COMP SCI 564: DBMS

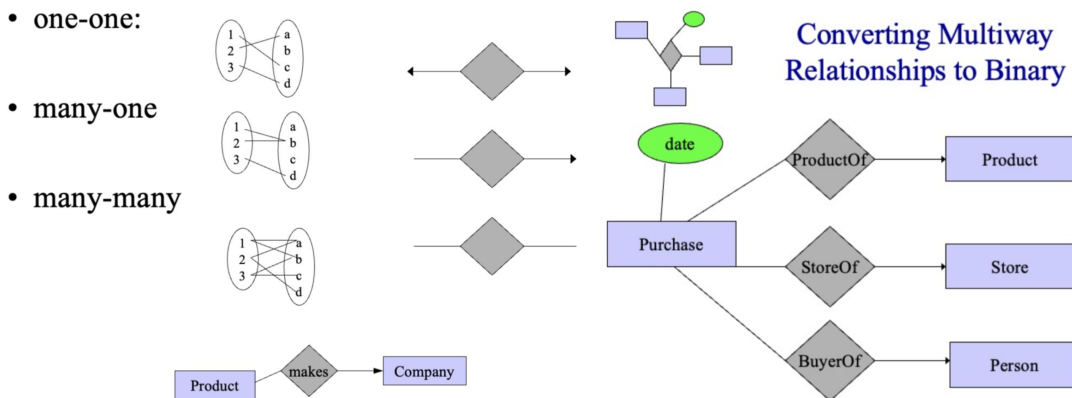
Midterm, Fall 2022 (Lecture: AnHai Doan; Slide: AnHai Doan, Paris Koutris, R. Ramakrishnan)

Ruixuan Tu (ruixuan.tu@wisc.edu), University of Wisconsin-Madison

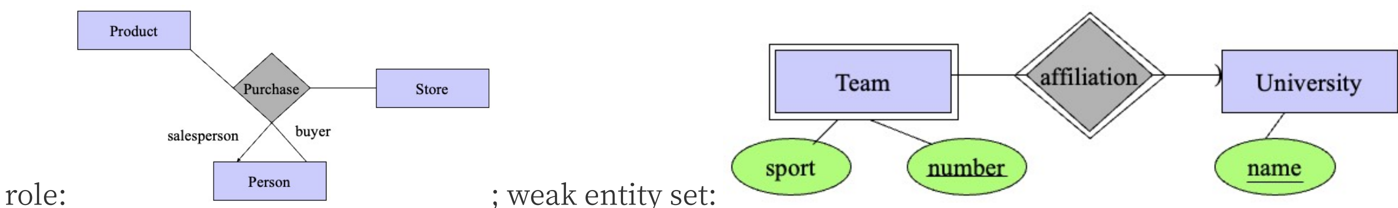
---

ER model

- **entity**: an object in the real world that is distinguishable from other objects which is described using a set of attributes; represented by rectangles (ER diagram)
- **entity set**: a collection of similar entities. more than the name of something: has at least one nonkey attribute; OR the “many” in a many-1 or many-many relationship
- **relationship/table**: a set of n-tuples/records of entities, i.e., $E_1 \times \cdots \times E_n$ where data is stored in
- **attributes**: a set to describe an entity or a relation, each has an atomic type/domain: string, integers, reals, etc.; represented by ovals attached to an entity set (ER diagram)
- **binary relations**:



- **multiway relation**: a relation connecting multiple entity sets, could convert to many binary relations by treating this multiway relation as an entity with many-1 relations to connected entities
- **tenary relation**: an association among 3 entity sets (3-tuple)
- **role**: an entity set multiple times in one relationship; label the edges to indicate the roles (ER diagram)



- **subclass/is-a hierarchy**: special case = fewer entities = more properties: is-a triangles (ER diagram) indicate the subclass relationship. (isa is 1-1 relation without arrow; the point of the triangle points to the superclass; isa structure must be trees)

- **weak entity set**: Entity sets are weak when their key attributes come from other classes to which they are related (i.e., part-of hierarchies, splitting n-ary relations to binary). Entity set E is said to be weak if in order to identify entities of E uniquely, we need to follow one or more many-1 relationships from E and include the key of the related entities from the connected entity sets
- **constraint**: an assertion about the database that must be true at all times; part of the database schema
- **key constraint/key**: requirement of a minimal set of attributes whose values uniquely identify an entity in the set
- **single-value constraint**: requirement of at most one value for a given attribute, relationship, or role for an entity; the attribute value must be present or can be missing (represented by `NULL` or `-1`); a many-1 relation implies single value constraint
- **referential integrity constraint**: requirement of exactly one value for a relationship or role; an attribute has a non-`NULL`, single value; more commonly used to refer to relationships; on relationships explicitly requires a reference to exist; different arrow, see weak entity set (ER diagram)
- **domain constraint/domain**: requirement of an atomic type, i.e., a set of possible values for each attribute associated with an entity set
- **primary key**: There could be more than one candidate key; if so, we designate one of them as the primary key. Underline primary key only suggested, multiple keys not formal (ER diagram). More than one primary key - composite key
- **design principles**: be faithful, avoid redundancy, KISS, limit the use of weak entity sets, don't use an entity set when an attribute will do

Relational model and translating ER to relational

- **schema**: entity set E = relation with attributes of E; relationship R = relation with attributes being keys of related entity sets + attributes of R; e.g., `Product(name, price, category, manufacturer)`; in practice, we add the domain for each attribute
- **instance** of a relationship set: a set of relationships, or a "snapshot" of the relationship set at some instant in time
- **translating/merging many-1 relations** / combining the relation for an entity-set E with the relation R for a many-1 relationship from E to another entity set: `Drinkers(name, addr)` + `Favorite(drinker, beer)` = `Drinker1(name, addr, favoriteBeer)`
- **translating many-many/1-1 relations**: risky many-many translation leads to redundancy if not with id (i.e., with repeated fields); 1-1 relationship should be merged with one of the entities tables
- **translating weak entity sets**: Relation for a weak entity set must include attributes for its complete key (including those belonging to other entity sets), as well as its nonkey attributes. A supporting (double-diamond) relationship is redundant and yields no relation
- **translating subclass entities**
  1. *Object-oriented*: each entity belongs to exactly one class; create a relation for each class, with all its attributes (good for querying a specific subclass)
  2. *E/R style*: create one relation for each subclass, with only the key attribute(s) and attributes attached to that E.S.; entity represented in all relations to whose subclass/E.S. it belongs (good for querying all subclasses)

3. *Use `NULL` s*: create one relation; entities have `NULL` in attributes that don't belong to them (good for space saving unless lots of `NULL`)

SQL

- **SQL Query Build**
  - Two single quotes inside a string represent the single-quote (apostrophe)
  - SQL is case-insensitive except inside quoted strings
  - **`SELECT S FROM R1, ..., Rn WHERE C1 GROUP BY a1, ..., ak HAVING C2`** : `SELECT` desired attributes (`S` may contain attributes `a1, ..., ak` and/or any aggregates but NO OTHER ATTRIBUTES); `FROM` one or more tables; `WHERE` condition about tuples of tables (`C1` is any condition on the attributes in `R1, ..., Rn`); `HAVING` (`C2` is any condition on aggregate expressions)
  - **Semantics/Evaluation**: (1) Compute the FROM-WHERE part, obtain a table with all attributes in `R1, ..., Rn` ; (2) Group by the attributes `a1, ..., ak` ; (3) Compute the aggregates in `C2` and keep only groups satisfying `C2` ; (4) Compute aggregates in `S` and return the result; Implementations (nested loops, parallel assignment without order, translation to relational algebra)
- **Single-Table Queries**
  - **`SELECT`** : `SELECT *` all attributes of this relation in `FROM` ; `SELECT name1 AS <new name>, name2` to rename an attribute `name1` ; Any expression that makes sense can appear as an element of a SELECT clause (including constant expression, e.g., `'likes Bud' AS whoLikesBud`)
  - **`WHERE`** : attribute names of the relation(s) used in `FROM` ; comparison operators: `=` , `<>` (not equal), `<, >` , `<=` , `>=` ; apply arithmetic operations: `stockprice*2` ; operations on strings (e.g., "||" for concat); Lexicographic order on strings; Pattern matching: `s LIKE p` ; Special stuff for comparing dates and times; `AND` , `OR` , `NOT` , parentheses in the usual way boolean conditions are built
    - **`s LIKE p`** : pattern matching on strings; `p` special symbols: `%` = any sequence of chars, `_` = any single char
- **Multi-Table Queries**
  - **Several relations** in one query by listing in `FROM` ; **Distinguish** attributes of the same name by `<relation>.<attribute>`
  - **Distinguish** copies of same relation by following the relation name by the name of a tuple-variable in the `FROM` clause (e.g., `FROM Beers b1, Beers b2`)
- **Aggregation**
  - **`SUM` , `AVG` , `COUNT` , `MIN` , and `MAX`** can be applied to a column in a SELECT clause to produce that aggregation on the column
  - **`COUNT(*)`** counts # tuples
  - **`DISTINCT`** inside an aggregation causes duplicates to be eliminated before the aggregation (e.g., `SELECT COUNT(DISTINCT price)`)
  - **`NULL`** never contributes to any aggregation; if no non- `NULL` values in column, the result of aggregation is `NULL`
  - **`SELECT` with aggregation**: If any aggregation is used, then each element of the `SELECT` list must be either: Aggregated, OR An attribute on the `GROUP BY` list (e.g., `SELECT bar, MIN(price)` is ILLEGAL)
- **Group-by and Having**

- `GROUP BY a1, ..., ak` : the relation is grouped according to values of all those attributes, and any aggregation is applied only within each group
- `HAVING <condition>` : groups not satisfying the condition are eliminated; conditions may refer to any relation or tuple-variable in the `FROM` clause; may be A grouping attribute OR Aggregated; (e.g., `SELECT beer, AVG(price) FROM Sells GROUP BY beer HAVING COUNT(bar) >= 3;`)

- `NULL`
  - **Meaning** depends on context, e.g., missing value, inapplicable
  - **SQL Conditions**: `TRUE` , `FALSE` , `UNKNOWN` (when comparing with `NULL`); Query with `WHERE` only returns values with `TRUE`
  - **Testing**: `x IS NULL` , `x IS NOT NULL`
- **Subqueries/Nested Queries**: a parenthesized SELECT-FROM-WHERE statement which can be used as a value in a number of places, including FROM and WHERE clauses; better use a tuple-variable to name tuples of the result; runtime error if there is not *exactly* one tuple
- **Boolean Operators**
  - `IN` : `<tuple> IN <relation>` is true iff the tuple is a member of the relation; `NOT IN` opposite; boolean appear in `WHERE` clauses; `<relation>` often a query
  - `EXISTS` : `EXISTS(<relation>)` is true if and only if the `<relation>` is not empty; boolean appear in `WHERE` clauses
  - `ANY` : `x=ANY(<relation>)` is a boolean condition meaning that `x` equals at least one tuple in the relation; `=` can be replaced by any comparison operator
  - `ALL` : `x<>ALL(<relation>)` is true iff for every tuple `t` in the relation, `x` is not equal to `t` (i.e., `x` is not a member of the relation); `<>` can be replaced by any comparison operator
- **DB Schema**
  - **Create/Remove Relation**: `CREATE TABLE <name> (<list of [element type properties]>);` , `DROP TABLE <name>;`
  - **Types**: `INT` / `INTEGER` , `REAL` / `FLOAT` , `CHAR(n)` (fixed n chars), `VARCHAR(n)` (variable-length up to n chars), `NOT NULL` (reject insertion if unknown), `DEFAULT <value>` (use `<value>` if unknown)
  - **Properties** for attribute element: `PRIMARY KEY` (index, only one for a relation, non-`NULL`), `UNIQUE` (non-index, could multiple for a relation, allow `NULL`)
  - **Multiattribute Keys**: key declaration be another element in the list of a `CREATE TABLE` statement; essential if more than one attributes in key; (e.g., `CREATE TABLE Sells (bar CHAR(20), beer VARCHAR(20), price REAL, PRIMARY KEY (bar, beer));`)
  - **Create/Remove Attribute**: (declaration = an element in `CREATE TABLE` list) `ALTER TABLE <name> ADD <attribute declaration>;` , `ALTER TABLE <name> DROP <attribute name>;`
- **DB Modification**
  - **Insert**: `INSERT INTO <relation>(<attributes>) VALUES (<list of values>);` or `<list of list of values>` ; `(<attributes>)` optional; (e.g., `INSERT INTO Likes VALUES('Sally', 'Bud');` = `INSERT INTO Likes(beer, drinker) VALUES('Bud', 'Sally');`); Reasons Specifying Attributes: (1) We forget the standard order of attributes for the relation; (2) We don't have values for all attributes, and we want the system to fill in missing components with `NULL` or a default value; Also support `INSERT INTO <relation> (<subquery>);`

- **Delete**: `DELETE FROM <relation> WHERE <condition>;` ; `WHERE <condition>` optional to delete all tuples; Procedure: (1) Mark all for deletion by `WHERE`; (2) Delete all marked
- **Update**: `UPDATE <relation> SET <list of attribute assignments> WHERE <condition on tuples>;` (e.g., `UPDATE Sells SET price = 4.00 WHERE price > 4.00;`)

- **Constraints**
  - **trigger**: only executed when a specified condition occurs
  - **types**: keys, foreign-key/referential-integrity, value-based (values of an attribute), tuple-based (relationship), assertions (any SQL boolean expression)
  - **foreign keys**: referenced key must be in the table; `FOREIGN KEY (<list of attributes>) REFERENCES <relation> (<attributes>)` OR `<attribute> REFERENCES <relation>(<attribute>)` in `CREATE TABLE`
    - **violations**: (1) insert/update to R *introduces* values not found in S; (2) deletion/update to S causes some tuples of R to "dangle"
    - **handle violations**: reject insert/update *introduces* nonexist reference; for delete/update *removes* reference: (1) Default - reject; (2) Cascade - make same change in R; (3) Set `NULL` - change referenced value to `NULL`
    - **policy declaration**: after foreign key declaration, add `ON [UPDATE, DELETE][SET NULL, CASCADE]`
  - **attribute-based checks**: after attribute declaration, add `CHECK(<condition> IN <subquery>)`; `<condition>` may use name of this attribute, any other relation or attribute name must be in a subquery; `IN <subquery>` optional
  - **tuple-based checks**: add `CHECK(<condition>)` as an element of `CREATE TABLE` list; `<condition>` may use name of this relation, any other relation or attribute name must be in a subquery; on insert or update only
  - **assertions**: for whole DB like relations (`CREATE TABLE`); `CREATE ASSERTION <name> CHECK (<condition>);` ; `<condition>` may use any relation or attribute in DB
- **Set Operators** `<subquery> <operator> <subquery>`
  - **`INTERSECT`** / **`UNION`**: Returns the tuples that belong in *both*/*either* subquery results
  - **`EXCEPT`** / **`MINUS`**: Returns the tuples that belong in the first and not the second subquery result
  - **`UNION ALL`**: # copies of each tuple is the sum of # copies in the subqueries
  - **`INTERSECT ALL`**: # copies of each tuple is the minimum of # copies in the subqueries
  - **`EXCEPT ALL`**: # copies of each tuple is the difference (if positive) of # copies in the subqueries
- **JOIN**
  - **`INNER JOIN`** / **`JOIN`**: includes tuples only if there is a match on other side; equivalence:

```sql
SELECT C.Name AS Country, MAX(T.Population) AS N FROM Country C, City T
  WHERE C.Code = T.CountryCode GROUP BY C.Name;
SELECT-FROM ... INNER JOIN City T ON C.Code = T.CountryCode GROUP BY C.Name;
```

- **`OUTER LEFT JOIN`**: includes the left tuples even if there is no match on right; fills the remaining attributes with NULL. **`OUTER RIGHT JOIN`**: reverse `OUTER LEFT JOIN`. **`FULL OUTER JOIN`**: includes both left and right tuples even if there is no match on other side; fills the remaining attributes with NULL

Storage and buffer management

- **layered architecture**: query optimization & execution, relational opterators, files and access methods (sorted file, heap file, hash index, B+-tree index), buffer management, disk space management (+ concurrency control manager, recovery manager)
- **reasons why DBMS, not OS, handles cache and disk**: better predicting reference patterns; buffer requires ability to pin page in buffer, force page to disk, adjust replacement policy, pre-fetch pages; better control I/O and computation overlap; leverage multiple disks effectively
- **table, file, page**: Table is stored as a file on disk; File has multiple pages (suppporting insert/delete/modify record, read record with id, scan all records with conditions); Each page has multiple tuples/records, DBMS works on a page at a time in buffer
- **disk blocks/pages**: units where data is stored in (location important); block size is a fixed multiple of sector size
- **HDD**: *location* (ϕ, r) at platter p, *cylinder* has r, *track* has r, p, sector has ϕ, r, p; e.g. `surface 3, track 5, sector 7`; *platters* spin to ϕ by spindle (rpm), *arms assembly* moves to r simultaneously, only one *head* R/W at one time
- **access (R/W) time** = seek time (arm move) + rotational delay (block rotate) + transfer time (actual data move) [sorted from long to short, transfer very short] (causes random time >> sequential time)
- **disk space management layer**: allocate/delete/read/write a page
- **buffer management layer**: serve page requests from higher levels (data must be in RAM to operate); table of <frame#, pageid> pairs is maintained

```
def access(REQUESTED_PAGE_ID):
  if REQUESTED_PAGE_ID not in POOL:
    FRAME_FOR_REPLACEMENT = replacement_policy()
    if FRAME_FOR_REPLACEMENT.page.dirty:  # if modified
      disk_write(FRAME_FOR_REPLACEMENT.page)  # flush
    FRAME_FOR_REPLACEMENT = disk_read(REQUESTED_PAGE_ID)
  POOL[REQUESTED_PAGE_ID].pin_count += 1  # candidate for replacement iff pin_count == 0
  return REQUESTED_PAGE
```

```
def release(REQUESTED_PAGE_ID):  # must call when done
  POOL[REQUESTED_PAGE_ID].pin_count -= 1
```

- **replacement policy**: chooses frame for replacement; can have big impact on # I/O's depending on the access pattern (80-20: OPT>LRU=Clock>FIFO=RAND, Loop: OPT>RAND>FIFO=LRU)
  - **optimal**: 82% hit rate; only **cold-start/compulsory miss** & predicts future
  - heavy computation: **LRU** (Least Frequently Used, Bad for **scan resistence** i.e. looping-sequential), MFU (Most-Frequently-Used), MRU (Most-Recently-Used)
  - less computation: **FIFO**, **Clock** or Approximate LRU (used bit; less computation), **Random**

```
P = RandPage(); // clock hand
if (P.Used == True) { P.Used = False;
  if (P == len(AllPages)) P = 0; // back to front
  else P++; }
else Evict(P)
```

- **sequential flooding**: nasty situation caused by LRU + repeated sequential scans; # buffer frames < # pages in file makes cache invalid; MRU doesn't have this problem
- **page formats**: slotted page format, each page contains a record; record id = <page id, slot #>; search/insert/delete records on page
  - **fixed length records**: unpacked, bitmap (if records in use) + # slots for a page; doesn't move records for free space [packed have problem for record id change for references]
  - **variable length records**: slot directory (records offset + record length) + # slots + pointer to start of free space for a page; can move records without changing record id; delete by setting offset to -1; insert by using any available slot OR reorganize when no available space; free space pointer points to last free space
- **record formats**
  - **fixed length**: base address B for each record; field $\text{Addr}(F_i) = B + \sum_{j=1}^{i-1} \text{Length}(F_j)$, i.e., doesn't need to scan record; field types are same for all records in a file stored in system catalogs
  - **variable length** with fixed # fields: field count + fields delimited by special symbols; OR array of field offsets (direct access to field without scan, efficient storage of `NULL`s, small directory cost)
- **system catalog**: structure (e.g., B+ tree) + search key fields for each *index*; name, file name, file structure (e.g., heap file) + atrribute name and type (for each attribute) + index name (for each index) + integrity constraints for each *relation*; view name and definition for each *view*; + statistics, authorization, buffer pool size, etc. [Catalogs are stored as relations, e.g., `Attr_Cat(attr_name, rel_name, type, position)`]

File organization and indexing

- **index**: a data structure that organizes records of a table to optimize retrieval; based on search key ($\neq$ primary key) attributes; contains a collection of data entries to locate the records [trade off: fast query, slow update]; SQL `CREATE INDEX index_name ON table_name (<list of column_name>);`
- **heap file**: unordered, pages alloc/dealloc for file growth/shrink; keep track of pages in file (pid), free space on pages, records on a page (rid); implementations:
  - **list**: (header page id, heap file name) for a file; 2 pointers + data for each page; 2 lists (full pages, pages with free space) linked to header
  - **page directory**: header page as directory (# free bytes for page free/full + pointer to data page) for a file; much smaller than linked list
- **sorted file**: sort on a single attribute OR on a combination of attributes ("search keys" or just "keys"); not keys of entity set
- **hash file/hash index**: a collection of *buckets* [**bucket** = primary page + overflow pages (linked list of pages), one or more data entries for each bucket, store <k, record> (slower search)/<k, rid> (smaller)/<k, list of rids> (more compact but variable size) for each data entry]; hash function h, search key k, # buckets N: $h(k) \mod N$ = bucket in which the data entry belongs; Records with different k may belong in the same bucket; Good for equality search & constant I/O cost for search/insert
- **B+ tree file**: most used
  - **operations**: scan, sort, equality/range search (efficient), insert/delete tuples
  - **complexity**

    - **search/insert/delete** $O(\log_F N) = h - L_B + OUT$; height-balanced; 1 node = 1 page [F =fanout (# pointers to child nodes coming out of a node, $d+1 \le F \le 2d+1$, higher fanout means smaller depth), N=# leaf pages, typical fill factor $f = \frac{2}{3}$ (% available slots in tree that are filled), height $h = \left\lceil \log_F \frac{N}{f} \right\rceil$, L_B = # levels, OUT = cost reading sequential pages for range]
    - minimum 50% **occupancy** (except for root): each node has $d \le \underline{m} \le 2d$ entries; root node has $1 \le \underline{m} \le 2d$ entries (d =order of tree)
    - **internal nodes**: index entries (direct search); leaf nodes: data entries ("sequence set")
    - one-way arrow to child node, two-way arrow between leaf nodes
    - read can have at **minimum** 1 disk I/O if all internal nodes are cached, and I/O at reading leaf
  - **insertion**
    - Find correct leaf node L after the smallest possible node or at the beginning
    - Insert data entry in L
      - If L has enough space, done!
      - Else, must *split* L (into L and a new node L')
        - Redistribute entries evenly, copy up middle key (inserted at right) at index $\left\lceil \frac{n}{2} \right\rceil$ in merged array
        - Insert index entry pointing to L' into parent of L
    - This can propagate *recursively* to other nodes
      - To *split index node*, redistribute entries evenly, but *push up* middle key (contrast with *leaf splits*)
    - Splits "grow" tree (wider/one level taller at top); root split increases height
  - **deletion**
    - Start at root, find leaf L where entry belongs
    - Remove the entry
      - If L is at least half-full, done!
      - If L has only $d-1$ entries,
        - Try to *re-distribute*, borrowing one or more from *sibling* (adjacent node with same parent as L; prefer choosing the sibling which will have minimal redistribution), middle key copied up
        - If re-distribution fails, *merge* L and sibling
    - If merge occurred, must delete entry (pointing to L or sibling) from parent of L
    - If redistribution occured in internal node, borrow from upper level, and push the edge element at another side to upper level, until balanced
    - Merge could propagate to root, middle key moved up, decreasing height
  - **duplicate keys** solutions: (1) All entries with a given key value reside on a single page, Use overflow pages; (2) Allow duplicate key values in data entries, Modify search operation
- **clustered index**: the order of records matches the order of data entries in the index
- **primary index**: an index which the search key contains the primary key (unique, one)
- **secondary index**: an index which is not primary index
- **unique index**: an index which the search key contains a candidate key