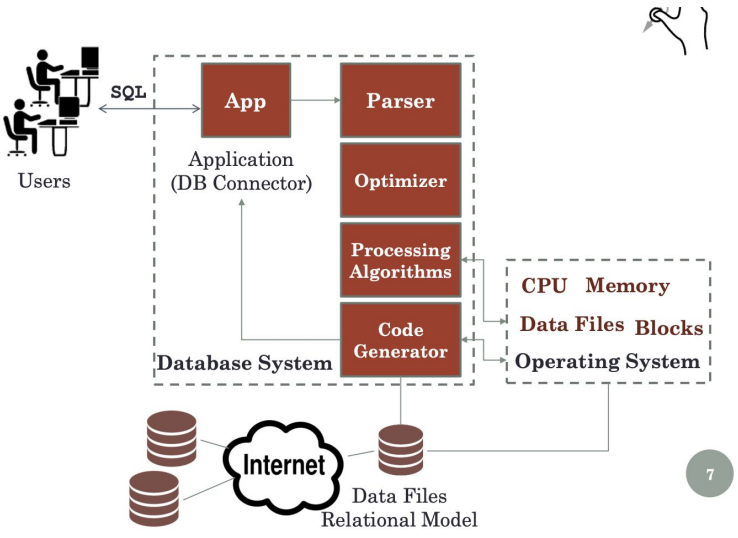
Database Fundamentals & The Relational Model

Contextual Database

A database holds data relevant to our current contextual activity. The **fundamental functionality** is to provide software to:

- **model data**: relational data modeling, OO data modeling, first order logic data modeling, fuzzy logic data modeling, etc
- **access data**: query for data, insert, delete and update
- **analyze data**: complex aggregation queries, function approximation, histograms, multi-dimensional visualization, outliers detection, etc
- **physically store data**: from memory to hard disks
- **secure data**: control access to sensitive and confidential data, cipher and encode data
- **maintain data consistency**: in the face of failures; recover from failures
- **optimise data access to efficiently retrieve data**: index and hashing data structures, optimisation algorithms

Database System Abstraction



A box with an interface for users/applications offering the fundamental functionality. It uses a declarative programming language (SQL) to manage and query data. Declarative means we tell the database what to do, but we do not care how it does it; due to the support of the fundamental functionality, a well-designed database system will take care of the how. Data abstraction generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail.

A **data model**—a collection of **concepts** that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.

Data

There are **three families of data** we distinguish:

1. **Structured data**: well-defined data structure for metadata, e.g., tables. For example, 3 Kg - 3 is the datum, and Kg is the metadata for it.
2. **Unstructured data**: e.g., web-pages, texts, sensor measurements. It has high entropy as less (or even no) information is provided for the data
3. **Semi-structured data**: e.g., XML or JSON documents. The data self-interprets itself - medium entropy.

Traditional DBSs are designed to manage structured data, since we require metadata to manage data and create the database schema.

Database schema is the description of the database, which is specified during design and is not expected to change frequently. Modern DBSs manage all families of data.

Conceptual Data Modelling

Data modelling requires to **transform a textual description into a set of concepts conveying exactly the same information in order to achieve abstraction**.

Entity-Relationship Modelling (Craig's shitty databases lol)	Relational Modelling YASSS 🌟🌟🌟
Invented by Prof P Chen; 1976	Invented by Dr Edgar Codd; 1970
Does not guarantee optimality in operations and query executions.	Guarantees query optimisation because **it is founded on relational algebra, set theory, and functional dependency theory.** Later normalization theory is added to it as well.

Relational modelling provides a mathematical model for interpreting our data. Interpretation of the data is required to understand the context and pin down the **entities, attributes and relationships.**

The first Relational Model

It was first proposed by Dr E.F. Codd in 1970 [link to publication].

Informally, any entity might relate with any other entity through specific common attributes. Any entity and any relationship are modelled as a relation, which is represented as a 2-dimensional table. The table consists of **an ordered set of attributes (columns)**, where each column gives an indication of the meaning of belonging to that relation; **a set of tuples (rows)**, which represent the specific instances of that relation; and **a specific attribute that uniquely identifies a tuple in the relation**.

Database Schema formalism

Given a schema of a relation **R(A_1, A_2, …, A_n)**, R is the name of the relation and (A_1, A_2, …, A_n) is an **ordered** set of attributes. Each attribute A_i assumes values in a domain D_i.

A **tuple t of R** is an ordered set of values corresponding to the attributes' ordering of R and satisfying the domain constraints. It is an n-dimensional point.

t = <v1, v2, … vn> where vi ∈ Di

An **instance r(R)** is a finite set of tuples (set of n-dimensional points).

r(R) = {t1, t2, …, tm}: ti is a tuple of R

r(R) ⊂ D1 x D2 x … x Dn, i.e., a subset of the n-dimensional Cartesian product of the domains; combining all possible values among all domains.

NULL represents an unknown, inapplicable, uncertain or missing value.

A **relational database schema is a set S of relational schemas that belong to the same database**. S is the name of the database.

S = {R1, R2, …, Rk} U {NULL}

Integrity constraints

Constraints are conditions that must hold on all valid relation instances for each relation. There are **three types of fundamental constraints**:

Key Constraint

Unique tuple identification.

- **Superkey SK** of a relation R is a subset of attributes containing **at least** one attribute that uniquely identifies any tuple. For any two distinct tuples t1 and t2 it holds true the implication:

t1 != t2 -> t1[SK] != t2[SK]

Which means that the given set of attributes identifies t1 and t2 uniquely

- **Candidate key K** is the **minimal superkey**; K = {A1, …, Ak} is minimal iff:

  K is candidate key ↔ K' ≡ K \ {Ai } is not a superkey for any Ai ∈ K

  i.e., the removal of any attribute from K results in K' that is no longer a superkey.

Entity Integrity Constraint

Keys cannot be NULL.

- **Primary key PK** is arbitrarily chosen from the set of candidate keys.

  t[PK] ≠ NULL for any r(R)
- If PK has several attributes, NULL is not allowed in any of these attributes.

Referential Integrity Constraint

Interpretation of relations.

Roles: *referencing* relation R1 and *referenced* relation R2.

- **Foreign Key FK** in R1 is an attribute that has exactly the same value as a Primary Key PK in R2 or NULL.

  t1[FK] references t2[PK] → t1[FK] = t2[PK] or t1[FK] = NULL
- If t1[FK] is NULL, FK attribute cannot be part of the primary key for t1 - it violates the entity integrity constraint

There are also **Domain constraints**:

- Each value from an attribute must come from the domain of that attribute.
- A value can be NULL if it is allowed by the designer.
- Similarly, an attribute can have a restriction to never be NULL if the designer decides.

Additionally, there are **Semantic Integrity Constraints**: based on application semantics and cannot be expressed by the relational model - need constraint specification language to be described.

Fundamental operations and violations

Fundamental operations are:

- INSERT
- DELETE
- UPDATE

Principle 1: Integrity constraints should not be violated by the operations.
Principle 2: Operations may propagate to trigger other operations automatically; Ensure to maintain integrity constraints, thus the database transits between consistent states.

If an integrity constraint is violated then:

- Cancel the operation that causes the violation (REJECT/ABORT option);
- Perform the operation but inform the user of the violation;
- Trigger additional operations so the violation is corrected (CASCADE/SET NULL option);
- Execute a user-specified error-correction procedure;

Violations for INSERT

- **Domain:** one of the attribute values for the new tuple is not in the attribute domain
- **Key constraint:** the value of a key attribute in the new tuple already exists
- **Referential integrity:** a FK value in the new tuple references a PK that does not exist in the referenced relation
- **Entity integrity:** the PK value is NULL in the new tuple

Violations for DELETE

- **Referential integrity:** delete a PK value, which is referenced by FK in other relations

Violations for UPDATE

- **Key constraint**: introduce duplicate PKs
- **Referential integrity**: updating FKs

Functional Dependency & Normalisation Theory

Guidelines for a good design

1. The attributes within a relation should make sense.
   Minimizes the similarity between entities. Any relationship between entities should be represented only through foreign keys.
2. Avoid redundant tuples (repetition of the same information).
   Replication of tuples wastes storage. Replicas also need to be kept consistent during insertion, deletion, update which requires extra work and may introduce consistency anomalies.
3. Guideline 3: Relations should be designed such that tuples have as a few NULL values as possible.
   NULL may mean that a value is not applicable, invalid, unknown, or unavailable. Attributes that are frequently NULL should be placed in a separate relations to avoid waste of storage and reduce uncertainty.
4. Design relations to avoid fictitious tuples after join.
   Find the best splitting attribute that does not generate fictitious tuples.

Functional Dependency Theory

Measures the degree of goodness of a relational schema.

Given a relation R, an attribute X functionally determines an attribute Y, if a value of X determines a unique value for Y. **FD: X → Y (X uniquely determines Y)** holds if whenever two tuples have the same value for X, they must have the same value for Y.

If t1[X] = t2[X] then t1[Y] = t2[Y]

Lemma:

If K is a candidate key of relation R, then it functionally determines all attributes in R:

FD: K → {R}

Lemma:

William Armstrong's inference rules for FDs:

- If Y ⊆ X then X → Y (Reflexive) - if X is composite, it functionally determines the atomic attributes it's made up from
- If X → Y then X U {Z} → Y U {Z} (Augmentation)
- If X → Y and Y → Z then X → Z (Transitive)

Theory of Normalisation

Progressive decomposition of unsatisfactory relations by breaking up their attributes into smaller relations, which can be used reconstruct the entire information efficiently without redundancy and fictitious tuples after their composition. The degree of composition is referred to as Normal Form. The higher the NF, the higher the quality of the schema - 3NF or BCNF are the optimal trade-off between quality and execution efficiency, since further decomposition would result in a high cost of reconstruction of the split information (inefficient).

- **Prime attribute** - belongs to some candidate key of the relation
- **Non-prime attribute** - not a member of any candidate key
- **Full Functional Dependency** - if we remove any prime attribute A from the primary key X, then X \ {A} does not functionally determine Y anymore.
- **Partial Functional Dependency** - if we remove any prime attribute A from the primary key X, then X \ {A} does still functionally determine Y - A does not need to be part of PK, it is verbose
- **Transitive Functional Dependency** - given a primary key X and non-prime attributes Z and Y such that: $X \rightarrow Z$ and $Z \rightarrow Y$, then the non-prime Y is transitively dependent on the primary key X *via the non-prime Z*.

First Normal Form (1NF)

A relation R is in 1NF if **it contains only atomic (indivisible) values**.
Remove nested or multivalued attributes.

Second Normal Form (2NF)

A relation R is in 2NF if **every non-prime attribute A in R is fully functionally dependent on the primary key of R**.
Remove all the prime attributes from the primary key which result in partial dependencies (remove redundant prime attributes) to transform them into full dependencies.

Methodology:

1. Identify all the partial FDs in the original relation (already in 1NF)
2. For each partial FD, split by creating a new relation such that all non-prime attributes in there are fully functionally dependent on the new primary key - the prime attribute in the original relation causing partial FDs.
3. The new relation will be in 2NF.

Third Normal Form (3NF)

A relation R is in 3NF (being already in 2NF) if there is no non-prime attribute which is transitively dependent on the primary key; That is, **all non-prime attributes should be directly dependent on the primary key**, and not dependent on it via another non-prime attribute. In fact, non-prime attributes can and should be non-transitively dependent on every candidate key in R - this does not violate 3NF.

Methodology:

1. Split the original relation into two relations: the non-prime transitive attribute is:
    - the PK to the new relation
    - the FK in the original relation referencing to the new relation.

Boyce-Codd Normal Form (BCNF)

A relation R is in BCNF if, **whenever there exists a FD: X → A then X is a superkey of R, i.e., the left-hand side should be a superkey, i.e., it contains a key**.

Theorem:

Let relation R not in BCNF and let X → A be the FD which causes a violation in BCNF (X is not a superkey).
Then, the relation R should be decomposed into two relations:

- R1 with attributes: R\{A} (all attributes in R apart from A)
- R2 with attributes: {X} U {A} (put together X and A) If either R1 or R2 is not in BCNF, repeat the process.

If you join R1 and R2 w.r.t. common attribute X then we obtain the original relation with no fictitious tuples!

SQL

SQL is declarative - specifies what to do, not how to do it.

- Create a schema:
  CREATE SCHEMA Company;
- Create a table:
  CREATE TABLE EMPLOYEE…;

Attributes & Domains

- Numeric data types:
  - Integers: INT
  - Floating point: REAL or DECIMAL(total digits, digits after decimal)
- Character/string data types:
  - Fixed length: CHAR(n) - exactly n characters
  - Variable length: VARCHAR(n) - up to n characters
- Bitstring data types:
  - Fixed length: BIT(n)
  - Varying length: BIT VARYING(n)
- Boolean data type:
  - TRUE
  - FALSE
  - NULL
- Other:
  - TIMESTAMP, DATE INTERVALS, etc

Value Constraints

- Default value of an attribute:
  - DEFAULT {value}
- NULL is not permitted for an attribute:
  - NOT NULL
- Range domain constraint:
  - CHECK(clause)

Key Constraints

- Primary key clause
  - PRIMARY KEY (attribute)
- Candidate key clause
  - UNIQUE (attribute)

Referential Constraints

- Foreign Key
    - FOREIGN KEY (attribute) REFERENCES SomeRelation(PK_attribute)

Triggered actions:

- ON DELETE (option)
- ON UPDATE (option)

Triggered actions options:

- CASCADE - propagates DELETE / UPDATE to all referential tuples
- SET NULL
- DEFAULT

Queries

- Declare what to retrieve, i.e., which are the attributes of interest
  SELECT <attribute list>
- Declare from where to retrieve, i.e., which is the table/relation
  FROM <table list>
- Declare with what condition to retrieve, i.e., which are the conditions
  WHERE <condition>

Table as a Variable

SELECT …
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE …
A relation might play different roles within a query, e.g., employee might be a supervisee and employee might be a supervisor…(used in recursive references…)

Missing WHERE

If there is no condition on tuple selection (no WHERE clause), and FROM involves more than one relation, a Cross product will be returned - all possible tuple combinations. This is computationally heavy and generally bad times.

Use of the Asterisk

Use * to SELECT all attributes in the relations specified in FROM clause.

Tables as Multisets in SQL

Multisets may have duplicate values. They allow for UNION, EXCEPT and INTERSECT operators.

Use to split either-or query into two sub-queries and then use the set UNION operator over the partial results.

```
(SELECT   DISTINCT Pnumber
 FROM     PROJECT, DEPARTMENT, EMPLOYEE
 WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn
          AND Lname='Smith' )
 UNION
( SELECT  DISTINCT Pnumber
 FROM     PROJECT, WORKS_ON, EMPLOYEE
 WHERE    Pnumber=Pno AND Essn=Ssn
          AND Lname='Smith' );
```

Three-valued Logic

SQL is a three-valued logic: TRUE (1), FALSE (0) and UNKNOWN (0.5) Each NULL value is different from any other NULL value! **Any value compared with NULL evaluates to UNKNOWN.** Use IS NULL and IS NOT NULL conditionals.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

min

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

max

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

$1\text{-}x$

Nested Query

Nested query is a query within another (outer) query. SELECT-FROM-WHERE block is within outer query's WHERE clause. Nested query's output is input to outer's WHERE via: IN, ALL, EXISTS.

Nested Uncorrelated Query

First execute the nested query, and then execute the outer query using the inner query's output.

Operator IN

Checks whether a value belongs to the inner's output set (or multiset), i.e., $v \in S$

```
SELECT  FNAME
FROM    EMPLOYEE
WHERE   DNO IN ( SELECT  DNUMBER
                 FROM    DEPARTMENT
                 WHERE   DNAME = 'Research');
```

Evaluates to **5**

Operator ALL

Compares a value with all the values from the inner's output set.

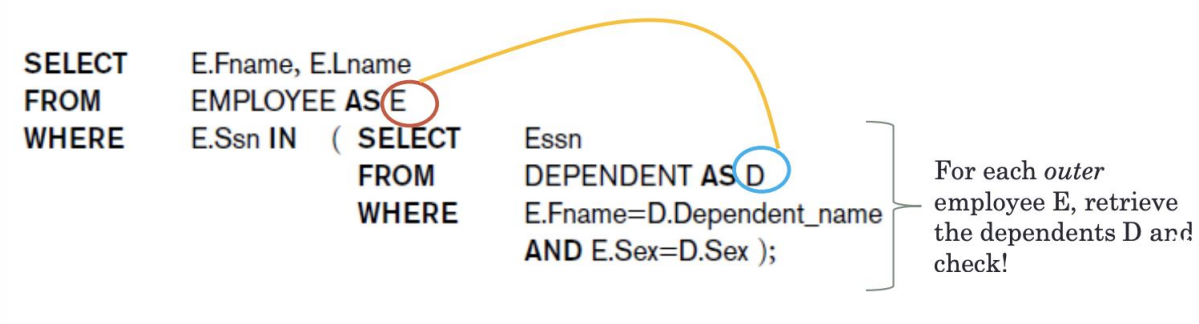
```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ALL  ( SELECT  Salary
                          FROM    EMPLOYEE
                          WHERE   Dno=5 );
```

First, find *all* salaries from employees in Department 5;

Nested Correlated Query

For each tuple of the outer query, execute the inner query.
Relation as a variable: global scope (outer) and local scope (inner).

Operator IN



Lemma: **Correlated queries using IN can be collapsed into one single block.**

```
SELECT E.Fname, E.Lname
FROM   EMPLOYEE AS E, DEPENDENT AS D
WHERE  E.Ssn=D.Essn
    AND E.Sex=D.Sex
    AND  E.Fname=D.Dependent_name;
```

Operator EXISTS

Checks whether the inner's output is empty set or not, and returns FALSE or TRUE, respectively, e.g., S = {} or $S \neq$ {}

```
SELECT E.Fname, E.Lname
FROM   EMPLOYEE AS E
WHERE  EXISTS
(SELECT * FROM DEPARTMENT AS D WHERE  E.DNO = D.DNUMBER)
```

- Checks if a *given* employee is working at *some* department.

Opposite: NOT EXISTS - TRUE if the returned set is empty, FALSE otherwise.

```
SELECT S.Name
FROM   STUDENT S
WHERE  NOT EXISTS
      (SELECT * FROM GRADES G
       WHERE G.StudentID = S.StudentID
             AND NOT (G.Grade = 'A')
       )
```

Retrieve the names of all students who have A's in all of their courses. For each Student tuple of the other query, check that the set of grades that are not A for that student is empty.

Advanced SQL and Analytics

Join Query

Inner Join

Matches tuples using FK and PK (theta-join). Matching operator is “=”, which is why it is referred to as EQUIJOIN: R1.PK = R2.FK. Join condition is in the WHERE clause along with the selection condition:

```
SELECT Fname, Lname, Address
FROM   EMPLOYEE, DEPARTMENT
WHERE Dname = 'Research'  %selection condition
AND    DNO = DNUMBER;     %equi-join condition
```

- A tuple is retrieved if and only if there exists a matching tuple;
- FK is not NULL

Outer Join

Left Outer Join

- LR LEFT OUTER JOIN RR
- Every tuple in the left relation LR must appear in result
- If no matching tuple exists for each tuple from LR in RR, add NULL values for attributes of RR

Query 1: Show the last name of an employee and the last name of their supervisor, *if there exists*!

```
SELECT  E.Lname, S.Lname
FROM    (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
         ON E.Super_SSN = S.SSN)
```

E.Lname	S.Lname
Smith	Wong
Borg	NULL
Franklin	Jennifer

Right Outer Join

- LR RIGHT OUTER JOIN RR
- Every tuple in RR must appear in result
- If there is no matching tuple in LR for each tuple in RR, add NULL values for attributes of RR

Aggregate Function

Summarizes statistical information from a group of tuples into a single value.

Built-in aggregate functions

- COUNT(*) - counts tuples; **does not discard NULL values**
- SUM(X) - sum all values for attribute X
- MAX(X) / MIN(X) - retrieve maximum/minimum value for attribute X
- AVG(X) - calculate average value for attribute X
- CORR(X,Y) - retrieve correlation between attributes X and Y
- etc

We can also define our own functions specific to the application.

```
SELECT    SUM (Salary) AS Total_Sal,
          MAX (Salary) AS Highest_Sal,
          MIN (Salary) AS Lowest_Sal,
          AVG (Salary) AS Average_Sal
FROM      EMPLOYEE
WHERE     DNO = 5;
```

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}

- Grouping attribute must appear in SELECT clause
- If the grouping attribute has NULL values, a separate group is created for them

Apply aggregation functions to each group.

Query 3: Show the number of employees per department & average salary per department.

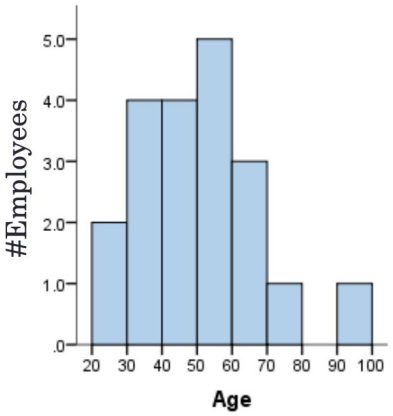
```
SELECT    DNO, COUNT (*), AVG (Salary)
FROM      EMPLOYEE
GROUP BY  DNO;
```

Histogram

Partition a relation with respect to a given attribute, then for each group, calculate its cardinality. This approximates how the values of this attribute is spread across tuples.

Query 4: Show the number of employees *per* age.

```
SELECT    E.AGE, COUNT (*)
FROM      EMPLOYEE AS E
GROUP BY  E.AGE;
```

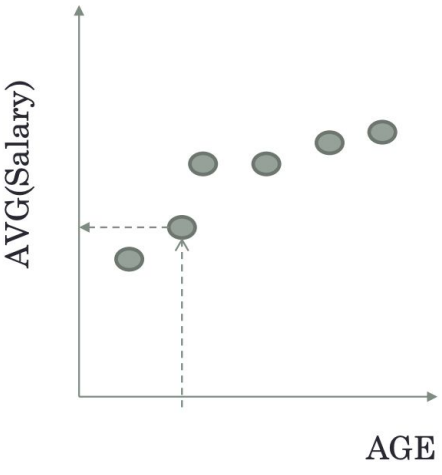


Regression

Approximate the dependency of one attribute on another. For example, given an age, predict the expected salary (assuming there is some dependency between the two). Partition the relation with respect to a given attribute, then for each group, calculate some aggregate of the dependent attribute.

Query 5: How is the average salary of employees distributed along the age?

```
SELECT    E.AGE, AVG(E.Salary)
FROM      EMPLOYEE AS E
GROUP BY  E.AGE;
```



Task 1: Which is the average salary of employees *per* department? Include the department name at the results.

```
SELECT      D.DNAME, AVG (E.SALARY) //avg salary/dept
FROM        DEPARTMENT AS D, EMPLOYEE AS E
WHERE       D.DNUMBER = E.DNO
GROUP BY    D.DNAME;
```

Step 1: Associate EMPLOYEE with DEPARTMENT
Step 2: Group *associated* tuples together w.r.t. **DNAME**
Step 3: For *each* group, *take* the average of salaries

GROUP BY + HAVING

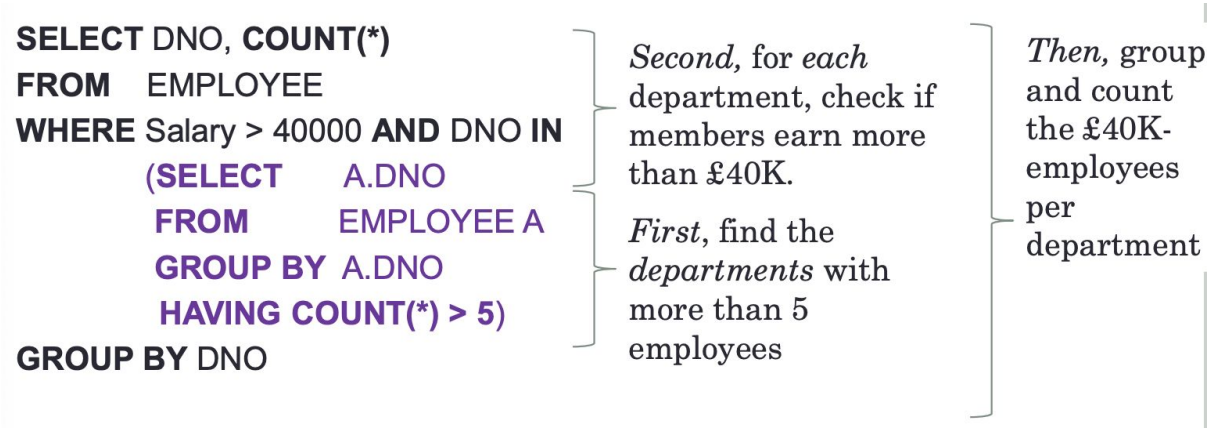
Having condition selects/rejects an entire group after grouping.

```
SELECT      P.PNAME, COUNT (*)
FROM        PROJECT AS P, WORKS_ON AS W
WHERE       P.PNUMBER = W.PNO
GROUP BY    P.PNAME
HAVING      COUNT(*) > 2
```

Show the number of employees per project only for projects with more than 2 employees. Include project name in returned tuples.

Task 1: Which are the managers (last names) of those departments with *more* than 100 employees?

```
SELECT M.LNAME
FROM   EMPLOYEE M, DEPARTMENT P
WHERE  M.SSN = P.MGR_SSN
AND    P.DNUMBER IN (
          SELECT     E.DNO
          FROM       EMPLOYEE AS E
          GROUP BY   E.DNO
          HAVING     COUNT(*) > 100);
```



If HAVING COUNT(*) > 5 is in outer query, the employees will get filtered out by the WHERE clause before counting.

Student(SSN, SchoolID, GPA)

Task 1: Show the number of students per School, whose GPA is greater than 17 and they are in Schools having more than 100 students.

```
SELECT SchoolID, COUNT(*)
FROM Student
WHERE GPA > 17 AND SchoolID IN
        (SELECT S.SchoolID
         FROM Student S
         GROUP BY S.SchoolID
         HAVING COUNT(*) > 100)
GROUP BY SchoolID
```

EMPLOYEE(SSN, …, DNO)
Task 1: Show the department(s) with the maximum number of employees. It might be the case that more than one department has the maximum number of employees.

```
SELECT DNO, COUNT(*)
FROM EMPLOYEE
GROUP BY DNO
HAVING COUNT(*) = (SELECT MAX(A.members)
                   FROM
                   (SELECT D.DNO, COUNT(*) AS members
                    FROM EMPLOYEE D
                    GROUP BY D.DNO) AS A
                   );
```

SQL

INSERT

Key, integrity and referential constraints are automatically enforced.

INSERT INTO (relation)
VALUES (tuple)

DELETE

Get a relation and use WHERE to specify which tuples to be deleted. If no WHERE clause is used, all the tuples in the relation will be deleted and an empty relation will be returned.

DELETE FROM (relation)
WHERE (condition)

Tuples are deleted only from one table at a time, unless ON DELETE CASCADE is specified as a constraint.

UPDATE

Modifies values in tuples that satisfy WHERE clause prior to the modification.

UPDATE (relation)
SET (attribute = new value)
WHERE (clause)

Tuples are updated in only one table at a time, unless ON UPDATE CASCADE is specified as a constraint.

Task: Give *all* employees in the department 5 a 10% raise in salary.

```
UPDATE    EMPLOYEE
SET       SALARY = SALARY *1.1
WHERE     DNO = 5
```

Modified SALARY depends on the original SALARY in each tuple:

- **SALARY** on the *right of* = refers to the **old** salary *before modification*
- **SALARY** on the *left of* = refers to the **new** salary *after modification*

DROP

Drop named schema elements, such as tables, domains or constraints.

DROP (element type) (element name)

Example: `DROP SCHEMA COMPANY CASCADE;`

It *removes* the schema and *all* its elements including tables, constraints, etc.

Example: `DROP TABLE EMPLOYEE;`

It *drops* the existing table `EMPLOYEE` and *all* of its tuples.

Example: `DROP TABLE EMPLOYEE CASCADE CONSTRAINTS;`

It *drops* the existing table `EMPLOYEE`, all of its tuples and *drop* the FOREIGN KEY constraints of the tables referring to `EMPLOYEE` (*but* not those tables).

ALTER

Add, drop columns; change column definition; add, drop table constraints.

Example: `ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);`

Example: `ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK;`

Example: `ALTER TABLE COMPANY.DEPARTMENT ADD CONSTRAINT NEW_UNIQUE UNIQUE (Dname);`

Example: `ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address;`

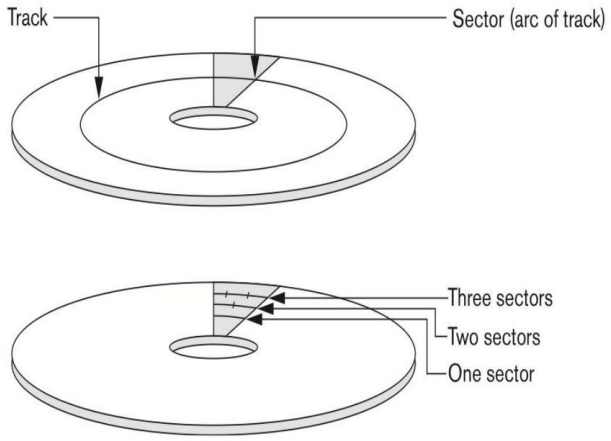
Example: `ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;`

Example: `ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';`

Physical Database Design

Level	Example	Storage capacity	Access Speed	Money Costs
Primary Storage	RAM: main memory, cache	small	fast	expensive
Secondary Storage	HDD, SSD	medium	average	average
Tertiary Storage	Optical drives	large	slow	cheap

Database is too large to fit in main memory; therefore by default it involves secondary storage. Since HDD is not CPU-accessible, data must be first loaded into main memory from disk, then processed in main memory. Data access from HDD is inherently slower.

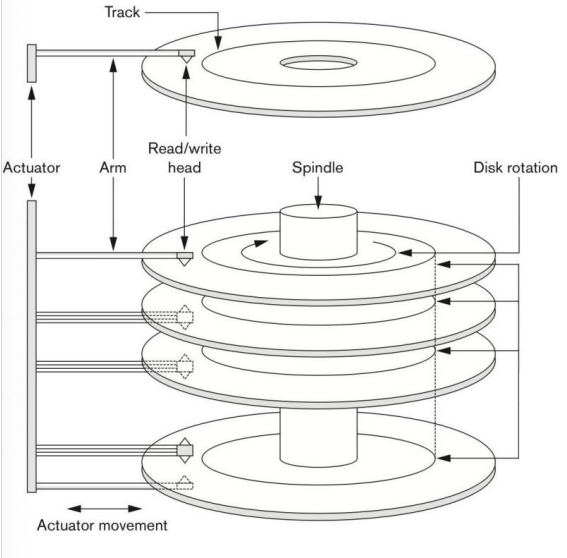


Recipe:

1. Take a (*magnetic*) disk
2. Dig *concentric* tracks
3. Split tracks into *circular* sectors

4. Take the DB data
5. Data are organized in *files* of *records*.
6. Each file consists of *blocks* to store these records: ~512bytes

7. Store each *block* in a set of sectors
8. Done!



Read/Write Access (I/O)

- **Get a physical address**
- **Position:** moving arms over the tracks (*seek*: ~6ms)
- **Spin:** right sector under the head (*rotation*: ~5ms)
- **Transfer:** data from track to memory via bus: *extremely* fast!

Spatial Optimization Problem: Spatially *organize* the file blocks to minimize the expected seek/rotation time.

Organization-based Optimisation

Organize tuples on the disk in order to minimize the cost of I/O access.

- Tuple is represented as a Record
    - Fixed length records - attributes have fixed size in bytes
    - Variable length records - attributes have variable size, some fields are optional with size >= 0 bytes
- Records are grouped together in a Block
    - Fixed length, normally 512 to 4096 bytes
- Blocks are grouped together in Files

Blocking Factor

Given a record of R bytes and a block of B bytes, the number of records stored in a block, i.e. records per block, is called blocking factor (bfr):

bfr = floor(B/R)

We assume we can fit at least one record per block, i.e. B >= R.

Number of Blocks

Given a number of records r and a blocking factor bfr, the number of blocks needed to store all the records is:

number of blocks = ceil(r/bfr)

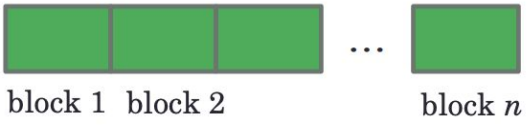
Blocks to Files on Disk

File is a set of Blocks. There are several allocation methods for allocating blocks in files.

Contiguous (neighbouring) allocation

A file contains spatially consecutive blocks. It requires available continuous space (defragmentation).

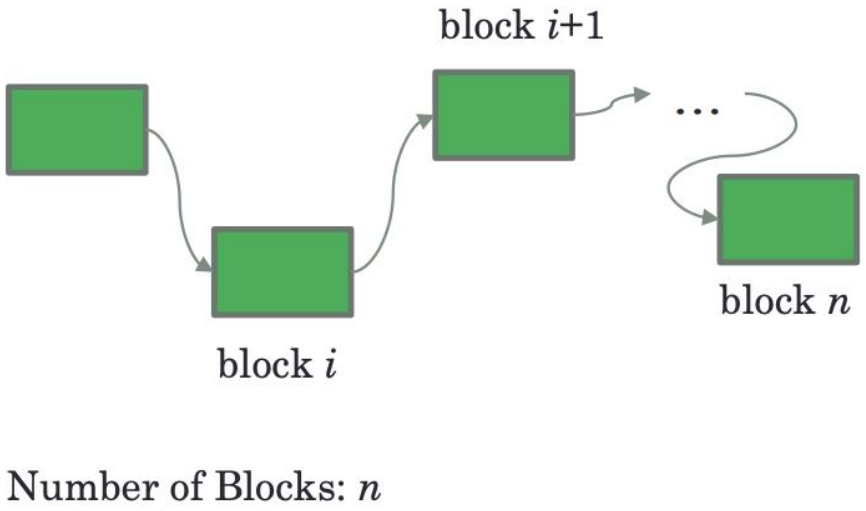
Self-navigation file:



Number of Blocks: n

Linked allocation

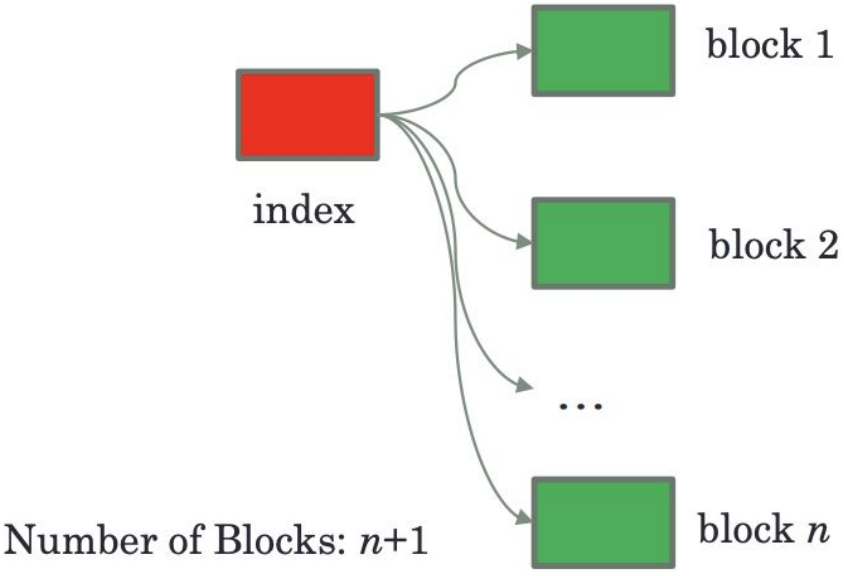
Each block i has a pointer to the logically next block i+1 anywhere on the disk, i.e. a linked list of blocks. Navigation information is stored in each block.



Number of Blocks: n

Indexed allocation

There exists a specific block (index), which maintains a list of pointers, each of which points to the physical address of each block. The index block can be anywhere on the disk. Navigation information is stored in a separate block.



Number of Blocks: n+1

Expected I/O Access Cost

We cannot transfer single records from memory to disk or vice versa - we have to transfer whole blocks that contain the given record(s) we are working with. Files containing blocks can be of different types - **heap, ordered, or hash**. For each of those we can measure the I/O cost for:

- **Retrieval/Search cost**: Retrieving the whole block containing the record x according to a search field from the disk to the memory
- **Update cost:** Inserting, deleting, updating a record x by transferring the whole block from memory to the disk

Given a specific file type provide a primary access path based on a specific searching field, e.g., search only via SSN.

Cost Objective function

Number of block accesses (read/write) to search, insert, delete, update a record x.

Heap File

Unordered file. A new record is added at the end of the file, i.e. appended at the end of the last block. It is good for log files, where we only need to append new information.

Inserting a new record - O(1)

Load the last block from the disk to the memory (address is in file header) - 1 block access. Insert a new record at the end of the block and write the block back to the disk - 1 block access. **Complexity is therefore O(1) i.e. constant - efficient**.

Retrieving a record - O(b)

Linearly scan through all the b blocks in the file, retrieving each block from memory. For each retrieved block, scan for the record. On average, we have to scan b/2 blocks, and in the worst case (if the record we are searching for is at the end, or is not in the file) we have to access b blocks. **Complexity is therefore O(b) i.e. linear - inefficient**.

Deleting a record - O(b) + O(1)

Find and load the block containing the record - do the retrieve process (which we already know is O(b) for b blocks). Then remove the record from the block and write the block back to the disk (this leaves gaps in blocks) - 1 extra block access. **Complexity is therefore O(b) + O(1) i.e. linear - inefficient.**

To deal with gaps of unused space after deleting records from blocks, adopt deletion markers - bits which indicate that a record is deleted (1) or not (0). Periodically, reorganize the file by gathering the non-deleted records (bit=0) and freeing up the space of the deleted records.

Updating a record - O(b) + O(1)

Scan to locate record to be updated linearly. Then retrieve it, update it and write it back again to the disk. **Complexity is therefore O(b) + O(1) i.e. linear - inefficient.**

Sequential File

All records are sorted by an ordering field and are kept sorted at all times. Suitable for SQL queries that require sequential processing (FROM <relation> ORDER BY <ordering field>), involve the ordering field. **Efficient for range queries over the ordering field.**

Retrieving a record

Via the ordering field - O(log2 b)

Use binary search to find the right block. **Complexity is therefore O(log2 b) - efficient.** For range queries, it takes O(log2 b) to find the record at the beginning of the range. Then linearly scan until record corresponding to the end of the range is reached - O(b). Therefore O(log2 b) + O(b) overall.

Via a non-ordering field - O(b)

Linearly scan files to find the right block. **Complexity is therefore O(b) - inefficient.**

Inserting a new record

Locate the block where the record should be inserted - use binary search. Then, shift the records that follow it by 1 to make room for the new record**. Complexity is therefore O(b) - inefficient.**

Chain pointers

If there is free space in the right block, insert the new record there. Otherwise, insert the new record in an overflow block and use chain pointers to restore the sequence. Pointers from and to the new record need to be updated - sorted linked list.

Deleting a record

First, locate the block from which the record is to be deleted via binary search. Use deletion markers and update the pointer not to point to the deleted record. Periodically, re-sort file to restore sequential order - external sorting is expensive. **Complexity is therefore O(b) - inefficient.**

Updating a record

Via the ordering field - O(b)

The record is deleted from its old position - O(b), and inserted into its new position O(b). **Complexity is therefore O(b) - inefficient.**

Via a non-ordering field - O(log 2b) + O(1)

Use binary search to locate the record - O(log2 b), then update the non-ordering field and write the record back to memory in its old position, since the ordering has not changed - O(1). **Complexity is therefore O(log 2b) + O(1) - efficient.**

Hash file

Hashing constitutes of partitioning records into M buckets, each bucket can have more than one blocks. Then choose a hash function y=h(k) where y ∈ {0, 1, …, M-1}; h must uniformly distribute records into the buckets - each bucket is chosen with equal probability 1/M:

y = h(k) = k % M

- **Indirect Clustering:** group tuples together with respect to their hashed values and not with respect to their hash-field values.

External Hashing

1. Load the hash block with the file header from the disk to memory. Hash block contains the hash map (from bucket to block address) - 1 block access
2. Compute the bucket id using the hash key k - y=h(k)
3. Access the hash map that's already in memory to get the block address that bucket y maps to
4. Retrieve the block at that block address - 1 block access
5. Search for the record with hash h(x.k) in the block
6. If found return record, else return not found

Retrieve a record - O(1)

Use external hashing to find record. **Complexity is overall O(1), which is efficient.**

Inefficient for range queries, since the buckets for continuous values are not continuous (two sequential numbers are not guaranteed to be in the same bucket), therefore each value in the range is a separate query. **Complexity is O(n) + O(1) + O(nm) for n distinct values in the range and m overflow buckets per bucket.**

Insert a new record - O(1) or O(1) + O(n)

If the record is hashed to a non-full bucket, **complexity is O(1)**. However, if thee bucket is full, use chain pointers - at the end of each bucket, insert a pointer to an overflow block,

which is initially NULL. If the bucket overflows, set the pointer to the address of the overflow block; if more than one overflow blocks are needed, at the end of the last overflow block, insert another pointer to another overflow block, and so on. **Complexity is O(1) + O(n) block accesses.**

Delete a record - O(1)

If the record is in the main bucket, it can be deleted immediately O(1). If there are any records in the overflow bucket, move one to the main bucket. **Complexity is O(1), which is efficient.**

Update a record

Via a non-hash field - O(1) or O(1) + O(n)

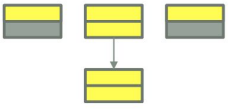
Locate the record in its main or overflow bucket (retrieve). Load into memory, update it, write it back (insertion).

Via a hash field - O(1)

Delete from old bucket, then insert in new bucket.

Distribution of data and Expected cost

Comparison

	Heap File 	Sequential File 	Hash File 
Worst	Linear scan: **3 block accesses**	Binary search: log2(3) = **1.58 block accesses**	0.33*1 + 0.33*(1 + 1) + 0.33*1 = **1.32 block accesses**
Average	Linear scan: **2 block accesses**	Binary search: log2(3) = **1.58 block accesses**	0.33*1 + 0.33*(0.5*1 + 0.5*2) + 0.33*1 = **1.15 block access**
Best	Linear scan: **1 block accesses**	Binary search: **1 block access**	0.33*1 + 0.33*1 + 0.33*1 = **1 block access**

The expected cost is unpredictable because the distribution determines the load per bucket. Therefore hash function should be designed to follow the data distribution and to uniformly distribute records over buckets.

For example, given bfr = 2 records/block, and attribute x of the records. Out of all the possible selection queries, p% will involve x, and (1-p)% will not. We can calculate the worst-case expected cost for a Hash and a Sequential file with respect to p, in order to identify a rule about when to use either of those.

- Hash worst case expected cost:
    - 3 main buckets, one overflown
    - p*(0.33*1 + 0.33*2 + 0.33*1) + (1-p)*4 = 4 – 2.68*p
- Sequential worst case expected cost:
    - 3 blocks
    - log2 (3)*p + (1-p)*3 = 3 – 1.41*p
- Heap worst case expected cost:
    - 3 blocks
    - p*3 + (1-p)*3 = 3; independent of p, grows proportionally to number of blocks

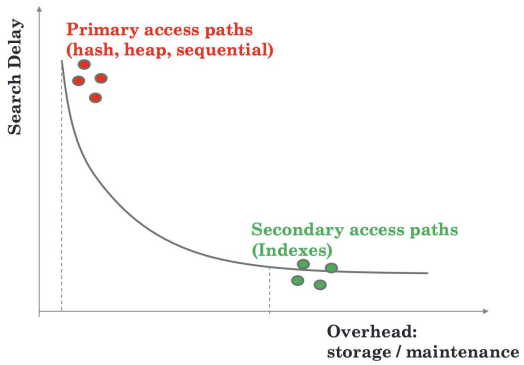
If Hash Worst Case < Sequential Worst case use hashing, otherwise sorting.

4 - 2.68*p < 3 - 1.41*p

p < 0.79

If at least 80% of the queries involve x, hash file with respect to x, otherwise sort file by x.

Indexing Methodology Part I



Given any file type provide a secondary access path using more than one searching field, e.g., SSN, Salary, Name, etc.

- **Cost**: Additional meta-data file on the disk & maintenance
- **Benefit**: Expedite significantly the search process avoiding Linear Scans.

Principles

1. Create one index over one field: index field
2. An index is a separate file
3. All index entries are unique and sorted with respect to the index field
4. First search the index to find the block pointer, then access the block from the data file

Index file occupies less blocks than the data file because the index entries are smaller records (index value, block pointer) and therefore we can fit more index entries in a block than data records.

bfr(index-file) > bfr(data-file)

Index files can be:

- **Dense**: index entry for each record in the data file
- **Sparse**: index entry only for some records

Searching over index is faster than over file, because the index is guaranteed to be sorted - we can use binary-based or tree-based methods to find the pointer to the block that we need.

Single-Level Ordered Indexes

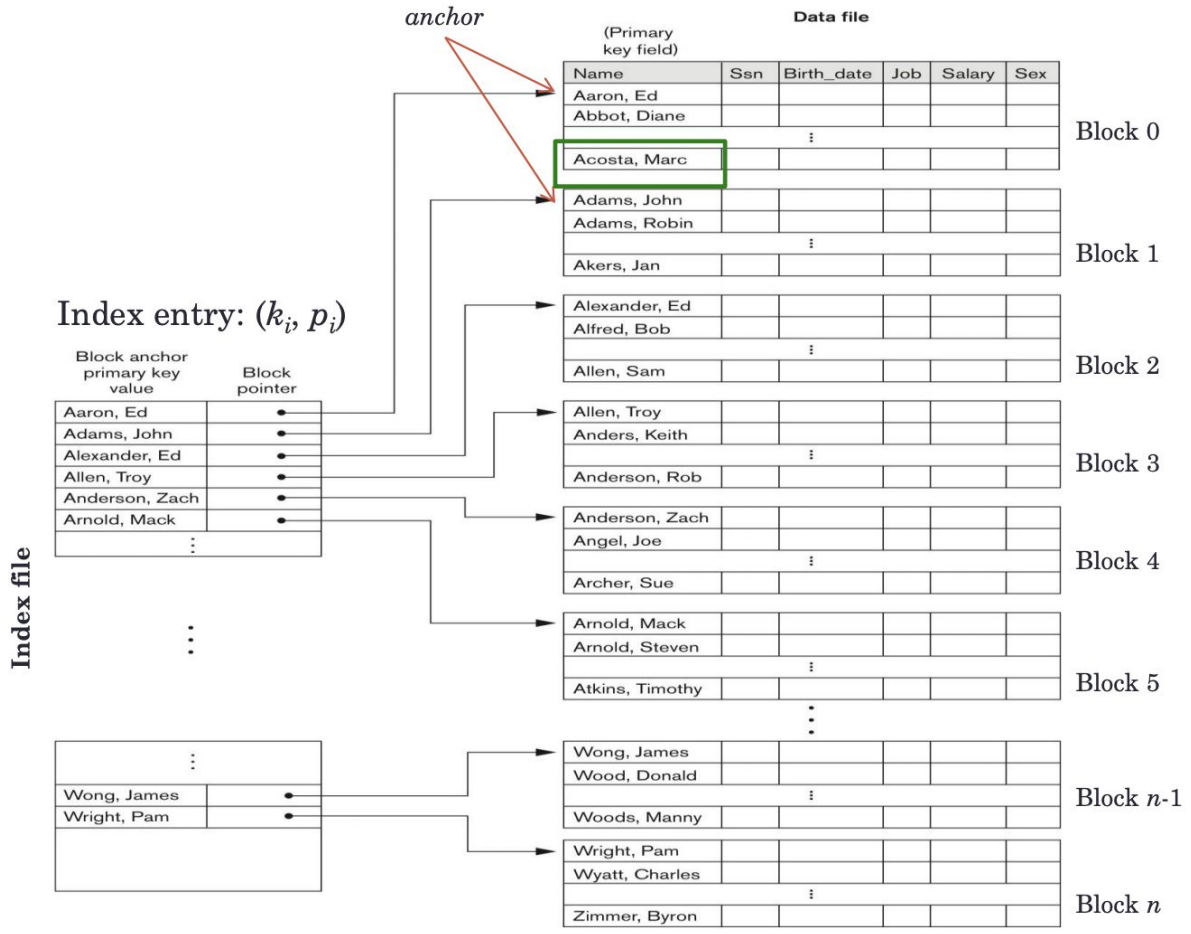
Involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value.

Primary Index

Index field is an ordering-key field of a sequential file, e.g., SSN; file is sorted by SSN.

- fixed length index entries = pair (ki, pi)

- ki is the unique value of the index field
- pi is the pointer to the ith block containing the record with key ki
- first data-record in block i with value ki is the anchor of block i.
- **sparse: only one index entry per data block**



- blocking factor bfr = floor(B/R)
- file size (in blocks) b = ceil(r/bfr)
- index blocking factor: ibfr = floor(B/(Value+Pointer))
- index file blocks ib = ceil(b/index blocking factor)
- binary search on index: ceil(log2(ib)), + 1 additional access to load the block pointed by index

When to use?

Observe the frequency of anchor records updates/deletions

Theorem

Primary Index is created iff the block can accommodate at least two index entries independently of the number of data-records. If that is not the case, there will be as many blocks as there are blocks in the file, which will have the same efficiency as just searching over the ordering field.

index blocks = ceil(data file blocks/index blocking factor)
index blocking factor = floor(B/index entry size)

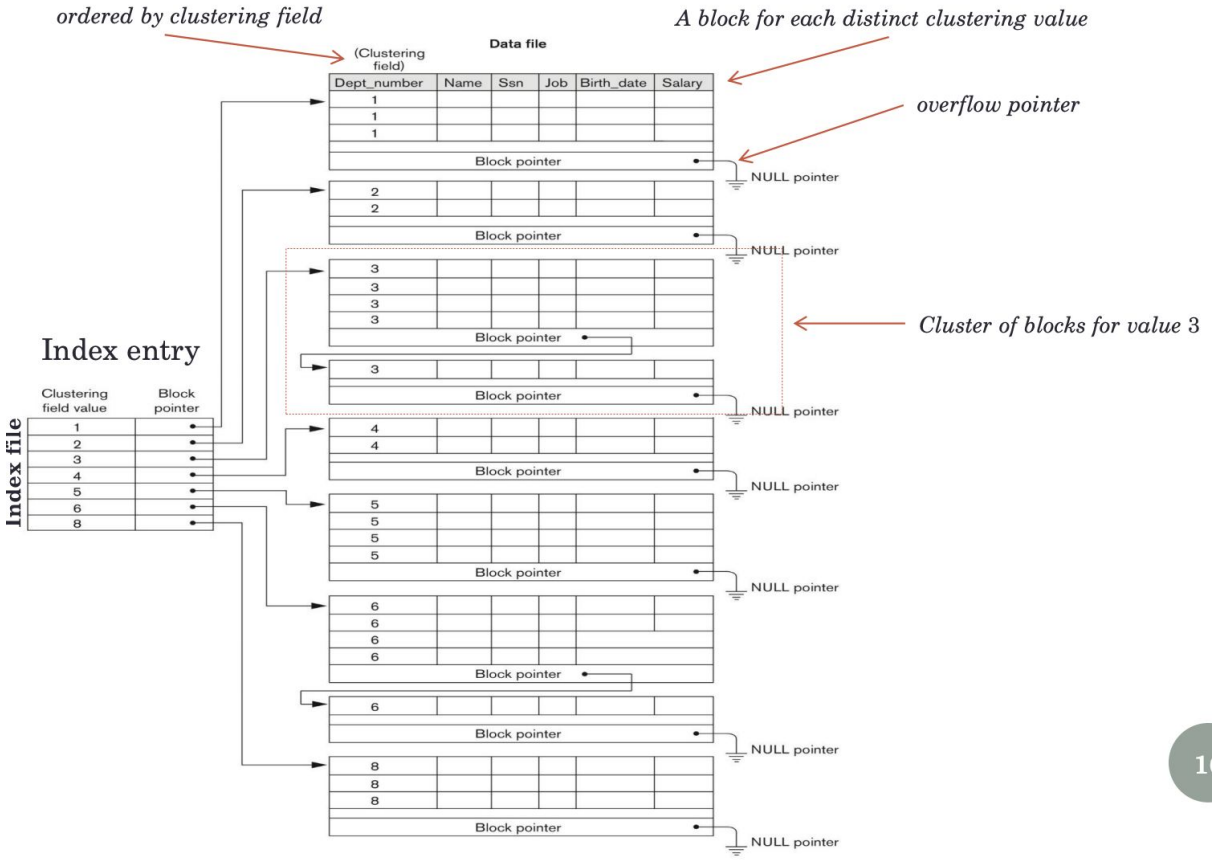
if index blocking factor is 1, then the index blocks = data file blocks

Clustering Index

Index a sequential file on an ordering, non-key field. The file is a set of clusters of blocks, where there's a cluster for each distinct value.

index-entry = (distinct-value, block-pointer)

- **sparse: one index-entry per distinct clustering value.**
- Block pointer points at the first block of the a cluster.



- blocking factor bfr = floor(B/R)
- file size (in blocks) b = ceil(r/bfr)
- index blocking factor: ibfr = floor(B/(Value+Pointer))
- index file blocks ib = ceil(n/index blocking factor), n - number of clusters
- log2(m) to find cluster, b/n to load it

When to use?

Compare performance with linear search over an ordered file by a uniformly distributed non-key field.

- Cost over linear search:

$$\frac{b(n+1)}{2n}$$

, where n - number of clusters, b - number of blocks

- Cost over clustering index:

$$\log(m) + \frac{b}{n}$$

, where n - number of clusters, b - number of blocks, m - index blocks

Theorem

A clustering Index of m blocks is always created over an ordering non-key field.
Proof: cost over the index < cost over linear search since index blocks m < file blocks b and:

$$\log_2 m < \frac{b(n+1)}{2n} - \frac{b}{n} = \frac{b(n-1)}{2n} < \frac{b}{2} < b$$

Lemma

The linear search over an ordering non-key field is bounded by b/2, i.e., half of the naïve linear expected cost, when number of distinct values n is very large

$$\lim_{n\to\infty} \frac{b(n+1)}{2n} = \frac{b}{2} < b$$

```
SELECT * FROM EMPLOYEE WHERE DNO >= 3
```

EXAMS

Step 1: Expected cost **$\log_2(m)$ block accesses for DNO = 3.**
Step 2: Expected cost **(b/n) block accesses for DNO = 3.**

Step 3: L := *max*(DNO)-3 number of extra *clusters* to be retrieved.

```
SELECT MAX(DNO) FROM EMPLOYEE
```

Step 4: Expected cost **L·(b/n) block accesses for DNO > 3 and DNO <= max(DNO) .**

Step 5: Total: **$\log_2(m)$ + (L+1)·(b/n) block accesses.**

Linear Search

$$\sum_{k=1}^{3}\left(\frac{b}{n}\right) + \sum_{k=3+1}^{n}\left(\frac{b}{n}\right) = b$$

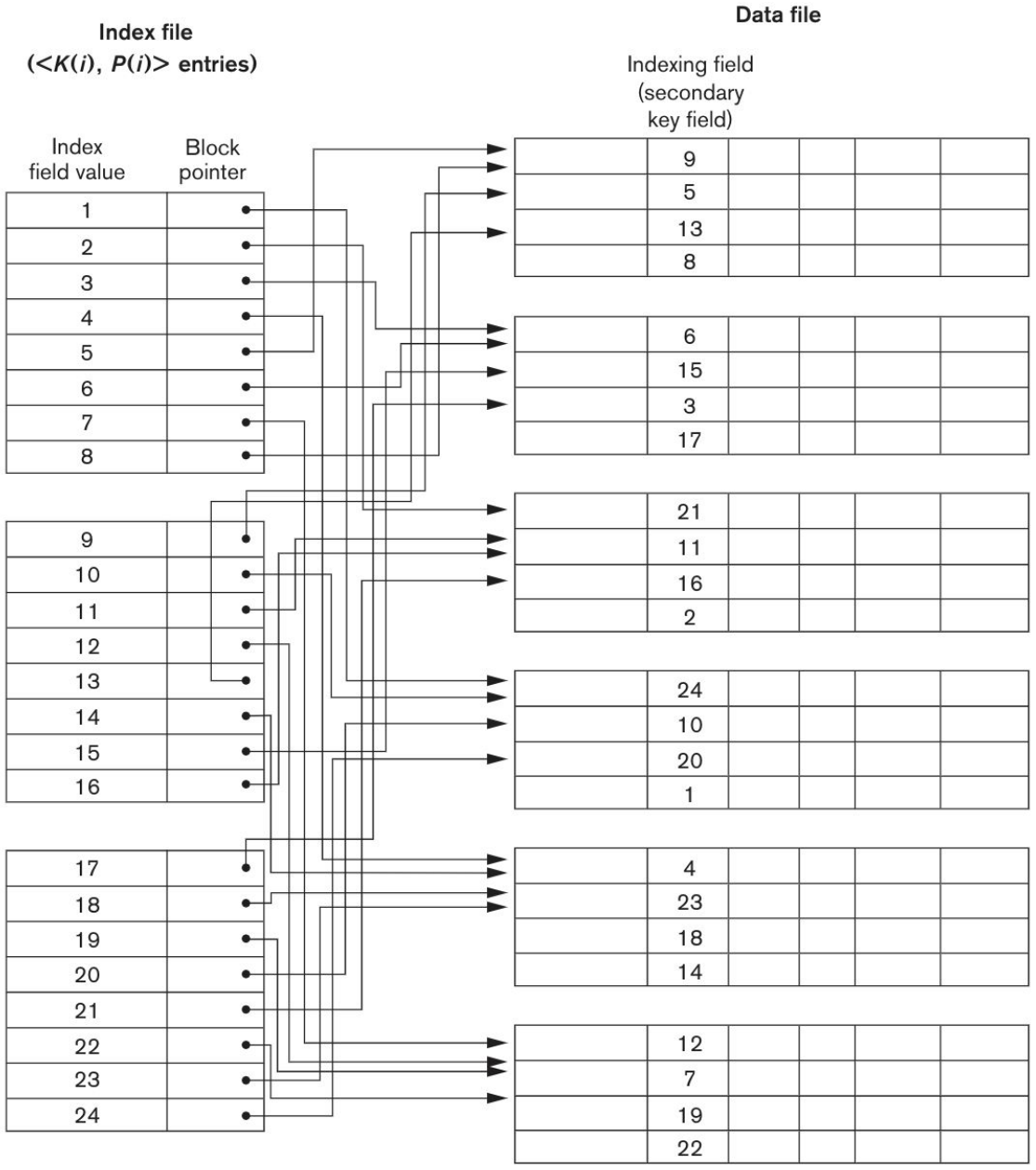
Benefit *iff*? $\log_2 m < \frac{b(n-(L+1))}{n}$ **Check @Home for *each* L value.**

Secondary index

Index a file on a non-ordering field. The file might be unordered, hashed, or ordered but not ordered according to the indexing field.

On a non-ordering, key field

- **Dense: one index per data record** - the file is not ordered according to the indexing field, thus, we cannot use anchor records
  index-entry = (index value, block pointer)



- blocking factor bfr = floor(B/R)
- file size (in blocks) b = ceil(r/bfr)

- index blocking factor: ibfr = floor(B/(Value+Pointer))
- index file blocks ib = ceil(r/index blocking factor) since there is an index entry for each record
- binary search on index: ceil(log2(ib)), + 1 additional access to load the block pointed by index

On a non-ordering, non-key field

Group the block addresses of those records having the same index value. Assign an index entry per group (cluster) of block addresses

index-entry = (distinct value, cluster pointer)

- **Sparse: one entry per cluster**

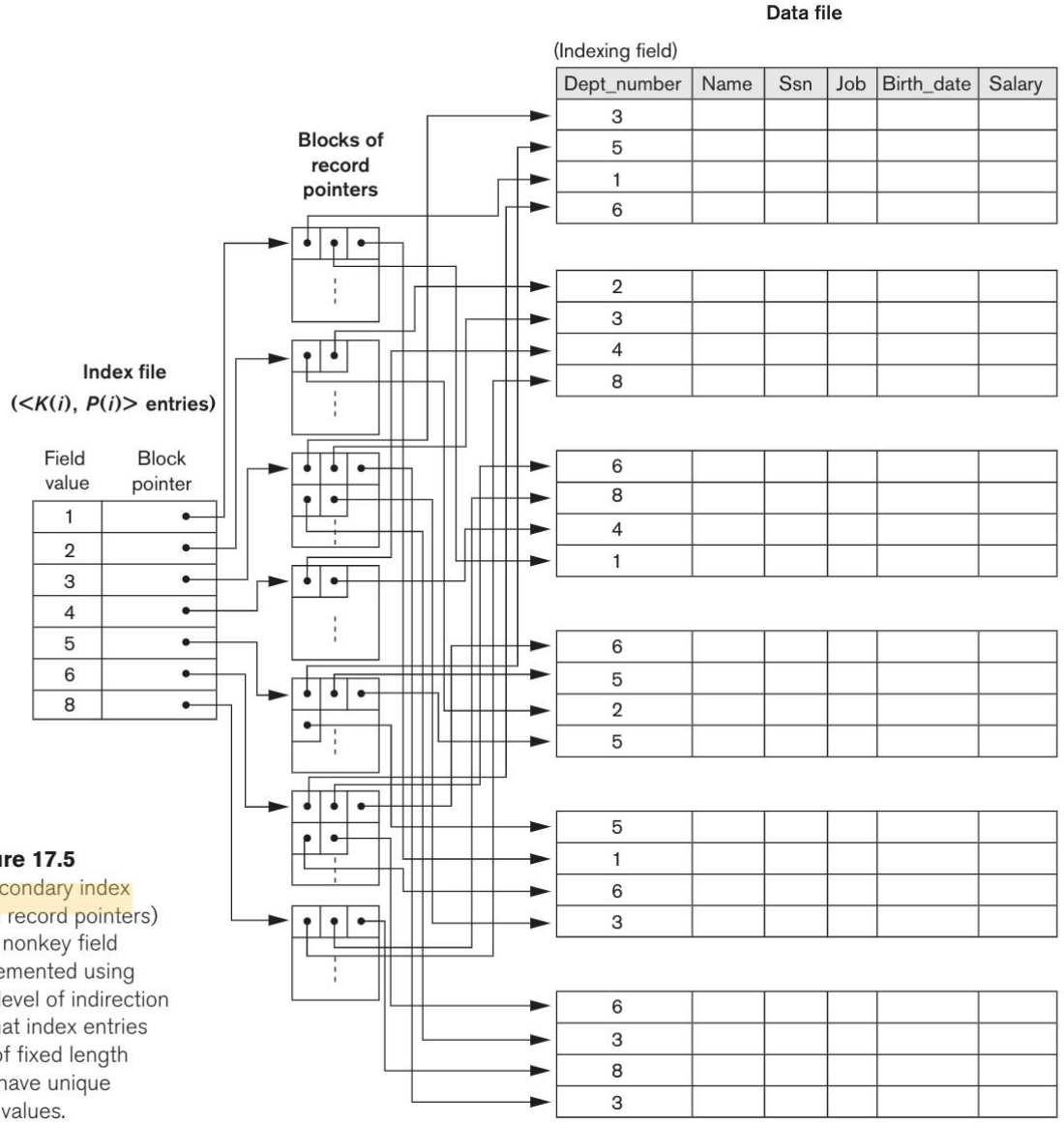


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

- Binary search Level 2 - 1 block access
- Direct access via pointer to Level 1 - 1 block access
- Load all the corresponding data blocks that Level 1 block points to

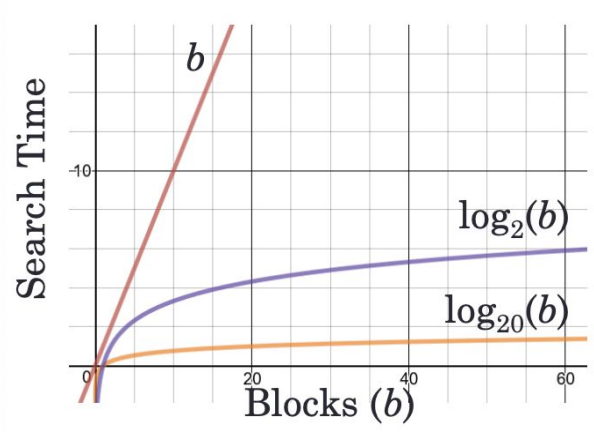
Multilevel Indexes

We can build a primary index over any index file, since it is an ordered file w.r.t. a key field (index of an index).

- the original index file is referred to as the base or Level-1 index,
- the additional index is referred to as Level-2 index (index of an index)
- …
- if we repeat this to level > 2 we obtain Level-t index, i.e., index of an index of an index

Reasoning

A logarithm with base m > 2 splits the search space into m sub-spaces until finding the unique block.



Theorem

Given a Level-1 Index with m being the blocking factor of the Level 1 index file, then the multi level index is of maximum level t = logm(b)

- m is known as the fan-out; it is equal to the index blocking factor = (B/(P+V))
- Level N index entries = Level N-1 index blocks
- Level N index blocks = ceil(Level N-1 index blocks/m)

- Level 1 secondary index - dense; Other levels in multilevel index - sparse
- Retrieval/Search is extremely efficient - t+1 block accesses for t-level index
- Insertion, deletion and update are costly - all updates need to be reflected on each level

Dynamic Multilevel Indexes

Indexes that are adjusted to the deletions and insertion of records, i.e., they expand / shrink (following the distribution of the key values).

Search tree

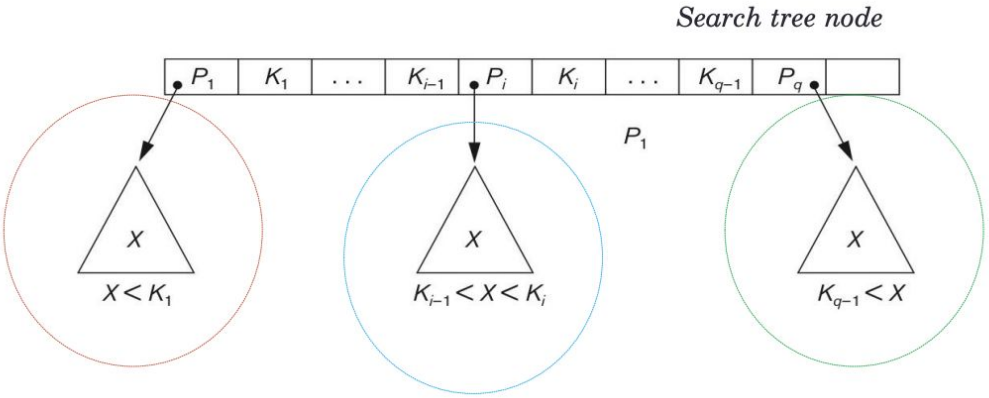
N level multilevel index represents a tree structure: Level N is the root, Level 0 are actual file data blocks - leaves.

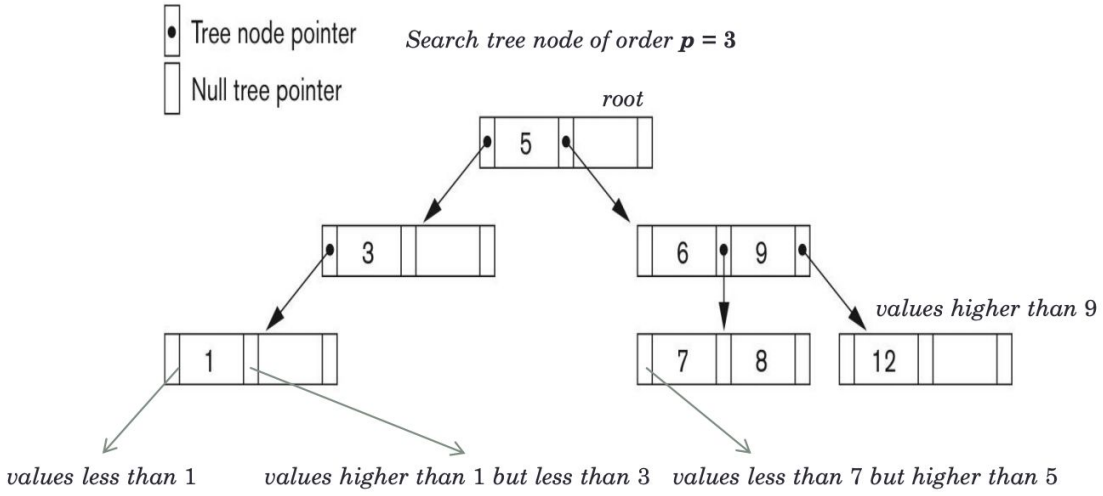
Over Non-ordering Key

Search Tree of order p (splitting factor) over a non-ordering key field k Each node has a set of q pairs: (pointer, key) where q ≤ p:

$$\textbf{Tree-Node} := \{(P_1, K_1), (P_2, K_2), \ldots, (P_{q-1}, K_{q-1}), P_q\}$$

- Pi either points to another tree node or is NULL
- Ki is a search key value
- Ki-1 < X < Ki , for 1 < i < q
- If i = 1, X < K1 and if i = q, Kq-1 < X
- Each node with q<=p pointers has q-1 keys





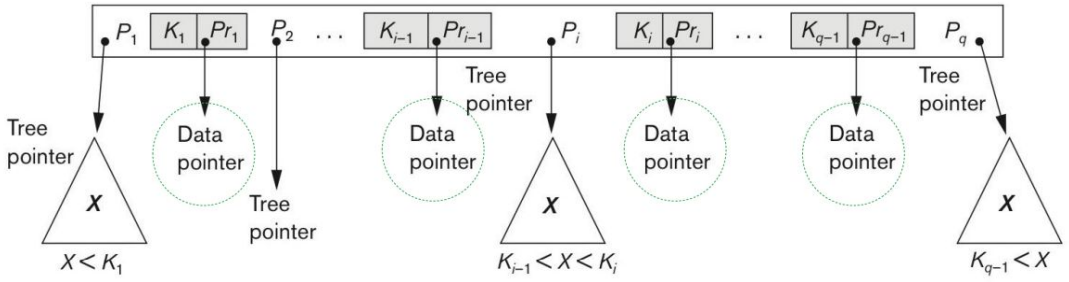
- Complexity is O(log p(n)), n - total key values, p - splitting factor/order/number of child pointers in a block
- It does not adjust to the distribution of the keys; i.e., leaf-nodes are at different levels. It can turn into a linked list of nodes instead of a tree structure - very high tree depth t

B-tree over Non-ordering Key

Extends Search tree with data pointers.

$$\textbf{B-Tree-Node} := \{\mathbf{P}_1, (\mathrm{K}_1, \mathbf{Pr}_1), \mathbf{P}_2, (\mathrm{K}_2, \mathbf{Pr}_2) \ldots, \mathbf{P}_{q\text{-}1}, (\mathrm{K}_{q\text{-}1}, \mathbf{Pr}_{q\text{-}1}), \mathbf{P}_q\} \text{ where } q \leq p$$

- Pri is a data pointer to the data block holding the value Ki
- Pi is a Tree pointer pointing to a subtree X
- Keys Ki are ordered K1 < K2 < Ki <... < Kq-1
- For all key values X in a subtree pointed to by tree-pointer Pi :
  - Ki-1 < X < Ki , for 1 < i < q
  - If i = 1, X < K1 and if i = q, Kq-1 < X
- Each node with q ≤ p tree-pointers has q-1 keys & q-1 data-pointers.
- Grants immediate access to the block of the searching key;
- search complexity = number of block accesses down the tree to find key + 1 block access to the block from key's data pointer (assuming each node is 1 block)

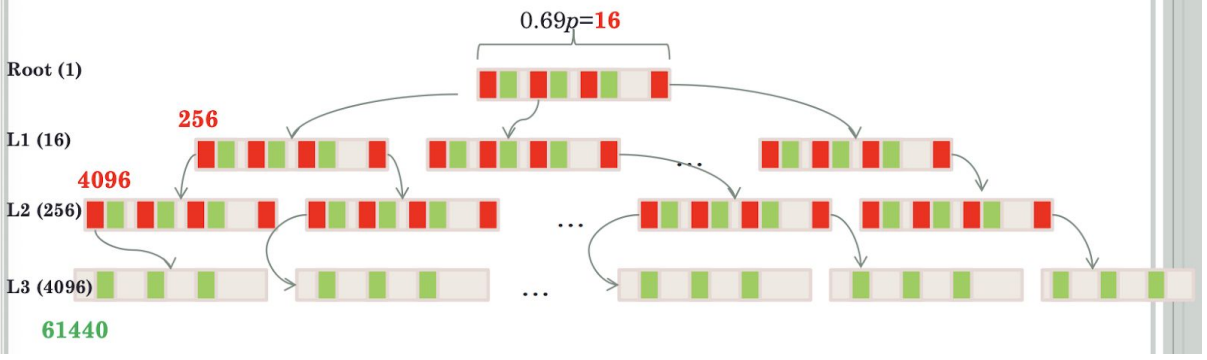


Usage

Task 1: Create a **3-level B-Tree index** of order $\boldsymbol{p}$ **= 23** over a *non-ordering / key field*

Context: Each B-Tree node is 69% *full* of information (pointers/keys).

- On average, each B-Tree node accommodates $0.69p$ = **16 tree pointers/ 15 key values**.
- Average **fan-out** = 16 per tree node, i.e., split the tree space into **16 sub-trees**.



- Root: 1 node with 15 keys/data-pointers; 16 pointers to tree nodes;
    - Pointers to tree nodes q can be maximum = p;
    - keys and data pointers are q-1 each
    - Since the tree is only 69% full, we take q = 0.69*p
- Level-1: 16 nodes with 16*15=240 keys/data-pointers;16*16 = 256 pointers to nodes;
    - Number of nodes = number of tree pointers from previous level = q;
    - number of keys/data pointers = number of nodes*(q-1) = q(q-1);
    - number of tree pointers = number of nodes*q = q*q
- Level-2: 256 nodes with 256*15=3840 keys/data-pointers; 256*16 = 4096 pointers to nodes;
    - Number of nodes = number of tree pointers from previous level = q^2;
    - number of keys/data pointers = number of nodes*(q-1) = q^2(q-1);
    - number of tree pointers = number of nodes*q = q^3
- Level-3: 4096 nodes with 4096*15 = 61440 keys/data-pointers; and null pointers (leaves);
    - Number of nodes = number of tree pointers from previous level = q^3;
    - number of keys/data pointers = number of nodes*(q-1) = q^3(q-1);
    - number of tree pointers = number of nodes*q = q^4
- Total keys that can be stored: 61440 + 3840 + 240 + 15 = 65,535 key entries pointing to data blocks
    - Sum of keys across levels = (q-1) * (q^0 + q^1 + … + q^(n+1) where n = tree level; in the example n=3

If tree is full, we need to create a new level to be able insert a new key. This new level would have q*(number of previous level's pointers) nodes and q-1*(nodes) keys and data pointers, but only one key/data pointer would be used - this is a massive redundancy.

- Total storage = total data pointers * data pointer size
  + total keys * key size
  + total tree pointers * tree pointer size
- Total blocks needed:
    - blocking factor = floor(block size/tree node size)
    - tree node size = q * (tree pointer size) +
      (q-1) * (key size)
      (q-1) * (data pointer size)
    - number of nodes = sum(blocks needed for the nodes of each level)
    - usually one node per block

B-trees store too much metadata. The fan-out indicates the number of search splits at every step - it is the number of tree pointers per node, q<=p. We want to maximize fan-out - to do so, we maximize the number of tree pointers per node; we can free up space for more tree pointers by removing data pointers from the tree nodes.

B+ Tree over Non-ordering Key

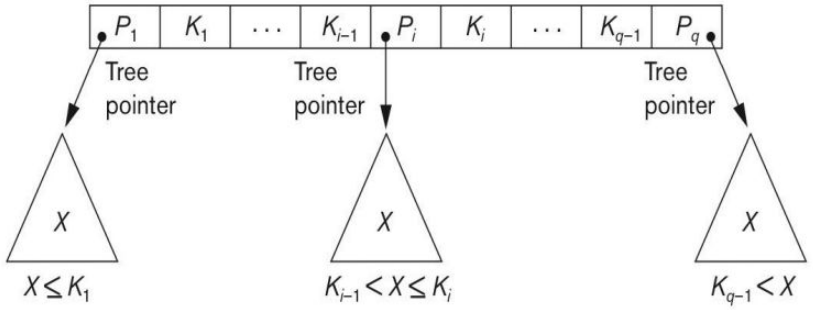
Differentiate between internal nodes and leaf nodes.

Principles:

1. Only leaf nodes have data pointers
2. Lead nodes hold all the sorted key values, along with their corresponding data pointers
3. Some key values are replicated in internal nodes to guide search process

Internal nodes

$$\textbf{B+-Internal-Node} := \{\mathbf{P}_1, \mathrm{K}_1, \mathbf{P}_2, \mathrm{K}_2, \ldots, \mathbf{P}_{q\text{-}1}, \mathrm{K}_{q\text{-}1}, \mathbf{P}_q\}, q \leq p.$$



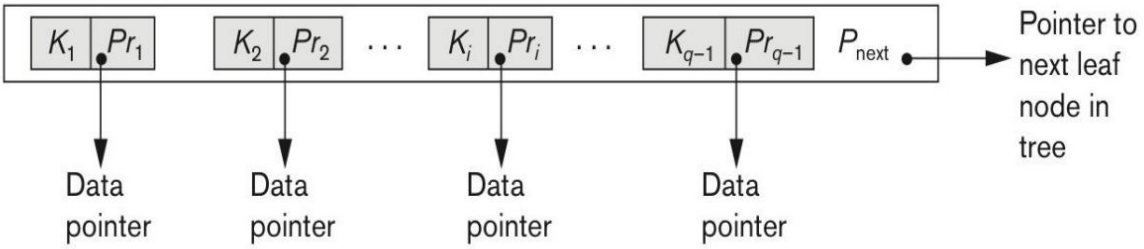
Just like standard search tree. Each internal node with q tree pointers can hold at most q-1 keys. Allows for higher fanout by saving space via removing data pointers.

To fit an internal node in a block of size 512 bytes, where p is max(q) i.e. maximum fanout, we need:

p * tree pointer size + (p-1) * key size <= block size
p <= (block size + key size) / (tree pointer size + key size)

Leaf nodes

$$\textbf{B+-Leaf-Node} := \{(K_1, Pr_1), (K_2, Pr_2) \ldots, (K_{q-1}, Pr_{q-1}), P_{next}\}, \; q \leq p.$$



- Pri is a data pointer to the actual block holding value Ki
- P next is a tree pointer to the next leaf node - leaf nodes comprise a linked list
- All leaf nodes are on the same level
- q-1 keys and data pointers
- 1 tree pointer
- Useful for range queries and aggregates

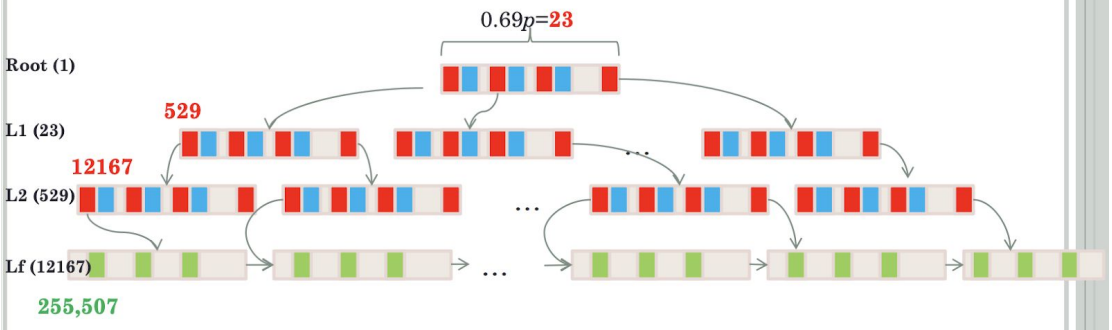
To fit a leaf node in a block of 512 bytes, where p=max(q), i.e. maximum fanout,

p * (data pointer size + key size) + tree pointer size <= block size

p <= (block size - tree pointer size) / (data pointer size + key size)

3-level B+ Tree index: 3 internal levels and one *leaf* level of order $p = 34$ and $p_L = 31$

Context: Assume that each internal & leaf tree node is 69% full.



- Internal tree-node has $0.69p$ = **23 tree-pointers**, i.e., **22 keys** (on average).
- Leaf node has $0.69p_L$ = **21 data-pointers** (on average).

Root: 1 node with **22 keys** and **23** pointers to tree nodes;
Level-1: 23 nodes with 23*22 = **506 keys** and 23*23 = **529** pointers to nodes;
Level-2: 529 nodes with 529*22 = **11638 keys** and 529*23 = **12167** pointers to leaves;
Leaf Level: 12167 nodes with 12167*21 = **255,507 keys/data-pointers**.

When to use?

B+ Tree as Secondary Index of Level t, $t > 1$, over *non-ordering key* SSN

```
SELECT AVG(SALARY) FROM EMPLOYEE
WHERE SSN >= L AND SSN <= U
```

Context:

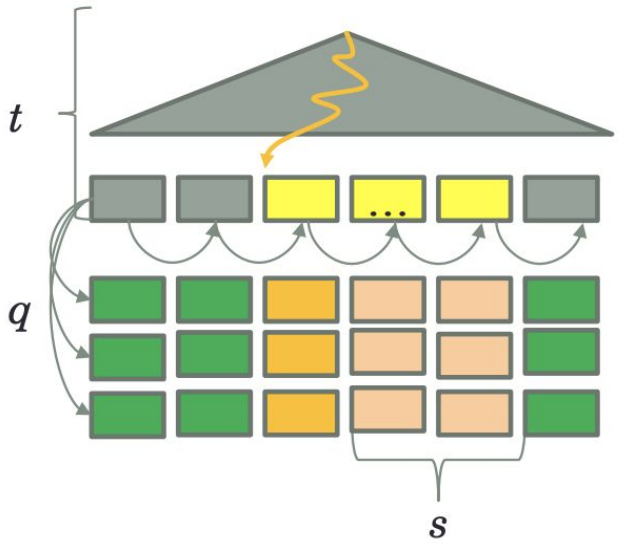
- File b = **1250 blocks**; n = **1250 employees**, 1 employee *per* block
- SSN = 10 bytes, Pointer P = 10 bytes, Leaf order q = 10, Node order p = 5,
- *All* leaf nodes are **100% full**
- **SSN** ∈ **[1, 2, …, 1250]**
- Let a = ratio of employees retrieved = (U-L)/n ∈ [0, 1] (range ratio)

Task 1: Which is the B+ Tree level t ?

Task 2: Which is the maximum range ratio a to avoid serial scan ?

1. Root node has p = 5 tree node pointers
2. L1 has 5 nodes, so 5*5=25 node pointers
3. L2 has 25 nodes, sp 5*25=125 node pointers
4. Leaf level has 125 nodes, storing 10 values/data pointers each, so it can store all 1250 SSNs

Therefore, levels t = 4 - root, 2 internal levels, 1 leaf level.

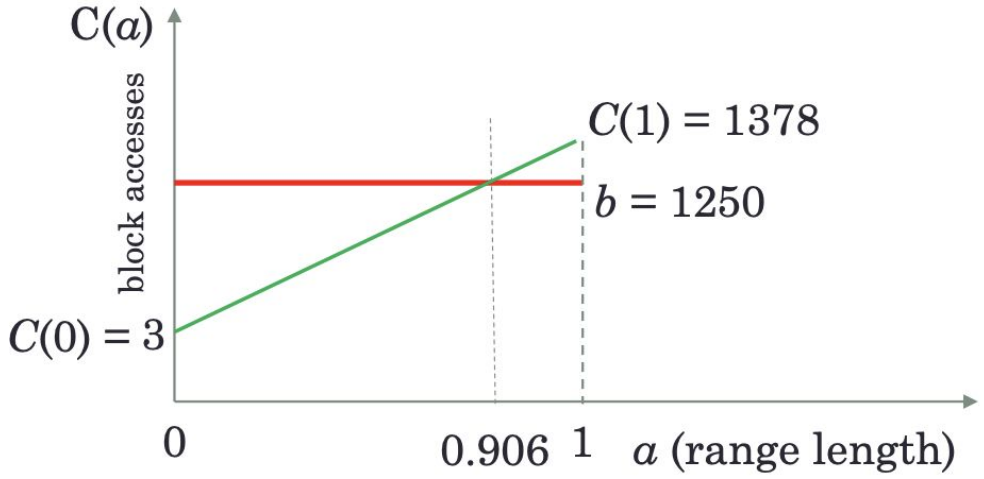


Expected cost for a given ratio a = (U-L)/n, where U and L are bounds of range query.

1. t block accesses to reach the leaf node with SSN = L
2. q block accesses for loading data-blocks and sum up Salary values
3. Leaf nodes accessed: (U-L)/q = a*n/q (values in range / values in each leaf node)
4. Visit a*n/q -1 sibling nodes to scan range - 1 block access for each
5. For each sibling access q data blocks and sum up Salaries

Total: C(a) = t + q + a*n/q - 1 + (a*n/q - 1)q

Use B+ Tree when C(a) < b, where b is linear search cost (scan through all records)



B+ Tree over Non-ordering Non-key

- ***n* = 1000** departments; ***r* = 100,000** tuples (employees)
- Blocking factor **bfr = 50 records/block**
- DNO is *uniformly* distributed: ***d* = *r*/*n* = 100 employees/department**
- **Leaf node order: *q* = 10 DNO values/data-pointers**
- **Tree node order: *p* = 10**

Task 1: How many blocks is the file?

***b* = ceil(100,000/50) = 2000 blocks**

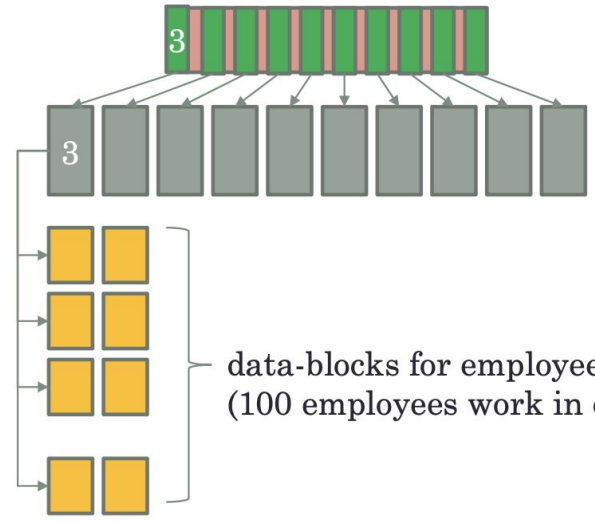
Task 2: How many leaf nodes do we need to accommodate *all* DNO values?

- *n* = 1000 DNO values can be fit in **100 leaf nodes;** each leaf node has ***q* = 10** depts.

Task 3: We have 100 leaf nodes; each one with 10 depts.

Leaf Node Structure:

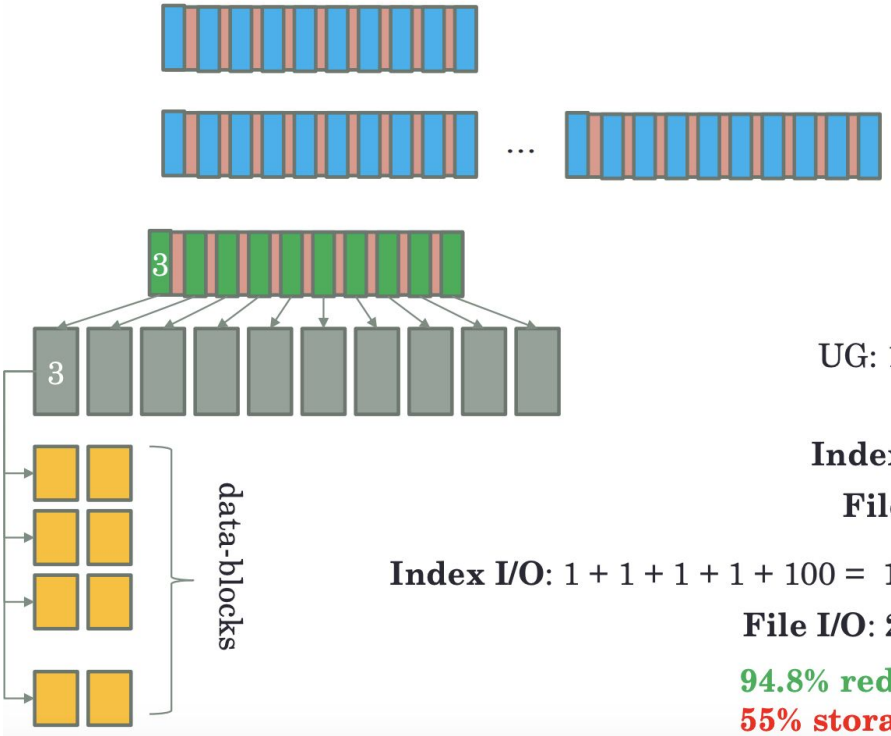
- For *each* DNO value, assign a pointer to a *block of pointers* of those blocks with employees working at this DNO (dept.)
- We need ***d* = 100** block pointers per DNO (dept.)
- We need ***m* = 10 blocks** of block of pointers (one block per DNO value), each block with 100 block pointers (*assumption*: fit in one block)



1 Leaf node (10 depts.)

10 blocks of pointers

100 data-blocks/dept.



Root: 1 block

L1: 10 blocks

Lf: 100 blocks

UG: 100*10 = 1000 blocks

Index Size: 1111 blocks.

File Size: 2000 blocks.

Index I/O: 1 + 1 + 1 + 1 + 100 = **104 block accesses**

File I/O: **2000 block accesses**

94.8% reduction

55% storage

Query Processing

External Sorting

Almost all SQL queries involve sorting of tuples w.r.t. ad-hoc sorting requests defined by the user. There exists a fundamental limitation - we cannot store the entire relation into main memory in order to sort it. External sorting represents sorting algorithm for large relations stored on disk that do not fit entirely in main memory.

Divide and Sort Principle

1. Divide a file into many smaller subfiles and write them to the disk
2. Load each small sub file into main memory, sort via quick sort or bubble sort and write back to the disk
3. Merge two or more sorted subfiles loaded from disk in memory creating bigger subfiles, that are merged in turn

Lemma

The expected cost of the sort and merge strategy is 2b(1+logM(L)), where:

- b is the number of file blocks
- M - degree of merging, i.e. how many sorted blocks are merged in each loop
- L - number of the initial sorted sub-files (before entering merging phase)
- M = 2 gives worst case performance since only a pair of blocks are merged at each step

Strategies for SELECT

SELECT * FROM relation WHERE selection-conditions

Linear Search

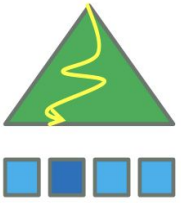
Retrieve every record, test whether it satisfies the selection condition.

- Precondition: none
- **Expected cost: b/2**
  SELECT * FROM EMPLOYEE WHERE SSN = ‘12345678’

Binary search

- Precondition: file must be sorted by an ordering key
- **Expected cost: log2(b) if sorted, log2(b) + 2b(1+logM(L)) if it needs to be sorted first**
  SELECT * FROM EMPLOYEE WHERE SSN = ‘12345678’

Use of Primary Index or Hash Function over a key

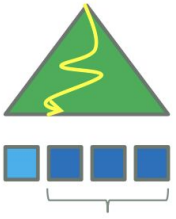


- Precondition:
  - Primary index of level t over the key (file is sorted by key)
  - File hashed with the key
- **Expected cost**
  - **Sorted file: t + 1**
  - **Hashed file: 1 (no-overflow buckets)**

  SELECT * FROM EMPLOYEE WHERE SSN = ‘12345678’

Use of Primary Index in a Range Query

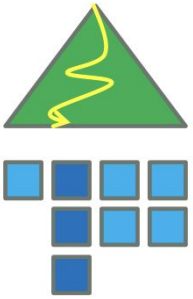
Use Index to find the record satisfying the equality (the boundary of the range), then retrieve all subsequent blocks from the ordered file.



- Precondition: Primary index of level t over the key (file is sorted by key)
- **Expected cost: (t+1) + O(b)**
  SELECT * FROM DEPARTMENT WHERE DNUMBER ≥ 5;

Use of Clustering Index to retrieve Multiple Records

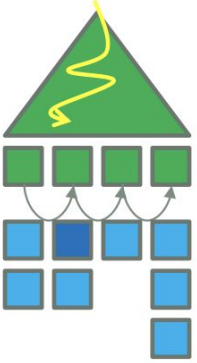
Involves equality on a non-key. Retrieve all the contiguous blocks of the cluster of the non-key value.



- Precondition: Clustering Index of level t on non-key (file sorted by non-key)
- **Expected cost (sorted file): (t+1) + O(b/n), where n is the distinct values of the non-key attribute**
  SELECT * FROM EMPLOYEE WHERE DNO = 5;

Use of Secondary Index (B+ Tree) over equality involving non-ordering key

Retrieve a single record. B+ Leaf Node points at the unique block.

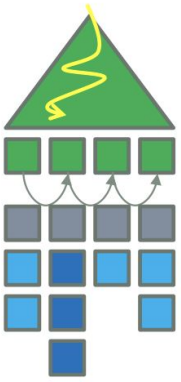


- Precondition: File is not ordered by key
- **Expected cost: t + 1**
  SELECT * FROM DEPARTMENT WHERE MGR_SSN = ‘1234567’;

Use of Secondary Index (B+ Tree) over equality involving non-ordering non-key

Retrieve multiple records from different blocks that have the same value over the non-key attribute used. B+ Leaf Node points to a block of pointers to data blocks with Salary = 40K

- Precondition: File is not ordered by non-key
- **Expected cost: t + 1 + O(b)**
  SELECT * FROM EMPLOYEE WHERE SALARY = 40000;



Strategies for Disjunctive SELECT

Disjunctive selections are conditions involving OR; they return tuples satisfying the union of all selection conditions.

SELECT * FROM EMPLOYEE
WHERE SALARY > 10000 OR NAME LIKE '%Chris%'

- If an access path exists, e.g. B+/hash/primary index for all of the attributes
  - use each to retrieve the set of records satisfying each condition
  - unite all sets to get the final result
- otherwise, if none or some of the attributes have an access path, linear search is unavoidable

Strategies for Conjunctive SELECT

Conjunctive selections are conditions involving AND; they return tuples satisfying the intersection of all selection conditions.

SELECT * FROM EMPLOYEE
WHERE SALARY > 40000 AND NAME LIKE '%Chris%'

- If an access path exists (index) for a single attribute, use it to retrieve the tuples satisfying the condition (intermediate result)
- go through the intermediate result to check which records also satisfy the other conditions in main memory
- If there are access paths for both conditions, use them in the order which minimises the cost. Use the index that generates the smallest intermediate result first, as it has the biggest chance of fitting in memory.

Strategies for JOIN

JOIN is the most resource consuming operator.

SELECT *
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNUMBER

Join query: R.A = S.B

Naive Join

A naïve natural strategy, which does not require any access path.

1. Compute the Cartesian product of R and S, i.e. all tuples from R combined with all tuples from S
2. Store result in a file T and for concatenated tuples t = (r, s), check if r.A = s.B

Algorithm Naïve Join

T = Cartesian R x S

Scan T, a tuple $t \in$ T at a time: $t = (r, s)$

If r.A = s.B **then** add (r, s) to the result file

Else go to next tuple $t \in$ T.

Inefficient - the result we are looking for is typically a small subset of the Cartesian product.

Nested-Loop Join

A non-naïve natural strategy, which does not require any access path.

1. Load a set (chunk) of blocks from the outer relation R; Load one block from inner relation S; Maintain an output buffer for the matching tuples (r, s): r.A = s.B
2. Join the S block with each R block from the chunk; for each matching tuple (r, s) add to output buffer; if output buffer is full, pause and write the current join result to disk, continue
3. Load the next S block and goto step 2
4. goto step 1

Algorithm Nested-Loop Join

For each tuple $r \in \mathbf{R}$

  For each tuple $s \in \mathbf{S}$

    If $r.\mathrm{A} = s.\mathrm{B}$ **then** add (r, s) to the result file;

The outer and inner loops are over blocks and not over tuples.

Optimisation

Optimise by choosing the smaller relation (fewer blocks) to go in the outer loop.

- Total number of blocks read for *outer* relation E: $\mathbf{n_E}$
- **Outer Loops:** Number of *chunks* of $(n_B\text{-}2)$ blocks of *outer* relation read: $\mathbf{ceil(n_E/(n_B\text{-}2))}$
- For *each* chunk of $(n_B\text{-}2)$ blocks read *all* the blocks of *inner* relation D:
- Total number of block read in all outer loops: $\mathbf{n_D * ceil(n_E/(n_B\text{-}2))}$

Total Expected Cost: $\mathbf{n_E + n_D * ceil(n_E/(n_B\text{-}2))}$ **block accesses**

Refined Expected Cost: $b_D + (\mathrm{ceil}(b_D/(n_B\text{-}2)) \cdot b_E + (js \cdot |E| \cdot |D| / f_{RS})$

Index-Based Nested-Loop Join

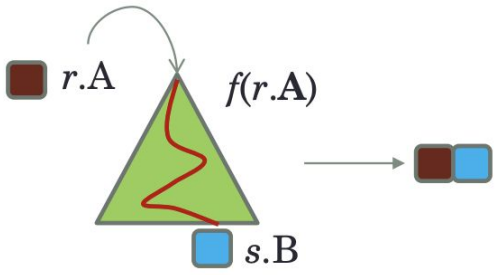
Use an index for either A or B joining attributes. Assuming an index f on attribute B of relation S.

Algorithm Index-Based Nested-Loop Join

For each tuple $r \in \mathbf{R}$

  Use *index* of B from **S** by $f(r.\mathrm{A})$, to *retrieve all* tuples $s \in \mathbf{S}$ having $s.\mathrm{B} = r.\mathrm{A}$

  For each *such* tuple $s \in \mathbf{S}$, add matching tuple (r, s) to the result file;



Much faster than Nested Loop Join, because we get immediate access to s where s.B = r.A by searching for r.A using the index f on B, avoiding linear search on S.

Optimisation

Use the PK index of the referenced relation (E) pointed by the FK of the referencing relation (D). Note: not for recursive FK-PK relationships, e.g., employee-supervisor.

Strategy Cost 2: $n_D + r_D*(x_E + 1) = 260$ **block accesses;**

Case: Primary Index on ordering / key

Refined Expected Cost: $b_E + |E| \cdot (x_D + 1) + (js \cdot |E| \cdot |D| / f_{RS})$

Case: Clustering Index on ordering/non-key

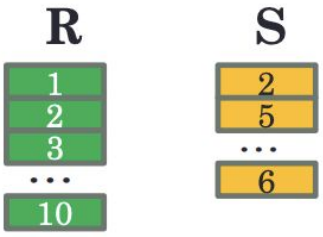
Refined Expected Cost: $b_D + |D| \cdot (x_E + \text{ceil}(s_E / f_E)) + (js \cdot |E| \cdot |D| / f_{RS})$

Case: B+ Tree Index on non-ordering / non-key

Refined Expected Cost: $b_D + |D| \cdot (x_E + 1 + s_E) + (js \cdot |E| \cdot |D| / f_{RS})$

Sort-Merge Join

Use of the merge-sort algorithm over two sorted relations with respect to their joining attributes. Requires that R and S are physically sorted by their joining attributes A and B.



1. Load a pair {R.block, S.block} of sorted blocks into the memory
2. Both blocks are linearly scanned concurrently over the joining attributes
3. If matching tuples are found, store them in a buffer

Efficient since there's no need to loop over R and S multiple times - they are scanned only once.

Strategy Cost: $n_E + n_D = 2{,}010$ **block accesses.**

If the files aren't initially sorted, use external sorting.

Total Strategy Cost:

$$n_E + n_D + 2 \cdot n_E + 2 \cdot n_E \cdot \log_2(\text{ceil}(n_E / n_B)) + 2 \cdot n_D + 2 \cdot n_D \cdot \log_2(\text{ceil}(n_D / n_B))$$

Refined Expected Cost: $b_R + b_S + (js \cdot |R| \cdot |S| / f_{RS})$

Hash Join

Requires that file R is partitioned into M buckets with respect to hash function h over attribute A, and that file S is partitioned into M buckets with respect to the same hash

function h over attribute B. Assuming R is he smallest file and M buckets of R fit into memory.

Algorithm Hash-Join

/*Partitioning phase */

For each tuple $r \in \mathbf{R}$,

Compute $y = h(r.\mathrm{A})$ /* address of bucket*/

Place tuple r into *bucket* $y = h(r.\mathrm{A})$ in memory

/*Probing phase*/

For each tuple $s \in \mathbf{S}$,

Compute $y = h(s.\mathrm{B})$ /*use the same hash function h*/

Find the *bucket* $y = h(s.\mathrm{B})$ in memory (of the **R** partition)

For *each* tuple $r \in \mathbf{R}$ in the bucket $y = h(s.\mathrm{B})$

If $s.\mathrm{B} = r.\mathrm{A}$ **add** (r, s) to the result file; /*join*/

Can be optimised by choosing the best relation to hash first.

Expected Cost: $3(n_E + n_D)$ block accesses.

Refined Expected Cost: $3 \cdot (b_R + b_S) + (js \cdot |R| \cdot |S| / f_{RS})$

Query Optimisation

- **Selection Selectivity:** fraction of tuples satisfying a condition.
- **Join Selectivity:** fraction of matching tuples in a Cartesian space.

Cost-based Optimisation

Information for each Relation File:

- number of records (r); (average) size of each record (R)
- number of blocks (b); blocking factor (bfr) i.e., records per block
- Primary File Organization: heap, hash, or sequential file
- Indexes: primary, clustering index, secondary index, B+ Trees.

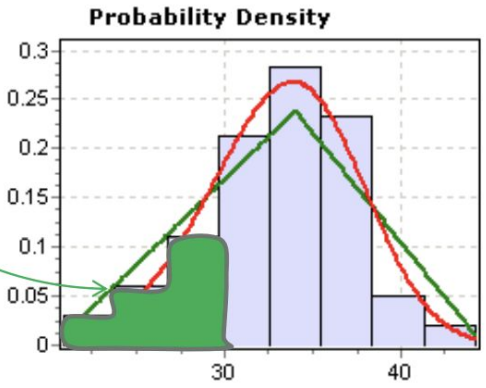
Information for each Attribute A:

- Number of Distinct Values (NDV) of attribute A
- Domain range: [min(A), max(A)]
- Type: continuous or discrete attribute; key or non-key
- Level of Index of the attribute A, if exists
- Probability Distribution Function P(A = x) indicating the frequency (probability) of each value x of the attribute A in the relation.
- A good approximation of a distribution: histogram

P(Salary ≤ 30K) = *integral*:
0 to 30 = 0.18 or 18%

Meaning 1: 18% of tuples have Salary ≤ 30K
Meaning 2: Each tuple has Salary ≤ 30K *w.p.* 18%

P(Salary = 30K) = 0.12 or 12%



Histogram of Salary (£K)

Selection Selectivity

selection selectivity $sl(A)$ of attribute A is a *real* number: $\mathbf{0 \leq sl(A) \leq 1}$

- $sl(A) = 0$: *none* of the records satisfies a condition over attribute A.

```
SELECT * FROM EMPLOYEE WHERE Salary = 1,000,000,000
```

- $sl(A) = 1$: *all* the records satisfy a condition over attribute A

```
SELECT * FROM EMPLOYEE WHERE Salary > 0
```

- $sl(A) = x$: $x\%$ of the records satisfy a condition over attribute A

```
SELECT * FROM EMPLOYEE WHERE Salary = 40000
```

Hence: $\mathbf{0 \leq sl(A) \leq 1}$ *or* as percentage: $\mathbf{0\% \leq sl(A) \leq 100\%}$
***i.e., probability* that a tuple satisfies a selection condition!**

Selection Cardinality

s = r . sl(A) ∈ [0, r] - average number of tuples satisfying a condition over attribute A.

Selectivity Prediction

Approximation

- Approximate distribution via a histogram
- No assumptions
- Accurate, but creates maintenance overhead

$\mathbf{sl(A = x) \approx P(A = x)}$, which *depends* on the value of $x \in [\min(A), \max(A)]$

Uniformity assumption

- Assume all values are uniformly distributed - equally probable
- No need for histogram
- Less accurate

$\mathbf{sl(A = x) \approx constant\ independent\ of\ the\ x\ value;}\ \forall x \in [\min(A), \max(A)]$

- For a key attribute A, sl(A = x) = 1/r, since only one tuple satisfies a condition
- For a non-key attribute A, sl(A = x) = 1/NDV(A) = 1/n

Range Selection Selectivity

```
SELECT * FROM RELATION WHERE A ≥ x
```

Definition 1: Domain range: **max(A) – min(A)**; A ∈ [min(A), max(A)]
Definition 2: Query range: **max(A) – *x*;** $x \in [\min(A), \max(A)]$

$sl(A \geq x) = 0$ if $x >$ max(A)
$sl(A \geq x) = (\max(A) - x)/(\max(A) - \min(A)) \in [0, 1]$

min = 100 x=1000 10000 = max

```
SELECT * FROM EMPLOYEE WHERE Salary ≥ 1000;
```

Salary ∈ [100, 10000]; r = 1000 employees **evenly distributed** among salaries:
sl(Salary ≥ 1000) = (10000-1000)/(9900) = 0.909 (**90.9%**) *or* s = 909 employees.

Conjunctive Selectivity

```
SELECT * FROM RELATION WHERE (A = x) AND (B = y)
```

$$sl(Q) = sl(A = x) \cdot sl(B = y) \in [0, 1]$$

```
SELECT * FROM EMPLOYEE
WHERE    DNO = 5 AND Salary = 40000;
```

NDV(Salary) = 100, NDV(DNO) = 10, r = 1,000 employees **evenly distributed** among salaries **and** departments:

- *Salary is independent of the department (accept?)*
- sl(Salary = 40K) = 1/NDV(Salary) = 1/100 = 0.01
- sl(DNO = 5) = 1/NDV(DNO) = 1/10 = 0.1

$sl(Q) = sl$(Salary) $\cdot sl$(DNO) = (1/10)·(1/100) = 0.001 or only s = 1 tuple.

P(A ∩ B) = P(A)·P(B) = *joint probability an employee satisfying both conditions, given that condition A is statistically independent of condition B.*

Disjunctive Selectivity

```
SELECT * FROM RELATION WHERE (A = x) OR (B = y)
```

$$sl(Q) = sl(A) + sl(B) - sl(A) \cdot sl(B) \quad \in [0, 1]$$

```
SELECT * FROM EMPLOYEE
WHERE    DNO = 5 OR Salary = 40000;
```

NDV(Salary) = 100, NDV(DNO) = 10, r = 1000 employees **evenly distributed** among salaries **and** departments:

- *Salary is independent of the department (accept?)*
- sl(Salary) = 1/NDV(Salary) = 1/100 = 0.01
- sl(DNO) = 1/NDV(DNO) = 1/10 = 0.1

sl(Q) = (10/100) + (1/100) - (1/10)·(1/100) = 0.109 or s = 109 tuples.

P(A U B) = P(A) + P(B) - P(A)·P(B) = *probability an employee satisfying either condition A or condition B; both conditions are statistically independent.*

- Naïve Join Cost: $n_E * n_D$: **20,000 block accesses**
- Nested-Loop Cost (*best*): $n_D + n_E * \text{ceil}(n_D/(n_B-2))$: **4,010 block accesses**
- Index-based Nested-Loop Cost (*best*): $n_D + r_D*(x_E + 1)$: **260 block accesses**
- Sort-Merge Cost (*already* sorted): $n_E + n_D$: **2,010 block accesses**
- Hash-Join Normal-Case Cost: $3(n_E + n_D)$: **6,030 block accesses**

Joint Selectivity

Join selectivity (js) is the fraction of the matching tuples between the relations R and S out of the Cartesian cardinality (#of concatenated tuples).

$$js = |R \bowtie S| / |R \times S| \text{ with } 0 \leq js \leq 1.$$

Theorem 1. Given n = NDV(A, **R**) and m = NDV(B, **S**):

$$js = 1 / \max(n, m)$$

$$jc = (|\mathbf{R}| \cdot |\mathbf{S}|) / \max(n, m)$$

Proof. *Beyond the scope…(last slide)*