1. ANSI-SPARC architecture:
    a. Conceptual structure (schema) - just 1
    b. External structure**s** (ex. views) - there can be several
    c. Physical (internal) structure (data files, indexes etc.) - just 1 => why we can have just one clustered index
    d. No such thing as a *symbolic* structure
2. **Data independence** -
    a. Logical - apps + external schemas are not affected by changes in the conceptual structure
    b. Physical - apps + ext sch are not... physical structure
3. From the simplest to the most complex: field, record, table, database
4. Data description model can be used to describe: the structures of the data, relationships with other data, consistency constraints
5. The rows in a relational table are not ordered, the records are distinct - but DBMSs allow duplicates
6. Degree of a relationship set = the no. of entity sets the participate in that rel
    a. The degree of a projection < .. of the original table
7. **Optimizer** - produces an efficient execution plan for query eval, based on storage information
8. Multiple candidate keys can be declared using UNIQUE; one of them can be chosen as the primary key
9. We can have a multiple FK that references a multiple PK
10. FK can be null
11. An att can be a PK and FK at the same time
12. **Key** = a restriction defined on an entity set; a set of attributes with distinct values in the entity's set instances
13. **Integrity constraints** = conditions specified on the DB schema, restricting the data that can be stored in the DB (includes key constraints, FK constraints)
    a. Key constraint = a constraint stating that a minimal subset of atts in a relation is an unique identifier for every tuple in the relation
    b. **Superkey** = set of fields that contains the key
14. **3-valued logic** (true, false, unknown = null)
15. "expression = ANY (subquery)" ⇔ "expression IN (subquery)"
16. "Expression <> → different ALL (subquery)" ⇔ "expression NOT IN (subquery)"
17. Copy data from one table to another: "INSERT INTO T2; SELECT * FROM T1"
18. SELECT can contain arithmetic operations
19. SELECT, FROM - mandatory; HAVING - optional
20. HAVING cannot contain row-level conditions because it works with groups

21. Patterns: "%" - 0 or more arbitrary characters, "_" - any one character (used with LIKE)
22. Can't group by a subquery
23. SELECT **R.*** ⇔ select all the columns in R, ex.

    SELECT **R.***
    FROM R
    RIGHT JOIN whatever ON whatever

24. LEFT JOIN = inner join + the left operands which don't match the condition (their columns are filled with NULL); same idea for RIGHT JOIN
25. If we have a FROM query, we need to give it an alias
26. Processing order: from, where, group by, having, select, distinct, order by, top
27. **Functional dependency**: left side = **determinant**; right side = dependent
28. Functional dep properties:
    a. K - key of R, β - key of a subset of R => K -> β
    b. $\beta \subseteq \alpha$ => α -> β (trivial func. dep., reflexivity)
    c. α -> β => γ -> β, for any γ $\boxed{\alpha \subset \gamma}$
    d. a -> b and b -> c => a -> c (transitivity)
    e. a -> b => a u c -> b u c (augmentation)
29. **Prime attribute** if it's included in a key K; non-prime otherwise
30. B is **fully functionally dependent** on A if **A -> B** and **B doesn't depend on any proper subset of A** (B cannot be derived from a smaller part of A)
31. **Multivalue dependencies** - each value from the determinant is associated with a set of values, not with just one, denoted by "-> ->" "$\twoheadrightarrow$"
32. **Join dependency** (JD) - we project an R[A] onto R1[α1] … Rn[αn] => R satisfies the JD {α1, …, αn} if R = R1 * … * Rn
33. **Normal forms**:
    a. 1NF ⊂ 2NF ⊂ 3NF ⊂ BCNF ⊂ 4NF ⊂ 5NF
        i. If *not* 1NF => not 2,3..NF; if not 2NF => not 3,BC..NF and so on
    b. 1NF = no repeating attributes
    c. 2NF = 1NF + **every** non-prime att is fully functionally dependent on every key of the relation
    d. 3NF if for any non-trivial X -> A, X is a superkey or A is a prime att
    e. (Boyce-Codd) BCNF if every determinant (left side of a func dep) is a key
        i. Most desirable form
    f. 4NF if for every multi-valued dep a ->-> b, either one is true
        i. b ⊆ a or $\underbrace{\text{a U b}}_{\text{cover all}}$ = R

ii. a - superkey

g. 5NF = every non-trivial join dependency (JD) is implied by the candidate keys in R

i. A JD *{α1, …, αn} is trivial if any αi is the set of all atts of R

ii. … is implied by the candidate keys in R if ALL αi are superkeys in R

34. **Armstrong's axioms**: reflexivity, augmentation, transitivity

35. Rules derived from AA

a. a -> b and a -> c => a -> b u c (union)

b. a -> b u c => a -> b and a -> c (decomposition)

c. a -> b and b u c -> d => a u c -> d (pseudotransitivity)

36. α+ = closure of α under a set of fds

37. **Example: Show that some fds are in α+**

We have a relation R = {a … f} and α = {a -> b, a -> c, cd -> e, cd -> f, d -> e}. Show that the following fds are in α+

a -> *bc*: union of a -> b and a -> c

cd -> *ef*: union of cd -> e and cd -> f

ad -> *e*: a -> c / augment with d => ad -> cd; from cd -> e through transitivity => ad -> e

ad -> *f*: pseudotransitivity a -> c and cd -> f or same as before, augment first

38. Algorithm for computing α+

```
closure := α;
repeat until there is no change:
  for every functional dependency β → γ in F
    if β ⊆ closure
      then closure := closure ∪ γ;
```

39. **Example: Compute α+**

R = {a … f} and F = {a -> b, a -> c, cd -> e, cd -> f, d -> e}

α = {a, d} (given)

α+ = {a, d}

a -> b => α+ = {a, b, d}

a -> c => α+ = {a, b, c, d}

cd -> e => α+ = {a, b, c, d, e} (because both c, d were in α+)

cd -> f => α+ = {a, b, c, d, e, f} (same reason)

d -> e => we don't add it because E already is in α => no change at this step => stop => α+ = R

40. F, G - sets of fds => F ≡ G (equivalent) if F+ = G+
41. **Minimal cover** Fm if
    a. Fm ≡ F
    b. Right side of every dependency in Fm has a single att
    c. Left side .. is irreductible (if we remove an att from it, we modify Fm's closure)
    d. No dependency in Fm is redundant (if we remove it, we modify the closure)
42. **Example: compute Fm**

R = {a … d}; F = {a -> bc, b -> c, a -> b, ab -> c, ac -> d}
Use decomposition on a -> bc => a -> c, a -> b (which already was in, so we don't add it again) => **{a->b, a->c, b->c, ab->c, ac->d}**
a -> c / augment with a => aa -> ac => a -> ac; we know ac -> d => a -> d (transitivity); now ac->d becomes redundant, we replace it with a -> d => **{a->b, a->c, b->c, ab->c, a->d}**
a -> c / augment with b => ab -> cb => (decomposition) ab -> c (it's already in, we can remove it) => **{a->b, a->c, b->c, a->d}**
b -> c can be obtained as a -> b and b -> c (transitivity) => remove b -> c
Fm = **{a->b, a->c, a->d}**

43. Just by looking at an instance, we can **only** say that a func. dep is **satisfied** or not, we can't conclude that it's **specified on the schema**
44. Algebra symbols
    a. $\sigma_C(R)$ = select * from R where C; distributive over set diff, intersection, union
    b. $\pi_\alpha(R)$ = projection; select (distinct?) α from R (α can specify multiple columns); it's distributive only over **union**
    c. R1 X R2 = select * from R1 cross join R2 (cross product); the schema will contain all the atts of R1 followed by the atts of R2
    d. R1 U R2, R1 - R2, R1 ∩ R2 = union, set diff, intersection
    e. R1 $\otimes_\Theta$R2 = .. inner join on Θ (condition join)
    f. R1 * R2 = natural join; schema will contain the union of the atts (= all atts, common ones appear **only once**); returns 1 relation instance

g. R1 $\ltimes_c$ R2 = left (outer) join on c; the reversed symbol is right (outer) join..

h. R1 $\bowtie_c$ R2 = full (outer) join on c; schema will contain all the atts of R1 followed by the atts of R2

i. R1 $\triangleright$ R2 = left semi join; the reverse is right semi join

   i. Left.. = the tuples in R1 that are used in the natural join; right.. = ..R2

j. R1 $\div$ R2 = division

k. δ = duplicate elimination

l. S $_{\{list\}}$ (R) = sort

m. $\gamma_{\{list1\}\ group\ by\ \{list2\}}$ (R)

n. ρ = renaming operator (nush ce plm e cu asta)

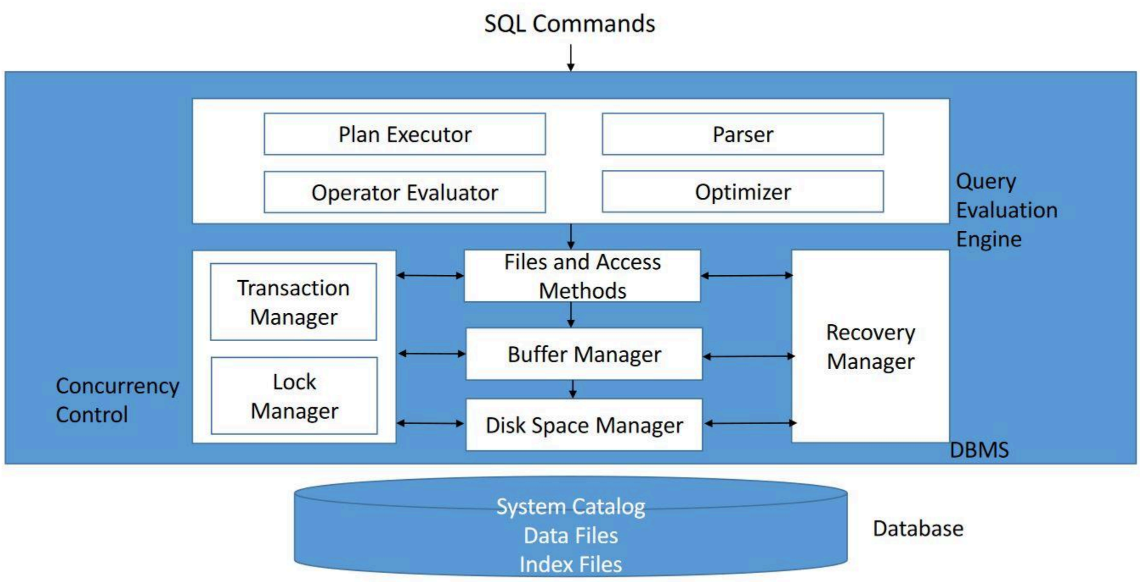
• for the previously described query language, with operators:
$\{\sigma, \pi, \times, \cup, -, \cap, \otimes, *, \ltimes, \rtimes, \bowtie, \triangleright, \triangleleft, \div\}$
an independent set of operators is $\{\sigma, \pi, \times, \cup, -\}$

lecture 7

45. All the operators (without duplicate elim, sort, group by) can be obtained from $\{\sigma, \pi, \times, \cup, -\}$ (= *independent set of operators*) (see l7, p3 on how to do it)

46. If there are multiple triggers defined for the same action, they are executed in a random order

47. DBMS structure



48. DBMSs operate on data when it is in memory

49. **Seek time fmm Carina** = time required to move the disk head to the desired track (smaller platter size => decreased seek time)

50. **Disk access time** (magnetic disk) = seek time + rotational delay + transfer time

51. **Block** = unit of data transfer between disk and main memory

52. **DSM** = disk space manager; monitors disk usage, it commands to (de)allocate, read / write a page

53. **BM = buffer manager**; brings new data pages from disk to main memory as they are required, manages the available main memory
    a. **Replacement policies** - LRU (least recently used), MRU, random, clock replacement, toss-immediate
    b. **Buffer pool** (BP) is made of frames, which can fit a page
    c. Each frame has
        i. *pin_count* = no. of current users on the crt frame; only those with 0 can be replaced; initial value = 0
        ii. *dirty* = true if the page has been changed since it was brought to the frame; initially off
    d. When a page is requested
        i. If the page is in the BP, increase its pin count => done
        ii. Else, select a frame (FR) to replace using the replacement policies; if the old frame was dirty, write it on the disk; pin_count(FR)++, the new page is read by the BM in that frame
        iii. If the BP is full, the operation may be aborted, or it waits
    e. It may pre-fetch pages

54. **Heap files**
    a. Simplest file structure, unordered records, best when we need to scan all the records
    b. Can use
        i. Doubly-linked list - it needs a header = stored by the dbms; it has 2 lists, for full / free pages
        ii. Directory of pages

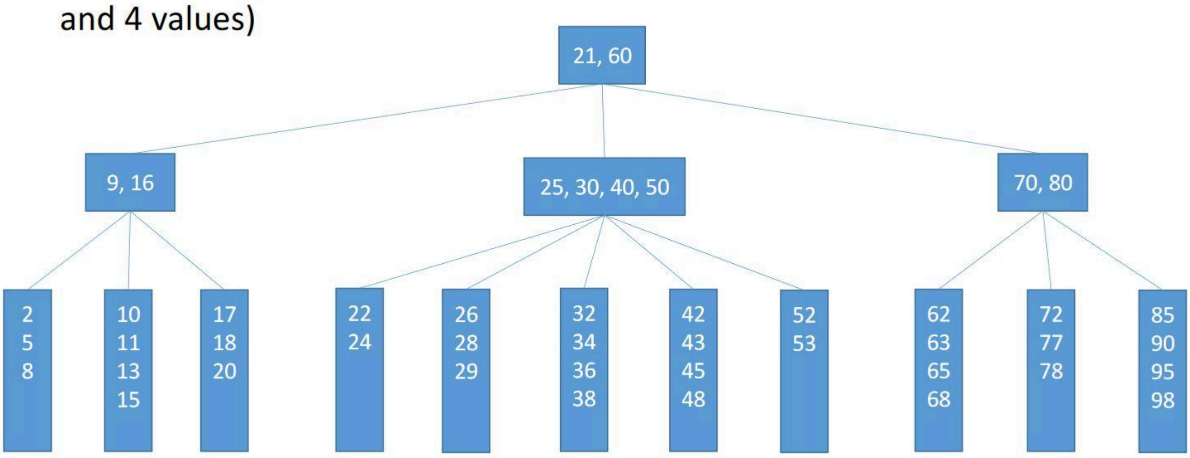
55. **Index**
    a. Stored on the disk, associated with a table or view
    b. Speeds up equality / range select queries on the search key
    c. If we change the data => the indexes need to be updated
    d. hashed files - very good when searching for equality
    e. **Data entry is:**
        i. A1 - the actual data record with search key value = k
        ii. A2 - <k, rid>, rid = id of a data record with sk = k
        iii. A3 - <k, list of rid> …
        iv. In general, a2, a3 are smaller than a1
    f. **Clustered**
        i. A1 - always clustered

ii. the order of the data records is close to / same as the order of the data entries
iii. Includes all the columns in a table
iv. At most 1 / collection of records
v. When creating a PK, if there’s no clus. on that table (and we don’t specify unclustered) => create an unique cl. Index
vi. Organized as a B+ tree

g. **Unclustered**:
  i. A2, A3 - are clus. only if the data records are ordered on the search key, which is uncommon in practice => A2, A3 are usually unclus
h. **Primary index** - contains the PK; **unique index** = contains a candidate key
  i. **secondary** = not primary (in our terminology, but in other places this might mean that it’s just unclus); it can be unclus
i. Primary / unique can’t contain duplicates, secondary can
j. Composite SK - contains several keys
k. Covering index - contains all the columns that are necessary in a query
l. Filtered index - can be created with a WHERE => only used when it satisfies that condition

56. **ISAM** = Indexed Sequential Access Method
  a. Not very good when doing lots of inserts / deletes
  b. Search complexity: $\log_c m$ (c = children per index page, m = primary leaf pages)
57. **2-3 tree**
  a. All terminal nodes on the same level
  b. If a non-terminal node has 1 key => 2 subtrees: one with values < key, the other > key
  c. If .. 2 keys (key1 < key2) => 3 subtrees: first < key1 < second < key2 < third
  d. **Values in the tree are distinct**
58. **B-tree of order m**
  a. Generalization of 2-3 tree (= b-tree of order 3)
  b. All terminal nodes on the same level
  c. Every non-terminal, non-root node has between ceil(m / 2) and m **subtrees**
  d. A node with p subtrees has p-1 ordered **values**

- non-terminal, non-root node – at most 5, at least 3 subtrees (between 2 and 4 values)

e. 

59. B+ tree

a. Variant of the B-tree
b. Uniform search time; few I/O ops needed for searching; ideal for range selection
c. Much better than ISAM for insert / delete etc.

60. Hash-based indexing - ideal for equality selections

61. **Static hashing** - the classic one (SDA gen)

62. **Extendible hashing** - there's a *directory* which points to the buckets, each determined by the last k (= *depth)* pro digits of the values; when a bucket becomes full, we split it into 2 buckets determined by the last k + 1 digits and we double the size of the directory

63. The client generates SQL statements and sends them to the server, which syntactically analyses + evaluates them and sends the results back

64. **Table scan** - high transfer time for large tables (the slowest)

65. **Index seek** - when we search for a key value (using =) that we have indexed (the fastest)

66. **Index scan** - the search condition can depend on either a key or a non-key att; some mem. blocks may be read multiple times

67. Joins are used in practice more often than cross-products. The sooner we apply selections / projections in the join implementation, the better

68. Some algorithms for **cross-join**, (indexed) nested loop join, merge join, hash join (found [here]())

69. Algebra equivalences

a. $\sigma_C(\pi_\alpha(R)) = \pi_\alpha(\sigma_C(R))$ (second is more efficient)
b. $\sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C1 \text{ AND } C2}(R)$ (second - more eff because it only does one pass of R)

c. $\sigma_C(R \times S) = R \otimes_C S$ (condition join - more eff)

d. $$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$
$$\sigma_C(R \cap S) = \sigma_C(R) \cap \sigma_C(S)$$
$$\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$$
(if the schemas of R and S are compatible)

e. $\sigma_C(R \times S) = \sigma_C(R) \times S$ **(if C contains only atts from R)**

f. $\sigma_{C1 \text{ AND } C2}(R \times S) = \sigma_{C1}(R) \times \sigma_{C2}(S)$ **(if C1 contains only atts from R, C2 .. S)**

g. $\sigma_{C1 \text{ AND } C2}(R \times S) = \sigma_{C1}(R \otimes_{C2} S)$ **(if C2 is a join condition between R and S)**

h. $\pi_\alpha(R \cup S) = \pi_\alpha(R) \cup \pi_\alpha(S)$

i. $\pi_\alpha(R \otimes_C S) = \pi_\alpha(\pi_{\alpha 1}(R) \otimes_C \pi_{\alpha 2}(S))$ ($\alpha 1$ = atts from R that appear in α or C, $\alpha 2$ = … from S…)

70. Intersection and union are associative and commutative
71. Cross-product and natural join are associative
72. When using the cross-join algorithm, the order of the data sources is important
73. **Surrogate key** = key that isn't obtained from the domain of the modeled problem
74. **DB design stages**
    a. Requirement analysis
    b. Conceptual design
    c. Logical design
    d. Schema refinement (normalization, eliminate redundancy)
    e. Physical design (ex. indexes)
75. We can have tables that reference themselves (recursive relationship); if there we have ON DELETE CASCADE => error; same if we have two tables that reference each other
76. The M:M table is also called *join table*
77. Reduce fragmentation
    a. In a heap - create then drop a clus index

b. In an index - ALTER INDEX REORGANIZE (if small fragmentation), ALTER INDEX REBUILD (if large fragmentation) or drop + recreate index

78. **Indexed views**
    a. cannot reference other views
    b. the index must be clus + unique;
    c. only allowed aggregation operators are sum, count
    d. subqueries, outer joins, set operations etc are not allowed
    e.

SET options	required value	default server value
ANSI_NULLS	ON	ON
ANSI_PADDING	ON	ON
ANSI_WARNINGS	ON	ON
ARITHABORT	ON	ON
CONCAT_NULL_YIELDS_NULL	ON	ON
NUMERIC_ROUNDABORT	OFF	OFF
QUOTED_IDENTIFIER	ON	ON

79. WAITFOR TIME ‘4:37’ => execution continues at 4:37
80. Traditional dbms -> one-shot query (executed on the current instance of the data)
    a. Human active, DBMS passive model (HADP)
81. **Data stream** = temporal sequence of values produced by a data source, potentially infinite; data is associated with timestamps; we can have data **stream** management sys
82. Event = elementary unit of information on a data stream (the equivalent of a record)
83. Sliding window = contiguous portion of the data stream, it has a size and a step size
84. **Continuous query** - perpetually running, DAHP