# ECE455 Final Review

Jonathan Lam

12/05/21

## Contents

Materials located at: `http://files.lambdalambda.ninja/materials/ece464/`.

# 1 Database design

Stages:

1. **Requirements analysis**: talk to users! Understand what data is to be stored, and what operations and requirements are desired.
2. **Conceptual design**: high-level description of the data and constraints.
    - Entities, relationships, constraints (functional view, ER model).
3. **Logical database design**: convert conceptual model to a schema in the chosen data model of the DBMS (ER to relational).
4. **Schema refinement**: schema normalization, look for ways to improve schema.
5. **Physical database design**: direct the DBMS into choice of underlying data layout (e.g., indices and clustering) to optimize performance.
6. **Applications and security design**: how the DB will interact with applications and users.

# 2 The ER (Entity-Relationship) model

ER is a way to visualize the high-level functional view of a relational database. The design is subjective and implicitly codes foreign-key constraints. Good design can also be influenced by schema normalization (in a later section).

## 2.1 Vocab/concepts

- **Entity**: an object that we store data and **attributes** (fields) about
  - Each attribute has a **domain** (type/range of possible values)
- **Entity set**: a collection of entities of the same type
  - Each entity set has a **key** (to find entities)
- **Relationship**: association between 2+ entities
- **Relationship set**: collection of similar entities
- **Key constraint**: whether a key must be unique in a relationship set
- **Participation constraint**: whether all entities must be in the relationship set (**total participation**)
- Also consider **one-to-many**, **many-to-one**, **many-to-many** relationships (**one-to-one** are not necessary, a bijection can be represented as a single record for most cases)
- **Weak entity**: when an entity is identified uniquely by an **owner entity** in another entity set
  - Relationship is called an **identifying relationship**
  - The weak entity must have total participation in this one-to-many identifying relationship (i.e., it has a key and participation constraint)
- **Is-a hierarchy**: as the name suggests, hierarchy of types as opposed to disjoint types
  - **Overlap constraints**: are the entity types disjoint?
  - **Covering constraints**: does one entity type subsume another?
- **Aggregation**: when a relationship set is treated as an entity set, and thus can have relations with other entity sets or relationship sets

  - Compare this to a **ternary relationship**: the distinction is the logical use case. If the relationship exists between three parties, then use a ternary relationship. If the relationship exists between an entity and the relationship between two others, use an aggregation. The latter is also useful if the same relationship is referred to multiple times in the aggregation relationship set.

### 2.2 Design choices

(Some of the answers to these are "it depends." Think logically!)

- Entity or attribute?
  - Use entities when we want referential semantics, i.e., **reused shared objects**.
  - Use entities when the value is non-atomic (e.g., want to destructure date value by month, day, year).
- Ternary relationship or attribute of relationship? Aggregation?
  - Use binary relationship + attribute versus ternary relationship using the same reasoning as attribute vs. entity.
  - See notes on aggregation above for ternary vs. aggregation.

### 2.3 Diagrams

- Entities are represented by boxes.
- Attributes are represented by circles.
- Relationships are represented by rhombi.
- Key constraints are represented by an arrow from the entity to the relationship.
- Participation constraints are represented by a bold line from the entity to the relationship.

# 3 The relational model

A very common tabular model for database design that is much closer to SQL (**relational query languages**) syntax than ER diagrams.

## 3.1 Vocab/concepts:

- **Relation**: made of two parts:
  - **Instance**: a table, with rows and columns.
    - **Cardinality**: number of rows/records.
    - **Degree*/*Arity**: number of fields/attributes.
  - **Schema**: prototype/structure for relation, e.g., name of relation, name and type of each column.
- **Relational database**: a set of relations.
- **Integrity constraints** (ICs, a.k.a., **domain constraints**): conditions that must be true for any instance of the database.
  - Types:
    - **Domain constraint**: data type checking.
    - **Entity integrity constraint**: PK can't be null.
    - **Referential integrity constraint**: foreign key cannot refer to invalid entity.
    - **Key constraint**: mentioned in ER section.
  - Prevents illegal data entry that violates integrity constraints.
  - Specified when schema is defined, and checked when relations are modified.
- **(Candidate) key**: a set of fields in a relation s.t.:
  - No two distinct tuples can have the same values in all key fields.
  - This is not true for any subset of the key.
  - (Candidate keys can be specified using the `UNIQUE` keyword.
- **Primary key** (PK): one of the candidate keys used to identify records in the relation.
- **Foreign key** (FK): set of fields in one relation that is used to "refer" to a tuple in another relation.

  - Must correspond to the PK of the referred-to relation.
  - Like a "logical pointer."
  - If foreign key constraints are enforced, then **referential transparency** (see "domain constraints") is enforced – no "dangling pointers." This can be enforced in multiple ways on deletes and updates (insertions are rejected if the FK is invalid):
    * No action (rejected)
    * Cascade (also delete/update records that refer to the current)
    * Set null/set default
  - Interesting application: FK as PK in one-to-one relationships (not used very often): `https://stackoverflow.com/a/10983016`

- **View**: a relation, but we store a definition rather than a set of tuples.
  - We can think of this as not duplicating the records, but generating a "pseudo-table" that is easier to query.
  - Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s) (think: information hiding/encapsulation).

### 3.2 Sample SQL queries

- Get names and logins of 18-yo students:

```
SELECT s.name, s.login,
FROM students s
WHERE s.age=18;
```

- Creating relations:

```
CREATE TABLE students (
    sid CHAR(20),
    name CHAR(20),
    login CHAR(20),
    age INTEGER,
    gpa REAL,
    PRIMARY KEY (sid, cid)
);
CREATE TABLE enrolled (
```

```
    sid CHAR(20),
    cid CHAR(20),
    grade CHAR(2),
    PRIMARY KEY (sid),
    UNIQUE (cid, grade)
);
```

### 3.3 Translating ER to relational model

- Entity sets become tables/relations.
- Relationship sets also become tables/relations, where the keys from all the participating entity set (w/ FK constraints) form a **superkey** PK.
- Key constraint can be indicated with a PK or the `UNIQUE` keyword for non-PK keys.
- Weak entity relationships do not require a separate relationship table.
- The `NOT NULL` constraint can be helpful for enforcing various domain constraints.
- Multiple possible approaches for is-a relationships – up to you to choose something reasonable.

# 4 SQL

Basic SQL query format:

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification;
```

## 4.1 Vocab/concepts

- **Relation lisnt**: a list of relation names.
- **Target list**: a list of attributes of relations or terms with aggregate operations in the relation list.
- **Qualification**: a set of constraints on the attributes in the relation list to filter by.
- **Set operations**: `UNION`, `INTERSECT`, `EXISTS`, `IN`, `ANY`, `ALL` are operations on sets.
  - `UNION` and `INTERSECT` are binary relations on **field-compatible** sets.
  - `EXISTS` is a unary function on a set returning a boolean.
  - `IN` is a binary function on a value and a set returning a boolean.
  - `ANY` and `ALL` are weird operators. Example:

    ```
    SELECT *
    FROM sailors s
    WHERE s.rating > ANY (
        SELECT s2.rating
        FROM sailors s2
        WHERE s2.sname='Horatio'
    );
    ```

  - Note: a lot of the set operations can be written in terms of each other. E.g., `INTERSECT` can be written in terms of `IN`, `EXCEPT` can be written in terms of `NOT IN`.
- **Nested queries** (**subqueries**): `WHERE`, `FROM`, and `HAVING` clauses can have nested queries.

  - **Correlated subqueries**: a subquery that is dependent on variables from the outside scope. Causes reevaluation of the subquery for each matched record of the superquery.

- **Aggregate operators**: allow us to compute a single value from a set of fields.
  - Common aggregate operators:
    * `COUNT(* | [DISTINCT] A)`
    * `SUM([DISTINCT] A)`
    * `AVG([DISTINCT] A)`
    * `MAX(A)`
    * `MIN(A)`
  - Note: these return a $1 \times 1$ relation, which is equivalent to a scalar (i.e., it is polymorphic, similar to scalars in MATLAB).
  - Aggregate operators can be applied to all tuples, or they can be applied to groups of tuples grouped by the `GROUP BY` clause (and filtered by the `HAVING` clause).

- **Null values**: used when field values are unknown or inapplicable.
  - Similar to Java: all objects/values are inherently nullable. (Bad design choice?)
  - **Three-valued logic** required for boolean expressions, which may also contain nulls.
  - This creates the chance for new operators (outer/left/right joins).

- **Checks**: a way to create arbitrary integrity constraints (see previous section) within a table.
  - Example:

```
CREATE TABLE sailors (
    sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK (rating >= 1 AND rating <= 10)
);
```

- **Assertions**: a way to create integrity constraints over multiple tables.
  - Example:

```
CREATE ASSERTION small_club CHECK (
    (SELECT COUNT(s.sid) FROM sailors s)
    + (SELECT COUNT(b.bid) FROM boats b)
    < 100
);
```

- **Triggers**: procedures that starts automatically if specified changes occur to the DBMS.
  - Three parts:
    - **Event**: activates the trigger.
    - **Condition**: tests whether the trigger should run.
    - **Action**: what happens if the trigger runs.
  - Triggers should not be used in place of purpose-built mechanisms for integrity. For example:
    - Triggers should not substitute FK constraints.
    - Triggers should not substitute transactions (triggers can fail, transactions can't).
    - Triggers can overlap with logic from your application, and incur a performance overhead.
  - Good uses for triggers might be for auditing or automatic domain-specific processes (e.g., batch jobs) that cannot be performed by automatic integrity checks.
  - Example:

```
CREATE TRIGGER young_sailor_update
    AFTER INSERT ON sailors
REFERENCING NEW TABLE new_sailors
FOR EACH STATEMENT
    INSERT INTO young_sailors (sid, name, age, rating)
    SELECT sid, name, age, rating
    FROM new_saliors n
    WHERE n.age <= 18;
```

- **Join**: a binary operation on relations that produces another relation, where each record in the resultant relation includes one record from each relation.

- Types:
  - **Inner join**: returns record pairs where the join condition is true.
  - **Left (outer) join**: returns record pairs where the join condition is true. If the join condition is false between a record from the left table and all records in the right table, a record is added in the resultant relation with `NULL` values filled in for the fields from the right relation. In other words, it is the inner join unioned with the unmatched records from the left relation.
  - **Right (outer) join**: opposite of the left outer join.
  - **Cross (outer) join**: the cartesian product of the tables.
  - **Full (outer) join**: the combination of a left and right join. In other words, it is the inner join unioned with all unmatched records from the left and right relations.
- The word "join" by itself typically refers to an inner join.
- Syntax:

```
-- inner join; INNER keyword is optional
SELECT field-list
FROM left-relation
[INNER] JOIN right-relation
ON join-condition;

-- left/right join
SELECT field-list
FROM left-relation
(LEFT|RIGHT) JOIN right-relation
ON join-condition;

-- cross join
SELECT field-list
FROM left-relation, right-relation;

-- alternative inner join using qualification
SELECT field-list
FROM left-relation, right-relation
WHERE join-condition;
```

### 4.2 Conceptual evaluation strategy

Conceptual evaluation strategy goes as follows (w/o optimization). Of course, if there is no qualification/grouping list/grouping qualification, then those steps will be omitted.

1. Compute cross-product of relation list.
2. Discard tuples if they fail qualifications.
3. Delete attributes that are not in target list.
4. If `DISTINCT` is specified, eliminate duplicate rows.
5. Fields are partitioned into groups by the value of the attributes in the grouping list.
6. The group qualiifcation is then applied to eliminate groups.
7. One answer tuple is generated per qualifying group.

# 5 ORMs

An **object-relational mapper** (ORM) is a tool for interfacing with a DBMS using native data structures inherent in a general-purpose programming language.

## 5.1 Advantages

- More natural to represent data structures using the general purpose language than SQL's narrow-minded data structures.
- Data queries can be hidden under an OO interface; e.g., fields of an object (entity) may not be stored on that entity but in a related field, and accessing that field may transparently perform a join query. **Lazy loading** may be utilized for performance.
- Prevents unsafe SQL (e.g., SQL injection) and enforces types.
- Can use without learning SQL syntax.
- Support most RDBMS features, e.g., sessions with transactions.

## 5.2 Disadvantages

- Performance overhead.
- Learning curve.
- Library may not exist in the programming language of choice.
- Less useful for non-OO languages (e.g., FP languages).
- Seems like most people don't use them anyways. (E.g., see: `https://fruty.io/2020/10/27/why-i-stopped-using-orms-to-get-the-job-done/`, `https://stackoverflow.com/q/194147`).

# 6 Transactions

A **transaction** is a sequence of actions that is considered to be one atomic unit of work. Transactions are ended with a `ROLLBACK` or `COMMIT` statement, which either undoes all the transaction changes or performs them as if atomically.

## 6.1 ACID properties

**ACID properties** are a set of principles *applied to transactions* that most RDBMS's aim to follow. Note that no-SQL (i.e., non-relational DBMS's) don't tend to achieve these.

- **Atomicity**: each transaction occurred, or it didn't; it cannot happen partway.
- **Consistency**: if the transaction is consistent (see following definition) and the database was consistent when the transaction began, then the transaction will preserve this consistency property.
  - **Consistent**: referring to a database in which all integrity constraints are enforced, or to a transaction which does not break any integrity constraints.
    - Note: this definition of consistency is different than that in the CAP theorem for distributed DBMSes.
- **Isolation**: the execution of one transaction will not affect the result of another transaction; i.e., the system state will be as if transactions ran serially, even if they were actually run concurrently.
- **Durability**: once a transaction has been committed, its effects persist (even despite power loss, crashes, or errors).

## 6.2 Concurrency control

**Concurrency control** is the process of managing simultaneous executions of transactions in a shared database. This ensures correct results while running transactions as fast as possible using concurrency.

A **schedule** is a way to describe the sequence of actions of one or more concurrent transactions. We assume that all individual actions are atomic and thus can be ordered in time. A **serial schedule** is one in which no transactions are interleaved. A **serializable schedule** is a schedule that is

equivalent (in its effects) to some serial schedule of the same transactions. Similarly, a **non-serializable schedule** is a schedule that is not a serializable schedule, which means data integrity and consistency problems. The conflicts listed below may cause a non-serializable schedule.

- **Lost update** or **blind writes** (**WW conflict**): one transaction overwrites uncommitted data of another. A sample schedule is shown below.

| T1 | T2 |
|---|---|
| W(A) | |
| | W(A) |
| | COMMIT |
| R(A) | |
| COMMIT | |

  `T1` should be atomic, but `T2` updates `A` after `T1` writes it and before `T1` reads it, which causes an isolation problem. `T1`'s write to `A` is lost. This schedule is not serializable because the value of `R(A)` would change. Note that blind writes can happen even without a read operation, unlike dirty and unrepeatable reads.

- **Dirty read** (**WR conflict**): one transaction reads changes caused by a concurrent transaction before they are committed. A sample schedule is shown below.

| T1 | T2 |
|---|---|
| W(A) | |
| | R(A) |
| | COMMIT |
| X | |
| COMMIT | |

  In this case, `T2` attempts to read the newly-written value of `A` before `T1` has committed it. If `T1` is rolled back, then the value would be inconsistent. `X` is a don't-care (arbitrary) action so that `T1` doesn't commit right away. This schedule is serializable (equivalent to the serial schedule `T1;T2`), but the following one is not.

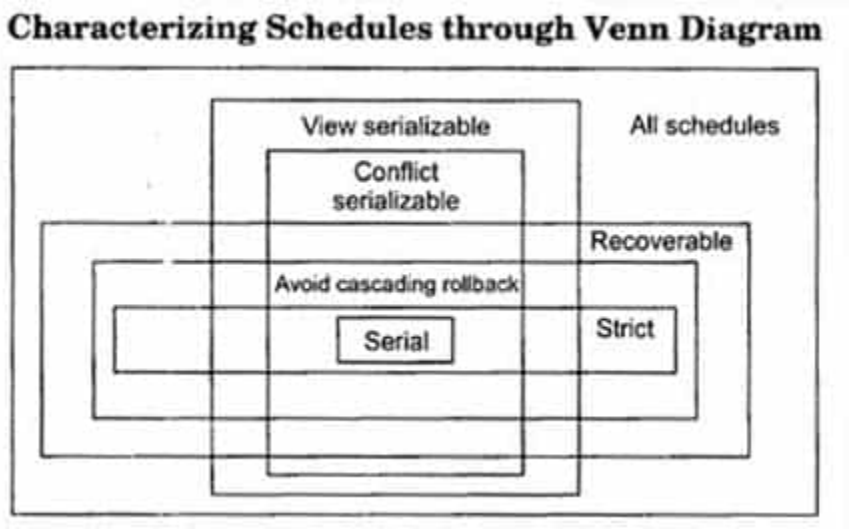
| T1 | T2 |
|---|---|
| W(A) | |
| | R(A) |
| | W(B) |
| | COMMIT |
| R(B) | |
| COMMIT | |

- **Inconsistent/unrepeatable read** (**RW conflict**): a transaction accesses data before and after another transaction writes data. (Perhaps more apt to call a "RWR conflict"?) See the following schedule.

| T1 | T2 |
|---|---|
| R(A) | |
| | W(A) |
| | COMMIT |
| R(A) | |
| COMMIT | |

Clearly, this schedule is not serializable and violates atomicity/isolation.

There are more complex serializability classifications for schedules. These classifications are not all covered here, but they are covered in the article: `https://byjusexamprep.com/transactions-and-concurrency-control-i-4c5d9b27-c5a7-11e5-bc`



We can attempt to improve concurrency control and avoid the aforementioned conflicts using locking. **Two-phase locking** (2PL) is when all locking operations percede the first unlock operation in the transaction. There are several variations of this:

- **Basic 2PL**: transaction locks data items incrementally. May cause deadlock.
- **Conservative 2PL**: prevents deadlock by locking all data items before transaction begins execution. Requires predeclaring read- and write-sets.
- **Strict 2PL**: like basic 2PL, but write locks are only unlocked after transaction terminates. Like basic 2PL, not deadlock-free. Ensures that the schedule is recoverable and cascadeless.
- **Rigorous 2PL**: like basic 2PL, but doesn't unlock any until transaction terminates.

# 7 Schema refinement

TODO

# 8 Physical database design

TODO

# 9 Indexing

TODO

## 10 Query evaluation

TODO

# 11 NoSQL

Idea of "polyglot persistence" as opposed to NoSQL: choose the best tool for the job, whether it be SQL/relational or not. SQL is standardized and provides ACID guarantees on transactions, but it has some problems, especially being difficult to scale horizontally, in a time when vertical scaling is limited.

## 11.1 Unifying characteristics

There is no definition for NoSQL databases, but they tend to have the following common features:

- Don't use a relational model nor SQL.
- Are designed to run on a cluster.
- Are open source.
- Don't have a fixed schema.

## 11.2 Examples

There are many types/classes of NoSQL databases, such as:

- Key-value pair (e.g., Redis)
- Document (e.g., MongoDB)
- Columnar (e.g., HBase)
- Graph (e.g., Neo4J)
- Spatial (e.g., OpenGEO)
- In-memory (real-time applications)
- Cloud

Some specific implementations include:

- MongoDB
- CouchDB
- Neo4J

- Cassandra
- Riak
- Apache HBase
- Redis
- Bigtable (Google)
- SimpleDB (Amazon)

### 11.3 Benefits/disadvantages

NoSQL databases are useful for strategic projects that (require rapid time to market and/or are data intensive). Benefits of NoSQL:

- Reduce development drag without using the relational model. It may be easier to model data in the more natural models that NoSQL databases have to offer.
- More easily scale (horizontally).

Benefits of SQL:

- Tabular data is still useful for many things.
- ACID properties may still be desirable.
- SQL tooling is much more mature.
- SQL is much more familiar.

### 11.4 Weakening ACID (also: the CAP theorem, rise of BASE)

Strict adherance to isolation (serializability in concurrent transactions) is difficult and severely limiting on the concurrency of DBMSes, so it may be useful to adhere to "weak isolation" (See Red Book, chapter 6: "Weak isolation and distribution"). This forces us to allow some conflicts or anomalies, but may be acceptable as long as we understand what they are.

Brewer's **CAP Theorem** states that only two of the following three can be guaranteed in a distributed data store (in the case of a **network partition**):

- **Consistency**: Every read receives the most recent write or an error.

  - (Note that this definition of consistency is different than that of ACID)

- **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write.

- **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

Most NoSQL implementations implement the less-strict and less-strictly-defined **BASE principles**:

- **Basically Available**: reading and writing operations are available as much as possible (suing all nodes of a database cluster), but may not be consitsent (the write may not persist after conflicts are reconciled, the read may not get the latest write).

- **Soft state**: without consistency guarantees, after some amount of time, we only have some probability of knowing the state, since it may not yet have converged.

- **Eventual consistency**: if we execute some writes and then the system functions long enough, we can know the state of the data; any further reads of that data item will return the same value.

## 11.5 Redis

A simple largely-in-memory key-value store.
Advantages:

- Very fast.

- Tiny.

- Complex data structures for many needs. (E.g., hashtables for sub-keys.)

- Pubsub functionality.

Disadvantages:

- Not as natural to have relational data.

- Poor consistency/durability guarantees.

### 11.6 MongoDB

"A document database designed for ease of application development and scaling." Allows you to model data more naturally in a hierarchical JSON (technically, BSON) format. Some important features:

- **Sharding**: sharding is the splitting of a relation across multiple servers. It allows us to easily scale a relation horizontally. The server that a document is located on is based on the **shard key**, and the choice of this key is important for performance. It is orthogonal to replicating data (think RAID 0 : RAID 1 :: sharding : replica sets).
- **Replication**: data replication is performed using **replica sets**. Each replica set has a **primary** server (the remaining servers are called **secondary** servers). This improves availability and provides redundancy (fault tolerance). We can establish **read/write preferences/concerns**, such as waiting until there is majority consensus on a value or the option to read from a secondary server. If the primary server goes down, then the secondaries automatically elect a new primary (**automatic failover**).
- **Indices**: allow index scans to be performed. Indices are implemented as non-clustered B-trees. Indices should be manually created by the user to speed up their queries. There is an default index on the `_id` field.
- **Atomicity and transactions**: operations on a single document are atomic. Multi-document operations (i.e., the `*Many()` operations) are not atomic per document. Later versions of MongoDB do support multi-document transactions for atomic R/W operations.

Other concepts, less specific to MongoDB's implementation:

- **Two-phase commit protocol** (2PC): (note: different from 2PL introduced before when talking about concurrency control).
- JSON vs. XML: JSON has become more popular due to the ubiquity of JavaScript. XML tends to be more verbose, require doctype specifications, and has some ambiguity over when to use attributes as opposed to nested tags.
- MongoDB vs. Redis: MongoDB offers a natural way to represent data, and thus be great for quick prototyping. Redis is good for high-speed querying and write-heavy workloads, such as a cache.

# 12 Big data

Our world is a data-driven one. . . . . We collect data, then do something with it (advertising, sports gambling, monitor climate trends, predict protein folding, etc.). Some sample tools and techniques include:

- Basic data manipulation and analysis (e.g., queries)
- Data mining (e.g., frequent item-sets, association rules)
- Machine learning (e.g., classification, clustering, regression)
- Data visualization (e.g., charts for public consumption; careful of misleading representations)
- Data collection and preparation (e.g., data cleaning, de-duping, etc.)

We have specialized data processing models for scalable systems: MapReduce/Hadoop and Spark, and cloud providers are able to provide much processing power for cheap (AWS, GCP, Azure). See this for a comparison of Hadoop and Spark: Hadoop is better for linear parallel processing, while Spark is faster for specialized workloads.

**Data privacy** is a big issue with big data. General cybersecurity stuff: e.g., systems that might be secure in isolation may become insecure when considered with other datasets (e.g., taxi medallion/location data combined with camera data.)

We can roughly measure big data by the **3 V's**:

- **Volume**: data size (TB)
- **Velocity**: speed of creation or change (TB/s)
- **Variety**: type (text, audio, video, images, geospatial. . . )

Big data has been accelerated by many factors: lower cost for processing and storage, more powerful computers, low latency of distributed computers, better networking, better virtualization, better ML techniques, better scalable algorithms (MapReduce/Hadoop), rise of cloud providers, more OSS, government grants.