Data Management and Databases

Spring 2021

Jean-Claude Graf

July 16, 2021

Contents

1 Introduction

1.1 Terminology

- **Database (DB):** collection of data
- **Database Management System (DBMS):** software for maintaining and utilizing a DB
  - ◇ Desired properties:
    - ∗ **Data Independance:** applications should not know how data are stored
    - ∗ **Declarative Efficient Data Access:** system should be efficient
    - ∗ **Transactional Access:** simulate each user that it is the only one interacting with the system
    - ∗ **Generic Abstraction:** users should not worry about all the issues above
  - ◇ Different flavours exists for different purposes

1.2 Different Models

- **Hierarchical Model**
  - ◇ Introduced by IBM
  - ◇ 1968 - today
  - ◇ Hierarchical structure of entities
  - ◇ Imperative query (say how we want it)
  - \- No data independence
  - \- No declarative efficient data access
- **Network Model**
  - ◇ 1969
  - ◇ Today in XML and JSON
  - ◇ Network of entities and relations
  - ◇ Imperative query (say how we want it)
  - \- No data independence
  - \- No declarative efficient data access
- **Relational Model**
  - ◇ 1969
  - ◇ One of the most popular today
  - ◇ Data stored in tables
  - ◇ Declarative query (say what we want)
  - \+ Data independence
  - \+ Declarative efficient data access

2 Relational Model

- Knowledge is represented as a *collection of facts*
- Inference is done using *mathematical logic*

2.1 Schema

- **Database Schema:**
  - Set of relation schema
- **Relation Schema:**
  - "Represented" as a table
  - Has a name
  - Contains a set of fields/attributes
  - Sometimes referred to as *Relation*
  - Described as $\mathrm{R}(\mathrm{f}_1 : \mathrm{D}_1, \ldots, \mathrm{f}_n : \mathrm{D}_n)$
    - **R:** relation name
    - **$\mathbf{f}_i$:** name of field i
    - **$\mathbf{D}_i$:** domain of field i
- **Field/Attribute:**
  - "Represented" as a single columns of the table
  - Has a name
  - Described by a domain (/type)
- Describes only the header (does not contain any content)
- Is not unique
  - Different schema have different advantages/disadvantages

2.1.1 Instance

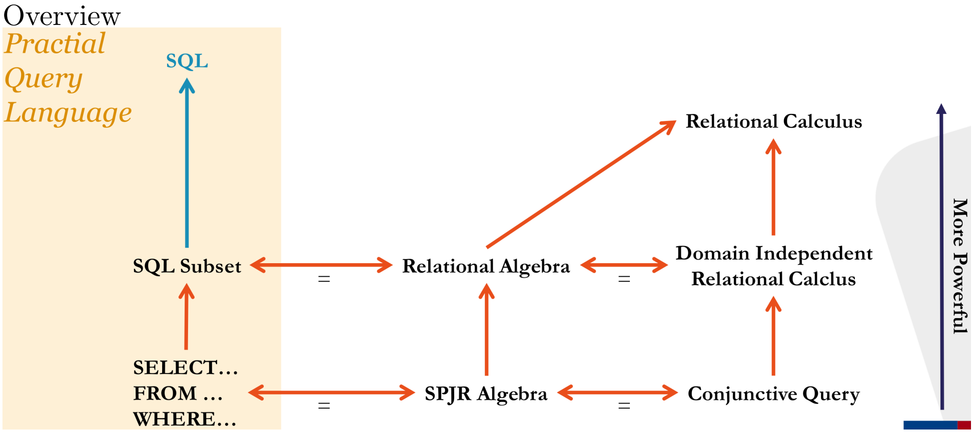
- "Represented" a set of rows in the table
- Set of tuples $\mathrm{I}_R \subseteq \mathrm{D}_1 \times \cdots \times \mathrm{D}_n$ for $\mathrm{R}(\mathrm{f}_1 : \mathrm{D}_1, \ldots, \mathrm{f}_n : \mathrm{D}_n)$
- **Domain Constraint:** For each filed the domain and the schema domain must match
- In practice a DB is a bag and not a set (allows duplicate entries)
  - But in theory we assume it is a set
- Attributes have no ordering principle, but ordering of attributes and tuples has to match

2.2 Key

- **Candidate Key:** minimal set of fields that uniquely identify a tuple
- **Primary Key:** one candidate key
  - Indicated by underlining the field: $\mathrm{R}(\underline{\mathrm{f}_1 : \mathrm{D}_1}, \ldots, \mathrm{f}_n : \mathrm{D}_n)$
  - Every relation must have one (but in a DB this is not required)
- **Key Constraint:**
  - The primary key must be unique in each an instance
  - All valid instances $\mathrm{I} \subseteq \mathrm{D}_k \times \mathrm{D}_a \times \mathrm{D}_b \wedge \forall (k, a, b), (k', a', b') \in I, k = k' \implies (a, b) = (a', b')$

3 Algebras

- Overview



4 Relational Algebra

- Imperative
  - Say how we want it
  - Different ways to implement the same query
- **Operators**
  - Create a new relation R' from one or two given relations R_1, R_2
  - **Union $\cup$:**
    - $x \in R_1 \cup R_2 \iff x \in R_1 \vee x \in R_2$
    - Schema of both operands must match
  - **Difference $-$:**
    - $x \in R_1 - R_2 \iff x \in R_1 \wedge \neg(x \in R_2)$
    - Schema of both operands must match
  - **Intersection $\cap$:**
    - $x \in R_1 \cap R_2 \iff x \in R_1 \wedge x \in R_2$
    - Schema of both operands must match
    - Follows from $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
  - **Selection σ:**
    - $x \in \sigma_c(R) \iff x \in R \wedge c(x) = True$
    - Return tuples which satisfy a given condition c
  - **Projection** π:
    - $\pi_{A_1,\ldots,A_n}(R)$ keeps only columns $A_1, \ldots A_n$
  - **Cartesian Product $\times$:**
    - $(x, y) \in R_1 \times R_2 \iff x \in R_1 \wedge y \in R_2$
    - If R_1 and R_2 have columns in common renaming is required
    - Rarely used in practice
  - **Renaming ρ:**
    - $\rho_{B_1,\ldots,B_n}(R)$ changes the name of the attributes to $B_1, \ldots B_n$
    - $\rho_S(R)$ changes the names of the attributes of R to the one of S
  - **Natural Join $\bowtie$:**
    - $R_1(A, B) \bowtie R_2(B, C) = \pi_{A,B,C}(\sigma_{R_1.B=R_2.B}(R_1 \times R_2))$
      - **No shared attributes:** Equivalent to cross product
      - **All attributes shared:** Equivalent to intersect
    - Is associative

- ◇ **Theta Join $\bowtie_\theta$:**
  - ∗ $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$
    - ○ θ can be any kind of condition
  - ∗ Flavour
    - ○ **Equi-Join:** $R_1 \bowtie_{A=B} R_2 = \sigma_{A=B}(R_1 \times R_2)$
- ◇ **Inner Join:**
  - ∗ Very similar to theta join with the difference that it keeps all matched columns
    - ○ I.e. matching columns are not collapsed into a single one
- ◇ **Outer Join:**
  - ∗ **Left Outer Join ⟕:** Natural join $\bigcup$ unmatched tuples from left operand
  - ∗ **Right Outer Join ⟖:** Natural join $\bigcup$ unmatched tuples from right operand
  - ∗ **Full Outer Join ⟗:** Natural join $\bigcup$ unmatched tuples from left and right operand
  - ∗ Unmatched tuples from the left/right are extended with `NULL` in the columns which do not match
- ◇ **Semi Join:**
  - ∗ **Left Semi Join $\ltimes$:** Tuples from left operand which match with any tuples from right operand
    - ○ $R_1(A_1, \dots, A_n) \ltimes R_2(B_1, \dots, B_2) = \Pi_{A_1,\dots,A_n}(R_1 \bowtie R_2)$
  - ∗ **Right Semi Join $\rtimes$:** Tuples from right operand which match with any tuples from left operand
- ◇ **Relational Division $\div$:**
  - ∗ $A \div B = C$ where C is the largest relation such that $B \times C \subseteq A$
  - ∗ Inverse of cross product

- **Expression:** Composition of multiple operations

\- No infinite (recursive) queries

- **Bags**
  - ◇ Real DBMS use bag semantic
  - ◇ Can have duplicates
  - ◇ Difficult to extend RA to bags
    - ∗ Need to add new operators (duplicate elimination)

5 Relational Calculus

- Declarative
  - ◇ Say what we want
- More powerful than RA
- **Database Schema:** $S = (R_1, \dots, R_m), R_i$ is a relation
- **Relation Schema:** $R(A_1 : D_1, \dots, A_n : D_n)$
- **Domain:** $\text{dom} = \bigcup_i D_i$
  - ◇ Infinite set of constants
- **Instance of Relation $\mathbf{R(A_1 : D_1, \dots, A_n : D_n)}$:** $I_R \subseteq \text{dom}^n$
  - ◇ I_R is finite
  - ◇ Set of facts over the relation
- **Instance of DB $\mathbf{S(R_1, \dots, R_m)}$:** Function $\mathcal{I}$ that maps relation to an instance of that relation
  - ◇ $\mathcal{I} : R_i \mapsto \mathcal{I}(R_i)$
  - ◇ Finite
  - ◇ Set of facts over all relations

- **Query Q_ϕ:** Has the form $Q_\phi = \{(x_1, \ldots, x_k) \mid \phi\}$
  - ϕ**:** First order formula (predicate) with free variables $x_1, \ldots, x_k$

  TODO: Full mathematical definition
- Output of queries may be infinite
- **Safe:** Query Q_ϕ which is finite for all $\mathcal{I}$
  - Undecidable problem if query is safe
- **Domain Independent Relational Calculus:** Query whose result does not depend on the interpretation of the relation and not on the domain
- **Active Domain:** $\text{adom}(Q_\phi, \mathcal{I})$ = all constaints in Q_ϕ and $\mathcal{I}$
- **Conjunctive Query:** $\phi = \exists y_1, \ldots, y_l(A_1 \wedge \cdots \wedge A_m), Q_\phi = \{(x_1, \ldots, x_n) \mid \phi\}, A_j$ is an atom
  - Has many good properties
- **SPJR Algebra:** Relational algebra with only select, project, join and rename operators

6 SQL

- Consists of three steps
  - ◇ Define the schema of the tables
  - ◇ Put information into the tables
  - ◇ Query the tables
- SQL is a family of standards
  - ◇ **Data Definition Language (DDL):**
  - ◇ **Data Manipulation Language(DML):**
  - ◇ **Query Language:**
- Released in 1974
- Constantly improved and extended
- Different DB engines implement slightly different standards
  - ◇ Important to choose the right engine for a certain project

6.1 DDL

- Defines schema
- Relation schema requires
  - ◇ Name
  - ◇ Set of columns
  - ◇ Type of columns
- Data Types
  - ◇ **char (n):** String of length n
    - ∗ Padded with whitespace to match length
  - ◇ **varchar (n):** String of length $\leq n$
  - ◇ **integer**
  - ◇ **blob of raw data**
  - ◇ **date**
  - ◇ etc.
- **Create Relation:**

```
CREATE TABLE Professor(
    PersNr integer,
    Name varchar (30),
    Level varchar(2),
    Room integer,
    PRIMARY KEY (PersNr));
```

- **Delete Relation:**

```
Drop TABLE Professor;
```

- **Add New Column:**

```
ALTER TABLE Professors ADD COLUMN (age integer);
```

  - ◇ Unclear what to insert into preexisting tuples for that relation
  - ◇ We can set a default value
    - ∗ If not provided it keeps the entry empty
- **Drop Column:**

```
ALTER TABLE Professor DROP COLUMN age;
```

6.2 DML

- Put and manipulation information
- **Extract, Transform, Load (ETL):** Used to populate DB automatically
  - ◇ Technique for populating DB
  - ◇ **E:** Get data from somewhere
  - ◇ **T:** Bring in right format
  - ◇ **L:** Insert into DB
  - ◇ Manual population is not feasible
- Manual population of DB
  - Error-Prone
  - Slow
    - ∗ Each query has to be parsed one-by-one
      - ◦ Even when all are pretty much equivalent
    - ∗ Constraints have to be checked for each query
- **Insert:**

```
INSERT INTO Student (PersNr, Name)
VALUES (1111, 'Fred');
```

- **Delete:**

```
DELETE Student
WHERE Semester > 13;
```

- **Update:**

```
UPDATE Student
SET Semester = Semester + 1;
```

- **From CSV:**

```
COPY Professors FROM '/profs.csv' WITH FORAMT csv;
```

6.3 Query Language

- **SELECT ... FROM ... WHERE ...**
  - ◇ **FROM:** List of relation whose cross product is taken
  - ◇ **WHERE:** Selection condition
  - ◇ **SELECT:** Projection
- Select uses bag semantic
  - ◇ **SELECT DISTINCT:** Set semantic over all fields
- Every RA expression can we written in *SQL Subset*
- **SQL Subset:**
  - ◇ **Base Query:** $\rho_{a_1,\dots,a_n}(\Pi_{A_1,\dots A_n}(\sigma_{P_1\wedge\dots\wedge P_m}(R_1 \times \dots \times R_k))) \approx$

```
SELECT A1 as a1 ... An as an
FROM R1 ... Rk
WHERE P1 AND P2 ... AND Pm;
```

  - ◇ **Union:** $R_1 \cup R_2$

```
(SQL1) UNION (SQL2);
```

    - ∗ Uses set-like semantic
    - ∗ **UNION ALL:** used bag semantic

- ◇ **Intersection:** $R_1 \cap R_2$

  (SQL1) INTERSECT (SQL2);
- ◇ **Difference:** $R_1 - R_2$

  (SQL1) EXCEPT (SQL2);
- ◇ **Selection:** $\sigma_c(R)$

  SELECT * FROM (SQL1) WHERE c;
- ◇ **Projection:** $\Pi_{A_1,\dots,A_n} R$

  SELECT A1, ..., An FROM (SQL1);
- ◇ **Cross Product:** $R_1 \times R_2$

  SELECT * FROM (SQL1), (SQL2);
- ◇ **Rename:** $\rho_{a,b,c} R$

  SELECT A as a, B as b, C as c FROM (SQL1);

- **Sorting**

  SELECT A, B FROM (SQL1) ORDER BY A DESC, B ASC;
  - ◇ ASC is default
- **Grouping, Aggregation**

  SELECT A, COUNT(*) FROM (SQL1) GROUP BY A;
  - ◇ Can only project to aggregated fields and aggregation function
  - ◇ **HAVING:** Filter similar to WHERE but for the aggregated field
- **EXISTS, ANY, ALL, SOME:**
  - ◇ Useful things
- **Snapshot Semantics:**
  - ◇ Problematic when deleting (updating) from a relation which we need in a subquery of that relation
  - ◇ Would lead to non-determinism
  - ◇ First all tuples which would get modified are marked
  - ◇ Then the updates are implemented

6.4 Tips

- Extract substring by index: substring(<string>, <base1startindex>, <length>)
- Operator giving bag:
  - ◇ SELECT column FROM table
  - ◇ SQL1 UNION ALL SQL2
- Operator giving set
  - ◇ SELECT DISTINCT column FROM table
    - ∗ We can also apply this to multiple columns. In that case the entries are combined and then duplicates filtered.
  - ◇ SQL1 UNION SQL2
- Get current datetime: NOW()
- Extract certain field from datetime: DATE_PART(<desired_field>, <source>)
  - ◇ desired_field can be anything like: day, hour, year etc.
- Data type conversion is done using <some_field>::<some_type>
- Casting data type using CAST(<some_filed> AS <some_type>)
- Round to n decimal points: ROUND(<source>, <n>)

7 Graphical Modelling

- Models an application
- Graphical way to represent entities and their relation
- Consists of three steps
  - ◇ **Conceptual Modeling:** Capture of domains to be represented
    - ∗ Create diagram from "real world"
    - ∗ We consider Entity Relation (ER) Model in this course
    - ∗ Specifies all DB instances that are valid/allowed in our application
  - ◇ **Logical Modeling:** Map concepts to a concrete logical representation
    - ∗ Convert diagram to table schema
  - ◇ **Physical Modeling:** Implementation in Hardware
    - ∗ Convert table to bits

7.1 Conceptual Modelling (ER-Diagram)

- **Formal Semantics**
  - ◇ Diagram defines valid DB instances
  - ◇ All values we can take $\mathcal{D} = \mathcal{B} \cup \Delta$
    - ∗ $\mathcal{B}$**:** Concrete values
      - ○ Int, String, Float, etc
    - ∗ $\boldsymbol{\Delta}$**:** Abstract values
      - ○ Correspond to an entity
  - ◇ **Entity Set** E**:** 1-ary Predicate $E(x)$
    - ∗ $E(x) =$ True if x is of Entity Type E
    - ∗ $E^{\mathcal{J}} \subseteq \Delta$
  - ◇ **Attribute** A**:** Binary Predicate $A(x, y)$
    - ∗ $A(e, a) =$ True if e has attribute a
    - ∗ $A^{\mathcal{J}} \subseteq \Delta \times \mathcal{B}$
  - ◇ n**-ary Relation** R**:** n-ary Predicate $R(x_1, \ldots, x_n)$
    - ∗ $R(x_1, \ldots, x_n) =$ True if $(x_1, \ldots, x_n)$ participate in R
    - ∗ $R^{\mathcal{J}} \subseteq \Delta^n$
  - ◇ Each subgraph introduces a first-order logic sentence
  - ◇ Entity E_1 and E_2 linked by relation R
    - ∗ $\forall x_1, x_2 \in \Delta . R(x_1, x_2) \implies E_1(x_1) \wedge E_2(x_2)$
  - ◇ Entity E with attribute A
    - ∗ $\forall x, E(x) \implies \underbrace{E^{=1}}_{\text{uniquely exists}} y . A(x, y) \wedge y \in \mathcal{B}$
- **Building Blocks**
  - ◇ **Entity:** Instance of an entity set which is distinguishable from other instances of the same set
  - ◇ **Entity Set:** Set of entities of the same "type"
    - ∗ Rectangular box
  - ◇ **Attributes:** Properties of a certain entity set
    - ∗ Round box
  - ◇ **Relationships:** Connection among ≥ 2 entity sets
    - ∗ Rhombus box
    - ∗ **Roles**
      - ○ Each entity set can have a role in a relation
      - ○ Label lines by the role the entity set is

  - **Key:** Minimal set of attributes which uniquely identify an entity in the entity set
    - **Candidate Key:** All possible sets of keys
    - **Primary Key:** One selected key
      - Every entity set must have one
      - Underlined
- **Cardinality**
  - Two main notations
  - **N/M-Notation**
    - **One to One (1/1):**
      - A is in a one to one relationship with B if:
        - $1A$ entity can only have one relation with a B entity and
        - $1B$ entity can only have one relation with an A entity
    - **One to Many (1/N):**
      - A is in a one to may relation with B if:
        - $1A$ can have relationships with multiple B entities and
        - $1B$ can only have one relation with an A entity
    - **Many to One (N/1):**
    - **Many to Many (N/M):**
      - A is in a many to many relation with B if:
        - $1A$ entity can have relationships with multiple B entities and
        - $1B$ entity can have relationships with multiple A entities
  - **(min, max)-Notation**
    - For a relation we give the min and the max value of relations one entity can have
    - Stronger than N/M-notation
    - * means infinity
    - (min, max) is written in opposite was to N/M
- **Weak Entity**
  - Some entity relation depends on other entity
    - I.e. it is not unique by itself
    - Can only be uniquely identified with the main entity
  - *Weak* entity is the one which depends on another
  - Indicated by dotted underline
  - Is a $1/M$ relationship
- **Generalisation**
  - Represent that a entity set is is an instance of another entity set
  - Entity A **is_a** entity of B
    - $A \subseteq B$
    - Draw an arrow from A to B
    - A shares Bs attributes and primary key
    - Are not enforced
      - I.e. possible that $B \notin A$
    - If A**is_a**B and C**is_a**B it is possible that $C \in A$ and $C \in B$
- There are many other flavours of ER
- **Design Principles**
  - Model should reflect the application we want to build
  - Avoid redundancy
  - Keep it as simple as possible; less entities is better
  - Entity if the concept has more than one relationship
  - Attribute if the concept has only one 1:1 relationship

  - ◇ Models are large, partition it

7.2 Logical Modelling

- Take ER-model and convert to relational model
- Some constraints get lost
- **Steps**
  1. **Entity Sets:** Become relations
  2. **Attributes:** Become attributes of the relation
  3. **Relationship:**
     - ◇ **Without Cardinality Constrain (or N:M):**
       - ∗ Become relation containing the attributes of all participating relations
       - ∗ The primary key of the relation are all the primary keys together
     - ◇ **With Cardinality Constraints:**
       - ∗ Very tricky
       - ∗ Become relation containing the attributes of all participating relations
       - ∗ The primary key of the relation are the keys of the entities with which the relation can be uniquely identified.Or the relation gets merged into the table on the *many* side.
     - ◇ **Role:** Can be used to distinguish columns with the same entity type.
       - ∗ Done by renaming the two columns appropriately
  4. **Weak Entity:**
     - ◇ Can be omitted
     - ◇ The week entity is modeled as a relation on its own with the primary key of the main relation and its own key
  5. **Generalisation:**
     - ◇ Two ways to represent this
     - ◇ Better way depends on application
     1) *Child* has its own relation and the *Parent* relation
     2) Each *Child* is a full blown relation containing all keys of the *parent* relation
        - Lot of redundant data if entity is multiple child and parent at the same time
        - Cannot constraint that entity is only on of them
- **Rezept**
  1. Convert entries to relations
     - ◇ All attributes of the model get attributes of the relation
     - ◇ All keys of the model get keys of the relation
  2. Convert relations to relations
     - ◇ All attributes of the model get attributes of the relation
     - ◇ All keys of the participating entries are attributes of the relation
     - ◇ Keys are the keys of the entities which uniquely identify the relation
     - ◇ Mark alternative keys
  3. Merge relation if it is $1:1, 1:N, N:1$ and it has the same key
  4. Do some other merging
  - ◇ Automatically generates at least 3NF
- Can be done (Semi-) automatically

8 Integrity Constraints

- Additional constraint to the key and domain constraint
- Makes sure changes are consistent
- Control the content of the date and its consistency
- Are enforced by the schema
- Can be defined when:
  - ◇ Creating the table (`CREATE table`)
  - ◇ Later (`ALTER table`)
- Checked at `INSERT` as well as `UPDATE`
  - ◇ For foreign key also on `DELETE`
- Check happen at tuple level and not at the semantic of the command
  - ◇ I.e. try to run it and see what happens instead of analysing the query
- Some check may fail or succeed depending on the order of the tuples
  - ◇ We have no influence on this

8.1 Types

- **NOT NULL**
  - ◇ Prevents attribute from being `NULL`
  - ◇ **Syntax:** `some_field any_type not null`
- **PRIMARY KEY**
  - ◇ Mark attribute as primary key
  - ◇ Must not be `NULL` and not empty
  - ◇ **Syntax:** `some_field any_type PRIMARY KEY`
  - ◇ If applied to a tuple, all field must not be `NULL`
  - ◇ **Syntax:** `PRIMARY KEY (field1, fiels2)`
- **UNIQUE**
  - ◇ Value must be unique or `NULL`
    - ∗ In contrast to `PRIMARY KEY` which cannot be `NULL`
  - ◇ Multiple entries may be `NULL` in the same column
  - ◇ Multiple fields can be marked as unique
  - ◇ **Syntax:** `some_field any_type UNIQUE`
  - ◇ Tuples of fields can be marked as unique
  - ◇ **Syntax:** `UNIQUE (field1, fiels2)`
- **CHECK**
  - ◇ Boolean check based on values of a single tuple
  - ◇ Reject if `False`
  - ◇ Accept if `True` or `Unknown`
  - ◇ Some engines treat check somewhat weirdly
  - ◇ Some engines allow subqueries to be part of a check
  - ◇ **Syntax:** `CHECK(some_expression_evaluation_to_bool)`
- **FOREIGN KEY**
  - ◇ Involve two relations
  - ◇ Field must be `NULL` or a valid reference to another table
  - ◇ The reference field is often a `PRIMARY KEY` or at least `UNIQUE`
  - ◇ **Referencing Table:** Table which references a tuple form another table
  - ◇ **Referenced Table:** Table being referenced by another table
  - ◇ **Syntax:** `FOREIGN KEY some_field any_type REFERENCES some_table(some_field)`
  - ◇ **Maintenance**

        - Changes to the referenced table influences the referencing table
            - And not the other way around!
            - On `UPDATE` or `DELETE`
        - Different ways of handling changes
        - **Cascade**
            - Propagate modification or delete
        - **Restrict**
            - Prevent modification or deletion if if violates constraint
                - By throwing an error
            - Check right after each command
        - **No Action**
            - Prevent modification or deletion if if violates constraint
                - By throwing an error
            - Check after a transaction
            - Is the default of PostgreSQL
            - Is equivalent to restrict in mySQL
        - **Set Default/Set Null:**
            - Set reference to default value or `NULL`
        - **Syntax:** `ON UPDATE method`, `ON DELETE method`
- Using `CONSTRAINT some_name` we can give a name to constraints
    - Useful in practice since it allows easy modification later on

9 Recursive Queries

- Repeatedly execute the same query
- Stop when it converges
  - I.e. when answer does not change
- Steps
  - Set R = Empty
  - Run (base query UNION recursive query) and set it as the new R
    - Repeat until R does not change
  - Query R
- SQL
  - ```
    WITH RECURSIVE some_table(some_attribute) AS
        (<base query>
    UNION
        <recursive query>)
    <Final Query involving some_table and other relations>
    ```
    - some_table(some_attribute) is the recursive table
      - Can have any number of attributes
    - <base query> is run only once at the beginning and populates some_table
    - <recursive query> queries some_table and updates it
    - <Final Query> run once after some_table converges
- Recursion can be infinite
  - Will not terminate
  - An error will be thrown
- UNION ALL makes bag semantics and may cause infinite recursion when replaces UNION
- Relational model is not well suited for recursion
  - SQL is based on FOL and FOL cannot express recursion
  - Other DBs types support recursion more nicely
    ```

10 NULL

- `NULL` is a state and not a value
  - Check: `some_value IS NULL`
    - `(NULL IS NULL) -> TRUE`
  - And not: `some_value = NULL`
    - `(NULL = NULL) -> UNKNOWN`
  - We cannot compare `NULL`, but we can check if it is `NULL`
- **Arithmetic:** Always gives `NULL` if an operand is `NULL`
- **Comparison:** Always gives `UNKNOWN` if one of the comparators is `NULL`
- **Logical operator:** Treats `NULL` as `UNKNOWN`
  - Returns `UNKNOWN` when the result depends on the concrete value of the variable assigned to `NULL`
- **Aggregation**
  - If `group by` a columns containing `NULL`s, `NULL` will be in one group
  - Most aggregation functions ignore the `NULL`
  - `Count(*)` ignores `NULL` not, `Count(column)` ignores it
- Some operators may introduce `NULL`
  - E.g. (Left/Right) outer join

11 Views

- It is an alias for a query
- Provides higher abstraction than relations
  - Provides logical data independence
- **Syntax:** `CREATE VIEW some_name AS some_query`
- Can be used similarly as a table
- DMDB convert statement containing a view into a statement without view
  - I.e. DMDB converts view to a select query
- Used for
  - **Privacy:** Give person access to a limited view and not the whole relation/DB
  - **Usability:** Simplify queries
- **Update View**
  - For base relation $R_1, \dots, R_n$ and view $V = Q(R_1, \dots, R_n)$
  - Update V into V' requires finding a set of updated to the base relation $(R'_1, \dots, R'_n) = f(R_1, \dots, R_n)$ s.t. $Q(R'_1, \dots, R'_n) = V'$
  - Not all view can be updated
    - Some data is missing
    - Primary key is missing
    - Update aggregates result
    - etc.
  - SQL tries to avoid indeterminism
  - SQL view is updatable iff:
    - Involved only one base relation
    - Involves the key of that base relation
    - Does no involve aggregates, group by or duplicate-elimination

12 Functional Dependency

- **Redundancy**
  - ◇ Keep same data in multiple relations and/or multiple tuples
  - \- Waste of storage space
  - \- Additional work to keep consistent
    - ∗ Else we get anomalies
  - \- Hard to keep consistent
  - \- Additional code to keep consistent
  - \+ Improve locality
  - \+ Better performance
  - \+ Fault tolerance
  - \+ Availability
- FD is one way to model and understand redundancy
- Models and helps to reason about redundant data
- **FD Definition**
  - ◇ For:
    - ∗ **Relation Schema:** $R(A : D_A, B : D_B, C : D_C, D : D_D)$
    - ∗ **Instance:** $R \subseteq D_A \times D_B \times D_C \times D_D$
  - ◇ Let $\alpha \subseteq R, \beta \subseteq R$
    - ∗ Subset of columns
  - ◇ **Functional Dependency** $\alpha \to \beta$**:** iff $\forall r, s \in R.\ r.\alpha = s.\alpha \implies r.\beta = s.\beta$
    - ∗ I.e. $\alpha \to \beta \iff$ for any two tuples r and s in DB instance R, if r and s share the same values on columns α, then they share the same values on column β
    - ∗ I.e. there is a mapping, mapping values in columns α to values in columns in β
    - ∗ **Notation:** $R \models \alpha \to \beta$ if R satisfies $\alpha \to \beta$
- **FD** $\alpha \to \beta$ **is minimal:** iff $\forall A \in \alpha.\ (\alpha \setminus \{A\}) \not\to \beta$
  - ◇ **Notation:** $\alpha \to \cdot\beta$
- **Keys**
  - ◇ **Superkey** is $\alpha \subseteq \mathcal{R} \iff \alpha \to \mathcal{R}$
    - ∗ I.e. if we know the value of columns α we know all values of all columns
    - ∗ All columns together are a trivial candidate key
  - ◇ **Candidate Key** is $\alpha \subseteq \mathcal{R} \iff \alpha \to \cdot\mathcal{R}$
    - ∗ I.e. a minimal super key
- **Implication/Inference**
  - ◇ Given:
    - ∗ Set F of some FDs on schema $\mathcal{R}$
    - ∗ FD $\alpha \to \beta$ which is $\notin F$
  - ◇ F **implies** $\alpha \to \beta$ if every relation instance R of $\mathcal{R}$ that satisfies all FDs on F also satisfies $\alpha \to \beta$
  - ◇ **Notation:** $F \models \alpha \to \beta$
- **Derivation**
  - ◇ Given:
    - ∗ Set F of some FDs on schema $\mathcal{R}$
    - ∗ FD $\alpha \to \beta$ which is $\notin F$
  - ◇ F **derives** $\alpha \to \beta$ if there is a derivation (using Armstrong's axioms) from F to $\alpha \to \beta$
  - ◇ **Notation:** $F \vdash \alpha \to \beta$
- **Closure**

- ◇ Given:
  - ∗ Set F of some FDs on schema $\mathcal{R}$
  - ∗ Set of attributes $\alpha \subseteq \mathcal{R}$
- ◇ **Closure** of α is the set of all attributes $y \in \mathcal{R}$, such that $\alpha \to y$ can be derived from F
- ◇ **Notation:** α^+
- ◇ $\alpha^+ = \{y \in \mathcal{R} \mid F \vdash \alpha \to y\}$
- ◇ $F \vdash \alpha \to \beta \iff \beta \subseteq \alpha^+$
- ◇ **Recipe:** Find α^+
  1) Set α^+ to α
  2) For all $\beta \in \alpha$ check if there is a $\beta \to \gamma$ and add γ to α^+ if so
  3) Repeat step 2 until α^+ converges

- **Minimal Basis/Cover**
  - ◇ **Goal:** Remove redundant FDs in a set of FDs
    - ∗ I.e. FDs which we can derive from the other FDs
  - ◇ Given: Set F of some FDs
  - ◇ A **Minimal Cover** of F is a set $G \subseteq F$ with:
    - ∗ $G \equiv F$
    - ∗ All FDs in G have the form $X \to \alpha$, where α is a single attribute
    - ∗ It is not possible to make G smaller by:
      - ○ $G \setminus \{X \to \alpha\} \not\equiv G, \ \forall X \to \alpha \in G$
        - ▷ I.e. remove a FD
      - ○ $(G \setminus \{X\alpha \to \beta\}) \cup \{X \to \beta\} \not\equiv G \ \forall X\alpha \to \beta \in G$
        - ▷ I.e. Remove an attribute from a FD
  - ◇ **Recipe:** Compute minimal basis
    1) Set G to the set of FDs obtained from F when decomposing the RHS of each FD to a single attribute
    2) Remove trivial FDs
       - ○ I.e. all $a \to b$ where $b \subseteq a$
    3) Remove all redundant attributes from the LHS of all FDs
       - ○ I.e. if $a \to b$ and $\exists x \in a$ such that $(a \setminus x) \to b$, replace $a \to b$ with $(a \setminus x) \to b$
    4) Remove all redundant FDs
       - ○ I.e. if $a \to b$ and $\{b\} \subseteq \text{Closure}(F \setminus \{a \to b\}, a)$, remove $a \to b$ from F
- **Equivalence**
  - ◇ Given: Set F and G of FDs on schema $\mathcal{R}$
  - ◇ F and G are **equivalent** if $F \models G$ and $G \models F$
  - ◇ **Notation:** $F \equiv G$
  - ◇ Cardinalities define FD
  - ◇ FD determine keys
- **Armstrong Axioms**
  - ◇ Axioms:
    - ∗ **Reflexivity:** $\alpha \subseteq \beta \implies \beta \to \alpha$
      - ○ Special case is $\mathcal{R} \to \alpha$
      - ○ Trivial FDs
    - ∗ **Augmentation:** $\alpha \to \beta \implies \alpha\gamma \to \beta\gamma$
      - ○ Where $\alpha\gamma := \alpha \cup \gamma$
    - ∗ **Transitivity:** $\alpha \to \beta \wedge \beta \to \gamma \implies \alpha \to \gamma$
  - ◇ These axioms are both:
    - ∗ **Sound:** $F \vdash \alpha \to \beta \implies F \models \alpha \to \beta$

        - **Complete:** $F \models \alpha \to \beta \implies F \vdash \alpha \to \beta$
    - All other FDs can be implied from these axioms
- **Other Rules**
    - **Union:** $\alpha \to \beta \wedge \alpha \to \gamma \implies \alpha \to \beta\gamma$
    - **Composition:** $\alpha \to \beta\gamma \implies \alpha \to \beta \wedge \alpha \to \gamma$
    - **Pseudo Transitivity:** $\alpha \to \beta \wedge \beta\gamma \to \theta \implies \alpha\gamma \to \theta$
- **Composition of Relations**
    - **Goal:** Split bad relations into ones containing only one concept
        - **Bad Relation:** Relation which combine several concepts
    - **Lossless Decomposition** Of R into $R_1, \ldots, R_n$ if $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$
        - For $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$, decomposition $R_1 = \Pi_{\mathcal{R}_1}(R), R_2 = \Pi_{\mathcal{R}_2}(R)$ is lossless, if $(\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_1 \vee (\mathcal{R}_1 \cap \mathcal{R}_2) \to \mathcal{R}_2$
    - **Recipe:** Show given decomposition R_1, R_2 is lossy
        - Decomposition is lossless iff $R = R_1 \bowtie R_2$
        - Find some instance of the relations which serves as a counterexample
    - **Recipe:** Show given decomposition R_1, R_2 is lossless
        - Decomposition is lossless if $(R_1 \cap R_2) \to R_i, i \in \{1, 2\}$
    - **Preservation of Dependencies:** $\mathrm{FD}(R)^+ = [\mathrm{FD}(R_1) \cup \cdots \cup \mathrm{FD}(R_n)]^+$

13 Normal Form

- **Normalisation:** Process of restructuring a DB to reduce data redundancy and improve data integrity
  - ◇ Done using a series of normal forms
- **Normal Forms:** Common way for normalization
- **First Normal Form (1NF)**
  - ◇ All values are of atomic domains
    - ∗ I.e. only int, double, char etc. and not tuples, lists etc.
- **Second Normal Form (2NF)**
  - ◇ R is in 2NF iff every non-key attribute is minimally dependent on every key
    - ∗ **Minimally Dependant:** No attribute depends on part of a key
    - ∗ I.e. none of the non-key attribute depends on part of a key
  - ◇ I.e. If $R(a, b, c, d)$ with primary key $\{a, b\}$ then $a \to c$ is not allowed
  - ◇ Alternative, less strong definition
  - ◇ R is in 2NF if for all $\alpha \to B$ at least one holds:
    - ∗ $B \in \alpha$
    - ∗ B is an attribute of at least one key
    - ∗ α is a superkey of R
    - ∗ No attribute in α is part of any key
  - \+ Improve insert, update and delete anomaly
  - \- Does not solve update and delete anomaly
    - ∗ Because we can have $C \to D$ where both are non-keys
  - ◇ Enforce 2NF by decomposing the relation into multiple relations
- **Third Normal Form (3NF)**
  - ◇ R is in 3NF iff for all $\alpha \to B$ at least one holds:
    - ∗ $B \in \alpha$
    - ∗ B is an attribute of at least one key
    - ∗ α is a superkey of R
  - ◇ I.e. If $\alpha \to \beta$ does not satisfy any of these conditions, α is a concept on its own
    - ∗ Gets rid of transitive dependency
  - \- Does not get rid of all redundant data
  - \+ Is lossless
  - \+ Preserves all dependencies
  - ◇ **Recipe Synthesis Algorithm:** Decompose $\mathcal{R}$ into $\mathcal{R}_1, \dots, \mathcal{R}_n$ according to 3NF
    1) Compute minimal basis F_c of F
    2) For all $\alpha \to \beta \in F_c$, create $R_{\alpha\cup\beta}(\alpha \cup \beta)$
    3) If none of the relations contains a superkey, add a relation with a key
    4) Eliminate R_α if there exists $R_{\alpha'}$ such that $\alpha \subseteq \alpha'$
- **Boyce-Cod Normal Form (BCNF)**
  - ◇ R is in BCNF iff for all $\alpha \to \beta$ at least one holds:
    - ∗ $B \in \alpha$
    - ∗ α is a superkey of R
  - ◇ I.e. Each relation stores the same information only once
  - \- Does not preserve all FDs
  - \- Does not get rid of all data redundancies
    - ∗ Only of all redundancies caused by FD
  - ◇ **Recipe Decomposition Algorithm:** Decompose $\mathcal{R}$ into $\mathcal{R}_1, \dots, \mathcal{R}_n$ according to BCNF

1) Set result to $\{\mathcal{R}\}$
2) If there is $\mathcal{R}_i$ with $\alpha \to \beta$ which is not in BCNF
   - $\mathcal{R}_i^1 = \alpha \cup \beta$
   - $\mathcal{R}_i^2 = \mathcal{R}_i \setminus \beta$
   - Result $=$ (Result $\setminus R_i) \cup \{\mathcal{R}_i^1, \mathcal{R}_i^2\}$
3) Repeat 2 as long as there are $\mathcal{R}_i$ which are not in BCNF

- **Non-First Normal Form (NFNF)**
  - Use an array to get rid of data redundancy
    - Will not be in 1NF
  - Can be used in SQL
- **4th Normal Form**
  - **Multi-Value Dependency (MVD)**
    - A is MVD on B and C means that the value of B does not have impact on the value of C, and that B and C can take multiple values for the same A.
    - **Notation:** $A \twoheadrightarrow B, A \twoheadrightarrow C$
    - $\alpha \twoheadrightarrow \beta$ for $R(\alpha, \beta, \gamma)$ iff
      - $\forall t_1, t_2 \in R, t_1.\alpha = t_2.\alpha \implies \exists t_3, t_4 \in R$:
        - $t_3.\alpha = t_4.\alpha = t_1.\alpha = t_2.\alpha$
        - $t_3.\beta = t_1.\beta; t_4.\beta = t_2.\beta$
        - $t_3.\gamma = t_2.\gamma; t_4.\gamma = t_1.\gamma$
    - **Intuitively:** Thinks about in terms of joins
      - $R(\alpha, \beta, \gamma)$ with $\alpha \twoheadrightarrow \beta$ can be decomposed into $R = R_1 \bowtie R_2$
        - $R_1 = \Pi_{\alpha,\beta} R$
        - $R_2 = \Pi_{\alpha,\gamma} R$
        - Is lossless if $\alpha \twoheadrightarrow \beta$ or $\alpha \twoheadrightarrow \gamma$
    - Can result in anomalies and redundancy
    - **Trivial:**
      - $\mathcal{R}(\alpha, \theta) : \alpha \twoheadrightarrow \alpha\theta$
        - I.e. $\alpha \twoheadrightarrow \mathcal{R}$
      - $\mathcal{R}(\alpha, \theta) : \alpha \twoheadrightarrow \theta$
        - I.e. $\alpha \twoheadrightarrow (\mathcal{R} \setminus \alpha)$
      - $\beta \subseteq \alpha \implies \alpha \twoheadrightarrow \beta$
    - **Promotion:** $\alpha \to \beta \implies \alpha \twoheadrightarrow \beta$
    - **Complement:** $\alpha \twoheadrightarrow \beta \implies \alpha \twoheadrightarrow (\mathcal{R} \setminus \alpha \setminus \beta)$
    - **Multi-Value Augmentation:** $\alpha \twoheadrightarrow \beta \wedge (\delta \subseteq \gamma) \implies \alpha\gamma \twoheadrightarrow \beta\delta$
    - **Multi-Value Transitivity:** $(\alpha \twoheadrightarrow \beta) \wedge (\beta \twoheadrightarrow \gamma) \implies \alpha \twoheadrightarrow \gamma$
    - Not all FD rules apply to MVD
      - Need to distinct between FD and MVD
  - Deals with MVD (and not FD)
  - R is 4NF iff for all $\alpha \twoheadrightarrow \beta$, at least one condition holds:
    - $\alpha \twoheadrightarrow \beta$ is trivial
    - α is a superkey of R
  - R in 4NF $\implies$ R in BCNF
  - **Recipe Decomposition Algorithm:** Decompose $\mathcal{R}$ into $\mathcal{R}_1, \ldots, \mathcal{R}_n$ according to 4NF
    1) Set result to $\{\mathcal{R}\}$
    2) If there is $\mathcal{R}_i$ with $\alpha \twoheadrightarrow \beta$ which is not in 4NF
       - $\mathcal{R}_i^1 = \alpha \cup \beta$
       - $\mathcal{R}_i^2 = \mathcal{R}_i \setminus \beta$

        - Result = (Result $\setminus R_i) \cup \{\mathcal{R}_i^1, \mathcal{R}_i^2\}$
    3) Repeat 2 as long as there are $\mathcal{R}_i$ which are not in 4NF

- $\overbrace{\underbrace{\overbrace{\text{1NF} \subset \text{2NF} \subset \text{3NF}}^{\text{preserve dependencies}} \subset \text{BCNF}}_{\text{deal with FD}} \subset \underbrace{\text{4NF}}_{\text{deal with MVD}}}^{\text{lossless}}$
- There are many different NF with different properties
- **Denormalisation**
    - ◇ Higher normalisation is not always better
    - ◇ Sometimes we deliberately denormalize a DB
    - + Faster due to better locality
    - - More redundant data

14 Analytic

- **This section is not exam relevant!** TODO: Finish summarizing this section
- Look at DB from a statistical view
- Useful for
  - Data mining
  - Machine Learning

14.1 Associated Rule Mining

- Aka Data Mining
- **Frequent Itemsets I:** Items appearing at least in s transaction together
  - s**:** Support threshold
- **Support:** Number of transactions containing all items of I
- **Association Rules**
  - We can say what item is likely in a transaction knowing parts of the transaction
  - $\{\mathbf{i_1}, \ldots \mathbf{i_k}\} \rightarrow \mathbf{j}$: If $i_1, \ldots i_k$ in transaction, then j in transaction
  - Allows creating suggestions/recommendations
  - There are many more such rules
    - Only interesting in relevant ones
  - **Confidence** c**:** Probability that transaction contains j given $i_1, \ldots, i_k$
    - $\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$
    - Number of transaction with $i_1, \ldots, i_k, j$ divided by the number of transaction with $i_1, \ldots, i_k$
  - Not all high-confidence rules are interesting
    - An item may be in many transaction anyways
  - **Interest:** Difference between its confidence and the fraction of baskets that contain j
    - $|$ confidence $- \#$ baskets with $j|$
    - Rules are interesting if value is $\geq\sim 5$
- **Mining Association Rules**
  - Find all frequent itemsets I
    - **Naive Algorithm:**
      - Brute force
      - Each itemset is a candidate
      - Time: $\mathcal{O}NMw$
      - Space: $\mathcal{O}M$
        - $M = 2^d$
    - **A-Priori**
      - **Idea:**
        - If itemset is frequent, then all of its subsets must also be frequent
        - If itemset is not frequent, then no superset will be frequent
        - Support of itemset never exceeds the support the its subsets
      - C_k**:** candidate itemset of size k
      - L_k**:** frequent itemset of size k
      - Steps
        - **Initial:** $k = 1, C_1 =$ all items
        - While C_k is not empty
          - Scan DB do find which itemsets in C_k are frequent and put them into L_k

        - Use L_k to generate a collection of candidate itemsets C_{k+1} of size $k+1$
            - Join two itemsets of size k that share the first $k-1$ items
            - Use principles to filter
        - $k = k + 1$
    - There are libraries for that
- For every subset A of I, generate a rule $A \to I/A$
- Output rules above the confidence threshold

14.2 Clustering

- Group objects into different categories
- Each cluster is a subset of TODO: ?
- **Clustering:** Set of clusters
- **Partition of Clustering:**
    - A division date objects into subsets (clusters) such that each data object is in exactly one subset
- **Hierarchical Grouping:**
    - Tree
- **Traditional Hierarchical Clustering:** Each is a subset of each other
- **Non-Traditional Hierarchical Clustering:** Different groups
- Types
- **Well-Separated Clusters:**
    -
- **Center-Based:**
    - Elements are closest to center of their own cluster
- **Contiguity-Base:**
    - Set of points such that a point in a cluster is closer to one or more other points in the cluster that to any points not in the cluster
- **Density-Based:**
    -
- **Conceptual Clusters:**
    - Cluster of objects which share, or not share a certain object
- **K-Means**
    - Partitional clustering approach
    - Each cluster is associated with a centroid
    - Each point is assigned to the cluster with the closest centroid
    - Number of clusters K is given
    -
    - Often uses Euclidean distance
    - Minimizing the Sum of Sqzuare Error (SSE)
    - NP-Hard if $d \geq 2$
    - Polynomial if $d = 1$
    - Can be estimated
    - Iterative algorithm

    ```
    Randomly select K point is the initial centroids
    repeat
        From K clusters by assigning all points to the closest centrid
        Recompute
    ```

    - Initial starting chaise results in different results

    * Do multiple runs and choose best result
  - ◇ Centroid depends on distance function
    * NP-hard to find centroid for some functions
  - \+ Will always converge
  - \+ Quick convergence
  - ◇ $\mathcal{O}nKId$
    * **n:** number of points
    * **I:** number of iterations
    * **K:** number of clusters
    * **d:** dimension
  - \- Problematic under certain conditions
  - ◇ Better control if start with many clusters and merge then later on into fewer
  - ◇ MADlib for SQL support

14.3 Classification

- Different from clustering
- Learning a target function f that maps attribute set x to one of the predefined class labels y
- Traiing set consits of records with known class labels
- Traiing set is used to build classigication model
-
-
- **Instance-Based Learning:**
  - ◇
- **Nearest Neighbour Classifier:**
  - ◇ Requirements
    * Set of stored records
    * Distance metrics to compute distance between records
    * The value of k
      - ◦ Number of nearest neighbours

15 DMDB System

- Complex application
- Simplification we consider
  - All data are stored on disk
  - Disk is larger than memory
  - Disk is slower than memory
  - Disk favours different access pattern
  - Single CPU
  - One relation is stored in a single file
- Overview
  - **Query Optimization:** SQL query to relational algebra converter
  - **Operator Execution:** Execute a relation algebra operation
  - **Access Methods:** Provides different ways of accessing data from a relation
  - **Buffer Pool Management:** Gives illusion that all data is stored on memory

15.1 Storage Hierarchy

- Storage is a hierarchy
- Challenge: keep CPU busy
- Rapidly changing
- Different hierarchies lead to different DB design
- **Hard Drive**
  - Plates spin
  - Plate is split into sectors of fixed size
  - Arm assembly is moved in or out to position a head on a desired track
  - Tracks under head makes a cylinder (kind off)
  - Only one head reads/writes at any time
  - Block size is a multiple of sector size
  - Performance
    - **Seek time t_s:** Moving arm to position disk head on track
      - $10 - 20ms$
    - **Rotate time t_r:** waiting for block to rotate under head
      - $10 - 20ms$
    - **Transfer time t_{tr}:** actually moving data to/from disk surface
      - $8KB/0.1ms$
    - **Random access D times:** $D(t_s + t_r + t_{tr})$
    - **Sequential access D time:** $t_s + t_r + Dt_{tr}$
- We consider abstraction HD $\rightarrow$ DRAM $\rightarrow$ CPU
- DRAM $\rightarrow$ CPU is much faster than HD $\rightarrow$ DRAM
  - We only consider HD $\rightarrow$ DRAM optimisation

15.2 Disk Manager

- Lowest level
- Used by higher levels for
  - Allocate/de-allocate pages
  - Read/Write pages
- Requests for sequential allocation must be satisfied
- Responsible for maintaining a database's files

- DB content is stored as one or multiple files
- Many DMDBs use the file system provided by the OS
  - People used to build custom file systems for DMDBs
- **Files** contain a collection of pages
- **Page** is a fixed-size block of data
  - Contains a collection of tuples
  - **page id:** unique identifier of each page
- A relation is stored as a collection of pages

15.2.1 File Layout

- How does a file manage all its pages?
- Unordered collection of pages
- Support record level operations
- We must keep track of:
  - the pages in the file
  - the records on each page
  - free space on each page
- **Heap File:**
  - Unordered collection of pages
  - Need to keep track where tuples are stored and of free space
  - Two ways to implement it
  - **Linked List**
    - **Header Page:** One single page
      - Kind of the root
      - Has to pointer to two linked lists
        - Free pages list
        - Data pages list
    - No global view on data
    - Performance
      - Assume
        - Directory fits in and is in memory
        - $\#Pages = D$
        - Pages are randomly allocated on disk
          - Models worst-case
      - **Insert:** $t_{s+r} + 2t_{trans}$
        - If page 1 has slot available
      - **Find Record:** By non-RID value (RID is a pair page, id and slot id)
        - $\frac{D}{2}(t_{r+s} + t_{trans})$
      - **Scan:**
        - $D(t_{s+r} + t_{trans})$
  - **Page Directory**
    - **Header Page:** Multiple pages
      - Each contains a list of pointer to data pages
    - Performance
      - Assume
        - Directory fits in and is in memory
        - $\#Pages = D$
        - Pages are sequentially allocated on disk
          - Models best case

        - **Insert:** $t_{s+r} + 2t_{trans}$
        - **Find Record:** By non-RID value (RID is a pair page, id and slot id)
            - $t_{s+r} + \frac{D}{2}t_{trans}$
        - **Scan:**
            - $t_{s+r} + Dt_{trans}$

15.2.2 Page Layout

- Page consists of
    - **Header:** Contains metadata like
        - Page size
        - DBMS version
        - Compression information
        - Encryption information
        - Checksum
    - **Data:** Actual tuples
- Header is at top of page
- Data starts after header
- Multiple strategies
- **Naive Strategy**
    - Page is split into slots of fixed size
        - One tuple goes into one slot
    - Header keeps track of occupied/free slots
    - \- Lots of space wasted when tuples are of different length
- **Slotted Page**
    - Record id = $<$ page id, slot # $>$
    - **Slot array:** Array of pointers and size of occupied slots
        - Comes right after the header
    - \+ Can move tuples on page without changing record id

15.2.3 Tuple layout

- Data access methods:
- **On-line transaction processing (OLTP):**
    - Simple query
    - Reads/writes a small amount of data related to a single entry
- **On-line analytical processing (OLAP):**
    - Complex queries
    - Read large portions of the DB spanning multiple entries
- Multiple implementations
- **Row Storage**
    - Store tuple together
    - Divided into
        - **Bitmap:** Indicates which attributes are `NULL` (somehow)
        - **Fixed-Length:**
            - Contains fixed-length fields
            - Arrangement and sizes are equal for all tuples
            - Can directly access the i-th field
        - **Variable-Length:**
            - Contains variable-length fields

- ◦ Two implementation
- ◦ **Field Delimited:** Special characters mark the end/start of fields
  - Access i-th fields requires scann of list
- ◦ **Field Offset Array:** Array where $A[i]$ contains the start of the i-th field
  - ▷ Stored at the beginning of the tuple
  - + Direct access to i-th field
- + All tuple information are together
- + Good for OLTP
- - Bad for OLAP
  - ∗ Read lots of data we do not care
- **Column Storage**
  - ◇ Store a whole column together
  - + Good for OLAP
  - - Slow for point queries (look for a single value), inserts, updates and deletes
  - - Bad for OLTP
  - + Easier data compression

15.3 Buffer Pool Management

- Buffer manager acts like the intermediate layer between the system and disk manager
- On fetch, check if desired page is in RAM. If not, bring to RAM and returns. Else directly return it.
- **Goal:** Provide illusion that all data in in RAM
- Page Replacement
  - ◇ If RAM is full, we need to evict a page
  - ◇ Eviction policy is of key importance
  - ◇ **Future access pattern known**
    - ∗ **Idea:** Evict block whose next access is farthest in the future
    - ∗ Called Belady's MIN algorithm
    - + Optimal under this assumption
  - ◇ **Future access pattern unknown**
    - ∗ Rarely know about the future access pattern
    - ∗ Different strategies
    - ∗ **Least Recently Used (LRU):** Evict the least recently used page
      - + Works well for repeated access to popular pages
      - - 100% miss under sequential flooding
        - ▷ **Sequential Flooding:** Access in a repeated pattern but such that not all pages fit into RAM
      - ◦ At most twice as bad compared to optimal when LRU has twice the memory TODO: What does this mean?
    - ∗ **Most Recently Used (MRU):** Evict the most recently used page
      - - Frequently accessed page has to be fetched often
        - ▷ E.g. Index scan
    - ∗ Access patterns
      - ◦ **Sequential:** Table Scan: $1 \to 2 \to 3 \to 4 \to \dots$
      - ◦ **Hierarchical:** Index Scan: $1 \to 4 \to 11 \to 1 \to 4 \to 12 \to 1 \to 3 \to 8 \dots$
      - ◦ **Random:** Index Lookup: $12 \to 9 \to 4 \to 21 \to 55 \to 6 \to 42 \to \dots$
      - ◦ **Cyclic:** Nested-Loop: $1 \to 2 \to 3 \to 4 \to 1 \to 2 \to 3 \to 4 \to \dots$
    - ∗ Depending on the pattern different strategies perform differently well
    - ∗ If we have some information about pattern, we can select an optimal policy

- This is the key different from the OS RAM manager
- OS vs DB
  - Principles are similar, but in DB we know more what is going on
  - Allows for better optimisation
  - DB uses own buffer
  - DB uses OS's filesystem
    - Has own buffer which can cause problems

15.4 Access Methods

- Used by upper layer to access information in tables
- Provides different was of accessing data from a relation
- Goal: find a tuple whose attribute X equals Y
- **Sequential Scan:**
  - Bring page 1 and scan, bring page 2 and scan etc.
  - Cost:
    - When pages are sequentially allocated: $t_{s+r} + D(t_{tr})$
    - When pages are randomly allocated: $D(t_{s+r} + t_{tr}$
    - When write back on scan pay extra: t_{tr}
- Improve: build data structure f over relation, which can easily answers our scan
  - Evaluation of f is usually cheaper than sequential scan
- **B-Tree (B+ Tree)**
  - Self-balancing tree that keeps data sorted
  - Properties
    - M-way search tree
      - Like a binary search tree but each node has up to M children
    - Perfectly balanced
    - Every inner node (except root) is at least half-full
    - Every inner node with k keys has $k + 1$ non-null children
    - Each page is represented as a node
    - $\mathcal{O}(\log n)$ for search, insertion, deletion
  - Two flavours
    - **Unclustered B+ Tree:** Leaf node contains RID (PageID, SlotID), pointing to the relation
    - **Clustered B+ Tree:** Lead node contains the actual tuple
      - Can have only one clustered B+ Tree per relation
  - **Point query**
    - Find single key
    - Node size $= M$, # Tuples $= N$
    - Depth $\mathcal{O}(\log_M N)$
    - Num I/O
      - Unclustered: $\underbrace{\log_M N}_{\text{search tree}} + \underbrace{1}_{\text{access actual tuple}}$
      - Clustered: $\underbrace{\log_M n}_{\text{search tree}}$
    - Much faster than sequential scan
  - **Range query**
    - Find all tuples in range
    - Node size $= M$, # Tuples $= N$
    - Depth $\mathcal{O}(\log_M N)$

- Num I/O
  - Unclusered: $\underbrace{\log_M N}_{\text{search tree}} + \underbrace{\frac{\#\text{tuples}}{\text{tuples-per-page1}}}_{\#\text{ leafs to read to get all RIDs}} + \underbrace{\#\text{tuples}}_{\text{cost of reading}}$
  - Clustered: $\underbrace{\log_M N}_{\text{search tree}} + \underbrace{\frac{\#\text{tuples}}{\text{tuple-per-page2}}}_{\#\text{ leafs to read to bet all RIDs}}$
  - tuple-per-page2 < tuple-per-page1 since clustered contains the actual tuples (which are larger than RID)

- **Insert**
  - Insert a tuple
  - Algorithm
    - Find the right leaf node L
    - Put data in L
      - If L has enough space we are done
      - Otherwise, split L, insert key to the parent of L
- **Delete**
  - Delete a tuple
  - Algorithm
    - Find the right leaf node L
    - Remove data in L
      - If L is at least half full we are done
      - Otherwise, merge two leaf nodes or borrow one tuple from neighbours, update parents
- Heap file vs B+ Tree
  - **Heap file:**
    + Lot of sequential scans
  - **B+ Tree:**
    + Small number of random access
  - Tradeoff, between few expensive, or many cheap accesses
- Use Btree: `CREATE INDEX name ON table USING btree(column);`
- Bulk build
  - Different approaches
  - Insert tuple by tuple
    - Slow
  - Sort and insert tuple bottom-up

+ Query time is independent of data distribution

- **Hash Table**
  - Goal: Do better than B+ Trees for point queries
  - **Hash Function f(x):** Maps each object into an entry in the table
    - E.g. For integer: $h(x) = (ax + b) \mod p$
    - E.g. For Strings: $h(s_1, \ldots, s_n) = (\sum_i s_i a^i) \mod p$
    - Hard to find ideal hash function
  - Ideally $a \neq b \implies f(a) \neq f(b)$
    - Else we get a collision
  - Different approaches to prevent collisions
  - **Closed Hashing:** We know how many elements are trying to index
    - **Linear Probe**
      - Working

- ▷ Compute hash to find right slot
- ▷ Insert object into next empty slot
  - Deletion is non-trivial
    - · We have to insert special marker into delete position to mark that this is a contiguous block
- ∘ Search/Insert: $\mathcal{O}$(size of largest cluster)
  - ▷ **Cluster:** Largest consecutive sequence of occupied slots
- ∘ For hash table
  - ▷ Size m
  - ▷ Contains $n = \lambda m$ keys
  - ▷ Full to $\lambda\%$

  we have on average:
  - ▷ **Insert:** $\frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$
  - ▷ **(Successful) search:** $\frac{1}{2}(1 + \frac{1}{(1-\lambda)})$
    - · Good if $\lambda \approx 50\%$ full
- \+ Very cache-efficient
- \- Very sensitive to hash function
  - ▷ Hash function must be "good"

◇ **Open Hashing:** We do not know how many elements are there
- ∗ **Chained Hashing**
  - ∘ Working
    - ▷ Compute hash to find right slot
    - ▷ Insert if no collision
    - ▷ Else append to linked list
  - ∘ Expected length of chain for
    - ▷ $h(x)$ is uniformly random
    - ▷ $m = \mathcal{O}(N)$
    - ▷ N: Number of slots

    is $\mathcal{O}(1)$

15.5 Operator Execution

- Execute a relational algebra operator
- Used different access methods to implement these operators
- **Select $\sigma_{\mathbf{C}}(\mathbf{R})$:**
  - ◇ Input R, condition C
  - ◇ Assume
    - ∗ R has $|R|$ tuples and $B(R)$ pages
    - ∗ Buffer size: M pages
  - ◇ **Selectivity:** $\alpha(C, R)$
    - ∗ Is a constant
    - ∗ Number of tuples in R that satisfy condition C divided by $|R|$
  - ◇ **Sequential Scan (Clustered) Heap File**
    - ∗ Bring pages to RAM one by one
    - ∗ Scan each tuple and check for the predicate C
    - ∗ If true, output tuple
    - ∗ Total cost: $\mathcal{O}(\underbrace{B(R)}_{\text{read}} + \underbrace{\alpha(C,R)B(R)}_{\text{write}})$
    - ∗ Cost is different for each query

- ◇ **Index Scan**
  - ∗ If $C \in \{<, >, =\}$
  - ∗ **Unclustered B+ Tree**
    - ∘ Steps
      - ▷ Find the right leaf node
      - ▷ Scan the index
      - ▷ Fetch and return corresponding tuple from heap file
    - ∘ Total Cost: $\mathcal{O}(\underbrace{\log|R|}_{\text{find leaf}} + \underbrace{\alpha(C,R)|R|}_{\text{fetch tuple}} + \underbrace{\alpha(C,R)B(R)}_{\text{write}})$
  - ∗ **Clustered B+ Tree**
    - ∘ Steps
      - ▷ Find the right leaf node
      - ▷ Scan the index
      - ▷ Return tuple
    - ∘ Total Cost: $\mathcal{O}(\underbrace{\log|R|}_{\text{find leaf}} + \underbrace{\alpha(C,R)B(R)}_{\text{fetch tuple}} + \underbrace{\alpha(C,R)B(R)}_{\text{write}})$

- **Sort**
  - ◇ `Sort(R, Attribute A)`
  - ◇ **Clustered B+ Tree**
    - ∗ Sorted leaves and constant access time
  - ◇ **Unclustered B+ Tree**
    - ∗ Sorted leafs but one random access per tuple
  - ◇ Different sorting algorithms
  - ◇ Assume: Data N is much larger than buffer B
  - ◇ **External Sort**
    - ∗ If $B = 3$ we can sort 2 pages
    - ∗ Working
      - ∘ If $N < B$
        - ▷ Run quicksort
      - ∘ Otherwise
        - ▷ Phase 1: Sorting
          - · Partition file into smaller chunks that fit into memory
          - · Sort smaller junks
        - ▷ Phase 2: Merging
          - · Combine smaller, sorted chunks into large file
    - ∗ TODO: Add Cost

- **Join S $\bowtie_\theta$ S**
  - ◇ **Nested Loop Join**
    - ∗
```
foreach tuple r in R:
    foreach tuple s in S:
        if Theta(r, s):
            output r, s
```
    - ∗ $M = 3$: $\mathcal{O}(B(R) + |R|B(S) + \text{print IO})$
    - ∗ Order of tables matters
      - ∘ Depends on table size, buffer size
      - ∘ Small M: Better if smaller relation is in outer loop
      - ∘ Large M (assume smaller table is fully cached): Better if smaller relation is in the inner loop
    - ∗ **Efficient if:** Both relations fit into memory

- ◇ **Block Nested Loop Join**
  - ∗
```
foreach block BR in R:
    foreach block BS in S:
        foreach tuple r in BR:
            foreach tuple s in BS:
                if Theta(r, s):
                    output r, s
```
  - ∗ $M = 3$: $\mathcal{O}(B(R) + B(R)B(S))$
  - ∗ $M > 3$: $\mathcal{O}(B(S) + B(R)\frac{B(S)}{M-2})$
  - ∗ Partition S into $a = \frac{B(S)}{M-2}$ junks
    - ◦ Fully cache junk
    - ◦ Pay $B(R)$ to join this junk with R: $(M-2) + B(R)$
    - ◦ Repeat $\frac{B(S)}{M-2}$ times TODO: Not sure what this means
- ◇ **Index Nested Loop Join**
  - ∗
```
foreach tuple r in R:
    foreach tuple s in IndexScan(S, r, Theta):
        output r, s
```
  - ∗ $\mathcal{O}(B(R) + |R|C)$
    - ◦ C is the cost of lookup in the index
- ◇ **Sort Merge Join**
  - ∗ Assume both relations are sorted
  - ∗ Working
    - ◦ Scan both relation
    - ◦ Compare to head
  - ∗ Cost $\mathcal{O}(B(R) + B(S) + \text{Sort}(R) + \text{Sort}(S))$
  - ∗ **Efficient if:** Index in attribute is present
- ◇ **Hash Join**
  - ∗
```
build Hash Table HT for R
foreach tuple s in S:
    if h(s) in HT:
        check forall r where h(r) = h(s) if r = s
            output
```
  - ∗ Cost $\mathcal{O}(B(S) + B(R))$
    - ◦ Assumed hash table fits into memory
  - \- Not good when hash table does not fit into memory
  - ∗ **Efficient if:** Result of join fits into memory
- ◇ **Grace Hash Join**
  - \+ Deals with the case where the hash table does not fit into memory
  - ∗ Idea
    - ◦ Partition R and S by hashing them using $h1$
      - ▷ Matching tuples of R and S are mapped into the same partition
      - ▷ Only one partition of R and S into memory and compare them
    - ◦ Rehash the hashes using $h2$ and check for equality
  - ∗ Cost $\mathcal{O}(3B(R) + 3B(S))$

15.5.1 Query Optimizer

- Given a SQL query generate a good execution plan.
  - ◇ **Execution Plan:** Tree of relational algebra operators

- Used query executor to actually execute the relational algebra operators
- Terminology
  - ◇ **Logical Plan:** What the user logically wants
  - ◇ **Physical Plan:** What the DMBS can understand and run
- Steps
  - ◇ User gives SQL query
  - ◇ Parse SQL to logical plan
  - ◇ Convert to physical plan
  - ◇ Run each operator using the operator execution
- **Execution Model:** How different operators are gut together
  - ◇ Different ways to put operators together
  - ◇ **Iterator Model**
    - ∗ Each operator is an iterator
      - ∘ **Input:** Set of streams of tuples
      - ∘ **Output:** Stream of tuples
      - ∘ Calling `next()` on a iterator returns its result
    - ∗ **Query Plan:** Is tree of iterators
      - ∘ Result s returned by calling `root.next()` over and over
    - ∗ **Volcano Model:** Data flow flow bottom to top
    - \+ Generic interface for all operators
    - \+ Easy to implement iterators
    - \+ No overheads in terms of main memory
    - \+ Supports pipelining
    - \+ Supports parallelism and distribution
    - \- High overhead of method calls
    - \- Poor instruction cache locality
  - ◇ **Materialization Model**
    - ∗ Each operator processes its inputs all at once and then emits its output all at once
      - ∘ **Input:** Full relation
      - ∘ **Output:** Full relation
    - ∗ Good when the intermediate result is not too much larger than the final result
      - \+ Good for OLTP
      - \- Bad for OLAP
  - ◇ **Vectorization Model:**
    - ∗ Similar to iterator model
    - ∗ Each operator returns a batch of tuples
      - ∘ Instead of a single tuple
    - \+ Good for OLAP
    - \+ Allows for vectored instructions to process batches of tuples
- **Cost Model:** How to estimate the cost of each physical plan given our execution model
  - ◇ Cannot be calculated but only estimate
  - ◇ Estimated based on lower level (operator execution)
  - ◇ Key variable to estimate is selectivity
    - ∗ Hard to estimate selectivity
  - ◇ **Cardinality Estimation**
    - ∗ How many tuples does a query involve?
    - ∗ **Histogram**
      - ∘ Create histograms from the data

- ◦ Lookup histograms to get an estimate of the number of tuples
- \- Hard to combine multiple histograms into one
  - ▷ Due to missing correlation
  - ▷ Assume they are independent
- ◦ For continuous values use bins
- ◦ We can also create multidimensional histograms
- ∗ Can benefit form machine learning

- **Space Space:** What are the logically equivalent sets of physical plans?
  - ◇ Given an input logical plane, there are different ways that one construct a physical plan
    - ∗ I.e. different orders of operators
  - ◇ **Query rewriting Rules:** Set of transformations
  - ◇ **Input:** Relational algebra expression E
  - ◇ **Output:** Relational algebra expression E'
  - ◇ **Property:** E is equivalent to E'
    - ∗ $\forall I \in \underbrace{\mathbf{I}}_{\text{Set of possible DB instances}}, E(I) = E'(I)$
  - ◇ Many different rules are possible
    1) Conjunctive selection operations can be deconstructed into a sequence of individual selections
       - ◦ $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
    2) Selection operations are commutative
       - ◦ $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
    3) Only the last in a sequence of projections operations is needed
       - ◦ $\Pi_{t_1}(\Pi_{t_2}(E)) = \Pi_{t_1}(E)$
    4) Selections can be combined with Cartesian products and theta joins
       - ◦ $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
       - ◦ $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
    5) Theta-join and natural join operations are commutative
       - ◦ $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
    6) Natural join operations are associative
       - ◦ $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

    6b) Theta join operators are associative
       - ◦ $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$ when θ_2 involves only attributes from E_2 and E_3

    7) Pushdown Selection
       - ◦ $\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie (E_2)$ if θ only involves attributes in E_1
    8) The projections operation distributes over the theta join operation
       - ◦ $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$ if θ only involves attributes from $L_1 \cup L_2$
    9) The set operations union and intersection are commutitive
       - ◦ $E_1 \cup E_2 = E_2 \cup E_1$
       - ◦ $E_1 \cap E_2 = E_2 \cap E_1$
    10) Set union and intersection are associative
       - ◦ $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
       - ◦ $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
    11) Selection operation distrubites over $\cup, \cap$ and $\setminus$
       - ◦ $\sigma_\theta(E_1 \setminus E_2) = \sigma_\theta(E_1) \setminus \sigma_\theta(E_2) = \sigma_\theta(E_1) \setminus E_2$
       - ◦ $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2) = \sigma_\theta(E_1) \cap E_2$

        - $\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2) \neq \sigma_\theta(E_1) \cup E_2$
    12) Projection operation distributes over union
        - $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$
- **Search Algorithm:** How can we search the best physical plan, given cost model?
    - Searching for the optimal query is hard
    - **Compromise 1:** Constraint search space
        - Only consider left-deep join trees
        - Allows fully pipelined plans
            - Intermediate values do not need to be written to temporary files
            - Not all trees are fully pipelined (e.g. Sort Merge Join)
        - Process:
            - Enumerate join orders
                - Only different left deep trees
            - Enumerate plans for each operator
            - Enumerate access method for each operator
        - Complexity $\mathcal{O}(n!)$, n is # relations
        - Can be done with DP
    - **Compromise 2:** Heuristic-based Optimisation
        - Optimize query-tree by applying a set of rules that typically improve execution performance
            - Early selection
            - Early projection
            - Restrictive selections and joins before other similar operators
        - Search algorithm that enables pipelining
- `EXPLAIN` gives actual physical model

15.6 Tuning

- DBMS have many internal parameters
- Improve performance

16 Transaction

- **Motivation**
  - ◇ Assume:
    - ∗ DB is a collection of objects
      - ○ I.e. one tuple is one object
    - ∗ Objects are fixed
      - ○ We cannot create new ones or delete old ones
    - ∗ System has only a single CPU
      - ○ CPU can only run one instruction at the time
  - ◇ If not dealt with correctly, simultaneous transactions may get mixed and we get wrong results
  - ◇ **Concurrent DB Access**
    - ∗ **Schedule:** One way of mixing instructions
      - ○ Different schedules may result in different results
    - ∗ Result of one query may be overwritten partly or completely
    - ∗ **Attribute-level Inconsistency:** Concurrent change of a single attribute of the same tuple
    - ∗ **Tuple-level Inconsistency:** Concurrent change of different attributes of the same tuple
    - ∗ **Table-level Inconsistency:** Concurrent change of full relation
    - ∗ **Multi-statement Inconsistency:** Interleaving of concurrent queries
    - ∗ When multiple groups of SQL statements are running at the same time, we want the effect as if they are executed sequentially
  - ◇ **System Failure**
    - ∗ Many thing which can break in a real system
    - ∗ We want that all or none changes apply, but not partial application
- **Transaction:** Collection of instructions which should not mix with other transactions
  - ◇ Concurrent transactions appear to run in isolation
  - ◇ On a crash, transaction changes appear entirely or not at all
  - ◇ `BEGIN; ..... COMMIT`: Encapsulates a transaction
    - ∗ Transaction has finished, database confirms to client whan all changes of the transaction have been made persistent
    - ∗ Transaction may also fail. Database rollback all changes done by the transaction
      - ○ Written as `BEGIN; ..... ABORT;`
      - ○ Can be initiated by the user or DBMS
  - ◇ **Autocommit:** If true, turns each SQL statement into own transition
    - ∗ SQL option
    - ∗ Activated by default, can be deactivated

16.1 ACID:

- Desired properties of transaction
- **Atomicity**
  - ◇ Transaction is executed in its entirety or not at all
- **Consistency**
  - ◇ A committed transaction goes from one consistent state to another consistent state
    - ∗ Before and after a transaction all integrity constraints must hold
  - ◇ Within a transaction constraints may be violated
  - ◇ Transaction leads from consistent state to consistent state

  - ◇ Granularity depends on the integrity constrains
    - ∗ I.e. some constrains are checked for each tuple, some for each statement and some for a transaction
    - ∗ Can be controlled and influenced to some degree
- **Isolation**
  - ◇ **Ideally:** Transaction executes as if it were alone in the system
    - ∗ I.e. enforce serializability
    - ∗ Much to hard to enforce
  - ◇ Implies that integrity constraints always hold if each transaction is correct
  - ◇ DMDB picks one execution order at random (if there are multiple)
    - ∗ If not desired, application must enforce this
- **Durability**
  - ◇ If system crashes after a transaction, the changes of the transaction must still remain in the DB
    - ∗ Or somehow recoverable

16.2 Isolation

- One of the key properties
- **Anomaly:** Misbehaviour of the DB
  - ◇ **Dirty Read:** Read a value which was updated by another transaction which has not yet committed
    - ∗ May contain values which were/are never in the DB
      - ○ When the other transaction aborts
  - ◇ **Non-repeatable Reads:** Reading the same tuple twice gives give us different values both times
    - ∗ It was updated by another transaction which committed (difference to dirty read)
  - ◇ **Phantoms:** During a transaction, another transaction added or removed tuples
    - ∗ Similar to non-repeatable reads
- **Isolation Level:** Defines for each transaction what anomalies we allow to happen
  - ◇

	Dirty Reads	Non-Repeatable Reads	Phantoms
Read Uncommitted	✓	✓	✓
Read Committed	×	✓	✓
Repeatable Read	×	×	✓
Serializable	×	×	×

↓ overhead ↑ concurrency

16.3 More on Serializable

- **Serializable:** Schedule that leads to the same answer as some serial schedule
  - ◇ Only depends on final result and not I/O pattern along
  - ◇ Not all sequential orders necessarily lead to the same result
  - Hard or impossible to enforce
- **Conflicts**
  - ◇ **Definition**
    - ∗ **Same Transaction:**
      - ○ Two operations are always conflicting
      - ○ $\implies$ Reordering within transaction is not allowed
    - ∗ **Different Transaction O_1 in T_1 and O_2 in T_2:**
      - ○ O_1 and O_2 are conflicting if one of them is a write to the same location
  - ◇ **Types**

        * **Read-Write:**
            - Leads to unrepeatable reads
        * **Write-Read:**
            - Leads to dirty read
        * **Write-Write:**
            - Leads to overwriting of uncommitted data
- **Conflict Equivalent** are two schedules iff:
    - One can be transformed into the other by swapping non-conflicting operations
- **Conflict Serializable:** Schedule if it is conflict equivalent to some serial schedule
    - I.e. schedule which can be translated into a serial schedule with a sequence of non-conflicting swaps of adjacent actions
    - Stronger than serializable
        * Conflict serializable $\subseteq$ serializable
    - Only depends on the read/write pattern
        * And not what we are writing
    - Easier than serializability for DB to handle as it does not require the DB to understand what each operator is doing
    - Enforced by most DBMSs
    - **Decide**
        * Each transaction is a node
        * $\exists$ edge T_i to T_j if:
            - Operator o_i in T_i is in conflict with operator o_j in T_j
            - o_j appears earlier than o_i in same transaction
        * Schedule is conflict serializable iff its dependency graph is acyclic
- Serializability and conflict serializability only concern committed transactions (and not aborted)
    - Operations can be in conflict with `ABORT`
        * E.g. `READ(X)`/ `WRITE(X)` before or after `ABORT` of other transaction may lead to different results (only if the other transaction did `WRITE(X)`)
    - But they are somehow still considered as serializable

16.4 Enforce Isolation

- **Goal:** Only allow schedules that are conflict serializable
- Two main approaches
    - **Pessimistic:** Assume that conflicts happen all the time
        * Use locks
    - **Optimistic:** Assume that most transaction do not conflicts
        * Use snapshot isolation
- Need to evaluate which approach is better for our application

16.4.1 Locking

- **Assume:** Do not know what transactions are going to do in the future
- **Idea:** Before the system access the data object X it locks X
    - Prevents access of X by other transaction
- Lock is released only when it is save to
    - I.e. the execution is guaranteed to be conflict serializable
- Allows to enforce serializable schedule
- Types

    - ◇ **Shared Lock (S Lock):**
        - ∗ For reading
    - ◇ **Exclusive Lock (X Lock):**
        - ∗ For writing
- Does not necessarily enforce conflict serializability
- **Two-Phase Locking (2PL)**
    - ◇ Consists of two phases
        - ∗ **Phase 1: Growing**
            - ○ Acquire required locks
            - ○ Cannot release any locks
        - ∗ **Phase 2: Shrinking**
            - ○ Release locks
            - ○ Cannot acquire new locks
    - \+ Guarantees conflict serializability
    - \- **Cascading Abort:** Abort of one transaction leads to abort of another transaction
        - ∗ Happens when we read from a transaction which gets aborted
        - ∗ Really bad if we have already committed
            - ○ Commit must be undone
            - ○ Conflicts with durability property
- **Strict Two-Phase Locking (Strict 2PL)**
    - ◇ **Phase 1:** is similar to 2PL
    - ◇ **Phase 2:**
        - ∗ All looks are kept until end of transaction
            - ○ I.e. `COMMIT` or `ABBORT` released all locks
    - \- Deadlocks possible
        - ∗ **Detection**
            - ○ Each transaction is a node
            - ○ ∃ edge T_i to T_j if T_i is waiting for a lock currently hold by T_j
        - ∗ Deadlock if we have a cyclic wait-for graph
        - ∗ Non-trivial to decide which transaction to kill
        - ∗ Prevented by locking in some global order
- **Granularity**
    - ◇ Problems
        - ∗ We need to lock every single tuple at its own
        - ∗ We need to hold locks for whole transaction to prevent phantoms
    - ◇ DB is hierarchical structure and hence needs to support hierarchical locking
    - ◇ New locks
        - ∗ **Intention Share (IS):**
            - ○ Some lower nodes are in S
        - ∗ **Intention Exclusive (IX):**
            - ○ Some lower node are in X
        - ∗ **Share and Intention Exclusive (SIX):**
            - ○ Root is locked in S
            - ○ Some lower node are in X
    - ◇ Old locks
        - ∗ **S:** All lower nodes are in shard
        - ∗ **X:** All lower nodes are in exclusive
    - ◇ Full overview

	Current Lock					
Mode	NL	IS	IX	S	SIX	X
Request NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	×
IX	✓	✓	✓	×	×	×
S	✓	✓	×	✓	×	×
SIX	✓	✓	×	×	×	×
X	✓	×	×	×	×	×

- ◇ Steps to lock a tuple
  1) Acquire IS on database
  2) Acquire IS on table
  3) Acquire S on tuple

16.4.2 Snapshot Isolation

- **Idea:** Assume that transaction are serializable and revert if they are not
- Working
  - ◇ Transaction receives timestamp $TS(T)$ when it starts
  - ◇ Reads are carried out as of the DB version of $TS(T)$
  - ◇ Writes are carried out on a separate buffer
  - ◇ When transaction commits, abort T_1 if $\exists T_2$ such that:
    - ∗ T_1 and T_2 update the same object and
    - ∗ T_2 committed after $TS(T_1)$ but before T_1 commits
  - ◇ Instead of aborting T_1, we can also let T_1 finish and only then merge T_2
- **Timestamps**
  - ◇ System clock or monotonically increasing for each transaction

\+ High concurrency and availability
  - ◇ Only block when transaction commits

\+ No cascading abort

\+ No deadlock

\- Unnecessary rollbacks

\- **Write Skew:** Interaction of multiple objects
  - ◇ Checking integrity constrains happens in the snapshots
  - ◇ Two concurrent transaction update different objects
  - ◇ Integrity constrains for each is ok but for the combination not

- Looser version
  - ◇ Idea
    - ∗ Object themselves have read (last read) and write (last written) timestamps
    - ∗ If transaction accesses object from the future (object has higher timestamp than transaction timestamp), transaction is aborted
    - ∗ Object timestamps are updated on read/writes
  - \+ No unnecessary rollbacks (I guess)
  - \- Long transaction may starve
  - \- Cascading aborts

17 Recoverability

17.1 Definition

- Important for durability property
- Ensures that the state in the DB is correct
- Need to recover from
  - ◇ Abort of single transaction
    - ∗ Undo all changes of the aborted transaction
  - ◇ System crash (loss of main memory but not disk)
    - ∗ Redo all committed transaction
- **$\mathbf{T_1}$ reads from $\mathbf{T_2}$:** if T_1 reads a value written by T_2 at a time when T_2 was not aborted
- Different families of schedules
  - ◇ Each family has different recoverability properties
- **Recoverable (RC):**
  - ◇ If T_i reads from T_j and commits, then $c_j < c_i$
  - ◇ If T_i reads from T_j and aborts, or if T_i writes etc. it is also RC
  - ◇ No need to undo a committed transaction
  - ◇ **If not RC:** Loss of data
- **Avoids Cascading Aborts (ACA):**
  - ◇ If T_i reads X from T_j, then $c_j < r_i[X]$
  - ◇ If T_i writes X from from T_j, it is also ACA
  - ◇ Aborting a transaction does not cause aborting others
  - ◇ **If not ACA:** Thrashing behaviour when transactions abort each other
- **Strict (ST):**
  - ◇ If T_i reads from or writes a value written by T_j, then
    - ∗ **If $\mathbf{T_j}$ commits:** $(c_j < r_i[X] \wedge c_j < w_i[X])$
    - ∗ **If $\mathbf{T_j}$ aborts:** $(a_j < r_i[X] \wedge a_j < w_i[X])$
  - ◇ Extends ACA to write
  - ◇ Undoing a transaction does not undo the changes of other transactions
  - ◇ **If not ST:** Recovery is very complex or impossible
  - ◇ Enforced by Strict 2PL
- All Schedules $\subset$ RC $\subset$ ACA $\subset$ ST $\subset$ Serial
- **Goal:** All allowed schedules lie in the intersection of ST and conflict serializable

17.2 Write-Ahead Log

- **Assume:**
  - ◇ Disk is save
  - ◇ `Write(A, v)` only changes object in memory but not disk
  - ◇ `OUTPUT(A)` writes changes from memory to disk
- **Idea:** Log changes and restore from log if required
- **Log:** File which only can be appended to
  - ◇ Stored in memory and periodically flashed to disk
    - ∗ When to flash?
  - ◇ Operation
    - ∗ Append Record
      - ○ `START T`
      - ○ `COMMIT T`
      - ○ `ABORT T`

        - `Update <T, X, v>`
      - Flush to disk
        - `FLUSH`
      - Log message do not have to mean that the action was actually done on the DB
- Two main strategies
- **Undo Logging**
  - If T modified DB element X log `<T, X, old value>` to disk before change X is written to disk
    - I.e. call `FLUSH` before calling `OUTUT`
  - If a transaction commits, `COMMIT` record must be logged to disk only after all other changes are written to disk
    - I.e. log `COMMIT` and call `FLUSH` only after calling `OUTPUT`
  - Recovery
    - **Committed Transaction:** Ones which have `COMMIT` in the log
      - `COMMIT` in log guarantees that changes are flushed to disk
        - Nothing to do
    - **Uncommitted Transaction:** Ones which do not have `COMMIT` in the log
      - We cannot be sure if changes were committed or not
        - Undo everything
      - Steps
        - Find all transaction i with `Start` T_i but not `COMMIT` T_i
        - If there is only a single transaction:
          - Scan from the end and undo updates
        - If there are multiple uncommitted transactions
          - Scan from the end (skipping logs from committed transaction) and undo updates
        - Write `ABORT` T_i at the end of log
        - Flush log
  - \- Lots of I/O
  - \- Log is almost the size of the transaction
- **Redo Logging**
  - If T modifies DB element X log `<T, X, new value>` to disk
  - Log `COMMIT` and call `FLUSH`, before calling `OUTPUT`
  - Recovery
    - Scan log from the beginning
    - If `COMMIT` T_i is not in the log
      - No changes of T_i appears on disk
      - Write `ABORT` T_i
      - Ignore changes if T_i during scanning
    - If `COMMIT` T_i is in the log
      - Does not mean all its changes are already on disk
      - Redo all changes of T_i
    - Flush log
  - \+ Less I/O than undo logging (I guess)
  - \- The log we need to keep can be very, very long
    - After commit we still do not know if the changes are reflected on the DB
  - \- Problematic when two transaction update different objects which are stored on the same page
- **Undo/Redo Logging**

- Combine both to get pro from both
- Before modifying any DB element X on disk, write `<T, X, old value, new value>`
- Flush log before actual changes are made on disk
- Recovery
  - If `COMMIT T` is not in the log
    - T is incomplete
    - Undo changes
  - If `COMMIT T` is in the log
    - T is complete
    - Redo changes

18 Distribution

- **Distributed Commit**
  - **Problem:** Multiple DBs and a single coordinator which manages them
    - Atomcity of a single nodes does not imply atomcity in a distributed setting
  - Two main methods
  - **Two Phase Commit**
    - Consists of two phases
    - **Voting Phase:** Coordinator inquires if all nodes are ready and willing to commit
      - Initiated by coordinator sending `Prepare`
      - If node says *OK* it cannot change its mind anymore
    - **Decision Phase:** Coordinator asks all nodes to commit
      - Initiated by coordinator sending `Commit`
      - Coordinator sends `Abort` if not all workers are ready
      - Only if all said *OK* in voting phase
      - If any worker or coordinator dies we have to rolleback
    - If a worker/coordinator is temporarily dead we may continue but let the other know that we were dead
  - **Linear Two Phase Commit**
    - Coordinator only communicates with one workers, which communicates with another worker etc...
  - **Two Phase Commit vs Linear Two Phase Commit**
    - Given 1 Coordinator and N Workers
    -

	Two Phase Commit	Linear Two Phase Commit
Total Messages	$3N$	$2N$
Latency t	$3t$	$2Nt$

- **Distributed Query Processing**
  - Execute query on multiple machines
    - Desirable for
      - Data too large to fit into one machine
      - Computationally intensive query
  - Different ways of construction
    - Shared Memory
    - Shard Disk
    - Nothing Shared
      - We consider this one
      - Master received query and distributes to several workers
  - **Goal:** Hide complexity from users
  - **Idea:** Partition DB to each worker and each only deals with own partition
  - Examples
    - **Table Partitioning**
      -
        ```
        SELECT * FROM R(a,b,c), S(a,d)
        WHERE R.b = 1
        AND S.d = 2
        AND R.a = S.a
        ```
      - Worker 1
        - Run
          ```
          SELECT * FROM R
          WHERE R.b = 1
          ```

        - ▷ Generate table $R'(a, b, c)$
    - ◦ Worker 2
        - ▷ Run

```
SELECT * FROM S
WHERE S.d = 2
```

        - ▷ Generate table $S'(a, d)$
    - ◦ Combine R' and S'
        - ▷

```
SELECT * FROM R', S'
WHERE R'.a = S'.a
```

    + Worker 1 and 2 can work concurrency
    + If R' and S' are small the communication is small
    - Problematic if one table is too large for a worker
    - Parallelizability of querries (if available) is not exploited
- ∗ **Horizontal Partitioning**
    - ◦

```
SELECT * FROM R(a,b,c), S(a,d)
AND R.a = S.a
```

    - ◦ Worker 1
        - ▷ Run

```
SELECT * FROM R1, S1
WHERE R1.a = S1.a
```

        - ▷ Generate table $T1(a, b, c, d)$
    - ◦ Worker 2
        - ▷ Run

```
SELECT * FROM R2, S2
WHERE S2.a = R2.a
```

        - ▷ Generate table $T2(a, b, c, d)$
    - ◦ Combine $T1$ and $T2$
        - ▷

```
SELECT * FROM T1
UNION
SELECT * FROM T2
```

    + When the result is small this can be fast TODO: Not sure why this even works
- ∗ **Distributed QO 1**
    - ◦

```
SELECT * FROM R(a, b, c), S(a, d)
WHERE R.a = S.a
```

    - ◦ Replicate one table on both nodes and split the other in two
    - ◦ Worker 1: $T_1 = R_1 \bowtie S$
    - ◦ Worker 2: $T_2 = R_2 \bowtie S$
    - ◦ Combine: $T_1 \cup T_2$
- ∗ **Distributed QO 2**
    - ◦

```
SELECT * FROM R(a, b, c), S(a, d)
```

```
WHERE R.a = S.a
```

            - Tables are portioned on the join attribute and each node performs the join locally
            - Worker 1: $T_1 = R_1 \bowtie S_1$
            - Worker 2: $T_2 = R_2 \bowtie S_2$
            - Combine: $T_1 \cup T_2$
        * **Distributed QO 3**
            - ```
              SELECT * FROM R(a, b, c), S(a, d)
              WHERE R.a = S.a
              ```
            - Tables are portioned on different keys
            - Worker 1: $T_1 = R_1 \bowtie (S_1 \cup S_2)$
            - Worker 2: $T_2 = R_2 \bowtie (S_2 \cup S_2)$
            - Combine: $T_1 \cup T_2$
    - **Replication**
        * Replicate data among several machines
        * **Group Mirroring**
            - Replicate at machine level
            - Each machine has data blocks which are stored on two other machines
            - If the two wrong machines die we loose data
            - Death of a single machine leads to $2x$ slowdown
        * **Spread Mirroring**
            - Each machine contains data available on all other machines
            - If two die we certainly loose (little) data
            - Single failure leads to $1/N$ more load on other machines
- **Distributed Key-Value Store**
    - Relational DB is expensive and does not scale well
    - Data Model
        * Key + Value
        * Indexed on key
    - Distributed Deployment
        * Horizontal partitioning
        * Replication of partitions
    - Build
        * Build as a simple hash table
        * If distributed, we use consistent hashing
    + Very fast lookups
    + Easy to scale
        * Add more copies as we add more machines
    - Only support point queries
    - Some operations are very expensive
    - Hard to keep data consistent among different copies
    - Complexity is pushed to user/application
              ```