Organizace a zpracování dat I

Sepsal Ondra Štumpf v roce 2013, podle přednášek RNDr. Davida Hokszy, Ph.D.

Autor se tímto zříká veškeré zodpovědnosti za chyby v tomto dokumentu, stejně jako za následky plynoucí z jeho použití.

Datová média

- **Klasifikace**
  - podle výkonu (latence – doba přístupu, propustnost)
  - podle způsobu přístupu (náhodný, sekvenční)
  - podle ceny (cena za jednotku dat)
  - podle možností změn (pouze pro čtení, čtení i zápis, pomalý zápis – rychlé čtení)
  - podle volatility (volatilní – závislá na napájení, nevolatilní – data se zachovají i když médium není pod proudem)
- **Hierarchie** (od pomalých, levných, s velkou kapacitou po rychlé, drahé a malé)
  - offline storage (magnetické pásky, CD/DVD)
  - online storage (disky)
  - hlavní paměti
  - CPU registry
- **Primární paměti**
  - rychlé, volatilní – CPU registry, cache, hlavní paměti
  - **Registry**
    - uvnitř procesoru, využívané přímo CPU
    - 32 nebo 64 bitová slova (bloky dat)
    - nejrychlejší a nejdražší
  - **Cache**
    - uvnitř procesoru nebo disku
    - nejčastěji se tam ukládají data z hlavní paměti (rychlejší přístup)
    - mohou být hierarchické
  - **Hlavní paměti**
    - hlavní úložiště dat (ale zpravidla příliš malé na to, aby se tam vešlo úplně všechno)
- **Sekundární paměti**
  - střední přístupová doba, nevolatilní
  - nepřístupné pro samotné CPU
  - **Magnetické disky**
    - náhodný přístup
    - nelze nad nimi pracovat přímo – musí se napřed načíst do hlavní paměti, tam změnit a pak zapsat zpátky
  - **Flash paměti**
    - přibližně stejně rychlé jako hlavní paměti
    - paměťové karty, USB disky, SSD disky
- **Terciární paměti**
  - pomalé, nevolatilní
  - přenosné
  - diskety, CD/DVD, magnetické pásky
  - **Magnetické pásky**
    - obrovská kapacita

        - velká persistence (data tam vydrží dlouho)
        - sekvenční přístup
        - levné
        - často se používají na zálohování

Magnetické disky

- disková sestava = několik disků (**ploten** – typicky oboustranné) na společné ose
- data čtena a zapisována **hlavou** (typicky dvě – na každou stranu plotny jedna)
- povrch ploten rozdělen na **stopy** (soustředné kružnice)
- sada všech stop se stejnou vzdáleností od osy (na všech plotnách) je **cylindr**
- stopy jsou příčně rozděleny na **sektory**
- hlava se při čtení pohybuje těsně nad plotnou (resp. plotna rotuje pod hlavou), data zapsána pomocí magnetických impulzů
- sektory představují nejmenší možný objem dat k čtení nebo zápisu
- sektorů je různý počet na vnitřních a vnějších stopách (kvůli různému obvodu stop)
    - první disky měly počet sektorů všude stejný (uvnitř nahuštěné na sebe, venku naopak mezery)
    - **Zoned bit recording** (ZOR) – stopy se shlukují do zón, stopy v jedné zóně mají stejný počet sektorů. Důsledek – přenos dat čtených z kraje disku je rychlejší (na jedno otočení disku se přečte víc dat)
- parametry disků
    - **seek -** čas na nastavení hlavy na správnou stopu
    - **rotational delay** (latence) – jak dlouho trvá, než se správná data dostanou pod čtecí hlavu
    - **block transfer time** – doba na čtení jednoho bloku dat
- rychlost přenosu (transfer rate)
    - závisí na aktuální pozici hlavy
    - (average) **media transfer rate** – rychlost čtení nebo zápisu na jednu stopu jedné plotny (nepočítá se s posouváním hlavy ani s jejím počátečním nastavováním)
    - **interface/external transfer rate** – rychlost s jakou mohou být data přesunuta z disku do controlleru disku (čistě elektronická operace, mnohem rychlejší než mechanické přístupy na disk)
    - (average) **sustained/sequential transfer rate** – skutečná rychlost (když čtená data přesahují přes několik ploten a cylindrů). Media transfer time + head switch time + cylinder switch (track-to-track) time
- **sběrnice** – infastruktura na přenos dat mezi jednotlivými komponentami (PATA, SATA, SCSI...)
- **controller** (řadič)
    - rozhraní mezi diskem a zbytkem systému
    - přijímá instrukce pro čtení a zápis
    - controller na straně sběrnice se nazývá **host bus adapter** (HBA) a controller na straně disku je **disk controller**
- **disková rozhraní**
    - PATA (Parallel Advanced Technology Attachment)
        - paralelní – přenáší více bitů současně
        - až 167 MB/s
    - SATA (Serial ATA)
        - rychlejší přenos (až 600 MB/s)

        - umožňuje hotplug
        - eSATA – externí zařízení, mSATA – pro malá zařízení...
    - SCSI (Small Computer System Interface)
        - sada standardů pro přenos dat mezi počítačem a ostatními zařízeními (instrukce, protokoly, HW rozhraní...)
        - až 16 zařízení na jedné sběrnici
        - náročnější řešení, typicky v serverovém prostředí
        - až 640MB/s
    - Fiber Channel
        - hlavně pro síťová úložiště (SAN – Storage Area Network)
        - komunikace pomocí SCSI
        - až 1600MB/s
- způsoby připojení disku
    - DAS (Direct Attached Storage)
        - úložiště dat připojeno přímo k počítači (bez sítě) – protokoly ATA, SATA...
    - NAS (Network Attached Storage)
        - jedno zařízení (file server) připojený k síti, poskytující úložnou kapacitu pro více klientů
        - může obsahovat více disků
        - přístup pomocí síťových protokolů na sdílení souborů (NFS, SMB...)
        - filesystem spravovaný NAS OS
    - SAN (Storage Area Network)
        - síť složená z více úložišť
        - na SAN je typicky připojený jenom server, ne přímo klienti
        - výhodou oproti NAS je větší škálovatelnost – přesouvaní dat, zvyšování kapacity...
        - protokoly ISCSI (Internet SCSI), Fibre Channel

RAIDy

- Redundant Arrays of Inexpensive (Independent) Disks
- původní motivace – použití více malých levných disků místo jednoho velkého drahého
- dnešní motivace – vyšší spolehlivost, zálohování, paralelismus
- musí je podporovat controller disku
- liší se podle ceny, výkonu a spolehlivosti
- **Mirroring/shadowing**
    - uchovávají se kopie disku – každý zápis se automaticky provádí na více discích
    - čte se pouze z jednoho – když se pokazí, vezme se jiný
    - pravděpodobnost, že selže i druhý dřív než se první stihne opravit, je velmi malá
- **Parita**
    - například 3 disky + 1 paritní disk
    - na paritní disk se zapisují XOR hodnoty z prvních tří – když se jeden z nich pokazí, lze díky paritě dopočítat, co tam bylo
- **RAID 0**
    - neredundantní – jenom se použije víc disků místo jednoho
    - data se rozkouskují – každý blok na jeden disk (block-level striping)
    - výhody – velký výkon, žádná režie s paritami, využije se celá kapacita, jednoduché
    - nevýhody – žádná redundance, není odolné proti chybám (pokud se pokazí jeden disk, ztratí se všechna data)
    - hodí se na vysoce výkonné aplikace bez velkých nároků na bezpečnost

- **RAID 1**
  - zrcadlení disků – každý zápis je paralelní
  - čte se z toho disku, který umí data poskytnout nejrychleji
  - výhody – rychlost srovnatelná s rychlostí samotného disku, uloženy kopie dat, jednoduché
  - nevýhody – vysoká redundance (50%), typicky neumožňuje hot-swap poškozeného disku
  - hodí se pro aplikace s velkými nároky na bezpečnost
- **RAID 2**
  - bit-level stripping
  - používá Hammingovo kódování k zajištění redundance
  - nepoužívá se
- **RAID 3**
  - byte-level stripping
  - jeden paritní disk – každý zápis vyžaduje ještě dopočítání a zápis paritního bitu – ty se kontrolují při čtení
  - výhody – vysoká propustnost pro velké zápisy
  - nevýhody – náročné na zdroje (nutné alespoň 3 disky), pro malé zápisy pomalé, operace vyžadují aktivitu všech disků
  - nahrazen RAID 5
- **RAID 4**
  - block-level striping se zvláštním paritním diskem
  - podobný RAID 0 (+ paritní disk), RAID 3 (ale block-level), RAID 5 (ale nedistribuovaný)
  - výhody – operace se mohou provádět paralelně (pokud jsou dané bloky na různých discích)
  - nevýhody – jeden paritní disk může být úzkým hrdlem, pomalé pro hodně malých operací, složitý návrh controlleru
  - nahrazen RAID 5
- **RAID 5**
  - block-level striping s distribuovanou paritou
  - nejčastější
  - paritní bloky se nacházejí střídavě na všech discích
  - výhody – vysoká propustnost (není tam jenom jeden paritní disk), vysoká rychlost, zvládne poruchu jednoho disku
  - nevýhody – pomalejší zápisy (nutnost počítat paritu), alespoň 3 disky
- **RAID 6**
  - block-level striping se dvěma distribuovanými paritními bloky
  - výhoda - zvládne poruchu dvou disků
  - nevýhoda – pomalý zápis
  - hodí se pro kritické aplikace
- **RAID 0+1** – mirrored stripes
  - kombinace vysokého výkonu RAID 0 a vysoké spolehlivosti RAID 1
  - uchovávají se kopie (zrcadlení) celých RAID 0 bloků
  - výhody – vysoká rychlost, odolnost proti chybám
  - nevýhody – velká režie, vyžaduje 4 disky, málo škálovatelné
- **RAID 1+0** striped mirrors
  - kombinace vysokého výkonu RAID 0 a vysoké spolehlivosti RAID 1
  - rozkouskované pole, jehož prvky jsou RAID 1 bloky

- výhody – velmi velká odolnost proti chybám
- nevýhody – jako RAID 0+1

SSD disky

- Solid-state Drive
- neobsahuje pohyblivé části
- ukládá data do flash pamětí (nevolatilní, levnější) nebo DRAM pamětí (rychlejší, volatilní)
- controller obsahuje samostatný procesor – simuluje rozhraní magnetického disku
- typy NAND pamětí
  - SLC (Single-level Cell) – každá buňka obsahuje 1 bit, nejvyšší výkon a cena, malá kapacita, odolnost proti opakovaným zápisům
  - MLC (Multi-level Cell) – každá buňka obsahuje 2 bity, nejčastější
  - TLC (Triple-level Cell) – každá buňka obsahuje 3 bity
- výhody – tiché, malá spotřeba, odolné proti nárazům, rychlejší, není třeba chladit
- nevýhody – menší kapacita, vyšší cena, omezená životnost
- Hybridní disky
  - kombinace HDD a SSD

Další úložiště

- **Optické disky**
  - data ukládána a čtena pomocí laseru, založeno na odrazivosti světla
- **Magnetické pásky**
  - sekvenční přístup – hodí se pro přenos velkých souvislých bloků
  - malá cena za bit
  - pomalé
  - archivace, zálohování
  - kapacita až několik petabytů

Typy souborů

Pojmy

- **záznam**
  - reprezentuje objekt reálného světa (logický význam)
  - fyzická reprezentace velikosti R – obsahuje dodatečné informace (oddělovače...), uložena v blocích velikosti B
  - sekvence atributů (základních datových typů)
  - atribut je definován jménem a datovým typem
  - pevná délka (hlavička souboru obsahuje počet záznamů a jejich velikosti; na záznam je možnost přistupovat přímo pomocí jeho čísla) nebo proměnná délka (např. nepovinné atributy, atributy obsahující záznamy...)
- **soubor**
  - pojmenovaná sada záznamů
- databáze
  - kolekce souborů obsahující navzájem související data
- **blokovací faktor**
  - b = B / R
  - počet záznamů v jedné stránce
  - fixed blocking
    - záznamy pevné délky, záznam vždy v jedné stránce – možnost interní fragmentace
  - variable length spanned blocking
    - záznamy proměnné délky, přesahující přes hranice stránky – v každé je pointer na další (navazující) stránku
    - využita celá kapacita stránky, obtížně implementovatelné
  - variable length unspanned blocking
    - záznamy proměnné délky, záznam vždy v jedné stránce – možnost vysoké fragmentace

Soubory

- homogenní (obsahuje záznamy pevné délky) vs. nehomogenní (obsahuje buďto záznamy proměnné délky, nebo záznamy různých typů)
- při čtení je nejprve nutné načíst obsah do hlavní paměti
- při změně dat se musí nejdřív načíst starý obsah, ten změnit a zapsat zpátky
- operace CREATE, REMOVE, INSERT, UPDATE, DELETE, FETCH, FIND, REBUILD
- dotazy nad soubory
  - **jednorozměrné** (podle hodnoty jednoho atributu)
  - **vícerozměrné** (podle hodnot více atributů)
    - na úplnou shodu (zadány hodnoty všech atributů)
    - na částečnou shodu (zadány jen některé)
    - na úplnou intervalovou shodu (interval hodnot pro každý atribut)
    - na částečnou intervalovou shodu
- organizace souborů
  - popis logické struktury spolu s algoritmy na její změnu
  - může se skládat z více souborů → **primární soubor** je ten, který obsahuje vlastní data

- **halda**
  - záznamy proměnné délky (soubor není homogenní)
  - nový záznam vždy na konec souboru (nesetříděný)
  - obvykle se používá jenom s nějakou pomocnou strukturou
  - INSERT – O(1)(s+r+btt)
  - FIND – musí prohledat celý soubor O(N) – O(N/b)(s+r+btt)
  - proměnná délka záznamu – kvůli potřebě ukládat různé typy podobných záznamů (zaměstnanci...) nebo neatomické hodnotě některých atributů
  - možno nahradit maximální velikostí
- **sekvenční soubor**
  - nesetříděný
    - jako halda, ale záznamy pevné délky
    - INSERT – O(1)
    - FIND – musí prohledat celý soubor O(N) – O(N/b)
  - setříděný
    - ukládá záznamy setříděně podle hodnoty vyhledávacího klíče v každém záznamu
    - data je možno setřídit na základě hodnoty pouze jednoho atributu – zvolí se ten, podle kterého se bude nejčastěji vyhledávat
    - FETCH je možno zajistit buď sekvenčním prohledáváním (O(N)) nebo půlením intervalu (O(log(N))
    - je homogenní
    - INSERT
      - přímé vložení by bylo náročné, protože všechny následující záznamy by se musely posunout
      - používá se **oblast přetečení** – tam se přidávají nové záznamy, čas od času se celý soubor reorganizuje
    - UPDATE – pokud změna neobsahuje vyhledávací klíč, je to jednoduché – proto se zpravidla zavádí umělé (nic neznamenající) klíče
    - DELETE – záznamy se neodstraňují hned, protože by se musel přestavět celý soubor; záznam se jenom označí jako smazaný
- **index-sekvenční soubor**
  - jako sekvenční soubor, ale s indexem
  - skládá se z primárního souboru, kde jsou data setříděná podle primárního klíče, indexového souboru a oblasti přetečení
  - data možno číst sekvenčně nebo přímo
  - **index**
    - pomocná struktura tvořená setříděnými dvojicemi hodnota - pointer
    - může být buď v hlavní paměti (pokud je dostatečně malý) nebo v sekundární paměti (přístup k indexu je nutno započítat do celkové délky operace)
    - jednoúrovňový nebo víceúrovňový (master level umístěn v hlavní paměti)
  - FETCH – hledání na shodu, podle hodnoty X
    - podívej se do nejvyšší úrovně indexu a najdi pár (hodnota – pointer) s nejvyšší hodnotou nižší než X
    - načti blok, na který pointer odkazuje
    - opakuj tak dlouho, dokud nenačteš data z primárního souboru
    - prohledej blok primárního soubor a najdi X
    - složitost závisí na výšce stromu = $\log_p \frac{N}{b}$, kde b je blokovací faktor, R velikost záznamu (průměrná), p je blokovací faktor pro pár hodnota – pointer

        - počet přístupů na disk odpovídá počtu úrovní stromu
    - FETCH – intervalový dotaz
        - najdi dolní mez (viz výše)
        - sekvenčně čti následující bloky tak dlouho, dokud nepřesáhneš horní mez
    - UPDATE
        - při vkládání dat zůstává index stejný – nové záznamy je třeba vkládat do vyhrazených míst v primárním souboru
        - dvě možnosti řešení – ISAM a přihrádky
    - ISAM
        - Indexed Sequential Access Method
        - nové záznamy se vkládají na zvláštní místa na disku – závislé na použití magnetických disků
        - master index – v RAM, cylinder index – jak jsou záznamy rozmístěny přes jednotlivé cylindry, track index – na první stopě
    - přihrádky
        - každý blok obsahuje přihrádku, kam se vkládají nové (přetečené) záznamy
        - velikost primárního souboru tím už není konstantní
        - přetečená data se dají do nového bloku, na který odkazuje blok z primárního souboru
        - přihrádky je možné zřetězovat → ale vede to snížení efektivity
        - pointery do přetokových oblastí
            - po každém záznamu (náročné na místo, kratší sekvence v přetokové oblasti)
            - po každém bloku (delší sekvence, lépe se optimalizuje místo)
    - výhody – rychlý přístup pomocí primárního klíče, stejné výhody jako sekvenční soubor
    - nevýhody – rychlý přístup pouze pomocí primárního klíče, problémy s UPDATE – potřeba reorganizací
- **indexový soubor**
    - jako index-sekvenční, ale s možností mít více indexů
    - motivace – možnost vyhledávat v souboru na základě více atributů, ne jenom primárního klíče
    - primární soubor je nesetříděný nebo setříděný podle primárního indexu
    - pro každý klíč se postaví zvláštní index (možno i různých typů)
    - **primární index**
        - nad tím atributem, podle kterého je setříděný primární soubor
        - neměl by se často měnit (změna vyžaduje reorganizaci primárního souboru)
        - nemusí vůbec existovat
    - **sekundární index**
        - může jich být více
    - přímé indexování
        - index postaven přímo nad záznamy v primárním souboru → reorganizace primárního souboru způsobí reorganizaci indexu
    - nepřímé indexování
        - sekundární indexy obsahují klíče primárního indexu a ne přímo pointery do primárního souboru
        - je potřeba o jeden přístup navíc
        - i když se primární soubor reorganizuje, sekundární indexu zůstanou stejné
- **hashovaný soubor**
    - to, do jakého bloku souboru má záznam patřit, se zjistí pomocí hashovací funkce, definované nad nějakými atributy záznamu
    - přímý přístup

        - hashovací funkce mapuje klíče na adresy bloků/stránek
        - hodí se pro přístup pomocí jednoho unikátního klíče
        - hashing field – atribut, pomocí něhož je možný rychlý přístup
        - O(1) (pokud nedochází příliš k ukládání do přetokových oblastí)
    - adresování
        - přímé (hodnota atributu je přímo adresa)
        - mapování (vyhledávací klíč se přetransformuje na adresu bloku)

Jeden vyhledávací klíč?

- NE – indexový soubor
- ANO
    - Pouze sekvenční přístup?
        - ANO – sekvenční soubor
        - NE
            - Pouze přímý přístup?
                - ANO – hashovaný soubor
                - NE – index-sekvenční soubor

Bitmapy

- vhodné k indexování atributů s malými datovými typy
- pro každou hodnotu z domény se ukládá vektor bitů určující, které objekty mají danou hodnotu
- velikost bitmapy odpovídá počtu záznamů
- dotazy se snadno vyjádří pomocí logických operací AND, OR, NOT

Hashování

- technika umožňující přistupovat k záznamu v externí paměti v konstantním čase použitím hashovací funkce, která mapuje vyhledávací klíče na (fyzické nebo logické) adresy
- klíče se mapují na **kapsy**
- hashovací funkce mapuje prostor klíčů na adresový prostor
- hashovací funkce by měla být:
  - uniformní (měla by rozmisťovat klíče do adresového prostoru rovnoměrně, nevytvářet shluky a zbytečné kolize) – měla by záviset na všech bitech klíče
  - náhodná (každá kapsa by měla být stejně naplněna)
  - deterministická (hodnota funkce je závislá pouze na hodnotě klíče)
  - rychlá (výpočet by měl probíhat v konstantním čase)
- **triviální hashovací funkce**
  - numerická reprezentace klíče je přímo adresa
  - rychlé, ideální pro malé domény
- **modulo**
  - h(k)=k mod M
  - hodnota závisí pouze na několika spodních bitech → hodně kolizí
  - M by mělo být prvočíslo
- **binning**
  - h(k)=k/M (celočíselně)
  - hodnota závisí pouze na několika horních bitech
  - obdoba k modulo
- **mid-square**
  - umocní hodnotu klíče na druhou a vezme prostředních r bitů, za předpokladu že adresový prostor obsahuje hodnoty $[0, 2^{r-1}]$
  - výsledek závisí na všech bitech klíče
- interní hashování – hashovací struktura je v hlavní paměti, jedna kapsa odpovídá jednomu záznamu; algoritmy se liší podle způsobu řešení kolizí
- externí hashování - hashovací struktura se nevejde do hlavní paměti, efektivita se počítá vzhledem k počtu načtených bloků

Interní hashování

- Hashovací tabulka je v podstatě asociativní pole – hashovací funkce mapuje klíč na odpovídající hodnotu
- nutno řešit kolize – když hashovací funkce přiřadí dvěma různým klíčům stejnou hodnotu
- **separate chaining** (zřetězení)
  - kapsy obsahují odkaz na spojový seznam záznamů
- **open addressing**
  - kolizní záznam se vloží do příští volné kapsy
  - prohledávání – spočítej adresu A vyhledávaného klíče K → pokud na A nic není, záznam v tabulce není → jinak sekvenčně čti tabulku dokud nenajdeš K (nalezeno) anebo volné místo (nenalezeno)
  - jak hledat příští volnou kapsu pro klíč K?
    - sondovací funkce (pro klíč K a i-tý pokus dává další adresu, která se má zkusit)
  - primární klastrování
    - vytváří se shluky zaplněných kapes → větší pravděpodobnost kolizí

        - optimální sondovací funkce by měla dát adresu všech kapes se stejnou pravděpodobností
        - volné místo za takovým shlukem má mnohem větší šanci, že do něj bude přiřazen záznam
        - malé shluky mají tendenci se spojovat a vytvářet velké
    - lineární sondování
        - h(k,i)=(k+c*i)%M
        - c by mělo být nesoudělné s M
    - kvadratické sondování
        - h(k,i)=(k+a*i+b*i^2) % M
        - nutno vhodně volit konstanty a,b
    - pseudonáhodné sondování
        - perm... tabulka permutace adres kapes tabulky
        - h(k,i)=(k+perm[i])%M
    - dvojité hashování
        - h(k,i)=k+i*g(k)%M
- **coalesced chaining**
    - spojuje separate chaining a open addressing
    - používá jednu tabulku a ošetřování kolizí jako v otevřeném adresování
    - kolizní záznamy jsou zřetězovány
- **Cuckoo Hashing**
    - používá dvě hashovací funkce h,g
    - nedochází ke zřetězování ani sekvenčnímu čtení hashovací tabulky
    - pokud je adresa h(K) plná, stejně tam prvek vlož a vykopni tak existující prvek – ten se stejným způsobem pokus vložit na adresu g(K')
    - často implementováno pomocí dvou tabulek, každá s vlastní hashovací funkcí

Externí hashování

- statické
    - hashovací funkce mapuje klíče na pevný počet adres
    - umožňuje přidávat záznamy, ale ne zvětšovat adresový prostor
    - vhodné pro statické databáze
    - stránka = kapsa, kapsa obsahuje několik slotů, každý slot obsahuje jeden záznam
    - v případě potřeby se použije přetoková stránka
- dynamické
    - umožňuje změnu velikosti adresového prostoru a tím pádem změnu hashovací funkce
    - vhodné pro databáze, jenž mění svou velikost

Statické hashování

- **Cormack**
    - perfektní hashování (není potřeba používat přetokové oblasti, hashovací funkce se může měnit, vhodné pro časté hledání a relativně málo insertů)
    - potřebuje dodatečný O(N) prostor – adresář
    - potřebuje sadu hashovacích funkcí – jednu iniciální, ostatní které budou hashovat kolizní záznamy do zbytku adresového prostoru
    - předpokládá se malý počet kolizí, tedy je možné najít perfektní hashovací funkci pro

kolizní záznamy v rozumném čase
    - Adresář se skládá z trojic (p, i, r). Každá taková trojice nám říká, že se na adrese p v primárním souboru za sebou nachází r záznamů, které jsou do prostoru velikosti r bezkolizně rozmístěny hašovací funkcí H_i. Trojice, které nejsou použity k indexování do primárního souboru, poznáme podle nulové hodnoty r.
    - ACCESS – hledám klíč K
        - aplikujeme iniciální hashovací funkci na K → získáme adresu trojice v adresáři
        - záznam se může nacházet v primárním souboru mezi adresami p a p+r-1 → o rozmístění mi říká funkce H_i → aplikujeme ji na klíč a najdeme záznam
    - INSERT – vkládám K
        - Zjistíme skupinu r záznamů, do které bude nový prvek patřit. To se provede stejným způsobem, jakým dostaneme při vyhledávání index trojice v adresáři, tedy použijeme na klíč záznamu primární hašovací funkci.
        - Najdeme v primárním souboru souvislý blok volného místa, který postačí pro uchování r + 1 záznamů. Adresu námi nalezené volné oblasti si označíme q
        - Použijeme i-tou sekundární hašovací funkci, která nám určí rozložení záznamů v nově alokované oblasti. Pokud pozice nějakých dvou záznamů kolidují, zkusíme použít hašovací funkci H_i+1. Takovýmto iterativním postupem nakonec najdeme j takové, že H_j bezkolizně rozhodí r+1 záznamů do stejně velkého volného prostoru.
        - Překopírujeme r záznamů z původního místa do místa nového a přidáme k nim nový záznam. Rozložení nám zajistí sekundární hašovací funkce H_i.
        - Modifikujeme trojici v adresáři na hodnoty (q, j, r+1).
        - Označíme původní oblast s r záznamy jako volné místo.
- **Larson & Kalja**
    - perfektní hashování
    - používá dvě sady hashovacích funkcí:
        - H_i pro i = 1 … M generuje adresy stránek, na kterých by se mohl záznam nacházet
        - S_i pro i = 1 … M generuje sekvenci d-bitových stringů zvaných **signatury**
    - stránky mají přiřazený d-bitový string **separátor**, který spolu se signaturou stránky a záznamu rozhoduje o tom, jestli tam záznam je (může být)
    - záznam s klíčem k může být vložen do stránky H_i(k) pouze pokud S_i(k) je menší než separátor na dané stránce
    - záznamy jsou v rámci jedné stránky setříděny vzestupně podle svých signatur
    - signatura stránky odpovídá nejmenší signatuře záznamu, který se do stránky už nevejde
    - výchozí hodnota separátoru je 2^{d-1}
    - během jednoho INSERTU může dojít ke kaskádě dalších
    - signatury jsou celkem malé, je možné je udržovat v paměti a při vyhledávání je tak potřeba jenom jeden přístup na médium
    - ACCESS
        - Postupně se pro zvyšující se i funkcemi H_i a S_i generují adresa a signatura tak dlouho, až se nalezne stránka s danou adresou a se separátorem nižším, než je adrese odpovídající signatura. Buď se v dané stránce nachází, nebo není v celé databázi
    - INSERT
        - Najdeme stránku, kde by se měl záznam nacházet
        - Pokusíme se záznam vložit
        - Vejde-li se, ok
        - Nevejde-li se, setřídíme záznamy dle signatur
        - Záznamy s největší signaturou z bloku vyjmeme
        - Separátor bloku nastavíme na signaturu vytlačených záznamů
        - Vyjmuté záznamy vložíme zpět do DB

Dynamické hashování

- narozdíl od statického není shora omezená velikost struktury
- při naplnění blízko kapacitě u statických metod dochází k velkému zpomalení (chaining...)
- základem dynamického hashování jsou trie (prefixové stromy) → rozhodování se neděje na základě celého klíče, ale pouze jeho části
- zkrácením trie se vytvoří adresář → zkrátí se čas na vyhledávání
- k adresáři se přistupuje pomocí hashovací funkce → uniformita zajišťuje vyváženost trie (což je jinak NP-úplná úloha)
- trie je možno zcela odstranit → adresa odpovídá cestě v trie → umožňuje pouze zvětšování
- s využitím adresáře
  - stránky mohou být libovolně rozmístěny v adresovém prostoru
  - můžou se vytvářet přetokové oblasti
  - dobré využití místa
- bez adresáře
  - adresa stránky přesně odpovídá hodnotě hashovací funkce
  - využití místa může být nehospodárné (velké mezery mezi obsazenými bloky)
  - musí podporovat přetokové oblasti
- hashovací funkce musí být schopná se měnit podle objemu uložených dat
- **Fagin**
  - využívá se adresář – ten se skládá z **globální hloubky** d_G (maximální počet bitů potřebný k rozlišení libovolných dvou záznamů), každá stránka obsahuje **lokální hloubku** d_L (maximální počet bitů potřebný k rozlišení záznamů v rámci dané stránky; v podstatě to odpovídá počtu adresářových záznamů odkazujících na danou stránku)
  - přetečení vyvolá změny v adresáři a primárním souboru, ale nedochází k chainingu
  - ACCESS – hledám K
    - K'=H(K)
    - K"=H_dG(K')
    - přistup na stránku, na kterou odkazuje adresářový záznam s klíčem K"
    - projdi stránku a hledej K → když tam není, není ani jinde (žádné přetokové oblasti)
  - INSERT – vkládám záznam R s klíčem K
    - najdi stránku B, do které by K mělo patřit
    - pokud B není plná, záznam tam vlož a skonči
    - pokud je plná, tak ji rozštěp a opakuj
  - štěpení stránky
    - pokud $d_L < d_G$, vytvoř dvě nové, rozděl tam záznamy a oběma zvyš d_L o 1 → z adresáře tam budou ukazovat takové záznamy, které mají stejných prvních d_L bitů
    - pokud $d_L < d_G$, musíš rozštěpit adresář → zvětšit d_G a následně postupovat lokálním štěpením
  - výhody – rychlost je víceméně nezávislá na zaplnění
  - nevýhody – další úroveň čtení navíc (adresář), adresář by se nemusel vejít do paměti, při insertu může dojít k hodně štěpením a vzniku hodně prázdných stránek
- **Lineární hashování**
  - nevyužívá adresář
  - u Fagina rozštěpení adresáře nevedlo ke vzniku nových stránek → tady nová stránka vzniká na základě předem dané podmínky (počet insertů...) → tedy nemusí se štěpit stránka, do které právě vkládáme → využívají se přetokové stránky
  - proces rozšiřování je rozdělen do fází – d-tá fáze začíná, když je počet stránek $s=2^d$ a končí když je 2^{d+1}
  - split-pointer p, který v každém okamžiku ukazuje na příští štěpenou stránku; na začátku

ukazuje na nultou stránku a při každém štěpení se posune o 1
    - po štěpení bude další štěpenou stránkou s+p
    - Máme hashovací funkci H vracející hodnoty o d bitech, nějaké stránky a oblast přetečení. Na začátku ukládáme všechna data do jedné stránky. Když provedeme L operací vkládání, rozdělíme tuto stránku na dvě a hodnoty v ní rozházíme podle posledního bitu hashovací funkce. Po dalších L operacích znovu dělíme stránku 0 a hodnoty nyní rozhazujeme podle posledních dvou bitů... po každých L krocích analogicky dělíme určenou stránku, určena je tímto způsobem: 0, 0,1, 0, 1, 2... (čímž nám, to je náhodička, stránky přibývají v pořadí dělení ?00,?1, ?10...).
    - Při vkládání máme stránku jednoznačně určenou, pokud se do ní záznam nevejde, použijeme oblast přetečení. Při každém dělení stránek oblast přetečení projdeme a přepočítáme
    - vyhledávání → spočtu si, v jaké fázi jsem (d = floor(log(pageCount)), jaká je poslední stránka v dané fázi (s = exp(2,d)), kolik stránek přebývá (pageCount % s). Spočtu K % s – pokud je to větší nebo rovno p, je to adresa stránky, jinak je adresa K % exp(2,d+1)
    - **odložené štěpení**
        - sdílení přetokových stránek
        - „buddy pages“ - logické párování stránek – při přetečení se data dají do spřátelené stránky, pokud to nejde, původní stránka přeteče
        - nekontrolované štěpení
            - stránka, na kterou ukazuje p, se rozštěpí po L insertech
            - nebo se rozštěpí, jakmile nějaká stránka přeteče
        - kontrolované štěpení
            - ke štěpení dojde, pokud stránka p dosáhne určité zaplněnosti (třeba 80%)
- **rekurzivní lineární hashování**
    - s přetokovou oblastí se pracuje jako s dynamicky hashovaným souborem → stránky v přetokových oblastech mohou samy o sobě přetékat → rekurze
    - přetokové stránky nejsou přímo spojeny s původní stránkou
    - štěpení
        - když dojde ke štěpení stránky na úrovni i, sesbírají se stránky na úrovni i+1 a níže
        - pokud primární stránka pořád přetéká, přetečené záznamy se dají do nižší úrovně → obvykle stačí 3 úrovně
    - vyhledávání probíhá podobně jako v lineárním hashování, ale musí se rekurzivně prohledávat nižší úrovně
- **multipage nodes**
    - řeší problém růstu adresáře u struktur, které ho využívají → nastaví se pevný horní limit
    - když se limitu dosáhne, zvětšují se uzly (→ multipage nodes), ne adresář
    - při přístupu je potřeba prohledat nejprv adresář a pak multipage nodes – ty se uchovávají za sebou, buď v pořadí, v jakém byly přidány, nebo jako samostatný hashovaný soubor
- dynamické techniky hashování nemají vyrovnaný výkon – všechny stránky se plní víceméně stejně rychle, takže pak dojde k hodně štěpení naráz; využití místa dosáhne skoro 100% a pak skokově spadne na 50% → řešení pomocí LHPE nebo spiral storage
- **Linear Hashing with Partial Expansion** (LHPE) (lineární hashování s částečnou expanzí)
    - na konci fáze se přetokové oblasti blízko pravého kraje zatím neštěpené oblasti prodlužují → nedávno štěpené stránky mají rezervy a naopak dlouho neštěpené stránky jsou maximálně využity
    - v lineárním hashování se stránka s-1 rozštěpí po s*L insertech → tady se rozlišuje částečná a úplná expanze
    - během úplné expanze se soubor rozšíří ve více částečných fázích a v každé částečné fázi se rozštěpí všechny stránky

- pokud je počet částečných expanzních fází 2, pak se stránka s-1 rozštěpí po s*L/2 insertech

- **Spiral Storage**
  - nevyužívá adresář
  - v lineárním hashování mají stránky na levém konci primárního souboru tendenci být nevyužity a naopak stránky na pravém konci přetékají
  - pokud by klíče exponenciálně klesaly, stránky vlevo by dostávaly více záznamy a tím by se rozdíl vyrovnal
  - spiral storage funguje tak, že bere klíče (s uniformním rozdělením), přiřazuje jim exponenciální rozdělení a pak logické adresy
  - na rozdíl od většiny schémat se spiral storage nerozšiřuje přidáváním stránek vpravo
  - během expanze dochází jak k mazání, tak přidávání stránek (přidává se jich víc) → záznamy ze smazaných stránek se rozhodí mezi ty nové
  - aplikuje se funkce G(hash(K)): <0;1> → <c;c+1>
  - logickou adresou je pak floor($b^{G(K)}$)
  - během expanze se musí vybrat nové c tak, aby nekolidovalo se starými stránkami: $new_c = \log_b(c+1)$

- **Membership Testers**
  - slouží jenom ke zjišťování přítomnosti klíče v nějaké množině (nevydávají obsah)
  - můžou vydávat falešné positivní nálezy (ne příliš často – jde ověřit), ale neměly by vydávat falešné negativní
  - exaktní
    - mějme universum U obsahující u prvků → tester musí nějak representovat 2^u podmnožin, na to potřebuje $\log(2^u)=u$ bitů
    - pokud se omezíme na podmnožiny pevné velikosti n, pak stačí $\log_2 \binom{u}{n} \approx n \log u$ bitů
  - přibližné
    - máme bitový string délky b a k hashovacích funkcí U → {1,...,b}
    - pokud prvek x leží v X, nastavíme bity ve stringu odpovídající h_1(x), …, h_k(x) na 1
    - při zjišťování, zda y leží v X, musí platit h_1(y)= …= h_k(y) = 1 (ale může to být planý poplach)
    - není možné mazat → 1 v bitovém stringu může odpovídat více prvkům

Stromové indexy

- cílem je najít daný záznam podle klíče pomocí jen malého počtu operací
- indexy založené na stromech představují alternativu k index-sekvenční organizaci souboru
- nevýhody index-sekvenční organizace
  - statická povaha
  - při vkládání na začátek se musí celý soubor reorganizovat
  - reorganizace může trvat dost dlouho → není použitelné, pokud se dějí často
- při vkládání do primárního souboru se částečně upraví indexovací struktura v sekundárním souboru → úpravy znamenají štěpení a slévání uzlů stromu
- stromy – neorientované grafy bez cyklů, s jedním kořenem; každý rodič má konečně mnoho synů, každý syn má právě jednoho rodiče – kromě kořene
- arita/řád stromu = maximální počet potomků
- hloubka uzlu = počet hran od kořene k uzlu
- hloubka stromu = maximum z hloubek uzlů
- úroveň stromu = uzly ve stejné hloubce
- šířka úrovně = počet uzlů na dané úrovni
- výška uzlu = počet hran na nejdelší cestě z uzlu do listu
- výška stromu = výška uzlu
- vyvážený strom = počet uzlů na každé úrovni je maximální; každý uzel má přibližně stejnou výšku → celý strom má nejmenší možnou výšku
- uspořádaný strom = závisí na pořadí potomků
- **binární strom**
  - každý uzel má nejvýše dva syny
  - perfektní (plný) binární strom = každý vnitřní uzel má právě dva potomky
  - úplný binární strom = plný strom plněný zleva
  - počet uzlů perfektního binárního stromu výšky h = $2^{h+1} - 1$
  - počet listů = 2^h
- **binární vyhledávací strom** (BST)
  - uspořádaný binární strom – všechny uzly v levém podstromě mají nižší hodnotu, všechny uzly v pravém podstromě mají vyšší hodnotu
  - vyhledávání – jasné
  - **neredundantní** – data obsažená i ve vnitřních uzlech
  - **redundantní** – data obsažená pouze v listech – hodnoty uzlů se tím pádem mohou opakovat
  - použití – expression trees, Huffmanovo kódování, optimalizátory dotazů v DB
- **M-ární stromy**
  - motivace – binární stromy se nehodí pro uložení na discích, protože jsou příliš vysoké
  - zvyšování arity → snižování výšky stromu
  - m-ární strom obsahuje v každém uzlu m-1 klíčů, každý uzel má m potomků
  - každý podstrom obsahuje pouze hodnoty omezené zdola levým klíčem v rodiči a shora pravým klíčem
  - počet uzlů (arita stromu m, výška h) = $\sum_{i=0}^{h} m^i = \frac{m^{h+1}-1}{m-1}$
  - maximální počet uzlů = $\frac{m^{h+1}-1}{m-1} \cdot (m-1) = m^{h+1} - 1$
  - minimální výška při n uzlech = $\log_m(n+1)$ → minimální počet operací potřebných k nalezení záznamu → O(log(n))

- maximální výška = $\frac{n}{m}$ → O(n)

- **B-stromy**
  - vyvážený m-ární (ne binární) strom s dodatečnými pravidly pro přidávání a odebírání prvků → cílem je minimalizovat výšku stromu, maximalizovat šířku
  - kořen má alespoň dva potomky, není-li listem
  - každý vnitřní uzel kromě kořene má alespoň ceil(m/2) a nejvýše m potomků
  - každý uzel obsahuje alespoň ceil(m/2)-1 a nejvýše m-1 ukazatelů na data
  - každá větev je stejně dlouhá
  - uzly mají strukturu: potomek,(klíč,potomek,data),(klíč,potomek,data),...,volné místo a jsou uspořádány podle klíče
  - hodnoty v podstromě leží v intervalu daném krajními klíči v rodiči
    $\forall k \in U(p_{i-1}) : k < k_i$
    $\forall k \in U(p_i) : k > k_i$
  - redundantní stromy nemají data ve vnitřních uzlech, ale jenom v listech → je povoleno opakování klíčů
  - obvykle jedna disková stránka obsahuje jeden uzel stromu
  - FIND – jasné
  - INSERT – vkládám K
    - pokud je strom prázdný, vytvoř nový kořenový uzel a vlož do něj nový klíč
    - jinak najdi (FIND) list L, do kterého K patří
    - pokud L není plný, vlož tam K tak, aby klíče v L byly setříděné
    - pokud je L plný → vytvoř nový uzel L'
    - dolní polovinu L tam nech, horní polovinu přesuň do L' → pokud existuje prostřední (lichý) klíč, označ ho K'
      - pokud L je kořen, vytvoř nový kořen, dej do něj K' a odkaž se z něj na L a L'
      - jinak přesuň K' do rodičovského uzlu a uprav jeho odkazy na L a L'
      - pokud rodičovský uzel přeteče, opakuj
  - DELETE – mažu K
    - najdi uzel N obsahující K
    - odstraň K
    - pokud má uzel pořád dost klíčů, skonči
    - pokud jich má málo:
      - zkus ho spojit s levým nebo pravým sourozencem → snížení klíčů v rodiči
      - jinak rozděl záznamy mezi N a jeho sourozence
      - pokud je to zapotřebí, opakuj pro rodičovský uzel
- **B+-stromy**
  - redundantní B-strom (data pouze v listech)
  - listy jsou navzájem provázané → hodí se pro intervalové dotazy
  - preferováno v současných DBMS
  - výhody B-stromu – neredundantní, rychlejší nalezení záznamu (pokud je ve vnitřním uzlu)
  - výhody B+-stromu – menší vnitřní uzly (→ více klíčů → nižší strom), snazší inserty a delety, vhodné pro intervalové dotazy

- **bulk-loading** („nahrávání ve velkém“)
  - vkládat prvky jeden po druhém je pomalé
  - možné řešení:
    - setřiď vstupní data podle klíče

        - postupně přidávej klíče do listů, do rodiče dej vždy nejpravější hodnotu z listové stránky
- **page-balancing**
    - úprava B-stromu, ve které přetečení nemusí vyvolat štěpení stránky
    - když dojde k přetečení
        - podívej se na levého nebo pravého souseda a vytvoř z něj, z přetečené stránky a nového záznamu novou množinu X, kterou rovnoměrně rozděl do dvou uzlů
        - musí se změnit dělící uzel v rodiči → ale nemusí dojít ke kaskádovému štěpení
- **B*-stromy**
    - zobecnění page-balancingu
    - kořen má alespoň dva potomky
    - všechny větve jsou stejně dlouhé
    - každý uzel kromě kořene má alespoň ceil[(2m-1)/3] potomků
    - dvě plné stránky se rozštěpí na tři poloprázdné → jedna nová
    - pokud je uzel plný, ale oba z jeho sousedů ne → page-balancing
    - pokud je uzel plný a má plného levého nebo pravého souseda:
        - z plného uzlu, plného souseda a nového záznamu se vytvoří X
        - vytvoří se nová stránka
        - X se rovnoměrně rozdělí mezi tři stránky → nový klíč se dá do rodiče a případně se upraví
- **odložené štěpení** (deferred splitting)
    - v původních B-stromech může jistá sekvence insertů vést pouze k polovičnímu využití místa
    - každý uzel bude mít přetokovou stránku
    - pokud jak uzel, tak přetoková stránka jsou plné, uzel se rozštěpí a přetoková stránka vyprázdní
- **záznamy proměnné délky** (VLR)
    - cílem indexovat nejenom pomocí čísel, ale pomocí stringů → různé m (arita) pro různé uzly
    - při štěpení stránek s VLR se bere v úvahu spíš délka záznamů než jejich počet → rozmístění se řídí výslednou délkou → mlže dojít k porušení invariantu ohledně minimálního počtu záznamů v uzlu
    - delší záznamy mají tendenci přibližovat se ke kořeni → zvětšuje se výška stromu
- **prefixové (B-)stromy**
    - modifikace redundantního B-stromu
    - vnitřní klíče nemusí být podmnožinou klíčů v listech → mohou být kratší → nižší strom
    - vhodnou volbou jsou prefixy listových klíčů (stále je musí rozdělovat)
- **klastrované vs. neklastrované indexy**
    - klastrované – logické pořadí klíčů v indexu odpovídá fyzickému uspořádání záznamů v primárním souboru; v podstatě to odpovídá index-sekvenčnímu souboru
    - neklastrované – pořadí klíčů v indexu neodpovídá pořadí záznamů v datovém souboru
    - klastrované indexy se hodí při intervalových dotazech – datový soubor se mlže číst sekvenčně
- **buffered repository tree**
    - rozšířený (2,4) strom
    - každý list má až 4 položky, každý vnitřní uzel obsahuje klíče a buffer na až b položek
    - položky v potomcích a bufferech dodržují obvyklá pravidla setřídění
    - insert
        - záznam se vloží do bufferu kořene

        - když buffer přeteče, data se vloží do potomků
    - search
        - prohledávají se rekurzivně buffery
- **Disk-Access Machine (DAM) vs. Cache-Oblivious Model (CO)**
    - DAM – dvě úrovně paměti (hlavní a sekundární), parametry – velikost bloku B a velikost paměti M → počítá se množství přenosů mezi paměťmi
    - CO – B a M jsou neznámé jak algoritmu, tak programátorovi → platformě nezávislé zobecnění víceúrovňové hierarchie pamětí
- **fraktální stromy**
    - cache-oblivious lookahead array (COLA)
    - skládá se z ceil(log N) polí, kde i-té pole má 2^i prvků
    - každé pole je buď úplně plné nebo úplně prázdné
    - i-té pole něco obsahuje pouze tehdy, pokud i-tý nejméně významný bit z N je 1
    - každé pole má své prvky setříděné vzestupně
    - při vkládání nového prvku se postupuje podobně jako u sčítání s přenosem – vytvoříme seznam délky 1, podíváme se, jestli seznam délky 1 existuje a pokud ano, tak je sloučíme → vznikne seznam délky 2 a pro něj to opakujeme
    - složitost insertu - $O(\frac{1}{b}\log_2 N)$ amortizovaně, $O(\log_2 N)$ v nejhorším případě
    - složitost vyhledávání - $O(\log_2^2 N)$ pro hloupé řešení s půlením intervalů, $O(\log_2 N)$ s využitím dopředných ukazatelů
    - **dopředné ukazatele**
        - každý prvek obsahuje pointer na místo, kam patří v následujícím poli → nemusí se prohledávat celé pole

SSD disky

- neobsahují pohyblivé komponenty
- obsahují flash paměť založenou na „plovoucích hradlech“ (floating gate transistors), která umožňuje nevolatilnost
  - buňky (transistory) jsou schopné udržovat dodané elektrony → a tím i náboj a tím pádem data
  - pokud je buňka nabitá, reprezentuje hodnotu 0, jinak 1 (nenabitá buňka je schopná vést proud)
- buňky jsou uloženy v mřížkách zvané **bloky**, s řádky nazývanými **stránky**
- stránky se skládají z hlavní paměťové oblasti a rezervované oblasti (oprava chyb, režijní data)
- SSD disk se skládá z několika takových bloků
- způsob propojení buněk určuje typ paměti
  - **NOR**
    - všechny řádky a sloupce jsou navzájem propojeny
    - čtení probíhá tak, že se pustí proud do horizontálních spojů (word lines) a sleduje se napětí na vertikálních spojích (bit lines) → pokud buňka obsahovala 1 (nenabitá), proud projde, jinak ho zastaví elektrony v nabité buňce
    - problémem je složitost a prostorová náročnost → v SSD se spíš používají NAND
  - **NAND**
    - vertikální spoje (bit lines) jsou řešeny sériově (ne paralelně jako u NOR)
    - buňka vede proud vždy, když je nabitá alespoň mezním proudem c_T
    - všechny horizontální spoje (word lines), z jejichž buněk se zrovna nečte, jsou nabity proudem alespoň c_T
    - ta word line, ze které se má zrovna číst, se nabije menším proudem a na bit linech se sleduje výstup → je možno číst více zároveň
- stránka je nejmenší adresovatelný blok dat (obvykle 8kB)
- zcela prázdná stránka obsahuje samé 1 (nenabité)
- není možné vrátit samotnou buňku nebo stránku do stavu samých 1 → mazání lze provádět pouze na úrovni celých bloků stránek → data nelze měnit „na místě“
- stránka se může nacházet v jednom ze tří stavů: **volná**, **používaná** (live), **zahozená** (dead)
- **update** stránky se liší podle toho, jestli je dostupná jiná volná stránka
  - pokud ano → změněný obsah je zapsán do volné stránky, stará stránka se zahodí a nemlže být použita do té doby, než se vymaže celý její blok
  - pokud ne (ale je tam nějaká zahozená stránka) → celý blok se načte do cache, smaže se a zapíše se zpátky s provedenými změnami
- při **zápisu** se všechny word lines nabijí a ty bit lines, které mají mít hodnotu 0, se uzemní, čímž se elektrony dostanou do správných buněk
- **mazání** stránky znamená uvolnit napětí, pod kterým stránka je
- každý takový cyklus zanechává v buňkách nějaké zbytkové elektrony → při příštím nabití je potřeba větší proud → data je možno číst, ale ne přepsat → dochází k **degradaci paměti** (po zhruba 5 letech)
- pokud je možno v buňkách pracovat jenom s jednou úrovní napětí, může buňka obsahovat jenom 1 bit → SLC (Single-level Cell)
- existují i MLC (Multi-level Cell) → jsou tam 4 možné úrovně napětí, tedy každá buňka obsahuje 2 bity (vyšší kapacita a složitost čtení a zápisu, nižší životnost)
- SLC jsou spolehlivější, ale dražší → podniková řešení
- hlavním problémem SSD disků jsou updaty – musí se kopírovat hodně dat, což snižuje

životnost disku. Řeší se buď hardwarově (návrh controlleru disku) nebo softwarově (aplikace; minimalizuje se množství update operací)

- **přístup přes SSD controller**
  - stará se o komunikaci mezi paměťovými bloky a zbytkem počítače
  - **paralelismus** – je možné adresovat více NAND buněk současně; controller kouskuje data podobně jako v RAIDu a zároveň poskytuje základní opravu chyb
  - **cachování** – když kouskování nestačí, použije se SDRAM k dočasnému zadržení dat, než mohou být zapsána na disk (SDRAM je volatilní → potřeba napájení)
  - **garbage collection**
    - uchovává se přehled o zahozených stránkách → jednou za čas se bloky, které mají dost mrtvých stránek vezmou a znovu zapíšou → mrtvé bloky se tím (teprve) odstraní
    - příkaz TRIM – současné OS neodstraňují soubory fyzicky, jenom na ně zruší ukazatele → controller se nedozví, že soubor už neexistuje → musí se zavolat TRIM
    - zvyšuje rychlost, ale snižuje životnost
  - **wear leveling** (vyvažování zátěže) – velmi často používané stránky by se brzo opotřebovaly → controller je občas přemístí jinam, aby se vyrovnala zátěž
- **aplikační přístup** (ke zvýšení životnosti disku)
  - cílem je snížit počet update operací a naopak využít paralelismu
  - **problém s B+-stromy**
    - využívá se tam blokového přístupu – související záznamy se dají na stejnou stránku → update se dotkne jen málo stránek
    - když se to samé udělá hloupě na SSD discích, vznikne problém „**putujících stromů**“
    - řekněme, že vnitřní uzly obsahují jenom klíče a pointery, data jsou v uzlech. Takže když se změní data (tj. změní se stránka, na které data jsou) – musejí se zapsat do jiné stránky, protože SSD neumožňuje in-place update. Takže se vytvoří nová stránka a tam se data přesunou – jenže tím se změní pointer na stránku ve vnitřním uzlu, takže se musí přepsat... končí to až v kořeni
  - **filesystémy pro flash paměti**
    - block mapping techniky (Flash Transaction Layer – FTL)
      - rozhraní emulující rozhraní HDD
      - simulace in-place updatů pomocí přepisování stránek
      - vyvažování zátěže distribucí dat uniformně přes disk
    - specifické filesystémy
      - založeny na filesystémech používající logy
      - často samy implementují B-stromy
    - **μ-stromy** (Minimally Updated Tree)
      - všechny uzly na jedné cestě od kořene do listu se dají na stejnou stránku
      - uzly mají proměnnou velikost (na rozdíl od klasických B-stromů)
      - list vždy zabírá půlku stránky
      - vyšší úroveň → poloviční velikost uzlu
      - pouze kořen má stejnou velikost jako jeho potomci
    - **BFTL** (B-tree FTL)
      - cílem je zamezit tomu, aby se po každém insertu kopírovala celá stránka → změny se shromažďují v Reservation Bufferu (RB) a provádí se po větších celcích
      - vlastně je to další vrstva mezi FTL a samotným filesystémem
      - Node Translation Table (NT) – logicky spojuje uzly uložené ve více stránkách → čtení takového uzlu pak znamená číst více stránek

- aby se takto nevytvářely moc dlouhé řetězce, je nastaven horní limit → když se ho dosáhne, dojde k přemístění na méně stránek *(defragmentace?)*

- **FlashDB**
  - obsahuje mechanismus, který se dynamicky uzpůsobuje momentální zátěži – jestli se spíš čte nebo spíš zapisuje
  - vytvořeno pro sensorické sítě – nejdřív se udělá hodně zápisů a pak se teprve čtou
  - dva režimy, využívající dva návrhy – systém se v danou chvíli nachází právě v jednom z nich, podle toho, jaké operace převažují
    - **B+-strom (disk)** – založený na FTL (abstrakci disku)
      - i při změně malé části uzlu se musí celý uzel nakopírovat jinam – nehodí se pro časté zápisy
    - **B+strom (log)** – založený na filesystémech využívající logy
      - operace zápisu představuje záznam v logu, který je umístěný někde v bufferu → teprve když se buffer naplní, zápisy se provedou
      - čtení je náročné, protože se musí navíc prohledávat logy → nehodí se pro časté čtení
  - když se „spíš zapisuje“, systém pracuje v logovém režimu; když se „spíš čte“, pracuje se v diskovém režimu
- **FD-tree**
  - náhodné zápisy se převádí na sekvenční (ty se lépe provádějí)
  - náhodné zápisy se omezí na malou oblast
  - podporuje se přístup k více stránkám současně
  - strom se skládá z:
    - head tree – malý B+-strom na vrcholu
    - uspořádané běhy rostoucí velikosti pod ním
  - obdoba fraktálního stromu
- **PIO B-strom**
  - optimalizovaný B-strom
- **paralelismus v SSD (?)**
  - Channel-level
    - při vyžádání dat z různých packagů využívající více kanálů může přenos po těchto kanálech probíhat současně
  - Package-level
    - podobné kouskování dat jako v RAIDu
  - parallel synchronous I/O (psync I/O)
    - jako standardní sychnronní I/O, akorát se každá operace rozloží na pole menších, které jdou vykonat paralelně
  - Multi Path Search (MPSearch)
    - při vyhledávání ve stromě na úrovni i je napřed nutné znát výsledky z úrovně i-1 → MP řeší paralelní vyhledávání
    - najdi pointery na odpovídající stránky v kořenovém uzlu
    - načti tyto stránky pomocí psync I/O
    - prozkoumej uzly jeden po druhém a posbírej platné ukazatele → opakuj, dokud se nedostaneš do listu
  - optimalizace update operací – vloží se do Operation Queue (OPQ)
  - OPQ má setříděnou a nesetříděnou část – po každých k insertech se provede třídění
  - update se spustí, až když je OPQ plná (**batch update**) – podobné MPSearch