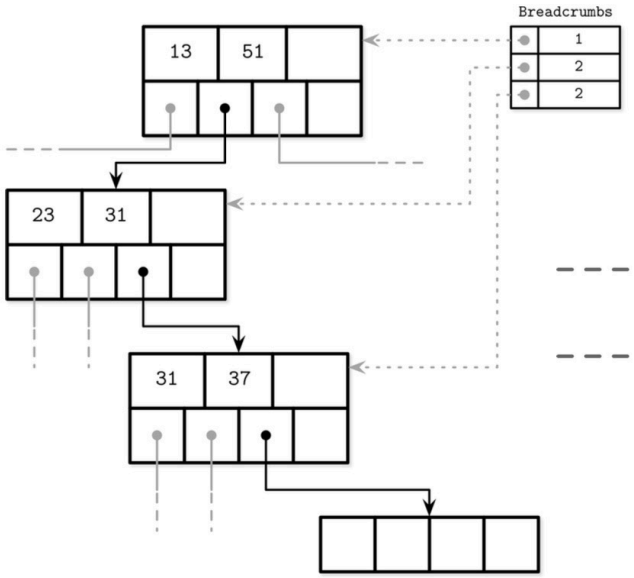
动态分析和维护



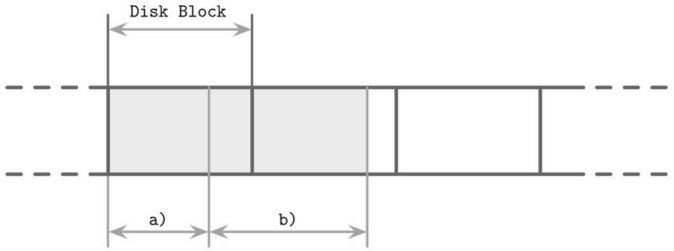
带间接指针的二分搜索

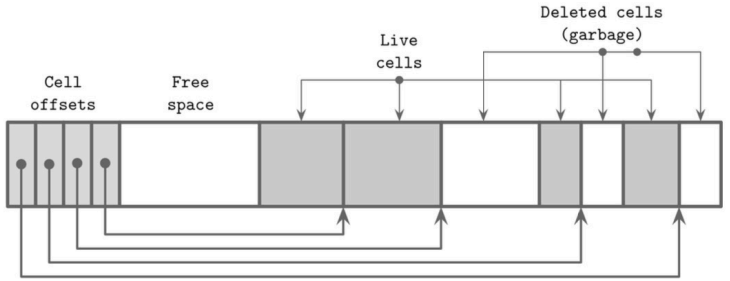


再平衡的优化手段



额外栈实现分裂合并的回溯





压缩，及空闲区域维护和整理

Practice in class 4-1

- 尝试构建/阅读MySQL/其他手边的数据库的B+树的结构，实现插入、更新和删除？以及相关的节点分裂和合并；
- 尝试阅读数据库其他索引结构的物理实现；
- 通过对其他平台的学习构建对实现级别的学习路径（从静态结构到动态行为再到维护管理）

数据库技术冲突的目标

- 并发用户数很大的系统
  - 尽量以紧凑的方式存储数据
  - 尽量将数据分散存储
- 没有并发的修改密集型（change-heavy）
  - 数据查询要快
  - 数据更新也要快…
- DBMS所处理的基本单元（页、块）通常不可分割
- 总结：读写不会和睦相处，怎么和谐啊～～

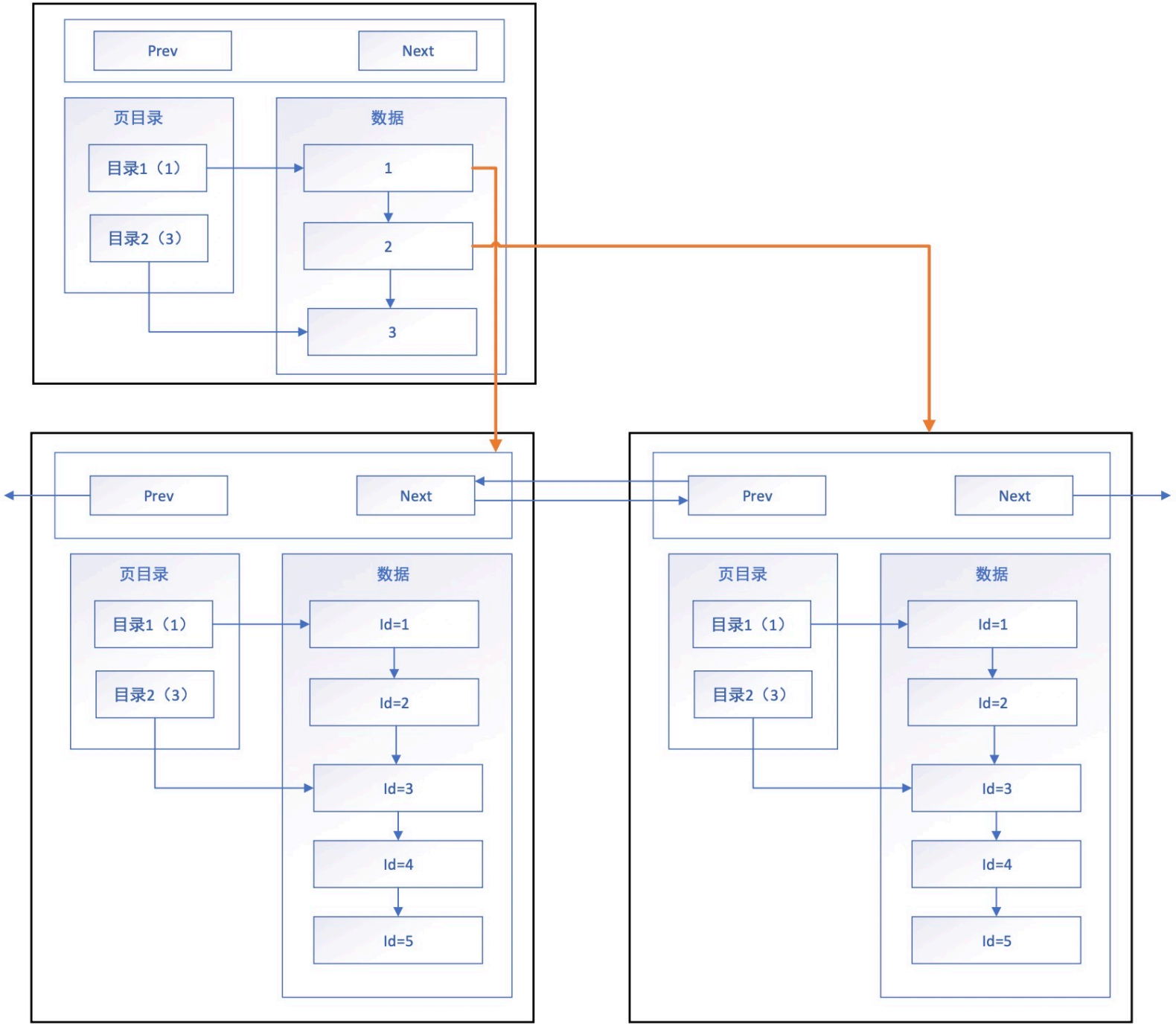
不要有任何抱怨，“和谐”就是每个程序员工作的全部

基本表的页模式

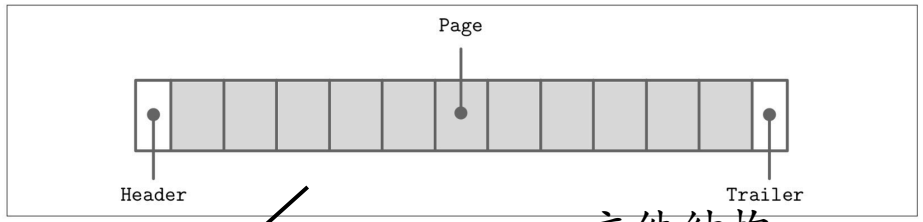
链表模式的目的是优化查询效率

目录页的本质也是页，普通页中存的数据是项目数据，而目录页中存的数据是普通页的地址。

基本表的本质仍然是类似树状结构，但是所有的叶节点都是随机存储的文件而已。

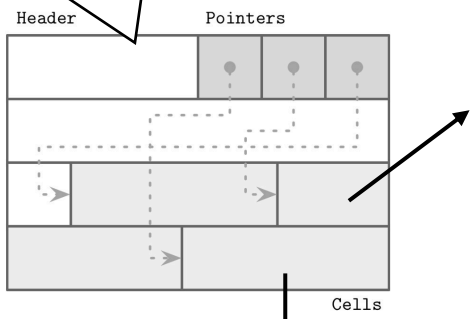


堆文件的静态分析



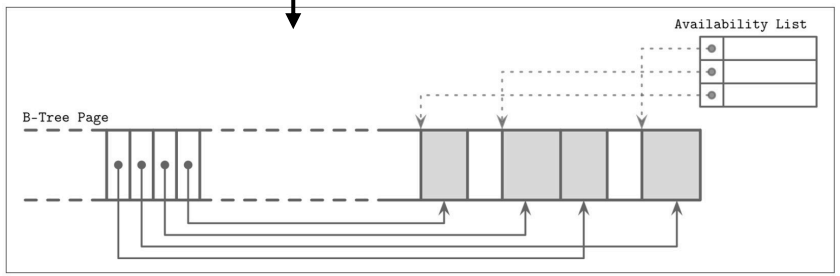
文件结构

表头的差异在标识

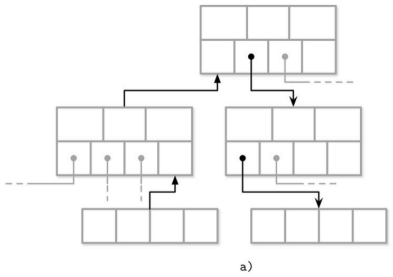


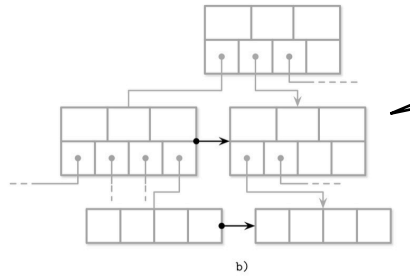

```
0               4               8
+---------------+---------------+-------------+
| [int] key_size | [int] page_id | [bytes] key |
+---------------+---------------+-------------+
```

页结构



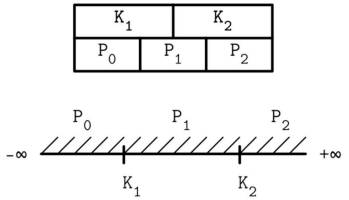
指针和数据分离

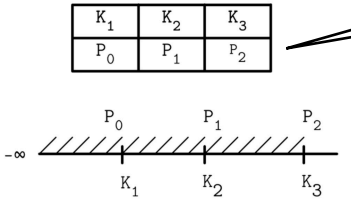




“叶节点”链表形态就是同级指针

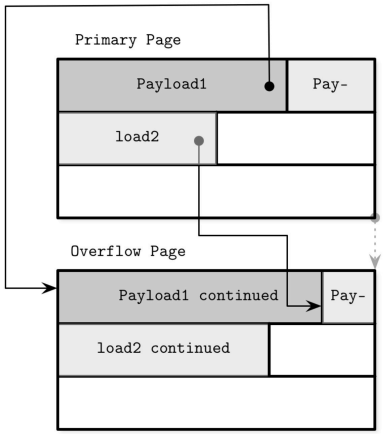
同级指针





高键恰好统一了

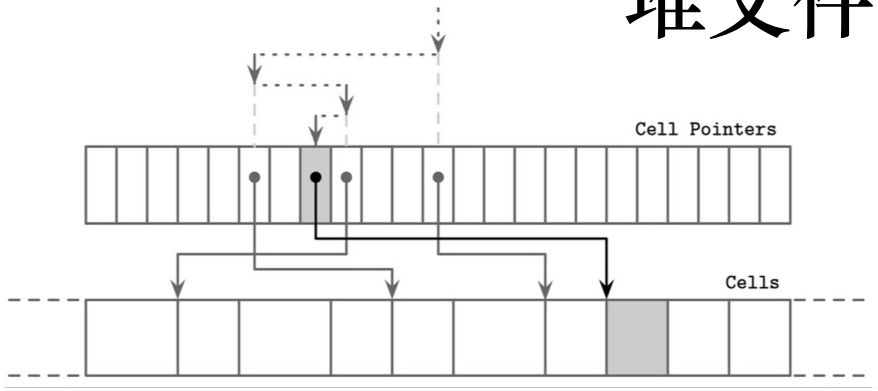
高键



溢出页的差异在次数

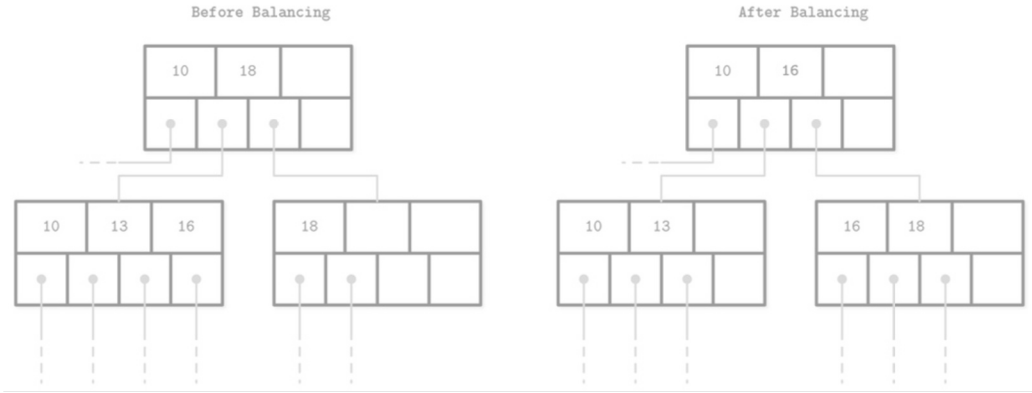
溢出页结构

堆文件的动态分析和维护

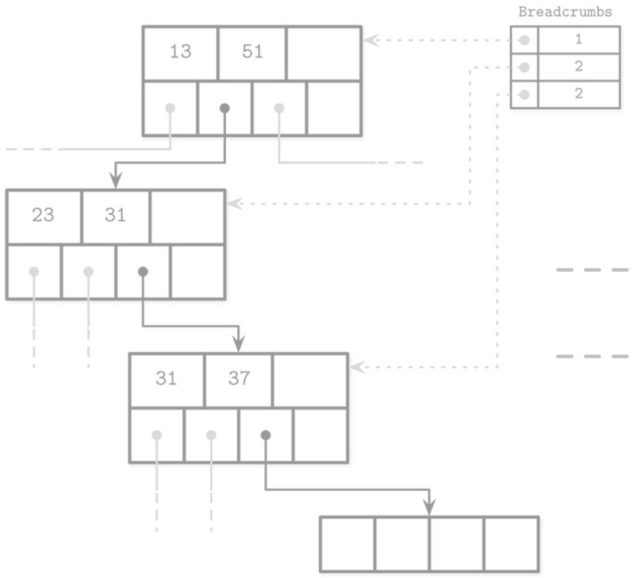


搜索算法的差异

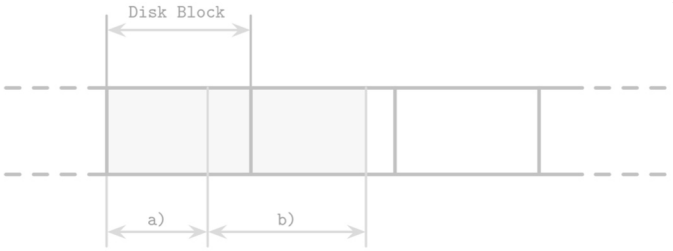
基于hash链表的查找

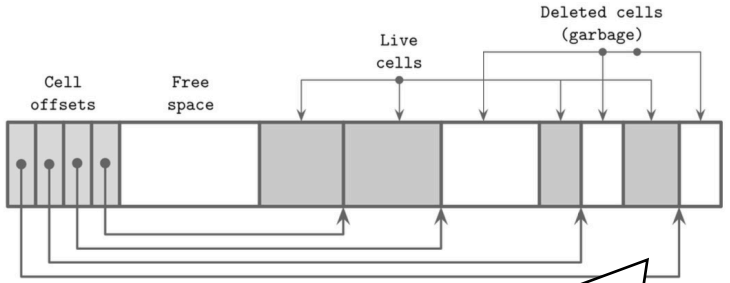


再平衡的优化手段



额外栈实现分裂合并的回溯





插入维护的算法依旧相似

压缩，及空闲区域维护和整理

数据库的具体文件的物理组织形式本质都是树

- 相似结构、但不同取值以及不同动态使用算法的树
  - 顺序文件（索引，B+树等）
  - 散列文件（hash）
  - 随机文件（堆文件，基本表结构）
  - 聚簇文件（融合性文件）

Clustered存在不同粒度，重点不是粒度，而是物理结构上的融合

Clustered Tables

Orders table
Not clustered

Order_Date	Country	Status
2022-08-02	US	Shipped
2022-08-04	JP	Shipped
2022-08-05	UK	Canceled
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-05	US	Processing
2022-08-04	JP	Processing
2022-08-04	KE	Shipped
2022-08-06	UK	Canceled
2022-08-02	UK	Processing
2022-08-05	JP	Canceled
2022-08-06	UK	Processing
2022-08-05	US	Shipped
2022-08-06	JP	Processing
2022-08-02	KE	Shipped
2022-08-04	US	Shipped

Orders table
Clustered by Country

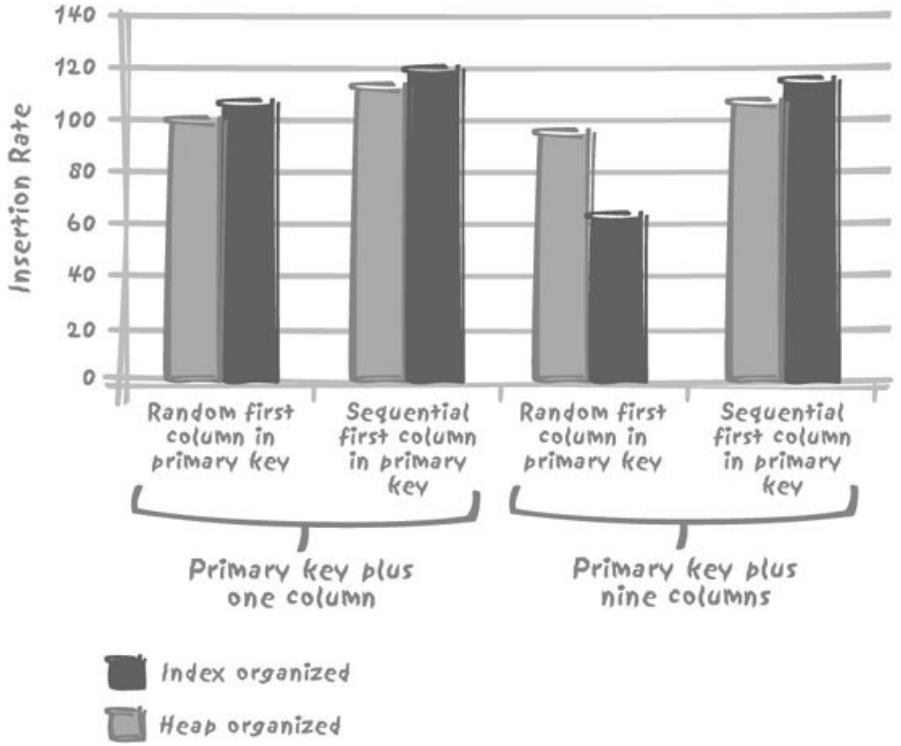
Order_Date	Country	Status
2022-08-02	US	Shipped
2022-08-05	US	Shipped
2022-08-05	US	Processing
2022-08-04	US	Shipped
2022-08-04	JP	Shipped
2022-08-04	JP	Processing
2022-08-05	JP	Canceled
2022-08-06	JP	Processing
2022-08-05	UK	Canceled
2022-08-06	UK	Canceled
2022-08-06	UK	Processing
2022-08-02	UK	Processing
2022-08-06	KE	Shipped
2022-08-02	KE	Canceled
2022-08-04	KE	Shipped
2022-08-02	KE	Shipped

Orders table
Clustered by Country and Status

Order_Date	Country	Status
2022-08-05	US	Processing
2022-08-02	US	Shipped
2022-08-05	US	Shipped
2022-08-04	US	Shipped
2022-08-05	JP	Canceled
2022-08-04	JP	Processing
2022-08-06	JP	Processing
2022-08-04	JP	Shipped
2022-08-05	UK	Canceled
2022-08-06	UK	Canceled
2022-08-06	UK	Processing
2022-08-02	UK	Processing
2022-08-02	KE	Canceled
2022-08-06	KE	Shipped
2022-08-04	KE	Shipped
2022-08-02	KE	Shipped

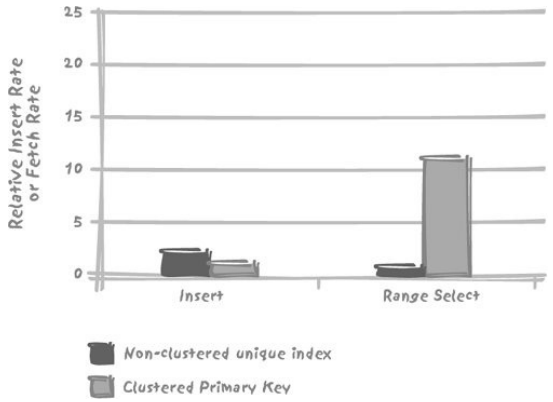
把索引当成数据仓库

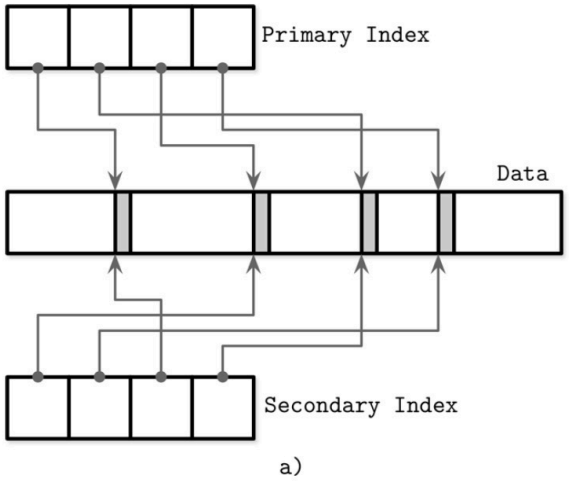
- 当索引中增加额外的字段（一个或多个，它们本身与实际搜索条件无关，但包含查询所需的数据），能提高某个频繁运行的查询的速度。
- 尽量在索引中多存储数据的极限是？--允许在主键索引中存储表中所有数据，表就是索引
  - Oracle：“索引组织表（index-organized table，IOT）”
  - MySQL innoDB的主键聚簇索引（MyISAM则是非聚簇索引）

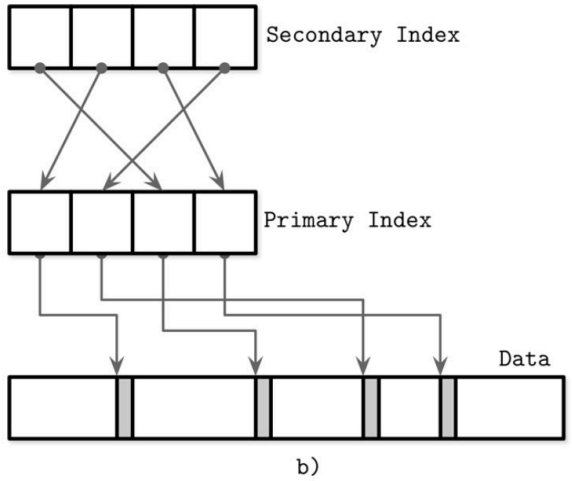


记录强制排序

- IOT最大的优点：记录是排序的…（效率惊人）
- 记住一点：任何有序数据便于某些处理的同时，必将对其他处理不利
  - 表变成了树状结构……
  - 这是失传已久的“层次型数据库”







一个性能问题解决通用的逻辑

时间和空间的平衡

分散与集中的平衡

包含——集中分布、实时异步、强弱一致性、存储密集与分散、顺序与随机等等

读取与写入的平衡

最关键的一点——计算机虽然是0和1，但决策是复杂的，在0和1之间有一片广阔的天地

既要……还要……又要……不是错，这就是面对复杂现实的基本思想，关键是要的“程度”

数据自动分组（grouping）

- 分区（partition）也是一种数据分组的方式
  - 提高并发性（concurrency）和并行性（parallelism）
  - 从而增强系统架构的可伸缩性（scalable）
- 面对两大问题
  - 管理问题（备份和恢复）
  - 数据量巨大的表，B+树索引失效，非聚簇索引的大量随机I/O问题

两个策略

1）全量扫描数据，不需要任何索引

2）索引数据，并分离出热点数据

循环分区

- 循环分区：不受数据影响的内部机制
  - 分区定义为各个磁盘的存储区域
  - 可以看作是随意散布数据的机制
  - 保持更改带来的磁盘I/O操作的平衡

数据驱动分区

- 根据一个或多个字段中的值来定义分区
  - 一般叫分区视图（partitioned view），而MYSQL老版本称为（merge table）
- 分区的实现方式
  - 哈希分区（Hash-partitioning）
  - 范围分区（Range-partitioning）/键值分区
  - 列表分区（List-partitioning）

分区表的底层实现原理

- 分区表是由多个相关的底层表实现的
  - 底层表也是由句柄对象（Handler object）表示，可以直接访问各个分区
  - 存储引擎的角度来看，底层表和一个普通的表没有任何去表，它并不管是普通表还是分区表的一部分
  - 分区表的索引是在底层表上各自加上一个完全相同的索引
- 分区表上的操作逻辑
  - SELECT，分区层打开并锁住所有底层表，优化器判断过滤部分分区，在调用引擎接口访问
  - INSERT，分区层打开并锁住所有底层表，确定哪个分区接受这条记录，在写入底层表
  - DELETE，分区层打开并锁住所有底层表，确定对应分区，在进行删除操作
  - UPDATE，分区层打开并锁住所有底层表，确定分区，取出数据更新，再决定放入哪个分区，删除原纪录

分区是把双刃剑

- 分区能解决并发问题吗?
  - 分区想解决的目标——查询能过滤掉很多额外分区、分区本身不会带来很多额外代价
- 又回到了IOT类似的问题：“冲突”
  - A. 通过分区键将数据聚集，利于高速检索；
  - B. 对并发执行的更改操作，分散的数据可以避免访问过于集中的问题
- So，A or B……完全取决于您的需求

分区是把双刃剑

- NULL值会使分区过滤无效（PATITION by RANGE COLUMN（order_date））
- 分区列和索引列不匹配（没有索引，或关联查询时关联条件不匹配索引）
- 选择分区的成本可能很高（范围分区的成本需要注意）
- 打开并锁住所有底层表的成本可能很高（开销和分区类型无关，主键查找单行会带来明显开销）
- 维护分区的成本可能很高

分区与数据分布

- 表非常大，且希望避免并发写入数据的冲突就一定要用分区吗?
  - 例如客户订单明细表……
- 对分区表进行查询，当数据按分区键**均匀分布**时，收益最大

假设有个很大的客户订单明细表，如果该表中大部分数据都来自于同一个客户，那么按照客户的ID对数据进行分区，就不会有太大的帮助。

我们可以粗略的把查询分为两大类，与大客户相关的查询及与较小的客户相关的查询。当查询小客户数据的时候，在客户ID上的索引可选择性就会会高。因此查询效率也会很高，这时候，其实完全就不需要分区。相反，查询大客户扫描表是效率最高的处理方式，由于大客户数据占表比例过高，扫描分区也不会快多少。

数据分区的最佳方法

- 整体改善业务处理的操作，才是选择非缺省的存储选项的目标
- 更新分区键会引起移动数据，似乎应该避免这么做
  - 例如实现服务队列，类型（T1…Tn）状态（{W|P|D}）
  - 按请求类型分区：进程的等待降低
  - 按状态分区：轮询的开销降低
  - 取决于：服务器进程的数量、轮询频率、数据的相对流量、各类型请求的处理时间、已完成请求的移除频率
- 对表分区有很多方法，显而易见的分区未必有效，一定要整体考虑

数据分区的问题

- 一个看上去很帅的东西，一定有地方令人非常讨厌（技术和人都是如此）
- 分区的一些缺点，大数据量和高并发下
  - 如果SQL不走分区键，很容易造成全表锁，或者多次调用相同索引
  - 在分区中实现关联查询，就是一个灾难
  - 分区表，隐藏复杂，使得工程师不可控

但对于应用和管理上，最困难的两个问题是：
索引的复制问题、分区层锁定的问题

分区、分表、分库

- 分区
  - 就是把一张表的数据分成N个区块，在逻辑上看最终只是一张表，但底层是由N个物理区块组成的
- 分表（手搓分区）
  - 就是把一张表按一定的规则分解成N个具有独立存储空间的实体表
  - 系统读写时需要根据定义好的规则得到对应的字表明，然后操作它
- 分库

分区、分表、分库

- IO瓶颈
  - 热点数据太多，数据缓存不够，每次查询产生大量IO——分库、垂直分表
  - 网络IO瓶颈，请求的数据太多，带宽不够、连接数过多——分库
- CPU瓶颈
  - SQL问题，join、group by、order by——SQL优化，构建索引
  - 单表数据量过大，扫描行太多，SQL效率过低——水平分表

分表解决的问题

- 分表后单表的并发能力提高了，磁盘I/O性能也提高了，写操作效率提高了
- 数据分布在不同的文件，磁盘I/O性能提高
- 读写锁影响的数据量变小
- 插入数据库需要重新建立索引的数据减少
- 分表的实现方式（复杂）
  - 需要业务系统配合迁移升级，工作量较大

如何保证插入不同表的多条记录（事务）要么同时成功，要么同时失效？（TCC柔性事务）

分区和分表的区别与联系

- 分区和分表的目的都是减少数据库的负担，提高表的增删改查效率。
- 分区只是一张表中的数据的存储位置发生改变，分表是将一张表分成多张表。
  - 当访问量大，且表数据比较大时，两种方式可以互相配合使用
  - 当访问量不大，但表数据比较多时，可以只进行分区
  - 分表可以多库，分区不能
- 常见分区分表的规则策略（类似）

分库

- 什么时候考虑使用分库?
  - 单台DB的存储空间不够
  - 随着查询量的增加单台数据库服务器已经没办法支撑
- 分库解决的问题
  - 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

分库的方法

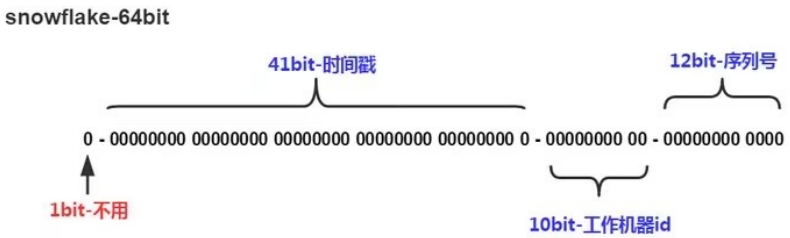
- 垂直拆分
  - 将不存在关联关系或者不需要join的表可以放在不同的数据库不同的服务器中
  - 按照业务垂直划分。比如：可以按照业务分为资金、会员、订单三个数据库
  - 需要解决的问题：跨数据库的事务、join查询等问题
- 水平拆分
  - 例如，大部分的站点。数据都是和用户有关，那么可以根据用户，将数据按照用户水平拆分
  - 按照规则划分，一般水平分库是在垂直分库之后的。比如每天处理的订单数量是海量的，可以按照一定的规则水平划分
  - 需要解决的问题：数据路由、组装
- 读写分离
  - 对于时效性不高的数据，可以通过读写分离缓解数据库压力
  - 需要解决的问题：在业务上区分哪些业务上是允许一定时间延迟的，以及数据同步问题

分库的问题和解决方案

- 问题
  - 事务的支持，分库分表，就变成了分布式事务
  - join时跨库，跨表的问题
  - 分库分表，读写分离使用了分布式，分布式为了保证强一致性，必然带来延迟，导致性能降低，系统的复杂度变高
- 常用的解决方案：
  - 对于不同的方式之间没有严格的界限。需要根据实际情况，结合每种方式的特点来进行处理
  - 选用第三方的数据库中间件（Atlas，Mycat，TDDL，DRDS），同时业务系统需要配合数据存储的升级

小问题：全局ID生成策略

- 自动增长列
  - 自带功能、有序、性能不错
  - 单库单表没问题，但分库分表需要手动规划（自增偏移+步长；全局ID映射表Redis）
- UUID（128位）：
  - 简单，全球唯一
  - 存储和传输空间大，无序，性能欠佳
- COMB（组合）
  - GUID（10字节）+时间（6字节）
- Snowflake（雪花算法）
  - Twitter开源的分布式ID生成算法，结果是long（64bit）数值。
  - 其特征是各个节点无需协调，按时间大致有序，且整个集群各个节点不重复。



Holy Simplicity

- 除了堆文件之外的任何存储方法，都会带来复杂性
- 除了单库单表之外任何的存储方式，都会带来复杂性
- 选错存储方式会带来大幅度的性能降低
- 总结
  - A. 测试，测试，测试
  - B. 设计是最重要的
  - C. 任何设计都有时效性