Lecture Notes

CS258 - Database Systems

Introduction

1. **Database System (DBS)** is used to model, access (query+update), analyse, store, secure, ensure/maintain consistency of (in face of failure/recovery or concurrent transactions) and optimise access to (indexing, best order of query operations) data.

2. DB manages structured data (e.g. tables). Recently, NoSQL/graph DBs manage unstructured (e.g. images) data, relying on **Information Retrieval (IR)**. **Data modelling**: identify data items, the relationships between them and their attributes (characteristics).

3. **Relation** can be represented as a **table** comprising **columns** of entity attributes, and a set of **rows** that store entity or relationship values, formally, unordered set **tuples** $\langle A_1, ...A_n \rangle$ with column-attribute values A_i. Attributes have **domains** (e.g. age $\in \mathbb{Z}^+$). Order of tuples and attributes doesn't matter.

4. Rows must be uniquely identifiable with a **key** - singular or composite. Can use row-ids/sequential numbers as **artificial/surrogate** keys, but they lack semantic meaning.

5. **Query** needs to specify table name and attribute names of interest. Combined with constraints, eligible rows (data items) are returned.

6. **Relation Schema** (**intension**) is the description of a relation, denoted $R(A_1, ..., A_n)$ for relation name R and attributes A_i `STUDENT(id, name)` all having **domains**. **Tuples** (**extension**) are unordered sets of values derived from appropriate domain, comprising a relation. Schema is formal description of structure of stored data, whereas data is the actual values stored under that schema.

7. **State** (instance) $r(R)=\{t_1, ...t_m\} \subset \text{dom}(A_i) \times ... \times \text{dom}(A_n)$ of a relation is the set of n-tuples $t_i = < v_1, ..v_n >$ of values $v_j \in \text{dom}(A_j)$ currently in the relation (current table).

8. **Database (DB)** is collection of relations. **Database Schema** is the set of all relation schemas in the DB.

9. **Structured Query Language (SQL)** is a declarative Data Definition Language (**DDL**): schemas, relations, constraints and Data Manipulation Language (**DML**): updates, queries, that simplifies interaction with DBMS. SQL relations use **bags** (multisets) of tuples, not sets.

Keys and Constraints

1. **Permissible states** of a relation satisfy R-DBMS **constraints**: key constraint (tuples have unique key), entity integrity (key attributes not NULL) and referential integrity (FK links shared cross-table attributes); domain ($0 \leq$ age ≤ 100) and semantic attribute integrity constraints aka business rules (e.g. manager pay > employee salary).

2. **Superkey (SK)** of relation R is a subset of R's attributes that follow the **key constraint**: in all valid states $r(R)$, any two distinct tuples $t_1, t_2 \in r(R) : t_1[SK] \neq t_2[SK]$. Superset of PK: $PK \subseteq SK$. **Candidate key** is the minimal SK: if any attribute removed, it's no longer a SK.

3. **Entity integrity** constraint: none of the PK attributes (even if composite) can be NULL in any tuple of $r(R)$, otherwise PK would prevent unique identification.

4. **Foreign Key (FK)** is the set of attributes $t_1 \in R_1$ that references t_2 in relation R_2 if: $t_1[FK] = t_2[PK]$ or $t_1[FK]$ is NULL and $\text{dom}(FK) = \text{dom}(PK)$. Can reference any unique key, not just PK. This ensures **referential integrity (RI)** constraint for cross-table relationships.

5. **Primary key (PK)** uniquely and minimally defines tuples in a relation - arbitrarily chosen out of candidate keys, more efficient than SK since no redundant key attributes.

6. **Insert violation** *domain*: attribute value for new tuple is outside the domain. *key*: inserted key already exists (duplicate keys), *referential integrity*: FK value of new tuple references PK value that doesn't exist in references relation. *Entity integrity*: PK value in new tuple is null.

   Delete violation: *referential integrity*: PK value of deleted tuple is referenced from other relations. Can handle with cascading delete.

   Update violation: similar to insert, may violate all constraints. Such violating operations can be rejected, or cascaded by the DB and the user is informed.

7. **Functional Dependencies (FD)** allow derivation of good DB designs. For attribute sets X, Y, Z, constraint on possible tuples in $r(R)$: $X \rightarrow Y$ means $t[Y]$ depend on values of $t[X]$. Formally $t_1[X] = t_2[X] \rightarrow t_1[Y] = t_2[Y]$. Real-world semantics (name $\rightarrow$ address may not hold).

8. **Theorem:** If K is SK, then $K \rightarrow A$ for any attributes $A \subseteq R$. Conversely, if $X \rightarrow R$, then X is an SK.

SQL as Data Definition Language

1. Query returns a relation. SQL is case-insensitive unless string is quoted. Modification commands insert, delete, update aren't queries

2. NULLs are not real values, so can't be compared with expressions or other NULLs, since they are all different from each other. Use 3-state logic (True/False/Unknown) - comparison with NULL yields UNKNOWN. Useful in contexts where the value is unknown, unavailable, or inapplicable (e.g. unmarried person's spouse).

3. **Catalog** is named dictionary of schemas, constraints and optimisation stats in SQL environment. Create schema using **CREATE SCHEMA** statement (**AUTHORIZATION** 'admin') optionally defines the owner of the schema.

4. **Base tables (relations)** are created and stored as a file by DBMS, **virtual relations (views)** don't necessarily correspond to physical/permanent files - used as shortcuts to data. Both need table name, attributes, their types, and any constraints Key attribute fields are underlined.

```
CREATE TABLE students( -- base table relation
  studentID INTEGER PRIMARY KEY, -- studentID
  studentName VARCHAR(30) NOT NULL,
  -- some constraints like FOREIGN KEY/CHECK);
CREATE VIEW viewname -- virtual relation
```

5. **Attribute constraints** imposes additional restrictions on attribute domains based on application semantics: **CHECK**(<bool expr>), e.g. CHECK(age>0). Applies restrictions on accepted values (condition evaluate to true), can be added to end of CREATE TABLE to run whenever tuple is inserted/updated.

6. Named **CONSTRAINT** <cname><bool expr> is like CHECK, but stored for easy reference: id INT CONSTRAINT <cname>, else DEFAULT sets default attr value e.g id INT DEFAULT 1

7. **Key constraints**: **PRIMARY KEY** for unique PK, candidate keys with **UNIQUE** allowing NULLs, and non-unique FK through **FOREIGN KEY** with optional **referential triggered action**: **SET NULL**, **CASCADE**, **SET DEFAULT** that define what happens to the field when referenced value is modified. Can't add value to FK attribute if not in PK it references. FK references values, so attribute names might vary, fixed operation order: PK exists before insert FK.

```
number INT PRIMARY KEY   -- define PK
name VARCHAR(15) UNIQUE -- candidate key
FOREIGN KEY (ID) REFERENCES Persons(ID) -- RI const-
 ON DELETE SET NULL ON UPDATE CASCADE   -- raint
```

8. **DROP TABLE** <name> deletes the entire relation, including schema and data, so not simply opposite of CREATE TABLE.

SQL as Data Manipulation Language

1. **SELECT** <$attr_1$,..> **FROM** <table> WHERE <condition> retrieves multisets of relevant tuples from FROM relation by applying **row-selection**: iterating over all tuples in the relation and checking for WHERE boolean clause, then **column-projection**: returning attributes from SELECT.

2. SELECT **DISTINCT** removes duplicates by treating resulting multiset as a set. Wildcard matching SELECT * FROM fetches all attributes. To check if NULL: attribute IS(NOT) NULL: "attribute=NULL" never true.

3. **Qualification** SELECT table.**attribute** references specific attribute from table. If tables in FROM have shared attribute name, avoids ambiguity.

4. **Aliasing**: SELECT table <newname> assigns a new name to the retrieved table. Can be used for qualification with SELECT T.attribute FROM table T. Can also rename selected attributes: SELECT <attribute> **AS** <newname>, or to custom strings using: SELECT name, ".." AS custom will create an attribute custom, with all values string "..".

5. .. WHERE .. **ORDER BY** <attribute(s)> **ASC/DESC** orders result tuples by one or more attributes (asc/desc)ending.

6. WHERE clause supports **AND**, **OR**, **NOT** and parentheses just like normal boolean conditions. Also support case sensitive string matching with <attribute> **LIKE** <pattern> or **NOT LIKE**, % for any string of ≥ 0 characters, or _ for any 1 character. e.g. .. WHERE phone LIKE '%167-___'

7. **INSERT INTO** <table> VALUES (<val_1>, ..) inserts a tuple of values in same position as table's attribute list, so need to remember the schema of the relation. INSERT doesn't produce output unless there's an error.

8. INSERT INTO <table> (A_1,...) VALUES (<val_1>,...) inserts **partial records** (values) for specified attributes, unspecified attributes are either set to NULL or their DEFAULT. Useful if don't know standard order of attributes. INSERT INTO likes(color,name) VALUES('red','Bob');. Can omit auto-incremented PKs. Inserts can be rejected.

9. **DELETE FROM** <table> WHERE <condition> deletes relation tuples that match the condition. Unlike DROP, doesn't delete the table itself. DELETE .. WHERE EXISTS <query> handles more complex cases, EXISTS returns true only if resulting relation is non-empty. Marks all tuples satisfying WHERE condition then deletes them. If WHERE is unspecified/abscent - all tuples are deleted.

10. **UPDATE** <table> **SET** (<A_1=newVal>,..) WHERE <cond> changes values of specified attributes in certain tuples of a relation. Can use arithmetic expressions (salary$\times$ 2), provided values, DEFAULT or NULL.

Multi-table and Nested (sub)queries

1. **Multi-table queries** `SELECT * FROM table1, table2,..` result in the **cartesian product** which concatenates each row from table_1 with each row of table_2. Size of the result is $|\text{table}_1| \times |\text{table}_2|$. `WHERE` is optional, but recommended to reduce output size and make it practical.

2. Cartesian product contains wrong info for the query as it combines all rows with each other, so restrict with `SELECT * FROM s, c WHERE s.FK = c.PK` (**equi-join**), or any other attributes that link tuples from both relations.

3. When need to distinguish copies by following the relation name with name of a **tuple-variable**. Given Beers(name, manf), find all same-manufacturer pairs avoiding duplicates using lexicographical order $<$ (**theta-join**):

```
SELECT b1.name, b2.name FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND b1.name < b2.name;
```

4. **Subqueries** are surrounded by parentheses **(<subquery>)** equivalent to creating new table $T1$, and running original query on it: `SELECT name FROM (SELECT * FROM T) T1`.

5. **Scalar subquery**: arithmetic allowed if subquery returns value (1x1 table): `SELECT x FROM..WHERE x=(SELECT x ..)` **Scoping rule**: attribute refers to the most closely nested relation with that attribute: x is selected from both query and the subquery, but they're separate local variables.

6. **IN** evaluates to true iff tuple is a member of the relation, **NOT IN** is the opposite, both refer to **set** operators $\in, \notin$. `SELECT x,..FROM..WHERE x IN (<subquery>)` will select sub-rows where x appears in resulting subquery bag. Also works on tuples: ... `WHERE(attr1,..attrn) IN` ...

7. Subqueries `(SELECT x1 FROM X)` vs multi-table queries `X.x1` are not identical. The latter may produce duplicates.

8. **ALL/ANY** return true if all ($\forall$) or ≥ 1 comparisons are true. `..WHERE <attr/tuple> <operator> ALL/ANY <subquery>`. `..WHERE x>ALL(<subquery>.x)` selects rows with x greater than all x's from subquery. `ANY=SOME` (can use either)

9. **EXISTS** `<subquery>` allows for emptiness testing ($\exists$) on relations, returns true if any value, including `NULL` is present. **NOT EXISTS** is the opposite ($\nexists$)

10. If inner query refers to outer query attribute, they are **correlated**. In this case, the query is evaluated once per each tuple in the outer query instead of usual single subquery evaluation followed by outer query evaluation.

11. `<subquery>` **UNION/INTERSECT/EXCEPT** `<subquery>`: subquery results are multisets, so support $\cup$, $\cap$, and $\setminus$, but by default set operators remove duplicates (more efficient). Can force set to be bag with `<SetOp>` **ALL** (e.g.`UNION ALL`)

Join Queries

1. **JOIN** allows to query data combined from n tables simultaneously (n-way joins). However, may produce inaccurate data when joining on non-foreign-key attributes.

2. `A` **CROSS JOIN** `B` produces cartesian product - every tuple of A is concatenated with every tuple of B. Equivalent to `SELECT * FROM A,B`. Excessive information - use `WHERE`.

3. `A` **NATURAL JOIN** `B` is equivalent to forming product of A,B, and keeping tuples whose same-name/type attributes have same value - avoids repeated columns, unlike inner join. Can be inner (default) and outer.

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \end{bmatrix} \text{NATURAL JOIN} \begin{bmatrix} B & C \\ 2 & 5 \\ 4 & 7 \\ 9 & 10 \end{bmatrix} = \begin{bmatrix} A & B & C \\ 1 & 2 & 5 \\ 3 & 4 & 7 \end{bmatrix}$$

4. `X` **INNER JOIN** `Y` **ON** `<condition>` combines rows from X with rows from Y if boolean condition satisfied. Same as `SELECT * FROM X,Y WHERE X.a = Y.a`. Mostly uses **equi-join** "=", but allows any operator. Unless specified, `JOIN` defaults to inner join.

5. Inner joins can lose information as the tuple that doesn't join with any tuple from the other relation disappears from the result. Prevented by padding such tuples with NULL (**dangling tuples**). Left/right outer join keeps such dangling tuples from left/right.

6. `X` **JOIN** `Y` **ON** `<condition>`, or **theta join**, extends natural, inner joins to allow more complex conditions, e.g. functional expressions. `ON` need not use matching column names, can refer via quantification and aliasing.

7. `A` **LEFT/RIGHT/FULL OUTER JOIN** `B ON <condition>` preserves tuples from first/second/both tables and joins them with rows from the other table upon successful condition, and a row of `NULL`s otherwise. Can use **NATURAL OUTER**, making `ON` optional.

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \end{bmatrix} \text{OUTER JOIN} \begin{bmatrix} B & C \\ 4 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} A & B & C \\ 3 & 4 & 5 \\ 1 & 2 & NULL \\ NULL & 6 & 7 \end{bmatrix}$$

rows: natural join, left and right outer joins respectively.

8. Can use **USING** instead of ON to specify attributes to equi-join on: `A JOIN B USING (id, name)` is equivalent to: `A JOIN B ON (A.id=B.id AND A.name=B.name)`. `USING` deduplicates columns unlike `ON`.

Aggregation Queries

1. Arithmetic **SUM**, **MAX**, **MIN**, **AVG** and **COUNT** that counts number of non-NULL attribute values can be used within queries to obtain aggregate values for matching tuples. But, `SELECT COUNT(*) FROM A` counts **all** A's rows.

2. NULL **never contributes** to aggregation queries. If all values in column are NULL then aggregations return NULL except for COUNT that returns 0, never min/max. Use `COUNT (DISTINCT <attribute>)` to discard duplicates.

3. **GROUP BY** `<attribute>` groups rows by attributes, so tuples with matching attribute values are combined. Summarises data using aggregate functions across each group instead of the entire query NULL counts as a separate group. e.g. SUMming revenues within each company branch.

```
SELECT branch, SUM(revnue) FROM Comp GROUP BY branch
```

4. Attributes in the `GROUP BY` clause must be listed in the `SELECT` clause. If any aggregation is used, each element of `SELECT` must either be aggregated or contained in `GROUP BY` attribute list, else error. Filter out rows, group them, then apply aggregation. Can group by multiple attributes.

5. Selective grouping `GROUP BY` .. **HAVING** `<condition>` eliminates aggregated groups not satisfying condition. `WHERE` does so before grouping, so aggregate functions can't be compared in `WHERE` clause - only in `HAVING` clause.

```
SELECT beer, AVG(price) FROM Sells GROUP BY beer
  HAVING COUNT (bar) >= 3 OR beer IN (<subquery>)
```

`HAVING` clauses have only grouping/aggregated attributes.

Views and DDL

1. SQL stores schemas within a catalogue/dictionary, which can be accessed using **DESCRIBE**. Tables are stored within schemas, even if not specified, still exists inside "working" schema. Most DBMSs store metadata in special schema `INFORMATION_SCHEMA` - standard.

2. **Assertions** are a type of constraint applied to the whole database state enforced by DBMS. Satisfied if no combination of tuples in database violates it. More heavyweight than normal constraints, so used as a last resort. Define with: **CREATE ASSERTION** `<name>` **CHECK** `(<condition>)`. Checked on updates, e.g. no student exceeds 5 modules.

3. `DROP TABLE (IF EXISTS) <name> CASCADE/RESTRICT` destroys table and the schema - modifying catalog **CASCADE** removes FK constraints of any table that references the one being deleted. **RESTRICT** drops iff no dependencies, else throw error. **IF EXISTS** doesn't throw an error if table doesn't exist.

4. **ALTER TABLE ADD/DROP/RENAME COLUMN/RENAME TO** `<new>` to add, remove or rename columns, or the table itself, **ADD/DROP CONSTRAINT** `<constName>` to add or remove constraints. Can also **ALTER COLUMN <attr> TYPE <newType>** to change attribute datatype. These modify original table. New columns initialised to `NULL` unless specified.

```
ALTER TABLE tableName
  ADD/DROP COLUMN <attr> -- add/destroy column
  ALTER COLUMN <attr> TYPE <newtype> -- datatype
  RENAME TO <newTableName> -- rename table
  RENAME COLUMN <attr> TO <newAttrName> -- column
  ADD/DROP CONSTRAINT <constraintName>
```

can't modify constraints: must drop old, then add new.

5. `CREATE (MATERIALIZED) VIEW <name> AS <query>` creates a virtual/real(materialised) table derived from other tables or views. Useful for frequent queries or authorisation without exposing original table. Queried like tables.

6. If `VIEW` replicates a table, can enforce constraints virtually without overhead of creating/dropping a real table, reflects updates from the **base** relation, hence infeasible to alter views as the changes don't propagate back to their base relation. Materialised tables need storage and incremental updates.

7. DBMS uses a spliced expression tree of **relational algebra operations** when interpreting queries as if the view were a base table. Optimise queries by pushing selections down the tree and eliminating unnecessary projections.

Relational Database Design

1. **Relational Model** vs SQL Model uses: relation=table, attribute=column, tuple=row. Uses set theory to develop a good model of the data that's easy to understand, query and manipulate. Adhere to the following **guidelines**:

2. **Guideline 1**: **make easy to understand**: each row in relation should represent one relationship instance (FD) or entity. Only refer to different entities with foreign keys.

3. **Guideline 2**: **remove redundancy** to avoid waste and inconsistency: don't store same things twice - refer to them using FK. Redundancy may cause update, insert and delete **anomalies** - have to prevent.

4. **Update anomaly**: modification to a row/column causes inconsistency in the database (only one of the related columns modified). **Insert anomaly**: creating a row where value of an attribute isn't known, using `NULL` isn't good. Finally, **delete anomaly**: deleting a tuple deletes all its values and information linked to it.

5. **Guideline 3**: **no `NULL`s** - they waste storage and complicate queries (may cause ambiguity). Under-utilised attributes (those where many values are `NULL`) shouldn't be included in the base relation, instead use **Decomposition** to split the table into multiple tables without losing adding any incorrect info to avoid anomalies.

6. **Guideline 4**: **no spurious tuples** (incorrect info): when decomposed tables are merged, they should result in the original table, so use keys as join attributes (PK/FK).

Functional Dependencies

1. A formal measure of the "goodness" of relational design can be measured using **functional dependencies**. Use them to define **keys** (normal forms for relations). FDs can express constraints such as attribute to attribute links that can't be expressed with composite keys.

2. FD's are constraints derived from meaning and interrelationships of data attrs of the relation schema R.

3. Given a set of attributes $X, Y \subseteq R$, say $X \to Y$, means: **determinant** X functionally determines **dependent** Y and holds if values of Y are uniquely determined by values of the X component. If X is a key, then $\forall Y \subseteq R : X \to Y$.

4. FD $X \to Y$: two rows sharing X must share Y value, for tuples $t_1, t_2 \in r(R)$: $t_1[X] = t_2[X] \to t_1[Y] = t_2[Y]$. Formal notation: $\{A_1, .., A_m\} \to \{B_1, .., B_n\}$, skip curly brackets if $m = n$ or $n = 1$: e.g. $A \to \{...\}$ or $A \to B$.

5. FD is a property of the relation schema, not of a state: can disprove FD with a counter example (e.g. same name maps to multiple surname) - can't prove by example (if the current state has some FD that can be broken by adding new tuples, it's not an FD).

Normalisation

1. **Normalisation theory** ensures easy understanding and retrieval of info, helps avoid redundancies. In unnormalised relations, data may be repeated within columns.

2. Start from **universal** all-attribute-listing relation, progressively **remove redundant** data to **avoid anomalies**. **Rejoin** at each query to avoid information loss using **Lossless** (non-additive) joins and **dependency preservation**. Remember that `JOIN` is an expensive operation!

3. Can quantify how efficient database design is using **Normal Forms (NF)**. Have $1^{st}, 2^{nd}, 3^{rd}$, Boyce-Codd, $4^{th}, 5^{th}, 6^{th}$ NFs. If relation R is of i^{th} NF then it's also of normal form $j < i$. NFs up to BCNF depend on FDs.

4. **1NF** (*"the key"*): requires all attributes and their domains to be **atomic** - without substructures (e.g. lists), composite values e.g. "10, 20", or repeating groups (multiple rows sharing same non-key attribute values), and a key. Can be thought of having a database for each non-key value. Might have insert, update and delete anomalies.

5. **2NF** (*"the full key"*): no key attribute is partially functionally dependent on any candidate key. For all key K and non-key A attributes: $\forall K, A \in R$: $K \to \{A\}$ is **irreducible**, or $\nexists K' \subset K : K' \to \{A\}$. Prevents 1NF anomalies and dependencies on subsets of keys.

6. **3NF** (*"and nothing but the key"*): non-key attributes depend only on the key. No transitive FDs such as key $\to$ *drug* $\to$ sideEffect since the latter depends on both the key and the drug - should separate the relations.

7. **BCNF**: if $X \to A$ then X is a superkey (the only arrows are "out of superkeys"). Provides lossless-join decomposition but not necessarily dependency preservation.

8. Set of relation schemas $\{R_1, .., R_k\}$ is a **lossless-join decomposition** of relation schema R in relation to its set of FDs F if for all **legal** (r satisfies F) instances r of R, it holds that: $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \ldots \bowtie \pi_{R_k}(r)$. In SQL:

```
r = (SELECT R1 FROM r) JOIN ... (SELECT Rk FROM r)
```

   Instance r is **legal** if it satisfies FD set F defined on R. Joining decomposed relations $R_1, ..R_k$ should yield (reconstruct) precisely r - no superfluous or missing tuples.

9. All implied FDs that can be logically deduced from original FD set F are called the **closure/cover** F^+ of F. This is known as **completeness property**. They can be derived using **Armstrong's inference rules**:
   - (IR1) **Reflexive**: $Y \subseteq X \Rightarrow X \to Y$ e.g. (XZ $\to$ X)
   - (IR2) **Augmentation** $X \to Y \Rightarrow XZ \to YZ$
   - (IR3) **Transitive**: $(X \to Y) \wedge (Y \to Z) \Rightarrow X \to Z$

   Where $XZ \stackrel{\text{def}}{=} X \cup Z$. These rules hold $\forall$ legal r of R.

10. **Proof** of IR1: Given $Y \subseteq X$, suppose $\neg(X \to Y)$, then $\exists t_1, t_2 \in r[R]$ s.t. $(t_1[X] = t_2[X]) \wedge (t_1[Y] \neq t_2[Y])$. Since $X \supseteq Y$, then $X = ZY$ for some set of attributes Z. Now, $(t_1[X] = t_2[X]) \Rightarrow (t_1[ZY] = t_2[ZY]) \Rightarrow t_1[Y] = t_2[Y]$. $\perp$

11. **Proof** of IR2: Given $X \to Y$, $\forall t_1, t_2 \in r(R)$ it holds that $t_1[X] = t_2[X]$ and $t_1[Y] = t_2[Y]$. Now consider $t_1[XZ] = t_2[XZ]$ for some $Z \subseteq R$, so $t_1[Z] = t_2[Z]$, so combine to get $t_1[YZ] = t_2[YZ]$. $\square$

12. IRs derived from IR(1-3) by completeness property:
    - **Decomposition**: $X \to YZ \Rightarrow (X \to Y) \wedge (X \to Z)$
    - **Union**: $(X \to Y) \wedge (X \to Z) \Rightarrow X \to YZ$
    - **Pseudotransitive**: $(X \to Y) \wedge (WY \to Z) \Rightarrow WX \to Z$

13. **Proof** of Decomposition: Given $X \to YZ$: $YZ \to (Y, Z)$ by reflexivity, so $X \to (Y, Z)$ by transitivity. □

14. **Proof** of Union: Given $X \to (Y, Z)$ then $XX \to XZ$ and $XZ \to YZ$ by augment., $X \to YZ$ by transit. □

15. Test for **non-additive join** ensures no superfluous data: R_1, R_2 is a **lossless decomposition** of R in F if either $R_1 \cap R_2 \to (R_1 - R_2)$ or $R_1 \cap R_2 \to (R_2 - R_1)$ exist in F^+.

16. **Dependency Preservation**: verify that decomposition of relation R into $\{R_1, ..R_n\}$ preserves all FDs in F by ensuring projected FDs $F_1, ..F_n$ satisfy $(\cup_{i=1}^{n} F_i)^+ = F^+$.

Relational Algebra

1. **Relational Algebra (RA)** defines declarative SQL operations using procedural set-theory-oriented expressions. Minimal set of operations is $\{\sigma, \pi, \rho, , \cup, -, \times\}$, everything else can be expressed through them. **Closure property**: RA takes relations as input outputs relations.

Name	Symb	Example	Equiv SQL
Projection	π	π_A	`SELECT A`
Selection	σ	σ_c	`WHERE c`
Product	$\times$	$R \times S$	`R CROSS JOIN S`
Natural Join	$*$ or $\bowtie$	$R * S$	`R NATURAL JOIN S`
Join	$\bowtie$	$R \bowtie_{a=b} S$	`R JOIN S ON a=b`
Set	$\cup$	$R \cup S$	`UNION`
	$\cap$	$R \cap S$	`INTERSECT`
	$-$	$R - S$	`EXCEPT`
Rename	ρ	$\rho_{R2(a,b,c)} R1$	... `AS` ...
Division	$\div$	$R \div S$	see point 9

2. **Projection** ($\pi_{<attr_1,..>}(R)$) selects specified attributes, contextually reducing the schema. If attribute list l_i doesn't have keys, result may contain duplicates, but RA uses sets, so they are eliminated. If $l_2 \subset l_1$ then $\pi_{l_1}(\pi_{l_2}(R))$ is illegal, as nested projection will contain less attributes. Only works if $l_1 \subseteq l_2$. **CHECK THE ORDER!!**

3. **Selection** ($\sigma_{<condition>}(R)$): independently apply condition to each tuple in R, select those evaluating to True. Have **commutativity**: $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$ and **Sequencing**: $\sigma_{c_1}(\ldots \sigma_{c_n}(R)) = \sigma_{c_1} \wedge \cdots \wedge \sigma_{c_n}(R)$ for any condition c_i.

4. **Renaming** ($\rho_{<newName_1,..>}(R)$) renames attributes $\texttt{attr}_i$ to $\texttt{newName}_1$. Rename the relation itself with with $\rho_S(R)$, or relation with attributes using $\rho_{S<newName_1,..>}(R)$.

5. Relations $R(A_1, .., A_m)$ and $S(B_1, .., B_n)$ are **type-compatible** if $m = n$, dom(A_i)=dom(B_i) for $1 < i < n$. Hence, can reliably perform set operations/theory.

6. **Join** ($\bowtie$) supports theta-join condition $\theta \in \{=, \neq, <, \leq, >, \geq\}$ applied as $A_i \theta B_j$. Eliminates spurious tuples unless equi-join ("="), which might entail duplication.

7. **Cross Join** ($\times$) of $R(A_1, .., A_m) \times S(B_1, .., B_n)$ is a relation $Q(A_1, .., A_m, B_1, .., B_n)$ of size $|Q| = |R| \times |S|$ s.t. each tuple in Q is a combination of an m-tuple from R and an n-tuple from S. However may produce spurious tuples, hence use σ after cross join.

8. **Natural Join** ($*$): join attrs must have same name, so no duplicates, often preceded by ρ to avoid ambiguity. The shared attributes are only projected once.

9. **Division** ($R \div T$) gives all entries in R that have an entry for all values in T. E.g. for each student's subject, subtract subjects they study from ones in T, if no subjects in T left, they must study all of them, so return their id.

```
SELECT id FROM S WHERE NOT EXISTS
  ((SELECT Topic FROM T) EXCEPT
   (SELECT Topic FROM R) WHERE R.id = S.id);
```

   Let $S(X)$ be a set of relations on X where $X \subseteq Z$. Write $Y = Z - X$, then division $R(Z) \div S(X)$ is equivalent to relation $T(Y)$ s.t. for $t_s = t_R[X]$ and $t = t_R[Y] \in T(Y)$ iff $\forall t_s \in S : \exists t_R \in R$. Simply, t comes from a tuple of R, keeping only attributes in Y, and it must be present in all combinations of tuples from S as a tuple in R. **REVIEW**

10. **"Skinny"** relation is one with a minimal set of attributes, typically reduced to only those necessary for a specific query or operation, often through projection to eliminate redundant or unused columns, so less wide ("skinny").

11. **Aggregate functions**: given list of attributes of R and a function list $f_1, .. f_n$ of form `<<function><attribute>>` pairs, use aggregation (sum avg, max, min, count) to combine such functions: `<grouping attrs>` $\Im$ $f_1, .., f_n(R)$

Relational Calculus

1. **Relational calculus (RC)** helps define and understand the relational model through predicates and propositions. RC is declarative with true/false statements that describe relationships among entities such as data tuples. RC concerns either **tuple** or **domain**.

2. **Predicate** is a declarative statement that evaluates to true or false. It describes the meaning of a sentence, in case of RA: what (not) to retrieve. Statements can be

composed of other statements connected with logical operators ($\wedge/\vee/\neg$). Variables can be quantified ($\forall/\exists$).

3. **Substitution**: assign a value/designator for a parameter/variable within a predicate. **Instantiation**: substitute all parameters/variables in predicate. **Proposition** is an instantiated predicate.

4. Predicate **intension** (meaning): properties/qualities associated with the statement. **Extension**: set of all instantiations for which the predicate holds true. Intension is specific, extension is generic/wide-ranging.

5. RC vs RA: predicate $\simeq$ schema, proposition $\simeq$ tuple, extension $\simeq$ relation state. **COME BACK TO THIS**

6. **Closed world assumption**: what is not currently known to be true (in the DB) is assumed to be False.

7. **Codd's Theorem**: RA and RC are equivalent in their expressive power. Languages equivalent in expressive power to RA are **relationally complete**: SQL, RC.

Tuple Relational Calculus (TRC)

1. **Tuple Relational Calculus (TRC)**: variables range over tuples (variable=tuple). To define queries with RC, need to know: 1. **range relation**: one over which tuple variables operate, 2. condition or formula used to select tuples and 3. attributes to be retrieved per selected tuple.

2. **Restricted** first order logic formula F over tuples:
$$F = \{t_1.A_1, .., t_n.A_n \mid \texttt{COND}(t_1, .., t_n)\}$$
where COND is a formula with free variables (tuples) $t_1, ..t_n$ not tied to a relation, so can take any possible value. Specify **range relation** R they come from:
$$F = \{t.A_1, ..t.A_n \mid R(t) \texttt{ AND } \texttt{COND}(t.A_1, ..t.A_n\}$$
now variable t is explicitly tied to relation R.

```
SELECT t.A1,..t.An FROM R(t) WHERE COND
```

3. Conditions can specify range relation, selection condition or inner join condition. Attributes preceding "|" are equivalent to projection π. All free variables must precede "|".

4. **Atomic formulas**: for relation name R and tuple variable t_i, constant c and $\texttt{op} \in \{=, <\leq, >, \geq, \neq\}$:
$$R(t_i), \quad (t_i.A \texttt{ op } t_j.B), \quad (t_i.A \texttt{ op } c), \quad (c \texttt{ op } t_i.A)$$

5. **Compound Formulas**: for nested formulas F_i:
$$\texttt{NOT(F)}, \quad \texttt{F}_1 \texttt{ OR F}_2, \quad \texttt{F}_1 \texttt{ AND F}_2, \quad (\forall t)F, \quad (\exists t)F$$

6. **Free variables** range over all possible tuples, and can take any value. Quantifiers $(\forall, \exists)$ are **bound variables**, fixed within their scope (all/some).

Formula	Equivalent
$(\forall x)(P(x))$	$\neg(\exists x)(\neg P(x))$
$(\exists x)(P(x))$	$\neg(\forall x)(\neg P(x))$
$(\forall x)(P(x) \wedge Q(x))$	$\neg(\exists x)(\neg P(x) \vee \neg Q(x))$
$(\forall x)(P(x) \vee Q(x))$	$\neg(\exists x)(\neg P(x) \wedge \neg Q(x))$
$(\exists x)(P(x) \vee Q(x))$	$\neg(\forall x)(\neg P(x) \wedge \neg Q(x))$
$(\exists x)(P(x) \wedge Q(x))$	$\neg(\forall x)(\neg P(x) \vee \neg Q(x))$

where $\forall x : P(x) \equiv (\forall x)(P(x))$: all x satisfy predicate P.

7. **Antecedent** $\Rightarrow$ **Consequent** is logical implication. **UNIVERSAL QUANTIFICATION EXAMPLE**

8. **Safe** RC expression always returns finite number of tuples. Ensure through stating the range relation. Equivalent to relational algebra. Example of **unsafe** relation: $\{t|\texttt{NOT (EMPLOYEE(t))}\}$: t could be any tuple in universe.

Domain Relational Calculus (DRC)

1. **Domain Relational Calculus (DRC)**: variables range over attribute domains. $\{x_1, ..x_n \mid \texttt{COND}(x_1, ..x_n)\}$. Only the desired attributes preceding "|" are allowed to be free variables in COND, but it may have other bound variables.

2. **Atomic formulas**: for relation name R of arity k, attributes x_i, constant c and $\texttt{op} \in \{=, <\leq, >, \geq, \neq\}$:
$$R(x_1, ..x_k), \quad (x_i \texttt{ op } x_j), \quad (x_i \texttt{ op } c), \quad (c \texttt{ op } x_i)$$

3. **Compound Formulas**: for nested formulas F_i:
$$\texttt{NOT(F)}, \quad \texttt{F}_1 \texttt{ OR F}_2, \quad (\forall t)F, \quad (\exists t)F$$

JDBC and Procedural Languages

1. Can access DBs through SQL commands, file/batch, application programs. DB interfaces can be embedded SQL commands, libraries, or DB programming languages.

2. **Client-Server Architecture** comprises a DB server (e.g. postgres) listening on same network **port** as clients using interactive queries and application programs (e.g. psql). Can embed SQL commands in host languages (Java/C etc.), and execute them using **JDBC** API.

3. **Impedance Mismatch** problem: host (Java) blended with DB sublanguage (SQL), structurally different. Need binding for datatype translation, convert SQL bags of tuples to Java classes. Loop through tuples using a **cursor**.

4. One Java program can have multiple connections to multiple databases handled by **driver manager class** comprising `getDriver`, `registerDriver`, `deregisterDriver` etc. Mostly use Pure Java driver for direct-to-database, but can use JDBC-ODBC bridge/Native-API/Middleware.

5. JDBC main classes: **`Connection`**, **`Statement`**, **`ResultSet`**.

6. Need a `Connection` object for each DB connection, created using driver manager's **`getConnection()`** that allows to connect to specific DB sources using urls, providing authentication to DBMS.

JDBC query walkthrough

```
import java.sql -- import the sql lib
Class.forName("DriverName"); -- import modules
passwd = readentry("input passwd: "); -- for login:
Connection con = DriverManager.getConnection(<url>
    + <user> + <passwd>); -- establish a connection
String query = "SELECT ... WHERE ssn=?";
PreparedStatement p=con.prepareStatement(query);
p.clearParameters; p.setString(1,readentry());
ResultSet r = p.executeQuery();
while (r.next()) {println(r.getString(1);}
```

7. `Statement` class comprises subclasses: **`PreparedStatement`** and **`CallableStatement`**. SQL queries require both to execute within JDBC and convert the output to **`ResultSet`** table-like class with row tuples and column attributes.

8. **`JUnixSocket`** interprets JDBC sockets as ports allowing to send precompiled SQL commands **`PreparedStatement`** (e.g. "`SELECT .. WHERE x=?`") where "?" are arguments, like a partially applied function. Can bind parameters to Java variables using `set<Type>(idx, val)` (e.g setString). Prep. stat. avoid runtime errors since already compiled. Safe unlike unchecked `createStatement(<query>)`.

9. Every statement has an **`executeQuery`** used for projection (`SELECT`) returning `ResultSet` object, and **`executeUpdate`** for all other operations, returns number of affected tuples. `ResultSet` cursor initially positioned right before the 1^{st} tuple in result, first `r.next()` retrieves the first tuple.

10. To keep track of the index (tuple #), use a pointer called **cursor** r, can "scroll" using r`.next()`.

11. Surround JDBC calls in `try/catch` to catch errors e.g. driver not installed, failed connection, wrong credentials, server down, statement was created or closed, permission issue, malformed query, constraint violation etc.

12. `(create/prepare)Statement` command supports **`RSType`**: navigation (e.g. forward-only, scrollable) and sensitivity to parallel updates and **`RSConcurrency`** that determines whether the `ResultSet` is read-only or updateable.

13. `ResultSet` type can be **scrollable** (doubly-linked list), **positionable** (array) and **sensitive** (updateable?)

ResultSet Type	Scrollable	Sensitive
ResultSet. TYPE_FORWARD_ONLY	Cursor can *only* move *forward. Not positionable.*	*Not sensitive* to changes made after ResultSet created
ResultSet. TYPE_SCROLL_INSENSITIVE	Cursor is scrollable – can move *forward and backward. Positionable*	*Not sensitive* to changes made after ResultSet created
ResultSet. TYPE_SCROLL_SENSITIVE	Cursor is scrollable – can move *forward and backward* and *positionable*	*Sensitive* to changes made by others to the database that occur after ResultSet created

14. Scrollability and sensitivity are independent of updatability. There are six valid `ResultSet` mode combinations (three types $\times$ two concurrency levels), but not all are supported by every DBMS. When updating, all columns (unless defaulted) must be included in the `ResultSet`.

15. `(get/update)Int(String column(Name/Index)` method is contained within each of the following methods. It retrieves or updates row components of each attribute addressed by its name or index.

Method	Description
`beforeFirst()`	Moves cursor just before the first row.
`afterLast()`	Moves cursor just after the last row.
`first()`	Moves cursor to the first row.
`last()`	Moves cursor to the last row.
`absolute(int row)`	Moves cursor to the given row.
`relative(int row)`	Moves cursor forward/backward by given rows.
`previous()`	Moves cursor to the previous row.
`next()`	Moves cursor to the next row.
`getRow()`	Returns current row number.
`moveToInsertRow()`	Moves cursor to a special row for inserting new data.
`moveToCurrentRow()`	Moves cursor back to the current row.
`updateRow()`	Updates RS row by modifying corresponding DB row.
`deleteRow()`	Deletes the current DB row in RS.
`refreshRow()`	Refreshes data in the current row from the DB.
`cancelRowUpdates()`	Cancels all updates to the current row.
`insertRow()`	Inserts a row into DB after moveToInsertRow() call.

16. By default, DB updates are auto-commited, but to avoid risk, can manually `conn.commit()` or `conn.rollback()` after doing `con.setAutoCommit(false)`.

17. Favour declarative code to procedural. SQL can consistently enforce all appropriate logic. Once you're done with connections, statements and result sets, need to close them with `<thing>.close()`, else might run out of cursors.

DB Programming architecture

1. Client applications use API to access server databases via standard interface. Between client and DB server there is some DB application - if embedded within the database - more efficient, can write functions, if not - more secure.

2. Can use persistent stored modules **PSM** procedural code (e.g. if .. then ..), but it's inefficient. **Triggers** allow for more complex checking/rejecting of new data - update DB when an action occurs, but inefficient.

3. **Embedded SQL** within in some languages use **precompiler** and **preprocessor** to parse SQL code.

4. **Communication variables**: SQLCODE returns 0 when statement executed successfully, 100 when no more available query data and < 0 upon an error. A more modern SQLSTATE: string of 5 chars returning '00000' upon success and '02000' when no data.

5. To retrieve tuples, can use INTO for single tuple and FETCH for whole rows, then browse using cursors.

Prepared Statements

1. **SQL injection**: attacker injects a string that changes the SQL command. Can cause unauthorised data manipulation/retrieval or execute harmful system commands.

2. **Mitigation**: only allow known good data: remove SQL keywords from input strings (can be limiting) or replace escape characters (e.g.”” with ”\””) to prevent further command injections (too cumbersome, use a library). Can keep the schema secret (attackers can still guess it), or keep errors vague, e.g. don't tell user value for 'username' column as now they know of this column, again, obscure. Best to use **access control (AC)**.

3. **Prepared Statement** `prepareStatement(<query>)` parses the query, and compiles it's plan (no data needed). Now, DBMS plugs input data into compiled query. Separates executable query from parameters (that shouldn't be interpreted as code). No need to remove keywords/escape characters, less complex/error-prone, immune to SQL injection. Moreover, more efficient since don't need to recompile per user query.

Database Security

1. DB security concepts: systems, law, ethics, politics, society and aim to maintain **CIA**: **confidentiality** (disclosure), **integrity** (corruption) and **availability** (DoS) using access control, inference control (access to stats about the data, not the data itself), flow control (prevent read from protected then write to unprotected) and encryption.

2. DBS has DB security and authorisation subsystem. Database security mechanisms can be **discretionary** (grant/revoke access privileges) or **mandatory** (classification levels for users and data items). MAC and DAC can be combined.

3. **DB administrator (DBA)** manages the DB including its security through: account creation (users/groups/software accounts), discretionary privilege granting/revocation and mandatory security level assignment.

4. **System Log** records each operation applied to DB. It's used to recover from transaction failure/system crash and serve as an **audit trail**.

5. **DB audit** is performed after suspicious activities are detected, reviews the log (audit trail) for all accesses and operations applied during the relevant period.

Discretionary Access Control (DAC)

1. **Discretionary Access Control (DAC)** can use queries to `GRANT` and `REVOKE` privileges. Privileges target *account level*: what user can run (`\du` in postgres), and *relation level*: who can access the relation and the types of operations they're allowed to perform on it (`\l`).

2. **Access matrix model** has **subject** rows (accounts) and **object** columns (relations, tuples, etc). $M(i, j)$ represents types of privileges that subject i holds on object j. Implemented by RDBMS, need not be stored in actual relation.

3. **Privilege control**: each relation has an owner (typically the creator) account with full privileges. In SQL, can assign owners to whole schemas (collections of tables/views/procedures), or multiple schemas (like subfolders) within database (not in MySQL). Owner can pass privileges on any of their objects to other users.

```
CREATE SCHEMA example AUTHORIZATION jonny;
```

4. Can grant SQL privileges on **retrieval** (`SELECT`), **modification** (`INSERT, UPDATE, DELETE`) or **referencing** with FK (`REFERENCING`). Commands `INSERT` and `UPDATE` are attribute-specific, so can specify their scope with `<table>(attr1,..)`, but `SELECT, DELETE` aren't, use views to scope them.

```
GRANT <command1,..> ON <table>(A1,..) TO <user>;
```

5. DAC can use views, for example, to access a subset of table's attributes, so create a view and grant `SELECT` on it. Same granting commands apply as for any table.

6. Can grant privileges temporarily, cancel or revoke them. `GRANT OPTION` propagates same privileges to others without knowledge of the owner, if revoked, also revoked from the propagations based on it.

```
GRANT .. ON .. TO .. WITH GRANT OPTION;
```

7. Some DBMSs (not SQL) allow users to impose horizontal and vertical propagation limits. **Horizontal** limit: `GRANT OPTION` grants permissions to $\leq i$ other accounts.
   $i = 2$ $A \rightsquigarrow B_1$ then $A \rightsquigarrow B_2$ but now $A \rightsquigarrow \varnothing$.

   Vertical limit of j: account can grant its privilege only to those with vertical limit $\leq j$. Vertical limit of 0 is same as having no `GRANT OPTION`.
   $i = 2$ $A_1 \rightsquigarrow A_2$ then $A_2 \rightsquigarrow A_3$ but now $A_3 \rightsquigarrow \varnothing$.

Mandatory Access Control (MAC)

1. DAC is mostly all-or-nothing (full/no access per command unless with views), but used by most commercial DBMSs. But might need to classify data and users into security classes for more fine-grained control, handled by MAC.

2. **Bell-LaPadula model** applies security classes:
   Top Secret(TS) $\geq$ Secret(S) $\geq$ Classified(C) $\geq$ Unclassified(U)
   each subject S has clearance `class(S)`, each object O has access classification `class(O)`. Enforces restrictions:

3. **Simple security**: S not allowed to read O unless class(S) $\geq$ class(O) ("no read up"). **Confinement** or start property: S not allowed to write O unless class(O) $\geq$ class(S) ("no write down"), enforcing flow control.

4. **Multi-level security**: assign security classifications to attributes; **tuple classification (TC)** = max attribute classification within the tuple. **Apparent key** would be the PK if ignored security classes, but same-key tuples may have different TCs (**polyinstantiation**), so the new real key is PK+TC.

5. **Multi-level relation schema** $R(A_1, C_1, .., A_n, C_n, TC)$: for attributes A_i, security classifications C_i and the tuple classification.

6. Filtering (class(S) < class(O)) hides classified data by putting `NULL` in its place. All attributes in apparent key must share security classification minimum within that tuple and be non-`NULL`, so if any non-key attribute can be seen, then the full apparent key has to be seen too.

7.

DAC	MAC
Flexible, more suitable in many domains	Rigid, requires security classification
Vulnerable to attacks, e.g. embedded Trojan horses	Prevents illegal flows of information
No mechanism controlling info usage once accessed	Controls information flow based on security labels
Better trade-off between security and applicability in most situations	Necessary for strict information control (e.g., military, government)

8. **Role-Based Access Control (RBAC)**: a **role** is a group of users/accounts with specific privileges. Can `CREATE/DESTROY ROLE <role>`, manage their privileges: `GRANT .. TO <role>`, or establish permission hierarchy with `GRANT ROLE <fromRole> TO <toRole>`, now toRole inherits all the permissions of fromRole. Often used in industry, can be combined with MAC or DAC.

9. **Statistical DB** represents aggregated/statistical data about populations (subgroups) to prevent direct access to confidential details about individuals. **Population** is a set of tuples that satisfy some condition. Queries only concern populations - no individual row retrievals - only statistical aggregate functions (`AVG, SUM,..`).

10. **Inference attack**: series of statistical queries may reveal individual values, e.g. filter the population until 1 person left, then request average data on them. Prevent by using a threshold on population size, prohibit repeated queries on same population, partition records into larger groups (disallow queries on their subsets), or introduce inaccuracies (noise) to the results with randomised responses.

11. **Randomised response**: insert noise to ensure **plausible deniability**. Flip a coin before recording each binary value, if heads - record the truth, if tails, record "YES". If p is percentage of "NO", then the correct percentage of "NO" is $2p$ as 50% will get tails (50% of YES is useless).

12. **Flow Control**: flow between X and Y happens when program reads from X and writes into Y. Need to specify flow policies: regulate distribution of information among accessible objects and specify allowed channels along which info is allowed to move.

13. **Crude binary classification**: confidential (C) and non-confidential (N) information classes, allow all flows except $C \rightsquigarrow N$. Assign security class to each program and memory region, read if program's class is $\geq$, and write if $\leq$ than the memory segment. **No read up, no write down.**

14. **Explicit flow** has direct class assignments $l = h \mod 2$. **Implicit flow** involves conditionals: if (h==1) then $l = 1$.

15. **Oracle label-based security**: row-level access control. Users and data have labels (security level) representing access control policies, which are executed along each query.

16. **Virtual Private Databases (VPD)**: each DB object is bound to a security policy (procedure stored in the DB). Execution of a policy adds a predicate (`WHERE` clause) to user's SQL statement, thus modifying user's data access.

Dimension	**Discretionary Access Control (DAC)**	**Mandatory Access Control (MAC)**
Decision Authority	Object owner (creator) decides who gets which rights	Central policy or OS kernel enforces based on security labels
Enforcement Mechanism	ACLs, GRANT/REVOKE, access-matrix model	Security labels on subjects & objects + fixed rules (e.g. Bell–LaPadula)
Granularity	Per-user or per-group; per-object and, with column-level grants, per-attribute	Per-classification level (coarser)
Delegation	Users can grant privileges to others (optionally limited with GRANT OPTION)	Users/processes cannot change or override labels
Flexibility	High: ad-hoc sharing, easy to extend	Low: policies must be defined up front and are difficult to change
Information Flow Control	None—once a user has rights, data can flow freely	Built-in: prevents read-up and write-down flows
Susceptibility to Attacks	Vulnerable to Trojan horses and privilege escalation	More resistant—users/processes can't grant rights they don't have
Typical Use Cases	Commercial DBMSes, collaborative environments	Military/government, high-assurance, regulated environments
Schema Changes Needed	None (works on existing relations via ACLs and views)	Requires labeling of data & users; schema extended with classification attributes