Index Concurrency Control

HASH TABLE LATCHING

Approach #1: Page Latches 一个page一个latch (acquire的次数少，block的内容多)
→ Each page has its own reader-writer latch that protects its entire contents.
→ Threads acquire either a read or write latch before they access a page.
find使用read latch。insert使用write latch。
read latch可以多thread一起获得。write latch只能一个thread获得。

Approach #2: Slot Latches 一个slot一个latch (acquire的次数多，block的内容少)
→ Each slot has its own latch.
→ Can use a single-mode latch to reduce meta-data and computational overhead.

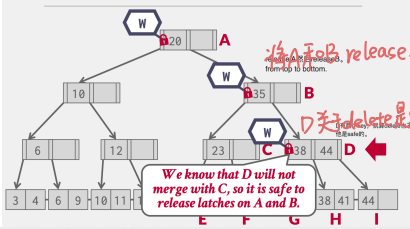
LATCH CRABBING/COUPLING

find: read latch. Insert/Delete: write latch.

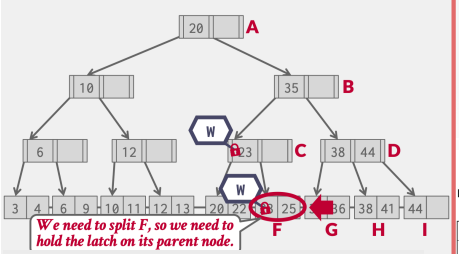
Protocol to allow multiple threads to access/modify B+Tree at the same time.
→ Get latch for parent
→ Get latch for child
→ Release latch for parent if "safe"
先获得parent的latch
获得child的latch
如果当前的node是safe的
就release他所有的parent的latch

A **safe node** is one that will not split or merge when updated.
→ Not full (on insertion)
→ More than half-full (on deletion)
safe是对于不同的操作是safe
对于insert：不会split
对于delete：不会merge with sibling

EXAMPLE #2 - DELETE 38



EXAMPLE #4 - INSERT 25



BETTER LATCHING ALGORITHM

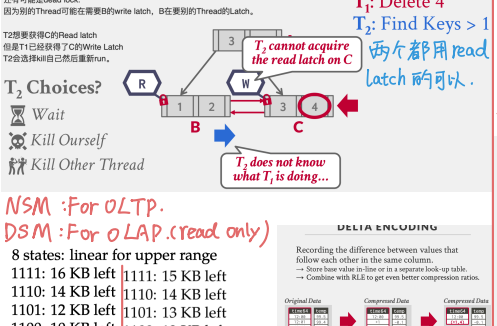
Better latch coupling. (乐观，假设只修改leaf).

Search: Same as before.

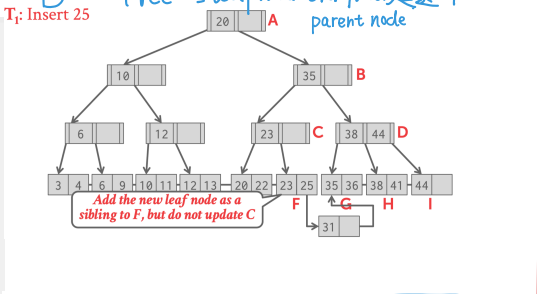
Insert/Delete: 使用Read latch，获得一个latch，release上一个latch。（就像find一样）
→ Set latches as if for search, get to leaf, and set W latch on leaf.
→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.
其实就是假设，只需要修改Leaf node。

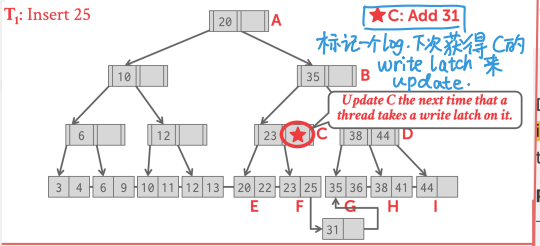
This approach optimistically assumes that only leaf node will be modified; if not, R latches set on the first pass to leaf are wasteful.

LEAF NODE SCAN EXAMPLE #3



B^link Tree：当leaf node overflow延迟update parent node





Versioned Latch Coupling

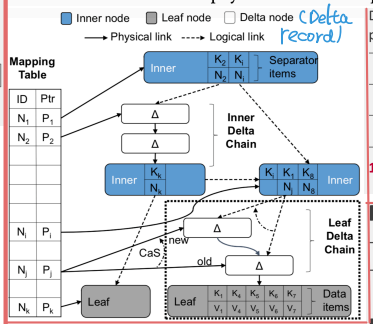
另一种通过 Version 的 Latch 方式.

- Optimistic coupling scheme where writers are **not** blocked on readers.
- Provides the benefits of optimistic coupling without wasting too much work.
- Every latch has a **version counter**.
- Writers traverse down the tree like a reader
  - Acquire latch in target node to block other writers.
  - Increment version counter before releasing latch.
  - Writer thread increments version counter and acquires latch in a single **compare-and-swap** instruction.
- Readers do not acquire latches.
- Readers traverse down the tree optimistically.
- Detect concurrent modifications by checking version counter.
- If version does not match, need to restart operation.
- May lead to **unnecessary aborts** if the node modification does not actually affect the reader thread.
- Rely on epoch-based **garbage collector** of **old nodes** to ensure node pointers are valid.

Bw-Tree.

- Latch-free B+Tree index built for the Microsoft Hekaton project.
- **Key Idea 1: Delta Updates**
  - No in-place updates.
  - Reduces cache invalidation.
- **Key Idea 2: Mapping Table**
  - Allows for CaS of physical locations of pages.

log struct storage.
compare and swap (CAS)
garbage collection.



Sorting :

① 如果 in memory 能 sort：使用 Quicksort.

② **TOP-N HEAP SORT**

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.
ORDER BY 且有限制（LIMIT）只需要遍历一次，使用priority queue
就如果order by还有LIMIT。是使用heapsort的很好场景。只需要一个priority queue记录在前面的元素就行。

Ideal scenario for heapsort: if the top-N elements fit in memory.
→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```
With ties的意思是，可以相等

Original Data: 3 4 6 2 9 1 4 4 8
Skip! Skip and done!
Sorted Heap: 4 4 4 3 2 1
Output

key-value 储存方式：
early materialization：存真的 Tuple 内容.
late materialization：存 record id. 延迟获得 data (对于 column store 很棒)

EXTERNAL MERGE SORT

Divide-and-conquer algorithm that splits data into separate runs, sorts them individually, and then combines them into longer sorted runs.

Phase #1 – Sorting 在memory中将data sort，再将sort的data写进disk
→ Sort chunks of data that fit in memory and then write back the sorted chunks to a file on disk.

#2 – Merging 将sorted的runs合并在一起
→ Combine sorted runs into larger chunks.

Simplified 2-way External merge sort.

Pass #0
→ Read one page of the table into memory 读一个page 排序好 写回disk
→ Sort page into a "run" and write it back to disk 将pages 变成排序好的run
→ Repeat until the whole table has been sorted into runs
无论有多少个pages需要sort，这需要有三个buffer pool有三个pages就可以做这个sort算法。

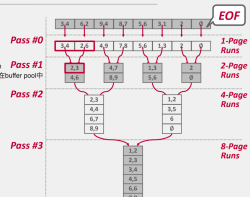
Pass #1,2,3,... 将两个runs变成一个run
→ Recursively merge pairs of runs into runs twice as long 将run变成两倍大小的runs
→ Need at least 3 buffer pages (2 for input, 1 for output)
至少需要3个pages。两个input page。一个output page。
注意，一个run不是一下子生成的。
如果只有三个pages的buffer pool，
就生成一个page，写进disk一个page，最终将两倍大小的run写回disk。

SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
= 1 + ⌈ log₂ N ⌉

Total I/O cost
= 2N · (# of passes)
2N 因为需要先从disk中读进来排序完再从memory中写进disk



GENERAL EXTERNAL MERGE SORT

如果 N=108, B=5.
∴ ⌈108/5⌉ = 22 个 run.
⌈22/4⌉ = 6 #Passes: 4.
⌈6/4⌉ = 2 8次 IO.

Pass #0
→ Use **B** buffer pages 使用B个buffer pages，B个buffer pages形成一个run
→ Produce **⌈ N / B ⌉** sorted runs of size **B** 总共有N个pages，就生成N/B个runs

Pass #1,2,3,... 使用K-way merge和merge这么多runs
→ Merge **B-1** runs (i.e., K-way merge) 将B-1个pages merge，1个page当作output。

Number of passes = **1 + ⌈ log_{B-1} ⌈ N / B ⌉ ⌉**

Total I/O Cost = **2N · (# of passes)**

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: **N=108, B=5**
→ **Pass #0:** **⌈ N / B ⌉** = ⌈108 / 5⌉ = 22 sorted runs of 5 pages each (last run is only 3 pages).
→ **Pass #1:** **⌈ N' / B-1 ⌉** = ⌈22 / 4⌉ = 6 sorted runs of 20 pages each (last run is only 8 pages).
→ **Pass #2:** **⌈ N'' / B-1 ⌉** = ⌈6 / 4⌉ = 2 sorted runs, first one has 80 pages and second one has 28 pages.
→ **Pass #3:** Sorted file of 108 pages.
1+⌈ log_{B-1} ⌈N / B⌉ ⌉ = 1+⌈log₄ 22⌉ = 1+⌈2.229...⌉ = 4 passes

Phase #1 – Partition 将tuples分类（不管hash的collision）变成buckets，写进disk
→ Divide tuples into buckets based on hash key
→ Write them out to disk when they get full

Phase #2 –ReHash 对于分类了的buckets做hash，建立真的hash table
→ Build in-memory hash table for each partition and compute the aggregation

如果要 sort 的是 B+tree 的 Index.
① clustered index：用 B+tree sequential scan 即可！
② unclustered index：是 random access. 不好.
Aggregation：如果有 Order BY：用 Sorting Aggregation.
如果不用 order. 只有 Group by，Distinct，使用 hashing (Always better).
For hash：如果 Fit in memory，直接建立 hash table.
如果 memory 空间不够，用 external hashing aggregate.

Approach #1: Code Specialization
→ Instead of providing a comparison function as a pointer to sorting algorithm, create a hardcoded version of sort that is specific to a key type.
避免通过function pointer找比较函数的overhead。
将comparison函数（对于多种key）直接写在sort函数里面。
对于不同类型的data，hard code比较函数
因为string的比较慢，通过pointer找string比较慢。
比较string的prefix的binary representation。
如果prefix一样，再进行string的比较。

Approach #2: Suffix Truncation
→ First compare a binary prefix of long **VARCHAR** keys instead of slower string comparison. Fallback to slower version if prefixes are equal.

Phase 1：

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

通过hash function1将相同的key hash到一个partition中（直接放入，不考虑重复）（mod partitions的个数）
也就是简单分类的作用，（不用考虑hash中collision之类的）
不同partition的keys肯定不同，同一个partition里也有很多keys

Filter → Remove Columns → B-1 partitions

可以将hash之后的值mod（partition的数量），将一个page的所有key hash进入partition

Phase 2：

```
SELECT cid, AVG(s.gpa)
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
GROUP BY cid
```

Running Totals
AVG(col) → (COUNT, SUM)
MIN(col) → (MIN)
MAX(col) → (MAX)
SUM(col) → (SUM)
COUNT(col) → (COUNT)
AVG需要多一个count计算

对分类后的bucket一个个 hash，建立 hash table.
Value 由 aggregate 决定

Phase #1 Buckets → Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Final Result

cid	AVG(gpa)
15-445	3.66
15-826	3.33
15-721	2.89

这里的value储存了（个数，数量）

NSM：For OLTP.
DSM：For OLAP. (read only)

DELTA ENCODING
Recording the difference between values that follow each other in the same column.
→ Store base value in-line or in a separate look-up table.
→ Combine with RLE to get even better compression ratios.

8 states: linear for upper range

1111: 16 KB left	1111: 15 KB left
1110: 14 KB left	1110: 14 KB left
1101: 12 KB left	1101: 13 KB left
1100: 10 KB left	1100: 12 KB left
1011: 8 KB left	1011: 11 KB left
1010: 6 KB left	...
1001: 4 KB left	0101: 5 KB left
1000: 2 KB left	0100: 4 KB left
0111: 1024 B left	0011: 3 KB left
....	0010: 2 KB left
0001: 16 B left	0001: 1 KB left
0000: 8 B left	0000: 0 KB left

DICTIONARY: ORDER-PRESERVING
The encoded values need to support the same collation as the original values.

RUN-LENGTH ENCODING

BITMAP ENCODING

Query Execution. 2
Process Model: (DBMS 控制一切)
1. Process per Worker. (旧少)
2. Thread per worker. (多好)
Inter-Query：处理多个 disparate queries.
Intra-Query：并行处理一个 query. (OLAP)
↳ Operator 有 parallel 版本.
I/O Parallelism：Split on multiple device. ① HW-based RAID.
Multi-disparallelism: ② SW-based. Faster.

Intra-Query Parallelism.
1. Intra-Operator (Horizontal) (OLAP)
2. Intra-Operator (Vertical)
3. Bushy.
1：多个 Fragments 同时执行.
2. Operation overlapped, pipeline. streaming system.
3. Hybrid. (以上，望真都加速)

Join:
①: Nested Loop Join.(若OLTP,或有index,用这个)
(Naive, Block, Index) Block Nested Loop Join.

```
foreach block B_R ∈ R:                  每一个out table page
  foreach block B_S ∈ S:                需要load所有的inner table page
    foreach tuple r ∈ B_R:              对于在memory中的这两个pages中的所有tuple进行match
      foreach tuple s ∈ B_S:            block就是page
        if r and s match then emit
```
这个算法只使用三个pages.

This algorithm performs fewer disk accesses.
→ For every block in **R**, it scans **S** once. (小的table放在outer loop)

Cost: $M + (M \cdot N)$ 只使用了三个pages in the buffer pool

将outer的一个page load 进来，然后将inner的page一个一个load进来。
所以每个outer page进来一次。每一个inner page进来M次。

The smaller table should be the outer table.

We determine size based on the number of pages, not the number of tuples. page个数少的作为 outer table

使用B个Buffer pages:(使outer table尽可能存在memory中)

If we have ***B*** buffers available: B-2: outer, 1 inner, 1 output
→ Use ***B*-2** buffers for each block of the outer table.
→ Use one buffer for the inner table, one buffer for output.

这一种，outer table利用了B-2个page，inner table利用了1个page。
对于每B-2个outer page，load所有inner table的pages

```
foreach B - 2 pages p_R ∈ R:        load outer loop中B-2个page
  foreach page p_S ∈ S:             load inner loop中一个page
    foreach tuple r ∈ B - 2 pages:  将这些page中所有的tuple
      foreach tuple s ∈ p_S:        结果写进剩下的一个page
        if r and s match then emit  写满了存进disk
```

This algorithm uses ***B*-2** buffers for scanning **R**.

Cost: $M + (\lceil M / (B-2) \rceil \cdot N)$ 相当于之前说的，对于每B-2个outer page load一次inner page

对于outer loop中的一个page，需要load一次。
对于inner node中的一个page，需要轮换 outer loop的次数（也就是outer loop来几轮）（一轮完成B-2个out page）

If the outer relation fits in memory ($M < B-2$):
→ **Cost:** $M + N$ = 1000 + 500 = **1500 I/Os** 如果buffer够大，每个page各一次就行。也就是一轮将outer page做完了。
→ At 0.1ms per I/O, Total time ≈ 0.15 seconds

根本的提升就是，减少每一个inner page需要load进入memory的轮次。

If we have ***B***=102 buffer pages:
→ **Cost:** $M + (\lceil M / (B-2) \rceil \cdot N)$ = 1000 + 10·500 = **6000 I/Os**
→ Or can switch inner/outer relations, giving us cost:
500 + 5·1000 = **5500 I/Os**

Index Nested Loop Join (不一个个比较)

```
foreach tuple r ∈ R:                 通过index找match的tuple.
  foreach tuple s ∈ Index(r_i = s_j):
    if r and s match then emit
```

Assume the cost of each index probe is some constant ***C*** per tuple. 每个pageload一次

Cost: $M + (m \cdot C)$ 对于每个tuple，寻找符合inner 中的tuple C是精准match的时间constant

Sort merge Join.(当table已经sort的时候，或需要sort的output,用这个.)

Phase #1: Sort
→ Sort both tables on the join key(s).
→ You can use any appropriate sort algorithm
→ These phases are distinct from the sort/merge phases of an external merge sort, from the previous class

先排序，排序好，对于排序好的两个table probing（可能会有trackback）（很好理解）

Phase #2: Merge
→ Step through the two sorted tables with cursors and emit matching tuples. MN是page个数.
→ May need to backtrack depending on the join type.

Sort Cost (R): $2M \cdot (1 + \lceil \log_{B-1} \lceil M / B \rceil \rceil)$ sort outer table
Sort Cost (S): $2N \cdot (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$ sort inner table
Merge Cost: $(M + N)$ 假设没有backtrack，每个遍历一遍
Total Cost: Sort + Merge.

Access Method: 3种:
① Sequential Scan.(就是从BufferPool获得)
Optimization: Data skipping.

Approach #1: Approximate Queries (Lossy)
→ Execute queries on a sampled subset of the entire table to produce approximate results. 从一部分数据produce result 有误差

Approach #2: Zone Maps (Lossless) 提前计算aggregate.
→ Pre-compute columnar aggregations per page that allow the DBMS to check whether queries need to access it.
→ Trade-off between page size vs. filter efficacy. page size大，比较难filter; page size小，一次filter的size小

② Index Scan. ③ Multiple Index Scan.
The DBMS picks an index to find the tuples that the query needs. 使用多个index 然后最后将结果 intersect 或 Union.

Simple hash Join Algorithm (Always Better)

Phase #1: Build 先将outer table的tuples的join attributes进行哈希
→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.
→ We can use any hash table that we discussed before but in practice linear probing works the best.

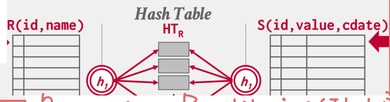
第一步，本质是将outer table的tuples分成几个partition

Phase #2: Probe (R有out table需要fit in memory)
→ Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

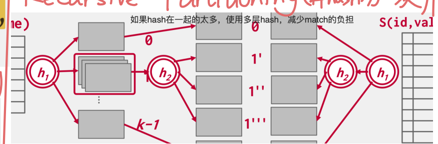
将inner table的tuples和hash，与partition内的outer tuple匹配。这样比起之前一个个匹配，减少了匹配的次数。
Bloom Filter中有Inner table 再去partition 中 match.

同时，outer table分类时建立bloom filter, inner table先check bloom filter，如果这个key可能在里面，再去partition中match

```
build hash table HT_R for R
foreach tuple s ∈ S        对于R的tuples进行hashing
  output, if h_1(s) ∈ HT_R
```
对于s的tuples进行hashing 然后比较



Recursive Partitioning (再hash分一次)



Cost: (不用 recursive hash)
If we do not need recursive partitioning:
→ Cost: $3(M + N)$ 总共就是3(M+N)

Partition phase: hash Join的效率很高.
→ Read+write both tables
→ $2(M+N)$ I/Os

Probe phase:
→ Read both tables (in total, one partition at a time)
→ $M+N$ I/Os

优化: 将hot partition一直放在Memory中.

Join的总结:

Algorithm	IO Cost	Example
Naïve Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (\lceil M / (B-2) \rceil \cdot N)$	0.55 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Variable
Sort-Merge Join	$M + N +$ (sort cost)	0.75 seconds
Hash Join	$3 \cdot (M + N)$	0.45 seconds

query execution.
Process Model: 3种.

① Iterator Model (从上往下调用Next()) 获得single tuple.

Each query plan operator implements a **Next()** function.
→ On each invocation, the operator returns either a single tuple or a eof marker if there are no more tuples. return一个tuple
→ The operator implements a loop that calls **Next()** on its children to retrieve their tuples and then process them. 这个children应该指的是些pipeline下面的operator 也就是operator调用别的operator的next，获得tuple

Each operator implementation also has **Open()** and **Close()** functions.
Analogous to constructors and destructors, but for operators.

缺点: 太多function call. 1次处理1个tuple.
对于一般的OLTP和OLAP,这个还好.

②: Materialization model. (1次获得所有tuples)

Each operator processes its input all at once and then emits its output all at once. 将所有数据全部处理？将结果一次输出 Operator at a time.
→ The operator "materializes" its output as a single result.
→ The DBMS can push down hints (e.g., **LIMIT**) to avoid scanning too many tuples. 如果有LIMIT，告诉child LIMIT 避免无效计算
→ Can send either a materialized row or a single column.

调用operator一次获得所有Tuples.
The output can be either whole tuples (NSM) or subsets of columns (DSM). 可能到disk.
对于OLTP,这个好.对于OLAP,数据量太大.

③ Vectorization Model. (能用SIMD来处理tuples)
获得batch tuples. Ideal for OLAP.

Like the Iterator Model where each operator implements a **Next()** function, but ...

Each operator emits a batch of tuples instead of a single tuple. SIMD: single Instruction Multiple Data.
→ The operator's internal loop processes multiple tuples at a time.
→ The size of the batch can vary based on hardware or query properties.

Modification Queries. (避免Halloween Problem)

UPDATE/DELETE: 记录之前的tuple，保证同一个tuple多次
→ Child operators pass Record IDs for target tuples.
→ Must keep track of previously seen tuples.

Anomaly where an update operation changes the physical location of a tuple, which causes a scan operator to visit the tuple multiple times.
→ Can occur on clustered tables or index scans.

通过Metadata避免scan operator一个query改一个tuple多次.

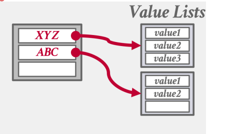
Expression Evaluation. (where clause)
① Expression Tree. (+,−, AND, OR) (slow)

② JIT (Just In Time Compilation) (直接逻辑比较)

hash: Time: Average O(1), Worst O(n) No cryptographic
Static hash: ① Linear Probe hashing. (两种)
hash到slot,冲突后,与3往后找,删除3放Tombstone.

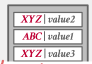
key不是unique，一个key对应lists中的value。

Choice #1: Separate Linked List
→ Store values in separate storage area for each key.
→ Value lists can overflow to multiple pages if the number of duplicates is large.



Non unique keys

Choice #2: Redundant Keys
→ Store duplicate keys entries together in the hash table.
→ This is what most systems do.



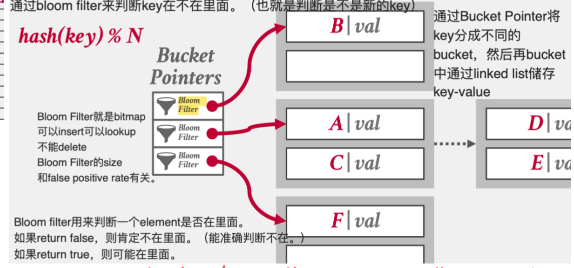
② Cuckoo Hashing 多个hash table.找空位存 找插覆盖.

Dynamic hash table: 3种
① Chained hashing
用Bucket Pointer,将key hash进不同bucket.
找的时候使用bloom filter, 再遍历.

Maintain a linked list of buckets for each slot in the hash table. 使用bucket链表，来动态扩张hash table
jave 中的hashmap

Resolve collisions by placing all elements with the same hash key into the same bucket.
→ To determine whether an element is present, hash to its bucket and scan for it.
→ Insertions and deletions are generalizations of lookups.

当很多key都map到一个bucket的时候，sequential scan来寻找key在不在里面，比较慢。
通过bloom filter来判断key在不在里面。（也就是判断是不是新的key）



通过Bucket Pointer将key分成不同的bucket，然后再bucket中通过linked list储存key-value

Bloom Filter就是bitmap
可以insert可以lookup
不能delete
Bloom Filter的size
和false positive rate有关。

Bloom filter用来判断一个element是否在里面。
如果return false，则肯定不在里面。（能准确判断不在。）
如果return true，则可能在里面。

② Extendible hashing. (放不进,Split这个,增加Examine Bits Rehash 当前Bucket.)

避免了chained hashing一个bucket会变的很大很大，导致sequential scan开销大。通过split bucket来dynamic。

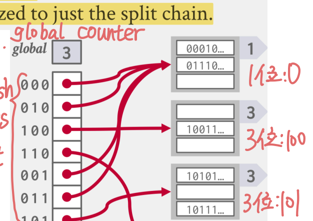
Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever. Extendible hashing 将满了的buckets进行split 而不是像chained-hashing一直在后面加入。

Multiple slot locations can point to the same bucket chain. 共享相同的前缀。多个slot可以指向一个bucket

将Split的bucket里面的entries进行reshuffle也就是重新hash。增加examine bits。

Reshuffle bucket entries on split and increase the number of bits to examine. 只在split chain中数据交换。
→ Data movement is localized to just the split chain.

当右边Bucket满了,insert不进去.
增大global counter. (加一位)
将满的Bucket Split.
将里面的entries rehash.
用Bits来match对应的Bucket



③ Linear Hashing
Split, split pointer指向的hash function 变化.

每一次overflow，Split pointer指向的blocks(bucket)就会split，然后会多出来一个slot，然后将split pointer下移或者回到最上面。

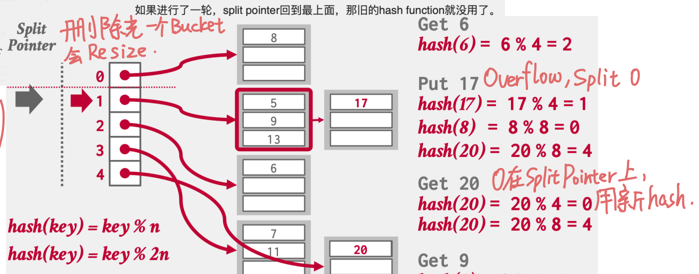
Overflow时,将split Pointer指向Bucket Split. 然后rehash. (新hash) Split Pointer下移.

The hash table maintains a pointer that tracks the next bucket to split.
→ When any bucket overflows, split the bucket at the pointer location. 使用hash function，如果位置在，split pointer指向的位置下面。就找到了

Use multiple hashes to find the right bucket for a given key. 如果位置在split pointer上面的位置，由于已经做过split操作，要重新用新的hash function寻找。

找的时候,先用旧hash. 如果在split Pointer上面,用新hash找. (因为rehash过)

如果进行了一轮，split pointer回到最上面，那旧的hash function就没用了。



Hash table 查询需知道 entire key. (不如B+ Tree)

major_program	semester	minor	matriculation_no	name_start	name_end	email_start	email_end	padding	name	email

type	INT	SMALLINT	INT	BIGINT	PTR	PTR	PTR	PTR	-	varchar	varchar
size	4B	2B	4B+1bit	8B	8B	8B	8B	8B	14B - 1bit	-	-

Null bit
64 byte cache line

[20 points] Draw an optimal record layout for the following schema:

```
CREATE TABLE students (
  name VARCHAR(128) NOT NULL,
  email VARCHAR(128) NOT NULL,
  major_program INT NOT NULL,
  semester SMALLINT NOT NULL,
  minor INT,
  matriculation_no BIGINT NOT NULL PRIMARY KEY
```

B+ tree search. SIMD, linear, binary, Interpolation.
Any Attribute can search.

FIFO: 先进先出.
LRU: 最长时间没用的先evict.
LFU: 频率最低先出
LRU-K: K次最久的先evict
2Q: 1个FIFO, 1个LRU. 2次访问,放进LRU.