Data Management Cheatsheet

Edoardo Riggio

June 19, 2021

Computer Networking - SP. 2021
Computer Science
Università della Svizzera Italiana, Lugano

Contents

1 Entity Relationship Models

1.1 Purpose

Provides a common, informal, and convenient method for communication between application end users and the database designers in order to model the information's structure.

The ER model frequently employs **ER diagrams**, which are pictorial descriptions to visualize the information's structure.

1.2 Basic Concepts

The three basic concepts are:

1.2.1 Entity

It is an "object". All entities of the same "type" form an **entity set**.

Pictorially, it is denoted by a rectangle, with its type written inside. It must be a singular noun and capitalized (all capitals for acronyms).
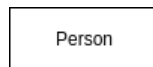
1.2.2 Attribute

An entity can have a set of zero or more attributes, which are some properties. All entities in the entity set has the same set of properties, though not generally with the same values.

Pictorially, attributes of an entity are written in ellipses connected to the entity.
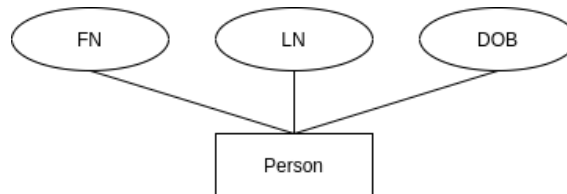
Attributes can be of several different types:

- **Base**

- **Simple**

- **Single-valued**

- **Derived**

  An attribute which value is derived from other factors (e.g. age from DOB and the current date).
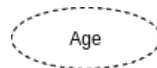
- **Composite**

  An attribute that has multiple component attributes attached to it.
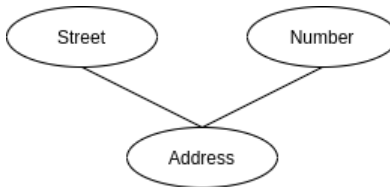
- **Multi-valued**

  Unspecified number of values in this type of attribute.

1.2.3 Relationship

Several entity sets can participate in a relationship.

Pictorially, a relationship is defined as a diamond with a verb inside. This verb should be in third person singular and capitalized.

Relationships can be:

- **Binary**

  A binary relationship can be **many-to-one** iff for each element of A there exists at most one element of B related to it (e.g. the relationship "born" between a person and a country).
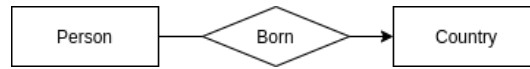
A binary relationship can be **one-to-one** iff for each element of A there exists at most one element of B related to it, and for each element of B there exists at most one element of A related to it (e.g. the relationship "heads" between a person and a country).
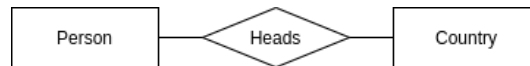
A binary relationship can be **many-to-many** if it is not many-to-one from A to B, and it is not many-to-one from B to A (e.g. the relationship "likes" between a person and a country).

- **Ternary**

  This relationship is between three distinct entity relationship.

- **Non-Distinct Entity Set**

  Frequently in this case is useful to gives roles to the participating entities.
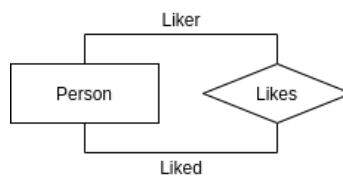
1.3 Keys

Some subset of the attributes of an entity has the property that two different entities in an entity set must **differ on the values of the attributes**. Such a set of attributes is called a **superkey**. A minimal superkey is called a **key** (also called a **candidate key**).

1.3.1 Primary Keys

If an entity as one or more keys, then one of them is chosen as the **primary key**.

1.4 Relationships as Entities

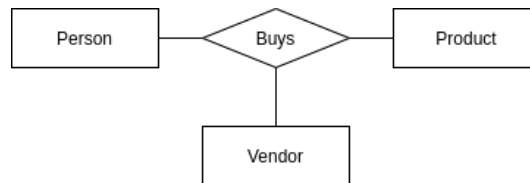By considering relationships as entities, it allows us to let relationships participate in other "high order" relationships.
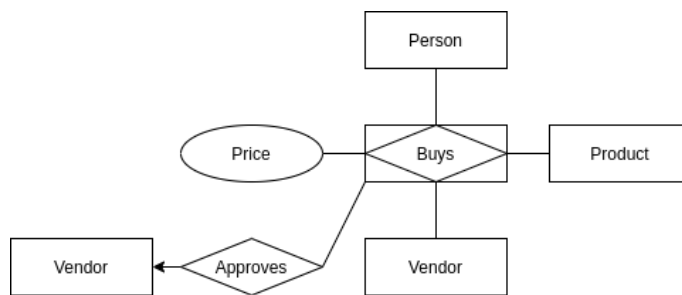
1.5 Strong and Weak Entities

The elements of a **strong entity set** can be identified by the values of their attributes. That is, it has a primary key made of its attributes.

The elements of a **weak entity set** cannot be identified by the values of their attributes. There is no primary key made from its own attributes.

In the case below, a Man can only be identified by the combination of: the Woman to whom he is related, and his Name – which is now a **discriminant**.
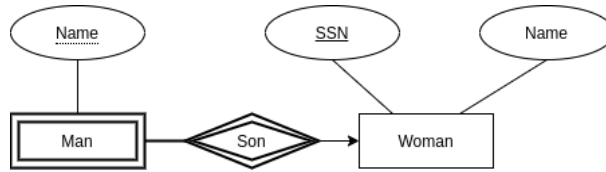
1.6   ISA Relationships

The subset relationship between the set and its subset is called ISA. The elements of the subset all have the same attributes and relationships as the elements of the set. In addition, they may participate in relationships and have attributes that make sense for them.

The elements of the subset are **weak entities**.
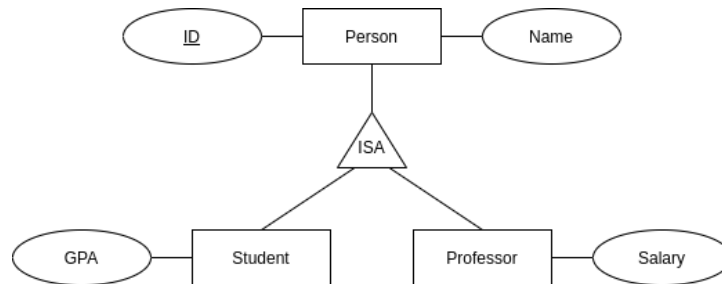
- **Disjoint**

  No Entity could be in more than one subclass.

- **Overlapping**

  An Entity could be in more than one subclass.

- **Total**

  Every Entity has to be in at least one subclass.

- **Partial**

  An Entity does not have to be in any subclass.

1.7 Cardinality Constraints

It is possible to specify how many times each entity from some entity set can participate in some relationship. Constraint can be specified as:

- **0..***

  Means there are no constraints.

- **i..i**

  Means that the constraint must be exactly i.

- **i..j**

  Means that the constraints can go from i up to j.

Constraints are **look across**, meaning that in the following example:

- Every person likes exactly 1 country

- Every country is liked by 2 or 3 people

2 Relational Model

2.1 Sets

A set is a "bag" of elements, some or all of which could be sets themselves. It is not possible to specify **how many times** or **in which position** does an element appear in a set.

2.2 Relations

Relations are elements such as **primary keys**, **keys**, **foreign keys**...

2.2.1 Keys and Superkeys

A set of columns in a relation is a **superkey** iff any two tuples that are equal on the elements of these columns are completely equal. A relation has always **at least one** superkey. The set of all the attributes is a superkey.

A minimal superkey is a **key**. A relation always has at least one key (but there could be more than one). Exactly one key is chosen to as the **primary key**.

2.2.2 Foreign Keys

A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables. It must always reference to a primary key somewhere in the database.

2.3 Crow's Feet

Crows Feet are used to indicate constraint. The following are the possible Crow's Feet:

3 Normalization

3.1 Natural Join

The natural join is a Cartesian join with condition of equality on corresponding columns. Only one copy of each column is left.

3.2 Lossless Join Decomposition

This is another term for information not being lost, that is we can reconstruct the original table by combining information from the two new tables by means of a natural join.

3.3 Normalization

Normalization deals with reorganizing a relational database by breaking up tables to remove various anomalies.

3.4 Normal Forms

A normal form applies to a table/relation schema, not to the whole database schema.

3.4.1 1NF (First Normal Form)

In 1NF we have a relation, this means that there are no repeating groups and that the number of columns is fixed.

3.4.2 2NF (Second Normal Form)

2NF means that the table is in 1NF and there are no **partial dependencies**.

3.4.3 3NF (Third Normal Form)

3NF means that the table is in 2NF and there are no **transitive dependencies**.

3.4.4 BCNF (Boyce-Codd Normal Form)

BCNF means that the table is in 1NF and every functional dependency is from a full key. A table in BCNF is automatically a table in 3NF

3.5 Functional Dependencies

The database rules can be described using functional dependencies. Let P and Q be sets of columns, then P functionally determines Q:

$$P \rightarrow Q$$

Iff any two rows that are equal on P must be equal on Q.

There are several types of keys:

- **From the full key**

- **Not from the full key**

    - **Partial Dependency**

        From a part of the primary key to outside the key.

    - **Transitive Dependency**

        From outside the key to outside the key.

    - **Into key dependency**

        From outside the key into (all or part of) the key.

A set of attributes is a **key** iff $X^+ = R$ (meaning that the key X is composed of all attributes R).

3.6 Finding Keys

In order to find all of the possible keys, we can divide the attributes into four distinct classes:

1. **Appearing on both sides of FDs**

2. **Appearing on left sides of FDs only**

3. **Appearing on right sides of FDs only**

4. **Not appearing in FDs**

Staring from attributes in classes 2 and 4, add as many attributes in class 1 as needed. Ignore the attributes in class 3.

3.7 Minimal Cover

It is needed in order to simplify the set of FDs. It is done by following these steps:

1. **Combine right hand sides**

   If the left hand sides of the two FDs are the same, then they can be merged into one FD.

2. **Right hand side simplification**

3. **Left hand side simplification**

3.8 Create Tables

In order to create the tables of the database, we need to follow these steps:

1. **Get a minimal cover of the FDs**

2. **Create tables from the minimal cover**

3. **Remove redundant tables**

4. **Ensure the storage of the global key**

   We need to check if one of the tables already contain the global key. If none contain the global key, then it is stored in another table.

3.9 Data Structures

3.9.1 Heap

Assuming continuous storage, time complexities for **finding** elements are:

$$O(1) \text{ or } O(n)$$

It takes $O(1)$ in the case we are searching the head or tail (assuming both of them are accessible in time $O(1)$) of the heap. It takes $O(n)$ if we are trying to find any element in the heap.

Time complexities for **deleting** elements are:

$$O(1) \text{ or } O(n)$$

It takes $O(1)$ in the case we are deleting the head or tail (assuming both of them are accessible in time $O(1)$) of the heap, and there is no memory compacting. It takes $O(n)$ if we are trying to delete any element which is accessible in 1, and there is memory compacting.

Time complexities for **inserting** elements are:

$$O(1) \text{ or } O(n)$$

It takes $O(1)$ to insert an element in the front or back of the heap (if it is accessible in $O(1)$). It takes $O(n)$ if the end of the heap cannot be accessed in constant time.

3.9.2 Sorted Sequence

Time complexity for **finding** an element is:

$$O(\log(n))$$

It takes $O(log(n))$ to find an element in the sorted sequence. This is because binary search is used.

Time complexities for **deleting** elements are:

$$O(\log(n)) \text{ or } O(\log(n) + n)$$

It takes $O(\log(n))$ to delete the element because binary search is used, and there is no memory compacting. Since the time of binary search is $O(\log(n))$, and

$O(1)$ to remove the found element, the total complexity is $O(\log(n))$. Deleting an element from a sorted sequence takes $O(\log(n) + n)$ when removing any element which is neither the head or the tail of the sequence (assuming both of them are accessible in constant time). In order to delete the element in the sequence it would take $\log(n)$ to find the element (assuming we are using some sort of binary search algorithm), $O(1)$ to delete the element, and then $O(n)$ to compact the memory.

Time complexities for **inserting** elements are:

$$O(\log(n)) \text{ or } O(\log(n) + n)$$

It takes $O(\log(n))$ to insert an item if no compacting is used. In this case the time is used to search the space, and then constant time is used in order to insert the element. It takes $O(\log(n) + n)$ if there is memory compacting. In this case $O(\log(n))$ is used to find the space, constant time to insert the element, and $O(n)$ in order to recompact memory.

3.9.3   Hashing

Hashing is used in the following cases:

- If the set of integers is large

- If many of the queries are not range queries

3.9.4   B+ Tree

B+ tree is used in the following cases:

- If the set of integers is large

- If many of the queries are range queries

In order to computer the maximum number of nodes an internal node can have, we can use the following formula:

$$m \cdot (P) + (m - 1) \cdot (K) \leq B$$

Where **P** is the size of the pointer, **K** is the size of the key, and **B** is the size of the block (number of sectors).

Given m, we know that the number of children of the **root** is between 2 and m, while the number of children of an **internal node** is between $\left\lceil \frac{m}{2} \right\rceil$ and m.

3.10 Indexes

3.10.1 Dense Index

An index is dense if for every key appearing in (some record) of the data file, a dedicated pointer to the block containing the record appears in (some record) of index file

3.10.2 Sparse Index

An index is sparse if for not every key appearing in (some record) of the data file, a dedicated pointer to the block containing the record appears in (some record) of index file

3.11 Files

3.11.1 Clustered File

In a clustered file, blocks are composed of elements which are "close to" in memory – as if the files were first sorted and then dispersed in memory.

3.11.2 Unclustered Files

In an unclustered file, blocks are composed of elements which are not "close to" in memory.

3.12 External Merge Sort

In order to compute the number of passes of the external merge sort, we use the following formula:

$$N_P = 1 + \left\lceil \log_{B-1} \left(\frac{N}{B} \right) \right\rceil$$

Where **B** is the number of RAM blocks, and **N** is the number of blocks to sort.

From the number of passes we can compute the cost as:

$$C = (2 \cdot N) \cdot N_P$$

Where **N** is the number of blocks to sort, and N_P is the number of passes.

4 Transaction Processing - Recovery

A **transaction** is an execution of a user's program. A transaction is supposed to satisfy the **ACID** conditions. These conditions are:

- **Atomic**

  It is not divisible into smaller units of executions. A transaction is either executed completely, or not executed at all.

- **Consistent**

  It preserves the consistency of the database. If started on a correct database, and finishes successfully, it will leave a correct database.

- **Isolated**

  A transaction is given the illusion of running on a dedicated system, so concurrency errors cannot be introduced. Transactions do not interact with one another.

- **Durable**

  Once completed correctly, the values the transaction produces will be never forgotten – written to the database disk.

4.1 Recovery

The job of recovery is to make sure that the transaction satisfies **ACD** properties – as **I** is not relevant in this case.

If a failure occurred while a transaction was executing, we cannot continue, and therefore we need to **restore the database** to the state before the failed transaction started.

If a failure occurred after a transaction finished executing, the state must **continue reflecting** this transaction.

A history is **recoverable** if, for every transaction T that commits, the commit of T follows the commit of every transaction from which T read from.

4.2 Cascading Aborts

A history **avoids cascading aborts** if every transaction reads only values produced by transactions that have already committed.

$$\text{No Cascading Aborts} = \text{Recoverable}$$

4.3 Strict Histories

A history is said to be **strict** if it satisfies the condition for avoiding cascading aborts, and – for every transaction that writes an item – all the transactions that previously wrote that item have already committed or aborted.

$$\text{Strict} = \text{No Cascading Aborts}$$

4.4 Checkpointing

This is a technique that is used in order to obviate the need to look at the complete log and thus reduce the work during recovery.

During normal execution, the following steps are followed in order to checkpoint:

1. Stop processing

2. Force the log buffers on the disk log

3. Force the database buffers on the database disk

4. Write onto the log a list of the transactions currently running

5. Write "CHECKPOINT DONE" onto the log

6. Resume processing

Checkpointing synchronizes the database with the log.

4.5 Recovery with Checkpointing

In order to recover after a crash, we start by scanning the log **backwards** until you reach the first checkpoint. At this point you have two empty lists: the undo and the redo list.

For every transaction that have a commit record, it is added to the redo list. For each transaction that have a start record but no commit record, it is added to the undo list. Finally, for each transaction that is listed in the checkpoint record for which there is no commit record, it is added to the undo list.

All of the transactions in the undo list are undone, and all the transactions in the redo list are redone.