Problem 1

- OS-prosess per arbeider
  Fordeler:
  - OS-et gjør scheduling
  - Privat adresserom
  - OS-et beskytter tråder fra hverandre (isolasjon)
  Ulemper:
  - "Tungvekt"
  - Ekstra arbeid for å synkronisere arbeidere
  - Varierer mellom OS
- OS-tråd per arbeider
  Fordeler:
  - OS-et gjør scheduling
  - Delt adreserom og andre ressurser mellom tråder
  - Mer lettvekt enn prosesser
  Ulemper:
  - Delt minne gir mindre isolasjon (ingen beskyttelse mot andre tråder)
- Lettvektstråd per arbeider
  Fordeler:
  - Effektivt
  - Delt adresserom, felles ressurser
  Ulemper:
  - Komplisert å implementere
  - Ingen OS-støtte

Problem 2

(Admission)
Parsing (inkl. lexing/scanning)
Resolving
- Oppslag av navn (kolonner, tabeller, views, funksjoner, etc.)
- Privilegiesjekk (grants)

(Rewrite) (kan inkluderes i optimalisering)
- Regelbasert (heuristisk omskriving)
- Kanonisk format

Optimalisering
- Kostnadsbasert optimalisering
- Valg av join-rekkefølge
- Valg av aksessmetoder

(Compiling) (hvis det brukes kompilert kode)
Eksekvering
- Typisk Volcano-iteratormodell (FetchNext)
- Kan også kjøre kompilert kode

Problem 3

SARG = Search ARGument. Et predikat som kan brukes som search argument til å gjøre indeksoppslag, dvs. at uttrykket kan brukes til å slå opp i en indeks. Etter å ha dyttet predikater så langt ned som mulig, vil sargable predicates ligge rett over table scan, og table scan + sargable predicate kan da gjøres om til indeksoppslag.

Problem 4

En interessant rekkefølge er en rekkefølge som kan være nyttig underveis i eksekveringen eller for å gi resultatet tilbake i den spesifiserte rekkefølgen. Hvis en interessant rekkefølge kommer automatisk fra en aksessmetode eller en annen intern operator, kan dette utnyttes til å slippe å eksplisitt sortere, noe som er en blokkerende operasjon. Det kan spare eksekveringstid, og sjansen øker for at eksekveringen kan pipelines. Typiske interessante rekkefølger: 1) sortering på kolonnene i GROUP BY kan brukes for å kunne gruppere én og én gruppe uten avbrudd, 2) sortering på join-kolonner kan brukes for å velge merge join, 3) sortering på kolonner i ORDER BY kan brukes for å returnere resultatet av spørringen.

Problem 5

Leis et al. (2015) argumenterer med og viser ved eksperimenter at kardinalitet er viktiere enn en god kostnadsmodell. En god kostnadsmodell er ikke uviktig, men eksperimentene viser at gode kardinalitetsestimater betyr mer. Kardinalitetsestimater er viktige for valg av join-rekkefølge. Feilaktige estimater fører til større og større feil jo mer kompleks spørringen blir siden estimatene multipliseres samen og feilene dermed øker. En god kostnadsmodell kan ikke kompensere for at utgangspunktet er feil, så derfor blir det viktigere at utgangspunktet (kostnadsestimatene) er gode.

Problem 6

Iflg. Leis et al. (2017): Det er en tendens til at kardinaliteter underestimeres. Sampling finner et mer nøyaktig estimat, og dette blir da som regel høyere. Hvis man da sampler i høyden først, og bygger noen 3-veis, 4-veis osv. estimater, mens andre 2-veis estimater ennå ikke er samplet, vil de sampling-baserte estimatene ofte være større enn de ikke-samplede, og optimizeren vil foretrekke de ikke-samplede, selv om de er mindre nøyaktige. Dette kalles å flykte fra kunnskap.

Problem 7

Bakgrunnen for LSM-trær er å redusere lese-ytelse for å øke skriveytelse. Likevel er LSM-trær fortsatt en trestruktur som kan gjøre effektive lese-operasjoner, men ikke like effektive som f.eks. et B-tre. Men skriving er mer optimalisert siden det brukes sekvensiell skriving i stedet for tilfeldig aksess.

Problem 8

Interleaving er å lagre rader fra forskjellige tabeller sammen basert på fremmednøkkel. F.eks. kan alle rader for bilder som tilhører en person lagres etter raden for denne personen. Ved å bruke interleaving lagrer Spanner data som ofte skal joines nær hverandre. Spanner kan da utføre join lokalt på hver node og ved å lese sekvensielle diskblokker.

Problem 9

a)

Her kan det være flere løsninger. Argumentasjonen er viktig.

Tilgjengelighet er viktig. Skriving og lesing foregår i hver sine faser. Hvis vi lagrer hver stemme som en rad (i stedet for å oppdatere en teller for hver deltaker), kan vi lagre stemmene helt uavhengig av hverandre, og vi kan da velge lazy update anywhere. Lazy gjør at vi ikke treger å sykronisere, så det er lite overhead for hver operasjon. Update anywhere gjør at vi kan skrive på alle noder, så vi kan spre skriveoperasjonene på mange noder og unngå flaskehalser. Selve opptellingen skjer etter at alle stemmene er avgitt, så det vil ikke være konflikt mellom skriving og lesing.

b)

Svaret kan variere med løsningen valgt i a).

Vi har prioritert tilgjengelighet (A). Vi har designet systemet slik at det ikke trenger tilby et konsistent bilde (C) underveis i avstemmingen. Et konsistent bilde er kun nødvendig etter at avstemmingen er over og resultatet skal telles opp, men da er det ingen annen last på systemet.

Problem 10

Her er det mange punkter, ikke alle trenger være med.

Horisontal skalerbarhet
 - Skalerbarheten kan ofte være høyere enn i RDBMS

Høy tilgjengelighet
 - Velger gjerne AP over CP
 - Fordelen over RDBMS er at systemet er mer tilgjengelig

Svak konsistensgaranti (weak consistency)
 - Ulempe at man ikke kan gi samme konsistenskrav som RDBMS

Horisontal sharding

Begrenset cross-shard-funksjonalitet
 - Mer ansvar på applikasjonen for å gjøre f.eks. JOIN enn i RDBMS

Enkel datamodell
 - RDBMS-er har en fordel ved å støtte representasjon av komplekse data

Skjemaløst
 - Raskere å utvikle mot skjemaløs modell
 - Skjema i RDBMS hindrer datamodellen i å "gli ut" og bli inkonsistent

Programmeringsspråknær datamodell
 - En fordel om man bruker riktig språk, naturlig for utvikleren
 - Kan bli vanskelig dersom man bruker flere språk
 - RDBMS bruker gjerne SQL uansett prog.språk

Denormalisert datamodell
 - Fører til duplisering av data, RDBMS forsøker å unngå dette

Mindre avansert spørrespråk
 - RDBMS kan gjøre mer avanserte spørringer med SQL
 - Enklere for databasesystemet å implementere et enkelt språk
 - Ansvaret for kompliserte spørringer dyttes mer over på applikasjonsutviklerne enn med RDBMS/SQL

Ingen transaksjoner
 - BASE, ikke ACID
 - ACID i RDBMS gir applikasjonsutviklerne stabile og enkle rammer å forholde seg til
 - BASE er enklere å implementere i databasesystemet enn ACID

Problem 11

Duplisering er at ett dataelement, f.eks. et polygon, lagres på flere steder i en indeks. Dette fører til at indeksen vokser i størrelse, og ved en endring må indeksen oppdateres på flere steder. Dersom duplisering er mulig må også søk gå gjennom alle steder hvor et duplikat kan være lagret. Duplikater må filtreres ut før resultatet returneres.

Problem 12

Space driven: Vi gjør inndeling/splitting basert på spennet på koordinatsystemet de geometriske objektene er i. Det defineres grenser for koordinatene på forhånd, og indeksen forholder seg til disse grensene når det gjøres inndeling.

Data driven: Inndeling skjer etter dataene som kommer inn og tilpasser seg den delen av koordinatsystemet som faktisk er i bruk.