CSE 344 Final Review

- Databases

  - Database: Collection of files storing **related data**.
  - Database Management System: an application program that allows us to nmanage efficiently the collection of data files.
  - Responsibility of DBMS:
    - Data storage and manipulation
    - Black box throught
    - Physical data independence

- Relational Databases

  - Motivation: avoid singular/flat files

  - Data Model: **Mathematical/Conceptual** way for decribling the data

    - Schemas and keys
    - Record and attributes
    - Attribute types/typing

    Three elements of data models:

    - Instance (actual data)
    - Schema (what data being stored)
    - Query language (how to retrieve qnd manipulate data)

  - Keys: one (or more) attributes that **uniquely identify** a record

    - When multiple keys, we can choose one key as **primary key**.

  - Foreign keys: attributes whose value is a key of a record in some other relation.

  - SQL Structure:

    - Flat tables:

      - First normal form
      - Crosswalks and joins
      - Breaking up data into multiple relations

Tables are **not ordered**, **flat**, and has **physical data independence** (do not prescribe how they are implemented/stored on disk).

- Code:
  - `create` statements: declaring **types** and **keys**.
  - `insert` / `delete` statements
  - `update`
  - `drop` table
- **`DISTINCT`** : return unique elements
  Projection: function which selects a subset of the attributes
- Inner vs. Outer Joining
  - **inner joining**: each row must come from both tables
  - **outer joining**: include rows from only one of the two tables
  - `tableA (left/right/full) outer join tableB` :
    - Left outer join: includes tuples from `tableA` even if no match
    - Right outer join: includes tuples from `tableB` even if no match
    - Full outer join: includes tuples from both `tableA` and `tableB` even if no match
- Nested loop semantics
  - Cross-join multiple tables with selection
- Self joins
  - includes one table multiple times
- Aggregation:
  - Everything in `SELECT` must be either a **GROUP-BY** attribute, ot an aggregate.
- `WHERE` vs. `HAVING` :
  - `WHERE` can contain any condition on attributes in `FROM` clause.
    - Applied to individual rows.
    - No aggregate.
  - `HAVING` can contain any condition on aggregate expressions and any group-by keys.
    - Entire group is returned, or removed.

- Executing in order of: **FWGHOS**
- Subqueries:
  - In `SELECT` : single attribute projection
  - In `FROM` : use `AS` and `WITH AS`
  - In `WHERE` : use **existential quantifiers** (`EXISTS` , `IN` , `ANY` , `ALL`)
  - **Correlated** subquery: depends on outside query.
  - Finding Witness (the product with max price): compute aggregate in subquery.
- Monotonicity:
  - A query is **monotone** if whenever we add tuples to one or more input tables, the answer to the query will not lose any of the tuples.
  - If Q is a `SELECT-FROM-WHERE` query that **does not** involve any subqueries, Q **is** monotone.
  - Queries with **universal quantifiers**(all) or **negation** must be nested.

◦ Relational Algebra:

- Set/Bag semantics: differs on allowing/disallowing duplicates in tuples, also referred as "Standard"/"Extended" Relational Algebra
- Operators:
  - Union $\cup$, intersection $\cap$, difference $-$
  - selection σ
  - projection π
  - Cartesian product $\times$, join $\bowtie$
  - Rename ρ
  - Duplicate elimination δ
  - Grouping and aggregation γ
  - sorting τ

  All operators take 1 or more relations as input and return another relation.

  Join in R.A.:

  - **Theta-join**: $R \bowtie_\theta S = \sigma_\theta(R \times S)$
  - **Equijoin**: Theta-joins which join condition θ only involves equilities.
  - **Natural Join**: $R \bowtie S = \pi_A(\sigma_\theta(R \times S))$

        - Relational Algebra is **equally expressive** with SQL.
    - Datalog:
        - For Queries that cannot be defined in RA (recursive queries)
        - Terminology:
            - Facts: tuples in database
            - Rules: queries
            - Extensional Predicates: predicates defined in database itself
            - Intensional Predicates: self-defined predicates
            - Head: the intensional predicates to be defined
            - Body: rule for each head (made with atoms/relational predicates)
            - head variables / existential variables
        - Fixpoint semantics: start with empty relations, repeat until $IDB_t = IDB_{t+1}$.
            - A datalog program without functions (+, *, ...) always terminates in polynomial time.
        - Minimal model semantics: returns the smallest IDB such that all vars in EDB satisfies rule defined for this IDB (recursively too).
        - Linear:
            - Right-linear: `T(x, y) :- R(x, z), T(z, y)`
            - Left-linear: `T(x, y) :- T(x, z), R(z, y)`
            - Non-linear: `T(x, y) :- T(x, z), T(z, y)`
        - Writing Rules:
            - Safe: if every variable appears in some positive relational atom
- Semi-structured Data
    - **Transactional Data**: Frequent read, simple update
      Analytical Data: Decision support, multiple joins & group bys

      OLTP (online transaction processing): simple look-up, many updates, consistency.
      OLAP (online analytical processing): Decision Support
    - **Partitioning**: Store data partitioned on multiple servers, easy to write, hard to read (Good for transactional)
      Dupplication: Duplicate data on multiple files, hard to write, easy to read (Good for analytical)
    - **JSON**: semi-stuctured data model
        - `{}` holds objects, `[a,b,...,x]` is an array.

        - Duplicate keys are allowed by standard, but not by mant implementations.
        - Allows <u>null values</u>, <u>duplicate attributes</u>, and <u>nested collections.</u>

    - **SQL++**:

        - Dataverse: is a database
            - `USE` + dataverse name
        - Type: defines a schema of a collection
            - Lists all required fields, `?` for optional fields.
            - `OPEN` permits other field, while `CLOSED` doesn't.
            - `PRIMART KET mykey AUTOGENERATED` to autogenerate a unique id for each entry.

    - indices:

        - Three type of indices available:
            - `BTREE` : for quality and range queries.
            - `RTREE` : for two-dimensional range queries:
              for example: `WHERE x > 20 AND x < 30 AND y > 20 AND y < 30`
            - `KEYWORD` : for substring search.
        - **CANNOT** index inside a nested collection.
        - Syntax: `CREATE INDEX countryID ON country(carCode) TYPE BTREE`

    - **Heterogeneous collections**: Same attribute, but in some object is array, in some object is not.

        - Solution: Use `(CASE ... WHEN...THEN...ELSE...) u` , and `is_array()` helper function.

    - `UNNEST` : distribute each element in this array to the rest of the row.

    - Multi-value join: split an array to "unnest" with `split(y.'country', " ")` .

- Physical Plan:

    - Overall execution of a SQL query:
      Parse SQL -> Select logical plan (Relational ALgebra) -> Select physical Plan -> execute.
      Physical Plans: Choose an effective implementation for a RA operator.

    - Basic Physical Operator: Join

        - Nested Loop Join: $O(n^2)$
        - Merge Join: $O(nlog(n))$
        - Hash Join: $O(n)$ to $O(n^2)$

- Physical Data Independence: Apps are insulated from how data is stored physically.
- Data Storage:
  - DBMS store data in files, each file is splitted into blocks, and each block contains serveral tuples.
  - Data file can be either a Heap file (unsorted), or Sequencial file (sorted according to some attribute/key).
  - Index: Additional file mapps from a search key (some attribute's value) to the actual tuple in memory. (*Could have many index for one table.*)
    - Clustered index: records close in index is close in data
    - Unclustered index: records close in index may be far in data
    - Primary: Over attributes including primary key
    - Secondart: Otherwise.
- Scanning a disk: Sequencial scan is much faster than random accessing the disk.
- Index should be selected on attributes which:
  - an exact match
  - a range match (clustered index works the best)
  - a join

  On such key is needed. (solution to **index selection problem**).
- Cost Estimation:
  - Parameters for cost estimation: for some relation R,
    - B(R): number of blocks in R
    - T(R): number of tuples in R
    - V(R, a): number of distinct values of attribute a
      - Note that when a is a key, V(R, a) = T(R); else, V(R, a) $\leq$ T(R)
  - Cost of:
    - Sequentially scan a table: B(R)
    - Index-based scanning:
      - Clustered: f * B(R)
      - Unclustered: f * T(R)
        [Note: f is usually 1/V(R, a), which a is the key that index is built on]
  - Join Cost:

        - **Hash Join**: Scan one table into memory and build hash table, scan in the other table and join.
            - Overall cost: B(R) + B(S)
            - Should build hashTable on the relation with more distinct attributes.
        - **Nested Loop Join**: Scan one table into memory, scan in the other and compare each tuple pair to determine if successfully joins.
            - Overall cost: B(R) + T(R) * B(S)
        - **Sort-merge Join**: Scan two tables and sort in memory, then merge-join them. (Not one-pass algorithm)
            - Overall cost: B(R) + B(S)
        - **Index Nested loop Join**: Assume S has an index page in memory. So iterate over R and fetch corresponding entries of S.
            - Cost if S is clustered: B(R) + T(R) * (B(S) / V(S, a))
            - Cost if S is unclustered: B(R) + T(R) * (T(S) / V(S, a))

- Parallel/Map-Reduce:

    - Speed up: More nodes(processors, computers), same data, more speed
      Scale up: More nodes, more data, same speed

    - Shared-nothing: each machine has its own disk and memory. Easy to maintain and scale, but hard to administer and tune.

    - Intra-operator parallelism:

        - Each operator is placed on multiple nodes.
        - Good for both transactional and analytical work.

    - Horizontal Data Partinioning:

        - Block Partition: partition arbitarily with block alignment.
        - Hash Partition on some attribute
        - Range Partition: a tuple goes to some server if value falls in some range.

    - Uniform Data/Skewed Data:

        - Uniform Data: Data is partitioned uniformly across servers.
            - Block Partition will be unform.
        - Skewed Data: Some server holds more data than other servers.
            - Hash Partition/Range Partition on some attribute may cause this problem.

    - Shuffle & Broadcast:

        - Shuffle Join: Reshuffle each table on the common attribute, and compute join locally.
        - Broadcast join: Reshuffle one table on the common attribute, join with another table, and broadcast the result.
    - Map/Reduce framework:
        - Map: extract something you care about for each record.
        - Reduce: aggregate, summerize, filter, transform
        - Fault-tolerance: map writes to local disk, reduce reads from disk. If fails, the reduce task is restarted on another server, but slow.
        - Spark uses "Resilient distributed datasets" = main memory + linerage.
    - Straggler: a machine takes unusual long to complete one of the last tasks.
        - Solution: pre-emptive back up execution of last few in-progress tasks.
- Database design:
    - Conceptual model for database design:
        - Entity: an object
            - Attribute: attributes of entity. Each entity set must have a key, which can be one attribute or serveral attributes.
        - Relationship: how entities are related.
        - Weak entities: entities which key come from other classes which they are related
    - Relationship between entities:
        - **One to one**: No seperate table
        - **One to many**: No seperate table
        - **Many to many**: Seperate table
    - Integrity constraint motivation: a consition specified on a database schema which restricts the data that can be stored on any instance of the database.
        - Key Constraints: the key in each schema must uniquely identifies a person
        - single-value constraints: a person can only have one biological father
        - Referetial integrity constraints: if you work for a company, it must exist in the database.
        - Other constraints: value needs to be in legal range.
    - Constrains in `SQL` :
        - Primary Key, Foreign Key

            - `UNIQUE(name)` can be used to ensure `name` is unique (even if it is not primary key). There can be many unique.
            - Foreign key constraints: the referential integrity cnostraints. The referred column must be a key in another table.
        - Attribute level constraints
            - `NOT NULL`
            - `CHECK` can check any condition
        - Tuple-level constraints
        - Global constraints/Assertions

    - Functional Dependencies: if one attribute implies another
      Closure: everything that an attribute determine

        - If $A \to B$ and $B \to C$, then $A \to B, C$.
        - Formal notation: $\{A\}^+ = \{A, B, C\}$
        - If one attribute determines all other attributes in the schema, then it is called **superkey**
        - The smallest subset of attributes that does it is called the **minimal key** or just **key**.

      Boyce-Codd Normal Form (BCNF): No bad functional dependencies.

        - For all attribute X, either $\{X\}^+ = \{X\}$ or $\{X\}^+ =$ all attributes.
        - Whenever $X \to B$ is a non-trival dependency, then X is a superkey.

- Transactions:

    - Changes to database should be all or nothing.

    - Transaction: Collection of statements that are executed atomically (logically speaking)

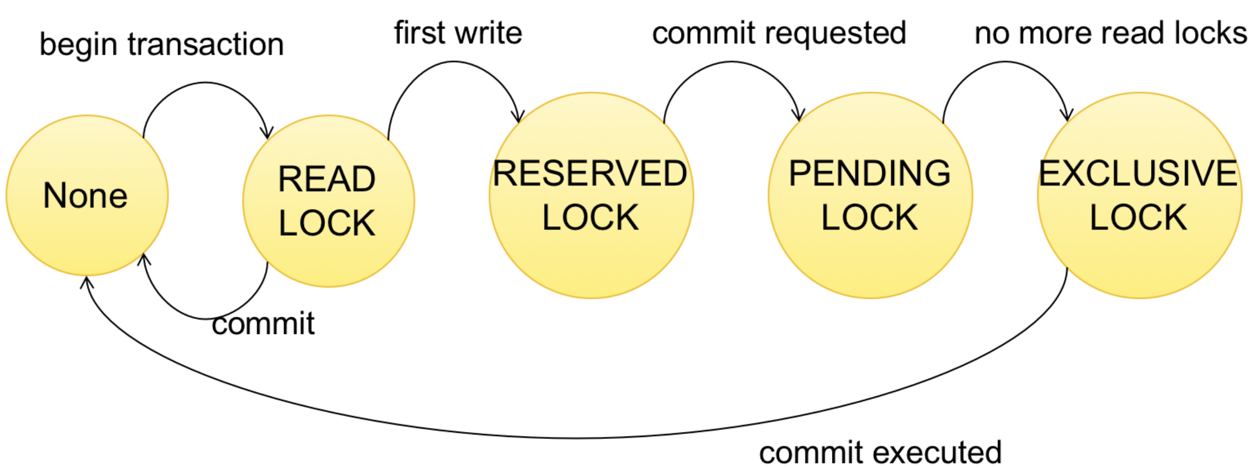
    - Transaction Properties:

        - **A**tomic: State shows either all effect for a transaction, or none of them.
        - **C**onsistent: Integrity must hold before and after transaction is executed.
        - **I**solated: effect of each transaction is as if it were the only transaction running on the system
        - **D**urable: transaction's effect remains after execution finishes.

    - **Serial Schedule**: transactions are executed sequentially

    - **Serializable Schedule**: equivalent to a serial schedule

    - Determine if schedule is serializable by looking at conflicts (swapping will change program behavior)
        - READ-WRITE
        - WRITE-READ
        - WRITE-WRITE
- **Conflict Serializable Schedule**: transform into a serial schedule by swapping adjacent non-conflicting actions. (Every conflict serializable schedule is a serializable schedule).
    - Testing Conflict Serializable Schedule with **Precedence Graph**: each vertex is a transaction; an edge from A to B if an action in A conflicts with action in B and comes before B.
    - The schedule is conflict serializable if and only if the graph is **acyclic**.
- Recoverable: Each transaction commits after all transactions from which it has read has committed.
- **Scheduler** (Concurrency Control Manager): the module which schedules transactions, ensuring serializability.
    - Locking Scheduler: each element acquires the lock, wait until other transactions release the lock, then use. Picture below is the locking scheduler for SQLite.



- If no concurrency control:
    - Dirty Read (including inconsistent read)
    - Unrepeatable reads
    - Lost updates
- Two-phase locking: In every transaction, all lock requests must precede all unlock requests. (Ensures Conflict serializability)

- Strict 2PL:
    - All locks are held until commit/abort:
    - All unlocks are done together with commit/abort.

  With strict 2PL, we can get schedules both conflict-serializable and recoverable.

- DEADLOCK: Locks waiting for other locks to release and stuck in infinite loop.
  Solution: Use graph to check for cycles and if detected, abort one transaction.

- Isolation Levels in SQL:
    - `READ UNCOMMITTED`
        - Strict 2PL for WRITE locks, no READ locks.
    - `READ COMMITTED`
        - Strict 2PL for WRITE locks, only acquire locks for READ locks (not 2PL)
    - `REPEATABLE READ`
        - Strict 2PL for both READ and WRITE
    - `SERIALIZABLE`
        - Strict 2PL for both READ and WRITE, and **Predicate locking** to deal with phantoms.
            - Phantom: a tuple that is invisible during part of a transaction execution but not invisible during the entire execution
            - Predicate lock: a lock on an arbitary predicate.