Exam notes

Functional dependencies

> Trivial FD: $X \rightarrow Y$ is trivial if: $Y \subset X$.
> Non-trivial FD: *see above*.

> Superkey: Superset of a candidate key.

> Candidate key: Minimal set of a key.

> Primary key: Candidate key chosen by designer.

> Prime attributes: Attributes contained in candidate key.

> Closure of **FDs**: Set of all dependencies that include *F* and all those that can be inferred from *F*. Denoted as F^+.

> Closure of **set of attributes**: Maximum set of attributes that can be inferred from the provided set of attributes.

Normalization

Database design principle to

1. *reduce redundancy*
2. *avoid complexities*
3. *organize data in a consistent way.*
4. *eliminate duplicates*
   but loses out on

- query efficiency

> 1NF: No multivalued attributes, i.e. only simple values.

> 2NF: Parts of the candidate key must not be functional dependent on non-prime attributes.

- *Example:* $\{\mathbf{A}, \mathbf{B}\}: \overbrace{\mathbf{A} \rightarrow C, D, E}^{\text{illegal}}$
  $\Rightarrow$ No arrows from part of candidate key!

Decompose: *Make new table containing LHS and RHS. Remove RHS from origin table.*

> 3NF: Non-prime attributes must not be functionally determined by other non-prime attributes.

- *Equivalent to no transitive dependencies*: $\mathbf{A} \rightarrow \overbrace{B \rightarrow C}^{\text{illegal}}$
  $\Rightarrow$ No arrows starting from non-prime attributes!

Decompose: *Make new table containing LHS and RHS. Remove RHS from origin table.*

> BCNF: No dependencies from non-key attributes to key attributes.

- *Example*: $\{\mathbf{A}, \mathbf{B}\} \rightarrow \overbrace{C \rightarrow \mathbf{A}}^{\text{illegal}}$
  $\Rightarrow$ No arrows pointing towards prime attributes!

Decompose: *Remove one of the prime-attributes*!
NB: BCNF is lossless, but it is not dependency preserving!

> Losslessness: Decomposition is lossless if we can recover initial table by performing multiple joins.
> Dependency preservation: We do not lose dependencies. True if all dependencies can be inferred from the current set.

Triggers

- Can not modify the database schema; solely DML (*Data Manipulation Language*) for data-level operations.

> Invoked *automatically*.
> Defined in terms of the event *that invokes it*, and the *action it performs*.
> May operate either `BEFORE` or `AFTER` the execution of the event that invokes it.

Storage

Main memory vs. disk:

- Data access from disk is typically 2 orders of magnitude slower.
- Data in main memory is **volatile**, while **non-volatile** for disk.
- Disk is solely mechanical moving parts, while main memory is completely electronic.

> Each time we read from disk, we retrieve a *block* of records.
> Size of these *blocks* is fixed, but depends on the OS.

Indexes

> Index: data structure that facilitates quicker access to a data.

- *can be* used for both main memory and disk!
- *stored in a data file*.

> Primary index: Indexes on the primary key (ordered on key).

- --> Dense index: Has exactly one index entry for each search key value.
- --> Sparse index: Has fewer index entries than search key values.

> Clustering index: Indexes on a non-key field (ordered on field).

- --> One index entry of each (distinct!) value of the field.
- --> Each index points to first data block of records for search key.

> Secondary index: Not ordered on the index's search key (purely a mess!)

> Multi-level index: Structure of index on index until all entries of the top-level structure fit into 1 disk block.

- --> Pin top-level index in main memory (RAM).

B^+-trees

- *multi-leveled* indexing structure
- tailored for disk-based data organization: *aligns with disk block sizes, so very efficient for disk storage and access.*
- grows horizontally by splitting the root.

$\Downarrow$ Operation	Average	Worst case
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

> Balanced: all paths from root to a leaf have the same length.

- --> guarantees good search performance!

Recovery strategies

> Checkpointing: process where the system periodically writes all modified (*dirty*) pages from memory to disk.

- improves **recovery** efficiency
- ensures a **consistent** state can be restored after unexpected crash.

> Shadow-paging: copy-on-write technique for avoiding in-place updates of pages

- when a page is modified (*dirty*), the system writes changes to a **new (shadow) page** instead of overwriting the old.

- upon commit the page table pointer is switched to the new page (**atomic**!).
  ≈ no-undo / no-redo

$\Longrightarrow$ *possible to use both techniques, but often redundant because both handle recovery well*.

> Undo: rollback changes of *uncommited* transactions
> Redo: reapply changes of *commited* transactions after a crash.

> Steal: *uncommited* dirty pages can be written, so you need to ***undo*** them.

- e.g. to save space in main memory you flush the dirty pages more often.

  > Force: *commited* pages are immediately written (forced!) to disk, so you do not need to repeat them (***no-redo***).

- generally not favorable because it leads to a ton of continuously costly I/Os

	Steal	**No-steal**
Force	Undo / no-redo	No-undo / no-redo
No-force	Undo / redo	No-undo / redo

> Undo/redo:

1. Undo all transactions that has a log entry of *"start"* but no *"commit"*.
2. Redo all transactions that has a long entry of *"start"* and *"commit"*.

Misc