CSC343 Notes

Database Management System (DBMS): Software providing UI for creating/managing/accessing database data

- Behind the scenes, they parse/optimize queries, basic algebra, access data, manage disk space and buffers
- Many uses beyond an Excel spreadsheet:
  - Specifying/enforcing logical structure of data
  - Querying and modifying data
  - High-performance under heavy loads (huge data, many queries)
  - Data durability, safe, remains "intact" even in external failures (eg. power outages)
  - Concurrent/simultaneous access by multiple users/processes

Relation: A database table. Formally, on domains $D_1, \ldots, D_n$ (ie. sets of values), a subset of $D_1 \times \ldots \times D_n$.

- In some models, they're bags/multisets – sets with duplicates – like in commercial DBMSs (eg. Postgres)
- In some models, they're sets – we will stick with this in our math

Tuple: A row — eg. row of data for a person
Attribute: A column — eg. columns for age, name, gender, notated something. attribute
Arity: Of a relation, the number of attributes/columns
Cardinality: Of a relation, the number of tuples/rows (doesn't include title row)

Schema: Rules/restrictions of data; the relation's structure. Rarely changes — eg. Person(age, name, gender)
Instance: An actual piece of data inside the database. Constantly changes
Conventional Database: Stores current version of data
Temporal Database: Stores history of data

Superkey: For a relation, a set of attributes that disallows duplicates; it uniquely identifies a row.

- Formally, set of attributes $a_1, \ldots, a_n$ for relation R where $\nexists$ tuples t_1, t_2 where $t_1.a_i = t_2.a_i$
- eg. Person(age, name, gender) means the database doesn't accept two people with the same age and name
- eg. Person(age, name, gender) is the same as Person(age, name, gender)

Key: A superkey, if none of its subsets is a superkey. Notated with underlines.

- eg. Course(dept, number, name, breadth)
  - $\{\text{dept, number}\}$ is a key and superkey
  - $\{\text{dept, number, name}\}$ is a superkey
- Superkeys are supersets of keys
- There can be multiple keys – they will always be disjoint
- Keys mean no duplicates are allowed **in principle**, not that there happens to be no duplicates.
- Domain experts decide keys, we often invent attribute keys anyways to keep tuples unique (eg. book ISBN)

Foreign Key: A key that refers to a key in another table. Reduces redundancy

- $R[A_1, \ldots, A_n]$ is the set all tuples in relation R with attributes $A_1, \ldots, A_n$
- $R[A_1] \subseteq S[A_2]$ limits A_1's possible values to A_2's values, but is not necessarily a foreign key
  - If A_1, A_2 are keys, then A_1 is a **foreign key** referencing A_2
  - $R[A_1, A_2] \subseteq S[B_1, B_2] \Leftrightarrow R[A_1] \subseteq S[B_1]$ and $R[A_2] \subseteq S[B_2]$

Artists

aID	aName	nat
1	Nicholson	American
2	Ford	American
3	Stone	British
4	Fisher	American

Artists[aName, nat]

aName	nat
Nicholson	American
Ford	American
Stone	British
Fisher	American

Artists[nat]

nat
American
British

(Integrity) Constraint: Any sort of protocol/rule that a relation must follow.

- **Key Constraint:** Keys must uniquely identify tuples
- **Inclusion Dependency/Referential Integrity Constraint:** A constraint of form $\mathrm{R}[\mathrm{A}_1] \subseteq \mathrm{S}[\mathrm{A}_2]$
    - **Foreign Key Constraint:** A constraint of form $\mathrm{R}[\mathrm{A}_1] \subseteq \mathrm{S}[\mathrm{A}_2]$ where $\mathrm{A}_1, \mathrm{A}_2$ are both keys
    - $\mathrm{R}[\mathrm{A}_1, \mathrm{A}_2] \subseteq \mathrm{S}[\mathrm{A}_3, \mathrm{A}_4]$ is a foreign key constraint if $\mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3, \mathrm{A}_4$ are all superkeys
- **Functional Dependency (FD):** The constraint $\forall$ attributes $\mathrm{A}_1, \ldots, \mathrm{A}_n, \mathrm{B}_1, \ldots, \mathrm{B}_n$ & tuples $\mathrm{t}_1, \mathrm{t}_2, \mathrm{t}_1[\mathrm{A}_1, \ldots, \mathrm{A}_n] = \mathrm{t}_2[\mathrm{A}_1, \ldots, \mathrm{A}_n] \Rightarrow \mathrm{t}_1[\mathrm{B}_1, \ldots, \mathrm{B}_n] = \mathrm{t}_2[\mathrm{B}_1, \ldots, \mathrm{B}_n]$
    - Notated $\mathrm{A}_1, \ldots, \mathrm{A}_n \to \mathrm{B}_1, \ldots, \mathrm{B}_n$ or $\mathrm{A}_1 \cdots \mathrm{A}_n \to \mathrm{B}_1 \cdots \mathrm{B}_n$
    - "Attributes $\mathrm{A}_1, \ldots, \mathrm{A}_n$ functionally determine $\mathrm{B}_1, \ldots, \mathrm{B}_n$"
    - **Trivial FD:** A FD like $\mathrm{A}_1 \to \mathrm{A}_1$, where the LHS and RHS share attributes
    - $\mathrm{A}_1, \ldots, \mathrm{A}_k$ is superkey of $\mathrm{R}(\mathrm{A}_1, \ldots, \mathrm{A}_n) \Leftrightarrow \mathrm{A}_1, \ldots, \mathrm{A}_k \to \mathrm{A}_1, \ldots, \mathrm{A}_n$

In LHS of a fd	*In RHS of a fd*	*Conclusion on Attribute*
✓	✓	Inconclusive
✓	✗	Is in all keys
✗	✓	In no keys
✗	✗	Is in all keys

```python
# attributes is a set like {'a', 'b', …}
# fds is a set of functional dependencies
# fd has attribute sets fd.lhs and fd.rhs

def closure(attributes, fds):
    """Find attributes⁺, everything that
    attributes functionally determines."""
    c = attributes
    updated = True
    while updated:
        updated = False
        for fd in fds:
            if all(attr in c for attr in fd.lhs):
                c.update(fds.rhs)
                updated = True
    return c

def project(fds, attributes):
    """Find the projection of fds to
    attributes – rules that fds
    determine that are in attributes"""
    p = set()
    for lhs in powerset(attributes):
        lhs_c = closure(lhs, fds)
        for rhs in lhs_c:
            if rhs in attributes:
                p.add(lhs -> rhs)
    return p
```

```python
def implies(fds, fd):
    """Find if fd follows from fds"""
    lhs_c = closure(fd.lhs, fds)
    return all(attr in lhs_c for attr in fd.rhs)

def minimal_basis(fds):
    """Find a smallest, simplest set of
    fds equivalent to fds. Not unique."""
    basis = fds.copy()
    # Break up fds with 2+ attributes in RHS
    for fd in fds:
        basis.update({fd.lhs -> x for x in fd.rhs})
        basis.remove(fd)
    # Break up fds with 2+ attributes in LHS
    for fd in basis:
        split = False
        if len(fd.lhs) == 1:
            continue
        for attr in fd.lhs:
            lhs = fd.lhs.difference({attr})
            if implies(fds, lhs -> fd.rhs):
                basis.add(lhs -> fd)
                split = True
        if split:
            basis.remove(fd)
    # Remove redundant fds
    for fd in basis:
        test_fds = fds.copy()
        test_fds.remove({fd})
        if implies(test_fds, fd):
            basis.remove(fd)
    return basis
```

- The powerset of set S is the set of all subsets of S, of which there are $2^{|S|}$ combinations.
- Projection is very costly, $\mathcal{O}(2^n)$ iterations from the powerset!
    - *Tiny speed-up:* If `attribute in subset`, no need to add trivial `subset -> attribute`
    - *Tiny speed-up:* No need to iterate over empty set or set of all attributes
    - *Minor speed-up:* If `subset_closure == attributes`, we can ignore all supersets of `subset`.

eg. Relation $\mathrm{R}(\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E})$ with functional dependencies fds $= \{\mathrm{A} \to \mathrm{BC}, \mathrm{C} \to \mathrm{E}, \mathrm{D} \to \mathrm{E}, \mathrm{BE} \to \mathrm{A}\}$

Here are closures:

$\mathrm{A}^+ = \mathrm{ABCE}$
$\mathrm{C}^+ = \mathrm{CE}$
$\mathrm{BDE}^+ = \mathrm{ABCDE}$
$\mathrm{CD}^+ = \mathrm{CDE}$

Here are (simplified) projections of fds:

To ABCD: $\{\mathrm{A} \to \mathrm{BC}, \mathrm{BC} \to \mathrm{A}, \mathrm{BD} \to \mathrm{A}\}$
To BCDE: $\{\mathrm{C} \to \mathrm{E}, \mathrm{D} \to \mathrm{E}, \mathrm{BE} \to \mathrm{C}\}$
To ACD: $\{\mathrm{A} \to \mathrm{C}\}$

Running the minimal basis algorithm,

1st loop – replace $\mathrm{A} \to \mathrm{BC}$ with:
$\{\mathrm{A} \to \mathrm{B}, \mathrm{A} \to \mathrm{C}\}$
2nd loop – add no dependencies.
3rd loop – nothing happens.

Relational Algebra

Relational Algebra (RA): Generalization of basic algebra to relations.

- Assumes relations are sets (no duplicates) and all cells have values. This is not true in SQL.
- Problems have multiple solutions – some are more efficient *(DBMS implementations care about this, RA doesn't)*

Unary Operators

Selection ($\sigma_c(\mathrm{R})$): Keep tuples/rows of relation R where logical property c holds

- c supports booleans & comparisons (ie. $\geq, \leq, =, \neq$) on constants or attributes of R

Projection ($\pi_A(\mathrm{R})$): Keep attributes/columns A from relation R. Eliminates any copies.

- $\mathrm{R}[\mathrm{A}_1, \ldots, \mathrm{A}_n]$ is $\pi_{\mathrm{A}_1,\ldots,\mathrm{A}_n}(\mathrm{R})$ in relational algebra notation

Rename ($\rho_{\mathrm{R}'}(\mathrm{R})$): Temporarily rename relation R to R′

- $\rho_{\mathrm{R}'(\mathrm{A}_1,\ldots,\mathrm{A}_n)}(\mathrm{R})$ renames R to R′ and renames R 's attributes to $\mathrm{A}_1, \ldots, \mathrm{A}_n$

Assignment ($\mathrm{R} := \mathrm{E}, \mathrm{R} \leftarrow \mathrm{E}$): Defining expression E as a temporary table R *(the name R can't already exist globally)*

- $\mathrm{R}(\mathrm{A}_1, \ldots, \mathrm{A}_n) := \mathrm{E}$ is equivalent to $\mathrm{R} := \rho_{\mathrm{R}(\mathrm{A}_1,\ldots,\mathrm{A}_n)}(\mathrm{E})$
- Even if not renaming $\mathrm{A}_1, \ldots, \mathrm{A}_n$, it is still good practice to write $\mathrm{R}(\mathrm{A}_1, \ldots, \mathrm{A}_n) := \mathrm{E}$

Binary Operators

Union ($\mathrm{R} \cup \mathrm{S}$): On relations R, S with matching attributes, tuples in any of R and S

Intersection ($\mathrm{R} \cap \mathrm{S}$): On relations R, S with matching attributes, tuples in all of R and S

Difference ($\mathrm{R} - \mathrm{S}$): On relations R, S with matching attributes, tuples in R and not S

Cartesian Product ($\mathrm{R} \times \mathrm{S}$): Combine every tuple in R with every tuple in S

- **Theta Join ($\mathrm{R} \bowtie_c \mathrm{S}$):** A shortcut of $\sigma_c(\mathrm{R} \times \mathrm{S})$ *("theta" is just a historical artifact. "⋈" is called a bowtie)*
- **Self-Join:** A relation multiplied by itself
- Differentiate common attributes in R and S like so: R. a and S. a
- Often introduces nonsense tuples for common attributes, which can be eliminated with selects. This is so common an issue, the **natural join** operation is invented

Natural Join ($\mathrm{R} \bowtie \mathrm{S}$): Find all tuples in $\mathrm{R} \times \mathrm{S}$ that are equal in all common attributes of R and S.

- *Commutative:* $\mathrm{R} \bowtie \mathrm{S} = \mathrm{S} \bowtie \mathrm{R}$
- *Associative:* $(\mathrm{R} \bowtie \mathrm{S}) \bowtie \mathrm{T} = \mathrm{R} \bowtie (\mathrm{S} \bowtie \mathrm{T})$
- If R, S share attributes $\mathrm{a}_1, \ldots, \mathrm{a}_n$,
  - $\mathrm{R} \bowtie \mathrm{S} = \pi_{\mathrm{R.a}_1,\ldots,\mathrm{R.a}_n} \sigma_{(\mathrm{R.a}_1=\mathrm{S.a}_1)\wedge\cdots\wedge(\mathrm{R.a}_n=\mathrm{S.a}_n)}(\mathrm{R} \times \mathrm{S})$
  - $\mathrm{R} \bowtie \mathrm{S} = \pi_{\mathrm{R.a}_1,\ldots,\mathrm{R.a}_n}(\mathrm{R} \bowtie_{(\mathrm{R.a}_1=\mathrm{S.a}_1)\wedge\cdots\wedge(\mathrm{R.a}_n=\mathrm{S.a}_n)} \mathrm{S})$
- If R, S share all attributes, $\mathrm{R} \bowtie \mathrm{S} = \mathrm{R} \cap \mathrm{S}$
- If R, S share no attributes, $\mathrm{R} \bowtie \mathrm{S} = \mathrm{R} \times \mathrm{S}$
- If R, S share some attributes, no shared attribute values match, $\mathrm{R} \bowtie \mathrm{S} = \emptyset$
- **Dangling Tuple:** Tuples in R, S whose shared attribute values doesn't match anything in the other relation

Constraints: A statement of form $\mathrm{R} = \emptyset$, where R is built with relation algebra

- Note that $\mathrm{R} \subseteq \mathrm{S} \Leftrightarrow \mathrm{R} - \mathrm{S} = \emptyset$

PostgreSQL-Specific (allows NULL)

Inner Join: A natural join that removes dangling tuples

Outer Join: A natural join that keeps dangling tuples, creating NULL values

- **Left (Outer) (Natural) Join:** Preserves dangling tuples from only R
- **Right (Outer) (Natural) Join:** Preserves dangling tuples from only S
- **Full (Outer) (Natural) Join:** Preserves dangling tuples from both R, S

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL JOIN S

A	B	C
1	2	3

R NATURAL FULL JOIN S

A	B	C
1	2	3
4	5	NULL
NULL	6	7

R NATURAL LEFT JOIN S

A	B	C
1	2	3
4	5	NULL

R NATURAL RIGHT JOIN S

A	B	C
1	2	3
NULL	6	7

Relation	Description	Query
$R(X)$	Max of X	$R - \underbrace{\pi_{T1.X}\sigma_{T1.X<T2.X}(\rho_{T1}R \times \rho_{T2}R)}_{\text{tuples that are not the maximum of X}}$
$R(X)$	Min of X	$R - \underbrace{\pi_{T1.X}\sigma_{T1.X>T2.X}(\rho_{T1}R \times \rho_{T2}R)}_{\text{tuples that are not the minimum of X}}$
$R(X, X')$	X/X pairs with duplicates removed	$\sigma_{X<X'}R$ *Impossible if X does not support inequality operations.*
$R(X, Y)$	X values with ≥ 2 unique Y values	$\pi_{T1.X}\sigma_{\underbrace{(T1.X=T2.X)}_{\text{same X}}\wedge\underbrace{(T1.Y\neq T2.Y)}_{\text{unique Y}}}\underbrace{(\rho_{T1}R \times \rho_{T2}R)}_{\text{pairs}}$
$R(X, Y)$	X values with ≥ 3 unique Y values	$\pi_{T1.X}\sigma_{\underbrace{(T1.X=T2.X=T3.X)}_{\text{same X}}\wedge\underbrace{(T1.Y\neq T2.Y)\wedge(T1.Y\neq T3.Y)\wedge(T2.Y\neq T3.Y)}_{\text{unique Y}}}\underbrace{(\rho_{T1}R \times \rho_{T2}R \times \rho_{T3}R)}_{\text{triples}}$
$R(X, Y)$	X values with k unique Y values	$\text{AtLeastkUnique} - \text{AtLeastkUnique}$
$R(X, Y)$	X pairs where ≥ 2 Y values match *(includes duplicate pairs)*	$\pi_{T1.X,T2.X}\sigma_{\underbrace{(T1.X\neq T2.X)}_{\text{unique X}}\wedge\underbrace{(T1.Y=T2.Y)}_{\text{same Y}}}\underbrace{(\rho_{T1}R \times \rho_{T2}R)}_{\text{pairs}}$
$R(X, Y)$	X pairs where all Y values match *(includes duplicate pairs)*	$\text{Pairs}(X(1), Y(1), X(2), Y(2)) := R \times R$ $\text{PairsFlipped}(X(1), Y(1), X(2), Y(2)) := \pi_{X(1),Y(2),X(2),Y(1)}\text{Pairs}$ $\pi_{X(1),X(2)}\text{Pairs} - \underbrace{\pi_{X(1),X(2)}(\text{Pairs} - \text{PairsFlipped})}_{\text{pairs whose Y values do not all match}}$
$R(X, Y)$	X values with all Y values	$R - \underbrace{((\pi_X R \times \pi_Y R) - R)}_{\text{X values without all Y values}}$

*Aggregation (counting number of rows, averages, sums) is not supported in this version of RA.

Database Design Theory

Redundancy: When an attribute contains duplicate/redundant data. Causes anomalies

- **Update Anomaly:** Inconsistent data when a cell is updated without updating duplicates of the cell
- **Delete Anomaly:** Loss of data after a tuple deletion because the data is not stored in a separate table
- Occurs if a functional dependency is not isolated into its own relation
- Decompose redundant $R(A_1, \dots, A_n)$ into smaller relations $R_1(B_1, \dots, B_{n_1}), R_2(C_1, \dots, C_{n_2})$
    - $\{A_1, \dots, A_n\} = \{B_1, \dots, B_{n_1}\} \cup \{C_1, \dots, C_{n_2}\}$
    - $R = R_1 \bowtie R_2$

Synthesis: The creation of a "good" relation from scratch

Decomposition: The splitting of a pre-existing "bad" relation into "smaller" better relations such that:

- *No anomalies*
- *Lossless join* – projecting R results in its decomposition $R_1, \dots, R_n$; doing $R_1 \bowtie \dots \bowtie R_n$ returns exactly R
- *Dependency preservation* – preserves functional dependencies

Lossy Join: When $R \subseteq R_1 \bowtie \dots \bowtie R_n$ *(always true)* but not $R_1 \bowtie \dots \bowtie R_n \subseteq R$ *(ie. natural joins add spurious tuples)*

Chase Test: Algorithm to determine for no lossy joins given subrelations $R_1, \dots, R_k$ of $R(A_1, \dots, A_n)$. Essentially, build a specific set of tuples in R where $(t_1, \dots, t_n) \in R_1 \bowtie \dots \bowtie R_k$, and test if $(t_1, \dots, t_n) \in R$ necessarily follows.

eg. Chase Test on relation $R(A, B, C, D)$ with decomposition $R_1(A, B), R_2(B, C), R_3(C, D)$ and functional dependencies $\{C \to D, B \to A\}$

A	B	C	D
t_1	t_2	…	…
…	t_2	t_3	…
…	…	t_3	t_4

Use the table on the left, the simplest set of tuples in R that, decomposed to R_1, R_2, R_3, results in (t_1, t_2, t_3, t_4) after $R_1 \bowtie R_2 \bowtie R_3$. Each subrelation gets its own tuple.

Applying fds, the second tuple must be (t_1, t_2, t_3, t_4), so the left table is valid instance of R and $(t_1, t_2, t_3, t_4) \in R$ necessarily follows. Chase Test succeeds, no lossy joins.

Normalization: Converting a relation to a "normal form", a structure that guarantees good properties

Boyce-Codd Normal Form (BCNF): Relational form where $\forall$ nontrivial FDs $X \to Y$ holding in R, X is superkey

- "Attributes that functionally determine everything can functionally determine anything"

```
def bcnf_decompose(R, fds):
    relations = []
    for fd in fds:
        if is_nontrivial(fd) and not is_superkey(R.attributes, fd.lhs):
            c = closure(fd.lhs, fds)
            d = R.attributes.difference(c.difference(fd.lhs))
            R1 = create_relation(c)
            R1_fds = project(fds, c)
            R2 = create_relation(d)
            R2_fds = project(fds, d)
            relations.extend(bcnf_decompose(R1, R1_fds) + bcnf_decompose(R2, R2_fds))
    if len(relations) == 0:
        relations.append(R)
    return relations
```

- Multiple possible results, depending on order of iterations
- Some ways for minor speedups:
    - Use closure test to find superkeys *(recall $A_1, \dots, A_k \to A_1, \dots, A_n$ means $A_1, \dots, A_k$ is superkey)*
    - Modify project – skip a functional dependency in loop if it being created in R1 violates BCNF
- Algorithm guarantees no anomalies, lossless join, but not dependency preservation

eg. BCNF decompose relation $R(A, B, C, D, E)$ with functional dependencies $\{AB \to DE, C \to E, D \to C, E \to A\}$

(1) Rule $E \to A$ is non-trivial, E is not a superkey of R The closure is $c = E^+ = AE$ So $d = ABCDE - (AE - E) = BCDE$ Split into $R_1(A, E), R_2(B, C, D, E)$ Recursive call to R_1: we have $R_1(A, E)$ and $\{E \to A\}$, which satisfies BCNF so we done! Recursive call to R_2: we have $R_2(B, C, D, E)$ and $\{C \to E, D \to C, EB \to D\}$ (simplified)	(2) Rule $C \to E$ is non-trivial, C is not superkey of R_2 The closure is $c = C^+ = CE$ So $d = BCDE - (CE - C) = BCD$ Split into $R_{2.1}(C, E), R_{2.2}(B, C, D)$ Recursive call to $R_{2.1}$: we have $R_{2.1}(C, E)$ and $\{C \to E\}$, which satisfies BCNF so we done! Recursive call to $R_{2.2}$: we have $R_{2.2}(B, C, D)$ and $\{D \to C, BC \to D\}$ (simplified)
(3) Rule $D \to C$ is non-trivial, D is not superkey of $R_{2.2}$ The closure is $c = D^+ = CD$ So $d = BCD - (CD - C) = BC$ Split into $R_{2.2.1}(C, D), R_{2.2.2}(B, C)$ with $\{D \to C\}$ and $\emptyset$, which satisfies BCNF so we done!	So the decomposition becomes $R_1(A, E), R_{2.1}(C, E), R_{2.2.1}(C, D), R_{2.2.2}(B, C)$ Unfortunately, $BC \to D$ is not preserved. The decomposition may change if we choose different rules to iterate from!

3rd Normal Form (3NF): Form where $\forall$ nontrivial FDs $X \to Y$ in R, X is superkey or Y is part of a key *("prime")*

```
def 3nf_synthesis(fds, attributes):
    relations = []
    basis = minimal_basis(fds)
    # Make each fd a relation
    for fd in basis:
        r = create_relation(fd.lhs.union(fd.rhs))
        relations.append(r)
    # If no relation contains a superkey for all attributes, create a relation containing it
    if not any(is_superkey(attributes, relation.attributes) for relation in relations):
        r = create_relation(get_superkey(fds, attributes))
        relations.append(r)
    return relations
```

- Algorithm guarantees lossless join, dependency preservation, but not no anomalies
- A relation is "good enough" if it satisfies 2 of 3 conditions

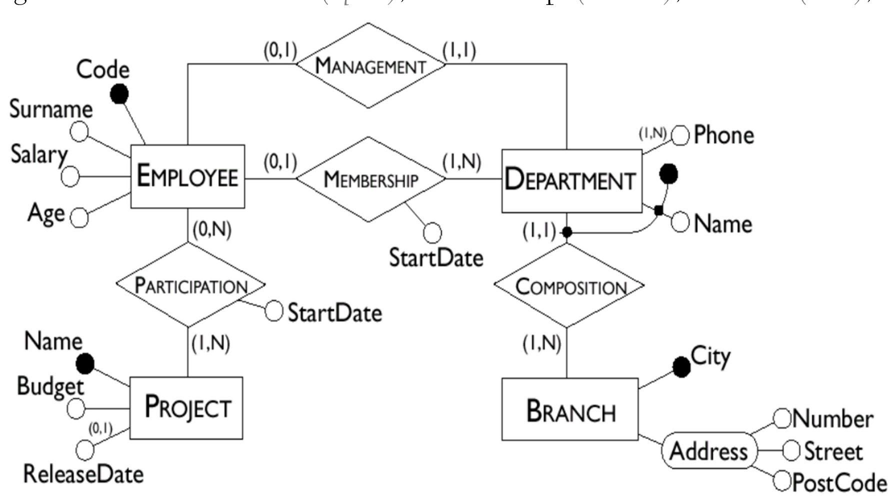
eg. 3NF on $R(A, B, C, D)$ with functional dependencies $\{A \to BC\}$

Since $\{A \to BC\}$ is the minimum basis, create $R_1(A, B, C)$
R_1 does not contain a superkey for $\{A, B, C, D\}$
Create table $R_2(A, D)$ since AD forms a superkey for $\{A, B, C, D\}$

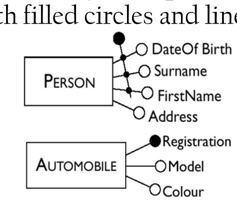
Modelling: The mapping of real-world entities/relationships to a database schema
Entity-Relation (E/R) Model: Visual, diagram-based data model to quickly "chart out" a database design

- **Entity Set:** A category of objects with common properties and exist independently *(eg. city, country)*
  - **Weak Entity Set:** An entity set with a foreign key
- **Entity:** An instance of the entity set *(eg. Dzhezkazgan is a city, Oman is a country)*
- **n-ary Relationship Set:** An association between n entity sets. Can be recursive – relate entity set to itself – or asymmetric – we specify roles played by entities in the relationship. *(eg. CityToCountry)*.
  - Keys are the keys of the n entity sets
- **n-ary Relationship:** An instance of a n-ary relationship set *(eg. ("Lusaka", "Angola"))*
- **Attribute:** Properties of entities/relationships – **composite attributes** are groups of attributes
- **Cardinality:** A tuple (min, max) between entities/relationships or attributes/entries specifying the min/max possible instances that any one entity/attribute can have in the relationship/entity.
  - Attributes are (1,1), or **single-valued** by default, but can be null or **multi-valued**.
  - Attributes in keys must be (1,1)
  - Foreign keys must come from an entity/relationship edge with (1,1) cardinality
  - Minimum values of 0/1+ mean optional/mandatory participation
  - Maximum values of 1/2+ mean ≤ 1/multiple possible participations. Use n to indicate no max limit
  - **Multiplicity:** Of relationship R where participating entities E_1, E_2 have cardinalities $(n_1, N_1), (n_2, N_2)$, the value N_1-to-N_2 AKA N_2-to-N_1.

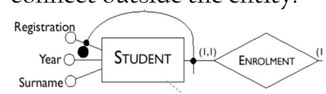
eg. E/R model with 4 entities *(square)*, 4 relationships *(diamond)*, attributes *(circle)*, a composite attribute *(rounded rect)*



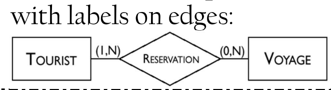
Internal Keys are specified with filled circles and lines:



Foreign Keys are similar, but connect outside the entity.



Cardinalities are specified with labels on edges:



Structured Query Language (SQL)

Structured Query Language: High-level formal programming language for database queries and schemas

- Data independence – details of data storage have no effect on queries
- Can focus on readability because DBMS optimizes query for efficiency
- **Data Definition Language (DDL):** Defining schemas
- **Data Manipulation Language (DML):** Writing queries, modifying the database

PostgreSQL (PSQL): Open-source bag-based DBMS. Below is syntax for queries *(SQL syntax may differ across DBMSs)*

- Statements ignore multiple whitespaces and end in `;`
- Comments start with `--`
- Strings have single quotes, `''`
- Logical statements have Python syntax, except `=` means equal, `<>` and `!=` both mean not equal.
- Chained comparisons don't always work – use AND and OR instead
- Keywords are optionally lowercase/uppercase.

Query Keyword	Description
`SELECT a, …`	Project attributes *(keeping duplicate rows)*; $\pi_{a,\dots}(R)$ Allows math/string operations on attributes (*eg.* `SELECT a + 10 FROM grades`)
`SELECT *`	Project all attributes *(keeping duplicate rows)*
`SELECT a1 AS a2, …`	Project attributes *(keeping duplicate rows)* and rename; $\rho_{R(a_2,\dots)}(\pi_{a_1,\dots}(R))$. Allows constant values (*eg.* `SELECT 'yes' AS attr`)
`SELECT DISTINCT a, …`	Project attributes *(removing duplicate rows)*; $\pi_{a,\dots}(R)$. Slower operation.
`FROM R, …`	The relation(s) to query from. If multiple, take Cartesian product: $R \times S \times \dots$ (*same as* `FROM R CROSS JOIN S`)
`FROM R1 R2, S1 S2, …`	Same as above, also renames relations; $R_1 \leftarrow R_2, S_1 \leftarrow S_2$
`FROM R (p) JOIN S ON c`	Theta-join; $R \bowtie_c S$; $\sigma_c(R \times S)$; or theta outer join (if `p` specified) `p` is optional and one of `LEFT, RIGHT, FULL`
`FROM R NATURAL (p) JOIN S`	Natural join; or outer join (if p is specified). In practice, a code smell; it doesn't communicate the purpose of query to coders as clearly, and queries easily break if the schema changes. Natural joins can be chained.
`FROM R JOIN S USING (a, …)`	Natural join, but joined attributes a, … are manually specified. Mandatory ().
`WHERE c`	Select rows based on condition c; $\sigma_c(\dots)$.
`ORDER BY a, … (DESC)`	Sort/order rows by attributes, ascending order default. Allows expressions.
`GROUP BY a, …`	Merges/groups rows with identical values in attributes a, … You must apply aggregating functions *(min, max, avg, count)* to unmerged columns: *eg.* `SELECT a, max(b) FROM R GROUP BY a` You can apply aggregating functions without `GROUP BY`; it aggregates the table into one row (*eg.* `count(*)` is # rows). You can add `DISTINCT` in all aggregating functions (*eg.* `count(DISTINCT *)` is # unique rows) Then the other attributes must also be aggregated to 1 row. <u>sum, count, max, min</u> Aggregating functions only return 1 row *(eg. even if two rows have the same max)* Aggregrate functions are not allowed in WHERE.
`HAVING c`	Select groups based on condition c. You must have applied `GROUP BY`. Conditions with aggregating functions belong in `HAVING`, not `WHERE`
`Q (E)`	**Subquery:** A query in a query. Must have () around it. Can substitute relations, but then must be given a name `E` Can be used in math/logical expressions *(but the subquery must return exactly 1 value)*
`Q1 UNION (ALL) Q2`	$Q_1 \cup Q_2$. `ALL` treats Q_1, Q_2 as multisets. $\{1,1,1,3\} \cup \{1,1,4\} = \{1,1,1,1,1,3,4\}$
`Q1 INTERSECT (ALL) Q2`	$Q_1 \cap Q_2$. `ALL` treats Q_1, Q_2 as multisets. $\{1,1,1,3\} \cap \{1,1,4\} = \{1,1\}$
`Q1 EXCEPT (ALL) Q2`	$Q_1 \setminus Q_2$. `ALL` treats Q_1, Q_2 as multisets. $\{1,1,1,3\} \setminus \{1,1,4\} = \{1,3\}$

Uncorrelated Subquery: A subquery referencing nothing from outer scopes. For ambiguous names, choose inner scope.
Correlated Subquery: A subquery referencing something from outer scopes. The subquery will be iteratively executed for every tuple in the outer scope.

Order of Execution:

1) `FROM` Retrieve relations
2) `WHERE` Keep certain rows
3) `GROUP BY` Aggregate rows to groups
4) `HAVING` Keep certain groups
5) `SELECT` Keep certain columns
6) `ORDER BY` Order rows

Conditional Operation	Description
`a LIKE 's'` `a NOT LIKE 's'`	If attribute a matches string pattern s, where: • `%x%` means 0 or more instances of `x` • `_` means any character
`a ~ 's'`	If attribute a matches string regex s. Probably slower than `LIKE`.
`a ★ SOME Q` `a ★ ANY Q`	If any(a ⋆ q for q in Q), where ⋆ is a logical operator, Q is a subquery.
`a ★ ALL Q`	If all(a ⋆ q for q in Q), where ⋆ is a logical operator, Q is a subquery.
`a IN Q` `a NOT IN Q`	Equivalent to `a = SOME (Q)` Equivalent to `a != ALL (Q)`
`EXISTS Q`	If the subquery is non-empty. Fast.
`a IS NULL` `a IS NOT NULL`	If a row contains NULL in attribute a

- For multiple attributes, write tuples like `(a, b, c, …) IN (…)`

String Operation	Description
`a || b || …`	String concatenation. Converts non-strings to strings
`char_length(a)`	Length of string `a`.
`lower(a)`	Convert string `a` to lower-case.
`upper(a)`	Convert string `a` to upper-case.
`position(a in b)`	Index of string `a` in string `b`. Indices start at 1!
`substring(a from i for j)`	The substring of length `j` starting at index `i`.

Null: A special SQL character for unknown/missing info. Any comparisons create **unknown (U)** truth values.

<table>
<tr><th></th><th>some nulls in A</th><th>All nulls in A</th></tr>
<tr><td>min(A)</td><td rowspan="5">ignore the nulls</td><td rowspan="4">null</td></tr>
<tr><td>max(A)</td></tr>
<tr><td>sum(A)</td></tr>
<tr><td>avg(A)</td></tr>
<tr><td>count(A)</td><td>0</td></tr>
<tr><td>count(*)</td><td colspan="2">all tuples count</td></tr>
</table>

<table>
<tr><th>A</th><th>B</th><th>A and B</th><th>A or B</th></tr>
<tr><td>T</td><td>T</td><td>T</td><td>T</td></tr>
<tr><td colspan="2">TF or FT</td><td>F</td><td>T</td></tr>
<tr><td>F</td><td>F</td><td>F</td><td>F</td></tr>
<tr><td colspan="2">TU or UT</td><td>U</td><td>T</td></tr>
<tr><td colspan="2">FU or UF</td><td>F</td><td>U</td></tr>
<tr><td>U</td><td>U</td><td>U</td><td>U</td></tr>
</table>

A	not A
T	F
F	T
U	U

- In tertiary logic: `True = 1`, `Unknown = 0.5`, `False = 0`. Then `A and B = min(A, B)`, `A or B = max(A, B)`
- Filtering operations (`WHERE`, `NATURAL JOIN`) do not accept unknown, but `CHECK` (next section) does!
- Behaviour when comparing `NULL`s is DBMS-dependent!
- `NULL` values are considered unique – duplicate `(NULL, …, NULL)` tuples are allowed.
- Any string or number operations on NULL will return NULL.

Editing Keyword	Description
`SHOW SEARCH_PATH`	Shows a search path – a list of schemas the system looks in. By default, returns `"$user", public`. For duplicates, chooses the 1st-found item. *1st Term:* The default location in for creating objects – `"$user"` means "search in schemas with the same name as the user". *2nd Term:* The backup location if the first term finds nothing. Optional. "`public`" is the default outer-most schema. If your path is S, then relation S. R can be accessed with just R.
`SET SEARCH_PATH TO S, public;`	Changes the search path to only look at tables in S.
`CREATE DOMAIN a2 FROM a1 …`	Define built-in type a2 from type a1 (usually a generic type, `int`, `text`)
`DEFAULT x`	Sets x as the default value for built-in type a2
`CHECK c;`	Check restriction/logical statement c holds for all values in a2. Accepts unknown truth values!
`CREATE SCHEMA S;`	Create a schema S. They're like folders with all relations S. R1, S. R2, … that follow S. The default schema that all relations follow is public.
`DROP SCHEMA (IF EXISTS) S (p);`	Delete schema S. Analogous to deleting tables (below).
`CREATE VIEW V AS Q;`	Create **view** V (a mini-relation, like a variable) from query Q. **Virtual View:** A view stored as a query. More common. **Materialized View:** A view whose items are constructed and stored.
`CREATE TABLE R(a1 TEXT, a2 INT);`	Create relation R with attributes a1 (string) and a2 (int). Also supports dates (`DATE`), booleans (`BOOL`), floats (`FLOAT`). See this for more. If data type is unspecified, it will take the form of the first tuple you add.
`ALTER TABLE R …`	Alter a table by…
`ADD COLUMN a3 VARCHAR(3)`	Adding a column a3 that is a string of max length 3…
`DROP COLUMN a1`	Deleting the column a1
`DROP TABLE (IF EXISTS) R (p);`	Delete relation R. Raises error if it does not exist (specify `IF EXISTS` to ignore it) `p` is optional and one of `RESTRICT` (raise error if tables depend on R) or `CASCADE` (delete R and other tables that depend on R)
`R(a1 TEXT, a2 INT DEFAULT 69)`	Have a default value for a2 when creating relation R. If inserting a tuple with no a2 value, 69 is added. If there is no default value, NULL is added to unspecified tuple parts
`R(a TEXT PRIMARY KEY)` `R(…, PRIMARY KEY (a1, a2))`	**Primary Key:** In PSQL, a set of non-null attributes forming a key • Helps DBMS optimize for speed by searching on primary keys • Tables have 0 or 1 primary key Top is limited to 1-attribute keys. Bottom can do multi-attribute keys.
`R(a TEXT UNIQUE)` `R(…, UNIQUE (a1, a2))`	Forces all values in an attribute to be unique. Unique does not include `NULL`; ie. you can have copies of `(NULL, a2)`
`R(…, FOREIGN KEY (a1, …)` `REFERENCES R(b1, …))`	Attributes $a_1, \ldots$ are foreign keys referencing table R's attributes $b_1, \ldots$, which must be unique or form a primary key
`R(…, (a1, …) REFERENCES R(b1, …))`	Attributes $a_1, \ldots$ references relation R's attributes $b_1, \ldots$, which must be unique or form a primary key.
`R(a TEXT NOT NULL)`	Special constraint that an attribute has no `NULL` values.
`R(…, CHECK c)`	Apply row-based restriction c to attribute a. c can be a subquery. Restriction only checked during insertion/update/deletion; if restrictions involve multiple tables, errors may occur.
`R(a TEXT CHECK c)`	Apply column-based restriction c to attribute a. Same restriction-checking as above. In PSQL, c cannot be a subquery.
`R(…, CONSTRAINT C CHECK …)` `R(…, CONSTRAINT C FOREIGN KEY …)`	Give a name C to any sort of constraint.
`INSERT INTO R VALUES ('a', 5), …;`	Manually add tuples ('a', 5), … into relation R. Must have ().
`INSERT INTO R Q;`	Add the result of query Q into relation R.

`DELETE FROM R;`	Delete all tuples from relation R. Cannot manually delete a specific tuple.
`DELETE FROM R WHERE c;`	Delete all tuples from relation R based on condition c. Uses the same syntax as querying (“`SELECT * FROM R WHERE c;`”)
`UPDATE R SET a1 = …, … WHERE c;`	Update the values in attributes a1, … of the tuples from relation R where condition c holds.

Assertion: A cross-table constraint, written in standard SQL as `CREATE ASSERTION A CHECK c;`

- Not supported in most DBMSs, including PSQL, due to high computational expense, and difficulty of testing and maintenance

Trigger: A set of events that triggers a predefined response (in the form of an SQL function). Don't worry about it.

```
CREATE FUNCTION RecordWinner() RETURNS TRIGGER AS
$$
BEGIN
  IF NEW.grade >= 85 THEN
      INSERT INTO Winners VALUES (NEW.sid);
  END IF;
  RETURN NEW;
END;
$$
```

```
CREATE TRIGGER TookUpdate
BEFORE INSERT ON Took
FOR EACH ROW
EXECUTE PROCEDURE RecordWinnter();
```

```
AFTER DELETE ON Courses
INSERT INTO Winners VALUES (SID);
```

```
BEFORE UPDATE OF grade ON Took
INSERT INTO Winners VALUES (SID);
```

Reaction Policy: Like a trigger, but limited to attributes

- If we delete many rows and one violates a constraint, behaviour varies based on DBMS – some may make no changes, make all changes except bad ones, halt at the error, etc.
- Deletion occurs "all at once" – if deleting a tuple affects a condition for deleting another, and both are being deleted, both will be deleted.

Keyword	Description
`(… FOREIGN KEY (a) REFERENCES R(b) ON DELETE p …)`	Trigger to do p to a when an attribute is deleted in b. p can be… ➢ `RESTRICT`: Do not allow the deletion/update ➢ `CASCADE`: Do deletion/update in the referring tuple ➢ `SET NULL`: Set null the referring tuple
`… ON UPDATE p`	Same as above, but for when b is updated.
`… ON UPDATE ON DELETE p`	Does p if updating or deleting occurs

Privilege Operation	Description
`ALTER TABLE R1, R2, … OWNER TO u;`	Set owner of relations R1, R2, … to u. Only owners can edit/delete an object. By default, an object's creator is its owner.
`GRANT p1, p2, … ON R1, R2, … TO u;`	Give user u access to query keyword p in relations R1, R2, …. p includes `ALL`, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, and much more. For select/insert/update, you can specify columns like `SELECT(A, B)`. Default value insertion rules apply. Deletion alone only allows users to delete all tuples Update alone only allows users to update an attribute of every tuple Query-based updates/deletions require select permissions You can still perform joins on whole tables even if you don't have selecting permission for all columns.
`REVOKE p ON R1, R2, … FROM u;`	Revoke user u's access to query keyword p in relations R1, R2, ….

- You need permission to access every column your query refers to – even those in key constraints!
  - Specifically, if inserting/updating A, which references B, grant select/update to B
  - CHECK WHAT HAPPENS when changing B in this case!
  - CHECK cascade – does it affect the referree or the referrer

Database Transaction: A unit of work in a DBMS that changes a database

- **Atomicity:** Transactions should be "single units", either succeeding completely or failing completely
- **Consistency:** Transactions preserve constraints
- **Isolation:** Concurrency control – simultaneous transaction executions should behave sequentially
- **Durability:** Transactions should remain committed even in hardware/system failure

Uncommitted: A transaction, if its changes have not been saved to the database *(ie. it's still running, or an error)*
Committed: A transaction, if its changes have been saved to the database

`BEGIN TRANSACTION;`
`…`
`END;`
	Specify a set of statements to be executed together as a transaction.
	By default, each statement is its own transaction.

Dirty Read: Reading a tuple that's been modified by a concurrent uncommitted transaction.
Nonrepeatable Read: Reading a tuple twice, but a concurrent uncommitted transaction modifies it in between.
Phantom Read: Executing a query twice, but a concurrent uncommitted transaction modifies results in between.
Serialization Anomaly: Executing many transactions creates different results than executing any order of them.

Isolation Level: Four levels in the SQL standard that determine how "strictly" isolation is enforced. Stricter means less errors, but slower system resources and a higher chance of transactions blocking each other.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Serializable	✗	✗	✗	✗
Repeatable Read	✗	✗	✓ *(✗ in PSQL)*	✓
Read Committed	✗	✓	✓	✓
Read Uncommitted	✓ *(✗ in PSQL)*	✓	✓	✓

- Serializable and repeatable read levels may return errors for transactions due to concurrent updates
- PSQL's implementation is slightly different
  - **Read Committed:** Uses most-recently committed data. Default. Equivalent to Read Uncommitted in PSQL.
  - **Repeatable Read:** Uses the same version of the database (plus its own modifications).
  - **Serializable:** Commit if the transaction's uncommitted result would stay the same as if all of its simultaneous transactions were committed in any order.

eg. Table R(A) for boolean A, T1 swaps True/False values, T2 deletes all False tuples. Both occur concurrently.

Read Committed	*Repeatable Read*	*Serializable*
Case 1: T1 commits first T2 selects tuples to delete on version of R before T1 commits. After T1 commits, T2 rechecks its selected tuples and deletes nothing.	Case 1: T1 commits first T2 selects tuples to delete on version of R before T1 commits. T2 doesn't care T1 commits, deleting what is now all True tuples.	Case 1: T1 commits first T2 selects tuples to delete on version of R before T1 commits. After T1 commits, T2's selected tuples no longer match, so raise an error.
Case 2: T2 commits first T1 selects tuples to edit on version of R before T2 commits. After T2 commits, T1 rechecks its selected tuples, swaps still-existing tuples.	Case 1: T2 commits first T1 selects tuples to edit on version of R before T2 commits. After T2 commits, some of T1's selected rows might not exist; raise an error if this is the case.	Case 1: T2 commits first T1 selects tuples to edit on version of R before T2 commits. After T2 commits, some of T1's selected rows might not exist; raise an error if this is the case.

Embedded SQL

Embedded SQL: Combining of SQL with other language's libraries *(eg. C's SQL/CLI, Java's JDBC, Python's psycopg2)*

- Helps overcome the fact that unlike most languages, standard SQL is not **Turing-complete** – it cannot solve every computational problem, even with arbitrary time and memory – due to no loops and recursion
- Allows controlling output format
- Allows hiding SQL syntax from regular users
- As most languages have no "relation" data type, tuples into SQL one at a time

```
import psycopg2

# Open connection to existing database
conn = psycopg2.connect("dbname=csc343h-youruserid user=youruserid password=")

# Open a cursor, which performs all database operations
cur = conn.cursor()

# Execute SQL commands
cur.execute("SELECT name, age FROM CoolPeople;")

# Hide away SQL syntax from regular users using Python
name = input("Enter name")
cur.execute("SELECT name, age FROM CoolPeople WHERE name = %s", (name, ))

# Save your changes onto the real database (like git commit and push).
conn.commit()

# Rollback any uncommitted changes
# conn.revert()

# Close connection to database
cur.close()
conn.close()
```

String Injection: One of the most common web hacking techniques, done by hacking SQL statements

```
# Suppose our input is "Something'; DROP TABLE CoolPeople; --"
name = input("Enter name")

# Result: SELECT name, age FROM CoolPeople WHERE name = 'Something'; DROP TABLE CoolPeople; --'
cur.execute(f"SELECT name, age FROM CoolPeople WHERE name = '{name}'")

# Note that 'a''b' is the same as 'ab'
# Moral of the story – don't build statements from f-strings or string concatenation!
```

Accessing PSQL from the University

```
ssh utorid@dbsrv1.teach.cs.toronto.edu

psql csc343h-csteachinglabusername
\q – quit PSQL
\d table – describe a table
\i file – run file like an sql file
```

Your username is `csteacherlinglabusername`, or alternatively `CURRENT_USER`.