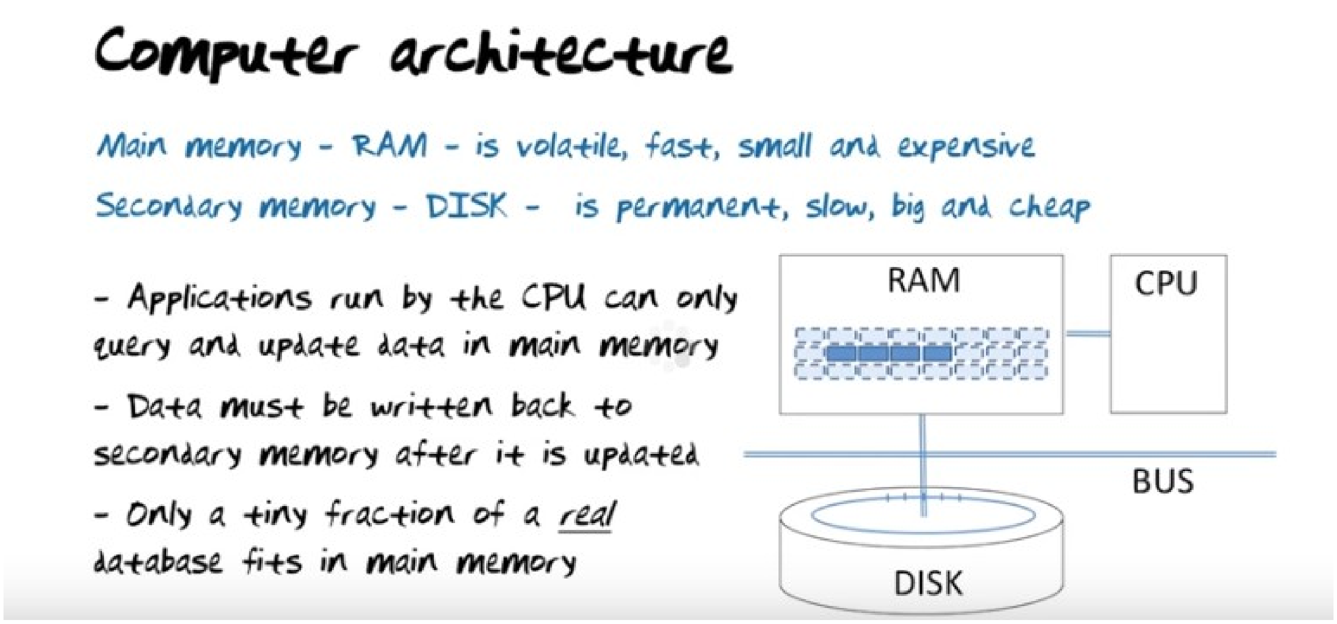
Efficiency, indexing, physical database design



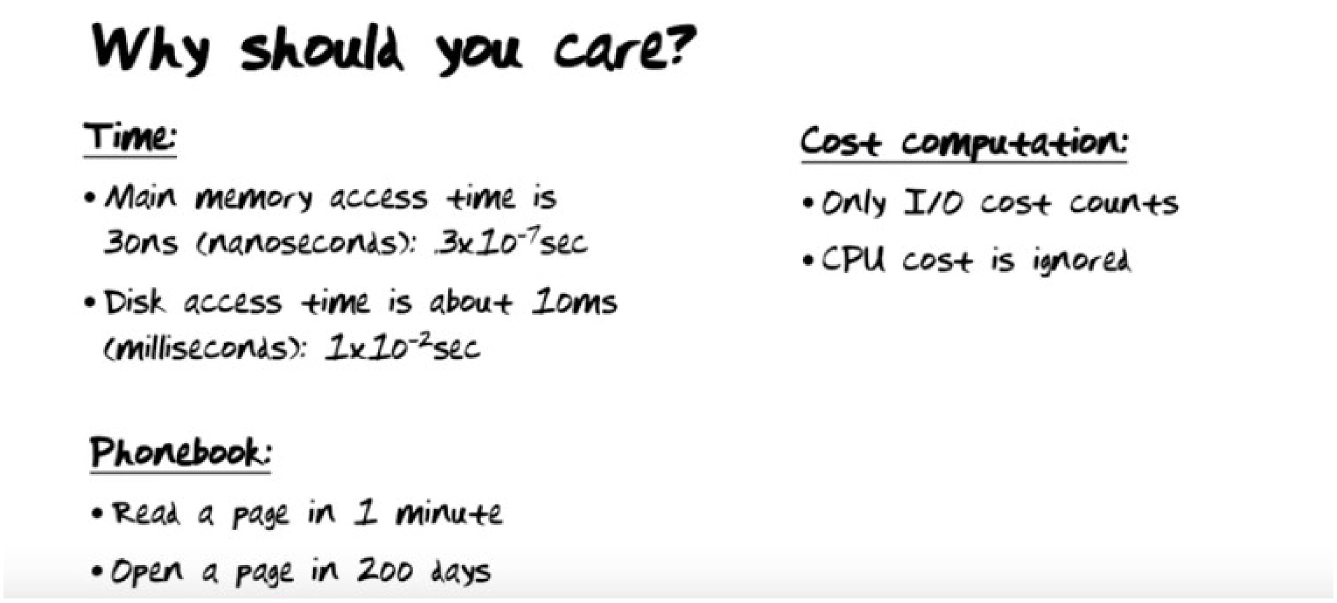
Computer architecture

Main memory - RAM - is volatile, fast, small and expensive

Secondary memory - DISK - is permanent, slow, big and cheap

- Applications run by the CPU can only query and update data in main memory
- Data must be written back to secondary memory after it is updated
- Only a tiny fraction of a *real* database fits in main memory

The above figure: the bus is there to allow us to transport data from the disk to main memory and to write it back again if it has been changed. … Our assumption going forward will be that the vast majority of the database will be stored on disk and only a tiny fraction will be in main memory at any given point in time.



Why should you care?

Time:

- Main memory access time is 30ns (nanoseconds): $.3x10^{-7}$sec
- Disk access time is about 10ms (milliseconds): $1x10^{-2}$sec

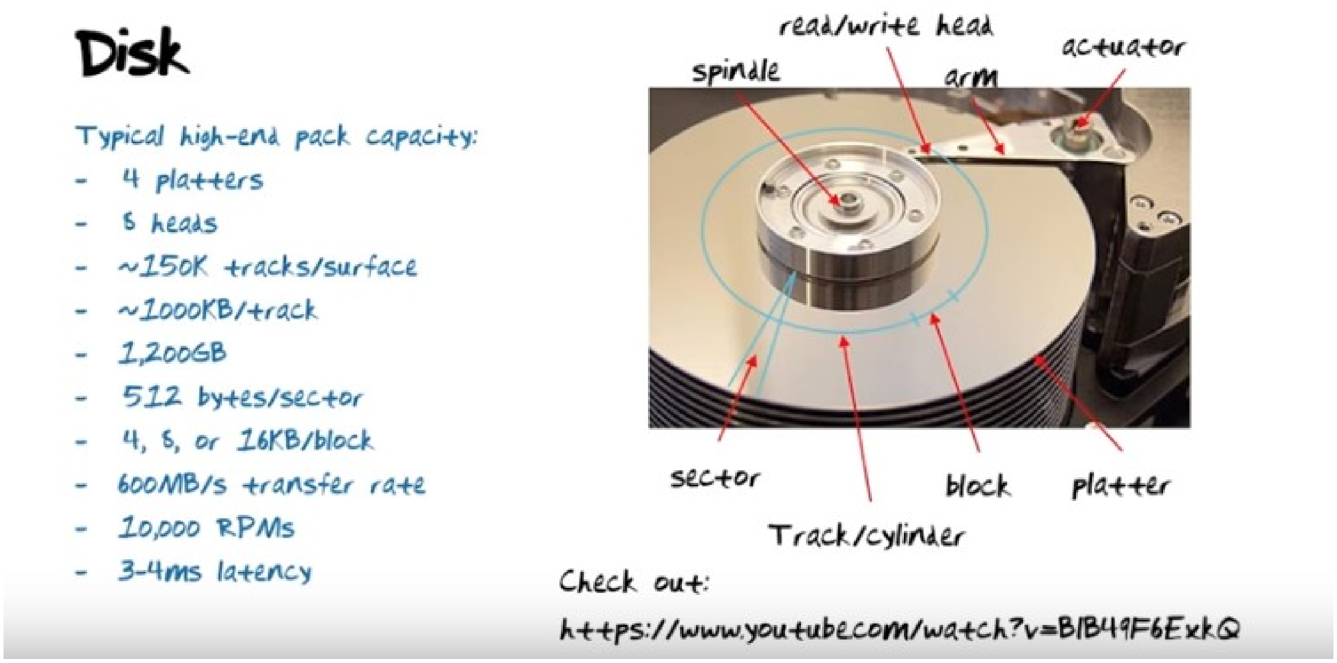
Phonebook:

- Read a page in 1 minute
- Open a page in 200 days

Cost computation:

- Only I/O cost counts
- CPU cost is ignored

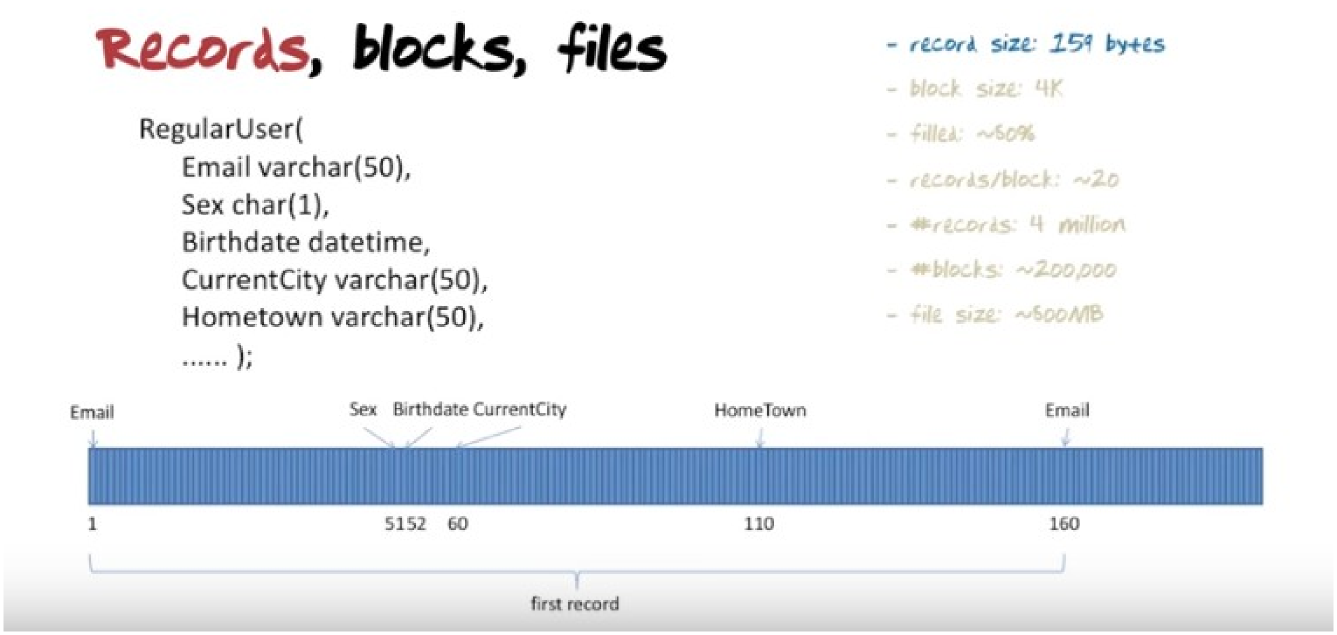
上圖中的 I/O cost 即 disk access time.


Disk

Typical high-end pack capacity:
- 4 platters
- 8 heads
- ~150K tracks/surface
- ~1000KB/track
- 1,200GB
- 512 bytes/sector
- 4, 8, or 16KB/block
- 600MB/s transfer rate
- 10,000 RPMs
- 3-4ms latency

Check out:
https://www.youtube.com/watch?v=BIB49F6ExkQ

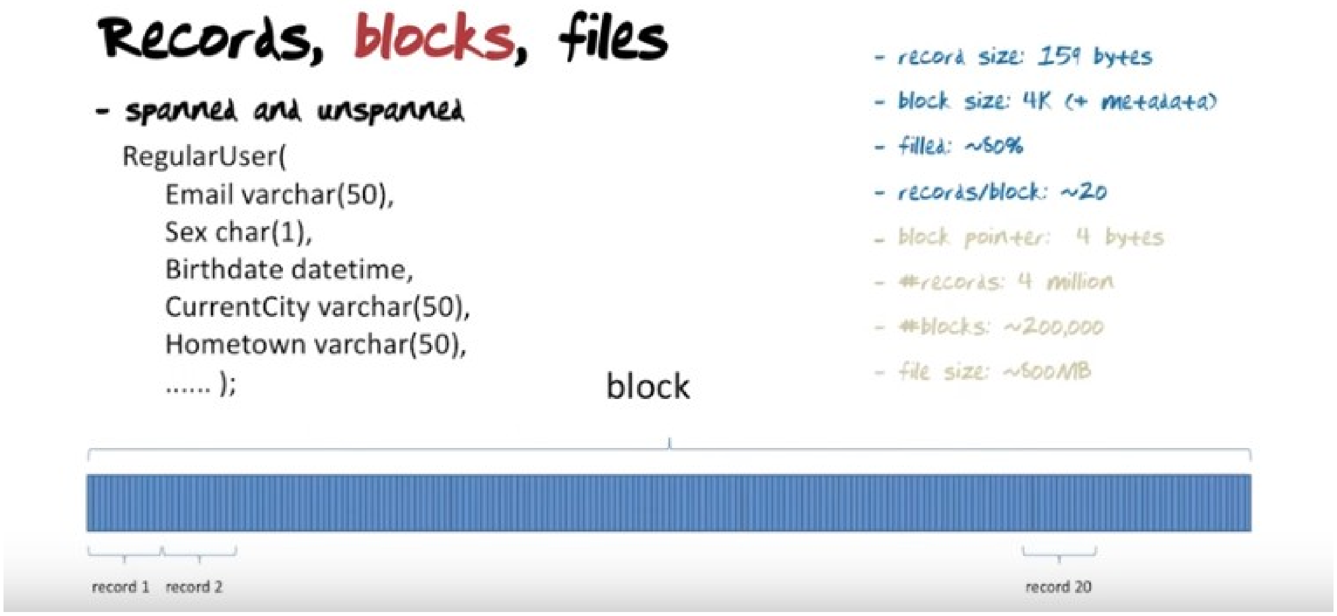
注意上圖的 block, 下下圖中會提到.


Records, blocks, files

```
RegularUser(
    Email varchar(50),
    Sex char(1),
    Birthdate datetime,
    CurrentCity varchar(50),
    Hometown varchar(50),
    ...... );
```

- record size: 159 bytes
- block size: 4K
- filled: ~80%
- records/block: ~20
- #records: 4 million
- #blocks: ~200,000
- file size: ~800MB

The above figure: 以上的 RegularUser(應該是 RegularUser 這個表中的一行)就是一個 record. Email 大小為 50 bytes, Sex 為 1 byte, Birthdate 為 8 bytes … (1 byte = 8 bit). Recrod size = 50+1+8+50+50 = 159.



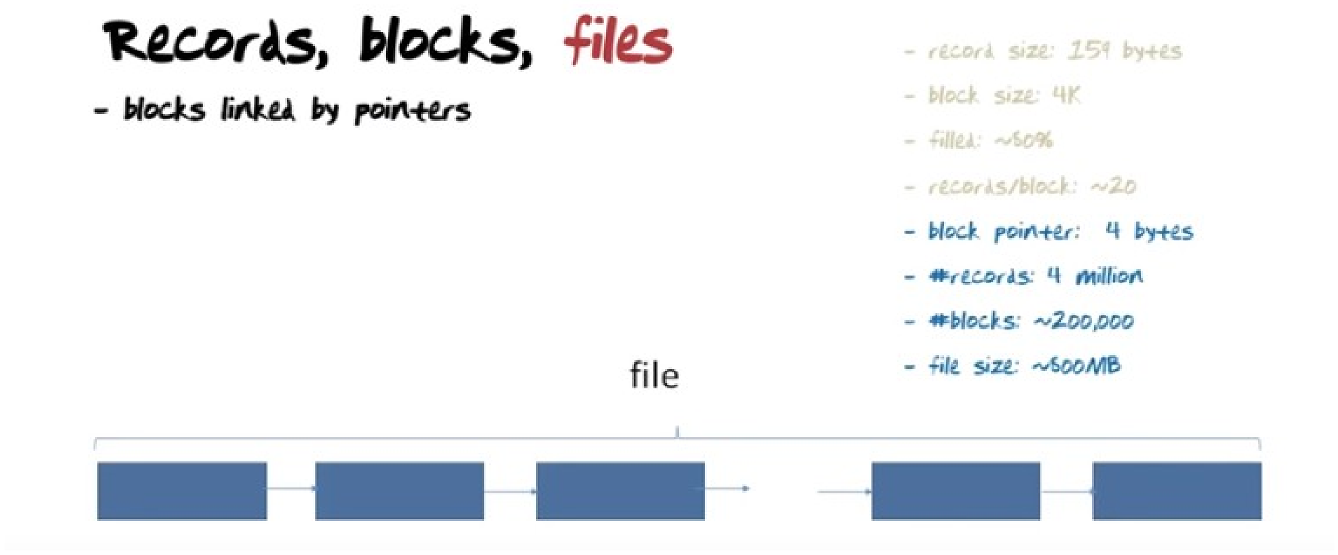
Records, blocks, files

- spanned and unspanned

```
RegularUser(
    Email varchar(50),
    Sex char(1),
    Birthdate datetime,
    CurrentCity varchar(50),
    Hometown varchar(50),
    ...... );
```

- record size: 159 bytes
- block size: 4K (+ metadata)
- filled: ~80%
- records/block: ~20
- block pointer: 4 bytes
- #records: 4 million
- #blocks: ~200,000
- file size: ~800MB

上圖中的 filled: ~80%: the reason why you wanna not fill a block completely with data is that if there's some additional record, you need to insert and there's no room for it, then you'll get an overflow, then it takes extra work.... You will fill about 20 records per bolck (4000*80% / 159=20.1), leaving a little bit of space ohhmm. Here is record number 20, you can see you might be able to fill some additional recrods in here(注意算的時候也經乘了 80%, 所以這裡是加入了 metadata 後還剩下的空間). There are two ways of thinking about what happens to the piece of the block that might not fill a complete record at the end if it comes to that. Do we break off the record we are trying to put in? Put the first part here and the second part on the next block, that would be called a spanned(跨距) representation. If we just ignore the tiny amount of waste in the end, that's called an unspanned representation. In most cases when multiple records, they will fit on a block, most database systems will run with unspanned representation. Simply to avoid the processing that's necessary to break off records. Obviously, if you have record sizes that are larger than block sizes, you don't have a choice. You have to run with spanned representation.



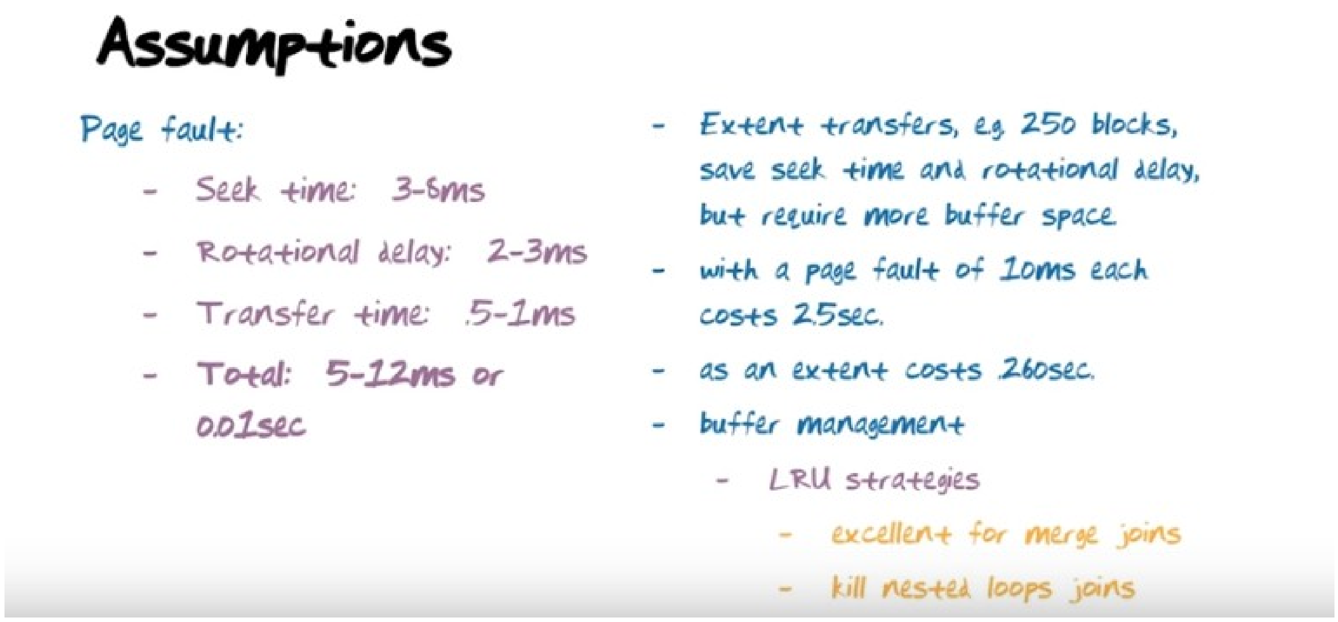
Records, blocks, files

- blocks linked by pointers

- record size: 159 bytes
- block size: 4K
- filled: ~80%
- records/block: ~20
- block pointer: 4 bytes
- #records: 4 million
- #blocks: ~200,000
- file size: ~800MB

上圖中每一個方框即一個 block. The blocks may not sit right next to each other on disk, but when we have seen one block, we can use the next pointer to get to the place on disk where the second block is etc. 4 million / 20 = 200 000 blocks. 200 000 * 4k = 200 000 * 4e3 = 8e8 = 800 MB. 最後的 file size 寫的是~800 MB, 而不是~500 MB ← not a big file, not a small one either.

MO: database 中的數据都是存在 disk 中的一些區域中的, 這些區域合起來, 稱為一個 file. 注意沒必要將這些 file 弄成一些可視的文件形式(如 txt 或 Excel 文件). 反正對於 database, 數据就存在 disk 中, 可以取來用就可以了, 沒有可視的文件形式.



Assumptions

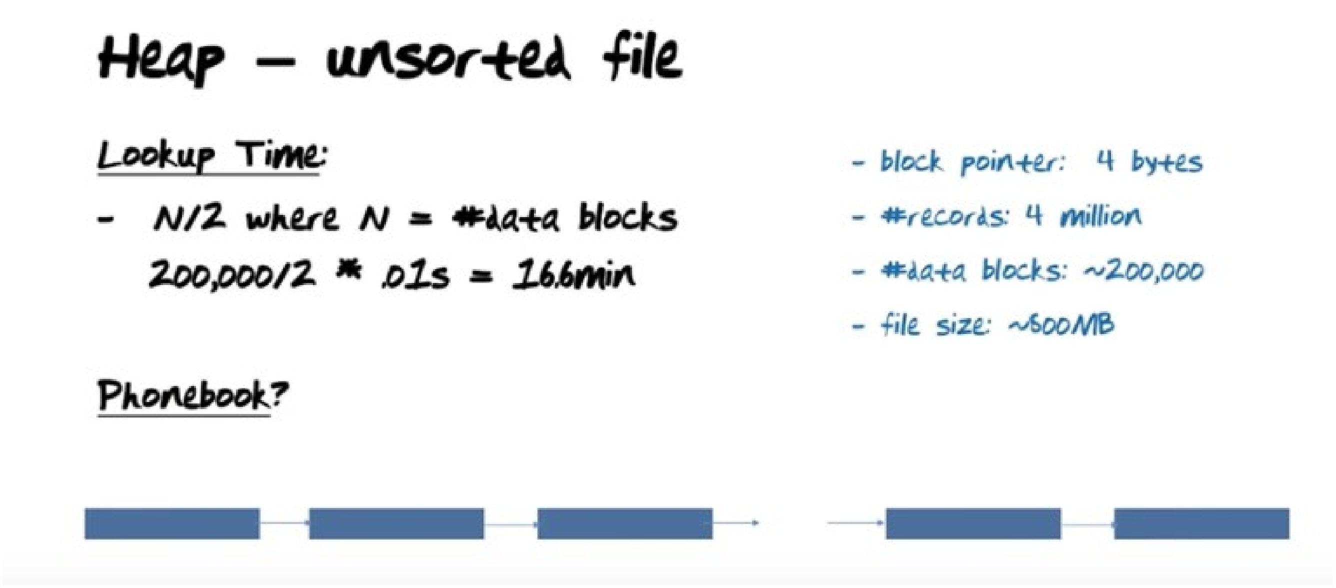
Page fault:

- Seek time: 3-6ms
- Rotational delay: 2-3ms
- Transfer time: .5-1ms
- Total: 5-12ms or 0.01sec

- Extent transfers, e.g. 250 blocks, save seek time and rotational delay, but require more buffer space
- with a page fault of 10ms each costs 2.5sec.
- as an extent costs .260sec.
- buffer management
  - LRU strategies
    - excellent for merge joins
    - kill nested loops joins

The above figure: To be able to compute the amount of time it takes to transport data from disk to main memory and back again, let's just make some assumptions. 上圖中右邊說 Extet transfer 250 blocks 意思是一次 transfer 多些, 就不用重復花 seek time 和 rotational delay 了. 注意左邊是 Page fault. 而右邊第二點也是 page fault, 所以是 0.01 sec * 250 = 2.5 sec. 右邊第三點是用 extent transfer, 由左邊知, transfer time

約為 1 ms, 所以是 1e-3 sec * 250 = 0.250 sec, 約為 0.260 sec. 右邊第四點中的 LRU 即 Least Recently Used, Leetcode 中有關於 LRU 的題. Kill nested loops joins 意思是, if you are doing nested joins, then LRU stategies would kill that computation, and other strategies like MRU or Most Recently Used might work well for nested loop joins.



Heap – unsorted file

Lookup Time:

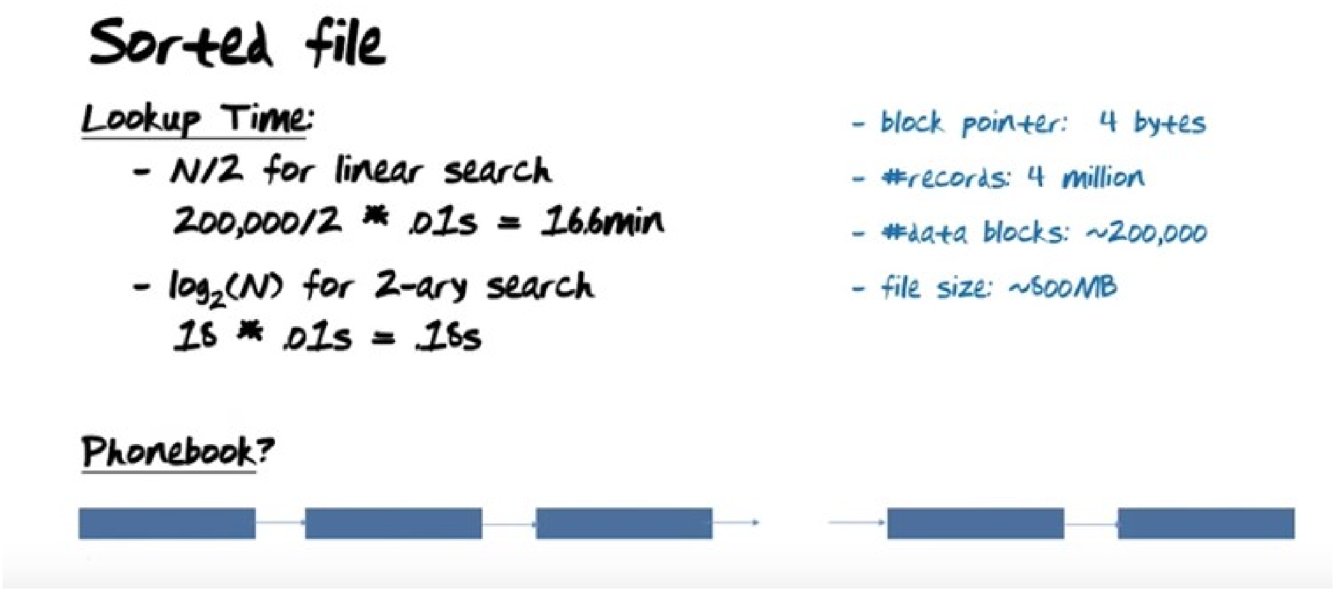
- N/2 where N = #data blocks
  200,000/2 * .01s = 16.6min

Phonebook?

- block pointer: 4 bytes
- #records: 4 million
- #data blocks: ~200,000
- file size: ~800MB

The above figure: a heap is defined as a file of data that is not sorted. (CLRS p172: The heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels exept possibly the lowest, which is filled from the left up to a point. 另有 max heap 和 min heap, 跟此處無關). So the records sit on a bunch of blocks, and there is no sorting among the records, any record we are looking for in the hoou file could sit on any block. 注意, again, 上圖右邊的 file size 寫的是 800MB, 而不是 500MB. 左邊的 N/2 是因為, 我們要在 N 個 unsorted 數中找某個數, 最好的情況是此數為第一個, 最壞的情況是此數為最後一個, 所以平均的時間複雜度為 N/2. 下一行的*.01s 即*0.01s.

然後用了五個小節(沒 record 到這裡)來講 binary search, 對於刷了兩遍 Leetcode 的我來講弱爆了.



Sorted file

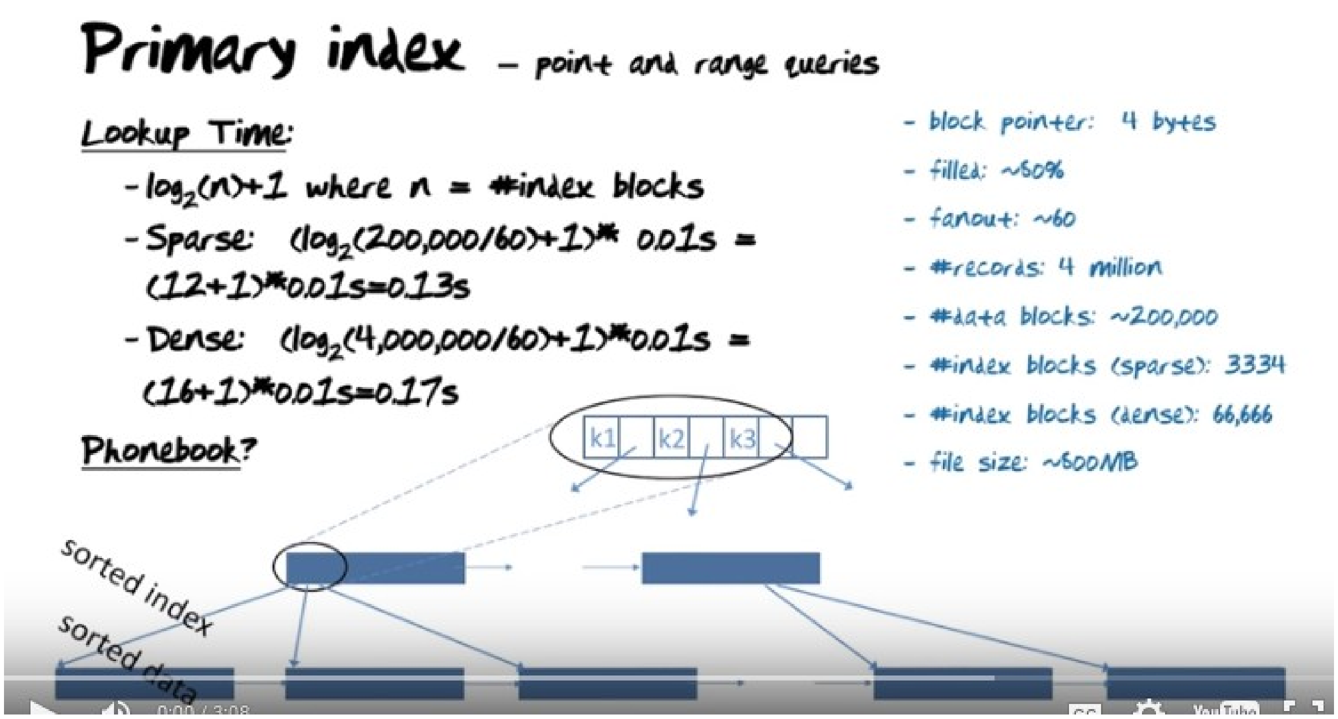
Lookup Time:

- N/2 for linear search
  200,000/2 * .01s = 16.6min
- $\log_2(N)$ for 2-ary search
  18 * .01s = .18s

Phonebook?

- block pointer: 4 bytes
- #records: 4 million
- #data blocks: ~200,000
- file size: ~800MB

The above figure: what if we were to utilize the fact that data is sorted. What if we had a little bit of metadata that would allow us to understand we are approximately the middle of the data file is.



Primary index – point and range queries

Lookup Time:

- $\log_2(n)+1$ where n = #index blocks
- Sparse: $(\log_2(200{,}000/60)+1) * 0.01s = (12+1)*0.01s = 0.13s$
- Dense: $(\log_2(4{,}000{,}000/60)+1)*0.01s = (16+1)*0.01s = 0.17s$

Phonebook?

- block pointer: 4 bytes
- filled: ~80%
- fanout: ~60
- #records: 4 million
- #data blocks: ~200,000
- #index blocks (sparse): 3334
- #index blocks (dense): 66,666
- file size: ~800MB

key 就是 SQL 中的 key, 比如一個表中, (ID, name, age, income)這樣一行的 ID 就是一個 key.

上圖中每一個方框即一個 block, 每一個 block 中有一些 record. we have data here at the bottom, and the data is sorted in some order. So the records come here with increasing or decreasing key values. To build a primary index, what we do is we take a look at the frist record (在最下面一行的第一個 block 的最左端) and pick up the key of that record and make a copy of it(指 key) in an index block (即從下往上數

的第二行的這些 block 中), together with a pointer that points back to the block on which the record with that key was the first one. We go to the second block (在最下面一行的第二個 block) here, look at the first record, pick to the key value, make a copy, insert in the index together with a pointer back to the block on which that key value was the key value of the first record, …, when that index block is full, we continue (即增加一個 index block, 這就是為甚麼 從下往上數的第二行 有多個 block). 從下往上數的第三行即一個 index block 的結構, k1 即 key value, k1 後面那格即 a pointer to the record where that was the first key value. Notice an interesting detail, since we are picking up the key values in sored order, once we build the index does not even have reason to sort that. It's born sorted. Also notice that the only element we are picking up from each record actually is the key value, so all the other CurrentCity, BirthDate, Hometown, etc, all of that is not picked up from here. What does that mean? That means that we can fit a whole lot of entries in an index page.

現在看右邊. 前面的圖中說 block size 為 4k, filled 80%, 故每個 block 中可用的為 3200 bytes. The key value (email) is 50 characters, block pointer is 4 bytes, 故共 54 bytes. 3200/54 約為 60. 即一個 index block 中有 60 個 key values and a corresponding set of pointers that come out from that. Thats called the fanout.

前面某小節中已算出 data blocks(最下面一行的 blocks)有 200 000 個, 從下往上數的第二行中的每個 index(即 key value)對應一個 data block, 即總共有 200 000 個 key values. 而一個 index block 中有 60 個 key values, 故 index block 的數量為 200 000/60=3334.

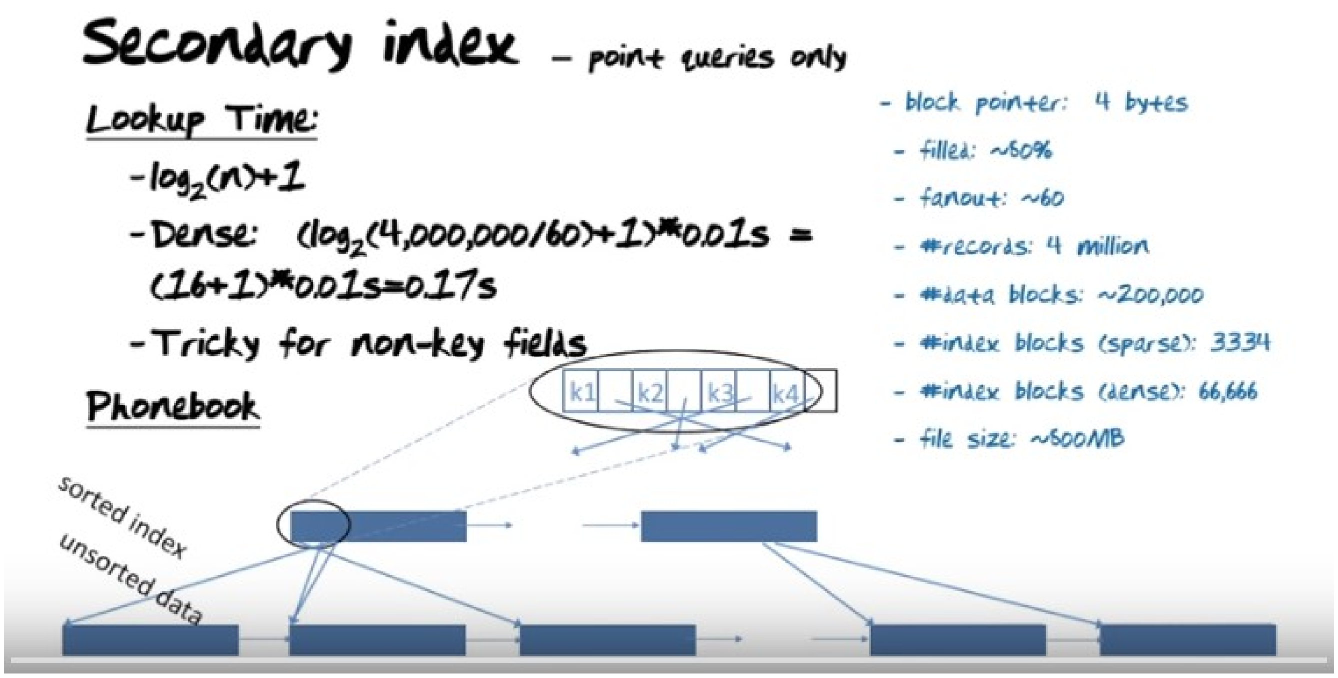
It would actually be possible to build a primary index as a dense index also. So instead of picking up the key value of the first block, we pick up every single key value, and then every single one of them will result in an insertion of the key value.... 後面太長不寫, 意思就是#index blocks(dense): 66,666 這行的意思是最下面一行的 blocks 中每一個 record 的 key value 都进入從下往上數的第二行中(66,666 怎麼得出來的, 後面馬上說). 注意#index blocks(sparse): 3334 那行中是在 最下面一行的 blocks 中 每一個 block 中只選一個 key value 进入從下往上數的第二行中. 由於前面某小節中已假定 records/block=20, 故#index blocks(dense)是#index blocks(sparse)的 20 倍, 即 3334*20, 准確的數字是 3333.3*20=66 666.

One advantage of having every single key value is some queries can now actually be asked that do not need to access the data but instead can just access the index. Such as, for example, what's the maximum key value, the minimum key value, average key value, etc, that can be determined by queriy on the index alone. Primay indices 還可以 allow you to do range queries. So once you find the first one in a range, then of course you can just follow the next values at the data level or the index level.

n = number of index blocks
N = number of data blocks

現在看左邊. so now the cost of doing a search in a sorted file is therefore log_2 (n). But once you find the correct value, then of course you have to access the data so we gonna add one. 前面已算過 200 000/60=3334 即 index block 的數量, 即 n. Notice that the 0.13s is lower than the 0.18 we used when we did binary search on the data file. So it is cheaper because we cut down the size. There's also a penalty involved there(log_2 (n)+1 中的+1) because there's an extra level to access now. 注意 dense 的時間 0.17s 也比 binary search 快.

一個例子舉得好: 以上的 index blocks(sparse)就相當於一本詞典, 我取每頁的第一個單詞(index)和它所在的頁碼(pointer), 將它們放在一起, 就形成了一個 index blocks(sparse).



Secondary index – point queries only

Lookup Time:

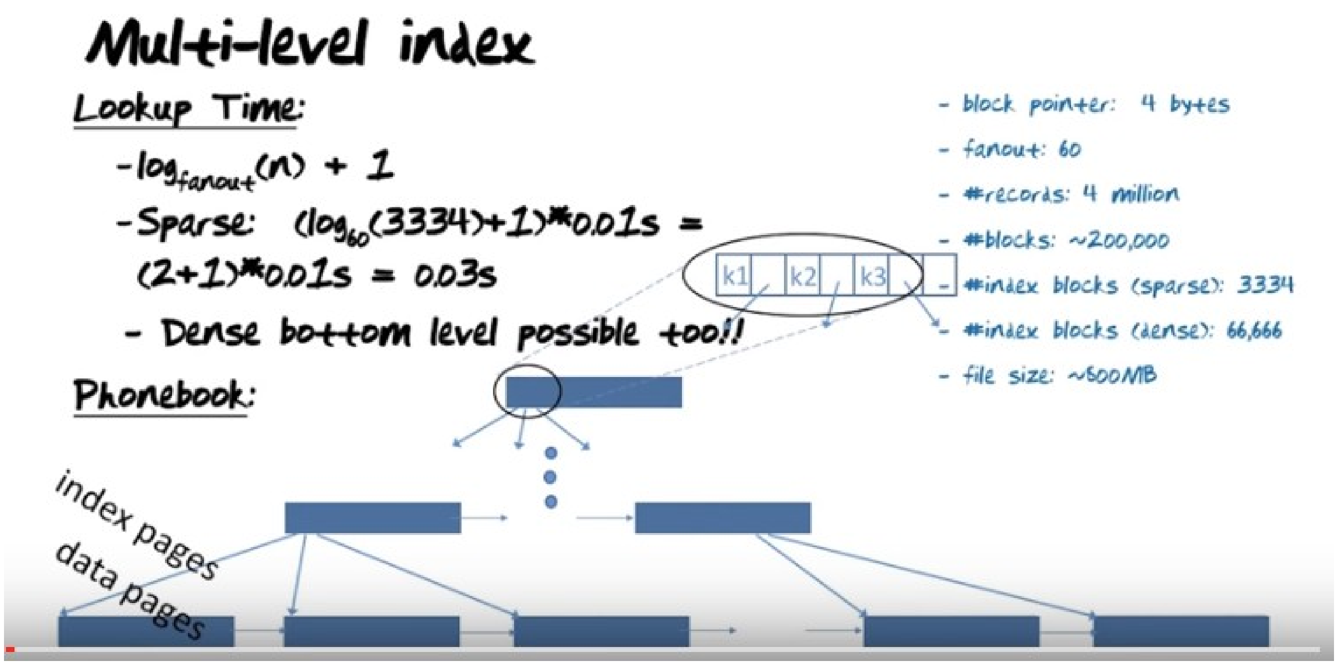
- $\log_2(n)+1$
- Dense: $(\log_2(4{,}000{,}000/60)+1)*0.01s = (16+1)*0.01s = 0.17s$
- Tricky for non-key fields

Phonebook

- block pointer: 4 bytes
- filled: ~80%
- fanout: ~60
- #records: 4 million
- #data blocks: ~200,000
- #index blocks (sparse): 3334
- #index blocks (dense): 66,666
- file size: ~800MB

The above figure: these data blocks are sorted on some key value. But what you want now is you want to build an index not on the key value, but on some other value. Now since the other value you want to build an index on will not be in sync with the order of the key values but will be all over the place, building this index is gonna be somewhat more painful. Also notice that since the values (of the secondary indices) here (in the data blocks) that we're going to build the index on are not sorted, we cannot build a sparse index. So all the indices that are secondary indices have to be dense indices. So here's the idea luginnai data. Go down to the first record (in the first data block), we find the value of the field we are building the secondary index on. That value, we insert into the index here (in the first index block) with a pointer back to where it came from. Look at the second record, pick up the key, put the pointer in, third record, etc, so every single one in our example 20 records here (in one data block) will have a key value and a corresponding pointer back to this... If you look at what these (index) blocks will look like,..., what you gonna see is of course that the pointers come nicely and point in order here because the way we explained construction... The key values are gonna not be ordered. So what we gonna need to do after this first initial step of building the index is we're gonna have to sort it. So after this sort, the key values we have picked up for the secondary index are all gonna come in order k1, k2, etc. But now the pointers are gonna point all over the place to different parts of the file. The implication of that is that a secondary index is good for point queires only. You cannot trust at once you follow the pointer afte a particular key value that the subsequent key values would come after that.

Lookup 的時間跟之前算的 dense index 一樣，都是 0.17s. Notice that I have implicitly made the following simplifying assumption. I made the assumption that the non-key field we are indexing on (應該就是那個 secondary index) here actually had unique values. So what if a value appears multiple times in that field that we want to build these secondary index on? Then we need to consider whether we are grouping those key values together so they don't have to be repeated all the time and then keep a bucket of pointers to all the different copies of that key value exists.



Multi-level index

Lookup Time:

- $\log_{fanout}(n) + 1$
- Sparse: $(\log_{60}(3334)+1)*0.01s = (2+1)*0.01s = 0.03s$
- Dense bottom level possible too!!

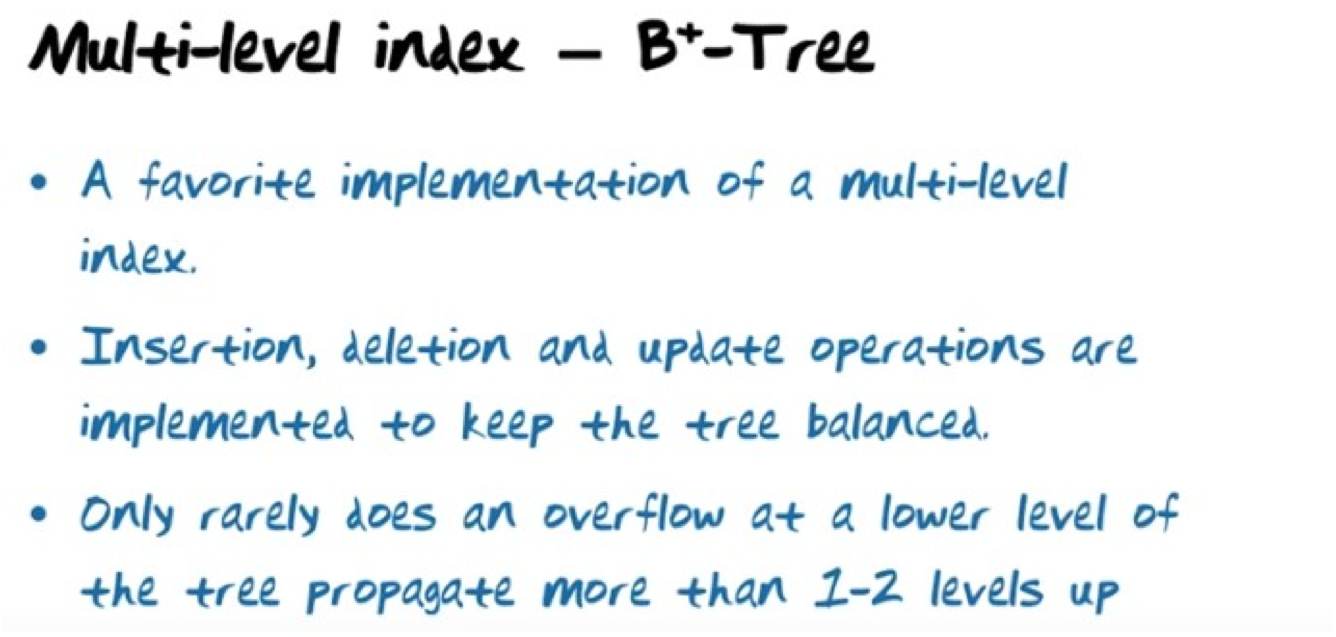
Phonebook:

- block pointer: 4 bytes
- fanout: 60
- #records: 4 million
- #blocks: ~200,000
- #index blocks (sparse): 3334
- #index blocks (dense): 66,666
- file size: ~800MB

上圖中的 log_60 的底為 60 之原因, 在後面會說. We have seen that reducing the size of the search space by building indices and then pay a slight extra penalty for accessing the data level, that is a smart way to proceed. But if it's a good idea once, maybe it's a good idea twice. So if you look at this index here(從下往上數的第二行), that's a sorted file. So why don't we build an index on the index (MO: 比如最初的 indices 為 : 1,2,3,4,5... 則 build an index on that index, build 出來的 index 為 : index_1,index_2,index_3,... 其中的 index_i 應該是一個 range, 比如 index_1 表示包括(最初的 index) 1~3, index_2 表示包括 4~6, ...這樣才方便, 也才更好地理解後面所說的 you cut in much smaller pieces like this. Then one of the pieces is going to qualify for continued search), and an index on the index of the index etc, until we end up at the root with just a single index page. So searching through this index would now work as follows: we look at the top level block, we know that block has a fanout of up to 60(即一個 index block 中有 60 個 key values). So among those 60 different key values, we pick the place where the key we are searching for either is equal to one of them or fits between two of them, and we follow the appropriate pointer to the next level. At that next level we have one data block, it has a fanout of 60. We fit in the key value where fits and follow the pointer to the index block at the next level. Eventually we come down to the bottom of the index level and we need to add one, in order to get to the data. So in this case here the search time is log but the base is not 2, the base is the fanout of the index. That fanout is substantially higher than the 2 we used for binary search in the data file. … So we can expect the cost here to be smaller and smaller and smaller, the higher the fanout is. … The more complicated construct like this beceoms, the more levels of indices we put on top, the more sensitive we are to overflow (此 overflow 的意思是某一層(即某 index level)裝不下了). So if we, for example, are trying to insert data at the bottom level (從下往上數的第一行), althogh we started out with some slos space and(at?) the end, once we come to the point where block actually needs to be broken up, and the question is that gonna have any impact on the first level index(從下往上數的第二行), and if it does how far is the ripple(漣漪) effect would be. It would be important to keep the ripple effect small and that's exactly why we have some slos space building at the end.

(Instructor speaks sitting there, so did not take screen shot)

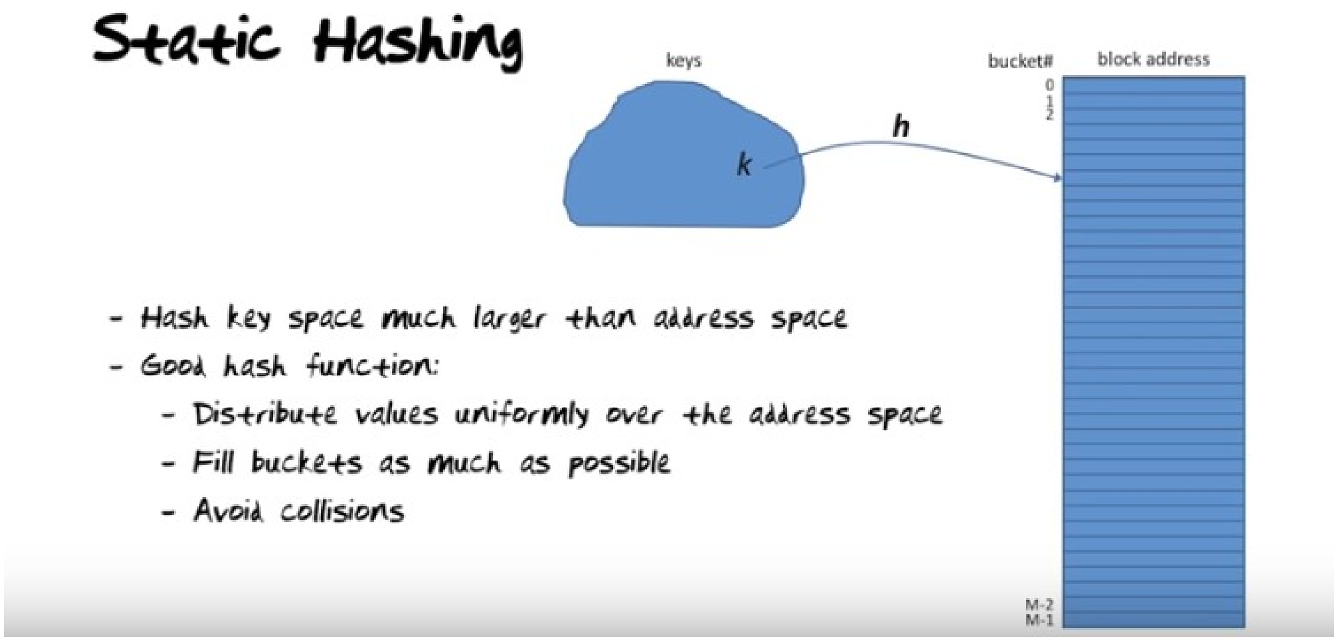
The above figure: So in a binary search, the idea was to cut the search space in two every single time.... The idea for multilevel index is that the multilevel index will allow us to cut the search space into much smaller pieces in each step. So using the multilevel index, I can illustrate as follows. Basically take the book, but instead of cutting in two, you cut magically, you cut in much smaller pieces like this. Then one of the pieces is going to qualify for continued search. 這就是為何前面 log_60 的底為 60 之原因.



Multi-level index – B^+-Tree

- A favorite implementation of a multi-level index.
- Insertion, deletion and update operations are implemented to keep the tree balanced.
- Only rarely does an overflow at a lower level of the tree propagate more than 1-2 levels up

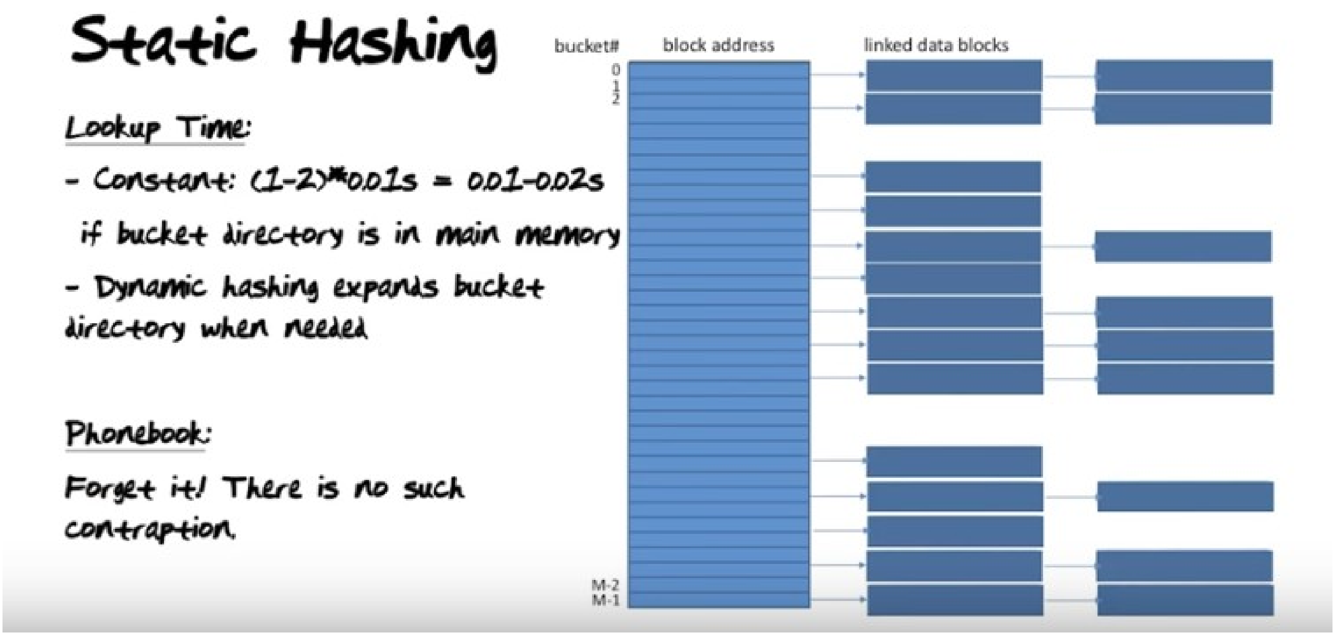
The above figure: In a B+ tree, all the data is at the bottom, and all nodes over that are index nodes in multiple levels. … B stands for balanced. So if the tree over time would deteriorate, so would the search time. So it's very very important that the distance from the root to the base level is the same at all times.



Static Hashing

- Hash key space much larger than address space
- Good hash function:
  - Distribute values uniformly over the address space
  - Fill buckets as much as possible
  - Avoid collisions

The above figure: you have a very large space of key values, think for example the email addresses of regular users.... The job of the hash function is to take keys from the key-value space and match them into an address in a bucket directory. 右邊的藍色大框中, 每一小格是一個 bucket (看一看 bucket # 0, 1, 2 就知). The addresses it can map to would be 0, 1, 2... M-2, M-1. "Fill buckets as much as possible" 意思是盡量將 右邊的藍色大框填滿. Collisions occur when the hash function maps actual keys that occur to the same space too many times.


Static Hashing

Lookup Time:

- Constant: (1-2)*0.01s = 0.01-0.02s
 if bucket directory is in main memory
- Dynamic hashing expands bucket directory when needed

Phonebook:

Forget it! There is no such contraption.

上圖中右邊第三列的意思是: now we actually had a second time that this address was here, and this data block was full, and therefore a second data block is allocated to continue to store record that were mapped in by the hash function to this bucket directory address. 若橫向加得太多, then this organization is going to be deteriorate, so the more data we insert with the same block directory size, the longer these lists of blocks will get. 若橫向太長, then the search in one of these linked list is actually gonna be like searching in a heap. So it's very very important to make sure that data is uniform distributed... contraption: 奇巧的設計;裝置