# Lecture 1: Database Fundamentals & Relational Model

Contextual database

* Base that can store, update, delete and recall/search data efficiently
* Database – holds data relevant to current contextual activity
  + ex: Web Search – DB with web page links
  + ex: Data Mining – DB managing multidimensional data for discovering patterns, outliers
  + ex: Customer/Retails – profiles, preferences, product (Amazon)

Data Management System

* Software Engineering perspective:
  + Provide software to:
    - **model** data: **relational** (OO/first-order logic/description logics/fuzzy logic/…) **data** modelling
    - **access** data: query for data, insert, delete, update
    - **analyse** data: complex aggregation queries, function approximation, histograms, multi-dimensional visualisation
    - **store** (physically) data: from memory to hard disks
    - **secure** data: control access to sensitive & confidential data, encode/cipher data
  + Maintain data consistency in the face of:
    - failures
      * recovery from failures
  + Optimise data access
    - index and hashing data structures
    - optimisation algorithms
* User perspective:
  + A box with an interface for users/applications offering the discussed functionality:
    - Data Modelling
    - Declarative Programming Language (SQL) to manage & query data
  + Declarative: tell database **what** to do (not how to do it)
* Systems engineering perspective:
  + Diagram

    Description automatically generated

Data

* Families of data:
  + Structured
    - Well-defined data structure, e.g., tables
    - ex: 3 kg (3 – datum, kg – metadata)
  + Unstructured
    - ex: web pages, texts, sensor measurements
    - Less information is provided on interpreting the data
  + Semi-structured
    - ex: XML/JSON
    - Self-descriptive; interpret themselves (medium entropy)
    - <data type=real; unit=’kg’>3</data>
* Modern DMSs manage all families of data

## Conceptual Data Modelling

* Approaches:
  + Entity-Relationship Modelling
    - Does not guarantee optimality in operations and query executions
  + **Relational Modelling**
    - Mathematics-driven: foundation of relational algebra, set theory, functional dependency theory
    - Guarantees query optimisation

Conceptual Data Model

* **Mathematical** model for **interpreting** data
  + Mathematical: theorems from set theory, functional dependency & normalisation theory, relational algebra
  + Interpretation: need to understand context
    - Entities
    - Attributes (characteristics)
    - Relationships

Relational Conceptual Model

* Informally – any entity might relate with any other entity **if and only if** they both share common attributes

Relational Model

* Any entity and any relationship are modelled as a **relation**, which maps to a 2D table
  + an ordered set of **attributes** (columns)
  + a set of **tuples** (rows), which represents instances
  + There exists a **specific** attribute that **uniquely** identifies a tuple in the relation
    - e.g., sequential numbers 1, 2, …
    - e.g., logical values (GUID)
* Query – attributes of interest to be retrieved and constrained attributes to filter out irrelevant tuples
* Formalism
  + Schema of a Relation:
    - **Relation** with name R and an ordered set of **attributes**
    - Each attribute assumes values in a domain
  + Tuple t of R is an ordered set of values corresponding to attributes of R satisfying the domain constraints
  + Instance r(R) – set of tuples
  + NULL: unknown/inapplicable/uncertain/missing value
  + **Relational Database Schema**: set of relations:

[example in lecture L1.3]

Relational Constraints

* Conditions that must hold on all instances for each relation
* Fundamental constraints:
  + **Key constraint** – unique tuple id
    - **Superkey** (**SK**) of relation R is a **set of attributes** containing at least 1 attribute that uniquely identifies any tuple
      * For any 2 distinct tuples:
      * If K is a SK, then any superset of K is a superset
    - **Candidate Key** [K] – **minimal** superkey, i.e., the set with the smallest no. of attributes that uniquely identify tuples
      * i.e., **the removal of any attribute from K results in K’ that is no longer a superkey**
      * A composite set of unique attributes is **not** a candidate key
    - **Primary Key (PK)** – if a relation has several candidate keys, **one** is chosen arbitrarily to be a **primary key** (if multiple attributes – **composite PK**); the rest of the candidate keys are *secondary keys*
      * Convention – underlined in schema
  + **Entity integrity** constraint – keys are never null
    - PK cannot be NULL in any tuple of instance r(R)
    - If PK has several attributes, NULL is not allowed in any of these attributes
    - Note: other attributes may be non-null, artificially constrained by DB designer
  + **Referential integrity** constraint – interpretation of relationships
    - Referencing R1 and referenced R2
    - There exists an attribute **Foreign Key (FK)** in R1 that either has exactly the same value with the Primary Key (PK) in R2 or is NULL
    - If it’s NULL, then the FK in R1 should **not be a part** of its own PK (not violating the entity integrity constraint)
    - Notation: directed arc from FK to PK

# Lecture 2: Functional Dependency Theory, Normalisation Theory

Goodness – whether attributes, e.g., PK, FK, form a good relation

Guidelines for a Good Design

* Attributes of a relation should make sense
  + Attributes of different entities, e.g., students, employees, courses, **should not be in the same** relation
    - Minimise similarity between relations
  + Any relationship between relations should be represented *only* through FKs and PKs
* Avoid **redundant** tuples (repetition of the same info)
  + Impact of repetition:
    - **Storage cost**
    - **Inconsistency cost** & **operation anomalies**
      * replicas must be kept consistent (more checks) during insertion, deletion, update of tuples
    - Replicas result in consistency errors
* Relations should have as **few** NULL values as possible
  + Reasons for NULL:
    - A value is N/A or invalid
    - Unknown
    - Known to exist, but unavailable
  + **Statistics**
    - Attributes that are **frequently NULL** should be placed in separate relations to avoid **wasting storage & having high uncertainty**
  + Ex:
    - DB with employees having 3 contact numbers (only 0.6% having all 3)
    - Make new relation – 3rdPhone (FK to employee, number)
* Design relations to **avoid** **fictitious** tuples after **join**
  + E. Codd’s Intuition – break relation into smaller (sub)relations that share a common attribute
  + Theory needed to split and join relations not to generate fictitious tuples (when joining split relations)

Functional Dependency (FD) Theory

* FD – metric that quantifies the degree of goodness of a relational schema
  + FD – **constraint** derived from the relationship between attributes
  + “Given a relation, an attribute X functionally determines an attribute Y, if a value of X determines a unique value for Y”, Codd, 1970
  + i.e., for a value of X, you can uniquely identify a value of Y (**X determines Y**)
  + If, whenever 2 tuples have the same value for X, they must have the same value for Y
* Principles
  + Lemma 1: If K is a **Candidate Key**, then K functionally determines **all** attributes in relation R, i.e.,
  + Lemma 2: William W Armstrong’s inference rules:
    - (Reflexive) If , then
    - (Augmentation) If , then
    - (**Transitive**) If and , then

Normalisation Theory

* Transforms a relational schema into a set of good and efficient relations
* Algorithm:

1. Assert FDs among attributes
2. Take a pool and put into all the asserted FDs
3. Create a universal big relation with all attributes
4. Recursively decompose the big relation based on the FDs into many smaller ones without information loss and without fictitious tuples on rejoining

* Prime attribute – attribute that belongs to some candidate key of the relation
* Degree of decomposition:
  + First Normal Form (1NF)
    - Domain of each attribute in a relation refers only to atomic (simple/indivisible) values
    - Disallows:
      * Nested attributes
      * Multivalued attributes
  + Second Normal Form (2NF)
    - **verbosity in the primary key X**
    - **Full FD** means that, if any prime attribute A is removed from the PK X, then (**X without A does not determine Y**)
    - A relation R is in 2NF if **every** non-prime attribute A in R is fully functionally dependent on the PK of R
      * Target from 1NF to 2NF: remove all prime attributes from the PK which cause partial dependencies
    - Methodology:

1. Identify all the partial FDs in the original relation (1NF)
2. For each partial FD, create a new relation s.t. all non-prime attributes in there are fully functionally dependent on the new PK
   1. i.e., the prime attribute in the original relation causing partial FDs
3. The new relation(s) will be in 2NF
   * Third Normal Form (3NF)
     + Focuses on **transitivity**
     + A relation R is in 3NF (being already in 2NF) if there is **no non-prime attribute** which is **transitively dependent** on the PK
     + All non-prime attributes should only be directly dependent on the PK
     + Methodology:
       - Split the original relation into 2 relations: the **non-prime transitive attribute**:
         * is the **PK of the new** relation
         * is the FK in the original relation referencing the new one
     + Generalised 3NF
       - If the transitive attribute is prime, no violation of 3NF
       - Every non-prime attribute A in relation R:
         * is **fully** functionally dependent on **every** candidate key in R
         * is **non**-transitively dependent on **every** candidate key in R
   * …
   * Boyce-Codd Normal Form
     + Remove **all inherent dependencies**: any attribute should be functionally dependent only on the PK
     + A relation is in BCNF if and only if, whenever there exists a , then X is a PK
     + Decomposition:
       - Goal: avoid fictitious tuples
       - BCNF Decomposition Theorem:
         * **Theorem 1**: Let relation R not in BCNF and let be the FD which causes a violation in BCNF

Then R should be decomposed into 2 relations:

R1 with all attributes except A

R2 with attributes X and A

If either R1/R2 is not in BCNF, repeat

# Lecture 3: SQL & Advanced SQL

SQL – Structured Query Language

Declarative language

* Declare what to do, not how

Standard: SQL-92 (advice, follow this)

Database schema:

* CREATE SCHEMA [name]; (notice the semicolon)

Create relation (table)

* CREATE TABLE [name] …;
* Specify name
* Specify attributes
  + Domains
    - Numeric data types
      * INT
      * REAL / DECIMAL (n, m)
    - Character/String data types
      * Fixed length: CHAR(n)
        + Exactly n
      * Variable length: VARCHAR(n)
        + From 0 to n
    - Bit-string data types
      * Fixed length: BIT(n)
      * Varying: BIT VARYING(n)
    - Boolean
      * TRUE, FALSE, NULL (yes, no, maybe)
    - DATE
    - etc (TIMESTAMP, DATE INTERVALS, …)
  + PK/FK
* Specify constraints
  + Default
    - DEFAULT
    - NOT NULL
  + CHECK (range)
  + Key
    - unique PK (can’t be null)
    - UNIQUE – specifies candidate keys
  + Referential
    - FK(…) REFERENCES [table]([attribute])
    - Triggered actions: [ON DELETE / ON UPDATE] [SET NULL / SET DEFAULT / CASCADE]
      * CASCADE – propagates DELETE/UPDATE to all referential tuples

Set – only unique elements

Multiset – might have duplicates

**Select**

* SELECT-FROM-WHERE
* SELECT <attribute list>
  + what to retrieve – **attributes of interest**
  + \* - all attributes
  + DISTINCT – no duplicates
* FROM <table list>
  + TABLE as a Variable
    - AS
    - ex: EMPLOYEE AS e, EMPLOYEE AS s
    - **A relation** might play **different roles** within query (as above), e.g., supervisor and supervisee
* WHERE <condition>
  + logical statements involving OR, AND, NOT
  + If missing:
    - No condition
    - If FROM involves 2+ relations, never miss!
* Join & Select
  + More than 1 table
  + Needs [FK] = [PK] clause in an AND clause
* UNION, INTERSECT, EXCEPT

3-valued Logic

* TRUE (1), FALSE (0), and NULL (0.5)
* Principle – any value compared with NULL evaluates to UNKNOWN
  + Use IS NULL or IS NOT NULL

Nested (Inner) Query

* Outer SQL and Nested SQL
* Nested query’s output is input to outer’s WHERE via: IN, ALL, EXISTS
* Types:
  + Nested Uncorrelated Query: **first**, execute nested query, **then** execute outer
    - Searched once
    - IN
      * Checks whether a value belongs to the inner’s output (multi)set
    - ALL
      * Compares a value with all the values from inner’s output set using >, >=, <, <=, =, <> (is not equal to)
  + Nested Correlated Query: **for each** tuple of the outer query, we execute the nested query
    - Searched n times
    - Relation variable: Global scope (outer) and local scope (inner)
    - Lemma 1: Correlated queries using IN are collapsed into 1 block
      * i.e., possible to get rid of nested nature if using IN
    - EXISTS
      * Checks whether the inner’s output is an empty set or not, and returns FALSE or TRUE, respectively
      * Opposite: NOT EXISTS

Worked example:

* Retrieve names of all students who have an A in all their courses
  + SELECT S.Name  
    FROM STUDENT AS S, GRADES AS G  
    WHERE G.GRADE = ALL (SELECT G.GRADE  
    FROM
* No non-A courses:
  + SELECT S.Name  
    FROM STUDENT AS S,  
    WHERE NOT EXISTS (SELECT G.Grade  
    FROM GRADES AS G  
    WHERE G.StudentID = S.StudentID AND Grade <> ‘A’)

# Week 4: Advanced SQL & Analytics

Join

* INNER JOIN
  + Matches tuples using FK and PK (THETA-JOIN)
  + The matching operator is usually ‘=’ – EQUIJOIN
  + No need for ‘FK = PK’ in WHERE clause
  + FK cannot be NULL
* OUTER JOIN
  + Use ON instead of WHERE
  + LEFT OUTER JOIN
    - **Every tuple** in the **left** **relation** must appear in **result**
    - If no matching tuple exists, just add NULL values for attributes of right relation
  + RIGHT OUTER JOIN
    - Same, but for right

Aggregate Function

* Statistical summary/value over group of tuples
* Built-in:
  + COUNT(\*): counts total
  + SUM(X)
  + MAX(X) / MIN(X)
  + AVG(X)
  + CORR(X, Y): correlation between two values
  + (NULL values discarded except from COUNT)
* In SELECT clause
* Can rename with AS

Analytics: Grouping Tuples

* **Partition** a relation into groups based on **grouping attribute**, i.e., clustering tuples having the same value in the grouping attribute
* GROUP BY {grouping attribute}
  + Need to SELECT the grouping attribute
  + Clause at the end
* Histogram
  + GROUP BY attribute, then COUNT(\*)
* Regression Analytics
  + GROUP BY attribute, then AVG(different attribute)
  + Approximate dependency of different attribute on attribute
* Worked example A1:
  + SELECT D.NAME, AVG(E.Salary)  
    FROM EMPLOYEE AS E INNER JOIN DEPARTMENT AS D  
    (WHERE E.DNO = D.DNUMBER)  
    GROUP BY E.DNO (or D.DNAME)
* **HAVING**: condition to **select**/reject a group **after** grouping
  + ex: show no. of employees per project only if there’s >2 employees
  + After GROUP BY
  + Condition can have aggregate function
* Worked example A2:
  + SELECT M.LNAME  
    FROM EMPLOYEE AS M, DEPARTMENT AS P  
    WHERE M.SSN = P.MGRSSN AND   
    PDNUMBER IN (SELECT E.DNO  
     FROM EMPLOYEE AS E  
     GROUP BY E.DNO  
     HAVING COUNT(\*) > 100)

Problem – WHERE executes before GROUP BY

* Filters out values before aggregating
* Solution: nested query – first GROUP BY, then WHERE … IN {inner query}

Worked Example A3

* Possible multiple same MAX()
* SELECT DNO, COUNT(\*)  
  FROM EMPLOYEE  
  GROUP BY DNO  
  HAVING COUNT(\*) = (SELECT MAX(A.Members)  
   FROM (SELECT D.DNO, COUNT(\*) AS Members  
   FROM EMPLOYEE AS D  
   GROUP BY D.DNO))

# Week 5: Physical Design, Indexing & Hashing

Physical Storage Hierarchy

* 3-level:
  + Primary storage
    - RAM: main memory, cache
  + Secondary storage
    - HDD, SSD
  + Tertiary storage
    - Optical drives
* Going down:
  + Storage capacity: increases
  + Access speed: decreases
  + Money costs: decrease

Physical storage for a DB:

* Default – secondary storage (too big for primary)
  + Consequences:
    - Data loaded into main memory, then processed
    - Speed of access is low
  + Challenge: Organise data in HDD to minimise latency

Organisation-based Optimisation

* Organise tuples on disk to minimise I/O access cost
  + Representation:
    - Record – Tuple
      * Can be of:
        + **Fixed length**
        + Variable length
    - Block – group of Records
      * Fixed length
      * No. of records stored in a block: **blocking factor**
        + B – bytes in block
        + R – bytes in record
    - File – group of Blocks
* Blocks to Files on Disk
  + Linked allocation
    - Each block i has a pointer to the physical address of the logically next block i+1 anywhere on the disk (i.e., a linked list of blocks)
  + No. of blocks in file:
    - r – bytes in relation

File Structures

* **Heap File**
  + unordered
  + Principle: a new record is added to the end of the file
  + Inserting – efficient
    - Load last block
    - Insert at end
    - Complexity: 2 block accesses – **O(1) (constant) block access**
  + Retrieving – inefficient
    - Linear search through all file blocks
    - Complexity: **O(b)** block accesses
  + Deleting – inefficient
    - Find and load
    - Remove record, write block back
    - Leaves unused spaces
    - Complexity: **O(b)+O(1)**
    - Use deletion marker
      * Periodically reorganise file by gathering non-deleted records
* **Sequential File**
  + ordered
  + Principle: records are kept physically sorted w.r.t. ordering field
  + Suitable for SQL queries that:
    - Require sequential scanning
      * ex: ORDER BY [ordering field]
    - Involve ordering field in search
      * ex: WHERE PK LIKE …
    - Range queries over the ordering field
      * ex: WHERE PK > … AND PK < …
  + Retrieving
    - Using ordering field – efficient
      * Using binary search [example, pseudocode in slides]
      * Complexity: **O(log2 b)** (sublinear with b)
    - Using non-ordering field – inefficient
      * Same as retrieving
      * Complexity: **O(b)**
    - Range queries – efficient
  + Inserting – expensive
    - Locate block where to insert: binary search
    - On average, half of the records need to be moved to make room
    - Alternative: chain pointers
      * Principle: pointers with each record
      * Pointers must be updated: sorted-linked list
  + Deleting – expensive
    - Locate block where to delete: binary search
    - Update deletion marker, update pointer
    - Periodically resort file to restore sequential order
  + Updating
    - On ordering field – costly
    - On non-ordering field – efficient
* **Hash File**
  + Principle: a hashing function y=h(x) is applied to each record field x (hash field)
  + Output y is the physical block address; mapping a record to a block
  + Hashing:
    - partition records into M buckets
    - Each bucket can have 1+ block
    - Choose hash function y=h(k) with output y as one of the buckets
    - Requirement: h uniformly distributes records into the buckets, i.e., for each value k, each bucket is chosen with equal probability 1/M
    - External hashing: mapping record to a bucket (over hash-field k)
    - Normally, collisions occur
    - Indirect clustering: group tuples together w.r.t. their hashed-values y and not w.r.t. their hash-field values k
    - Algorithm:

1. Hash k and get the corresponding bucket
2. Use hash map to get the block address in disk of the bucket
3. Fetch block from disk to memory
4. Linear search in memory to find record with bucket
   * Retrieving – efficient?
     + O(1)?
     + External Hashing Collision Problem
       - Solution: Use chain pointers
     + Retrieving with collision solution
       - **O(1) + O(n)** 
         * n – number of overflow blocks (n < b)
     + Range queries – inefficient
   * Deleting
     + If in main bucket, O(1)
     + Else, O(1) + O(n)
   * Updating
     + O(1) or O(1) + O(n)
   * Expected cost is unpredictable

Expected I/O Access Cost

* Fix a file type heap/ordered/hash
* Cost for:
  + **Retrieving** whole **block** from disk to memory (search cost)
  + **Inserting/Deleting/Updating** record by transferring **whole block** from memory to disk (update cost)
* **Cost Function**: **expected number of block accesses** (read/write) to search/insert/delete/update a record
* Assign each value x a specific real-valued function C(X) indicating the cost for accessing X in number of block accesses

If at least 79% of the queries involve k, then hash file w.r.t. k; otherwise, sort file by k

# Week 6: Indexing Methodology I (Indices)

Index – alternative access path using any field

Index design –

* Objective – given **any** file type, provide a secondary access path using more than 1 searching field, e.g., SSN, Salary, Name, etc
* Cost: Additional files (metadata) & maintenance
* Benefit: Expedite search process avoiding Linear Scan
* Trade off: Overhead vs Search Speed
  + primary access (hash, heap, sequential) – slow but no overhead
  + secondary – fast but storage and maintenance

Principles:

1. Create 1 index over 1 field: **index field**
2. An index is *another separate* **file**
3. All index entries are **unique & sorted** w.r.t. index field
   1. index-entry = (index-value, block-pointer)
4. First search within index to find block-pointer, then access data block from data-file

Facts:

* Index file occupies fewer blocks than the data-file
  + **Dense Index**: an index entry for *every* record in the file
  + **Sparse Index**: index entries only for some of the records
* Searching over index is faster than over file
  + Index – ordered file
  + Binary-based and tree-based methods

Index Types

* Primary Index
  + index field is ordering, key field of a sequential file
  + ex: SSN, file sorted by SSN
* Clustering Index
  + ordering, non-key field of a sequential file
  + ex: DNO, file sorted by DNO
* Secondary Index:
  + index file is:
    - non-ordering, key field
      * ex: unique passport no, over an ordered (by SSN) or non-ordered file
    - non-ordering, non-key field
      * ex: salary, over ordered or non-ordered file

Primary Index

* Ordered file over ordering, key k of a sequential data-file
  + fixed length index entries: pair
    - ki – unique value of index field
    - pi – pointer to i-th block containing record with key ki
* Sparseness: 1 index-entry per data block

Clustering Index

* Challenge: Index seq file on an ordering, non-key field (ex: create index on EMPLOYEE ordered by DNO)
* Idea: file is a set of clusters of blocks; a cluster per distinct value
* index-entry = (distinct-value, block-pointer)
  + 1 index-entry per distinct clustering value
  + Block pointer points at the 1st block of the cluster. The other blocks of the same cluster are contiguous and accessed via chain pointers

When to create a clustering index:

* A clustering index of m<b blocks is created over an ordering non-key field if an only if:
* If , i.e., infinite number of distinct values, then the linear search over an ordering non-key field with exiting feature is bounded by b/2, i.e., half of the naïve linear search:

Secondary Index

* Challenge: index a file on a non-ordering field. File might be unordered/hashed/ordered but not ordered w.r.t. the indexing field
* Cases:
  + S1: Secondary index on a non-ordering, key field, e.g., SSN
    - 1 index entry per data record (dense index)
      * Can’t use anchors because it’s not ordered
  + S2: Secondary Index on a non-ordering, non-key field, e.g., DNO
    - Group the block addresses of those records having the same value
    - Assign an index entry per group (cluster) of block addresses
      * index-entry = (distinct-value, cluster-pointer)
    - **Cluster pointer** points to:
      * (Level 1) a block of [block-pointers] of a cluster
      * (Level 2) a block-pointer points to the data-block that has records with this distinct index value

Properties of all index methods:

* **Ordered** on indexing field
* Indexing field has **unique** (distinct) **values**
* Each index entry is of **fixed** **length**

Multi-Level Index

* Index of an index
  + OG index file: **base**/**Level-1 index**
  + Additional index: **Level-2 index** (index of an index)
  + >2 – **Level-t** index
* Challenge: find best Trade-off of speed vs overhead regarding amount of indices
* Theorem: Given a Level-1 Index with blocking factor m entries/block, the multi-level index is of **maximum level** 
  + m – **fan-out**
  + Scalable Design: independent of data size
  + In any SQL/NoSQL system, always adopt multi-level indexed for data access
* Amazing gain: 1 block/level plus 1 data-block = **t+1 = ceil(logm(b)) + 1 block accesses**

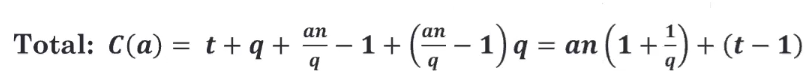
# Week 7: Indexing Methodology II (B Trees)

Multi-level index

* Search: t+1
* Insertions, deletions, updates are costly
  + Because changed in all levels
* Define dynamic multi-level indices
  + adjust to delete/insert
  + expand/shrink
  + self-balancing (sub-trees – same depth)
* Proposal: B Tree, B+ Tree, etc

Multi-level Index as a tree

* Limitation:
  + Unbalanced
    - Not adjusted to distribution (leaf-nodes are at different levels)
    - Higher expected search time – O(t) (worst case)
* B-Tree index on non-ordering key
  + Node order p: splits the searching space up to p subspaces (p > 2)
  + Includes:
    - key values sorted K1 to K­p-1
    - block/data pointer Q
    - tree pointer P pointing to sub-tree
  + Stores too much meta-data
    - data-pointers to blocks (addresses)
    - tree-pointers to tree nodes (structural metadata)
    - search key values (data values)
  + fan-out – splitting factor of the search space
* B+ Trees
  + Internal Nodes: guide searching process (super-fast)
  + Leaf Nodes: point to actual data blocks (Access point)
  + Principles:
    - Internal Nodes have no data-pointers to maximise fan-out
    - Only leaf Nodes gold the actual data pointers
    - Leaf Nodes hold **all** the key values sorted and their corresponding data pointers
    - Some key values are replicated in the Internal Nodes to guide & expedite search process (corresponding to medians of key values in sub-trees)
  + Diagram

    Description automatically generated
  + Pnext – points to sibling node
  + General:
    - Leaf Nodes are linked & sorted by key
    - All keys of the file appear at the Leaf nodes!
    - Leaf nodes contain data-pointers only (expedite navigation)
    - Leaf nodes are balanced (constant I/O cost)
    - Some selected keys are replicated in the internal nodes
  + Better:
    - Higher fan-out => more index entries => quicker search process
    - Stores more entries
  + Decision Rule for use:
    - 
    - C(a) < b (linear search cost)
    - ex: if range ratio is less than 90.6%

# Week 8: Query Processing

Fundamental: Sorting

* Almost all SQL queries involve sorting of tuples (defined by user
* ex: ORDER BY, GROUP BY, DISTINCT all involve sorting
* Limitation: cannot store entire relation into memory for sorting
* **External sorting**: sorting algorithm for large relations stored **on** **disk** that don’t entirely fit in main memory
  + Principle: Divide & Sort (Conquer)
    - Divide: file of b blocks divided into L smaller sub-files (b/L blocks each)
    - Sort: load each small subfile to memory, sory and write back to disk
    - Merge: merge 2 sorted subfiles
  + Expected cost:
    - b – file blocks
    - M – degree of merging (no. of sorted blocks merged in each loop)
      * M = 2 – worst performance
        + Merge in parallel only a pair of blocks at each step
      * M > 2
        + M-way merging
    - L – no. of initial sorted subfiles (before merging)

## Strategies for SELECT:

* Linear Search over a key
  + Expected cost: b/2
* Binary Search over key
  + Expected Cost (unsorted):
  + Expected Cost (sorted):
* Primary Index/Hash Function over key
  + Precondition (Index): Primary Index of level t over key (sorted by key)
  + Precondition (Hash): File hashed with the key
  + Expected Cost (sorted): t+1
  + Expected Cost (hashed): 1 + O(n)
    - n – overflown buckets
* Primary Index over key involved in a range query
  + Use Index to find record satisfying **equality**, then retrieve all subsequent blocks from ordered file
  + Precondition: Primary Index of level t over the key (sorted by key)
  + Expected Cost (sorted): t + O(b)
  + (don’t use Hashing!)
* Clustering Index over ordering, non-key
  + Precondition: Clustering Index of level t on non-key (sorted by non-key)
  + Expected cost (sorted): t + O(b/n)
    - n – distinct values of non-key attribute
    - Attribute is uniformly distributed
* Secondary Index (B+ Tree) over non-ordering key
  + Precondition: not ordered by key
  + Expected Cost: t + 1
    - B+ Leaf Note points at unique block
* Secondary Index (B+ Tree) over non-ordering, non-key
  + Precondition: Not ordered by non-key
  + Expected Cost: t + 1 + O(b)
    - B+ Leaf Node points to a block of pointers to data blocks

Strategies for Disjunctive (OR) SELECT:

* **IF** an access path exists (B+/hash/primary index) for all attributes:
  + use each to retrieve set of records satisfying each condition
  + union all sets
* **ELSE** if none/some attributes have access path, **linear search**

Strategies for Conjunctive (AND) SELECT:

* Methodology:
  + **IF** and access path (index) exists for an attribute, use it to retrieve the tuples satisfying the condition (**intermediate result**)
  + GO through this intermediate result to check which record also satisfies the other condition(s)
  + [i.e., do one, then the other]
* If 2 indexes:
  + Use first the one that generates **smallest** intermediate results
  + **Optimisation**: find the execution sequence of conditions that minimises expected cost
  + Principle: Predict **selectivity** beforehand

## Strategies for (EQUI-) JOIN:

* Naïve join
  + No access path
  + Algorithm:

1. Compute Cartesian product of both relations (all tuples)
2. Store result and for each concatenated tuple, check if same
   * **Inefficient**, typically result is a small subset of Cartesian
   * Cost (block accesses):

* Nested loop
  + No access path
  + Algorithm:

1. For each tuple r in R (outer)
   1. For each tuple s in S (inner)
      1. If same, then add to result file
      * Text

        Description automatically generated
   * Loops are over **blocks**, not tuples
   * Reform pseudocode in block-centric programming mode (system programming using files)
   * Optimisation: choosing inner/outer
     + Choose file with fewer blocks as outer
   * Expected Cost (block accesses):
     + nO – blocks in outer relation
     + nI – blocks in inner relation
     + nB – blocks available in memory

* Index-based nested-loop
  + index; B+ Trees
  + Use index on A/B joining attributes (R.A = S.B)
  + Assume an index I on attribute B of S
  + Algorithm:

1. For each tuple r in R
   1. Use index of B: I(r.A) to retrieve all tuples s of S with equality
   2. For each such tuple s in S, add matching tuple to result file
   * Faster than nested-loop because of immediate access on s (r.A=s.B) by searching for r.A using index I, avoiding linear search on S
   * Optimisation: which index to use?
     + Use the index built on the PK (not on recursive relationships)
   * Expected Cost (block accesses):
     + nI – blocks in indexing relation
     + rI – tuples in indexing relation
     + xN – level of non-indexing relation

* Sort-Merge
  + Sorted relations
  + Precondition: R and S are physically ordered on their joining A and B
  + Algorithm:

1. Load a pair (R.block, S.block) of sorted blocks into memory
2. Both blocks are linearly scanned concurrently over joining attributes (sort-merge in memory)
   1. Advance the one that’s smaller
3. If matching tuples found, store in buffer
   * Pro: blocks of each file scanned only once
   * Con: needs to be sorted first (if not, sort it)
   * Cost (block accesses):
     + If sorted:
     + If not, with external sorting:

* Hash
  + Hashed relations
  + Preconditions
    - File R partitioned into M buckets w.r.t. hash function h over joining attribute A
    - File S also portioned with same h over attribute B
  + Assumption: R is smallest file and fits in memory: M buckets of R are in memory
  + Algorithm:

1. (Partitioning) For each tuple r in R
   1. Compute y = h(r.A) (address of bucket)
   2. Place r into bucket y in memory
2. (Probing) For each tuple s in S
   1. Compute y = h(s.B) (same hash)
   2. Find bucket y in memory (of R partition)
   3. For each tuple r in R in bucket y
      1. If s.B = r.A, add to result (join)
   * Cost (block accesses):
     + Best Case: Memory
       - nS – blocks in smallest relation
       - Cost:
     + Usual Case (smallest relation cannot fit in memory):

# Week 9: Query Optimisation I

Input: Query  
Output: Optimal *execution* plan

Types:

* Heuristic Optimisation
  + Transform an SQL query into an equivalent and efficient query using Relational Algebra
* Cost-based Optimisation
  + Task 1: Provide *alternative* execution plans and *estimate* (predict) their costs
  + Task 2: Choose the plan with the **minimum cost**
  + **Cost Function** c(x1, x2, …) with optimisation parameters:
    - **# block accesses**
    - **memory requirements**
    - CPU computational cost
    - network bandwidth
    - …
  + Exploit: statistical information to estimate the execution cost of a query
  + Information for each Relation:
    - number of records (r); average size of each record (R)
    - number of blocks (b); blocking factor (f), i.e., records per block
    - Primary File Organization: heap/hash/sequential file
    - Indexes: primary, clustering, secondary index, B+ Trees
  + Information for each Attribute A of each Relation
    - **Number of Distinct Values** (NDV) **n** of attribute A
    - Domain range: [min(A), max(A)]
    - Type: continuous or discrete attribute; key/non-key
    - Level t of Index of the attribute A, if it exists
    - **Probability Distribution Function P(A=x)** indicates the frequency (probability) of each value x of the attribute A in the relation
      * A good approximation of a distribution: **histogram**

Selection Selectivity

* sl(attribute A) – real number (0% to 100%)
* = 0: **none** of the records satisfy a condition over attribute A
* = 1: **all** of the records satisfy a condition over attribute A
* = x; x% of the records satisfy a condition over attribute A
* Probability that a tuple satisfies a selection condition

Selection Cardinality

* Challenge 1: Given r tuples and a selection condition over A, predict the number of tuples satisfying this condition **without scanning the file**
* Average number of tuples satisfying a condition over attribute A
* Assumptions are almost always made

Selectivity Prediction

* Solution 1 (no assumption; approximation)
  + Approximate the distribution values via a histogram
  + Pro: accurate selectivity estimate
  + Con: maintenance overheads
* Solution 2 (uniformity assumption)
  + **Assumption**: All values are **uniformly distributed** (equiprobable, thus, no histogram
  + Pro: no need to maintain (update) a histogram
  + Con: provide a less accurate prediction for sl(A)
  + **sl(A=x) constant independent of the x value**
  + If A is a **key** attribute, then a good estimate is
    - since only 1 tuple satisfies the condition; selection cardinality s = 1 tuple
    - **Selection cardinality** = 1 tuple
  + If A is a **non-key** attribute with n = NDV(A) (number of distinct values) <= r, then a not-so-good estimate is:
    - Because all records are **uniformly distributed** across the n distinct values
      * r/NDV(A) = r/n: number of tuples having a distinct value
      * Fraction: P(A=x) = (r/n)/r = 1/n
    - **Selection cardinality** > 1 tuple
  + **Text

    Description automatically generated**
  + BUT: Probability of an attribute having uniform distribution is **almost zero**

Range Selection Selectivity

* SELECT \* FROM RELATION WHERE A x
* **Domain range**: max(A) – min(A)
* **Query range**: max(A) – x
  + query range over domain range

Conjunctive Selectivity

* SELECT \* FROM RELATION WHERE (A = x) AND (B = y)
* Assumptions:
  + All values of A are uniformly distributed
  + All values of B are uniformly distributed
  + A and B are independent

Disjunctive Selectivity

* SELECT \* FROM RELATION WHERE (A = x) OR (B = y)
* Same assumptions as conjunctive selectivity
* from:
  + probability of an employee satisfying condition A or condition B; both are independent

Selection Cost Refinement

* Target: be more accurate by **expressing cost as a function of sl(A) = r/n**
* SELECT \* FROM RELATION WHERE A = x
* Context: b blocks, f blocking factor (tuples per block), r records, n = NDV(A)
* Binary Search where the relation is sorted w.r.t. A
  + If A is a **key**, then **Expected Cost: log\_2(b)** block accesses (ind. sl(A))
  + If A is **not a key**:
    - log\_2(b) block accesses to reach first block with record A=x
    - Access all contiguous blocks that satisfy A=x
    - Selection cardinality: tuples
    - Blocking factor: f tuples per block: access ceil(s/f) – 1 more blocks
    - **Expected Cost: log\_2(b) + ceil(s/f) – 1** = **log\_2(b) + ceil(r\*sl(A)/f) – 1**
* Multilevel Primary Index of level: t over the **key** A
  + equality A=x
    - **Expected Cost: t + 1**
  + Range query Ax
    - Tree traversal: t block accesses
    - Range selection cardinality: s = r\*sl(A)
    - Blocking factor: f records per block: ceil(s/f) blocks
    - **Expected Cost: t + ceil(s/f)** **= t + ceil(r\*sl(A) / f)** block accesses
    - (sl(A) – *range* selection selectivity)
* Hash File Structure (key with equality)
  + Apply the hash function h(A) over the **key** A and retrieve the block
  + **Expected Cost: 1**
    - best case; no overflown buckets
* Clustering Index over an **ordering**, **non-key** attribute:
  + Tree traversal: t
  + Selection cardinality: s = r\*sl(A)
  + Blocking factor: f: ceil(s/f) blocks
  + **Expected Cost**: **t + ceil(s/f)** **= t + ceil(r\*sl(A) / f)**
  + **Fine-grained Cost: t + ceil()**
* B+ Tree over a **non-ordering** attribute with equality A=x
  + Key:
    - Tree traversal: t
    - One data block since s=1
    - **Expected Cost: t + 1**
  + Non-key
    - Tree traversal: t
    - 1 block access to load the block of block pointers
    - Selection cardinality s = r\*sl(A) tuples
      * Each tuple may be in a different data block (worst case); thus, up to s blocks
    - **Expected Cost: t + 1 + s** = **t + 1 + r\*sl(A)**

# Week 10: Query Optimisation II

For relations E and D:

* Cartesian cardinality = |E x D| = |E|\*|D| (without a WHERE clause)

**Join cardinality** = |ED| (with a WHERE clause)

* Alternative:

**Join selectivity** (js)= |ED| / |E x D| (probability of selecting a matching tuple out of all tuples in the Cartesian space)

* Fraction of the matching tuples between the relations R and S out of the Cartesian cardinality (no. of concatenated tuples)

Join Selectivity Theorem

* Given n=NDV(A, R) and m=NDV(B, S):
* [Proof not needed]

Result blocks after join: **k = (js \* |R| \* |S|) / f\_RS**

Join Cost Refinement

* Strategy: **Nested-Loop Join**
  + SELECT \* FROM EMPLYEE E, DEPARTMENT D  
    WHERE E.SSN = D.MGR\_SSN
  + D is the outer relation (b\_D < b\_E):
  + **Expected Cost**:
  + Matching tuples: jc
  + Number of result blocks: k
  + Include k result-block **writes** from memory to disk during the execution
  + **Refined Expected Cost:**
* Strategy: **Index-based Nested-Loop Join**
  + Primary Index on MGR\_SSN with x\_D levels
  + Case: **Primary Index** on **ordering/key**:
    - **Refined Expected Cost**:
  + Case: **Clustering Index** on **ordering/non-key**:
    - Selection cardinality s\_E
    - Blocks of employees per department: ceil(s\_E/f\_E)
    - Number of result blocks k
    - **Refined Expected Cost**:
  + Case: B+ Tree Index on non-ordering/non-key
    - 1 block (block of pointers) + blocks of employees: s\_E
      * worst case: each employee belongs to a different block
    - Number of result blocks k
    - Refined Expected Cost:
  + (Can rewrite s\_E as n, js as 1/max(n, m))
* Strategy: **Sort-Merge** (both files are sorted on A and B)
  + **Refined Expected Cost**:
* Strategy: **Hash-Join** (both files are hashed with same hash)
  + **Refined Expected Cost**:

3-Way Optimisation

* SELECT \* FROM EMPLOYEE E, DEPARTMENT D, DEPENDENT T  
  WHERE T.E\_SSN = E.SSN AND E.SSN = D.MGR\_SSN
* [example in slides]