DATABASE ENGINEERING

Unit 2

The Relational Data Model, Keys and ER-to-Relation Mapping

Topic-wise study notes with importance ratings

30 topics — terminology and keys, integrity constraints, the complete mapping rule set, and storage architecture

Diagrams are drawn wherever a concept needs one

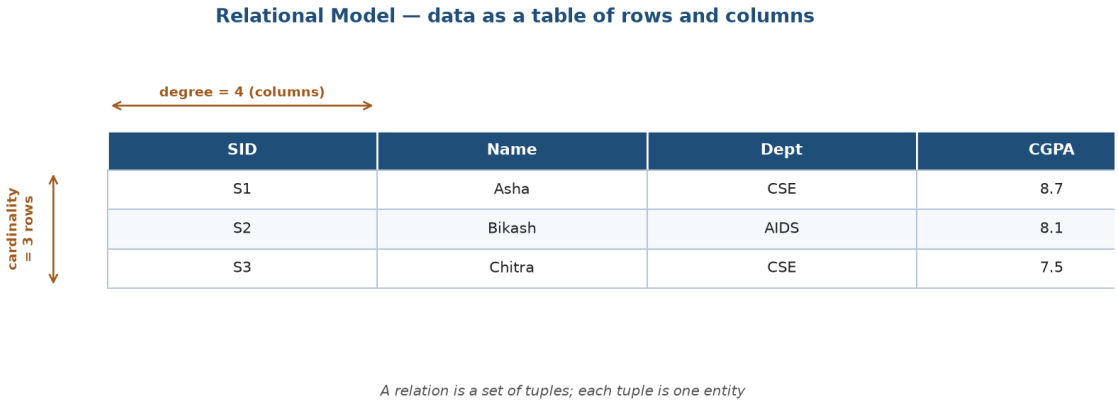
Topic-wise study notes. Every topic carries an importance rating from one to three stars: three stars means the topic is asked almost every year and must be known perfectly, two stars means it is regularly asked and should be prepared, one star means it is supporting material to read once. The notes explain the concept only; worked questions and answers are kept in a separate file.

Index

Topic, importance rating and page number. ★★★ must know, ★★ should know, ★ read once.

Topic	Importance	Page
2.1 Relation, Tuple, Attribute and Domain	★★★	3
2.2 Degree and Cardinality	★★★	3
2.3 Relation Schema and Relation Instance	★★★	3
2.4 Intension and Extension	★★★	4
2.5 Super Key	★★★	5
2.6 Candidate Key	★★★	5
2.7 Primary Key	★★★	5
2.8 Alternate Key	★★	6
2.9 Foreign Key	★★★	6
2.10 Composite Key	★★	6
2.11 Partial Key and Descriptive Attribute	★★	7
2.12 Relational Model Constraints	★★★	7
2.13 Domain and Key Constraints	★★	8
2.14 Entity Integrity Constraint	★★★	8
2.15 Referential Integrity Constraint	★★★	8
2.16 Referential Actions	★★	8
2.17 Schema Diagram	★★	9
2.18 Mapping — Strong Entity Set	★★★	9
2.19 Mapping — Composite Attributes	★★	9
2.20 Mapping — Multivalued Attributes	★★	10
2.21 Mapping — 1:1 Relationship	★★	10
2.22 Mapping — 1:N and N:1 Relationships	★★★	11
2.23 Mapping — M:N Relationships	★★★	11
2.24 Mapping — Weak Entity Set	★★★	12
2.25 Mapping — Specialization	★★	12
2.26 Detailed Storage Architecture	★★	13
2.27 File Organisation	★★	13
2.28 Indexing	★★	14
2.29 B Tree and B+ Tree	★★	15
2.30 RAID	★★	15

2.1 Relation, Tuple, Attribute and Domain ★★★



The relational model represents all data as **tables**. Four terms describe the table itself.

Term	Meaning	Example
Relation	A table of values; formally a subset of the Cartesian product of a set of domains	STUDENT
Tuple	One row of the relation — one entity	{S01, Asha, CSE, 8.7}
Attribute	One named column — one property	Name, Dept
Domain	The set of atomic values an attribute may take	Age: integers 0 to 120

Why the domain matters

The domain fixes both the type and the permitted values. A value outside the domain is rejected by the DBMS, which is what makes the domain constraint enforceable without any application code.

The formal definition

If D1, D2, ..., Dn are domains, a relation R is a subset of D1 × D2 × ... × Dn. Because it is a **set**, duplicate tuples are not allowed and the order of tuples is not significant — although a DBMS may physically store them in any order and may keep duplicates in an SQL result unless DISTINCT is used.

2.2 Degree and Cardinality ★★★

Term	Also called	What it counts	Changes over time
Degree	Arity	The number of attributes, that is the columns	No — fixed by the schema
Cardinality	—	The number of tuples, that is the rows	Yes — on every insert and delete

Reading a question

A relation with degree 4 and cardinality 12 has four columns and twelve rows. If a question says "the degree of the relation after the join", it is asking for the number of columns of the result, which is the sum of the two degrees minus the common attributes in a natural join.

2.3 Relation Schema and Relation Instance ★★★

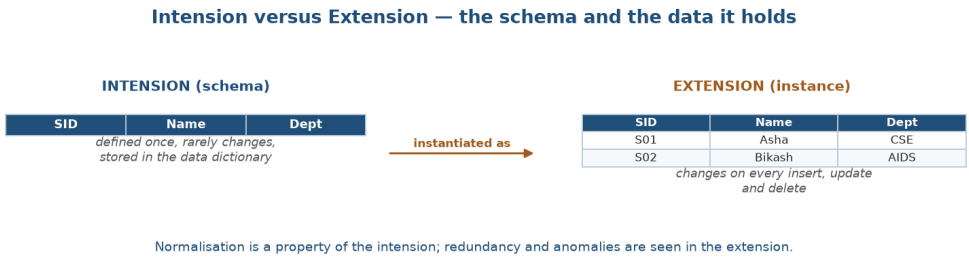
A **relation schema** is the name of the relation together with its list of attributes, written R(A1, A2, …, An). It describes the structure and does not change with time.

A **relation instance** is the set of tuples present at a particular moment. It changes whenever rows are inserted, deleted or updated.

Example

Schema: STUDENT(SID, Name, Dept, CGPA). Instance: the three rows currently stored. The schema is written once by the designer; the instance is what the users keep changing.

2.4 Intension and Extension ★★★



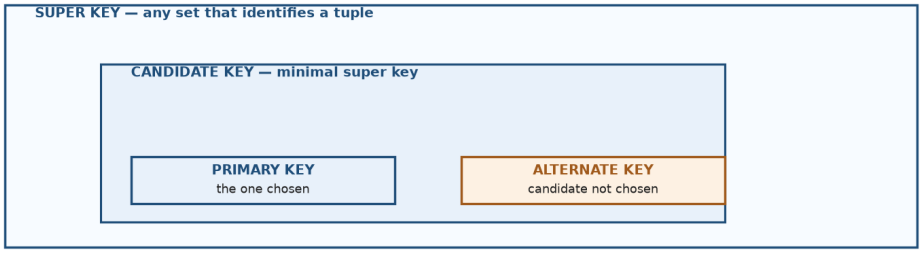
Intension (schema)	Extension (instance)
The permanent, structural part of a relation	The changing part — the tuples currently stored
Name, attributes, domains and constraints	The actual data values
Defined at design time using DDL	Populated and modified using DML
Stored in the data dictionary	Stored in the data files

Why the distinction is useful

Normalisation is a property of the **intension** — you normalise a schema. Redundancy and anomalies are observed in the **extension** — you see repeated facts in the rows. Understanding the difference also explains data independence: the intension can be changed with little effect on programs precisely because the extension is reached only through it.

2.5 Super Key ★★★

The Family of Keys — Super key contains Candidate key contains Primary key



Example: STUDENT(SID, RollNo, Email, Name) — candidate keys {SID}, {RollNo}, {Email}

{SID, Name} is a super key but NOT a candidate key, because Name can be removed

A **super key** is a set of attributes whose values uniquely identify each tuple of the relation. It may contain attributes that are not needed for identification.

Example

In STUDENT(SID, RollNo, Email, Name) where SID, RollNo and Email are each unique, all of these are super keys: {SID}, {SID, Name}, {SID, RollNo}, {RollNo, Email}, and every other superset of a candidate key.

Counting super keys

Every superset of a candidate key is a super key. If a relation has n attributes and a candidate key of size k, that candidate key alone generates 2^{n-k} super keys. This is the quick way to answer "list at least eight super keys".

2.6 Candidate Key ★★★

A **candidate key** is a **minimal super key** — a super key from which no attribute can be removed without losing uniqueness.

How to find candidate keys

1. Any attribute that **never appears on the right-hand side** of any functional dependency must belong to every candidate key. 2. Start with those attributes and compute the closure. 3. Add attributes only while the closure is not the whole relation. 4. Check minimality by removing each attribute in turn.

Example

R(A, B, C, D, E) with F = {A → B, B → C, CD → E}. A and D never appear on a right-hand side, so start with AD; (AD)+ gives every attribute, so {A, D} is the candidate key.

2.7 Primary Key ★★★

The **primary key** is the candidate key chosen by the designer to identify the tuples of the relation.

Its properties

- Unique for every tuple.
- Can never be NULL — this is the entity integrity rule.
- Only one per relation.
- Usually the smallest and most stable of the candidate keys.
- Should not be changed, because other relations may reference it.

Choosing one

Prefer a single attribute over a composite key, a value that never changes over one that might, and a short value over a long one — because the key is copied into every foreign key that references it.

2.8 Alternate Key ★★

An **alternate key** is a candidate key that was **not** chosen as the primary key. In STUDENT(SID, RollNo, Email), if SID is chosen as the primary key then RollNo and Email are alternate keys.

Its use

An alternate key still enforces uniqueness and still provides a fast access path, which is why it is normally given a UNIQUE constraint rather than being ignored.

2.9 Foreign Key ★★★

A **foreign key** is an attribute, or set of attributes, in one relation whose values must either match a value of the primary key of another relation or be entirely NULL.

Example

DeptID in STUDENT references DEPARTMENT(DeptID). Every value of DeptID in STUDENT must exist in DEPARTMENT.

Primary key	Foreign key
Identifies a tuple in its own relation	References the primary key of another relation
Cannot be NULL	May be NULL when the relationship is optional
Only one per relation	A relation may have many foreign keys
Enforces entity integrity	Enforces referential integrity
Must be unique	May repeat many times

How to identify one

Look for an attribute whose name or meaning matches a primary key of another relation, whose values all come from that key's value set, which repeats across rows, and which may be NULL. Confirm it from the declared REFERENCES clause if the schema is given.

2.10 Composite Key ★★

A **composite key** is a key made of two or more attributes together. It is used when no single attribute is unique.

Example

ORDER_ITEM(OrderID, ProductID, Quantity). OrderID alone repeats because an order has many products; ProductID alone repeats because a product appears in many orders; the pair is unique.

Mapping consequence

Every M:N relationship produces a composite primary key made of the two participating keys, and every weak entity produces a composite key made of the owner key plus its own partial key.

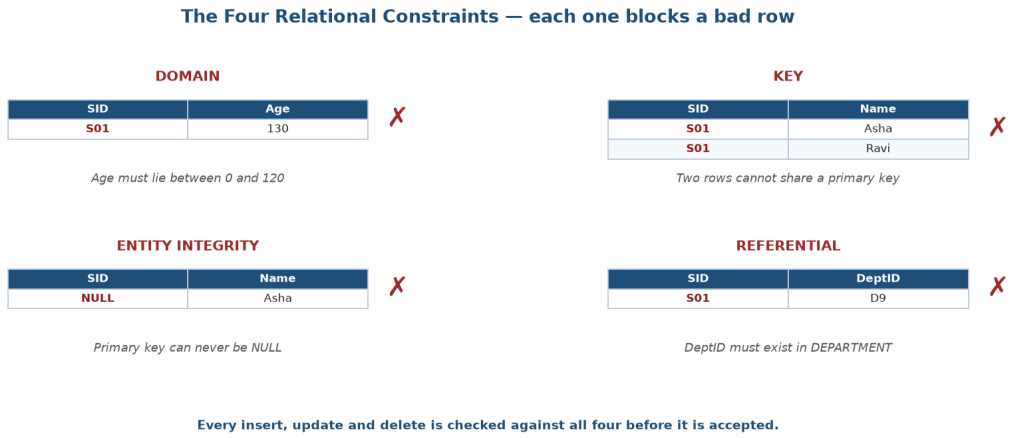
2.11 Partial Key and Descriptive Attribute ★★

Partial key	Descriptive attribute
Belongs to a **weak entity set**	Belongs to a **relationship**
Distinguishes entities within one owner	Describes one particular association
Example: Dependent_name, Payment_no	Example: Grade in ENROLLS, Hours in WORKS_ON
Combined with the owner key to form the full key	Stored in the table created for the relationship

Why both matter in mapping

A weak entity's table is keyed by owner key + partial key. A relationship's descriptive attributes are the reason an M:N relationship cannot be merged into either entity table — there would be nowhere to put them.

2.12 Relational Model Constraints ★★★



Constraints are the rules that every valid instance of the database must satisfy. The DBMS checks them on every insert, update and delete.

Constraint	Rule	Example
Domain	Each value must come from the attribute's domain	Age between 0 and 120
Key	No two tuples may share a primary key value	SID unique
Entity integrity	The primary key may not be NULL	SID never empty
Referential integrity	A foreign key must match an existing key or be NULL	DeptID must exist in DEPARTMENT
Semantic	Business rules the schema cannot express	Salary not above the manager's

Where each is enforced

Domain, key, entity and referential constraints are declared in the schema and enforced automatically. Semantic constraints need assertions or triggers, because they depend on values in other rows or other tables.

2.13 Domain and Key Constraints ★★

Domain constraint

Every value must belong to the attribute's domain — the right data type, the allowed range and the permitted format. Enforced by the declared type together with CHECK clauses. Example: Month must be one of January to December.

Key constraint

No two tuples may have the same primary key value. This is what makes identification of a tuple possible. It is declared with PRIMARY KEY and is enforced on every insertion and update.

2.14 Entity Integrity Constraint ★★★

The **entity integrity constraint** states that the primary key of a relation can never be NULL, because a tuple whose key is unknown cannot be identified.

Example

SID in STUDENT can never be NULL, whereas a non-key attribute such as Phone may be NULL. Note that this applies to the primary key only — a candidate key declared UNIQUE may, in some DBMSs, still permit NULL, which is why NOT NULL is usually added explicitly.

2.15 Referential Integrity Constraint ★★★

The **referential integrity constraint** states that a foreign key value in the referencing relation must either equal some existing primary key value in the referenced relation, or be wholly NULL.

Example

Inserting an EMPLOYEE with DeptID = D9 when DEPARTMENT has no D9 is rejected. Setting DeptID to NULL is allowed if the employee's department is not yet known.

Where it is checked

On insert and update of the child, and on delete and update of the parent key. The action taken when a parent changes is decided by the referential action, covered next.

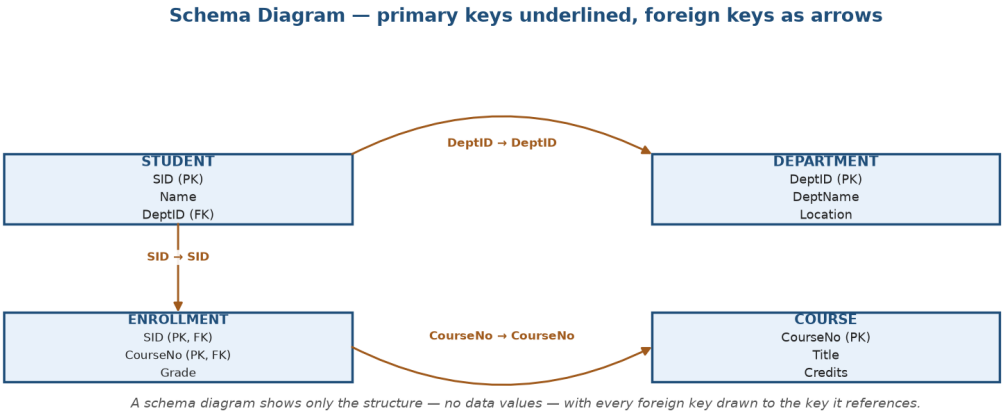
2.16 Referential Actions ★★

Action	Effect when a parent row is deleted or its key is updated
RESTRICT	The operation is rejected if child rows exist
CASCADE	The same operation is applied to the child rows
SET NULL	The child foreign key is set to NULL
SET DEFAULT	The child foreign key is set to a default value
NO ACTION	Deferred check — the operation fails only if the constraint is still violated at the end

Choosing one

RESTRICT is the safe default, because it never destroys data silently. CASCADE is convenient for genuinely dependent data, such as order lines that cannot exist without their order. SET NULL suits optional relationships.

2.17 Schema Diagram ★★



A **schema diagram** is a picture of the database schema in which each relation is drawn as a box listing its attributes, the primary key is underlined, and arrows run from each foreign key to the primary key it references.

What it shows and what it hides

It shows structure only — relation names, attributes, keys and references. It shows no data values, no cardinality and no participation, because those belong to the ER model rather than the relational model.

2.18 Mapping — Strong Entity Set ★★★

Each strong entity set becomes **one table**. Its key attribute becomes the primary key and its simple attributes become columns.

Example

The entity set STUDENT with key Student_id and attributes Name, Program becomes STUDENT(Student_id, Name, Program) with primary key Student_id.

The full rule set at a glance

ER construct	Tables produced	Rule
Strong entity set	1	Key becomes the primary key
Composite attribute	1	Store only the simple components
Multivalued attribute	2	One for the entity, one for the values
1:1 relationship	2	Foreign key on either side, or merged
1:N and N:1 relationship	2	Foreign key on the many side
M:N relationship	3	Its own table with a composite key
Weak entity set	2	Owner key + partial key
Specialization	1 or more	Depends on the chosen strategy

2.19 Mapping — Composite Attributes ★★

A strong entity set with composite attributes still needs only **one** table. The simple components of the composite attribute are stored; the composite attribute itself disappears.

Example

Student with attribute Name (First_name, Last_name) and Address (House_no, Street, City) becomes Student(Roll_no, First_name, Last_name, House_no, Street, City).

Why

The relational model allows only atomic values, so a composite attribute has no representation of its own. Its parts are what actually get stored, and the grouping exists only in the design.

2.20 Mapping — Multivalued Attributes ★★

A strong entity set with multivalued attributes needs **two** tables: one holding the simple attributes and the primary key, and one holding the primary key together with the multivalued attribute.

Example

Student(Roll_no, Name, City) and Student_Phone(Roll_no, Mobile_no), where the second table's primary key is the pair (Roll_no, Mobile_no).

Why two tables

A single column cannot hold a set of values without violating 1NF. Separating the values removes the repetition and lets a student have any number of phone numbers.

2.21 Mapping — 1:1 Relationship ★★

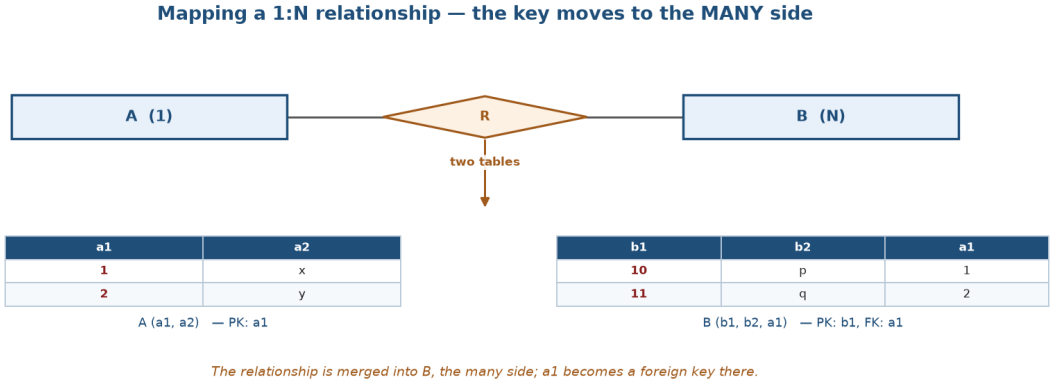
A 1:1 relationship produces **two tables**, and the foreign key may be placed on either side, or the two entity sets may be merged.

Participation	Rule
Both sides partial	Place the foreign key on either side
One side total	Place the foreign key on the **total** side, because that side always has a value
Both sides total	Merge everything into a single table and choose either key as the primary key

Example

EMPLOYEE manages DEPARTMENT. Since only some employees manage a department but every department has a manager, the total side is DEPARTMENT, so Mgr_ssn is placed inside DEPARTMENT.

2.22 Mapping — 1:N and N:1 Relationships ★★★



A 1:N or N:1 relationship produces **two tables**. The relationship is merged into the **many** side, so the key of the one side becomes a foreign key inside the many side.

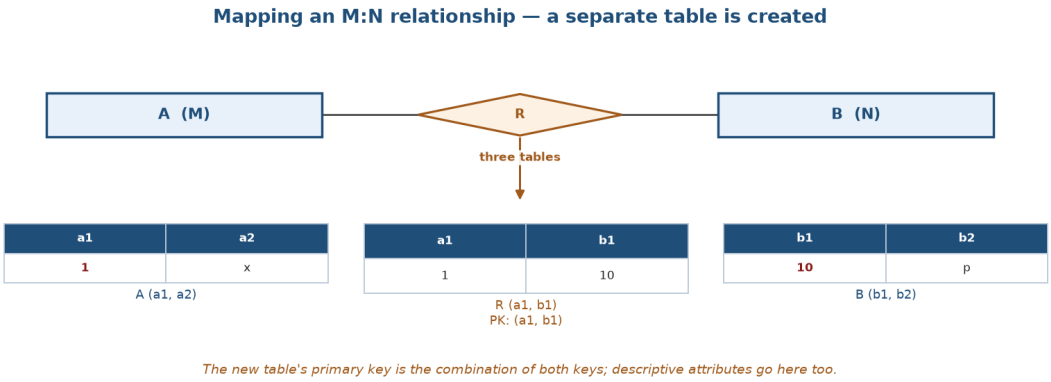
Example

DEPARTMENT 1 : N EMPLOYEE. Employee becomes EMPLOYEE(SSN, Name, …, Dnumber), where Dnumber is a foreign key to DEPARTMENT.

The total participation effect

If the participation on the many side is **total**, the merged foreign key acquires a **NOT NULL** constraint, because every row of that table must have a value. In the example, every employee must belong to a department, so Dnumber is NOT NULL.

2.23 Mapping — M:N Relationships ★★★



An M:N relationship produces **three tables**: the two entity tables plus a separate table for the relationship.

The relationship table

Its attributes are the primary keys of both participating entity sets plus any descriptive attributes of the relationship. Its primary key is the **combination** of the two keys.

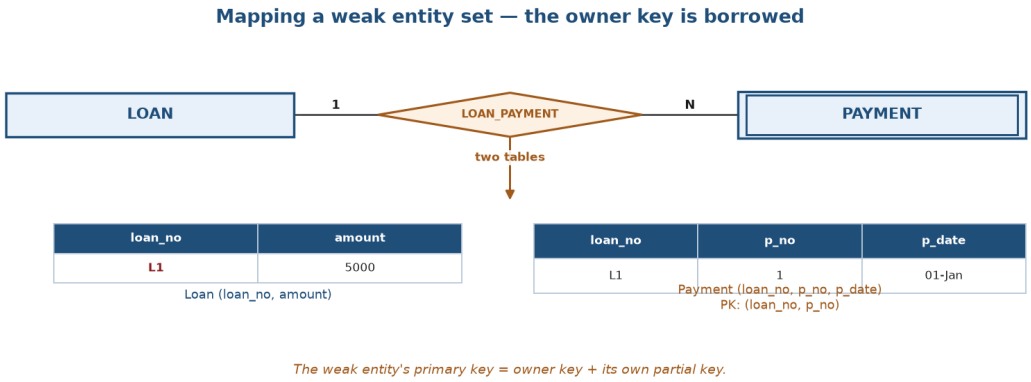
Example

STUDENT and COURSE with the M:N relationship ENROLLS carrying Grade become STUDENT(Student_id, Name), COURSE(Course_no, Title) and ENROLLS(Student_id, Course_no, Grade) with primary key (Student_id, Course_no).

Why it cannot be merged

Because each side relates to many rows of the other, there is no single place in either table to put the foreign key, and the descriptive attributes would have nowhere to go.

2.24 Mapping — Weak Entity Set ★★★



A weak entity set always appears with an identifying relationship and total participation. It needs **two tables**: the owner table and the weak table, whose primary key is the **owner's key plus the partial key**.

Example

LOAN(Loan_no, Amount) and PAYMENT(Loan_no, P_no, P_date, P_amount), where the primary key of PAYMENT is (Loan_no, P_no). P_no alone repeats across loans, so it cannot serve as a key on its own.

2.25 Mapping — Specialization ★★

There are three strategies for a specialization.

Strategy	Tables	Good when
Single table	One table for the superclass with a type attribute and all subclass attributes	Total and disjoint specialization with few subclass attributes
Table per subclass, superclass key only	Superclass table plus one table per subclass holding the subclass attributes and the superclass key	Many subclass attributes, any constraint
Table per subclass, all attributes	One complete table per subclass, repeating the inherited attributes	Queries read one subclass only

What to state in an exam

Say which strategy you are using and why. A disjoint, total specialization with few subclass attributes is best mapped to a single table; if the subclasses have many of their own attributes, one table per subclass avoids a large number of NULLs.

2.26 Detailed Storage Architecture ★★

Storage architecture is the **internal level** of the DBMS: how data is physically held and reached. It decides how quickly the logical operations can actually be executed.

Level	Characteristics
Cache and main memory	Fastest, volatile, small
Flash memory	Fast, non-volatile, moderate size
Magnetic disk	Slower, non-volatile, large
Optical and tape	Archival, very slow, very large

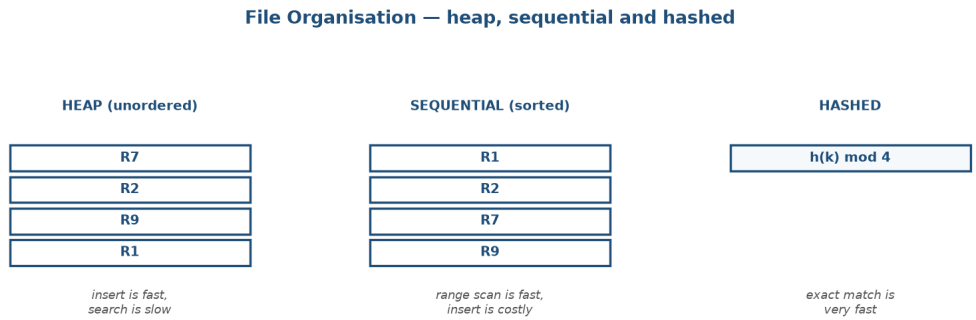
Records, blocks and files

A **record** is a collection of fields; records are grouped into **blocks** or pages, which are the unit of transfer between disk and memory. A **file** is a sequence of records, and its file organisation decides where each record is placed.

Why disk I/O dominates

Because a disk access is thousands of times slower than a memory access, the DBMS tries to keep frequently used blocks in memory and to minimise the number of blocks read. Every index and file organisation in this unit exists to reduce that count.

2.27 File Organisation ★★

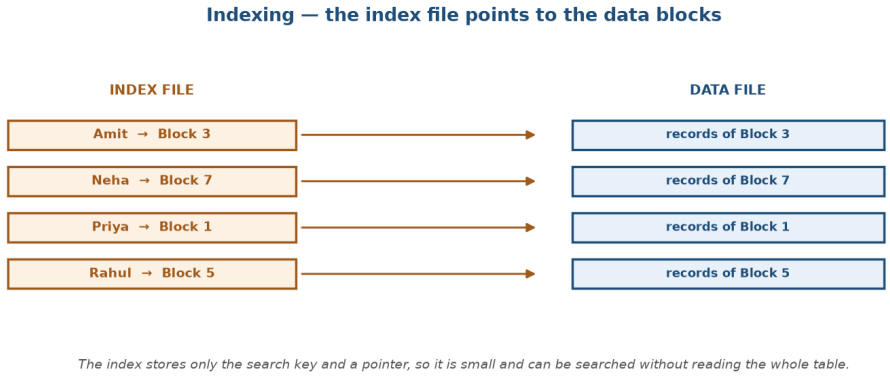


Organisation	Placement	Advantage	Disadvantage
Heap (unordered)	Anywhere with free space	Fast insert	Search needs a full scan
Sequential (sorted)	In order of a key	Fast range scan, ordered output	Costly insert and delete
Hashed	Address computed by a hash function	Very fast exact match	Poor for range queries
Clustered	Related records of several files stored together	Fast join	Reorganisation needed as data grows

Choosing one

A heap suits a table that is mostly inserted into; a sorted file suits reporting on a key; hashing suits lookup by an exact key such as an account number. Real DBMSs combine a heap for the data with indexes for access.

2.28 Indexing ★★



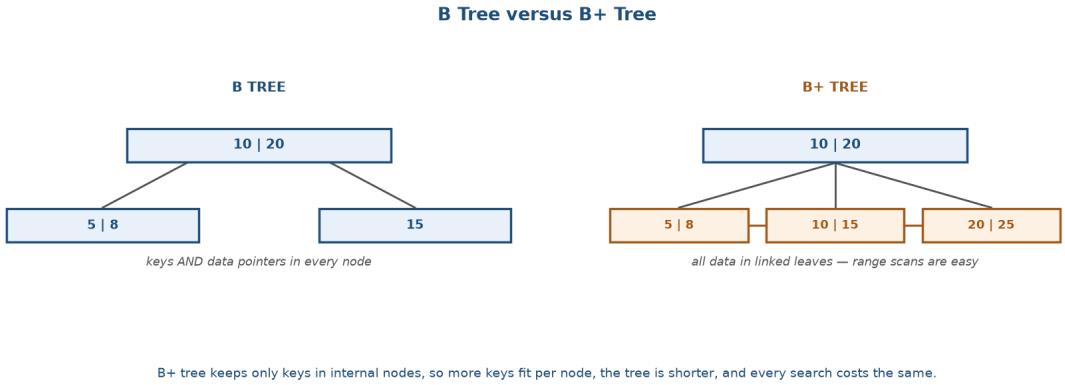
An **index** is an auxiliary structure holding search-key values together with pointers to the records, so that records can be found without scanning the whole file.

Type	Built on	Entries	How many per file
Primary index	The ordering key field of an ordered file	One per block	One
Clustering index	A non-key ordering field of an ordered file	One per distinct value	One
Secondary index	Any non-ordering field	One per record, or one per value with a pointer list	Many

The trade-off

An index speeds up selection and join but costs storage and slows down inserts, updates and deletes, because the index must be maintained too. This is why indexes are created on the columns that queries actually search, not on every column.

2.29 B Tree and B+ Tree ★★

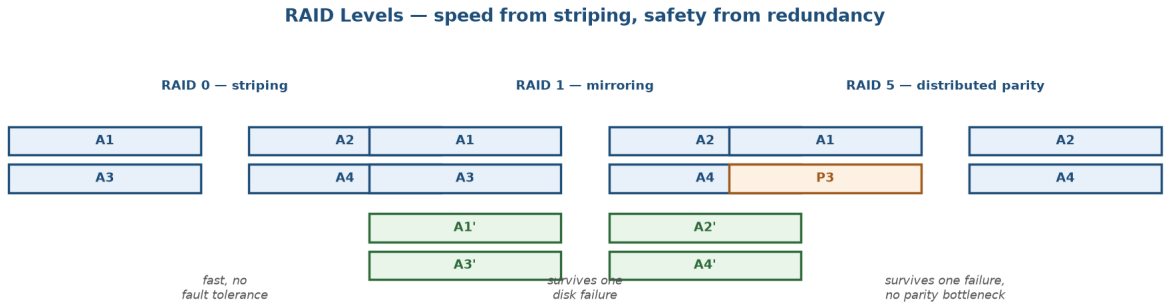


B tree	B+ tree
Keys and data pointers in every node	Keys only in internal nodes; all data pointers in the leaves
No linked leaves	Leaves linked in order, so range scans are easy
A search may end at an internal node	Every search reaches a leaf
Lower fan-out, taller tree	Higher fan-out, shorter tree

Why B+ tree is the standard

Because internal nodes hold only keys, more keys fit in one block, so the tree has fewer levels and therefore fewer disk accesses per search. Every search costs the same number of accesses, and the linked leaves make range queries efficient.

2.30 RAID ★★



RAID (Redundant Array of Independent Disks) uses several disks together to improve speed, reliability or both.

Level	Technique	Benefit
RAID 0	Block striping, no redundancy	Maximum speed, no fault tolerance
RAID 1	Mirroring	Survives one disk failure, costs half the space

Level	Technique	Benefit
RAID 4	Block striping with one parity disk	Survives one failure, but the parity disk is a bottleneck
RAID 5	Block striping with distributed parity	Survives one failure, no bottleneck
RAID 6	Two independent parities	Survives two failures

Why a database uses RAID

Durability requires that a disk failure not destroy committed data, and performance requires that reads be spread across several disks. RAID 1 or RAID 5 gives both, which is why database servers rarely run on a single disk.