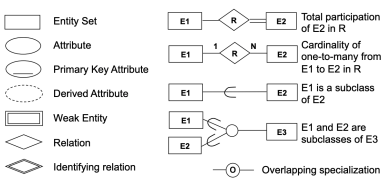
ERD

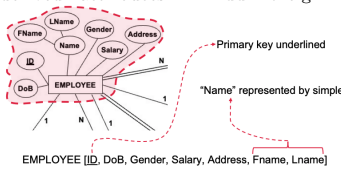


Super Key: Set of attributes that uniquely identifies entities.
Candidate Key: A super key without redundancy; every entity set has at least one.
Primary Key: A candidate key selected to uniquely identify the entity set.
Foreign Key: An attribute(s) in one table that uniquely identifies a row of another table.
Derived Attribute: An attribute calculated from other attributes.
Composite Attribute: An attribute that can be divided into meaningful sub-parts.
Cardinality: Indicates the relationship strength between entity sets.
• **One-to-One:** Each entity in A relates to at most one entity in B, and vice versa.
• **One-to-Many:** Each entity in A relates to multiple entities in B; each entity in B relates to at most one in A.
• **Many-to-Many:** Entities in A and B can relate to multiple entities in each set.
Weak Entity: An entity that cannot be uniquely identified by its attributes alone and needs a foreign key.
Eg. of a Relation Schema:
Student [sID : integer, $name$: varchar, $phone$: string, $major$: string]
Integrity Constraints (ICs): Conditions that every database instance must satisfy.
• **Domain Constraints:** Attributes must adhere to their defined domain.
Eg.: Age, defined as an integer, cannot accept a string or a NULL if disallowed.
• **Entity Integrity:** Primary keys must be non-NULL.
Eg.: The student ID, as a primary key, cannot be NULL.
• **Referential Integrity:** Foreign keys must correspond to existing records in their reference table.
Eg.: A student's major must match an existing record in the Majors table.
• **Key Constraint:** No two rows can have the same value for primary key attributes.
Eg.: No two students can have the same student ID.
• **User-Defined Constraints:** Specific rules defined by users that exceed standard SQL constraints.
Eg.: Full-time students cannot register for more than four courses per semester.

Relational Model

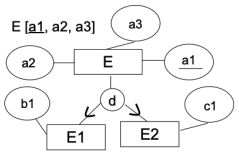
1. Regular Entity Mapping
2. Weak Entity Mapping
3. Binary one-to-one Relationship Mapping
4. Binary one-to-many/many-to-one Mapping
5. Binary many-to-many Relationship Mapping
6. N-ary Relationship Mapping
7. Multi-valued Attribute Mapping

Step 1: Entity Mapping For regular entities, create a relation with all simple attributes, underlining primary keys. Include only simple components of composite attributes; omit derived attributes. Add foreign keys later



EMPLOYEE [ID, DoB, Gender, Salary, Address, Fname, Lname]

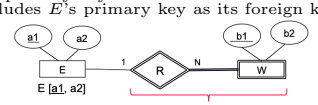
Subclasses: Create a table for each subclass, inheriting the primary key from the superclass. Applicable to disjoint, overlapping, total, or partial specializations.



E1 [a1, b1]
E2 [a1, c1]
Foreign Keys:
E1.a1 → E.a1
E2.a1 → E.a1

Step 2: Weak Entity Mapping

For each weak entity W connected to owner entity E via identifying relationship R, create a relation RW that:
• Includes all simple attributes of W and R (if any).
• Uses E's primary key and W's partial key as the primary key.
• Includes E's primary key as its foreign key.



E [a1, a2]
RW [a1, b1, b2], Foreign Key: RW.a1 → E.a1

Note 1: The primary key of RW is W's partial key + E's primary key combined
Note 2: Need to specifically note down the foreign key

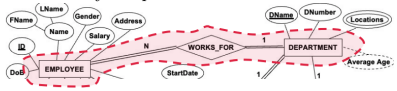
Step 3: Binary One-to-One Relationship Mapping

For each binary one-to-one relationship R connecting relations S and T, choose one relation (e.g., S) and expand it as follows:
• Include all simple attributes of R and S in S.
• Underline S's primary key.
• Include T's primary key as a foreign key in S.

DEPARTMENT [DName, DNumber, StartDate, ManagerID]
Foreign Key: DEPARTMENT.ManagerID → EMPLOYEE.ID
Extended at the full participation side

Step 4: Binary One-to-Many/Many-to-One Relationship Mapping

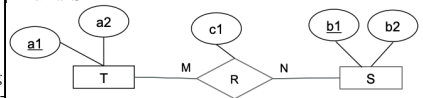
Assuming S is the "many" side, expand S as follows:
• Include T's primary key as a foreign key in S.
• Include any simple attributes from R.



EMPLOYEE [ID, DoB, Gender, Salary, Address, Fname, Lname, DName]
Foreign Key: EMPLOYEE.DName → DEPARTMENT.DName

Step 5: Binary Many-to-Many Relationship Mapping

For each binary many-to-many relationship R connecting entities T and S, create a new relation R that:
• Includes both T and S's primary keys and any simple attributes of R.
• Uses the combination of T and S's primary keys as the primary key of R.
• Adds foreign key references from R to both T and S.

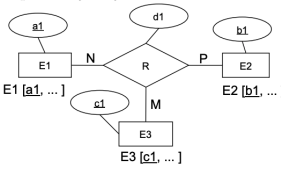


T [a1, a2]
R [a1, b1, c1]
S [b1, b2]
Foreign Keys: R.a1 → T.a1
R.b1 → S.b1

Notes:
• Mapping R optimizes storage by recording T-S pairs with minimal information.
• This strategy reduces the number of NULL values in foreign keys.

Step 6: N-ary Relationship Mapping

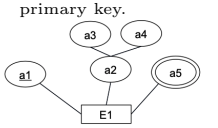
For each N-ary relationship R, create a new relation R that:
• Includes all connected entities' primary keys and any simple attributes of R.
• Adds foreign key references from R to all connected entities.
• Uses combination of all foreign keys as the primary key of R.



E1 [a1, ...]
E2 [b1, ...]
E3 [c1, ...]
R [a1, b1, c1, d1]
Foreign Keys:
R.a1 → E1.a1
R.b1 → E2.b1
R.c1 → E3.c1

Step 7: Multi-valued Attribute Mapping

For each multi-valued attribute A of entity $E1$, create a relation $E2$:
• Include $E1$'s primary key as a foreign key.
• Include attribute A (only its simple components if composite).
• Use combination of A and the foreign key as primary key.



E1 [a1, a3, a4]
E2 [a1, a5]
Foreign Key: E2.a1 → E1.a1

FDs and Normalization

Modification Anomaly: Data inconsistency from redundancy. *Eg.:* Changing a customer's address in multiple rows; if one row is missed, inconsistency arises.
Deletion Anomaly: Loss of certain attributes when deleting others. *Eg.:* Deleting an employee record might remove necessary dept. info.
Insertion Anomaly: Cannot insert some attributes independently. *Eg.:* Cannot add department info without corresponding employee details.

The Theory: Armstrong's Axioms
Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$.
Eg.: If X = {Staff ID, Department}, Y = {Department}, then {Staff ID, Department} → Department.
Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z. *Eg.:* If Staff ID → Department, then {Staff ID, Gender} → {Department, Gender}.
Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. *Eg.:* If Staff ID → Department and Department → Dept Address, then Staff ID → Dept Address.
Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.
Closure Definition: For an attribute or set A, its closure A^+ is all attributes functionally determined by it.
Normalization: Removing redundancy from data.

First Normal Form (1NF)
Definition: A relation schema is in 1NF if attribute domains are atomic and each attribute value is a single value from its domain.
1NF disallows sets, tuples, or combinations as attribute values.
Rationale: 1NF ensures attribute simplicity, facilitates detecting redundancies and anomalies, and is the minimal requirement for further normalization.

Second Normal Form (2NF)
Definition: A relation is in 2NF if:
• It is in 1NF.
• For every functional dependency (FD) $X \rightarrow Y$, where X is a candidate key and Y is a non-prime attribute, no proper subset of X determines Y.
2NF ensures the candidate key is minimal for every non-prime attribute it determines.

Boyce-Codd Normal Form (BCNF)
Definition: A relation R is in BCNF if for all non-trivial dependencies in R:
If $X \rightarrow Y$, then X must also be a super key for R.
Trivial Dependency: An FD is trivial if the LHS contains the RHS, e.g., $XY \rightarrow X$ is trivial.
Informal Interpretation:
• In BCNF, whenever a set of attributes of R determines another attribute, it should determine all attributes of R.
• In BCNF, only a super key of R can be on the LHS of non-trivial dependencies.

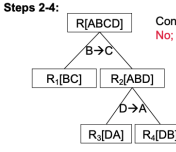
BCNF Decomposition Example 1

R[A, B, C, D]
FDs:
1. B → C
2. D → A

Decompose the given relation into BCNF:

Step 1: Derive all FDs – using the closure expression is recommended:
A+ = A, B+ = BC, C+ = C, D+ = AD

Steps 2-4:



Considering B→C, is B a super key of R?
No; we decompose
Considering D→A, is D a super key of R2?
No; we decompose

Step 5: Summarize – final schema is R_1[BC], R_3[DA], R_4[DB]
Note: Bracket styles can vary between "[]" and "()"

SQL

Data Definition Language (DDL): Statements to define the database schema.
Data Manipulation Language (DML): Statements to manipulate/query the data.

Main SQL DDL Statements:
CREATE, ALTER, DROP

```
CREATE TABLE <table name> (
<column name 1> <data type> <optional attribute constraint>,
<column name 2> <data type> <optional attribute constraint>,
<table constraint – must have primary key>,
<other optional table constraints, e.g., foreign key>);
```

ON DELETE (UPDATE) SET NULL | SET DEFAULT | CASCADE

```
CREATE TABLE StarsIn (
  StarID     INTEGER,
  MovieID  INTEGER,
  Role       CHAR(20),
  PRIMARY KEY (StarID, MovieID),
  FOREIGN KEY (StarID)  REFERENCES MovieStar(StarID)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY (MovieID)  REFERENCES Movie(MovieID)
     ON DELETE CASCADE
     ON UPDATE CASCADE );
```

ALTER TABLE Command
Used to change the schema. Actions include:
• Adding or dropping a column.
• Changing a column definition.
• Adding or dropping constraints.

```
ALTER TABLE <table name>
   ADD <column name> <data type> <optional attribute constraint>
       {, can optionally have more than one added columns}
   | DROP <column name> [can optionally append "CASCADE"]
   | ALTER <column name> <new data type>
   | ADD <constraint name> <constraint-options>
   | DROP <constraint name> [can optionally append "CASCADE"];
```

Dropping Tables with SQL
• Removes the table definition from the system catalog.
• Deletes all tuples within the table.
• Drops all constraints defined on the table, including constraints in other tables that reference this table.

```
DROP TABLE [can optionally append "IF EXISTS"]
<table name 1>, <table name 2>, ...
[can optionally append "RESTRICT" or "CASCADE"]
```

DML Statements
DML statements are used for managing data in a database. The main operations are:
SELECT, INSERT, UPDATE, DELETE
The Big Four Operations:
• **Projection:** Vertical filter.
• **Selection:** Horizontal filter.
• **Join:** Combine tuples from different relations for search purposes.
• **Sorting:** Order the resulting tuples according to a given sort key.

Projection vertically selects the attributes of a given collection of tuples.

```
SELECT [can optionally add "DISTINCT"] <attribute list, or "*">
FROM <table list>
[can optionally add "WHERE <condition>"];
```

• DISTINCT: Eliminate duplications in the values combos of the attribute list.
• *: Select "all" columns/attributes.

The syntax for selection:

```
SELECT [can optionally add "DISTINCT"] <attribute list, or "*">
FROM <table list>
WHERE <search condition and/or join condition>;
```

Substring Comparisons:
• **LIKE:**
  – `WHERE Address LIKE '%St Lucia'` *("%" for unrestricted length)*
  – `WHERE Name LIKE 'Rob__'` *("_" for one character)*
• **IN:**
  – `WHERE LName IN ('Jones', 'Wong', 'Harrison')`
• **IS:**
  – `WHERE DNo IS NULL`

Arithmetic Operators and Functions:
+, -, *, /, date and time functions, etc.
• `WHERE Salary * 2 > 50000`
• `WHERE Retire_Date - Bdate > 55`

BETWEEN:
• `WHERE Salary BETWEEN 10000 AND 30000`

In SQL, join operations combines tuples from different relations, done via FROM:

```
SELECT <attribute list>
FROM <table list of more than one table>
WHERE <join condition and/or search_condition>
```

Asymptotic Analysis Algorithm efficiency is measured by:
• **Time:** Execution duration.
• **Space:** Memory usage.
Efficiency is expressed as a function of input size n using Big-O notation $O(\dots)$. In Big-O:
• **Eliminate low-order terms:** Only the highest-order term matters as $n \rightarrow \infty$.
  – $O(4n + 5) \rightarrow O(4n)$
  – $O(0.5n \log n - 2n + 7) \rightarrow O(0.5n \log n)$
  – $O(2n + n^3 + 3) \rightarrow O(n^3)$
• **Eliminate scalar coefficients:** They don't significantly impact the growth rate.
  – $O(4n) \rightarrow O(n)$
  – $O(0.5n \log n) \rightarrow O(n \log n)$
  – $O(1000n) < O(n^2)$ for $n > 1000$

The common asymptotic values and their ranks when n → ∞

smaller ↓ bigger

• constant:	O(1)	
• logarithmic:	O(log n)	($\log_k n$, $\log n^2 \in O(\log n)$)
• poly-log:	O($\log^k$ n)	(k is a constant >1)
• sub-linear:	O(n^c)	(c is a constant, 0 < c < 1)
• linear:	O(n)	
• log-linear:	O(n log n)	(usually called "n log n")
• superlinear:	O(n^{1+c})	(c is a constant, 0 < c < 1)
• quadratic:	O(n^2)	
• cubic:	O(n^3)	
• polynomial:	O(n^k)	(k is a constant)
• exponential:	O(c^n)	(c is a constant > 1)

Follow these steps to perform asymptotic analysis on code:
1. Identify the input size.
2. Determine the cost of each line of code.
3. Treat the cheapest line's cost as 1 and find $O(\dots)$ in its raw form.
4. Simplify $O(\dots)$ based on the rules of thumb.

Example 2:

```
i = 1                              (line 1)
while i <= n do{                   (line 2)
   for j = i to n do{              (line 3)
      sum = sum + 1}               (line 4)
   i++}                            (line 5)
```

Step 3: Cost = $1 + \sum_{i=1}^{n}((\sum_{j=i}^{n} 1) + 1)$
line 1 line 2 line 3 line 4 line 5

Step 4: Lines 1 and 5's component will all become low-order components, hence can be directly omitted from now. Then,
Cost = $\sum_{i=1}^{n}((\sum_{j=i}^{n} 1)) = \sum_{i=1}^{n}(n - i + 1) = n + (n-1) + \cdots + 1 = \frac{1}{2}(n^2 + n)$
Hence, the time complexity is $O(\frac{1}{2}(n^2 + n)) \rightarrow O(n^2)$.

Example 3:

```
for (i=1; i<=n; i++){              (line 1)
   for (j=1; j<=n; j=j×2){         (line 2)
      sum = sum + 1}               (line 3)
}
```

Let's simply look at the core question:
How many times will line 3 be executed?

$j = 2^0, 2^1, 2^2, 2^3, \dots 2^k$ until k cannot go larger because $2^k <= n < 2^{k+1}$

Hence, assuming n is the **integer power of 2**, then k = log n
Relaxing the assumption to any natural number n, then k = ⌊log n⌋ ≈ log n

So the cost is $\sum_{i=1}^{n}(\log n) = n \log n$!
Finally, the time complexity of this algorithm is O(n log n).

Time complexity of arrays, linked lists, and BSTs for the four common operations:

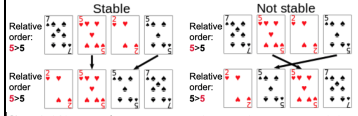
Operation	Time Complexity				
	Array – Unsorted	Array – Sorted	Linked List – Unsorted	Linked List – Sorted	Binary Search Tree
Insert	O(1)	O(n)	O(1)	O(n)	O(log n)
Find/Search	O(n)	O(log n)	O(n)	O(n)	O(log n)
Delete by index	O(1)	O(n)	O(1)	O(1)	O(log n)
Delete by value	O(n)	O(n)	O(n)	O(n)	O(log n)

Due to the need for searching for successors, if deleted nodes has two children

Sorting

- **Computational Efficiency:**
  - Average case Big-O
  - Worst-/Best-case Big-O
- **Storage/Memory Efficiency:**
  - Extra memory consumed beyond the unsorted array

A stable algorithm vs. a non-stable algorithm:



Stability: A sorting algorithm is stable if it preserves the relative order of equal elements.

Insertion Sort

Description: Insertion Sort builds the final sorted array one element at a time, by repeatedly picking the next element and inserting it into its correct position.

Example: Sorting [5, 3, 8, 6, 2]

1. Start with the first element: [5]
2. Insert 3: [3, 5]
3. Insert 8: [3, 5, 8]
4. Insert 6: [3, 5, 6, 8]
5. Insert 2: [2, 3, 5, 6, 8]

Merge Sort

Description: Merge Sort divides the array into halves, sorts each half, and then merges the sorted halves back together.

Example: Sorting [5, 3, 8, 6, 2]

1. Divide: [5, 3, 8, 6, 2] → [5, 3, 8] and [6, 2]
2. Divide: [5, 3, 8] → [5] and [3, 8]; [6, 2] → [6] and [2]
3. Merge: [3, 8] → [3, 8]; [2, 6] → [2, 6]
4. Merge: [5], [3, 8] → [3, 5, 8]; [2, 6] → [2, 6]
5. Merge: [3, 5, 8] and [2, 6] → [2, 3, 5, 6, 8]

Selection Sort

Description: Selection Sort repeatedly selects the minimum element from the unsorted part and moves it to the beginning.

Example: Sorting [5, 3, 8, 6, 2]

1. Select minimum 2: [2, 3, 8, 6, 5]
2. Select next minimum 3: [2, 3, 8, 6, 5]
3. Select next minimum 5: [2, 3, 5, 6, 8]
4. Select next minimum 6: [2, 3, 5, 6, 8]

Quick Sort

Description: Quick Sort selects a 'pivot' element and partitions the array into elements less than and greater than the pivot, then recursively sorts the partitions.

Example: Sorting [5, 3, 8, 6, 2] (pivot: 5)

1. Partition: [3, 2] (pivot) [8, 6]
2. Quick Sort [3, 2] → [2, 3]
3. Quick Sort [8, 6] → [6, 8]
4. Combine: [2, 3], [5], [6, 8] → [2, 3, 5, 6, 8]

Algorithm	Best Time	Average Time	Worst Time	Stable	Memory
Selection Sort	O(n²)	O(n²)	O(n²)	No	O(1)
Insertion Sort	O(n)	O(n²)	O(n²)	Yes	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Yes	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n²)	No	O(log n)

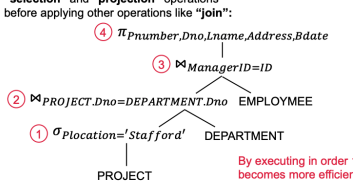
Rules of Relational Algebra Optimization

Outline of relational algebra optimization:

1. Break up selections (with conjunctive conditions) into a cascade of selection operators. Push selection operators as far down in the tree as possible.
2. Rearrange leaf nodes, such that the most restrictive select operators is executed first (depends on knowledge on selectivity).
3. Convert cross products into joins.
4. Move projections as far down as possible.

Follow-up Example

The **improved** query tree after applying **"selection"** and **"projection"** operations before applying other operations like **"join"**:



By executing in order 1-4, this query becomes more efficient.

Note: This tree is still not "optimized"! Why?

Binary Search Trees

Array: Collects elements of the same data type, stored in adjacent memory locations.

Linked List: A series of nodes connected by pointers.

Tree Terminology:

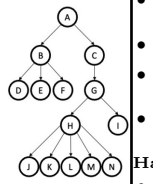
- **Child:** Node directly connected at one level lower.
- **Parent:** Node directly connected at one level higher.
- **Root:** Node with no parent.
- **Leaf:** Node with no children.
- **Sibling:** Another child of the same parent.
- **Ancestor:** Parent or its parent's ancestor.
- **Descendant:** Child or its child's descendant.
- **Subtree:** A node and its descendants.

Depth (of a node): The number of edges you need to travel from root to this node

Example: The depth of H is 3.

Height (of a node or tree): The number of edges in the longest path from this node to a leaf. If this is the height for the whole tree, then it is the number of edges in the longest path from root to a leaf

Example: The height of this tree is 4. The height of B is 5.



Degree (of a node): The number of children a node has.

Example: The degree of B is 3.

Branching Factor: Maximum degree of any node.
N-ary Tree: Branching factor > 2.
Binary Tree: Branching factor of 2.

- **Complete:** Fully packed for its height.
- **Nearly Complete:** Only misses some leaf nodes at the lowest level, filled from left to right.

Binary Search Tree (BST) Properties:

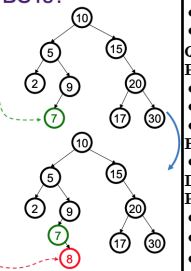
- Branching factor of 2.
- All keys in the left subtree are smaller than the node's key.
- All keys in the right subtree are larger than the node's key.

Largest Node: The largest node in a BST is the right-most node (not necessarily a leaf node).
Predecessor: The predecessor (next smaller node) of any node is the right-most node of its left subtree.
Smallest Node: The smallest node in a BST is the left-most node (not necessarily a leaf node).
Successor: The successor (next larger node) of any node is the left-most node of its right subtree.
Using BSTs is only efficient if they are fairly balanced.
The search time complexity of a BST is associated with its height.

How to Insert A Value into BSTs?

Insertion example 2: Insert 8

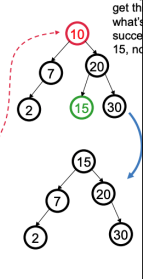
1. Perform search for value X
2. If X is found in tree
   return and do nothing
   Else search will end at a node. Call it Y
   and perform steps 3-4
3. If X < Y
   insert X as new left child of Y
4. If X > Y
   insert X as new right child for Y



How to Delete a Value from BSTs?

Deletion example 5: Delete 10 (different tree!)

1. Perform search for value X
2. If X is not in tree
   return and do nothing
   Else X is in tree, and perform steps 3-5
3. If X has no child,
   delete X
4. Else if X has only one child
   promote this child to X's place.
   delete X
5. Else X must have two children, then
   identify X's successor. Call it Y
   replace X with Y
   delete Y by repeating steps 3-5



Conclusion: All search, insert, and delete operations in BSTs depend on the search operation's time complexity. When a BST is mostly balanced, all operations can achieve $O(\log n)$ time complexity.

Hashing

Setting of Hashing Process

- In DBMSs, hashing deals with the "key" of our data, e.g., the primary key of a table.
- A fixed-sized array is used to store these keys.
- The index in the array is determined by index_in_array $= h(\text{key})$, where $h(\cdot)$ is the hash function.

Hash Collision

- Keys are often integers or convertible to integers.
- Common hash function for integer keys: `h(key) = key % N`, where N is the size of the hash table.
- **Example:** If $N = 4$, then `h(4) = 0`, `h(9) = 1`, etc.
- A hash table of size N can store at most N keys optimally if all keys hash to different indexes.
- **Collision:** When more than one key hashes to the same index.
- **Observation:** Non-uniform key distribution and poor hash function uniformity increase collision likelihood.
- **Quick Fix:** Use a prime number for N to improve hash function uniformity, but collisions can still occur.

Hashing with Chaining

- **Load Factor (λ):** $\lambda = \frac{\text{Number of keys}}{\text{Table size}}$
- Allow $\lambda > 1$ by using an array with linked lists to handle collisions.
- **Chaining:** Keys hashing to the same index are linked together in a list.

Hashing Complexity

Insertion: $O(1)$

Search:

- **Unsuccessful:** $O(\text{hash} + \text{failed search in linked list}) = O(1 + \lambda) = O(\lambda)$
- **Successful:** $O(\text{hash} + \text{successful search in linked list}) = O(1 + 0.5\lambda) = O(\lambda)$

Deletion: Dominated by search time complexity.

Pros and Cons of Chaining in Hashing

- **Pros:** Fewer wasted cells in the hash table; $O(1)$ for handling collisions.
- **Cons:** Search time can become $O(n)$ due to imbalanced, long chains.

Open Addressing

Question: Can we handle hash collisions while keeping $\lambda \leq 1$?
Idea: Add rules to the hash function to move colliding keys to empty spots, known as probing.
Linear Probing: $f(i) = i - 1$
Probing Sequence for Table Size N:

- 1st probe: $(h(x) + 0)\%N = h(x)$
- 2nd probe: $(h(x) + 1)\%N$
- 3rd probe: $(h(x) + 2)\%N$

Quadratic Probing: $f(i) = (i - 1)^2$
Probing Sequence:

- 1st probe: $(h(x) + 0)\%N = h(x)$
- 2nd probe: $(h(x) + 1)\%N$
- 3rd probe: $(h(x) + 4)\%N$

Example: If $h(x) = 2$ and $N = 7$:

- Probing sequence: 2, 3, 6, 4, 5, 1, 0

Double Hashing: $f(i) = (i - 1)h'(x)$
Probing Sequence:

- 1st probe: $(h(x) + 0)\%N = h(x)$
- 2nd probe: $(h(x) + 1 \cdot h'(x))\%N$
- 3rd probe: $(h(x) + 2 \cdot h'(x))\%N$

Example: If $h(x) = 2$, $h'(x) = 3$, and $N = 7$:

- Probing sequence: 2, 5, 1, 4, 0, 3, 6

A good $h'(x)$ should be quick to compute, different from $h(x)$, and never return 0. Common choice: $h'(x) = R - (x\%R)$, where R is a prime number smaller than N.

Rehashing

When the load factor λ becomes too large (over a specific threshold), rehashing is needed:

- Failed insertion in probing due to all indexes being occupied.
- Too many collisions to handle for probing.
- Linked lists becoming too long in chaining.

Rehashing involves rehashing all elements into a new, larger hash table. The threshold for λ is application-specific.

- **Conclusion 1:** Handling hash collisions can be costly.
- **Conclusion 2:** Hashing is useful for equality search (exact matches) but not for range queries or "similar values" (e.g., "LIKE" in string comparison).
- **Conclusion 3:** Hashing does not commonly handle non-key attributes.

Indexing

Types of Indexes

- **Dense Index:** Entry for every search key value in the data file.
- **Sparse Index:** Entries for only some search values.

Whether an index is dense or sparse is based on the types of index:

1. **Single-level** indexes, for which we mainly have

Primary, clustering, and secondary indexes

	Ordered Data	Unordered Data
Key Attribute	Primary Index	Secondary Index
Non-key Attribute	Clustering Index	Secondary Index

2. **Multilevel** indexes
3. Indexes on **multiple attributes**

Primary Index

- Defined on an ordered data file, sorted by a key attribute.
- The i-th entry: $< K(i), P(i) >$
  - $K(i)$: Key field value.
  - $P(i)$: Pointer to the exact record.
- A primary index is a sparse index to ensure fewer rows in the index file.

Advantages of Primary Index

- **Lower Storage Overhead:**
  - Fewer rows and columns.
  - Can be stored in faster memory while raw data is on slower storage.
- **Reduced Search Complexity:**
  - Direct search in a file of size n is $O(n)$.
  - With primary index slicing the data into k parts, the search space is $O(n/k)$.

Clustering Index

- Defined on an ordered data file, sorted by a non-key attribute.
  - $K(i)$: Value of the non-key attribute.
  - $P(i)$: Pointer to the first block with that value.
- A clustering index is a sparse index.

Secondary Index

- Provides a secondary access path in addition to primary indexes.
- A dense index with one entry per record.
- Defined on unordered data files and can be on key or non-key attributes.

Secondary Index on Key Attributes

- $K(i)$: Key attribute value.
- $P(i)$: Pointer to the block or exact value.

The secondary index is ordered.

Secondary Index on Non-key Attributes

- $K(i)$: Non-key attribute value.
- $P(i)$: Pointer to an array of pointers to corresponding records.

The main index file has rows equal to the number of distinct non-key attribute values (same as clustering index).

Tree-Structured Index

The search keys are organized into a tree structure.

- Supports both range and equality searches efficiently.
- Provides a largely reduced search space compared with primary indexing.
- Generally the default extension to primary indexing.

Hash-Based Index

The search keys, with their associated record pointers, are organized into a hash table structure.

Indexes on Multiple Attributes

Justified if data is frequently queried with combinations of the same attributes or the primary key has multiple attributes.

Solution 1: Composite Attribute-based Index

- Combine multiple attributes into a composite one.
- Create an ordered index using lexicographic ordering.
- **Example 1:** If `attribute = [Dno, Age]`, then `[3, 44] < [4, 44] < [4, 45]`.
- **Example 2:** If `attribute = [Age, Dno]`, then `[44, 3] < [45, 2] < [45, 3]`.

Solution 2: Partitioned Hashing

- Use hashing to create index files.
- The key to be hashed consists of n attributes, producing a hash address with n subaddresses.
- Note: Good for equality searches.

Solution 3: Grid File

- Define a grid (2D) array with a linear scale for each search attribute.
- Can handle both equality and range searches.

Relational Algebra and Query Optimisation

Operator	Notation	Description
Selection	σ	Selects a subset of rows from one relation.
Projection	π	Selects wanted columns from one relation.
Set operations	$\cup, \cap, -$	Union, intersection, and set difference for selecting a subset of rows from two relations.
Cartesian product	$\times$	Combines two relations.
Rename	ρ	Assigns a (new) name to a relation or a column.
Join	$\bowtie$	Combines two relations with some constraints.
Assignment	$\leftarrow$	Assigns the result of an expression to a temporary relation.
Division	/	Allows for expressing queries that include a "for all" or "for every" phrase.

Selection $\sigma_p(r)$

- p: selection predicate (condition).
- r: relation.

Projection $\pi_{A1,A2,\ldots,An}(r)$

Set Operations

- $r1 \cup r2$ (union)
- $r1 \cap r2$ (intersection)
- $r1 - r2$ (difference)

```
SELECT Title
FROM   Movie
WHERE  Year = 1944
UNION
SELECT Title
FROM   Movie
WHERE  Year = 1974
```

$\pi_{Title}(\sigma_{Year=1944}(Movie)) \cup \pi_{Title}(\sigma_{Year=1944}(Movie))$

Cartesian/Cross Product $r1 \times r2$

Rename $\rho(E \rightarrow X)$

- E: relation or relational algebra expression.
- X: assigned name.

Rename After Cartesian Product
$\rho((1 \rightarrow \text{Name}, 3 \rightarrow \text{Age}), r1 \times r2)$

Join $r1 \bowtie_c r2$

- c: join condition.
- Result: $r1 \bowtie_c r2 = \sigma_c(r1 \times r2)$
- **Theta Join:** General case of join with conditions.
- Attributes in c can be referenced by position ($r.position$) or by name ($r.name$). Multiple conditions can be linked by "AND" or $\wedge$.

Equi-Join
StarsIn $\bowtie$StarsIn.StarID=MovieStar.StarID MovieStar

Natural Join StarsIn $\bowtie$ MovieStar

- Performed on all common attributes between the relations.
- Common attributes matched automatically.

Query Tree:

- Tree structure representing a relational algebra expression.
- Leaf nodes: relations of the query.
- Internal nodes: relational algebra operations.
- Execution:
  - Start at the lowest level, execute operations when operands are available.
  - Replace internal nodes with resulting relations, move up the tree.
  - Execute the root node for the final result.

Main Heuristic:

- Prioritize operations reducing intermediate result size.
- Apply **selection** (σ) and **projection** (π) before other operations like **join** ($\bowtie$).

General Transformation Rules:

Cascade of σ: A conjunctive selection with conditions C_1 *AND* C_2 *AND* ... *AND* C_n can be broken up into a cascade of individual σ operations.

$$\sigma_{C_1 \text{ AND } C_2 \text{ AND } \ldots \text{ AND } C_n}(R) = \sigma_{C_1}(\sigma_{C_2}(\ldots(\sigma_{C_n}(R))\ldots))$$

Commutativity of σ:

$$\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$$

Cascade of π: In a cascade of π operations, all but the last one can be ignored.

$$\pi_{list_1}(\pi_{list_2}(\ldots(\pi_{list_n}(R))\ldots)) = \pi_{list_1}(R)$$

Note 1: *list* denotes different attribute lists used for projection.
Note 2: "$list_1$" is the last attribute list to be visited in this projection cascade.

Commuting σ with π: If selection condition c involves only attributes in $A_1, A_2, \ldots, An$, then the two operations can be commuted.

$$\pi_{A_1,A_2,\ldots A_n}(\sigma_c(R)) = \sigma_c(\pi_{A_1,A_2,\ldots A_n}(R))$$

Commutativity of $\bowtie$ and $\times$: If all the attributes in the selection condition c involve R, the two operations can be commuted as follows.

$$R_1 \bowtie_c R_2 = R_2 \bowtie_c R_1 \qquad R_1 \times R_2 = R_2 \times R_1$$

Commuting σ with $\bowtie$ or $\times$: If all the attributes in the selection condition c involve R, the two operations can be commuted as follows.

$$\sigma_c(R_1 \bowtie R_2) = (\sigma_c(R_1)) \bowtie R_2$$