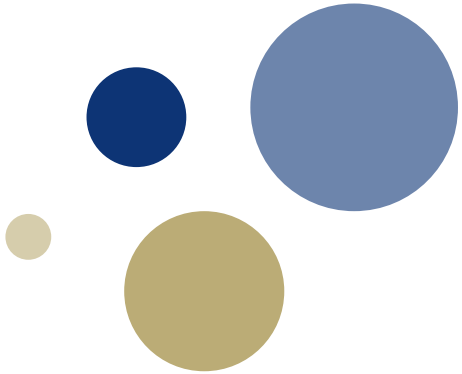
NTNU

Kunnskap for en bedre verden



Lagring og indeksering

Svein Erik Bratsberg

IDI/NTNU, versjon 19. mars 2026

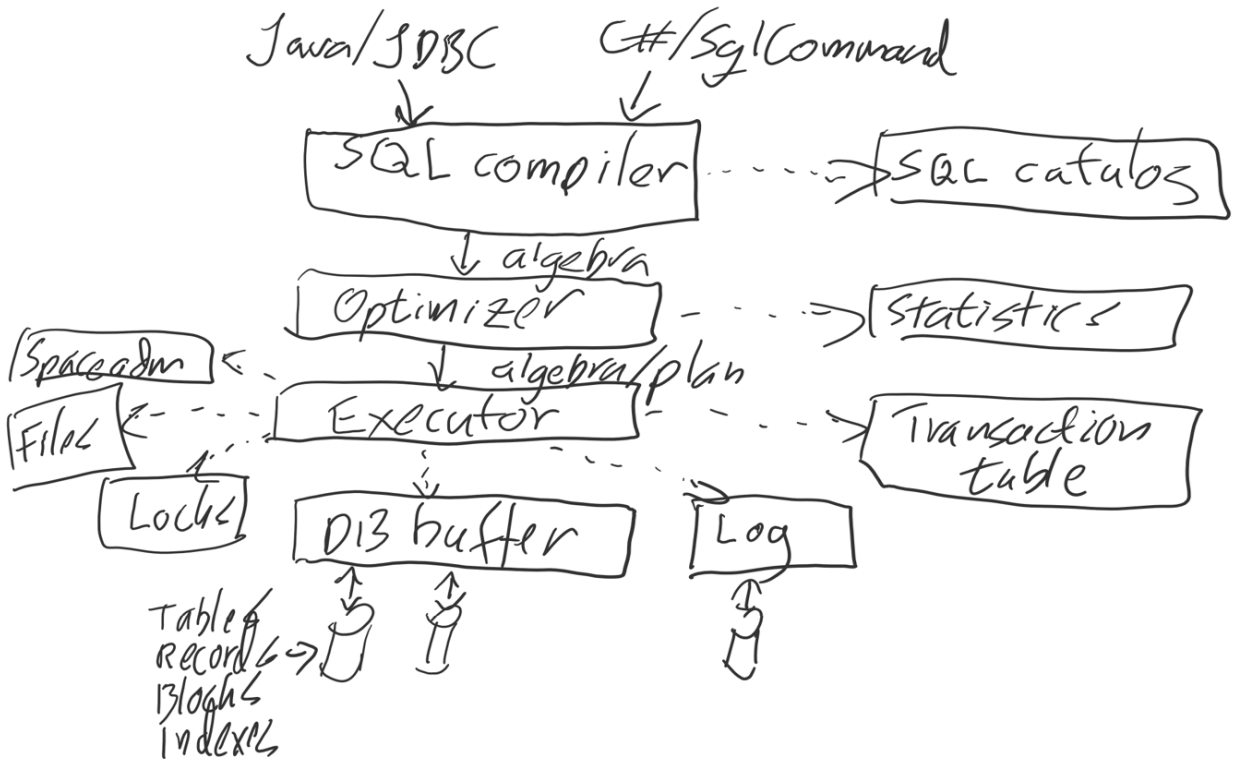
Læringsmål for del 2 av TDT4145

- Kunne forklare om databasesystemer
  - Lagring
  - Indekser
  - Queryutføring
  - Transaksjoner
- Ha evne til å utnytte databasesystemer effektivt og pålitelig
- Kunnskap og evne til å se å se databasedesign i sammenheng med programvareutvikling

Hvorfor lære om DB-«internals»?

- For å kunne bruke databaser bedre
  - Indekser
  - Queries
  - Transaksjoner
- For å lage en komplett forståelse av databaser
- For de som ønsker å jobbe mer med databaser

Database-arkitektur



Databasetjener vs. Innebygd DB

- Vanligvis er et databasesystem på egen tjener(server) (datamaskin) som kontaktes via et nettverksgrensesnitt
- Eksempel: Programmering i Java med JDBC mot MySQL
- Noen databaser er innebygde (embedded) i applikasjonen via bibliotekskall.
- Eksempel: Programmering i Python mot SQLite3.
- Fordelen med tjenerutgaven er at mange applikasjoner kan dele databasen, og derfor får databasen mye større «verdi» enn en innebygd database.

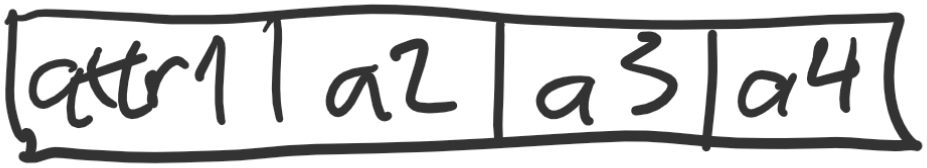
Databaselagring

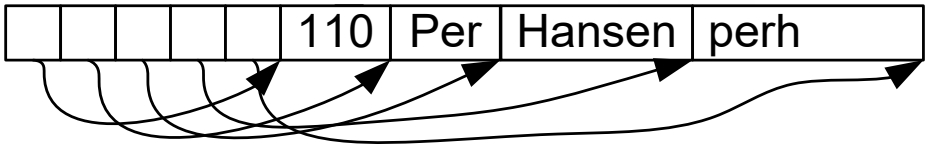
- Databaser lagres i filer eller på «raw devices»
  - Mest vanlig med filer, Direkte I/O på filer
  - «raw devices» unngår operativsystemets buffer
  - Noen bruker MMAP-segmenter
- Lagring av tabeller
  - Heapfil
  - B+-trær
  - Hashfil
  - LSM-trær
- Lagring av indeksfiler (på attributter)
  - For å få rask tilgang til data i tabellene
  - Eller tvinge gjennom PRIMARY KEY / UNIQUE-restriksjoner
    - Hashing
    - B+-trær
    - R-trær (flere dimensjoner)
  - For sortering av nøkler
- Kan kombinere lagring og indekser: Clustered index.

Lagring av poster

- En rad i en tabell er vanligvis lagret som en post (record) i en fil: tuppel vs. rad vs. post.
- En post har felter med navn og datatype
  - Integer, long integer, floating point (4 og 8 byte)
  - String (fastlengde og variable lengde) (1 eller 2 byte per tegn). Kalles også TEXT.
  - Date/time
  - Blobs – lange felter
  - JSON (tekst)
- SQL-dictionary (Catalog) beskriver hvordan en tabell/post er lagret.
  - Lengde og datatyper
  - + mye annet

Postlayout – 2 metoder





Blocklayout

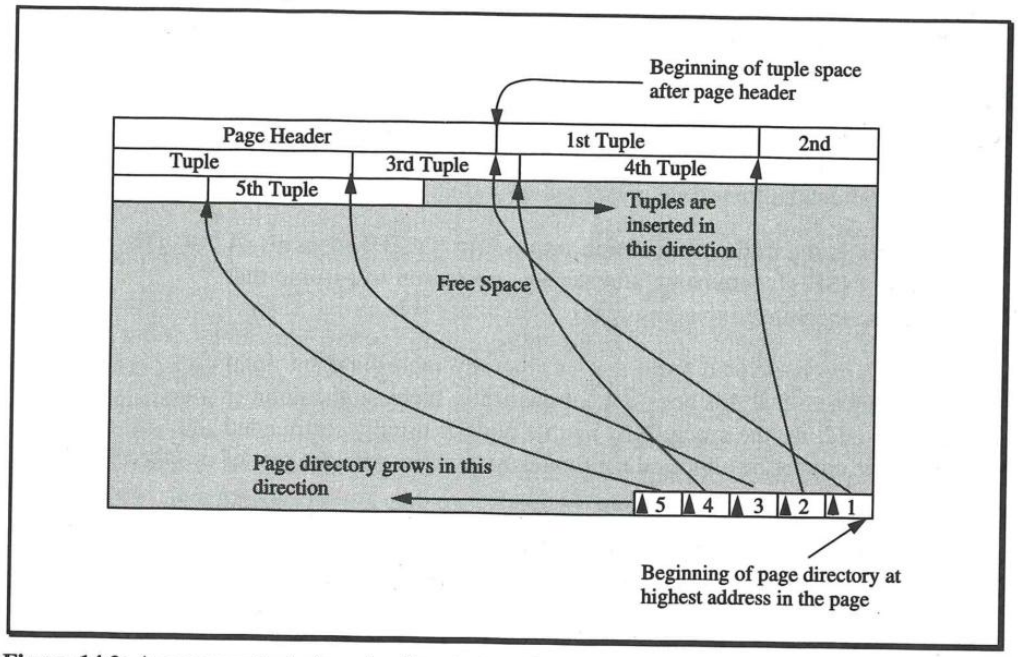


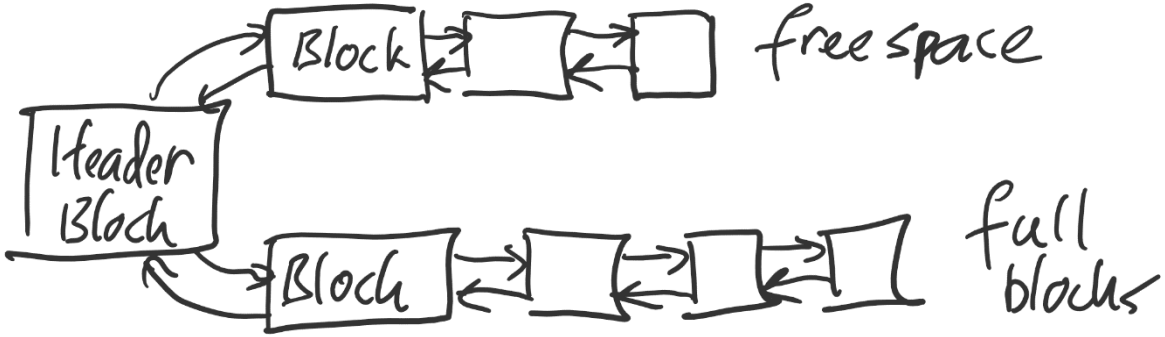
Figure 14.2: A common technique to allocate two dynamic data

Buffer

- Kopi i RAM av ofte brukte blokker fra databasen
- Databaser vil gjerne ha kontroll på dataene i RAM, så de «pinnes» slik at «virtual memory» ikke kaster de ut.
- Adaptive cache-algoritmer, med flere klasser av aksessmønstre
- Kan også støtte «pre-fetching» av blokker.
- Det brukes som regel en hashbasert inngang til buffer basert på BlockId.
- Blokker som tilhører samme hashinngang lenkes sammen i RAM.
- Blokker skrives til disk som en del av sjekkpunkting i forbindelse med logging og recovery. Evt. helt uavhengig av sjekkpunkting.

Heapfiler

- «Rått» og usortert lager av poster
- Poster settes inn på slutten av filen. Kan være to lister.



- Aksesseres med en **RecordId** (BlockId, nr. innen blokk)
- Vanligvis har man indekser i tillegg til heapfiler
- + lett å sette inn posten
- + god til tabellscan
- + bra skriveytelse
- - dårlig til søk på attributter og rangesøk (verdiområdesøk)

Quiz 1 A

- Hva er en komponent i et databasesystem?
  - Optimalisator
  - Filsystem
  - L3 Cache
  - Logg
- Hvorfor har vi indekser?
  - For å få oversikt
  - For å tvinge gjennom UNIQUE
  - Holde orden på databasen
  - For å få queries til å gå raskere

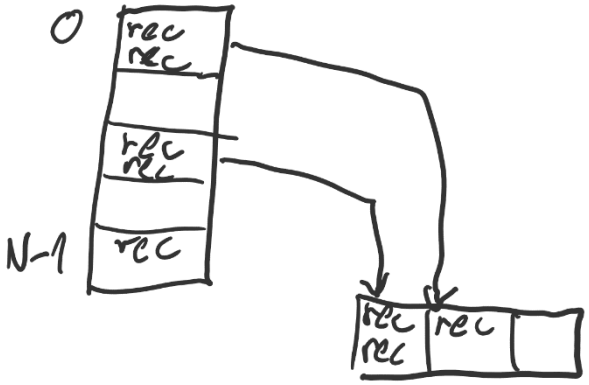
Quiz 1 B

- Hva er en SQL Dictionary?
  - Forklaring på Keywords i SQL
  - Oversetter SQL til algebra
  - Oversikt over tabeller og attributter
  - Cacher queryresultater
- Hvorfor bruker man Heapfiler?
  - Raske å sette inn poster
  - Gode til tabellscan
  - Raskt å finne igjen en post
  - Gode til å randomisere poster

Hashbaserte indekser

- Bra for direkte aksess på søkenøkkel
- h(K) – hashfunksjon av søkenøkkel
  - Sprer postene bra utover slik at de lett kan gjenfinnes
- F.eks. h(K) = K MOD M (restfunksjon)
- Mange mulige hashfunksjoner
- En god hashfunksjon har god spredning, men er avhengig av hva som skal spres
- Hvordan håndtere overflyt
  - Åpen adressering: lagre posten i første ledige etterfølgende blokk i fila. Kan også lenke her.
  - Separat overløp: lenk sammen overløpsblokker
  - Multippel hashing: bruk en ny hashfunksjon når det blir kollisjoner

Statisk hashing (1)



Statisk hashing (2)

- Partition by key i MySQL
- CREATE TABLE tm1 (

  s1 CHAR(32) PRIMARY KEY

)

  PARTITION BY KEY(s1)

  PARTITIONS 10;
- Fast antall blokker/partisjoner i samme fil
- Må bruke overløp for dynamiske datamengder (ukjent antall poster)
- Lange overløp kan ødelegge ytelsen

Separat vs. Intern overflyt

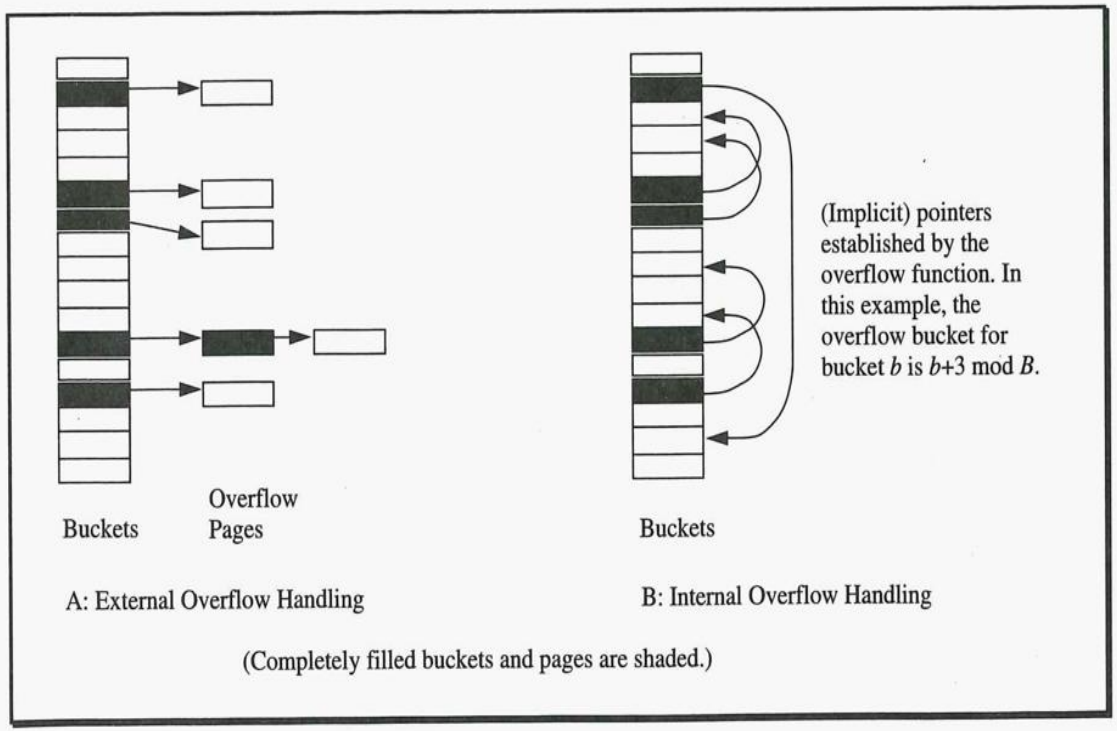


Figure 15.4: External versus internal methods for overflow handling in hash file organizations. External collision handling chains overflow pages behind those buckets that have overflowed from the primary area. This can be viewed as extending the size of a bucket as needed by assigning more pages to it. This decreases the overall space utilization whenever an overflow page is allocated. Internal overflow handling only uses the original buckets in the primary area. In case of a bucket overflow, a search strategy is used to determine another bucket that can accommodate the tuple from the overflowed bucket. This keeps space utilization high, but it increases the collision probability of those buckets, which—in addition to their "own" tuples—have to keep overflow tuples from other buckets.

Extendible hashing

- Problem ved statisk hashing.
  - Utvidelse av filen, dvs. dobling av antall blokker
  - Les alle blokker og skriv alle blokker på nytt.
  - Rehash alle poster
- Extendible hashing
  - Bruk katalog med pekere til blokker og doble katalogen ved behov
  - Splitt (les og skriv) kun den blokken som ble full
  - Lokal og global dybde
  - Hvis en blokk er full og lokal dybde == global dybde: Directory doubling

Hvorfor bruke indekser?

- For å gjøre queries raskere
  - **Student** (pnr, studnr, navn, adresse, epost)
  - SELECT navn, adresse FROM Student WHERE studnr=123456;

  - **Film** (id, fnavn, år, selskap, nasjonalitet, score)
  - SELECT fnavn, år FROM Film WHERE score > 7.0;
- For å tvinge gjennom UNIQUE og PRIMARY KEY-restriksjoner
  - **Student** (pnr, studnr, navn, adresse, epost)

Begreper innen indeksering

- **Indeksfelt**: Felt / attributt av posten som indeksen bruker
- **Primærindeks**: Indeks på primærnøkkelen
- **Clustered indeks**: Indeks på en tabell hvor postene er fysisk lagret sammen med (i) indeksen
- **Sekundærindeks**: Ekstra indeks på et annet felt hvor det også finnes en primærindeks.
- Sekundærindeksen kan være brukt for å tvinge gjennom UNIQUE, dvs. en unik verdi for hver post i tabellen
  - Student (pnr, studnr, navn, adresse, epost)

Lagrings- og indekseringsmuligheter (1)

- Systemspesifikt, se dokumentasjon av «ditt» system
- Clustered B+-tree / clustered index
  - B+-tre på primærnøkkel
  - Løvnivå av treet lagrer selve posten
  - MySQL: InnoDB
  - SQL Server: Clustered index når primærnøkkel er definert
- Heapfil og B+-tre
  - Tabell lagret i heapfil
  - B+-tre på primærnøkkel. Postene blir da (key, RecordID)
  - Evt. sekundærindeks på et annet felt
  - MySQL: MyISAM
  - SQL Server: Heap + unclustered index

Lagrings- og indekseringsmuligheter (2)

- Heapfil
  - Postene lagres forløpende uten noen annen organisering
  - SQL Server: Hvis primærnøkkel ikke er definert
- Clustered hash index
  - Hashindeks på primærnøkkel
  - Posten lagret i indeksen
  - Oracle: Hash cluster
- LSM-trees (log-structured merge trees)
  - Moderne lagrings- og indekseringsmetode for Big Data
  - «Cacher» de nyeste innsatte/oppdaterte postene.
  - Høy skriveytelse, lav «write amplification», bedre komprimering
  - Eldre poster flyttes over i «langtidslager» (flere nivåer)
  - Sqlite3 (utvidelse), NoSQL, RocksDB, MySQL/myRocks, Apache Hbase

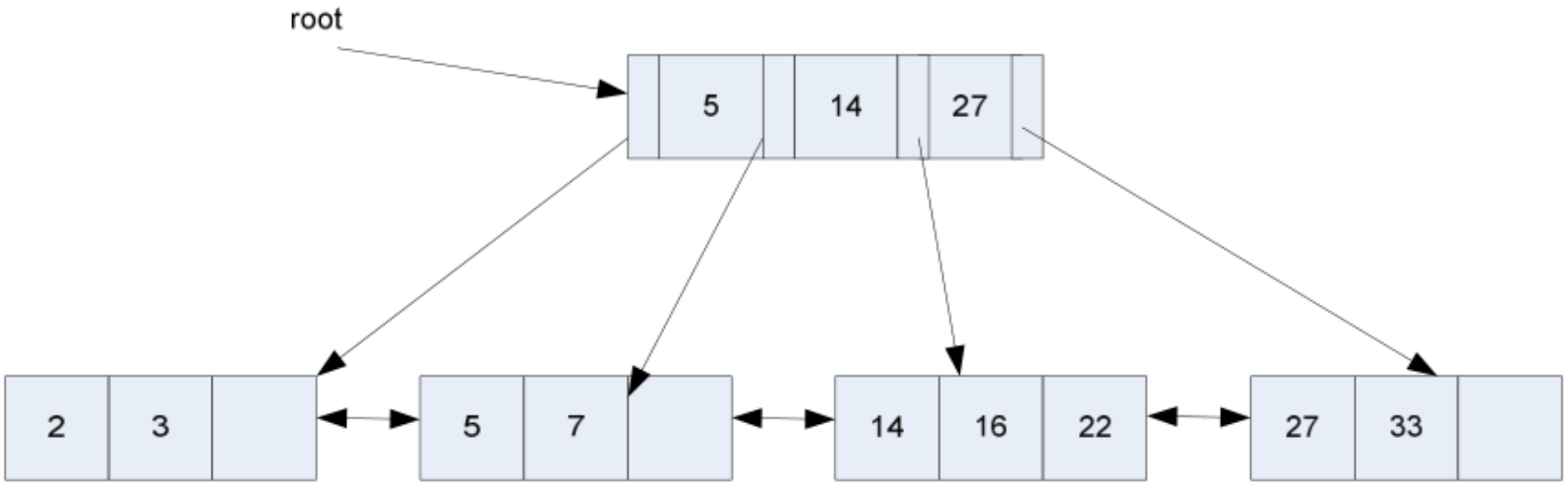
Lagrings- og indekseringsmuligheter (3)

- Column stores
  - Tradisjonelt lagres tabellens rader i SQL-databaser
  - Analyseapplikasjoner / datavarehus vil oppleve bedre ytelse med kolonnebasert lagring
  - Leser mindre data ved queries og kan bruke komprimering
  - Kan komprimere på flere måter
  - SELECT MAX(score) FROM Film;
  - SQL Server: Columnstore index + *delta store* for å samle nok oppdateringer til å flette sammen med kolonnen.
  - Apache Kudu (Hadoop platform), C-store/VoltDB
- AI-genererte indekser (f.eks. Recursive Model Indexes)
  - Indekser laget basert på maskinlæring
  - Svært effektive på read-only data, men sliter med oppdateringer

B+-trær

- Den mest brukte indeksen
- Høydebalansert tre med blokker som noder
- Alle «brukerposter» er på løvnivå («nederst»)
- Typisk høyde: 2, 3 eller 4.
- Minimum 50 % fyllgrad i blokker
- Gjennomsnittlig 67 % fyllgrad i blokker
- Postene er sortert på nøkkelen, og treet støtter da
  - Likhetssøk (direktesøk)
  - Verdiområdesøk
  - Sekvensielle, sorterte skan
  - Gode på det meste, også for dynamiske datamengder

B+-tre, eksempel



B+-trær i praksis

- Typisk fanout: 133
- Typisk fyllgrad: 67 %
- Typisk kapasistet (non-clustered B+-tree)
  - Høyde 4 : 133^4 = 312 900 700 poster
  - Høyde 3 : 133^3 = 2 352 637 poster
- Clustered B+-tre
  - Høyde 4: 133*133*133*20 = 47 millioner poster
- I praksis er de øverste nivåene alltid i buffer (RAM)
  - Level 3 — 1 blokk — 8 KiB
  - Level 2 — 133 blokker — 1 MiB
  - Level 1 — 17 689 blokker — 133 MiB
  - Level 0 — 2,3 mill blokker — 18 GiB

Poster i B+-trær

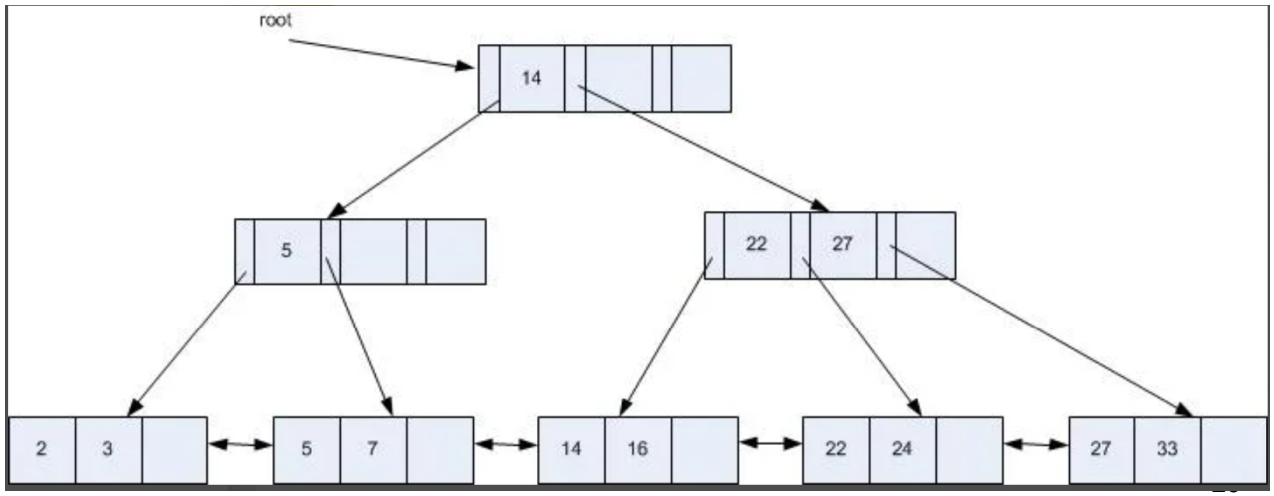
- Tabell: Student (pnr, studnr, navn, adresse, epost)
- Clustered B+-tree
- Løvnivå (level=0):
  - Hver post på løvnivå vil se slik
    ('010195 12345', 123456, 'Hans Hansen', 'Revekroken 1',
    'hans@stud.ntnu.no')
  - Hver blokk på løvnivå kan inneholde ca. 50 poster (avhengig av blokkstørrelse)
- Level > 0 (indeksnivå):
  - ('020194 23456', BlockId)
  - Hver blokk kan inneholde ca. 200 poster (antar indekspost er ¼ av en vanlig post)

Quiz 1

- Hvorfor brukes B+-trær?
  - De finnes overalt
  - Superraske til å søke på mobilnr
  - Gode på det meste
  - Passer veldig bra til moderne cachearkitektur
- Hvordan lagre: Det søkes etter primærnøkkel og hele posten trengs?
  - Heapfil
  - Hashfil
  - Clustered B+-tre
  - Clustered Hashfil

Quiz 2

- Et B+-tre har kun ett nivå, hvor mangel blokker i treet?
  - 1
  - 10
  - 100
  - 1000
- Det søkes etter nøkkel 15. Hvor mange blokker aksesseres?
  - 0
  - 1
  - 2
  - 3



Blokksplitting i B+-tre

- Splitting vanligvis ved midterste post
- Størrelsesmessig midterste post ved variabel lengde poster
- Maks poststørrelse kan være ½ blokk, men spesialløsninger for virkelig lange poster (BLOBs)
- Indeksposter (level > 0) er små: nøkkel + BlockId
- Løvnodeposter kan være større. Hvorfor?

Indekser på sammensatte nøkler

- Employee (ssn, dno, age, street, zip, salary, skill)
- SELECT * FROM Employee WHERE dno=4 and age>50;
- Hvilke indekser kan hjelpe her?
  - Indeks på dno: finn alle poster med dno=4 og sjekk om age > 50
  - Indeks på age: scan indeksen fra 50 og finn alle poster med dno=4.
  - Sammensatt indeks på
    - (age, dno)
    - (dno, age)
    - Bruk den som er mest selektiv først, altså den som gir færrest poster i resultatet
  - Indekspostene har leksikalsk sortering

Fyllgrad i blokker, regning

- Regneregler for fyllgrad ved innsetting
- Statisk 2/3 fyllgrad eller 0.67 desimalt
- Ved innsetting i flere blokker regner vi at en blokk aldri fylles mer enn 2/3
- Eksempel: 4096 bytes (4KiB) blokker
- Poster på 120 byte (recSize)
- nRecs = floor((4096 * 2/3)/recSize) = 22.
- 22 poster fyller blokka.
- 10000 poster i en tabell og blokker på 4KiB
- B-tre: ceiling(10000/22) = 455 blokker på level 0.
- 455 poster på level=1. Hvor mange blokker på level=1?

LSM-trær, laget for BigData