

DIAGRAM SYMBOLS

Rectangle=Entity(Strong) | **Double**
Rectangle=Weak Entity | **Diamond**=Relationship
Double Diamond=Identifying Relationship |
Oval=Attribute | **Double Oval**=Multivalued Attribute | **Dashed Oval**=Derived Attribute | **Underline**=Primary Key | **Double line**=Total Participation(all must participate) | **Single line**=Partial Participation(some may) | **1,N,M**=Cardinality Ratios

SERIALIZED SYMBOLS

d in circle=Disjoint subclasses(no overlap) | **o in circle**=Overlapping subclasses | **Double line**=Total specialization(all must be in subclass) | **Single line**=Partial specialization | **U**=Subset/union symbol |

Specialization=Top-down(Employee→Secretary) | **Generalization**=Bottom-up(Car, Truck→Vehicle) | **ISA Rule**=Only use if subclass IS A superclass. Ex: Laptop IS A Computer✓, Country IS A Continent X ER→RELATIONAL MAPPING (RULES)

1. Regular Entity: Create table with all simple attributes, choose PK. $T(attr_1, attr_2, \dots)$

2. Weak Entity: Include owner's PK as FK, combine with partial key as PK. $T(ownerPK^*, partialKey, attrs)$

3. Binary 1:1: Add FK to one table(prefer total participation side). $S(PK_S, attrs, T_PK^*, relAttrs)$

4. Binary 1:N: Add FK on N-side.

N-side($PK, attrs, onePK^*, relAttrs$)

5. Binary M:N: Create new table with both PKs as FK, combined as PK. $R(PK_1^*, PK_2^*, relAttrs)$

6. Multivalued Attribute: Create new table with attribute + owner's PK. $T(attr, ownerPK^*)$

7. N-ary Relationship(n>2): Create table with all PKs as FK, combined as PK. $R(PK_1^*, PK_2^*, PK_3^*, attrs)$

8. Super/Sub(EER): **8A(Any)**: Super($k, Atts$) + Sub($k^*, Atts$). **8B(Total)**: Sub($k, AllAtts$). **8C(Disj)**: 1Tbl($k, All, Type$). **8D(Over)**: 1Tbl($k, All, Flags$).

RELATIONAL MODEL TERMINOLOGY

Relation=Table | **Tuple**=Row/Record |

Attribute=Column/Field | **Domain**=Valid values for attribute | **Cardinality**=# of tuples |

Degree=# of attributes | **Relation**

Schema= $R(A_1, A_2, \dots, A_n)$ where R=relation name, A=attributes | **Relation State**= $r = \{t_1, t_2, \dots, t_m\}$ where each tuple $t = \langle v_1, v_2, \dots, v_n \rangle$

PROPERTIES OF RELATIONS

Each relation name is **unique** | Each cell contains **atomic value**(1NF) | Attribute names **unique** within relation | Attribute values from **same domain** | Order of attributes has **no significance** | Each tuple is **distinct**(no duplicates) | Order of tuples has **no significance**

KEYS

Superkey=Any set of attributes ensuring uniqueness | **Candidate Key**=Minimal superkey(no proper subset is superkey) | **Primary Key**=Chosen candidate key(NOT NULL, UNIQUE), underline it | **Foreign Key**=Attribute(s) referencing another relation's PK

Composite Key=Multiple attributes combined as key | **Prime Attribute**=Member of any candidate key | **Non-prime Attribute**=Not member of any candidate key

CONSTRAINTS

1. Domain Constraints: Valid data types and value ranges

2. Key Constraints: Primary Key: UNIQUE, NOT NULL. Uniqueness enforced

3. Referential Integrity: FK must reference existing PK or be NULL. **Actions on DELETE/UPDATE**: RESTRICT/NO ACTION=Prevent operation | CASCADE=Propagate change to dependent rows | SET NULL=Set FK to NULL | SET DEFAULT=Set FK to default value

4. Semantic Integrity: Business rules(CHECK constraints, triggers)

RELATIONAL ALGEBRA OPERATIONS

Unary: Selection $\sigma_{condition}(R)$ =Select rows satisfying condition | **Projection**

$\pi_{attr_1, attr_2}(R)$ =Select columns, no duplicates |

Rename $\rho_{new}(old)$ =Rename relation/attributes

Binary: Union $R \cup S$ =All tuples(union compatible required) | Difference $R - S$ =In R but not S |

Intersection $R \cap S$ =In both R and S | **Cartesian Product** $R \times S$ =All combinations, creates $|R| \times |S|$ tuples | **Natural Join** $R \bowtie S$ =Join on common attributes(show once) | **Theta Join** $R \bowtie_\theta S$ =Join with condition | **Division** $R \div S$ =R tuples matching ALL S tuples

Join Attributes(R has m, S has n, k common): Equi Join ($m + n$ attrs): Keeps all columns (duplicates included). **Natural Join** ($m + n - k$ attrs): Removes duplicate common columns. **Outer Joins**: Left $R \bowtie L S$ | Right $R \bowtie R S$ | Full $R \bowtie F S$ =All from both

Key Equivalences: $R \bowtie_\theta S = \sigma_\theta(R \times S)$ |

$R \cap S = (R \cup S) - ((R - S) \cup (S - R))$ |

$R \cap S = R - (R - S)$

TUPLE RELATIONAL CALCULUS

General Form: $\{t | COND(t)\}$ or $\{t, A_1, t, A_2 | COND(t)\}$ where t=tuple variable, COND=Boolean expression

Quantifiers: \exists (EXISTS)=At least one tuple satisfies | \forall (FOR ALL)=Every tuple must satisfy | \neg (NOT)=Negation

Division Template (Find X who did ALL Y): $\{t | X(t) \wedge (\forall y)(\neg Y(y) \vee (\exists r)(R(r) \wedge r.x = t.x \wedge r.y = y))\}$

English: "Find X where for every Y, there is a relationship R linking them."

Join Rule: Use \exists if NOT displaying tuple's attribute but need it from another table for join

Original Example:

{e.Fname, e.Lname}|EMPLOYEE(e) \wedge $(\exists d)(DEPT(d) \wedge d.Dname = 'Research' \wedge e.Dno = d.Dno)$

DOMAIN RELATIONAL CALCULUS

Form: $\{x_1, x_2, \dots, x_n | COND(x_1, x_2, \dots)\}$

Ex: $\{u, v | (\exists q, r, s)(EMPLOYEE(qrstuvwxyz) \wedge q = 'Jon' \wedge r = 'R.' \wedge s = 'Mortensen') \wedge u, v, w, x, y, z\}$

Alt: $\{u, v | EMPLOYEE('Jon', 'R.', 'Mortensen', t, u, v, w, x, y, z)\}$

SQL - DATA DEFINITION LANGUAGE(DDL)

Database Operations: CREATE DATABASE dbname;

DROP DATABASE dbname; | USE dbname; | SHOW DATABASES;

Table Operations: CREATE TABLE t(id INT PRIMARY KEY, name VARCHAR(50) NOT NULL, fk_id INT, FOREIGN KEY(fk_id) REFERENCES other(id) ON DELETE CASCADE ON UPDATE SET NULL); | ALTER TABLE t ADD COLUMN col TYPE; | ALTER TABLE t DROP COLUMN col; | ALTER TABLE t MODIFY COLUMN col NEWTYPE; | DROP TABLE t; | SHOW TABLES; | DESCRIBE t; | SHOW COLUMNS IN t; | Index Operations: CREATE INDEX idx ON table(column); | DROP INDEX idx ON table; | SHOW INDEX FROM table; | SQL - DATA MANIPULATION LANGUAGE(DML)

Execution Order: FROM → JOIN → WHERE → GROUP → HAVING → SELECT → DISTINCT → ORDER → LIMIT

Allowed Conditions: $= < > < > \leq \geq = \neq \text{AND OR NOT} \text{ AND OR NOT} \text{ BETWEEN } x \text{ AND } y \text{ IN } (a, b) \text{ IS NULL} \text{ LIKE } 'pat'$

Insert: INSERT INTO t VALUES(v1, v2, v3); | INSERT INTO t(col1, col2) VALUES(v1, v2);

Update: UPDATE t SET col=val WHERE condition; | UPDATE t SET col1=v1, col2=v2 WHERE cond;

Delete: DELETE FROM t WHERE condition;

Select Basic Syntax: SELECT [DISTINCT] cols [*|AGG(col)] FROM table1 [JOIN table2 ON condition] [WHERE condition] [GROUP BY cols [ASC|DESC]] [LIMIT n];

Aliases: Column: SELECT col AS "Name" | Table: FROM table AS t. Required for derived tables/self-joins.

Joins in SQL: SELECT * FROM t1 INNER JOIN t2 ON t1.id=t2.id; | SELECT * FROM t1, t2 WHERE t1.id=t2.id; | SELECT * FROM t1 LEFT JOIN t2 ON t1.id=t2.id; | RIGHT JOIN | FULL JOIN | NATURAL JOIN

Aggregate Functions: COUNT(*), COUNT(col), COUNT(DISTINCT col), SUM(col), AVG(col), MIN(col), MAX(col)

Grouping: SELECT dept, COUNT(*), AVG(salary) FROM employee GROUP BY dept HAVING AVG(salary)>50000;

Note: WHERE filters rows before grouping, HAVING filters groups

Subqueries: WHERE col IN(SELECT col FROM t2) | WHERE EXISTS(SELECT * FROM t2 WHERE cond) | WHERE col>ANY(SELECT col FROM t2) | WHERE col>ALL(SELECT col FROM t2) | WHERE col=(SELECT MAX(col) FROM t2)

Subquery Note: The (SELECT...) inside IN/NOT IN generates a temporary **Derived Table** (Result Set). **Constraint**: It must return exactly one column.

Pattern Matching: WHERE name LIKE '%A%'=Starts with A | WHERE name LIKE '%son%'=Ends with son | WHERE name LIKE '%om%'=Contains om | WHERE name LIKE 'A_...'=A + 1 character

Views: CREATE VIEW vname AS SELECT...; | DROP VIEW vname; | SELECT * FROM vname;

QUERY INTERPRETATION

NOT EXISTS usually means "Division" (Students who have taken ALL courses).

GROUP BY x HAVING count(*) > 1 usually means "Find duplicates" or "Find x with multiple y".

LEFT JOIN... WHERE is NULL means "Find X with NO Y" (e.g., Students with no classes).

SQL - ADVANCED FEATURES

Triggers: DELIMITER // | CREATE TRIGGER tname {BEFORE|AFTER} {INSERT|UPDATE|DELETE} ON table FOR EACH ROW BEGIN | -- NEW.col(INSERT/UPDATE), OLD.col(DELETE/UPDATE) | IF NEW.amount<0 THEN SET NEW.amount=0; END IF; | END// | DELIMITER ; | DROP TRIGGER tname; | Note: Cannot have multiple triggers for same event+action time on table

Stored Procedures: DELIMITER // | CREATE PROCEDURE pname(IN p1 INT, OUT p2 INT) BEGIN |

SELECT col1 INTO p2 FROM t WHERE id=p1; | END// | DELIMITER ; | CALL pname(10, @result); | SELECT @result;

System Catalog: USE information_schema; | SELECT * FROM SCHEMATA;=databases | SELECT * FROM TABLES WHERE TABLE_SCHEMA='db'; | SELECT * FROM COLUMNS WHERE TABLE_NAME='t';

PHYSICAL STORAGE

3-Schema Arch: External=User Views |

Conceptual=Relational Schema(Tables) |

Internal=Physical storage on disk (Efficiency/Speed determined here)

Storage Hierarchy: Primary(RAM)=Fastest, Volatile, Highest cost | Secondary(HDD/SSD)=Medium speed, Non-volatile, Medium cost |

Tertiary(Tape)=Slowest, Non-volatile, Lowest cost

Disk Access Components: Seek Time=Move arm to correct track(cylinder) | Rotational

Latency=Spin to correct sector | Transfer

Time=Read/write data(fastest component)

Record Organization: Fixed-length=All records same size | Variable-length=Records vary in size |

Spanned=Records can cross block boundaries |

Unspanned=Records stay within blocks

Blocking Calculations: bfr(blocking factor)= $\lceil \frac{\text{BlockSize}}{\text{RecordSize}} \rceil$ | b(blocks needed)= $\lceil \frac{\#records}{bfr} \rceil$

FILE ORGANIZATIONS

Type	Ins	Del	Search(key)	Search(non)
Heap	O(1)	O(b/2)	O(b/2)	O(b/2)
Ordered	O(b)	O(b)	O(log b)	O(b/2)
Hash	O(1)	O(1)	O(1)	N/A

Heap(Unordered File): Insert at end: Fast | Search/Delete: Linear scan | Use: Bulk loading, small files, full scans

Ordered(Sequential File): Sorted on ordering key field | Binary search on ordering key |

Insert/Delete expensive(maintain order) | Use: Range queries, with primary index

Hash File: Hash function: $h(K) = K \bmod M$ | Direct access to bucket | Collision handling: Open addressing, Chaining, Multiple hashing | Load

factor: Keep ~80% full | Use: Exact match queries, fixed size | Not good for ordering or range queries

INDEXING

Index Types: Primary Index=On ordering key, sparse(one entry per block) | Clustering

Index=On non-key ordering field, sparse |

Secondary Index=Any field, dense(one entry per record)

Dense vs Sparse: Dense=Index entry for every record | Sparse=Index entry for every block
Multi-level Index: Index on index to reduce search space | Continue until top level fits in 1 block | Access time: $\log_{bfr}(indexBlocks) + 1$

B+ TREES

Properties(Order p $)$: Internal nodes= $[p/2]$ to p pointers, $p - 1$ keys | Leaf nodes= $\lceil(p - 1)/2\rceil$ to $p - 1$ values, linked | Root= ≥ 2 children if internal node | **Balanced**=All leaves at same level | **Data**=Only in leaf nodes

Split Rules: **LEAF Split**: Copy middle key UP, keep in leaf. (Left: <, Right \geq). **INTERNAL Split**: Push middle key UP, remove from node.

Insertion Algorithm(Order 3): 1.Traverse tree to appropriate leaf node | 2.Insert value in sorted order within leaf | 3.If leaf has ≤ 2 values: Done | 4.If leaf has > 2 values: **Split**: Middle value copied up to parent (if leaf). Lower values \rightarrow left child, Upper values \rightarrow right child | 5.If parent full: Repeat split process upward | 6.If root splits: Create new root(tree height increases)

Search Algorithm: Start at root \rightarrow Compare search key with node keys \rightarrow Follow appropriate pointer \rightarrow Repeat until leaf reached \rightarrow Search within leaf

Time Complexity: $O(\log_p n)$ where p =order, n =records

Example Calculation: 100,000 records, order 120 B+ tree | Depth $\approx \lceil \log_{120}(100000) \rceil \approx 3$ | Access time = depth + 1 = 4 block accesses

NORMALIZATION

Goals: Reduce redundancy | Eliminate insertion/deletion/update anomalies | Minimize NULL values | Preserve dependencies

Functional Dependencies(FD): $X \rightarrow Y$ means X functionally determines Y | If $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$ | **Full FD**=Cannot remove any attribute from X | **Partial FD**=Can remove attribute(s) from X | **Transitive FD**= $X \rightarrow Y$, $Y \rightarrow Z$ implies $X \rightarrow Z$

Armstrong's Axioms: **Reflexive**=If $Y \subseteq X$ then $X \rightarrow Y$ | **Augmentation**=If $X \rightarrow Y$ then $XZ \rightarrow YZ$ | **Transitive**=If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$ | **Union**=If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$ | **Decomposition**=If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

Normal Forms: **First Normal Form(1NF)**=All attributes have atomic(indivisible) values | No multivalued or composite attributes | No repeating groups | **Second Normal Form(2NF)**=Must be in 1NF | No partial dependencies(non-prime attributes fully dependent on entire key) | **Third Normal Form(3NF)**=Must be in 2NF | No transitive dependencies | No non-prime attribute depends on another non-prime attribute

Normalization Process: $1NF \rightarrow 2NF$: Identify partial dependencies \rightarrow Create new table for each partial dependency \rightarrow Remove partially dependent attributes from original | $2NF \rightarrow 3NF$: Identify transitive dependencies \rightarrow Create new table for transitive dependency \rightarrow Keep determinant as FK in original table

Decomposition Properties: **Lossless Join**: Must NOT generate spurious tuples (fake rows).

Rule: Common attribute must be Key for R_1 or R_2 ($R_1 \cap R_2 \rightarrow R_1 \text{ OR } R_2$) | **Dependency**

Preservation: All original FDs representable in decomposed relations

TRANSACTIONS

ACID Properties: **Atomicity**=All or nothing execution | **Consistency**=Transform DB from valid state to valid state | **Isolation**=Transactions execute as if alone | **Durability**=Committed changes persist despite failures

SQL Transaction Commands: **START TRANSACTION**; **/BEGIN**; | **COMMIT**;=Make permanent | **ROLLBACK**;=Undo all changes

Transaction States: Active \rightarrow Partially Committed \rightarrow Committed | Active/Partially Committed \rightarrow Failed \rightarrow Aborted

Concurrency Problems: **1.Lost Update**= T_1 and T_2 update same data, one update lost | **2.Dirty Read**= T_1 reads uncommitted data from T_2 (then T_2 rolls back) | **3.Unrepeatable Read**= T_1 reads same data twice, gets different values | **4.Phantom Read**= T_1 re-executes query, different rows appear

CONCURRENCY CONTROL

Locking Mechanisms: **Binary Locks**=Locked(1) or Unlocked(0), Restrictive: blocks readers when writers present | **Shared/Exclusive Locks**: Shared(S)=Multiple transactions can read | Exclusive(X)=One transaction can write | Rule=Multiple S locks allowed, X lock blocks all others

Two-Phase Locking(2PL): Growing Phase=Acquire locks, no releases | **Lock Point**=All locks acquired | **Shrinking Phase**=Release locks, no acquisitions |

Guarantees=Serializability | **Problem**=Doesn't prevent deadlock

Deadlock: Two or more transactions wait eternally for each other | **Handling Strategies**:

Prevention=Abort if deadlock possible(conservative) | **Detection**=Check periodically, kill one transaction |

Avoidance=Acquire all locks before starting | **Livelock**=Transaction never gets lock(use FCFS or priority aging)

Timestamp Ordering: Each transaction gets unique, monotonically increasing timestamp | READTS(X)=timestamp of youngest transaction that read X | WRITETS(X)=timestamp of youngest transaction that wrote X

Rules for Transaction T: Write X=If $READTS(X) > TS(T)$ or

$WRITETS(X) > TS(T) \rightarrow ABORT$ | **Read X**=If $WRITETS(X) > TS(T) \rightarrow ABORT$

Advantages=No deadlocks | **Disadvantages**=Cyclic restarts possible, overhead for timestamps

Optimistic Concurrency Control: Three Phases: 1.**Read**=Transaction reads DB, updates local copies | 2.**Validate**=Check if serialization violated at commit time | 3.**Write**=If valid, write changes; else restart | **Best for**=Read-heavy workloads, low conflict probability

Serializability: Serial Schedule=No interleaving(always correct) | **Serializable**

Schedule=Equivalent to some serial schedule |

Testing (Precedence Graph):

- Nodes:** Create a node for each transaction.
- Edges:** Draw $T_i \rightarrow T_j$ if they access item X and:
 - T_i writes X before T_j reads X (WR)
 - T_i reads X before T_j writes X (RW)
 - T_i writes X before T_j writes X (WW)

Result: Schedule is serializable \iff no cycles.

RECOVERY MANAGEMENT

Transaction Log: Write-Ahead

Logging(WAL)=Log written before DB changes |

Log Records: $[T, start]$ =Transaction T begins |

$[T, X, old_val, new_val]$ =T updates X |

$[T, commit]$ =T commits | $[T, abort]$ =T aborts |

$[checkpoint]$ =Checkpoint taken

Checkpoint: Suspend all transactions temporarily \rightarrow Force write modified buffers to disk \rightarrow Write checkpoint record to log \rightarrow Resume transactions

Recovery Techniques: **Deferred Update(NO UNDO/REDO)**: Changes written to DB after commit | On crash: Transactions committed before checkpoint=Nothing | Transactions committed after checkpoint=REDO(use NEW values) |

Uncommitted transacts=Nothing(never wrote DB)

Immediate Update(UNDO/REDO): Changes written to DB before commit | On crash:

Transactions committed before checkpoint=Nothing

| Transactions committed after checkpoint=REDO(NEW values) | Uncommitted after checkpoint=UNDO(OLD values, newest \rightarrow oldest)

QUERY OPTIMIZATION

Query Representation: SQL is translated into a **Query Tree** (Relational Algebra Tree). Nodes = operations (σ, π, \bowtie) , Leaves = relations. Optimizer rearranges this tree.

Heuristic Rules: 1.Perform **selection**(σ) early(reduce tuples) | 2.Perform **projection**(π) early(reduce attributes) | 3.Most restrictive operations first | 4.Combine σ with \times into \bowtie (join) | 5. **Pipelining**: Stream results between operators to avoid disk I/O (vs **Materialization**: writing temp tables).

Selection Strategies: 1.Linear search(scan all blocks): b accesses | 2.Binary search(if ordered): $\log_2 b$ accesses | 3.Primary index: $\log_2(b_i) + 1$ accesses | 4.Clustering index(range): Access multiple blocks | 5.Secondary index: $\log_2(b_i) + 1$ (or more for non-unique)

Join Algorithms: **Nested Loop Join**: For each tuple in R, scan all of S | Cost: $b_R + (b_R \times b_S)$ (use smaller as outer) | Best: $b_{small} + (b_{small} \times b_{large})$ |

Sort-Merge Join: Sort both relations on join attribute, Merge sorted relations | Cost: Sort cost + $b_R + b_S$ | **Hash Join**: Hash both relations on join attribute to buckets, Join tuples in matching buckets | Cost: $2(b_R + b_S)$ for partitioning + joining | **Index Join**: Use existing index on one relation | For each tuple in other, use index to find matches | Cost: $b_R + (|R| \times indexAccessCost)$

Selectivity Estimation:

$S = \frac{\# \text{tuples Satisfying Condition}}{\text{total \# tuples}}$ | Lower S = more selective = do first

DATABASE SECURITY

Access Control Models: Discretionary(DAC): Owner controls access, Grant/revoke privileges |

Mandatory(MAC): System-enforced levels: Top Secret>Secret>Confidential>Unclassified
Bell-LaPadula Rules: 1.No read up: Can't read higher classification | 2.No write down: Can't write to lower classification

SQL Security Commands: CREATE USER

'user'@'host' IDENTIFIED BY 'pass'; | DROP USER 'user'@'host'; | GRANT

{SELECT|INSERT|UPDATE|DELETE|ALL} ON

database.table TO 'user'@'host' [WITH GRANT OPTION]; | REVOKE privileges ON db.table FROM

'user'@'host'; | CREATE ROLE role_name; | GRANT

privileges TO role_name; | GRANT role_name TO

'user'@'host';

Privilege Levels: Global=ALL PRIVILEGES |

Database=CREATE, DROP, ALTER |

Table=SELECT, INSERT, UPDATE, DELETE |

Column=SELECT(col), UPDATE(col)

Statistical Database Security:

Problem=Aggregate queries can reveal individual data | Solutions=Limit queries if result < threshold

| Limit repeated queries on same data | Add "noise" to results | Query set size restrictions

PERFORMANCE FORMULAS

Access Time Estimates: Heap(avg)= $b/2$,

Heap(worst)= b | Ordered(binary)= $\log_2 b$ |

Hash(avg)=1(no collisions) | Primary

Index= $\log_2(b_i) + 1$ | B+ Tree=tree depth +

$1 \approx \lceil \log_p(n) \rceil + 1$

Example Problem: Given: 100,000 records, 500B each, 2048B blocks, order-120 B+ tree | Calculate:

$bfr = \lfloor \frac{2048}{500} \rfloor = 4$ | $b = \lceil \frac{100000}{4} \rceil = 25000$ blocks | Heap

avg= $25000/2 = 12500$ accesses |

Ordered= $\log_2(25000) \approx 15$ accesses | B+

depth= $\lceil \log_{120}(100000) \rceil = 3$ | B+ access=3 + 1 = 4 accesses

DATA TYPES(MYSQL)

INT, SMALLINT, BIGINT | DECIMAL(p,s), FLOAT, DOUBLE | CHAR(n), VARCHAR(n) |

TEXT, BLOB | DATE, TIME, DATETIME |

BOOLEAN

COMPARISON: RELATIONAL ALGEBRA VS SQL

$\sigma_{cond}(R)$ =WHERE condition | $\pi_{attrs}(R)$ =SELECT

attrs | $R \cup S$ =UNION | $R - S$ =EXCEPT/NOT IN |

$R \cap S$ =INTERSECT/IN | $R \times S$ =FROM R,S(no join) | $R \bowtie S$ =NATURAL JOIN | $R \bowtie_{cond} S$ =JOIN

ON condition

PHP & MYSQL IMPLEMENTATION

Basics: Server-side \rightarrow HTML | $<?php \dots ?>$ |

$\$var$ & $\$row['Col']$ are Case Sensitive | Concat with . | Ends with ; | **Forms:** Radio: $<\input type="radio" name="x" value="1">$ sends ID.

Uploads: $<form enctype="multipart/form-data">$ \rightarrow $\$FILES['f'][tmp_name']$ & move_uploaded_file().

DB Pattern:

```
$db = mysqli_connect("host", "u", "p", "db");
if(mysqli_connect_errno()) die(mysqli_connect_error());
$id = $_POST["id"]; // Get data from form
$res = mysqli_query($db, "SELECT * FROM t WHERE id=$id");
if(!$res) die("Query Fail"); // Result is FALSE on failure
while($row = mysqli_fetch_assoc($res)) { // Returns NULL at end
    echo $row["col"]; // Case Sensitive match to DB!
}
mysqli_free_result($res); mysqli_close($db);
```