Redundancy: information stored repeated in the database, the root of several problems associated with relational schemas
-creates ANOMALIES (update, insertion, deletion anomalies)
-only one tuple is updated, data is inserted with other different info, deletion of important info
-Eliminate redundancy by decomposition
Functional dependency: how a set of attributes can determine another set of attributes X -> Y holds over relation scheme R if following holds:
-for any two records t, t' in R, if t[x] = t'[x], then t[y] = t'[y]
X-Y does not imply that Y->X
If (X, Y) -> Z, then X -> Z and Y -> Z is WRONG
K-> all attributes of R, K is a superkey for R
Given some FD, we can infer additional FDs
F+ = closure of F is the set of all FDs that are implied by F
Armstrong's Axioms (AA):

- **Reflexivity**: if X ⊆ Y, then Y -> X
- **Augmentation**: if X -> Y, then XZ -> YZ for any Z
- **Transitivity**: if X -> Y and Y -> Z, then X -> Z
- **Union**: if X -> Y and X -> Z then X -> YZ
- **Decomposition**: if X -> YZ, then X -> and X -> Z

Find the candidate key by checking whether X(superkey) is minimal
Search space for candidate keys: k attributes, 2^{k-1} attribute sets to check

<u>Efficient Solution</u>

L category: attributes that only appear at left side in FD -> each candidate key should include ALL attribute in L category
R category: only appear at right side in FD -> should NOT be included
M category: left and right side of FD -> may or may not b part of keys
Try L as keys, if it's not a key, combine some from M; if L is a key, that's the candidate key
Normal form: determine if any refinement is needed based on the type of normal form that a relation is in
-if a relation is in a certain normal form, problems are avoided/minimized, can help decide whether decomposition will be needed to eliminate data redundancy

<u>Normal Forms (NF)</u>

1st: 1NF if and only if the domains of all attributes of R are atomic, a domain is atomic if elements of the domain are considered to be indivisible units (i.e. CS 442 - Alan, CS442 - Betty, CS 442-Carol)
2nd: there is no partial dependency
partial dependency: an FD X -> Y is said to be a partial dependency if there exists an FD Z -> Y such that Z ⊆ X (Z is a subset of X)

<u>Violation of 2NF: check if exists a FD X -> Y such that :</u>

(1) X is a subset of candidate keys of R
(2) Y is a non-key attribute

3rd: no partial dependency, no transitive dependency, for each candidate key K of R, all non-key attributes of R are directly dependent on K
Transitive dependency: exist transitive dependency when a non-key attribute A determines another non-key attribute (K -> A -> B, k is the key)

<u>3NF Violations</u>

Case 1: X is a subset of a candidate key K
Case 2: X is not a subset of a candidate key and Y is a non-key attribute

<u>Shortcut Rules</u>

R1: if all candidate keys are singleton keys, then R must satisfy 2NF
R2:if all attributes are part of some candidate keys, R must be 2NF&3NF
Boyce-codd (BCNF or 3.5 NF): satisfies -> if for all X -> Y in F, it satisfies that X is a superkey for R (nothing but the key)

<u>3NF vs BCNF</u>

3NF requires that (1) either X is a candidate key for R, (2) OR Y is part of some candidate key
BCNF requires that (1) X is a superkey

Normal Form	Constraint
1NF	Atomic value
2NF	No partial dependency (i.e., there does not exist an FD X->A such that X is a subset of a candidate key and A is a non-key attr.)
3NF	No partial dependency & No transitive dependency (i.e., for each FD X->A, either X is a candidate key or A is a subset of a candidate key)
BCNF (3.5 NF)	All non-trivial FDs are key FDs (i.e., for each X->A, X is a superkey)

<u>Problems with Decompositions</u>

1. May be impossible to reconstruct the original relation
2. Dependency checking may require joins

Good decomposition: are lossless, dependency preserving

	BCNF Decomposition	3NF Decomposition
Data redundancy	NONE	May still have some
Lossless	Guaranteed	Guaranteed
dependency-preserving	Not guaranteed	Guaranteed

Lossy decomposition: join result of the tables after decomposition is NOT the same as the original dataset
Lossless decomposition: join result of the tables after decomposition is the same as the original dataset

<u>Example: Is it a lossless decomposition?</u>

The decomposition of R into X and Y is lossless with respect to F if and only if the F+ satisfies that: X ∩ Y -> X or X ∩ Y -> Y
i.e. the common attribute of X and Y (that X and Y natural joins on) is a superkey of either X or Y.
If W -> Z holds over R and W ∩ Z is empty, then (1) decompose R into 2 tables R1 = R-Z and R2 = WZ (2) decomposition (R-Z, WZ) are guaranteed to be lossless (as R-Z and WZ joins at W, and W -> Z)

<u>BCNF Decomposition</u>

Step 1: ensure all FDs in F contain only single attribute at right hand side
Step 2: identify all FDs F' ⊆ F that hold on R

- A FD f holds on R if R contains all attributes in f (superkey)

Check if R satisfies BCNF according to F'
If not, for any X -> Y in F' that violates BCNF, decompose R into R1 = R-Y and R2 = XY
Repeat Step 2 on R1 and R2, until all the decomposed tables satisfy BCNF
BCNF decomposition is guaranteed to be lossless
Dependency Preserving Decomposition: if R is decomposed into X and Y, then the projection of F on X (denoted Fx) is the set of FDs U -> V in F+ such that all the attributes U, V are in X or if

$(F_x \cup F_Y)+ = F+$

<u>3NF Decomposition</u>

<u>Minimal Cover:</u> G is the minimal cover of a set of FDs F if

1. Right hand side of each FD in G is a single attribute
2. F+ = G+
3. G is minimal; if we modify G by deleting an FD in G, G+ changes

<u>Finding Minimal Cover</u>

Step 1: Minimize right hand side of FDs so that they only contain single attributes; X -> YZ to be X ->Y and X ->Z
Step 2: Minimize left hand side; A -> B and ABX -> Z, replace ABX -> with AX -> Z
Step 3: remove redundant FDs; if X -> Y can be inferred from other FDs, remove X -> Y

<u>3NF Decomposition</u>

Step 1: Find the minimal cover F' of F
Step 2: Generate a BCNF decomposition of R
Step 3: Identify the dependencies N in F' that is not preserved by BCNF decomposition
Step 4: for each X -> Y in N, create a relation schema XY and add it to {R1...Rn}
-step 2 guarantees a lossless decomposition
-step 3 and 4 ensure it is a dependency preserving decomposition
<u>For any given R, there is always a 3NF decomposition</u>
-but may not have a BCNF decomposition,
-BCNF is stricter than 3NF
-BCNF decomposition is always a 3NF decomposition

Step 1: Find minimal cover F' of F

- F' = {A->D, C->A, C->B, E->B, E->D}

Step 2: create a lossless-join BCNF decomposition D of R

- Step 2.1. find candidate key(s): CE
- Step 2.2. construct BCNF decomposition based on <u>F</u>': D = {BC, CE, AC, AD} (

Step 3: Identify the dependencies N in <u>F</u>' that is not preserved by D

- N = {E->D, E->B}

Step 4: For each X-> Y in N, create a relation schema XY and add it to D

- D={BC, CE, AC, AD, ED, EB}

Query languages: allow manipulation and retrieval of data from a database

Unary Operators (only one table as input)

Selection (σ): pick rows for output, no duplicates

Predicates = $<, <=, =, \neq, >=, >, \wedge, \vee, \neg$

Projection (π): pick columns for output, can contain duplicate records

Selection is evaluated before projection

Set Operations: take 2 input relations, relations must be union-compatible

Union compatible = same schema, same # attributes, corresponding attributes have the same data type

Set-difference (-): R-S returns relation instance containing all tuples in R but not S, not symmetric, no duplicates

Union (U): returns a relation instance containing all tuples in either or both, symmetric, no duplicates

Non-set operation

Cross-product (X): attributes of R followed by the attributes of S, in order

If tables contain same attribute A, output schema includes R.A and S.A

Instance: cartesian product of R and S, pair each tuple of R with each tuple of S; S and R do not have to be compatible, symmetric, no duplicates

Intersection ($\cap$): outputs the tuples in both R and S, must be union compatible, not a basic operation $R \cap S = R-(R-S)$

Join ($\bowtie$): natural join, schema returns all attributes in R and S, no duplicates, output is all rows in R X S where they have equal values on the common attributes; if no common attributes, return the cartesian product (RXS)

Condition Join: $R \bowtie_c S$, C is the condition that the output must satisfy; schema same as cross-product; instances are only those records in RXS that satisfies condition C

Equi-join: special case of condition join where condition contains only equalities =, schema same as cross product

Division (/): A/B or A%B, express queries with keyword "ALL"

A/B output attributes of B is proper subset of attributes of A; relation returned by division operator will return the tuples from relation A which are associated to every B's tuple

Renaming operation (ρ): ρ(X,E), allows to name results as new instance, not a compound operation

Name of sailors who've reserved a red and a green boat

$\rho(ReserveRed, \sigma_{color='red'}(Boats) \bowtie Reserves \bowtie Sailors)$

$\rho(ReserveGreen, \sigma_{color='green'}(Boats) \bowtie Reserves \bowtie Sailors)$

$\pi_{sname}(ReserveRed \cap ReserveGreen)$

- **Find the names of sailors who have reserved at least two different boats**

S1: Sailors who have reserved at least ONE boat $\rightarrow$ $\rho(R, (\pi_{sid,sname,bid}(Reserves \bowtie Sailors)$

S2: Sailors who have reserved at least TWO boats $\rightarrow$ $\rho(RPairs, (1->sid1, 2->sname1, 3->bid1, 4->sid2, 5->sname2, 6->bid2), R \bowtie_{Sid1=Sid2} R)$

Renaming to eliminate duplicate attributes

$\pi_{sname1}(\sigma_{bid1 \neq bid2} RPairs))$

Final output: Same sailor but two different boats

Select = projection operator, can use * to get all attributes, duplicates are preserved by default, can include arithmetic expressions in SELECT

Can write SELECT DISTINCT to eliminate duplicates

WHERE: selection operator can use <, <=, = , <>, >=, >, AND, OR, NOT, BETWEEN, IN, LIKE, arithmetic operations, string operations ("||") for concatenation, Use single quotes (' ') around text values

WHERE prod_desc BETWEEN 'C' AND 'S';

WHERE catno NOT BETWEEN 200 AND 400;

WHERE memno IN (100, 200, 300, 400);

WHERE Date-returned IS NOT NULL; or NULL;

LIKE operator in where clause: used for string approximate matching

'_' stands for any character, '%' stands for any string with 0 or more characters

WHERE address LIKE '_T%';

AS and = are two ways to name new attributes defined in the output

SELECT 2*S.Salary AS DoubleSalary, TripleSalary =3*S.Salary

FROM clause: Cross-product (X) or Join($\bowtie$) of tables, distinguish attributes of the same name like Sailor.name

Without WHERE clause: cross-product(X);

With WHERE clause: checks equivalence on ALL common attributes (natural join), order of tables does not matter and non-joinable tables still can be put side-by-side in FROM class

Another way to natural join

SELECT A1, ..An

FROM R NATURAL JOIN S;

Order of tables matters and only join-able tables are put at both sides

Range Variables: can associate tables in the FROM clause, useful when ambiguity arises

FROM Sailors S, Reserves R

Union: excludes duplicate rows, two subqueries must have the same attributes in SELECT clause

Union of two SELECT * Clause: different non-key values are considered as different and add into the union result

INTERSECT, UNION: union compatible, must have SELECT-FROM

Subquery 1

INTERSECT/ EXCEPT=MINUS/UNION

Subquery 2

ID of Sailors who've reserved a red AND a green boat

```
SELECT R.sid
FROM Boats B1, Boats B2, Reserves R
WHERE R.bid = B1.bid
        AND R.bid = B2.bid
        AND B1.color = 'red'
        AND B2.color = 'green';
```

Name of Sailors who've reserved at least 2 different boats

```
SELECT S1.name
FROM Reserves R1, Reserves R2, Sailors S1, Sailors S2
WHERE R1.sid=R2.sid //same sailor
        AND R1.bid<>R2.bid //two different boats
        AND R1.sid.=S1.sid //natural join R1 and S1
        AND R2.sid=S2.sid //natural join R2 and S2
```

ID of sailors who've reserved a red boat but never reserved a green boat

```
SELECT sid
FROM Boats B NATURAL JOIN Reserves R
WHERE B.color='red'
EXCEPT
SELECT sid
FROM Boats B NATURAL JOIN Reserves R
WHERE B.color='green'
```

Subquery: (nested query) a query within another SQL query and embedded within the WHERE clause

Subqueries must be enclosed within parentheses

Subquery can have only 1 attribute in the SELECT clause, unless multiple attributes are in the main query for the subquery to compare its selected attributes

WHERE attribute_name [NOT] IN (subquery)

WHERE [NOT] EXISTS (subquery)

WHERE expression op [ANY|ALL] (subquery)

WHERE EXISTS -> returns true of the result of subquery is not empty; otherwise returns false

WHERE expression NOT Exists (subquery) -> negation

ANY operator: syntax = v op ANY S

V = single value, S = set of values, op = =, <>, >, <, etc

Find sailors who've reserved all boat/ find sailors for whom there is no such boat that he/she has not reserved

```
SELECT S.name
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
                  FROM Boats B
                  WHERE NOT EXISTS (SELECT R.bid
                                    FROM Reserves R
                                    WHERE R.bid = B.bid
                                          AND R.sid = S.sid));
```

SUM: returns the summation of all non-null values in a set

AVG: calculates the average of non NULL values in a set

MIN, MAX: min or max of non null value

COUNT: returns the # of rows in a group, including rows with NULL values

For aggregate queries without GROUP BY clause, DO NOT put non-aggregate attributes and aggregate functions together in SELECT clause

GROUP BY: "For each…", groups rows of the same values into groups, often used with aggregate functions to group the result-set by one or more columns

Non-aggregate attributes must appear in GROUP BY clause, always placed after WHERE or FROM clause (at the end)

Split-apply-combine strategy

Split phrase: divides the groups by the values on the grouping attributes

Apply: applies the aggregate function and generates a single value

Combine: for each group, combines its grouping value with aggregated results

HAVING clause: a conditional clause with GROUP BY clause

Filters groups based on GROUP BY results, requires a GROUP BY clause to be present; only columns that appear in GROUP BY clause can appear in HAVING

HAVING vs. WHERE: where clause filters the individual records tuple by tuple while having clause filters the whole group

Where is applied before group by while having is applied after

If selection condition is specified on the aggregate values, use HAVING clause

Otherwise, if the selection condition is specified on individual records, use WHERE clause

```
SELECT B.bid COUNT(*) AS rCount
FROM Boats B NATURAL JOIN Reserves R
GROUP BY B.bid, B.color
HAVING B.color = 'red';
```

```
SELECT DISTINCT S.name
FROM Student S
WHERE S.snum IN (SELECT E.snum
                 FROM Enrolled E
                 GROUP BY E.snum
                 HAVING COUNT (*) >= ALL
                        (SELECT COUNT (*)
                         FROM Enrolled E2
                         GROUP BY E2.snum))
```

Name of sailors who never reserved a boat

```
CREATE TABLE Temp_Sid AS
SELECT sid
FROM Sailors
EXCEPT
SELECT sid
FROM Sailors NATURAL JOIN RESERVES;

SELECT snake
FROM Temp_Sid NATURAL JOIN Sailors;
```

```
GROUP BY E.snum
 HAVING COUNT (*) >= ALL
(SELECT COUNT (*)
FROM Enrolled E2
GROUP BY E2.snum))
```