Exam prep

Lecture 1

File Storage

- quick answers to questions about the data
- protecting the data from inconsistent changes made by users accessing it at the same time
- restricting access to some parts of the data
- multiple data storage formats
- data redundancy: some parts of the data can be stored in multiple files => potential
  inconsistencies
- read/write operations are described in the program (using certain record structures) => difficulties in program development (changes in the file structure lead to changes in the program)
- changing data (modifying/removing records), retrieving data based on search criteria - difficult operations
- integrity constraints - checked in the program
- main memory management, e.g., how is a data collection of tens/hundreds of GB loaded for processing?
- no adequate security policies, allowing different users to access different segments of data
- concurrent data access is difficult to manage
- data must be restored to a consistent state in the event of a system failure, e.g., a bank transaction transferring money from account A to account B is

interrupted by a blackout after having debited account A, but before crediting account B; money must be put back into account A

- files: useful for single-user programs dealing with a small amount of data

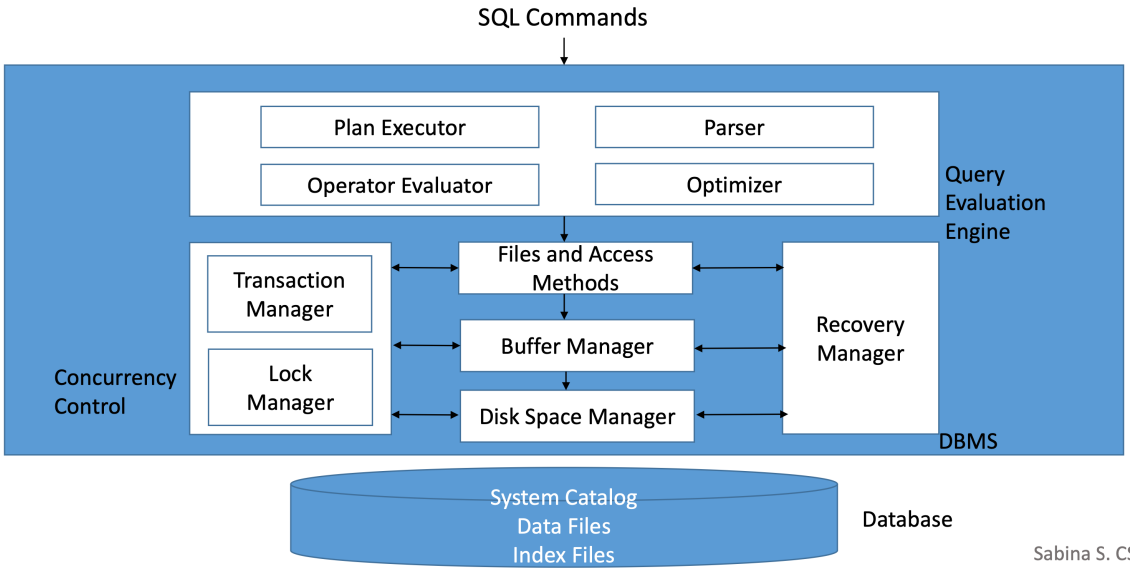
- Data describing models: data must be described by a model
  - I.E. a set of concepts and rules (structure of the data, consistency constraints, relations between data)
  - Schema
    - data structures used to describe collections of data stored in the DB
    - instance: data in a collection
      - analogy: class-object in OOP
- Relation: main concept in describing data
  - schema: relationName(fieldName1: f1Type, fieldName2: f2Type, …)
- Entity-Relationship Model
  - such a semantic model is useful since, even though the database management
    system's model hides many details, it's still closer to how the data is stored to the user's perspective on the data
  - a design in such a model is subsequently expressed in terms of the database
    management system's model

- Main Concepts:
  - **entities, attributes, relationships**

  - **entity**
    - a piece of data, an object in the real world
    - described by attributes

    - **entity set**
        - entities with the same structure
        - name, set of attributes
    - **attribute**
        - name, the domain of possible values, conditions to check the correctness
    - **key**
        - a restriction defined on an entity set
        - minimal set of attributes with distinct values in the entity set's instances
    - **relationship**
        - specifies an association among 2 or more entities
        - descriptive attributes can be used
    - **relationship set**
        - describes all relationships with the same structure
        - name, entity sets used in the association, descriptive attributes
    - Schema: the set of entity sets and relationship sets
- Binary relations:
    - 1:1: a T1 has relations with at most 1 T2, a T2 has relations with at most 1 T1
    - 1:n: a T1 has relations with at most 1 T2, a T2 has relations with any number of T1
    - m:n: a T1 has relations with any nr. of T2, a T2 has relations with any number of T1
- ANSI-SPARC architecture
    - conceptual structure (database schema)

        - data structures and restrictions
    - external structures
        - data structures used by a certain program/user
        - relations computed on demand (think views, queries)
    - physical structure (internal structure)
        - storage structure
    - N ext. struct. → ONE conceptual struct. → ONE int. struct. → DBs
- Data independence
    - 3 levels of abstractions ⇒ Data is protected from changes in the data structure
    - logical data independence ⇒ programs are not affected by changes at a conceptual level
        - programs are developed in several stages
    - physical independence ⇒ application are protected from changes in the physical structure of the DB
- database definition → definition language
- DB management → insert, update, delete, querying
- DB administration → access authorization, usage and performance monitoring, optimizations, etc.
- DB protection → integrity (vs inconsistent changes), confidentiality (vs unauthorized access)



- **Optimizer:** produces an efficient execution plan for query evaluation, taking into
- **File & Access Methods, Buffer Manager, Disk Manager:** abstraction of files, bringing pages from the disk into memory, managing disk space
- **Transaction Manager, Lock Manager:** concurrency control, monitoring lock requests, granting locks when database objects become available
- **Recovery Manager:** recovery after a crash

Lecture 2

- Relational Model: dominant data model
  - simple visual representation
  - querying through a simple, high-level language (SQL)

- in an application, a **data collection** is used according to a **model**
- in the **relational model**, the data collection is organized as a set of **relations** (tables)
- a relation has a relation **schema** and a relation **instance**

- **$\{A_1, A_2, ..., A_n\}$** - a set of attributes
- **D_i** = **$Dom(A_i)$** ∪ **{?}**
  - **domain of possible values** for **attribute A_i**
  - the **undefined** (**null**) value is denoted here by {?}
    - unknown / not applicable
    - can be used to check if a value has been assigned to an attribute (the attribute could have the *undefined* value)
    - this value doesn't have a certain data type; attribute values of different types can be compared against this value (numeric, string, etc.)
- relation of **arity** (degree) n: $R \subseteq D_1 \times D_2 \times ... \times D_n$
- **$R[A_1, A_2, ..., A_n]$** - the **relation schema**

- Values of attributes are atomic and scalar
- Rows NOT ordered
- Records are stored as distinct tuples (although DBMS allow duplicates)
- the cardinality of an instance is the number of tuples it contains
- Integrity constraints
  - Checked when data is changed: violate the constraints → not allowed
  - conditions specified in DB schema limiting the data
  - Domain constraints:
    - conditions that must be satisfied by every relation instance: a column's values belong to its associated domain
  - Key constraints:
    - constraint stating that a subset of attributes must be unique identifiers
    - should be minimal
    - Superkey: a set of fields containing the key
    - relation can have multiple keys:
      - One is selected as the primary key
      - Others are candidate keys

    - Foreign key constraints:
        - values of some attributes can appear in other relation
        - establishes links between relations
        - used to store 1:n relations
        - for m:n relations, we use an auxiliary relation that has foreign keys towards the 2 connected tables
- DBMS reject changes that violate integrity constraints
- depending on the options set when creating tables, the DBMS may not reject the operations, but make additional data
- Relational databases → collection of relations with distinct names
- Relational DB schema → collection of schemas of relations in the DB
- DB instance → collection of relation instances that are legal (satisfy integrity constraints)

4. Managing Relational Databases with SQL

- defining components
    - CREATE, ALTER, DROP (SQL DDL)
- managing and retrieving data
    - SELECT, INSERT, UPDATE, DELETE (SQL DML)
- managing transactions
    - START TRANSACTION, COMMIT, ROLLBACK

- **restrictions associated with a column**:
  - NOT NULL - can't have undefined values
  - PRIMARY KEY - the column is a primary key
  - UNIQUE - the values in the column are unique
  - CHECK(condition) - the condition that must be satisfied by the column's values (simple conditions that evaluate to *true* or *false*)
  - FOREIGN KEY REFERENCES parent_table[(column_name)] [ON UPDATE action] [ON DELETE action]

- **possible actions for a foreign key:**
  - NO ACTION
    - the operation is not allowed if it violates integrity constraints
  - SET NULL
    - the foreign key value is set to *null*
  - SET DEFAULT
    - the foreign key value is set to the default value
  - CASCADE
    - the delete / update is performed on the parent table, but it generates corresponding deletes / updates in the child table

- add / remove a constraint
  - ADD [CONSTRAINT constraint_name] PRIMARY KEY(column_list)
  - ADD [CONSTRAINT constraint_name] UNIQUE(column_list)
  - ADD [CONSTRAINT constraint_name] FOREIGN KEY (column_list) REFERENCES table_name[(column_list)] [ON UPDATE action] [ON DELETE action]
  - DROP [CONSTRAINT] constraint_name

- DDL (Data Definition Language) → a subset of SQL used to create/remove/change components
- Create/Alter/Delete

Lecture 3

- DML (Data Manipulation Language) → a subset of SQL used to pose queries, to add/update/remove data

INSERT INTO table_name[(column_list)] VALUES (value_list)
INSERT INTO table_name[(column_list)] subquery,

UPDATE table_name
SET column_name=expression [, column_name=expression] ...
[WHERE condition]

DELETE FROM table_name
[WHERE condition]

- expression comparison_operator expression
- expression [NOT] BETWEEN valmin AND valmax
- expression [NOT] LIKE pattern ("%" - any substring, "_" - one character)
- expression IS [NOT] NULL
- expression [NOT] IN (value [, value] ...)
- expression [NOT] IN (subquery)
- expression comparison_operator {ALL | ANY} (subquery)
- [NOT] EXISTS (subquery)

- Select
  - SELECT [DISTINCT] select-list
    FROM from-list
    [WHERE condition]

        - compute cross product of tables from from-list
        - eliminate rows that don’t satisfy condition
        - select only the columns in select-list
        - if DISTINCT is specified, remove duplicate tuples
- Nested queries
    - SELECT in FORM or WHERE (with IN, EXISTS, ANY/ALL)
    - union, intersection and set difference can be done with AND, OR and [NOT] IN
- Joins
    - Inner join
        - source1 [alias] [INNER] JOIN source2 [alias] ON condition
        - takes the elements that have matching values in both tables
    - Left outer join
        - source1 [alias] LEFT [OUTER] JOIN source2 [alias] ON condition
        - takes all elements from source1 and the matched elements from source2
    - Right outer join
        - source1 [alias] RIGHT [OUTER] JOIN source2 [alias] ON condition
        - takes all elements from source2 and the matched elements from source1
    - Full outer join
        - source1 [alias] FULL [OUTER] JOIN source2 [alias] ON condition
        - takes all elements from source1 and source2 whether or not there are matches in the other table
    - Other joins:
        - JOIN … USING [column-list]
        - NATURAL JOIN
        - CROSS JOIN

Lecture 4

- GROUP BY, HAVING
  - GROUP BY grouping-list
    - set of columns to group by
    - group → set of rows with identical values for the columns in grouping-list
    - results correspond to the group
    - select-list
      - columns from grouping-list
      - aggregation operations based on columns from the group
        - SUM, COUNT, AVG, MIN, MAX
  - HAVING group-quali
    - expressions with a single value/group
    - a column in group-quali appears in grouping-list or as an argument to an aggregation operator
- ORDER BY
  - sorts SELECT result
    - desc - decreasing, ASC - increasing
- TOP
  - TOP n [PERCENT]
    - return only n rows/n% of all possible rows
  - differs in other DBMS (MySQL: LIMIT n)

$$\text{SELECT} \left[\left\{ \begin{matrix} \text{ALL} \\ \text{DISTINCT} \\ \text{TOP n [PERCENT]} \end{matrix} \right\} \right] \left\{ \text{expr1 [AS column1]} \left[, \text{expr2 [AS column2]} \right] \ldots \right\}^{*}$$

$$\text{FROM source1 [alias1]} \left[, \text{source2 [alias2]} \right] \ldots$$

$$\text{[WHERE qualification]}$$

$$\text{[GROUP BY grouping\_list]}$$

$$\text{[HAVING group\_qualification]}$$

$$\left[\left\{ \begin{matrix} \text{UNION [ALL]} \\ \text{INTERSECT} \\ \text{EXCEPT} \end{matrix} \right\} \text{SELECT\_statement} \right]$$

$$\left[\text{ORDER BY} \left\{ \begin{matrix} \text{column1} \\ \text{column1\_number} \end{matrix} \right\} \left[\left\{ \begin{matrix} \text{ASC} \\ \text{DESC} \end{matrix} \right\} \right] \left[, \left\{ \begin{matrix} \text{column2} \\ \text{column2\_number} \end{matrix} \right\} \left[\left\{ \begin{matrix} \text{ASC} \\ \text{DESC} \end{matrix} \right\} \right] \right] \ldots \right]$$

- SELECT statement ⇒ relation/table
- data sources can be tables/views, other select statements, join statements, all can be associated with an alias
- SELECT statement operation ordering (first → last) — FROM → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → TOP
- Functional dependencies and normal forms
  - To reduce data redundancy and logical inconsistency
  - 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
  - Data redundancy
    - can be tackled by replacing relations with smaller relations, containing strict attributes from the original relation
    - ideally, only redundancy-free schemas should be allowed, but they should at least be easily identified
  - if a relation is not in normal form X, it can be split into relations of multiple normal form X

Definition. The projection operator is used to decompose a relation. Let $R[A_1, A_2, \dots, A_n]$ be a relation and $\alpha = \{A_{i_1}, A_{i_2}, \dots, A_{i_p}\}$ a subset of attributes, $\alpha \subset \{A_1, A_2, \dots, A_n\}$. The projection of relation R on α is:

$$R'\left[A_{i_1}, A_{i_2}, \dots, A_{i_p}\right] = \Pi_\alpha(R) = \Pi_{\{A_{i_1}, A_{i_2}, \dots, A_{i_p}\}}(R) = \{r[\alpha] | r \in R\}$$

where $\forall r = (a_1, a_2, \dots, a_n) \in R \Rightarrow \Pi_\alpha(r) = r[\alpha] = \left(a_{i_1}, a_{i_2}, \dots, a_{i_p}\right) \in R'$,

and all elements in R' are distinct.

Definition. The natural join operator is used to compose relations. Let *R[α, β]*, *S[β, γ]* be two relations over the specified sets of attributes, $\alpha \cap \gamma = \emptyset$. The natural join of relations *R* and *S* is the relation:

$$R * S[\alpha, \beta, \gamma] = \left\{\left(\Pi_\alpha(r), \Pi_\beta(r), \Pi_\gamma(s)\right) \middle| r \in R, s \in S \text{ and } \Pi_\beta(r) = \Pi_\beta(s)\right\}$$

- a relation R can be decomposed into multiple new relations $R_1, R_2, \dots, R_m$; the decomposition is good if $R = R_1 * R_2 * \cdots * R_m$, i.e., R's data can be obtained from the data stored in relations $R_1, R_2, \dots, R_m$ (no data is added / lost through decomposition / composition)

Lecture 5

- composite attribute → set of attributes in a relation
- attribute can have multiple values → repeating attributes
- attributes must be scalar and atomic (can't be further decomposed)
- in the relational model, we should avoid repeating attributes

- let $R[A]$ be a relation; A - the set of attributes
- let α be a repeating attribute in R (simple or composite)
- R can be decomposed into 2 relations, such that α is not a repeating attribute anymore
- if K is a key in R, the two relations into which R is decomposed are:

$$R'[K \cup \alpha] = \Pi_{K \cup \alpha}(R)$$
$$R''[A - \alpha] = \Pi_{A-\alpha}(R)$$

- 1NF - A relation is of the first normal form iff it doesn't have repeating values
- Functional dependency
  - alpha → beta
    - iff every value of alpha is associated with a precise, unique value of beta
    - or iff a value a value of alpha appears in multiple rows, those rows contain the same value of beta

---

Definition. Let $R[A_1, A_2, \ldots, A_n]$ be a relation and α, β two subsets of attributes of R. The (simple or composite) attribute α functionally determines attribute β (simple or composite), notation:

$$\alpha \rightarrow \beta$$

if and only if every value of α in R is associated with a precise, unique value for β (this association holds throughout the entire existence of relation R); if an α value appears in multiple rows, each of these rows will contain the same value for β:

$$\Pi_\alpha(r) = \Pi_\alpha(r') \ \ implies \ \ \Pi_\beta(r) = \Pi_\beta(r')$$

• in the dependency $\alpha \rightarrow \beta$, α is the *determinant*, and β is the *dependent*

- if a relation contains a functional dependency → associated values can be stored multiple times → data redundancy
  - wasting space → same values multiple times
  - update anomalies → if the func. dep. is changed, it must be changed everywhere in the table
  - insertion anomalies → data in the func. dep. requires additional data to be stored
  - deletion anomalies → deletion of data leads to deletion of data that was meant to hold
- K be the key of R[A1, A2 … An] ⇒ k→beta, for any beta subset of {A1, A2 … An}

1. If *K* is a key of $R[A_1, A_2, \dots, A_n]$, then $K \rightarrow \beta$, $\forall\beta$ a subset of $\{A_1, A_2, \dots, A_n\}$.

- such a dependency is always true, hence it will not be eliminated through decompositions

2. If $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ - *trivial functional dependency* (*reflexivity*).

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\beta \subseteq \alpha}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \alpha \rightarrow \beta$$

3. If $\alpha \rightarrow \beta$, then $\gamma \rightarrow \beta$, $\forall\gamma$ with $\alpha \subset \gamma$.

$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \underset{\alpha \subset \gamma, prop.2}{\Rightarrow} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \rightarrow \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \gamma \rightarrow \beta$$

4. If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ - *transitivity*.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \rightarrow \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \underset{\beta \rightarrow \gamma}{\Rightarrow} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \rightarrow \gamma$$

5. If $\alpha \rightarrow \beta$ and γ a subset of $\{A_1, \dots, A_n\}$, then $\alpha\gamma \rightarrow \beta\gamma$, where $\alpha\gamma = \alpha \cup \gamma$.

$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \begin{vmatrix} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{vmatrix} \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

- Attributes A is prime if it is part of a key K if itself is a key. Any attribute that is not part of any key is non-prime
- Beta fully functionally dependent on Alpha iff
  - alpha → beta
  - there is no subset gamma of alpha such that gamma → beta
- 2NF →
  - is 1NF
  - any non-prime attributes (simple or composite) are fully functionally dependent on every key of the relation

obs. Let R be a 1NF relation that is not 2NF. Then R has a composite key (and a functional dependency $\alpha \rightarrow \beta$, where α (simple or composite) is a proper subset of a key and β is a non-prime attribute).

decomposition

- relation $R[A]$ (A - the set of attributes), K - a key
- β non-prime, β functionally dependent on α, $\alpha \subset K$ (β is functionally dependent on a proper subset of attributes from a key)
- the $\alpha \rightarrow \beta$ dependency can be eliminated if R is decomposed into the following 2 relations:

$$R'[\alpha \cup \beta] = \Pi_{\alpha \cup \beta}(R)$$
$$R''[A - \beta] = \Pi_{A-\beta}(R)$$

- Transitive dependency: Z is trans. dep. on X if exists Y s.t. X→Y, Y→Z, Y→X doesn't hold (and Z not in X or Y)
- 3NF →
  - is 2NF
  - no non-prime key is transitively dependent on any key in the relation
  - or
    - iff for every non-trivial func. dep. X→A that holds over R:
      - X is superkey, or
      - A is prime attribute
  - OR
    - **every non-key attribute depends only on the key, the whole key and nothing but the key**
- BCNF (Boyce-Codd) →
  - is 3NF
  - every determinant for a func. dep. is a key
    - **every attribute depends only on the key, the whole key and nothing but the key**

Lecture 6

- R[A] - a relation
- F - a set of functional dependencies
- α – a subset of attributes

- I. compute closure of F: F+
  - F+ → set of all func. dep implied by F
  - use Armstrong's Axioms to compute (reflexivity, transitivity, augmentation)
    - Union: if alpha→beta and alpha→gamma ⇒ alpha→beta-gamma
    - Decomposition: if alpha→beta-gamma ⇒ alpha→beta and alpha→gamma
    - Pseudotransitivity: if alpha→beta and beta-gamma→delta ⇒ alpha-gamma→delta
- II. compute closure of alpha under F: alpha+
  - i.e. the set of attributes func. dep on alpha

closure := α;
repeat until there is no change:
  for every functional dependency $\beta \rightarrow \gamma$ in F
    if $\beta \subseteq$ closure
      then closure := closure $\cup$ γ;

- III. compute minimal cover for a set of dependencies
  - DEF: F, G, sets of func. dep., be equivalent (3 line equal) if F+=G+
  - DEF: For F, set of func. dep., a minimal cover is a set of func. dep Fm where:
    - F equiv. Fm
    - the right side of every dep. in Fm is a single attribute

        - the left side of every dep. in Fm is irreducible
        - no dep. in Fm is redundant
- Multi-values dependency
    - Let R[A] be a relation, A = alpha U beta U gamma
    - alpha→→beta (alpha multi-determines beta) iff each value u of alpha is associated with a set of values v of beta (beta(u) = {v1, v2, …}) and the association hold regardless of gamma
        - the sign is 2 arrows stacked one on the other, not in sequence

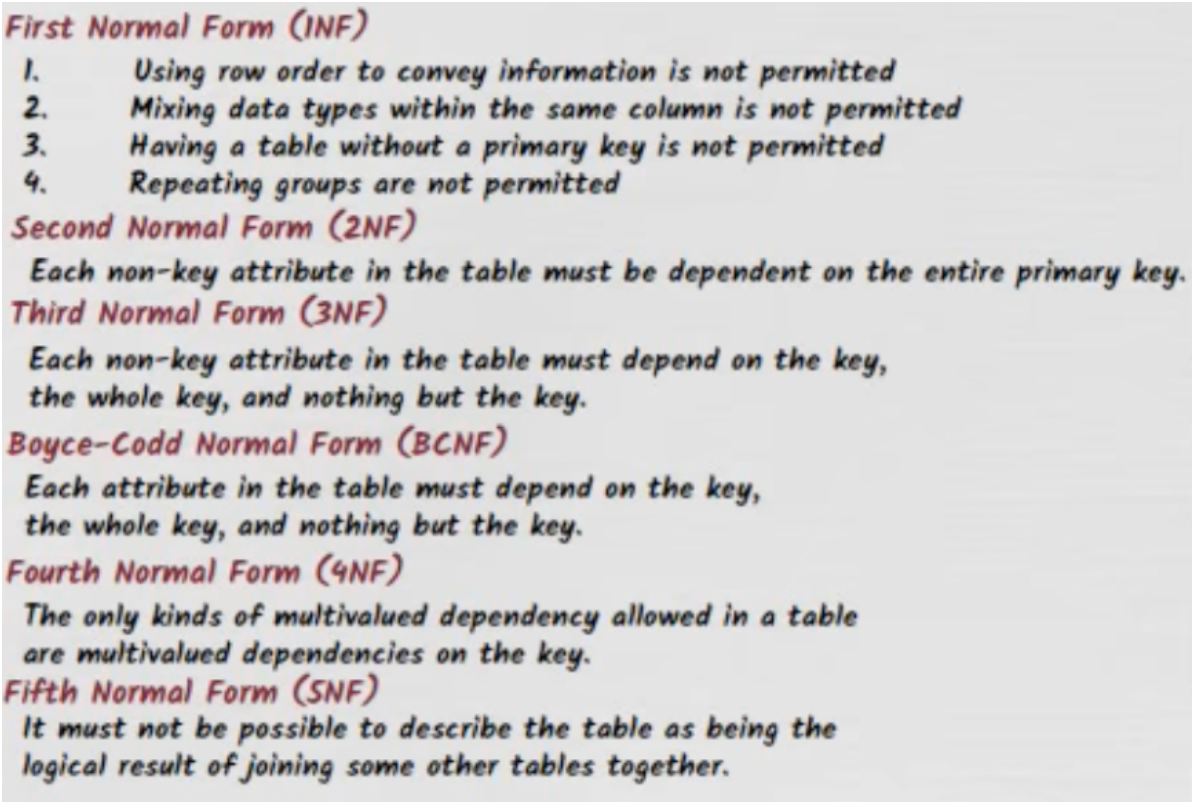
    - obs. $\sigma_{\alpha=u}(R)$ produces a relation that contains the tuples of R where $\alpha = u$

    - let $R[A]$ be a relation, $\alpha \rightrightarrows \beta$ a multi-valued dependency, and $A = \alpha \cup \beta \cup \gamma$, with γ a non-empty set
    - the association among the values in $\beta(u)$ for β and the value u of α holds regardless of the values of γ (the context)
    - i.e., these associations (between u and an element in $\beta(u)$) exist for any value w in γ:
        - $\forall w \in \Pi_\gamma(\sigma_{\alpha=u}(R)),\ \exists r_1, r_2, \dots, r_n$ such that $\Pi_\alpha(r_i) = u,\ \Pi_\beta(r_i) = v_i,\ \Pi_\gamma(r_i) = w$

    - If R[A], A = alpha U beta U gamma, alpha→→beta ⇒ alpha→→gamma
- 4NF →
    - is 3NF
    - for every alpha→→beta over R ⇒
        - beta part of alpha, or
        - alpha U beta = R, or
        - alpha superkey
    - Only multi-dependency allowed is multi-dependency on the key
- Join dependency

- R[A] relation, Ri[alphai], i = 1,2,…,m, Ri[alphai] - projection of R of set of attributes alphai
- R satisfies join dependency * {alpha1, alpha2,…, alpham} iff R = R1 * R2 * … * Rm

- 5NF →
  - is 4NF
  - for every non-trivial join dependency is implied by the candidate keys of R
    - JD * {alpha1, alpha2,…, alpham} is trivial iff at least one alphai is the set of all attributes of R
    - JD * {alpha1, alpha2,…, alpham} is implied by candidate keys iff each alphai is a superkey of R
  - The table can’t be described as a logical result of joining some decomposed tables



Lecture 7

- Relational algebra
  - Algebra on bags → duplicates no eliminated
  - Selection (equiv. WHERE)

- notation: $\sigma_C(R)$
- resulting relation:
  - schema: R's schema
  - tuples: records in R that satisfy condition C
- equivalent SELECT statement

```
SELECT *
FROM R
WHERE C
```

  - Projection (equiv. select)

- notation: $\pi_\alpha(R)$
- resulting relation:
  - schema: attributes in α
  - tuples: every record in R is projected on α
- α can be extended to a set of expressions, specifying the columns of the relation being computed
- equivalent SELECT statement

```
SELECT DISTINCT α
FROM R

SELECT α
FROM R          -- algebra on bags
```

  - cross-product (similar cartesian product)

notation: $R_1 \times R_2$
resulting relation:
- schema: the attributes of R_1 followed by the attributes of R_2
- tuples: every tuple r_1 in R_1 is concatenated with every tuple r_2 in R_2

equivalent SELECT statement

```
SELECT *
FROM R1 CROSS JOIN R2
```

- Union, set-difference, intersection

union, set-difference, intersection
- notation: $R_1 \cup R_2$, $R_1 - R_2$, $R_1 \cap R_2$
- R_1 and R_2 must be union-compatible:
  - same number of columns
  - corresponding columns, taken in order from left to right, have the same domains
- equivalent SELECT statements

```
SELECT *        SELECT *      SELECT *
FROM R1         FROM R1       FROM R1
UNION           EXCEPT        INTERSECT
SELECT *        SELECT *      SELECT *
FROM R2         FROM R2       FROM R2
```

- conditional/theta join (equiv. inner join)

- *condition join* (or *theta join*)
  - notation: $R_1 \bigotimes_{\Theta} R_2$
  - result: the records in the cross-product of R_1 and R_2 that satisfy a certain condition
- definition ⇒ $R_1 \bigotimes_{\Theta} R_2 = \sigma_{\Theta}(R_1 \times R_2)$
- equivalent SELECT statement

```
SELECT *
FROM R1 INNER JOIN R2 ON Θ
```

- natural join

- *natural join*
  - notation: $R_1 * R_2$
  - resulting relation:
    - schema: the union of the attributes of the two relations (attributes with the same name in R_1 and R_2 appear once in the result)
    - tuples: obtained from tuples <r_1, r_2>, where r_1 in R_1, r_2 in R_2, and r_1 and r_2 agree on the common attributes of R_1 and R_2
  - let $R_1[\alpha]$, $R_2[\beta]$, $\alpha \cap \beta = \{A_1, A_2, \ldots, A_m\}$; then:

$$R_1 * R_2 = \pi_{\alpha \cup \beta}(R_1 \otimes_{R_1.A_1=R_2.A_1 \; AND \; \ldots \; AND \; R_1.A_m=R_2.A_m} R_2)$$

  - equivalent SELECT statement

```
SELECT *
FROM R1 NATURAL JOIN R2
```

- left outer join

left outer join

- notation (in these notes): $R_1 \ltimes_C R_2$
- resulting relation:
  - schema: the attributes of R_1 followed by the attributes of R_2
  - tuples: tuples from the condition join $R_1 \otimes_c R_2$ + the tuples in R_1 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_2
- equivalent SELECT statement

```
SELECT *
FROM R1 LEFT OUTER JOIN R2 ON C
```

- right outer join

right outer join

- notation: $R_1 \rtimes_C R_2$
- resulting relation:
  - schema: the attributes of R_1 followed by the attributes of R_2
  - tuples: tuples from the condition join $R_1 \otimes_c R_2$ + the tuples in R_2 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_1
- equivalent SELECT statement

```
SELECT *
FROM R1 RIGHT OUTER JOIN R2 ON C
```

- full outer join

full outer join

- notation: $R_1 \bowtie_C R_2$
- resulting relation:
  - schema: the attributes of R_1 followed by the attributes of R_2
  - tuples:
    - tuples from the condition join $R_1 \otimes_c R_2$ +
    - the tuples in R_1 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_2 +
    - the tuples in R_2 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_1
- equivalent SELECT statement

```
SELECT *
FROM R1 FULL OUTER JOIN R2 ON C
```

- left semi join

left semi join

- notation: $R_1 \triangleright R_2$
- resulting relation:
  - schema: R_1's schema
  - tuples: the tuples in R_1 that are used in the natural join $R_1 * R_2$

- right semi join

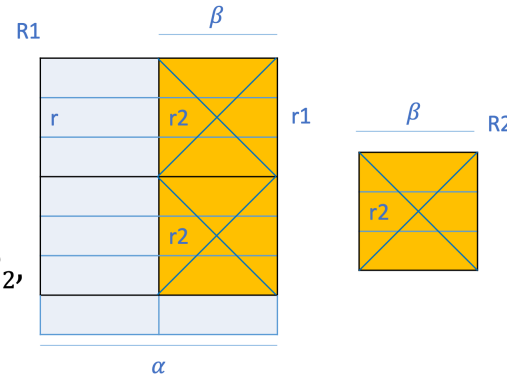
right semi join

- notation: $R_1 \triangleleft R_2$
- resulting relation:
  - schema: R_2's schema
  - tuples: the tuples in R_2 that are used in the natural join $R_1 * R_2$

- division

division

- notation: $R_1 \div R_2$
- $R_1[\alpha], R_2[\beta], \beta \subset \alpha$
- resulting relation:
  - schema: $\alpha - \beta$
  - tuples: a record $r \in R_1 \div R_2$ iff $\forall r_2 \in R_2$, $\exists r_1 \in R_1$ such that:
    - $\pi_{\alpha-\beta}(r_1) = r$
    - $\pi_{\beta}(r_1) = r_2$
    - i.e., a record r belongs to the result if in R_1 r is concatenated with every record in R_2



- Independent set of operators M: there will be a relation that can be described with M's operators, but not M-{op}, where op is an operator from M

- for the previously described query language, with operators:

  $\{\sigma, \pi, \times, \cup, -, \cap, \otimes, *, \ltimes, \rtimes, \bowtie, \rhd, \lhd, \div\}$

an independent set of operators is $\{\sigma, \pi, \times, \cup, -\}$

- Renaming operator

- the *renaming* operator

  $\rho(R'(A_1 \to A_1', A_2 \to A_2', A_3 \to A_3'), E)$

  - E - relational algebra expression
  - the result, relation R', has the same tuples as the result of E
  - attributes A_1, A_2, and A_3 are renamed to A_1', A_2', and A_3', respectively

- assignment

  $R[list] := expression$

  - the expression's result (a relation) is assigned to a variable (R[list]), specifying the name of the relation [and the names of its columns]
- eliminating duplicates from a relation

  $\delta(R)$
- sorting records in a relation

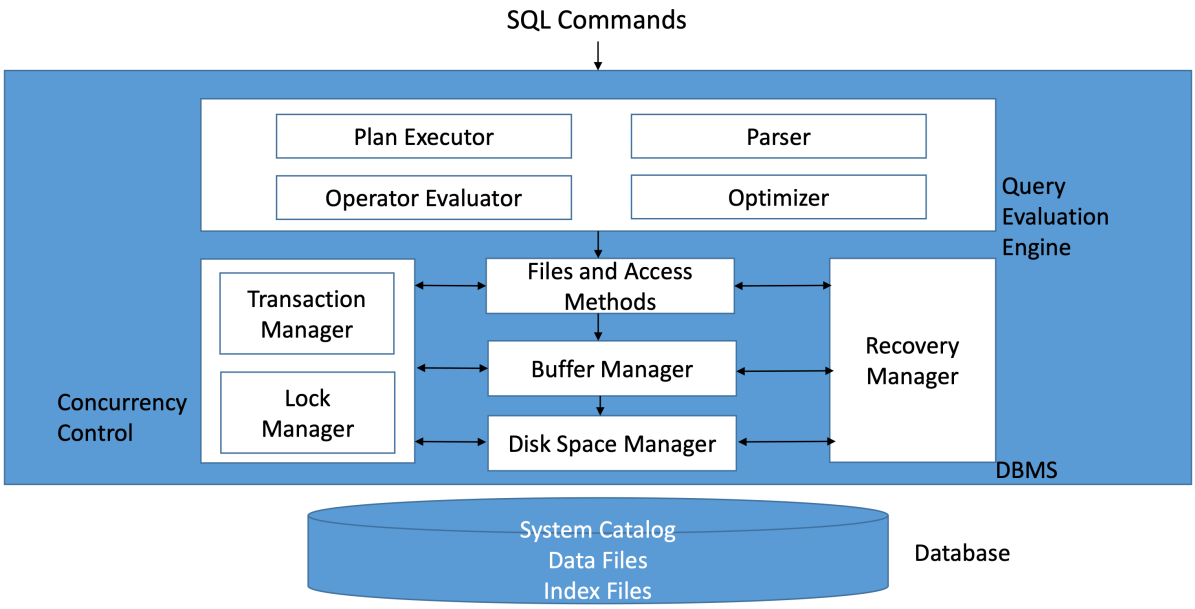
  $S_{\{list\}}(R)$
- grouping

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

  - R's records are grouped by the columns in *list2*
  - *list1* (that can contain aggregate functions) is evaluated for each group of records

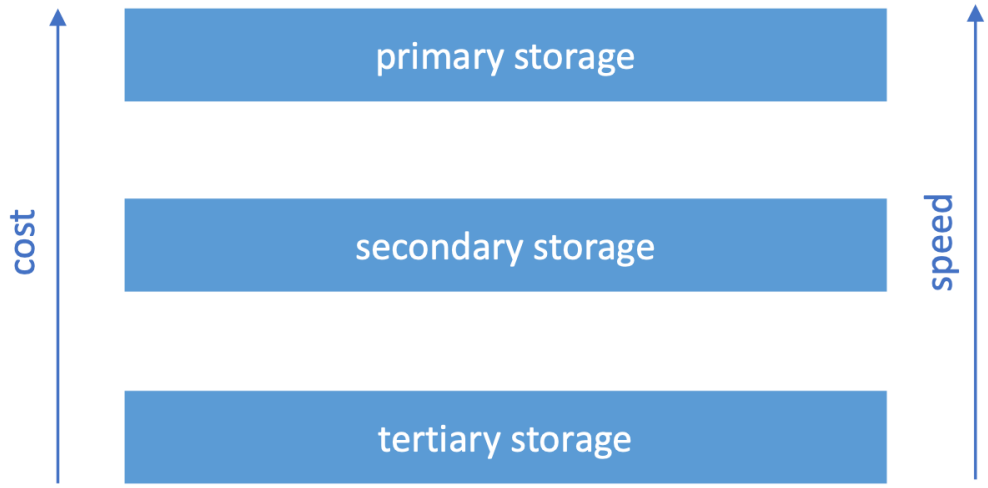
Sabina S. CS

Lecture 8



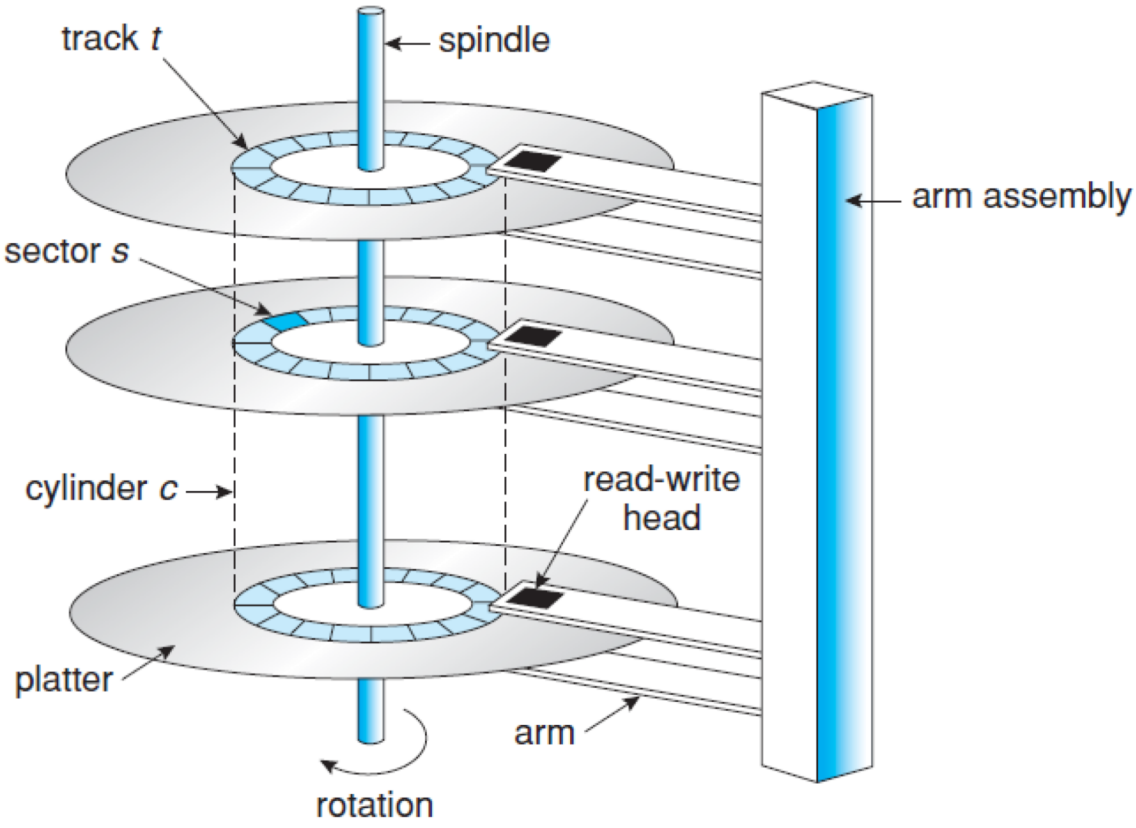
- Memory Hierarchy
  - Primary memory
    - Cache, main memory
    - very fast access
    - volatile
    - currently used data
  - Secondary memory
    - magnetic disks
    - slower storage devices
    - non-volatile
    - disks - sequential, direct access
    - main database
  - Tertiary memory
    - optical disks, tapes
    - slowest storage devices

        - non-volatile
        - tapes
            - only sequential access
            - good for archives/backups
            - unsuited for data frequently accessed



            - significantly cheaper than main memory
            - for large amounts of data that aren't discarded upon shutdown
                - need for DBMS to bring data from disks to main memory for processing
- Secondary memory — Magnetic disks
    - direct access
    - no need for DBMS to know whether data is from disks or cache
    - Disk block
        - sequence of contiguous bytes
        - unit for data storage and transfer
        - reading/writing a block → an input/output operation

- Tracks
  - concentric rings of blocks recorded on one or more disks
- Sectors → arcs on tracks
- Platters (single or double-sided)
- Cylinder → set of all tracks of the same diameter
- Disk Heads
  - 1/recorded surface
  - for IO, a head must be on top of the block
  - all heads are moved as a unit
  - systems with one active head
- Sector size → characteristic of the disk, immutable
- Block size → multiple sector size



- DBMS operate on data when it is in memory
- Block → unit for data transfer between disk and main memory
- time to access a desired location
  - main memory → aprox. the same for all the data
  - disks → depends where the data is located
- disk access time
  - seek time + rotation delay + transfer time
  - seek time
    - time to move the disk head to the desired location
  - rotation delay
    - time for the block to get under the head
  - transfer time
    - time for IO
- time required for DB operations → dominated by the time taken to transfer blocks from disks to main memory
  - minimize access time
  - place data carefully, in close proximity
    - same block
    - same track
    - same cylinder
    - adjacent cylinder
  - accessing data in sequential fashion reduces seek time and rotation delay
- Characteristics
  - storage capacity (GB)
  - platters (amount, single or double sided)
  - average/max seek time

- average rotation delay
- number rotations/min
- data transfer rate (MB/s)

Moore's Law

- Gordon Moore: "the improvement of integrated circuits is following an exponential curve that doubles every 18 months"
- parameters that follow Moore's law
  - speed of processors (number of instructions executed / sec)
  - no. of bits / chip
  - capacity of largest disks
- parameters that do not follow Moore's law
  - speed of accessing data in main memory
  - disk rotation speed

=> "latency" keeps increasing

- time to move data between memory hierarchy levels appears to take longer compared with computation time

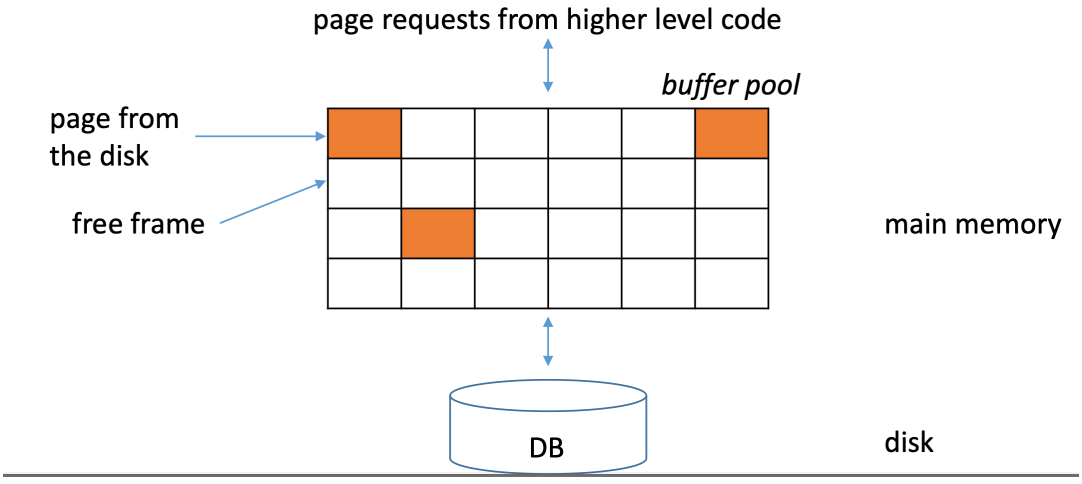
Solid-State Disks

- NAND flash components
- faster random access
- higher data transfer rates
- no moving parts
- higher cost per GB
- limited write cycles

- Managing disk space
  - Disk Space Manager (DSM)
    - Page
      - unit of data
      - size of page - size of disk block
      - R/W a page <> one IO operation
    - commands to allocate/deallocate/read/write a page
    - knows which page on which disk block
    - monitors disk usage
    - keeps track of available disk blocks
    - free blocks can be identified

- keeps a linked list of free blocks
- maintaining a bitmap where one bit/block indicates whether it is used or not (fast identification of contiguous available areas)

- Buffer manager BM
  - brings new data from disks to main memory
  - decides the main memory pages to replace
  - manages available main memory
    - collection of pages names buffer pool BP
    - frame
      - page in BP
      - slot that can hold a page
  - replacement policy → policy that dictates the choice of replacement frames in BM
  - Example
    - Layer L of DBMS requests from BM page P
    - If P not in BP, BM brings it into a frame F
    - P not longer needed → L notifies BM → P realease → F can be reused
    - P modified → L notifies BM → changes occur of disk



- BM maintains 2 variables for every frame F
  - pin_count
    - number of current users
    - pin_count == 0 ⇒ F can be replacement frame
  - dirty
    - bool
    - indicates if F has been changed in BP
- L asks for page P
  - if P in BP, pin_count(F)++, F frame containing P
- if not
  - if BP contains multiple frames with pic_count == 0, choose FR based on replacement policy
  - pin_count(FR)++
  - if dirty(FR), BM writes the page from FR in the disk
  - read P in frame FR
  - return the address of FR
- if no frame has pin_count = 0, BM waits for a possible replacement frame
- Write-Ahead Log → crash recovery, additional restrictions
- replacement policies
  - Least Recently Used
    - queue of pointer with pin_count = 0
    - when a frame reaches pin_count = 0 it is added to the back of the queue
    - frame at the head of the queue is chosen for replacement
  - Most Recently Used
    - (I guess same but with stack)

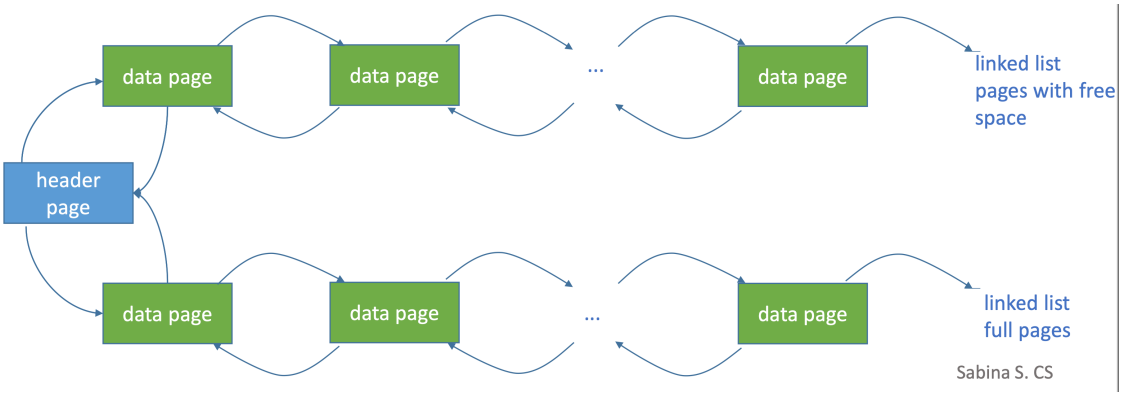
- random
- Can have significant impact on performance

- *clock replacement*
  - LRU variant
  - *n* – number of frames in BP
  - frame - *referenced* bit; set to *on* when *pin_count* becomes 0
  - *crt* variable - frames 1 through *n*, circular order
  - if the current frame is not chosen, then *crt*++, examine next frame
  - if *pin_count* > 0
    - current frame not a candidate, *crt*++
  - if *referenced* = *on*
    - *referenced* := *off*, *crt*++
  - if *pin_count* = 0 AND *referenced* = *off*
    - choose current frame for replacement

Lecture 9-10

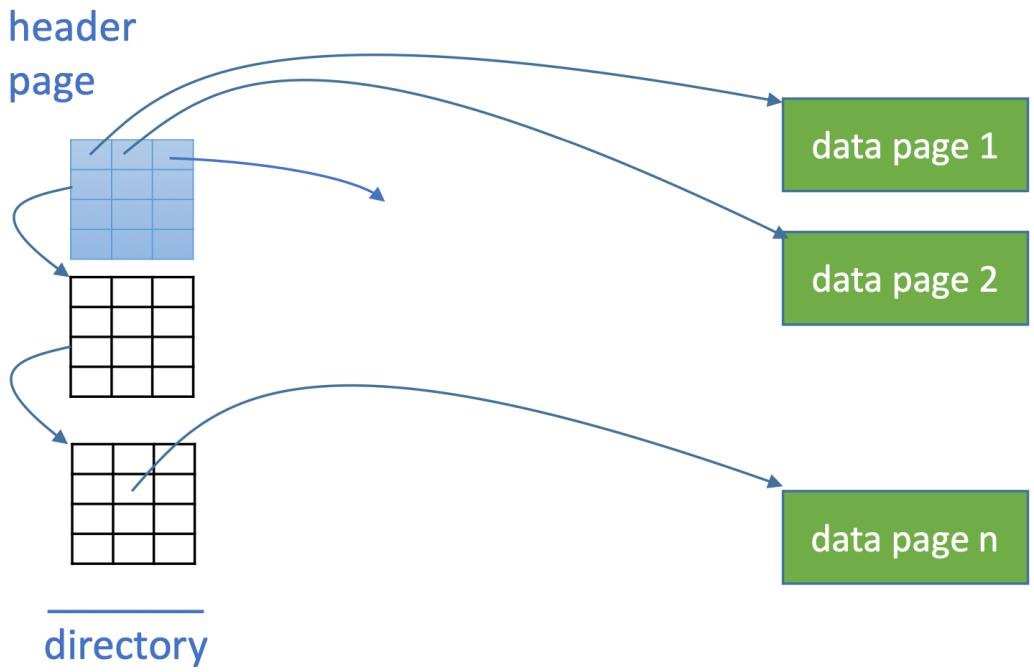
- File records
  - higher level layers in DBMS treat pages as a collection of records
  - file of records → collection of records → one or more pages
  - every record has its identifier RID
  - with the rid, the page containing the record can be identified
- Heap files
  - records not recorded
  - Operations
    - create/delete file
    - insert record (monitor pages with free space)
    - delete/update record by RID
    - scan all records (keep track of all pages in a file)
  - Linked Lists

- doubly linked lists
- stores the address of the header page in each file
- 2 lists → with or without free space



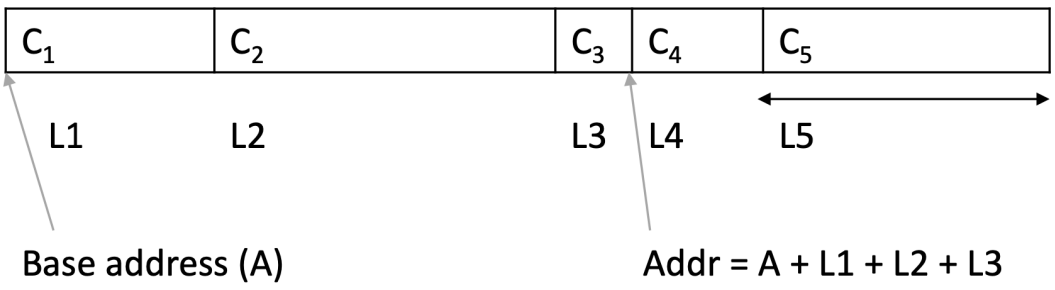
- Drawbacks
  - variable-length records: pages mostly in files with free space
  - when adding a record, multiple pages have to be checked before one with enough free space is found

- Directory of pages
  - DBMS stores the location of the header page for each heap file
  - directory → collection of pages
  - directory entry → identifies page in file
  - directory entry size → much smaller than the size of the page
  - directory size → much smaller than the size of the file
  - free space management
    - 1 bit/directory entry → corresponding page has space or not
    - count/entry → available space on the page ⇒ efficient search when adding a variable-length record

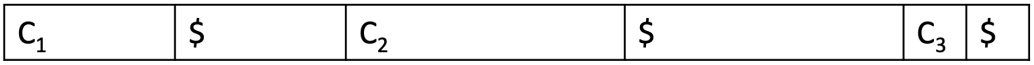


- Others:
  - Sorted lists: useful when data must be sorted for range selections
  - Hashed files: files hashed on some fields (records sorted according to the hash function); good for equality selections

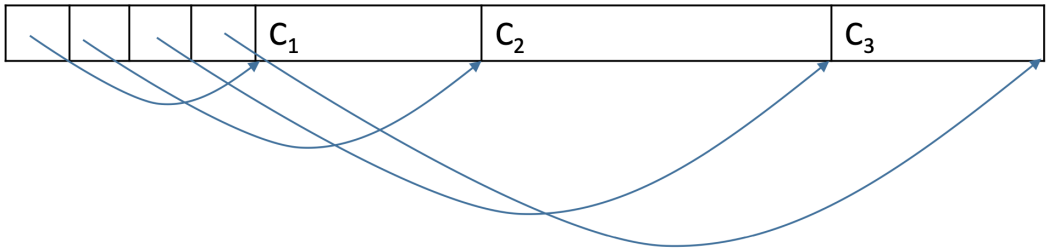
- Record formats
  - Fixed-length
    - fixed number of fields
    - fields stores consecutively
    - computing field address: record address + number preceding fields



- Variable-length
  - V1
    - fields: stores consecutively separated by delimiter
    - find field → record scan

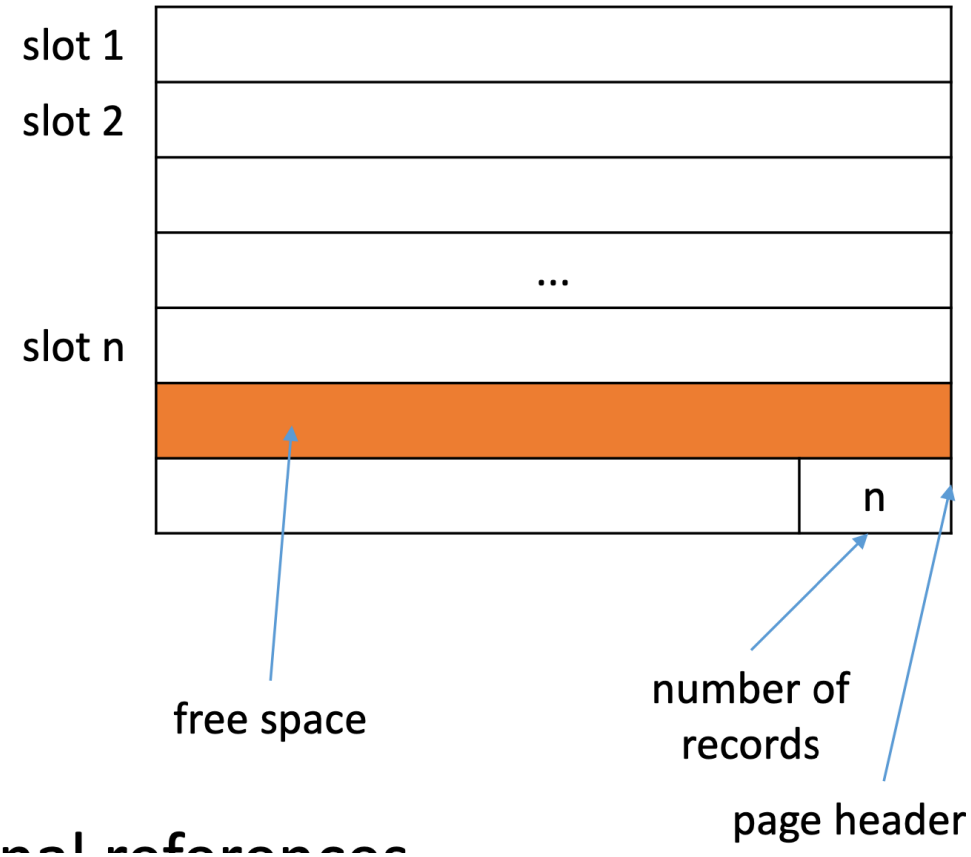


  - V2
    - beginning reserved for array of offsets (including to the end)
    - array overhead, but direct access to fields



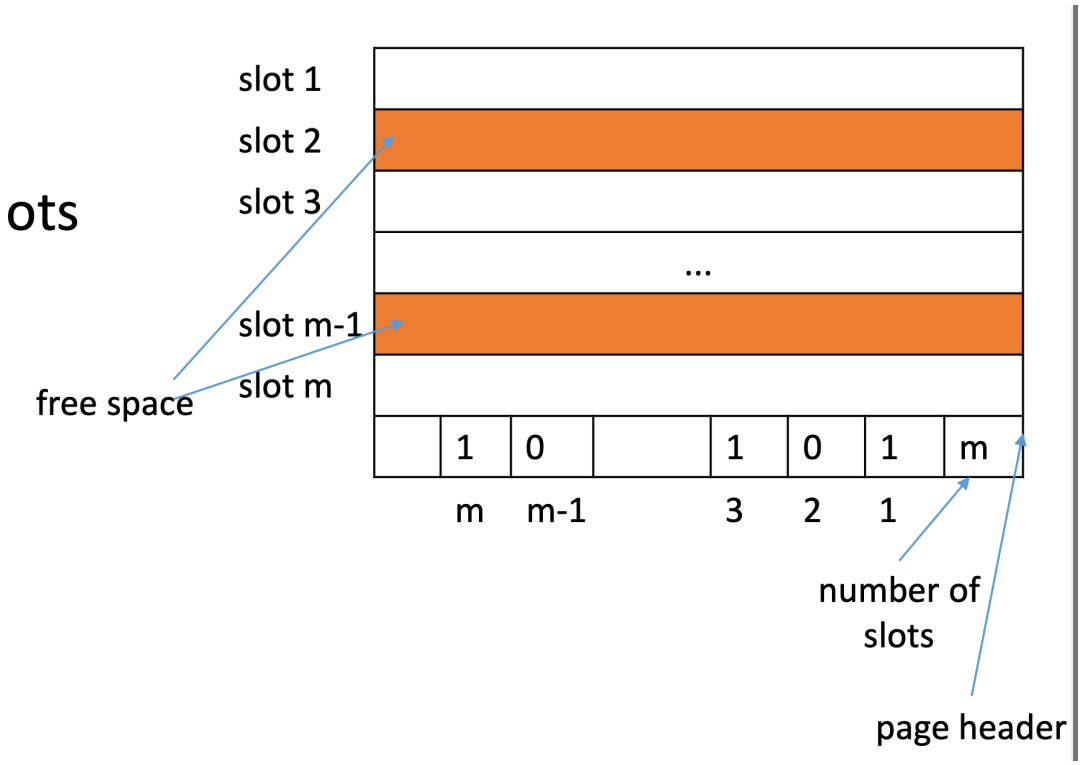
- Page formats
  - Page → collection of slots, 1 record/slot
  - Identify record: RID<page_id, slot number>
  - fixed-length records
    - records have the same size
    - uniform, consecutive slots
    - adding record → finding available slot
    - problems: keep track of available records, locating records
    - V1
      - n = number records/page
      - records stored in first n slots

- compute records i → locate offset i
- delete record → last record on the page is moved into the empty slot
- empty slots → end of page
- problem: record with references
  - slot number changes, but the RID contains slot number

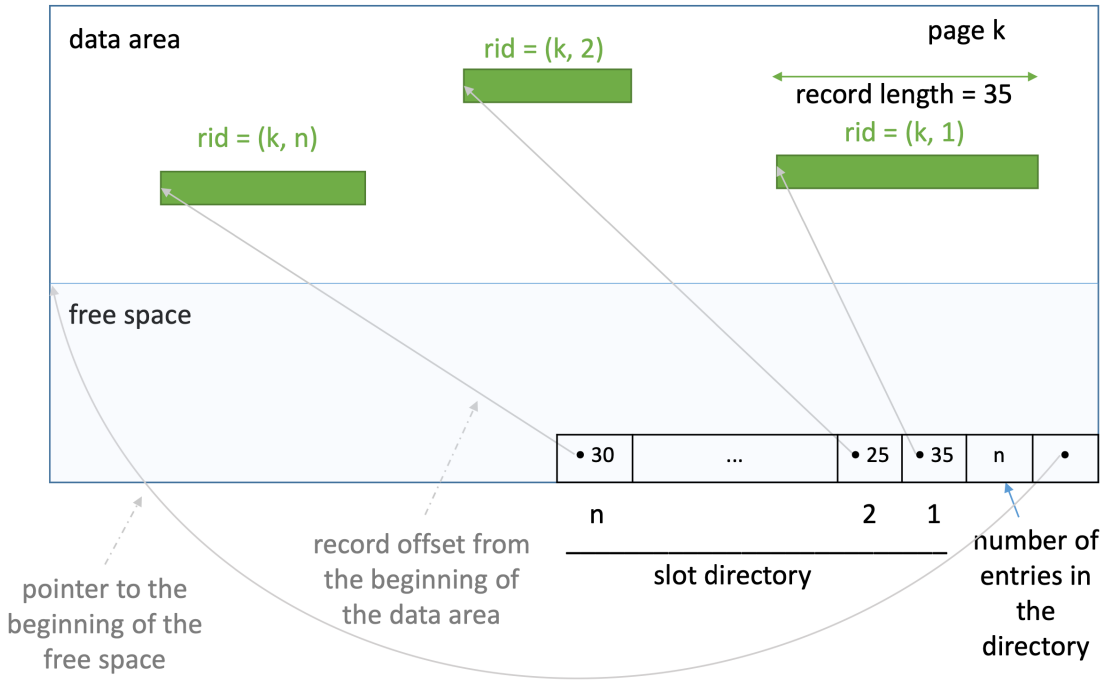


rnal references

- V2
  - bit array to monitor available slots
  - delete record → turn off corresponding bit



- variable-length
  - add record → find empty slot of right size
  - delete record → contiguous free space
  - directory of slots / page
  - pair <record_offset, record_length>/slot
  - pointer to the beginning of the free space area
  - moving record on a page → offset changes, not its slot number
  - can be used for fixed-length records (sorted data)

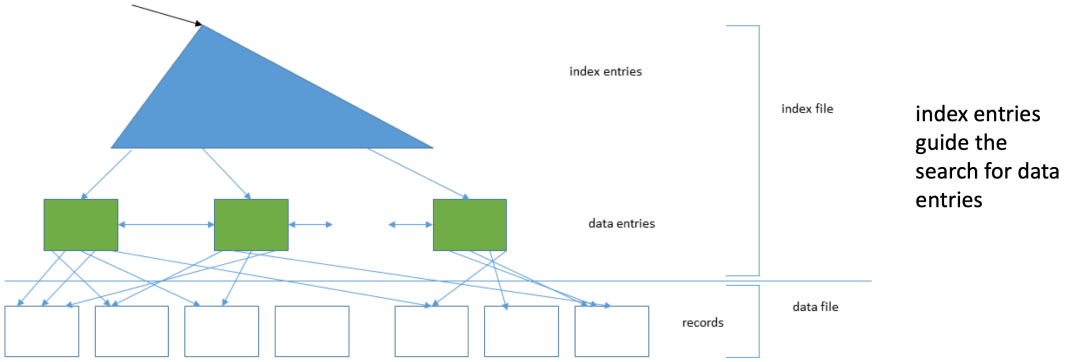


- Indexes
  - auxiliary data structure that speeds up the operations that can't be efficiently carried out given the file's organization
  - enables retrieval of rids of records that meet condition
  - search key
    - one or more attributes of the indexed file
    - indexes speed up equality/range selections on the search key
  - entries
    - record in the index <search_key, rid>
    - enable retrieval of records with a given search key value
  - indexes speed up certain queries (mainly the ones based on the search key)
  - organization techniques
    - B+ trees
    - hash-based structure
  - changes in the file → update associated indexes

- index size: as small as possible
  - brought into main memory
- let k* be data entry in index
  - A1: is actual data record where search_key = k
  - A2: is <K, rid> → rid is id of data record where search_key = k
  - A3: is <K, rid_list> → rid_list is list of data record where search_key = k

- A1
  - file of data records don’t need to be stored in addition to the index
  - index = special file organization
  - at most 1 A1 index/collection of records
- A2, A3
  - data entries point to corresponding data records
  - size of entry << size of record (generally)
  - A3 more compact than A2, but can contain variable-length records
  - can by used by several indexes
  - independent of file organization

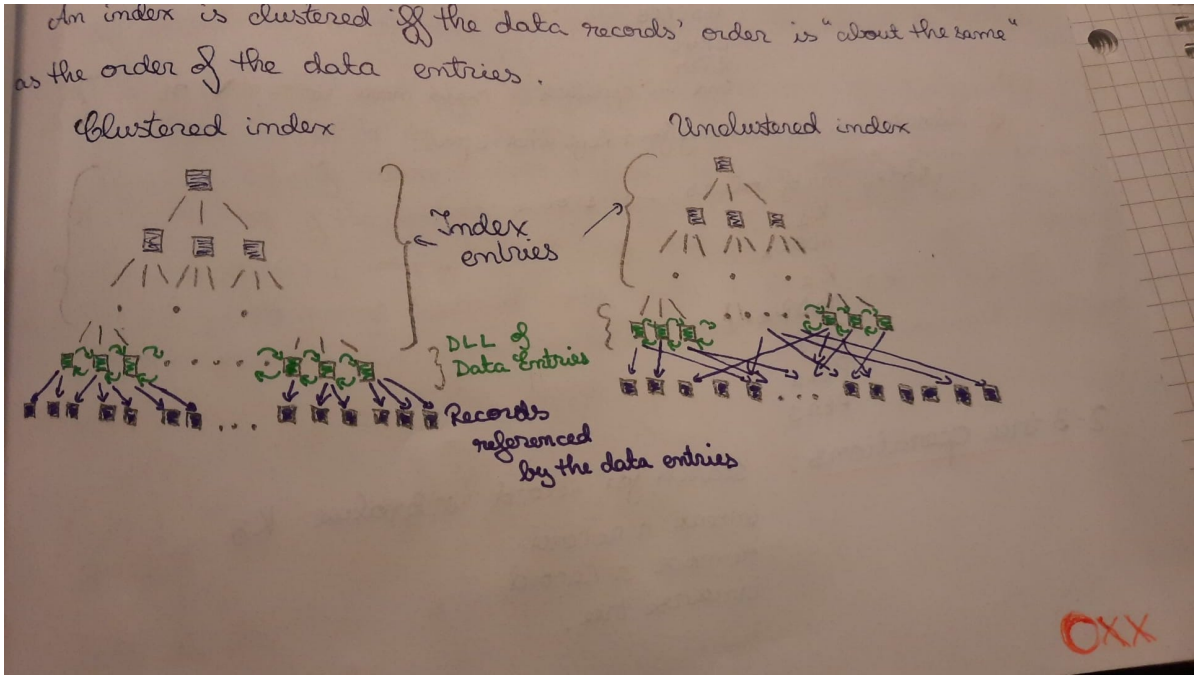
- Clustered/Nonclustered indexes
  - Clustered: the order of the data records is close to / the same as the order of the data entries
  - A1 → clustered, since the data entries are the data records
  - A2, A3 →
    - clustered only if data records are ordered by the search key
    - in practice, nonclustered as maintaining the order of file is expensive



unclustered index, alternative 2

- at most 1 clustured, several nonclustered
- range search query
  - in case of unclustered indexes, each data entry can contain rids that lead to distinct pages,
    - as such the number of IO operation could be equal to the number of data entries satisfying the condition
- Primary index → search key contains the primary key
- Secondary index → not primary
- Unique index → search key contains the candidate key
- Duplicates
  - same search key
  - unique, primary index can’t contain them

- Composite search key → search key containing several fields



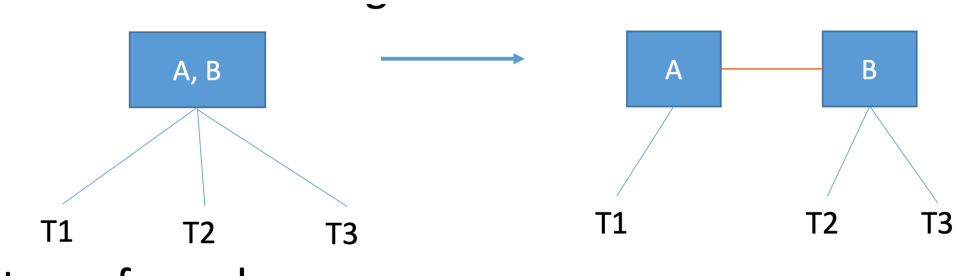
Lecture 11

- exam samples

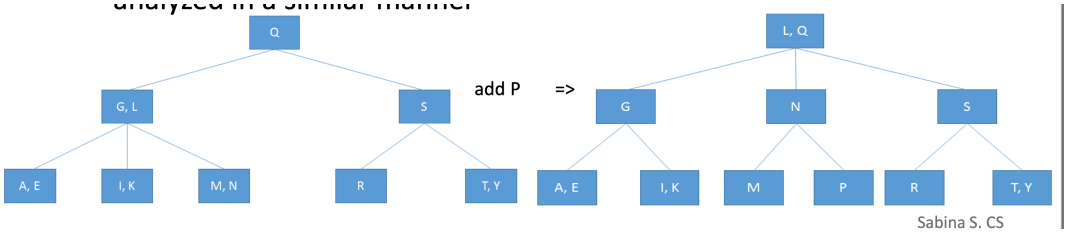
Lecture 12

- 2-3 trees
  - all terminal nodes on the same level
  - every node has 1 or 2 key values
    - non-terminal nodes with one value k has 2 subtrees
      - one with values less than k
      - the other with values larger than k
    - non-terminal nodes with 2 values k1 < k2 has 3 subtrees
      - one with values < k1
      - one with values in [k1, k2]
      - one with values > k2
  - (2,3 tree where all nodes have only one value → levelled binary search trees)
  - Storing

- tree → key value + address of the record
- 2 options
  - transform into binary tree
    - node with 2 values are transformed, with one unchanged
    - node structure (K:key value, ADDR:address of the record, PL and PR: pointer to the left and right subtrees, IND: specifies link to the right)



  - memory area for the node can store 2 values and 3 subtrees
    - node structure (NK: number values, K1 and ADDR1: first key value and correspond record address, K2 and ADDR2: second key value and correspond record address, P1 P2 and P3: pointers to the subtrees)
- add a value
  - values must be distinct → search first
  - if the terminal node has 1 value, the new value can be stored in the node
  - it the terminal node has 2 values, it is added, then sorted and then split into 2 node: one containing the smallest value, one with the largest value, and the middle value goes to the parent node where a similar process is executed



- delete a value
  1. if in inner node, change with neighbouring value value from terminal node
  2. perform this until a or b
     a. if the node is the root node or with 1 value, algorithm ends
     b. if delete empties a node and the sibling has 2 values, one of sibling's values goes to the parent node and one of parent's values is transferred to current node, algo ends
     c. else current node is merged with one of the siblings and a values from the parent value and the 2. is repeated for the parent

- B-trees
  - 2-3 trees generalization
  - B tree off order m
  - root not terminal → at least 2 subtrees
  - terminal nodes → same level
  - non terminal trees → at most m subtrees
  - node with p subtrees → p-1 ordered values
  - non terminal node → at least m/2 subtrees (celling if rational)
  - storing
    - turn in binary tree: same method as 2-3 trees
    - allocated area of a node allows max number of values

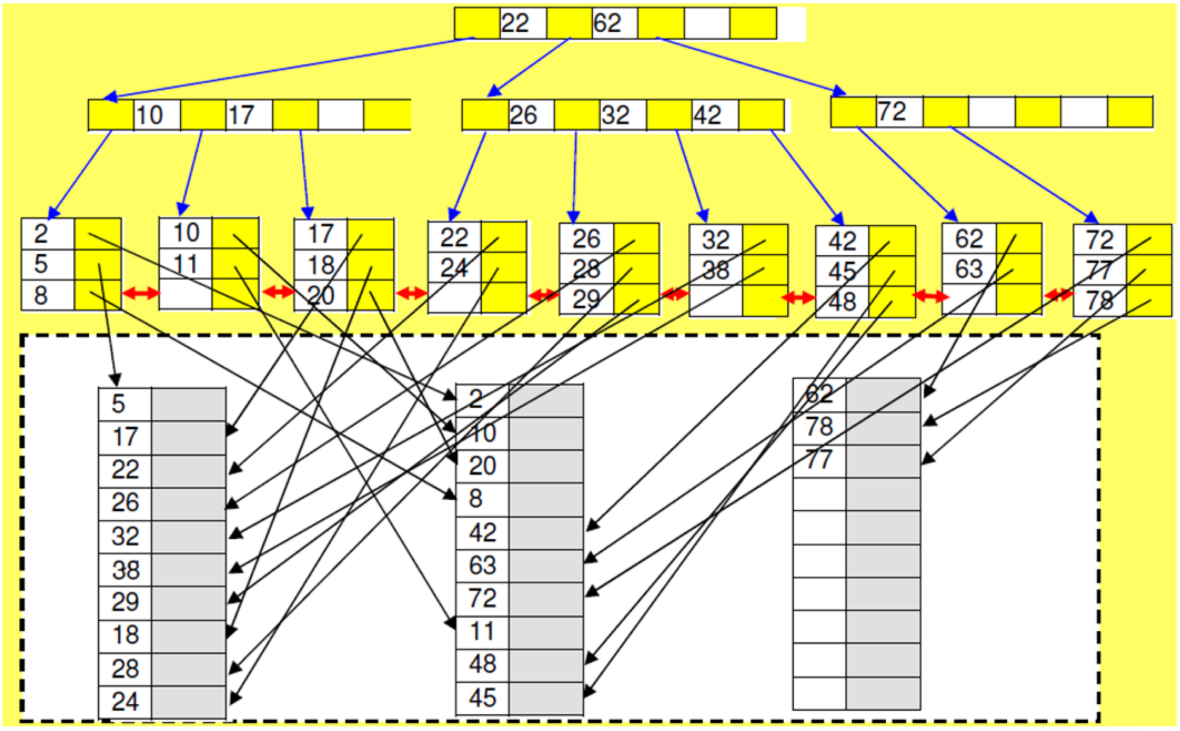
NV	K_1	$ADDR_1$	...	K_{m-1}	$ADDR_{m-1}$	$Pointer_1$	...	$Pointer_m$

- NV - number of values in the node
- K_1, ..., K_{m-1} - key values
- $ADDR_1$, ..., $ADDR_{m-1}$ - the records' addresses (corresponding to the key's values)
- $Pointer_1$, ..., $Pointer_m$ – subtree addresses

Sabina S. CS

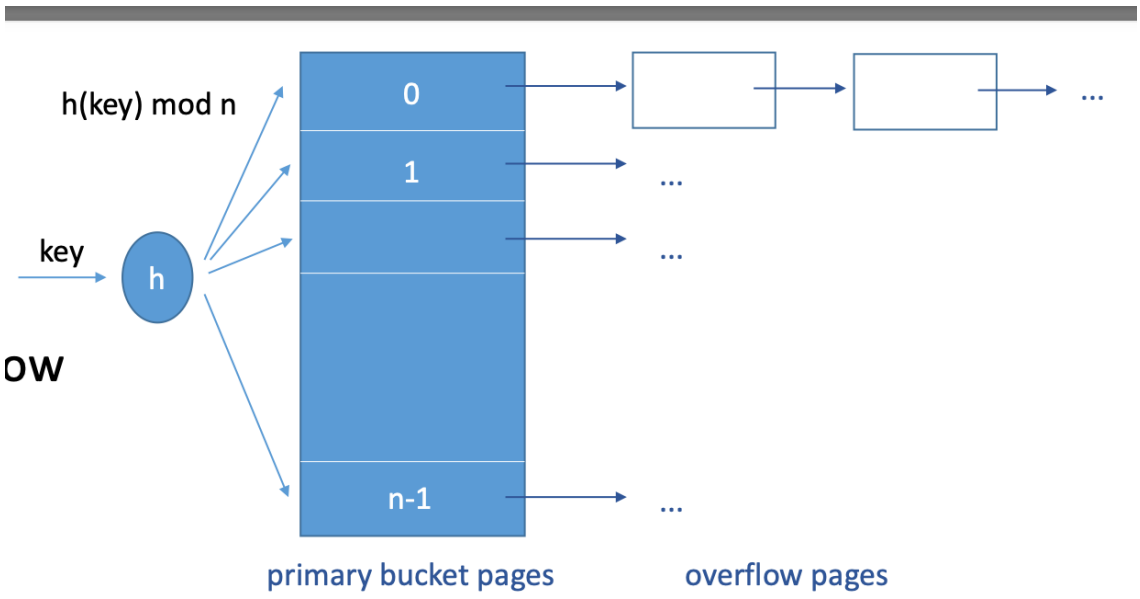
  - a block stores a node
  - the maximum number of required blocks when searching for a value is the maximum number of levels of the tree

- B+ trees
  - B-tree variant
  - last level contains all values
  - some key values appear in non-terminal nodes without record's address → separate values from the terminal nodes
  - terminal nodes are maintained as doubly linked lists



    - order → relaxed, replaced by physical space criterion
    - terminal and non-terminal nodes store different number of entities (generally, inner nodes more)
    - variable-length search_key → variable-length entries → variable number of entries / page
    - if A3 <search_key, RID_list> ⇒ variable-length entries (including duplicates)
    - larger key size → less index pages fit on a page → larger height
    - keys in index entries → often can be compressed
    - balanced indexing → uniform search times
    - rarely more than 3-5 levels → first 3 can be stored in main memory → veri few IO operation
    - widely used in DBMSs
    - ideal for range selections, good for equality selection
- Hash-based indexing
    - Hashing function → maps search key into a range of bucket numbers
    - hashed file
        - search key
        - records grouped into buckets
        - determine record’s bucket → apply hash func. on the search key
    - ideal for equality selection
    - Static hashing
        - 0 to n-1 buckets
        - One primary page, possible overflow pages/bucket
        - data entries : A1/A2/A3
        - search → apply hash func. on search tree → search bucket
        - good Hashing func.

        - h(val) = a*val + b
        - h(val) mod n
    - file grows → overflown chains
    - create new file with more buckets
    - problem: static number of buckets
    - solutions: periodic rehash, dynamic hashing



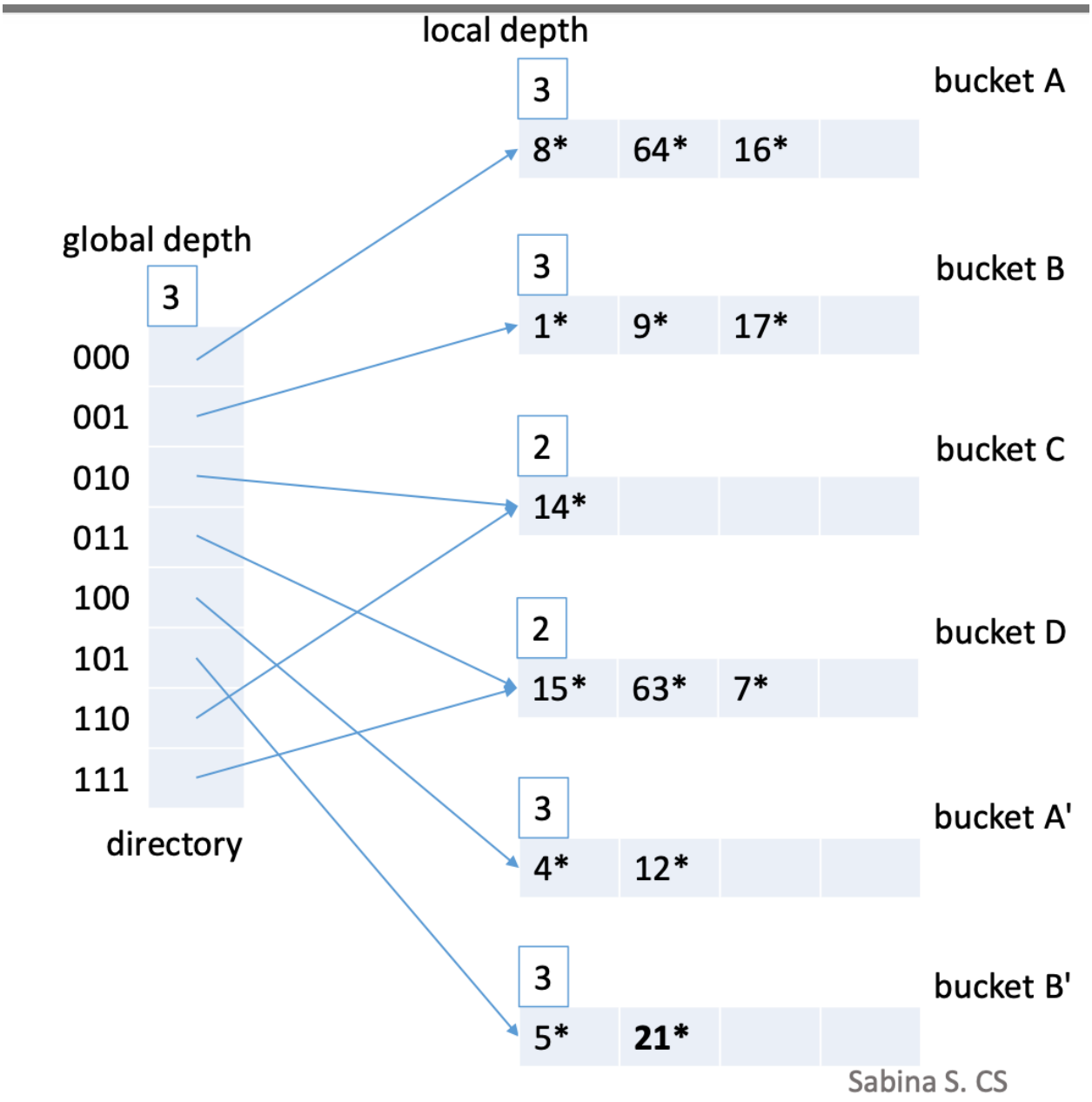
- Extendible hashing
    - dynamic hashing technique
    - directory of pointer to buckets
    - double the size of the number of buckets
        - double the directory
        - split overflowing buckets
    - directory that uses hashing func. to determine the right bucket → go to bucket
    - insertion →
        - if bucket full → split bucket

- if gd = ld → double the directory

- gd - global depth
- ld - local depth or a bucket
- 2^(gd-ld) elems. point to the bucket Bk with ld
  - if gd==ld and Bk split ⇒ double directory
- manage collision ⇒ overflow pages

bucket split accompanied by directory doubling

- allocate new bucket page nBk
- write nBk and bucket being split
- double directory array (which should be much smaller than file, since it has 1 page-id / element)
  - if using *least significant bits* (last gd bits) => efficient operation:
    - copy directory over
    - adjust split buckets' elements


local depth
global depth
3
000
001
010
011
100
101
110
111
directory
3
8*
64*
16*
bucket A
3
1*
9*
17*
bucket B
2
14*
bucket C
2
15*
63*
7*
bucket D
3
4*
12*
bucket A'
3
5*
21*
bucket B'
Sabina S. CS