07- File Organization and Indexing

School of Computer Science
University of Windsor

Dr. Shafaq Khan

Agenda

- **Lecture**
  - Organizing the data on the disk
  - Introduction to Indexing
  - Single-Level Ordered Indexes
    - Primary Indexes
    - Clustering Indexes
    - Secondary Indexes
  - Multilevel Indexes
    - Two-Level Primary Indexing
  - Dynamic Multilevel Indexes
    - B Trees
    - B+ Trees
  - CREATE INDEX Command in SQL
- **Assignment-2 Quiz**

Announcements

P2: Milestone report submission

(Sec 2: Feb 27; Sec 3: Feb 28; Sec 4: Mar 1)

P2: Milestone presentation: Next week (Feb 28 – Mar 2)

Introductory Questions

Why is indexing important?

What is the aim of having an index?

What are the downsides of indexes?

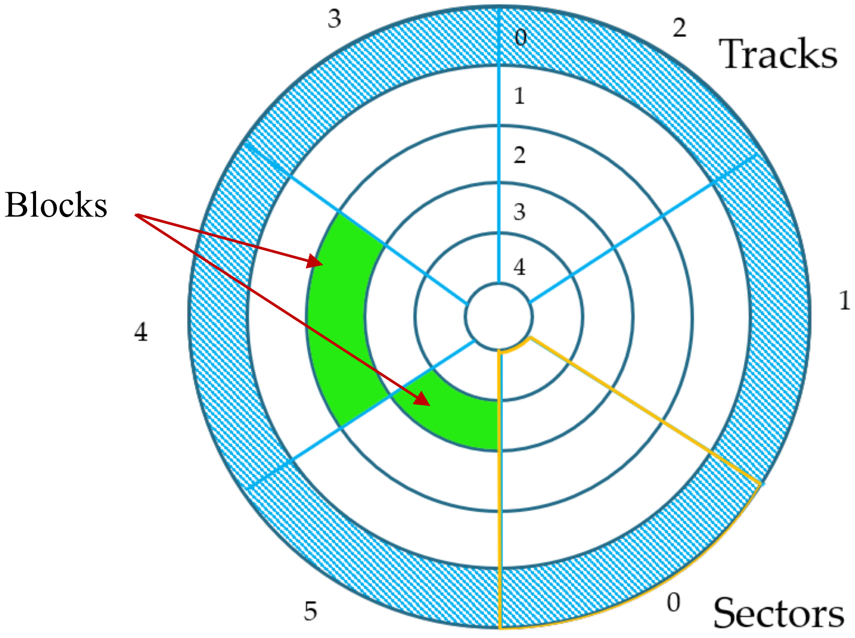
What are the different types of indexes used for query optimizations?

What to choose for creating indexes?

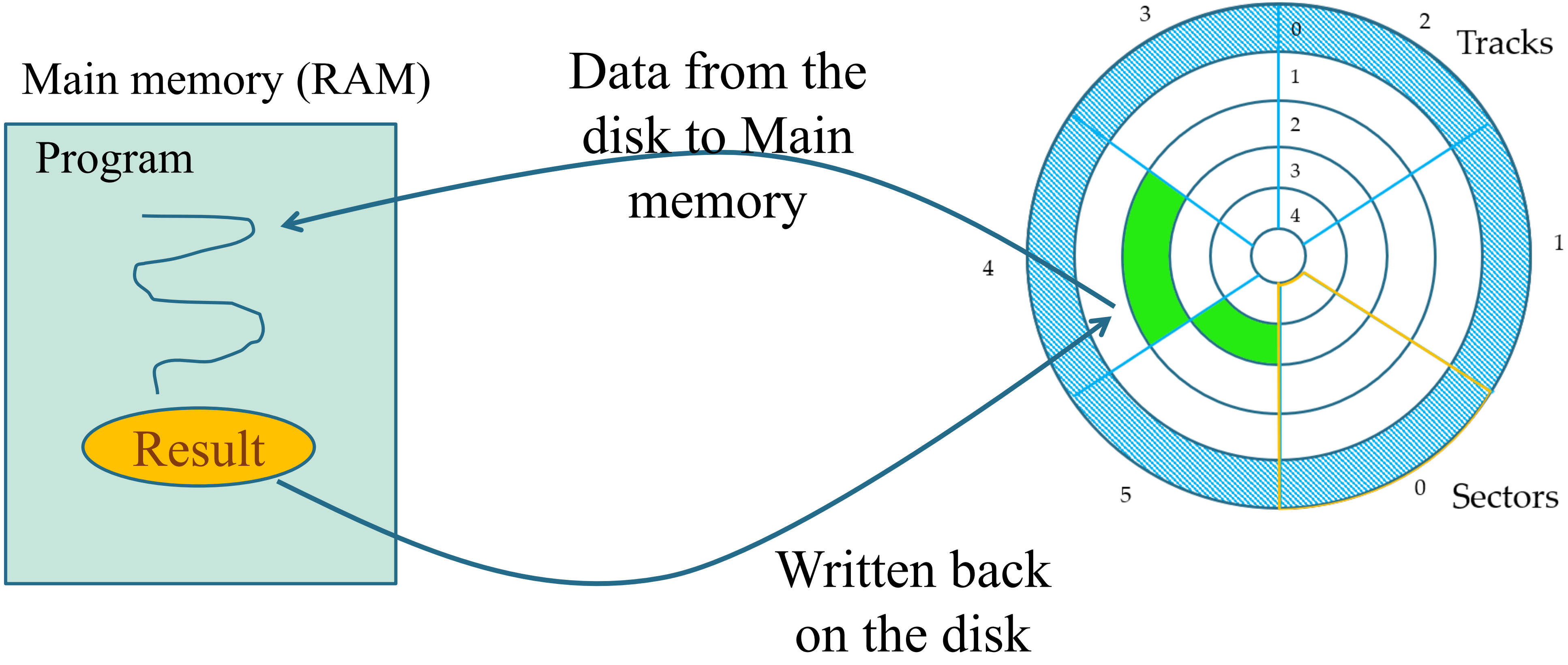
Disk Structure

$$\mathit{Block\ Address} = \langle \mathit{Track\ Num}, \mathit{Sector\ Num} \rangle$$

Let's assume a block size is 512 bytes

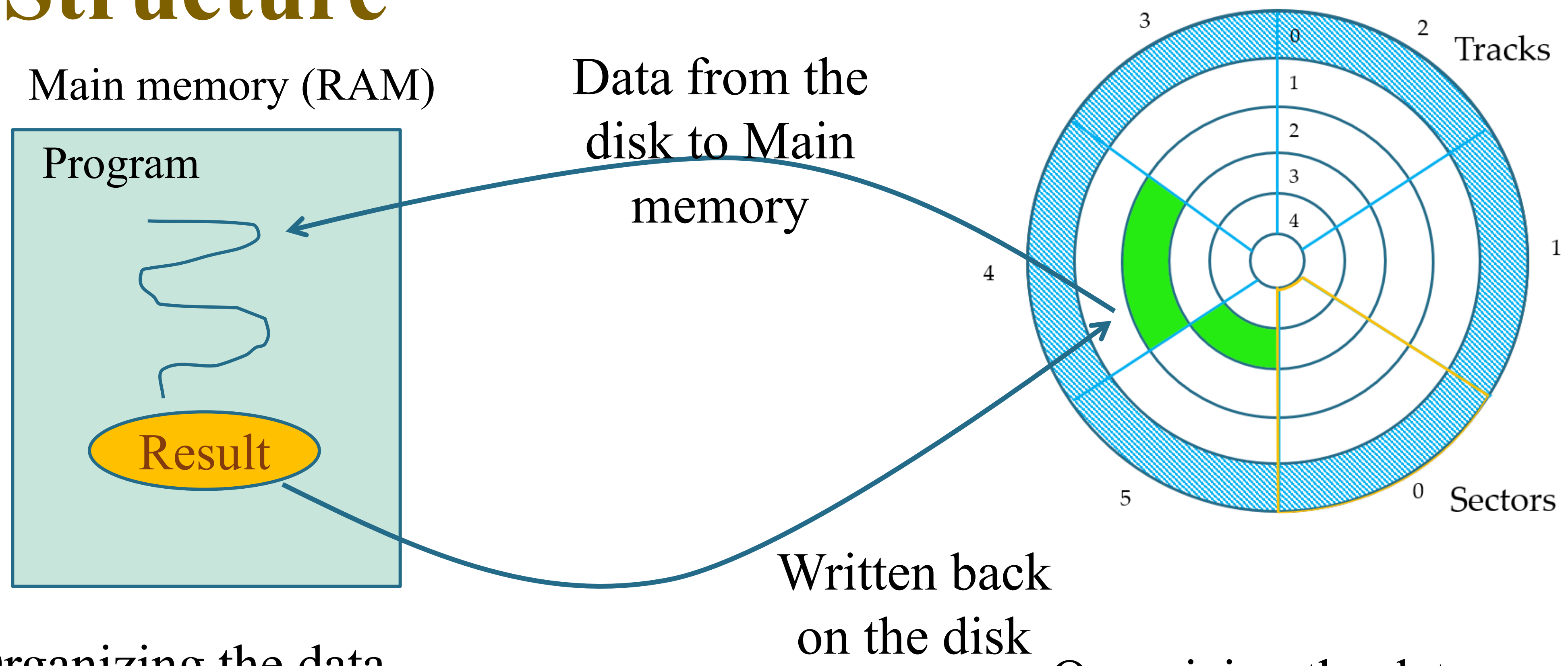


Disk Structure



So, the data cannot be directly processed upon the disk it has to be brought into the main memory and then access .

Disk Structure



Organizing the data inside the main memory that is directly used by the program is **Data Structures**.

Organizing the data on the disk efficiently so that it can be easily utilized that is **DBMS.**

University of Windsor

How is data stored on Disk?

Employees

Fields	Size
Eid	10
Name	50
Depart	30
Gender	8
Salary	30
Total size	**128 bytes**

In each block how many rows can be stored?

Number of Records/ Block = 512/128 = **4**

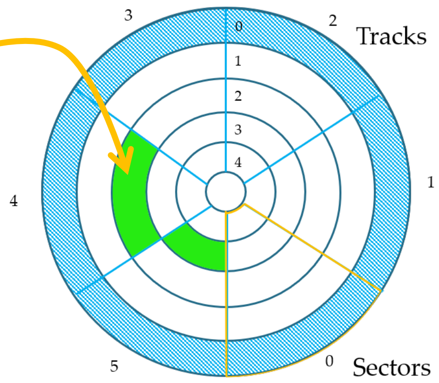
Employees

128 bytes

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

100 records

What is the size of a record?



Block Size = 512 bytes

How is data stored on Disk?

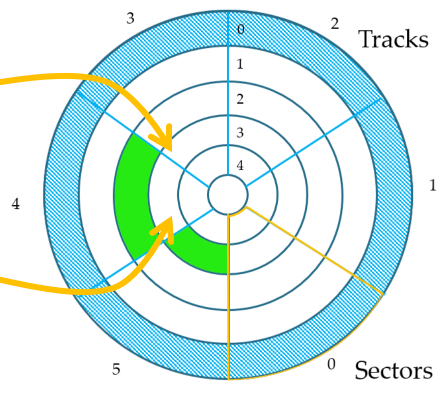
Employees

Fields	Size
Eid	10
Name	50
Depart	30
Gender	8
Salary	30
Total size	**128 bytes**

SELECT Eid=7
FROM Employees

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager



Block Size = 512 bytes

Number of Records/ Block = 512/128
= 4

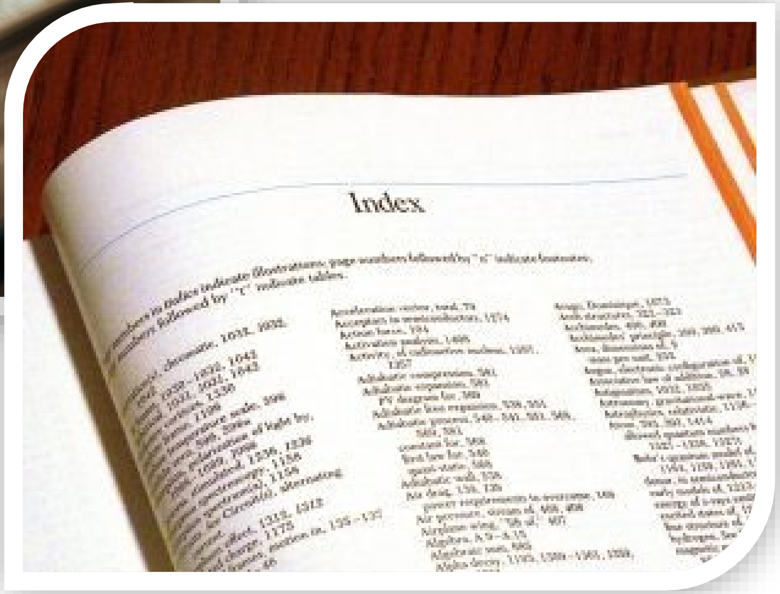
How many blocks are required for 100 records?

100 Records can be stored in = 100/4 or 12800/512
= **25 blocks**

100 records

What is Indexing?

An index is a structure that provides accelerated access to the rows of a table based on the values of one or more columns.



It makes our search simpler and quicker.

- ✓ How do we create the indexes?
- ✓ How these indexes help to access the data?

Motivation- Searching for a Key Value

- Ex: Unsorted List
  - 1024x1024x1024 = 1,073,741,824 (I Billion elements –Approx.)
  - Takes **1 Billion comparisons** in the worst case
- Ex|: Sorted List
  - 1024x1024x1024 = 1,073,741,824 (I Billion sorted elements –Approx.)
  - Takes **30 comparisons** in the worst case
- 30 comparisons vs. 1 Billion comparisons
- Overhead involved in sorting is much lesser than that in searching?

What is Indexing?

Where do we store the index?

How many blocks for storing index?

Index

Key	Pointer
1	
2	
3	
4	
5	

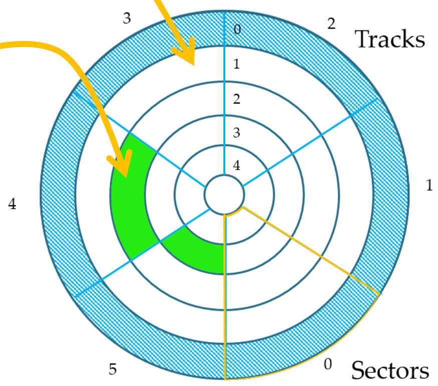
100 records

Dense Index

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

100 records



Block Size = 512 bytes

What is Indexing?

Index

Fields	Size
Eid	10
Pointer	6
Total size	**16 bytes**

Index

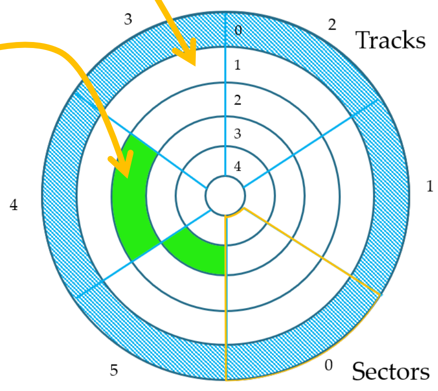
Key	Pointer
1	
2	
3	
4	
5	

100 entries

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

100 records



Block Size = 512 bytes

1 index entry requires : 16 bytes
100 entries require: 1600 bytes
100 entries can be stored in = 1600/512= 3.1 ≅ 4 blocks

What is Indexing?

SELECT Eid=7
FROM Employees

100 entries can be stored in = 1600/512
≅ **4 blocks**

Total number of blocks required = 4 +1
= 5 blocks

Index

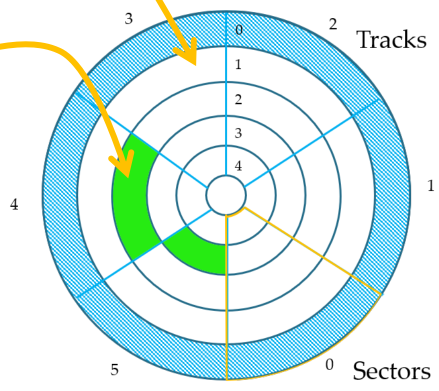
Key	Pointer
1	
2	
3	
4	
5	

100
entries

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

100 records



Block Size = 512 bytes

What is Indexing?

Index

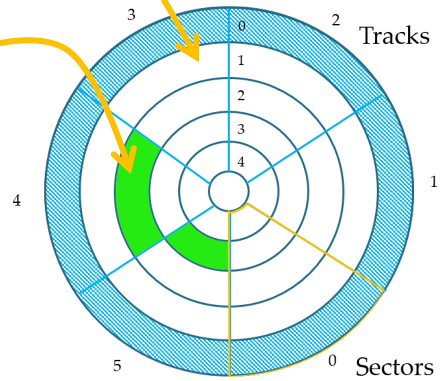
Key	Pointer
1	
2	
3	
4	
5	

1000 entries

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

1000 records



Block Size = 512 bytes

What is multi-level indexing?

Index

Key	Pointer
1	
33	

Index

Key	Pointer
1	
2	
3	
4	
5	

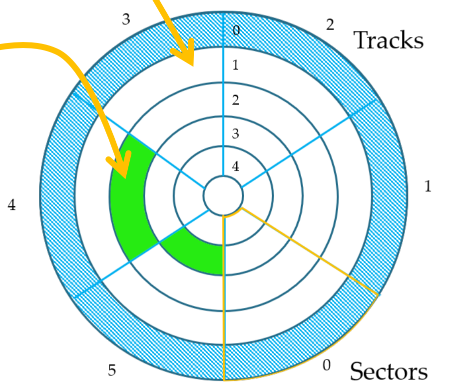
1000 entries

We require 40 blocks.

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

1000 records



Block Size = 512 bytes

Can we have an index above an index?

We require 250 blocks

What is multi-level indexing?

Index

Key	Pointer
1	
33	
65	

Sparse Index

Number of entries/ Block = 512/16
= 32

40 entries can be stored in = 40/32 = 1.25

$\cong$ **2 blocks**

Index

Key	Pointer
1	
2	
3	
4	
5	

32
entries

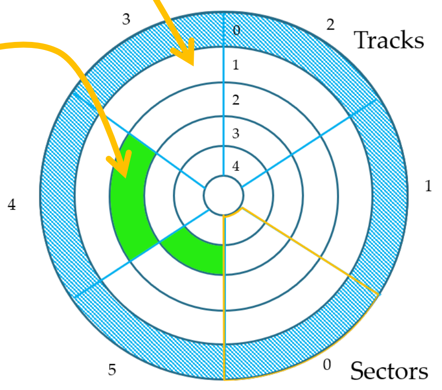
32 entries

40 blocks

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

1000 records



Block Size = 512 bytes

We require 250 blocks

What is multi-level indexing?

Index

Key	Pointer
1	
33	
65	

Sparse Index

Index

Key	Pointer
1	
2	
3	
4	
5	

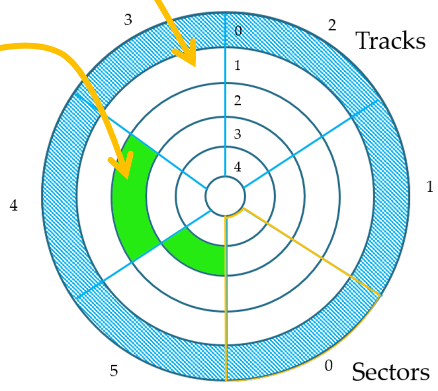
32 entries

32 entries

40 blocks

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager



Block Size = 512 bytes

1000 records

We require 250 blocks

What is multi-level indexing?

SELECT Eid=7 FROM Employees ?

Index

Key	Pointer
1	
33	
65	

Index

Key	Pointer
1	
2	
3	
4	
5	

32 entries

32 entries

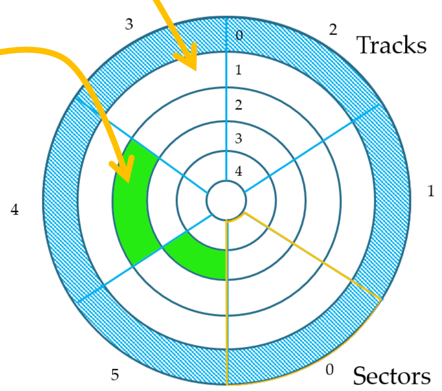
40 blocks

Employees

Eid	Name	Depart	………
1	Jenkins	Manager	
2	Williams	Sales Rep	
3	Smith	Sales Rep	
4	Crosby	Manager	
5	Albright	Secretary	
6	Sawyer	Sales Rep	
7	Thomas	Secretary	
8	Albright	Worker	
9	Crawford	Manager	
⋮	⋮	⋮	

1000 records

We require 250 blocks



Block Size = 512 bytes

Number of entries/ Block = 512/16 = 32

40 entries can be stored in = 40/32 = 1.25 ≅ 2 blocks

Total number of blocks required = 2 +1 + 1 = **4 blocks**

What is multi-level indexing?

Index

Key	Pointer
1	
33	
65	

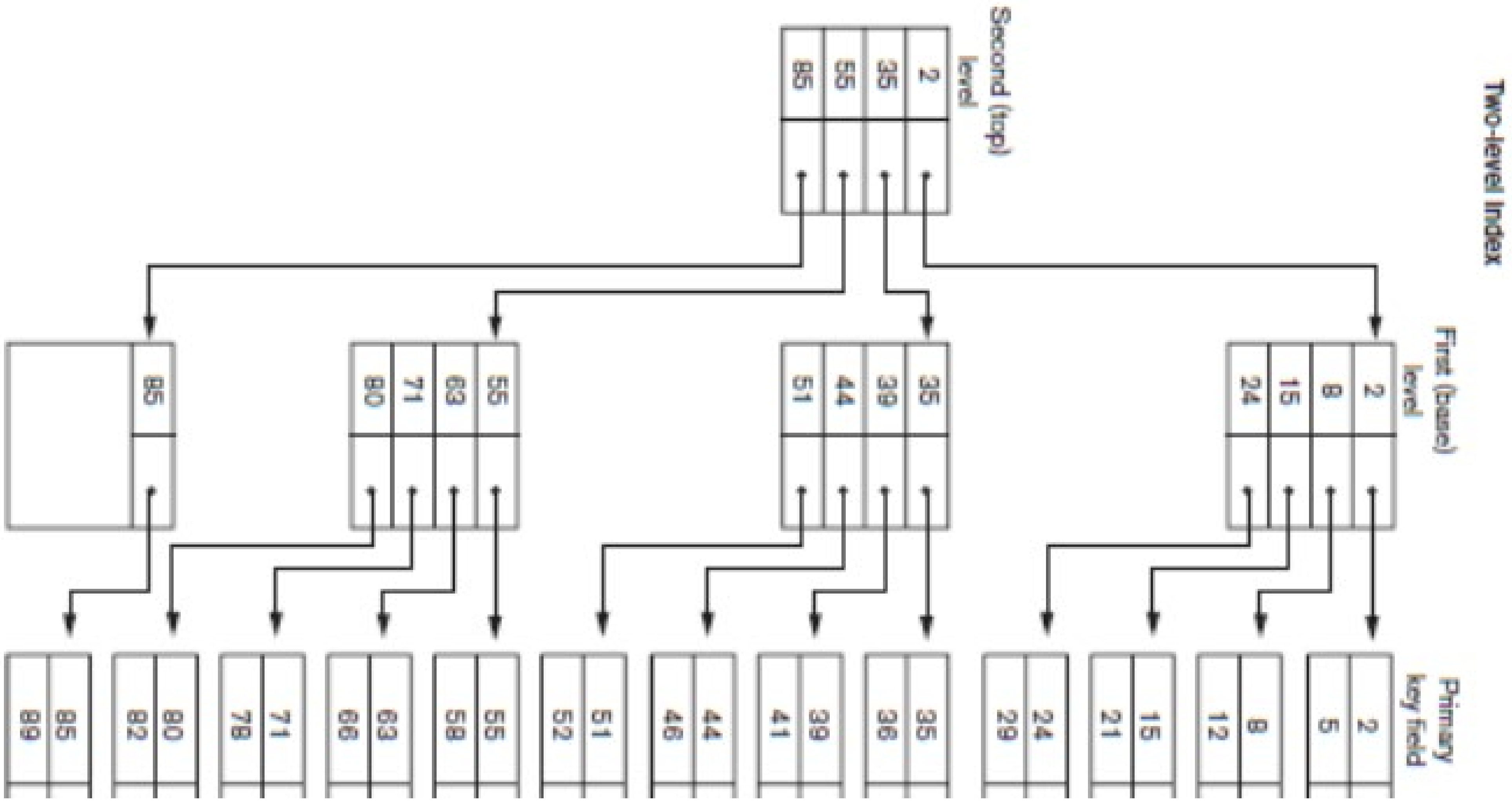
Index

Key	Pointer
1	
2	
:	
32	
33	
34	
:	
64	
65	
67	
:	
78	

Employees

Eid	Name	Depart
1	Jenkins	Manager
2	Williams	Sales Rep
3	Smith	Sales Rep
4	Crosby	Manager
5	Albright	Secretary
6	Sawyer	Sales Rep
7	Thomas	Secretary
8	Albright	Worker
9	Crawford	Manager

Sparse Index


Two-level Index
Second (top) level
First (base) level
Primary key field

Indexing cont.

- Database Indexes are special files which are stored along with the Data files of the Database
- Database indexes help the query processor to **expedite** search and retrieval of records from Blocks
- By default, the DBMS typically creates an index based on the PK of each table
  - However, depending on the type and frequency of queries, <u>users can create their own indexes for better overall performance</u>
- When queries corresponding to particular tables are executed, the corresponding index files (if any) are transferred to the main memory for the **query optimization** module to work on

Single-Level Ordered Indexes

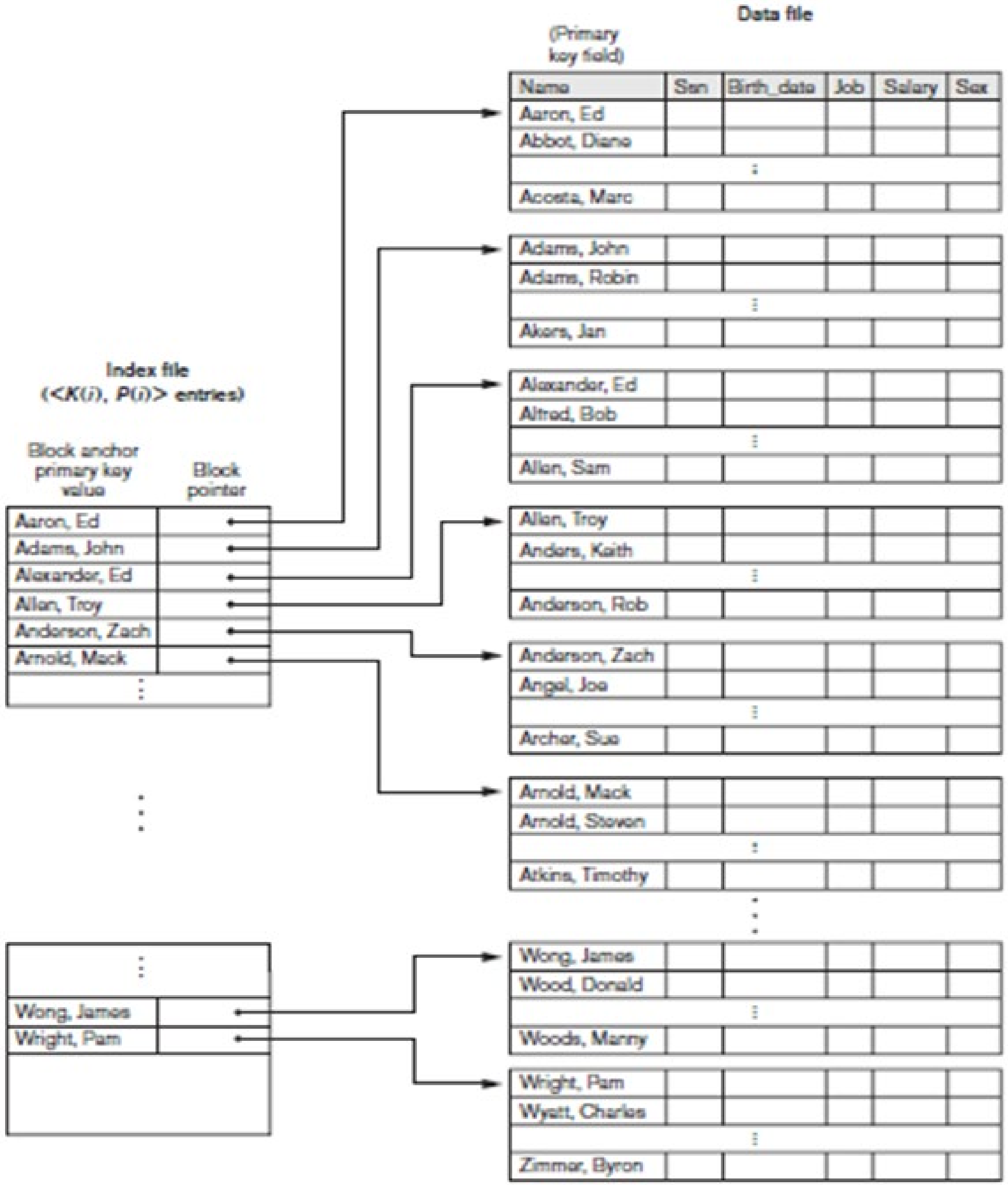
Single-Level Ordered Index

- Primary Index
- Clustered Index
- Secondary Index

Primary Indexes

- Records are **sorted and stored** across blocks based on a key
- The first Record in each block is called the **Block anchor**
- Each block anchor has an entry in the index file that has a **pointer** to the block

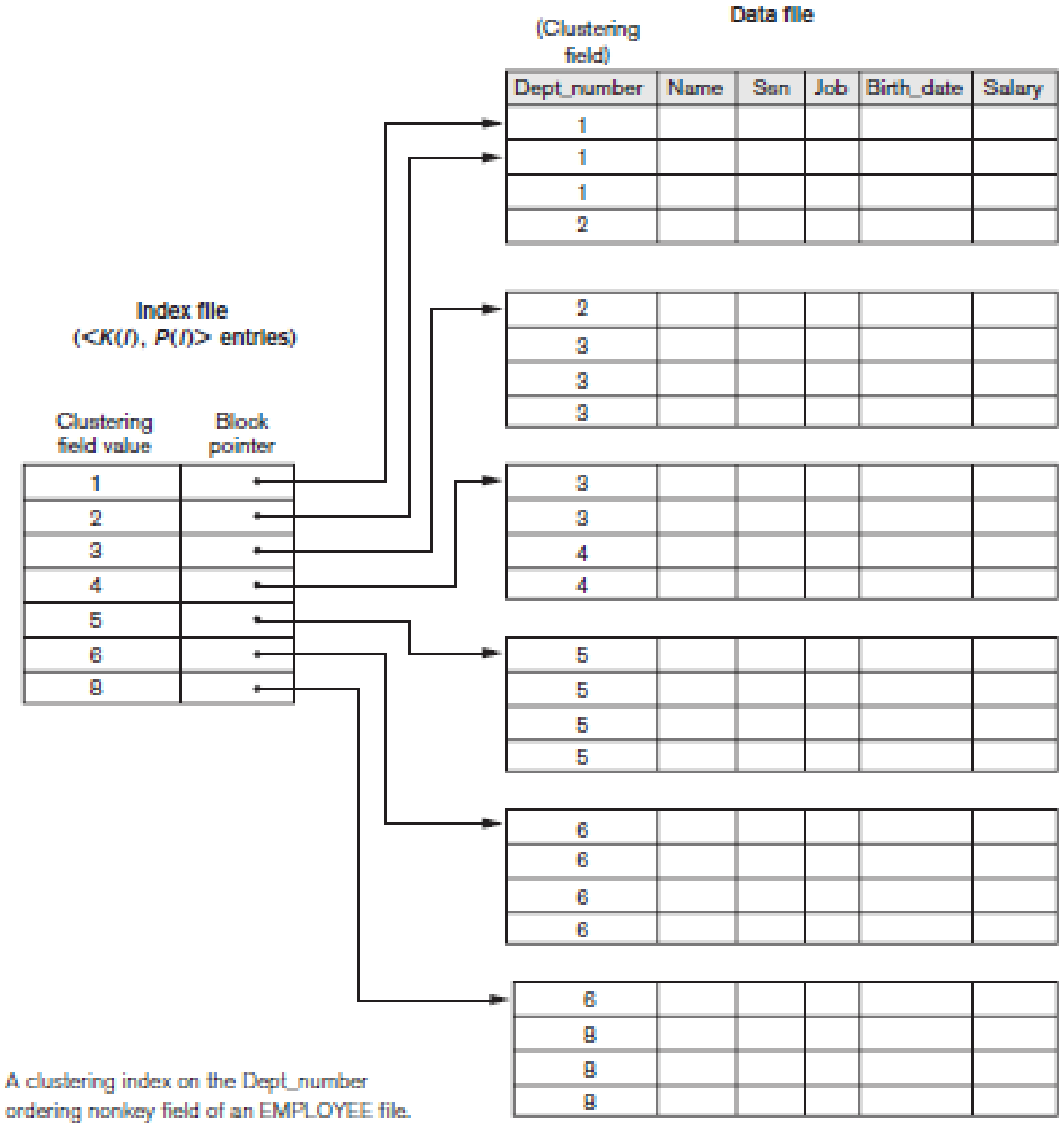
Is it dense or sparse index ?



Clustering Index

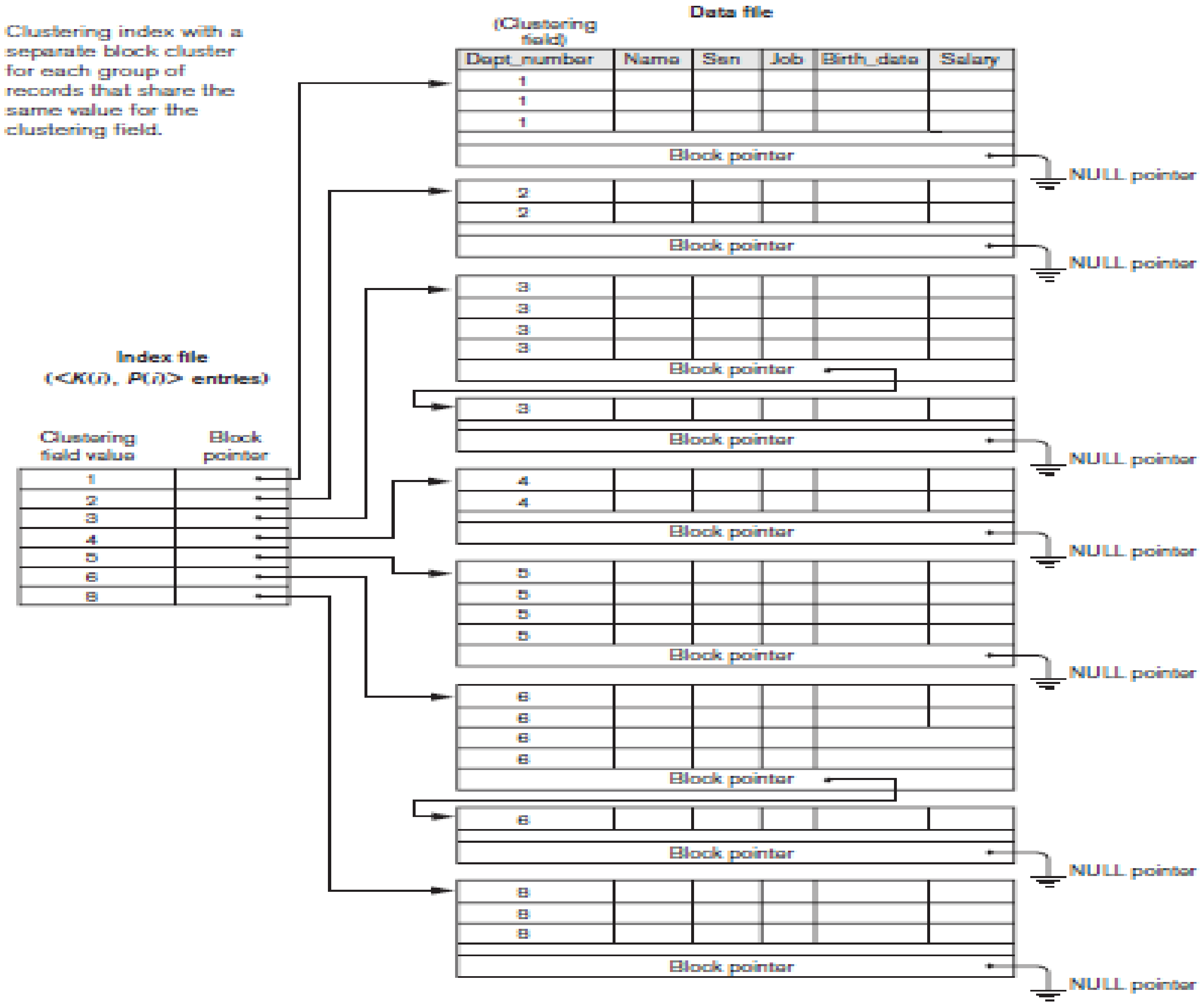
- Records are **sorted and stored** across blocks based on a **Clustering Field value** (non-key value)

- Each Clustering Field value has an entry in the index file that has a **pointer** to the **first block** that contains the Clustering Field value



A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

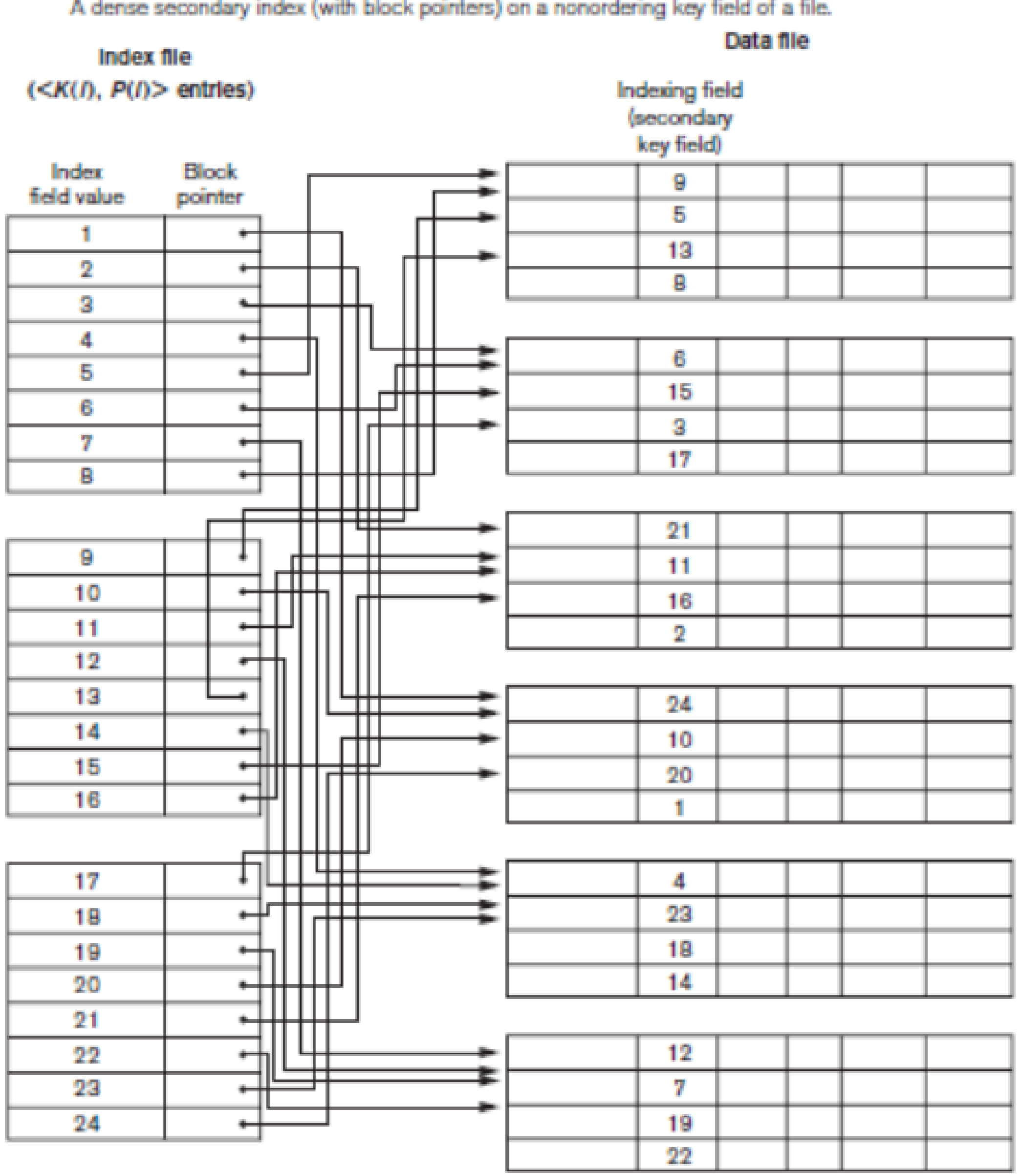
Clustering Indexes with Separate Block Clusters



University of Windsor

Secondary Index

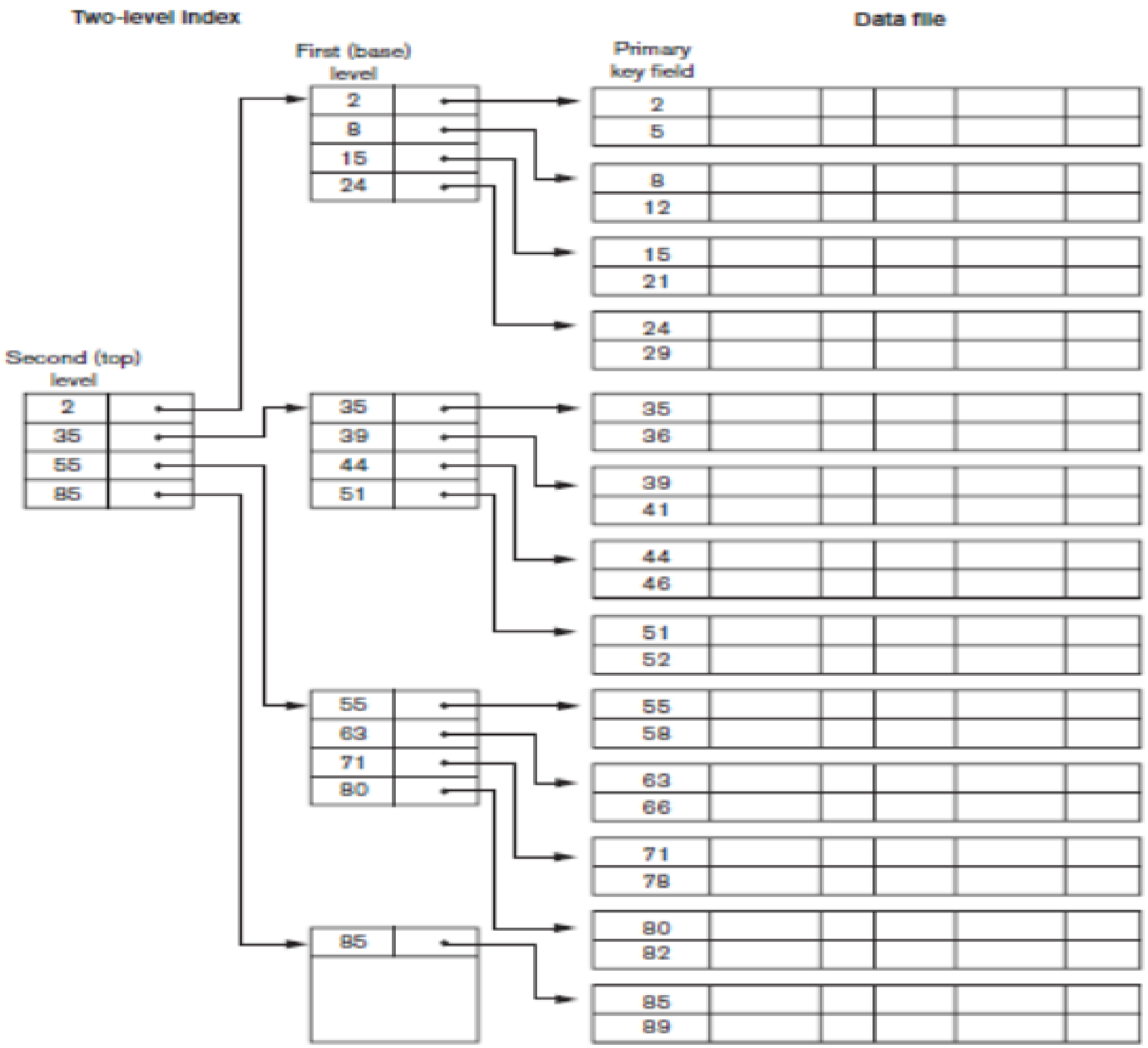
- Indexes based on **other key values** (which are not the PK)
- The records in this case are **not sorted** across the blocks based on the secondary key fields (Since they are sorted using the PK)
- Anchoring is not possible (Like in Primary Index)



University of Windsor

Multilevel Indexes

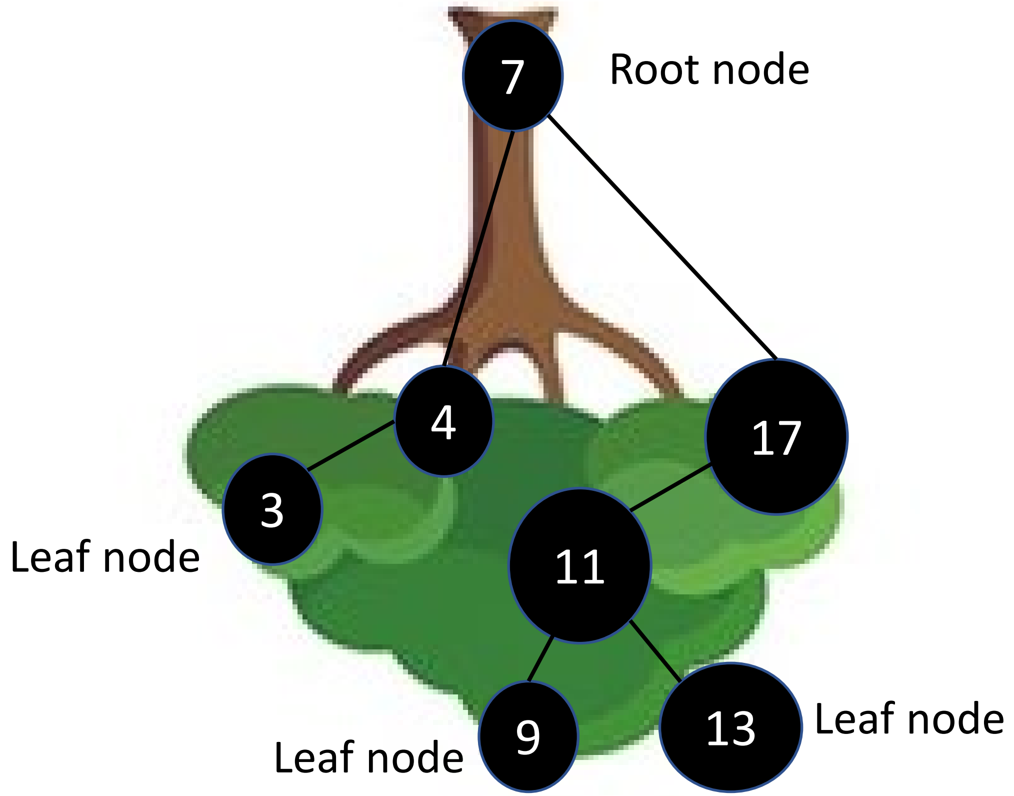
Two-Level Primary Index



Dynamic Multilevel Indexes using B Trees and B+ Trees

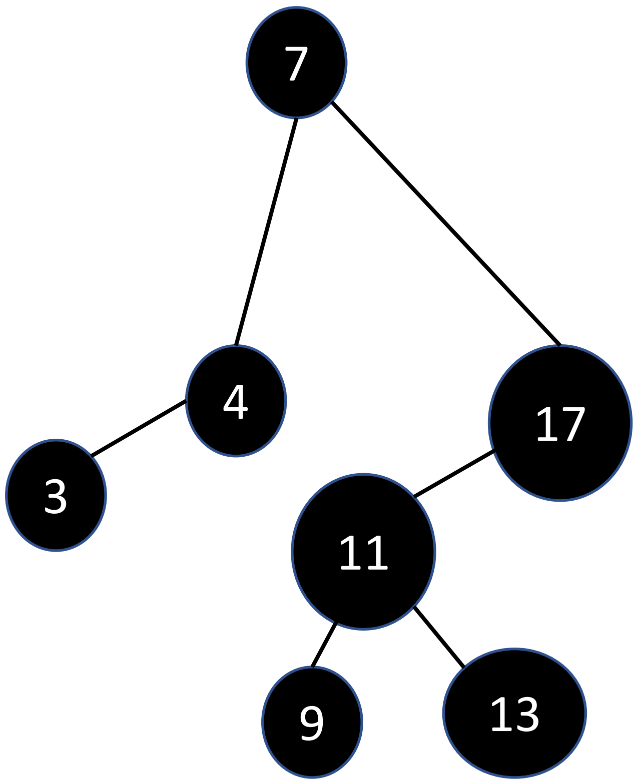
TREE

Tree: An organizational structure for the storage and retrieval of data

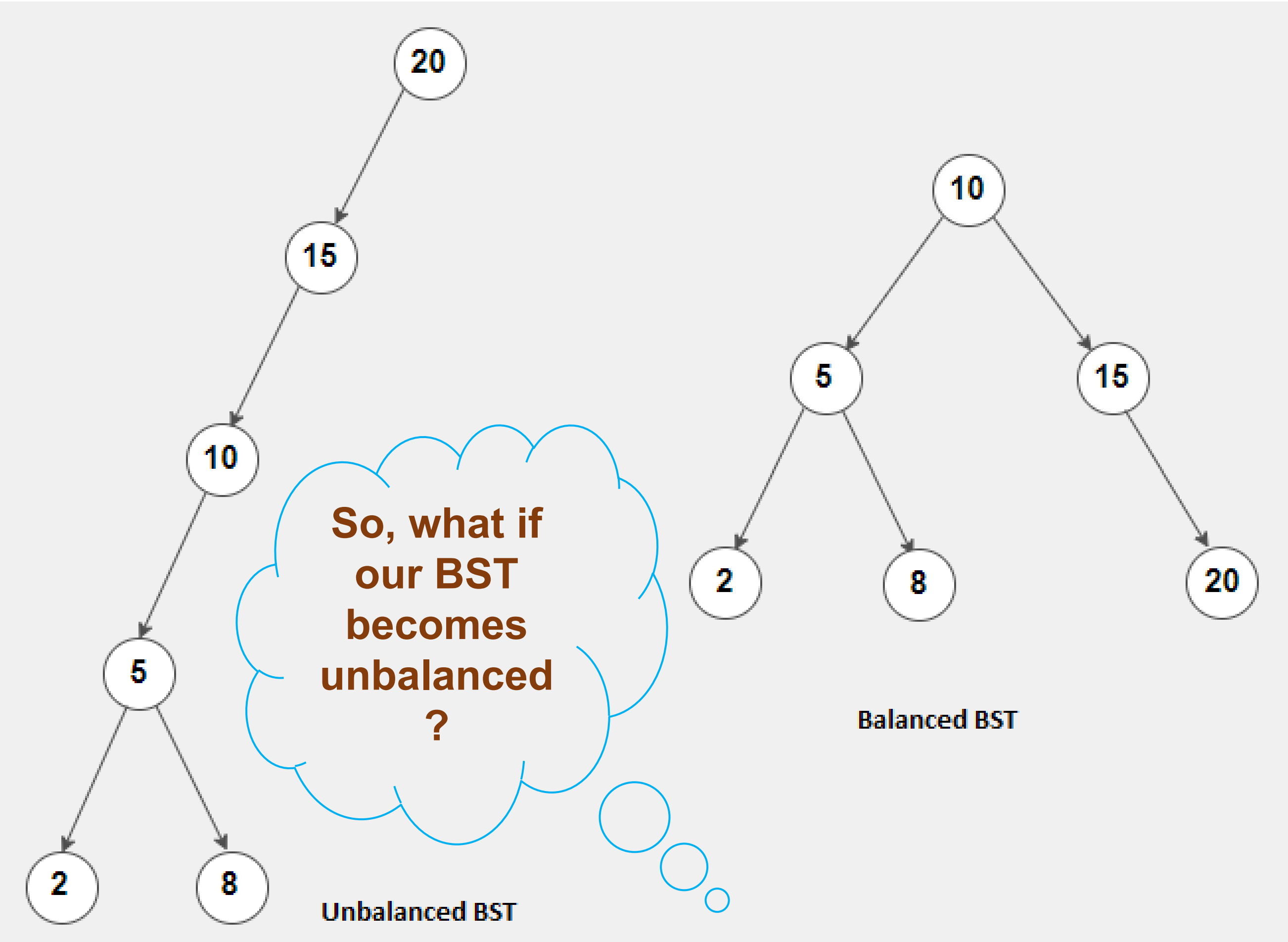


Binary Search Tree

- Binary search trees allow binary search for fast lookup, addition, and removal of data items.
- A node can have utmost two children.



Binary Search Tree BST (Balanced vs. Unbalanced)



- **Balanced BST:** The difference between the height of the left sub-tree and the right sub-tree of **every node** is not more than 1

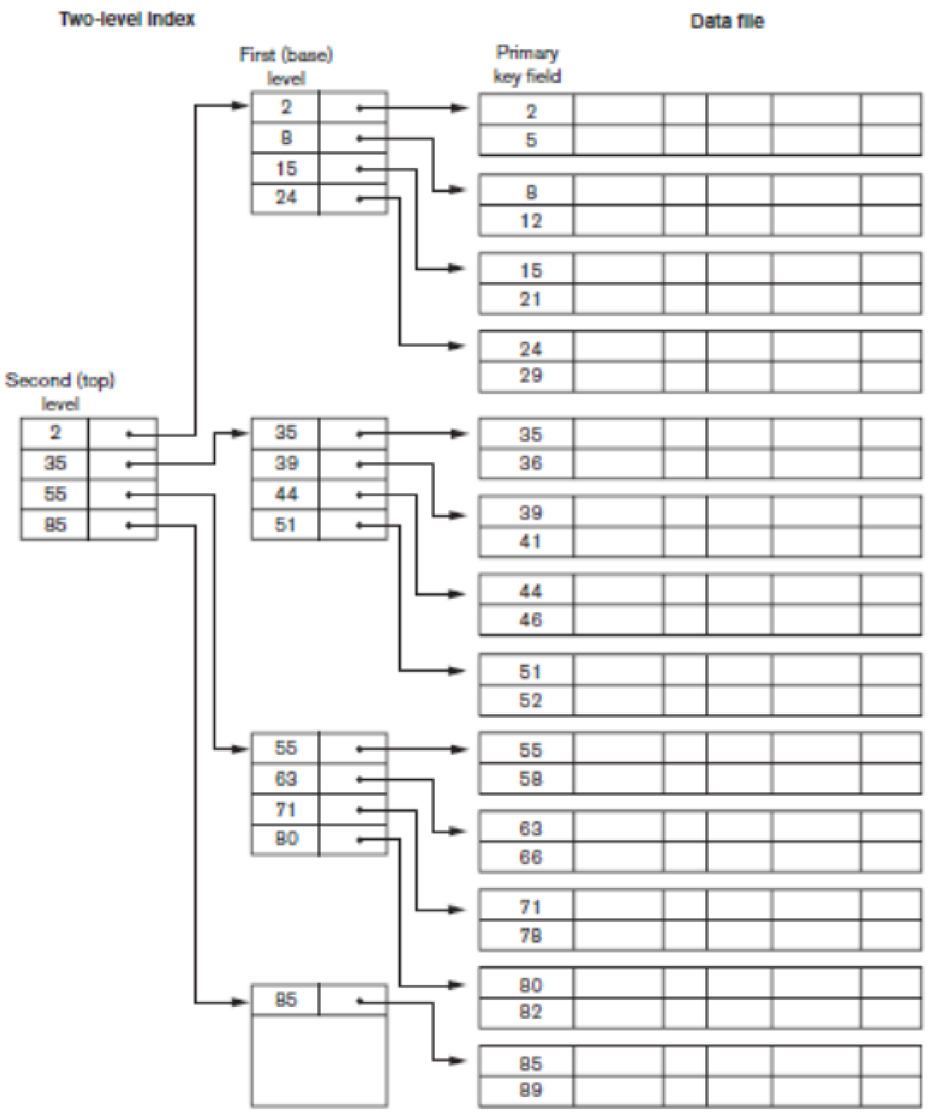
- The time complexity of operations on the binary search tree is directly proportional to the height of the tree.
  Balanced BST: O(log(n))
  Unbalanced BST: O((n))

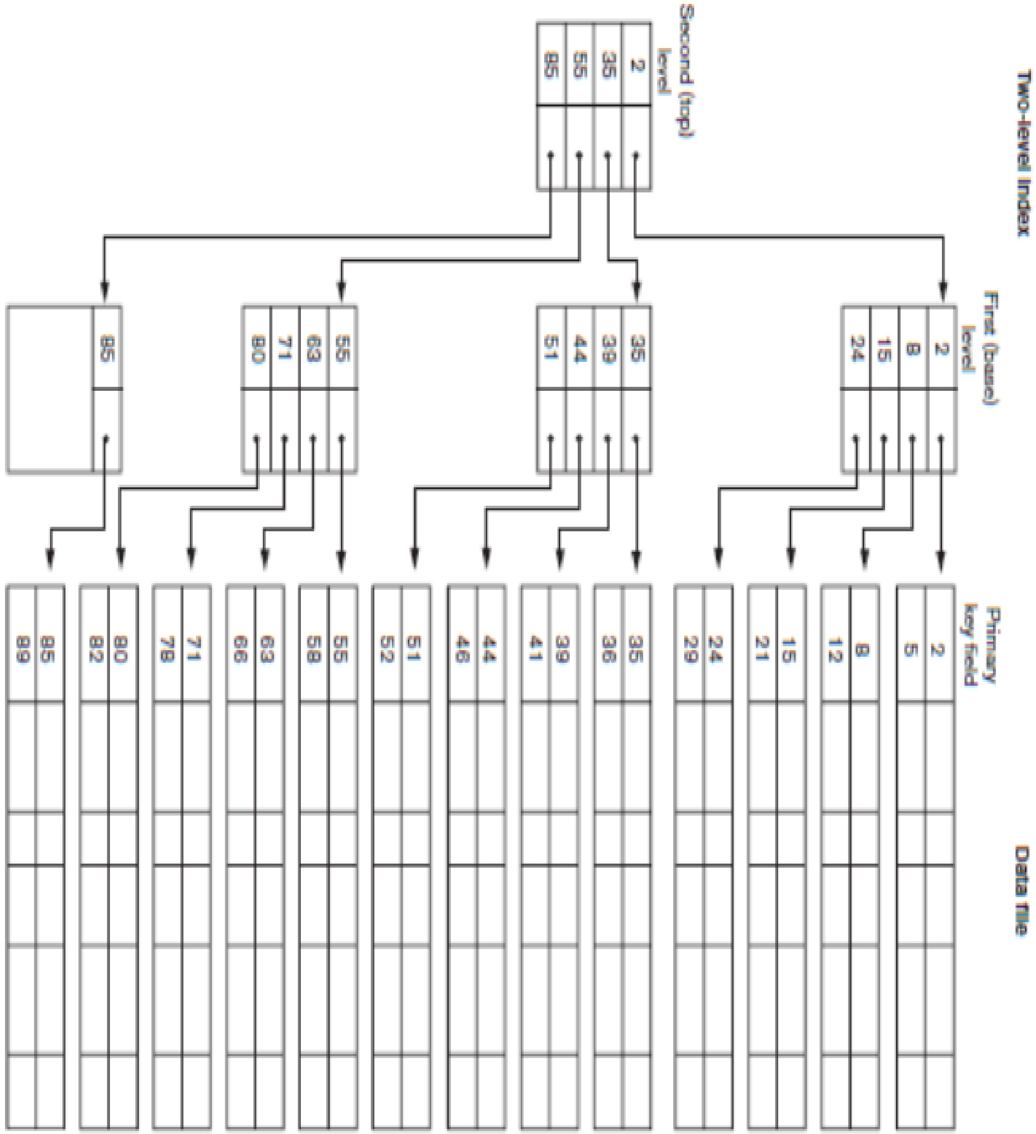
- It can also be expressed in terms of its height (h) as *O(h)*

- A balanced BST minimizes the search time by minimizing the value of *h*

B-Tree

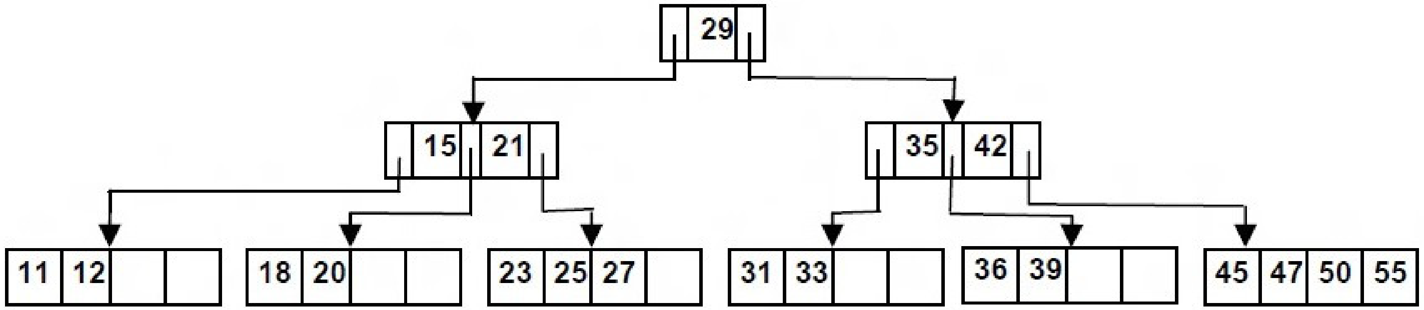
- A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
- Invented by Rudolf Bayer and Ed McCreight in 1972
- The B-tree **generalizes** the binary search tree
  - Allows nodes with more than two children (Whereas a node in a BST cannot have more than two children)
- Useful for implementing multi-level indexing


Two-level index
Data file
First (base) level
Primary key field
Second (top) level


Two-level index
Second (top) level
First (base) level
Primary key field
Data file

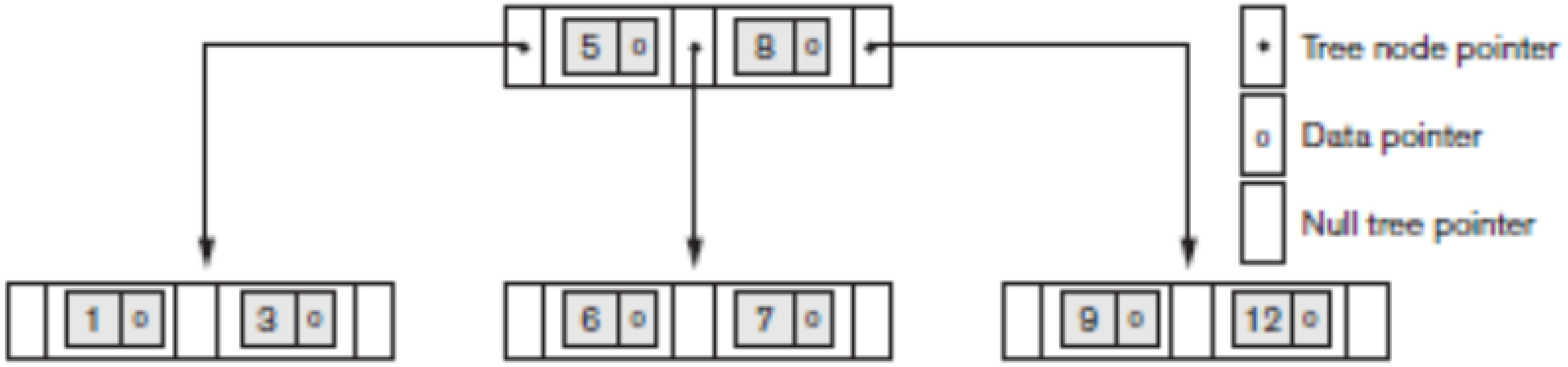
B Tree Rules

- **A B-tree of order m is a tree which satisfies the following properties:**
  - Every node has at most m children.
  - Every non-leaf node (except root) has at least ⌈m/2⌉ child nodes.
  - The root has at least two children if it is not a leaf node.
  - A non-leaf node with k children contains k − 1 keys.

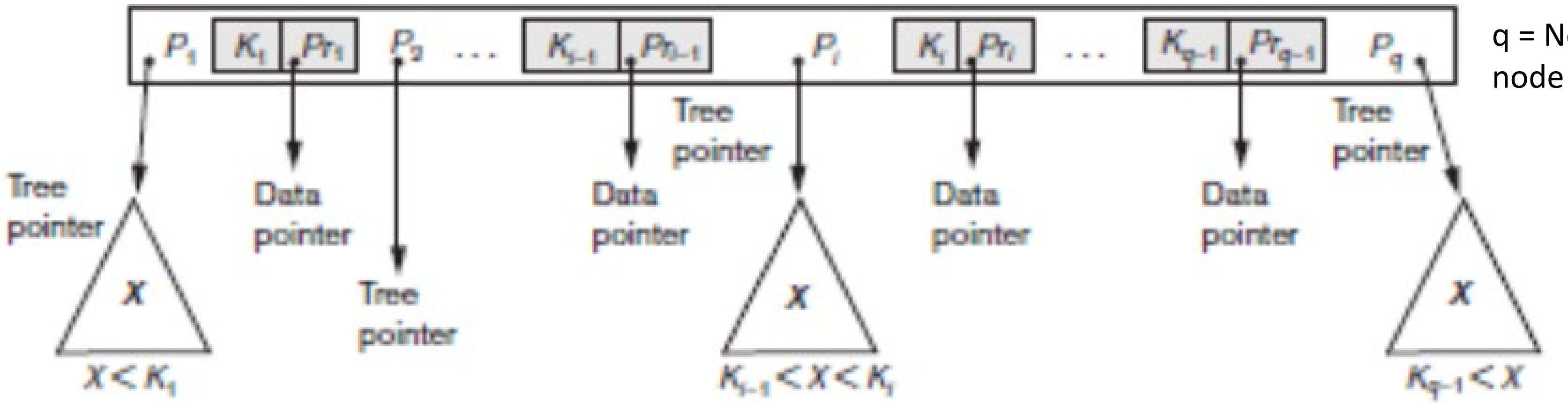


B-Tree of order 5

https://www.pdfprof.com/PDF_Image.php?idt=87914&t=39



B-Tree of order 3 (With Tree-Node Pointer and Data Pointer

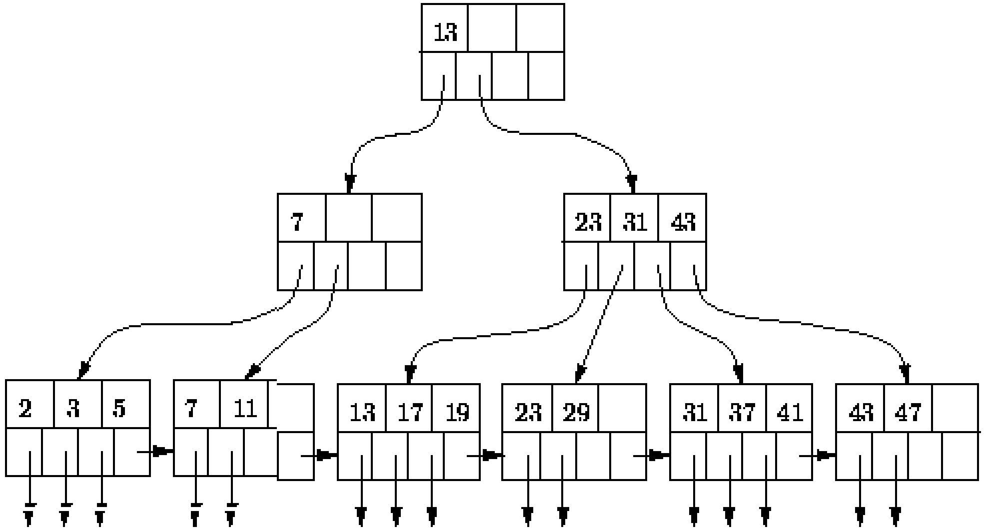


q = No of children of a non-leaf node // q-1 keys

A node in a B-Tree with q-1 search values

B+ Trees

- Data pointers <u>only at the leaf</u> node to facilitate faster search



B TREE DEMO

INDEX CREATION

The general form

CREATE [UNIQUE] INDEX <index name>

ON <table name> (<column name> [<order>] { , <column name> [<order>] })

[CLUSTER] ;

- **CREATE INDEX** DnoIndex
  ON EMPLOYEE (Dno)
  CLUSTER ; //Clustered Index

- **CREATE INDEX** EmpAgeIndex
  ON EMPLOYEE(Age DESC)
  CLUSTER; //Clustered Index

- **CREATE INDEX** EmpEmailIndex
  ON EMPLOYEE (Email) // Creates a secondary index

- **CREATE INDEX** EmpAgeService
  ON EMPLOYEE(Age,YearsOfService); //Dense Index

Choose indexes – Guidelines for choosing ‘wish-list’

1. Do not index small relations.
2. Add secondary index to a FK if it is frequently accessed.
3. Add secondary index to any attribute heavily used as a secondary key.
4. Add secondary index on attributes involved in: selection or join criteria; ORDER BY; GROUP BY; and other operations involving sorting (such as UNION or DISTINCT).
5. Add secondary index on attributes involved in built-in functions.
6. Avoid indexing an attribute or relation that is frequently updated.
7. Avoid indexing an attribute if the query will retrieve a significant proportion of the relation.
8. Avoid indexing attributes that consist of long character strings.

Choose indexes

Suppose, we have a table named buyers where the SELECT query uses indexes like below:

```
SELECT buyer_id                                   /* no need to index */
 FROM buyers
WHERE first_name='Carolyn'                        /* consider to use index */
AND last_name='Begg'                              /* consider to use index */
```

Since "buyer_id" is referenced in the SELECT portion, MySQL will not use it to limit the chosen rows. Hence, there is no great need to index it

The below is another example little different from the above one:

```
SELECT  buyers.buyer_id, country.name              /* no need to index */
FROM buyers LEFT JOIN country ON buyers.country_id=country.country_id   /* consider to use index */
WHERE first_name='Carolyn'                         /* consider to use index */
AND last_name='Begg'                               /* consider to use index */
```

According to the above queries first_name, last_name columns can be indexed as they are located in the WHERE clause. Also an additional field, country_id from country table, can be considered for indexing because it is in a JOIN clause. So indexing can be considered on every field in the WHERE clause or a JOIN clause.

Question

Select * FROM Student, Apply, University

WHERE Student.sID=Apply.sID AND Apply.uName=University.uName AND Student.GPA>1.5 AND University.uName<‘Uwindsor’

Suppose we are allowed to create two indexes, and assume all indexes are tree based. Which two indexes do you think would be most useful for speeding up query execution?

1. Student.sID, University.uName
2. Student.sID, Student.GPA
3. Apply.uName, University.uName
4. Apply.sID, Student.GPA

Downsides of Indexes

- ✓ Extra space
- ✓ Index creation
- ✓ Index maintenance

Summary and Conclusion

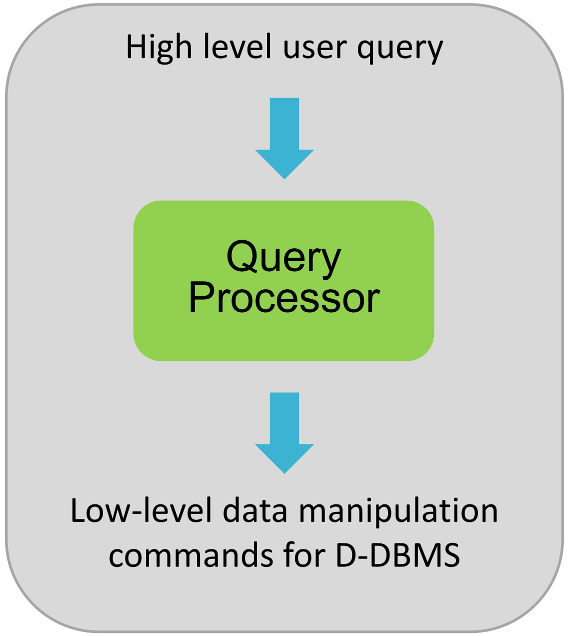
- File Organization
- Introduction to Indexing
- Single-Level Ordered Indexes
  - Primary Indexes //SPARSE
  - Clustering Indexes // SPARSE (Optionally with Block Pointer)
  - Secondary Indexes //DENSE
- Multilevel Indexes
  - Two-Level Primary Indexing //SPARSE
- Dynamic Multilevel Indexes
  - B Trees //Generalized version of Balances Search Tree- Allows more than 2 children per node
  - B+ Trees // SELF BALANCING – Data Pointers only at the leaf level
- CREATE INDEX Command in SQL
- Choosing an INDEX

Extra Material

Query Processing

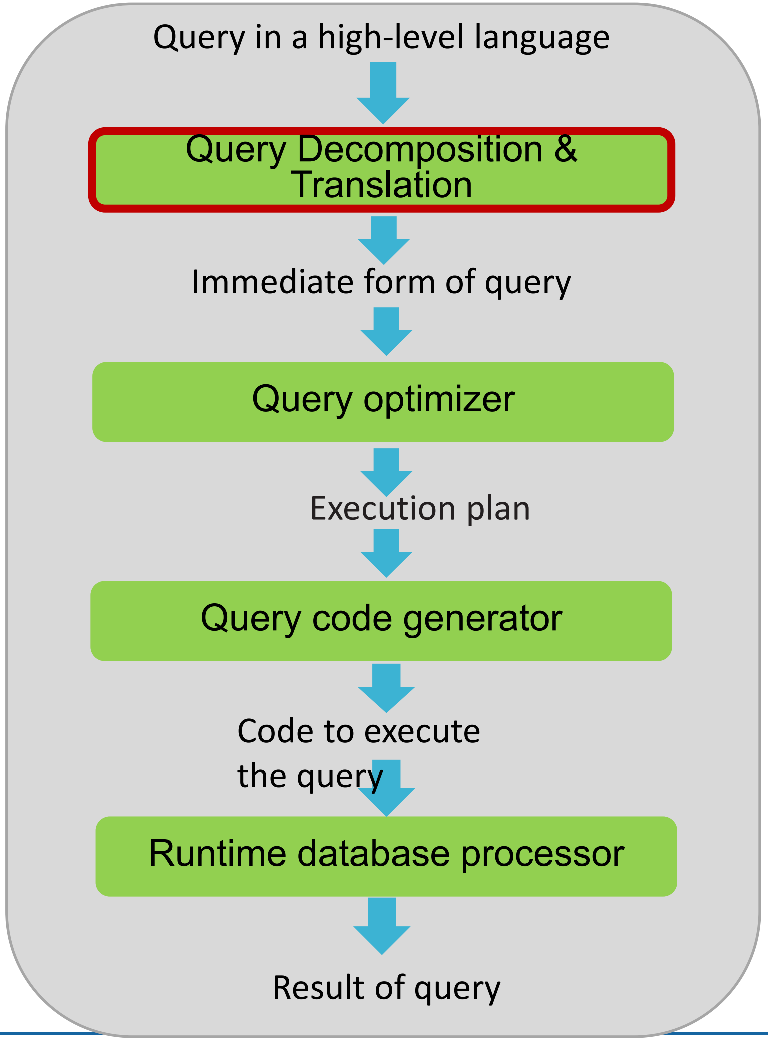
- **Aim**:

  Transform a query written in a high-level language, typically SQL, into a correct and efficient execution strategy expressed in a low-level language (implementing the relational algebra), and to execute the strategy to retrieve the required data.



An important aspect of query processing is query optimization.

Query Processing



Transform a high-level query into a relational algebra query and to check whether the query is syntactically and semantically correct.

The typical stages:

- ✓ Analysis,
- ✓ Normalization,
- ✓ Semantic analysis,
- ✓ Simplification, and
- ✓ Query restructuring.

Query Decomposition

- **Analysis:**

```
Select staffNumber
From Staff
Where position > 10;
```

Staff

StaffNo	fName	lName	Position	Sex	DOB	Salary	BranchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

This query would be rejected on two grounds:

(1) In the select list, the attribute staffNumber is not defined for the Staff relation (should be staffNo).

(2) In the WHERE clause, the comparison ">10" is incompatible with the data type position, which is a variable character string.

Query Decomposition

Normalization:

- ✓ The normalization stage of query processing converts the query into a normalized form that can be more easily manipulated.
- ✓ Predicate can be converted into one of two forms:
  - **Conjunctive normal form**: (position= 'Manager' v salary >20000) ^ branchNo = 'B003'
  - **Disjunctive normal form:** (position = 'Manager' Ù branchNo = 'B003') Ú (salary >20000 Ù branchNo = 'B003')

Semantic analysis:

- ✓ The objective of semantic analysis is to reject normalized queries that are incorrectly formulated or contradictory.
- ✓ For example:

  the predicate (position = 'Manager' Ù position = 'Assistant') on the Staff relation → **contradictory**

University of Windsor

Query Decomposition

Simplification:

- ✓ detect redundant qualifications,
- ✓ eliminate common subexpressions, and
- ✓ transform the query to a semantically equivalent but more easily and efficiently computed form.
- ✓ Access restrictions, view definitions, and integrity constraints are considered at this stage.
  - o introduce redundancy.
- ✓ Assuming user has appropriate access privileges, first apply well-known idempotency rules of Boolean algebra.

p Ù (p) ≡ p	p Ú (p) ≡ p
p Ù false ≡ false	p Ú false ≡ p
p Ù true ≡ p	p Ú true ≡ true
p Ù (~p) ≡ false	p Ú (~ p) ≡ true
p Ù (p Ú q) ≡ p	p Ú (p Ù q) ≡ p

Query Decomposition

For example, consider the following integrity constraint,:

```
CREATE ASSERTION OnlyManagerSalaryHigh
CHECK ((position <> 'Manager' AND salary< 20000)
OR (position= 'Manager' AND salary> 20000));
```

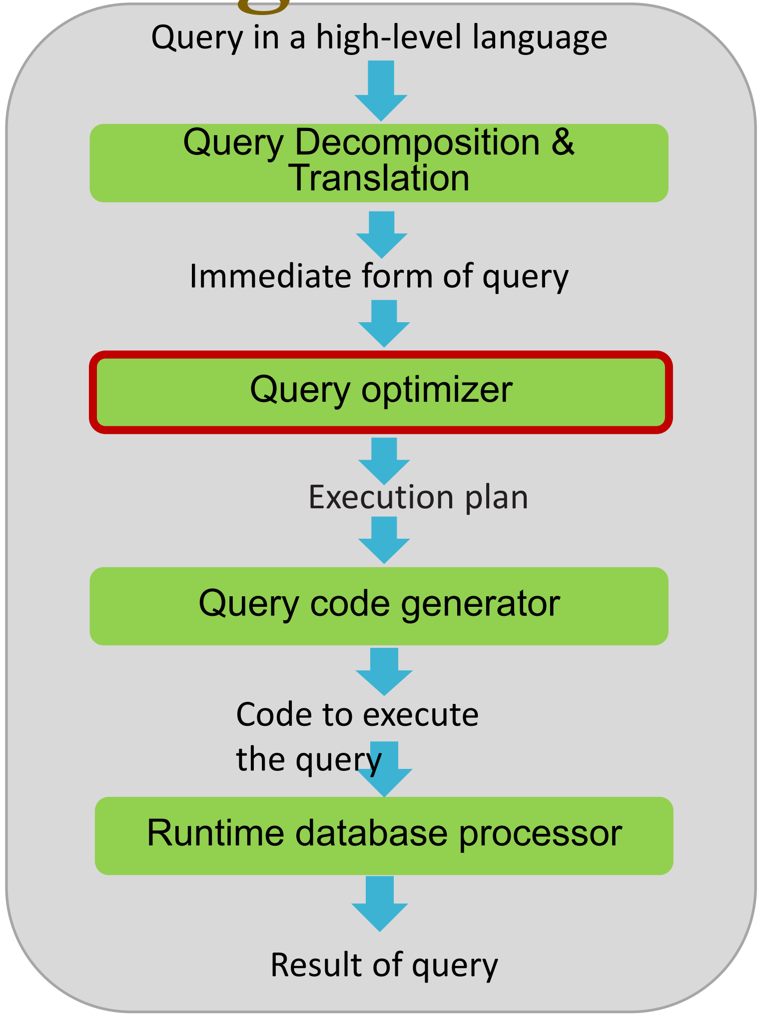
and consider the effect on the query:

```
SELECT *
FROM Staff
WHERE (position = 'Manager' AND salary = 15000);
```

A contradiction of the integrity constraint so there can be no tuples that satisfy this predicate.

Query restructuring: the query is restructured to provide a more efficient implementation

Query Processing



Aim:
As there are many equivalent transformations of the same high-level query, choose the one that minimizes resource usage.

There are two main techniques for query optimization.

- ✓ **Heuristic rules**
- ✓ Systematically estimating

Query optimization

Heuristic rules:

- ✓ Uses **transformation rules** to convert one relational algebra expression into an equivalent form that is known to be more efficient.

Transformation Rules for the Relational Algebra Operations

1. **Conjunctive Selection operations can cascade into individual Selection operations (and vice versa).**

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

Sometimes referred to as cascade of Selection.

$$\sigma_{branchNo='B003' \wedge salary>15000}(Staff) = \sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff))$$

Transformation Rules for the Relational Algebra Operations

2. Commutativity of Selection operations.

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

For example:

$$\sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff)) = \sigma_{salary>15000}(\sigma_{branchNo='B003'}(Staff))$$

Transformation Rules for the Relational Algebra Operations

3. **In a sequence of Projection operations, only the last in the sequence is required.**

$$\Pi_L \Pi_M \ \ldots \ \Pi_N(R) = \Pi_L \ (R)$$

For example:

$$\Pi_{lName}\Pi_{branchNo,\ lName}(Staff) = \Pi_{lName} \ (Staff)$$

Transformation Rules for the Relational Algebra Operations

4. Commutativity of Selection and Projection.

If predicate p involves only attributes in projection list, Selection and Projection operations commute:

$$\Pi_{Ai,\ \ldots,\ Am}(\sigma_p(R)) = \sigma_p(\Pi_{Ai,\ \ldots,\ Am}(R)) \qquad \textbf{where } p \in \{A_1, A_2, \ldots, A_m\}$$

For example:

$$\Pi_{fName,\ lName}(\sigma_{lName='Beech'}(Staff)) = \sigma_{lName='Beech'}(\Pi_{fName,lName}(Staff))$$

Transformation Rules for the Relational Algebra Operations

5. Commutativity of Theta join (and Cartesian product).

$$\mathbf{R \bowtie_p S = S \bowtie_p R}$$

$$\mathbf{R \times S = S \times R}$$

Rule also applies to Equijoin and Natural join

For example:

$$\mathbf{Staff \bowtie_{staff.branchNo=branch.branchNo} Branch = Branch \bowtie_{staff.branchNo=branch.branchNo} Staff}$$

Transformation Rules for the Relational Algebra Operations

6. Commutativity of Selection and Theta join (or Cartesian product).

If the selection predicate involves only attributes of one of the relations being joined, then the Selection and Join (or Cartesian product) operations commute:

$$\sigma_p(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r S$$

$$\sigma_p(R \times S) = (\sigma_p(R)) \times S \qquad \text{where } p \in \{A_1, A_2, \ldots, A_n\}$$

If selection predicate is conjunctive predicate having form ($p \wedge q$), where p only involves attributes of R, and q only attributes of S, Selection and Theta join operations commute as:

$$\sigma_{p \wedge q}(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r (\sigma_q(S))$$

$$\sigma_{p \wedge q}(R \times S) = (\sigma_p(R)) \times (\sigma_q(S))$$

For example:

$$\sigma_{position='Manager' \wedge city='London'}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch) =$$
$$(\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'}(Branch))$$

Branch

BranchNo	Street	City	Postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX

Staff

StaffNo	fName	lName	Position	Sex	DOB	Salary	BranchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005

Transformation Rules for the Relational Algebra Operations

7. **Commutativity of Projection and Theta join (or Cartesian product).**

If projection list is of form $L = L_1 \cup L_2$, where L_1 only has attributes of R, and L_2 only has attributes of S, provided join condition only contains attributes of L, Projection and Theta join commute:

$$\Pi_{L1 \cup L2}(R \bowtie_r S) = (\Pi_{L1}(R)) \bowtie_r (\Pi_{L2}(S))$$

If join condition contains additional attributes not in L ($M = M_1 \cup M_2$ where M_1 only has attributes of R, and M_2 only has attributes of S), a final projection operation is required:

$$\Pi_{L1 \cup L2}(R \bowtie_r S) = \Pi_{L1 \cup L2}((\Pi_{L1 \cup M1}(R)) \bowtie_r (\Pi_{L2 \cup M2}(S)))$$

For example:

$$\Pi_{position,city,branchNo}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch) = (\Pi_{position,\ branchNo}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\Pi_{city,\ branchNo} (Branch))$$

and using the latter rule:

$$\Pi_{position,\ city}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch) = \Pi_{position,\ city} ((\Pi_{position,\ branchNo}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\Pi_{city,\ branchNo} (Branch)))$$

Transformation Rules for the Relational Algebra Operations

8. **Commutativity of Union and Intersection (but not set difference).**

$$R \cup S = S \cup R$$

$$R \cap S = S \cap R$$

9. **Commutativity of Selection and set operations (Union, Intersection, and Set difference).**

$$\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$$

$$\sigma_p(R \cap S) = \sigma_p(S) \cap \sigma_p(R)$$

$$\sigma_p(R - S) = \sigma_p(S) - \sigma_p(R)$$

10. **Commutativity of Projection and Union.**

$$\Pi_L(R \cup S) = \Pi_L(S) \cup \Pi_L(R)$$

Transformation Rules for the Relational Algebra Operations

11. Associativity of Theta join (and Cartesian product).

Cartesian product and Natural join are always associative:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$(R \times S) \times T = R \times (S \times T)$$

If join condition q involves attributes only from S and T, then Theta join is associative:

$$(R \bowtie_p S) \bowtie_{q \wedge r} T = R \bowtie_{p \wedge r} (S \bowtie_q T)$$

For example:

$$(Staff \bowtie_{Staff.staffNo=PropertyForRent.staffNo} PropertyForRent) \bowtie_{ownerNo=Owner.ownerNo \wedge staff.lName=Owner.lName} Owner = Staff \bowtie_{staff.staffNo = PropertyForRent.staffNo \wedge staff.lName=lName} (PropertyForRent \bowtie_{ownerNo} Owner)$$

Transformation Rules for the Relational Algebra Operations

12. Associativity of Union and Intersection (but not Set difference).

$$(R \cup S) \cup T = S \cup (R \cup T)$$

$$(R \cap S) \cap T = S \cap (R \cap T)$$

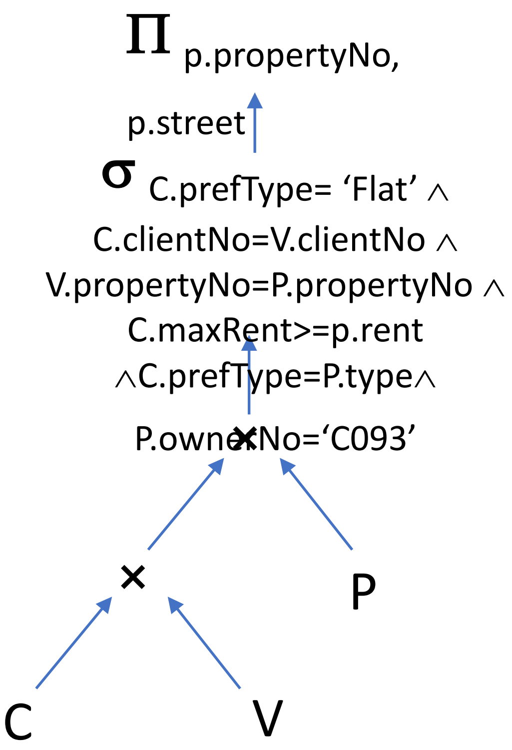
Example : Use of Transformation Rules

For prospective renters of flats, find properties that match requirements and owned by CO93.

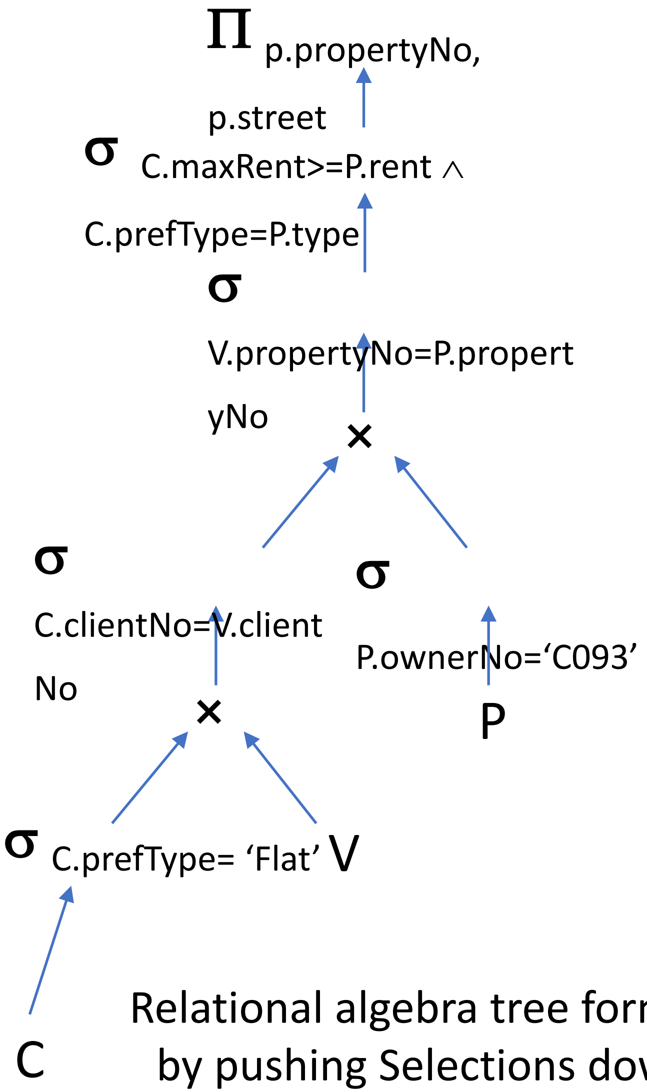
```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE   c.prefType = 'Flat' AND
        c.clientNo = v.clientNo AND
        v.propertyNo = p.propertyNo AND
        c.maxRent >= p.rent AND
        c.prefType = p.type AND
        p.ownerNo = 'CO93';
```

$$\Pi_{p.propertyNo,\ p.street}(\sigma_{c.prefType='Flat' \land c.clientNo=v.clientNo \land v.propertyNo=p.propertyNo \land c.maxRent>=p.rent \land c.prefType=p.type \land p.ownerNo='C093'}((c \times v) \times p))$$

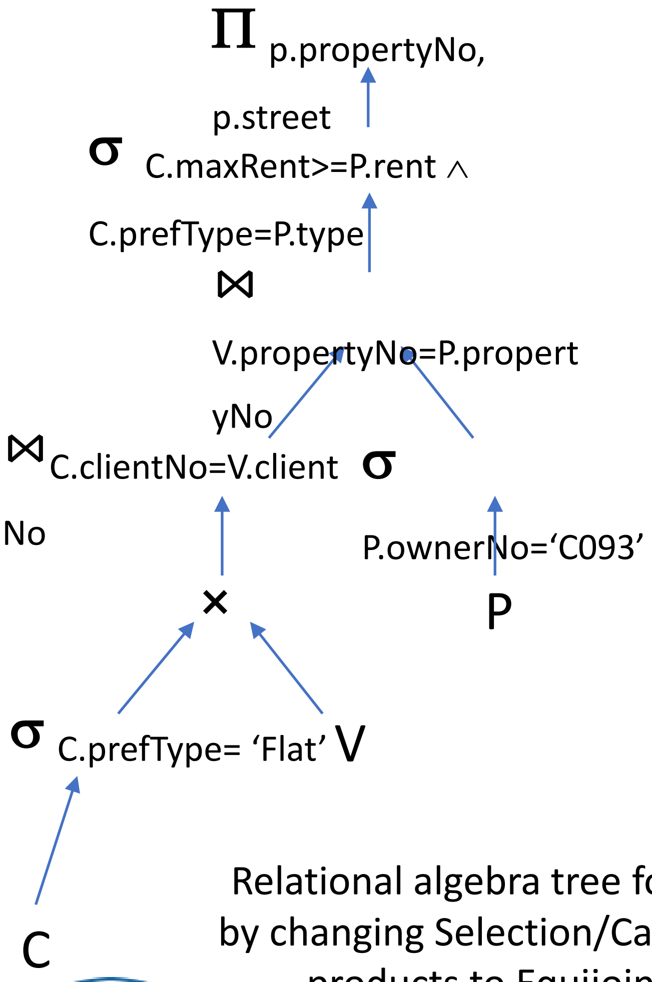
$$\Pi_{p.propertyNo,\ p.street}(\sigma_{c.prefType=\text{'Flat'} \wedge c.clientNo=v.clientNo \wedge v.propertyNo=p.propertyNo \wedge c.maxRent>=p.rent \wedge c.prefType=p.type \wedge p.ownerNo=\text{'C093'}}((c \times v) \times p))$$



Canonical relational algebra tree

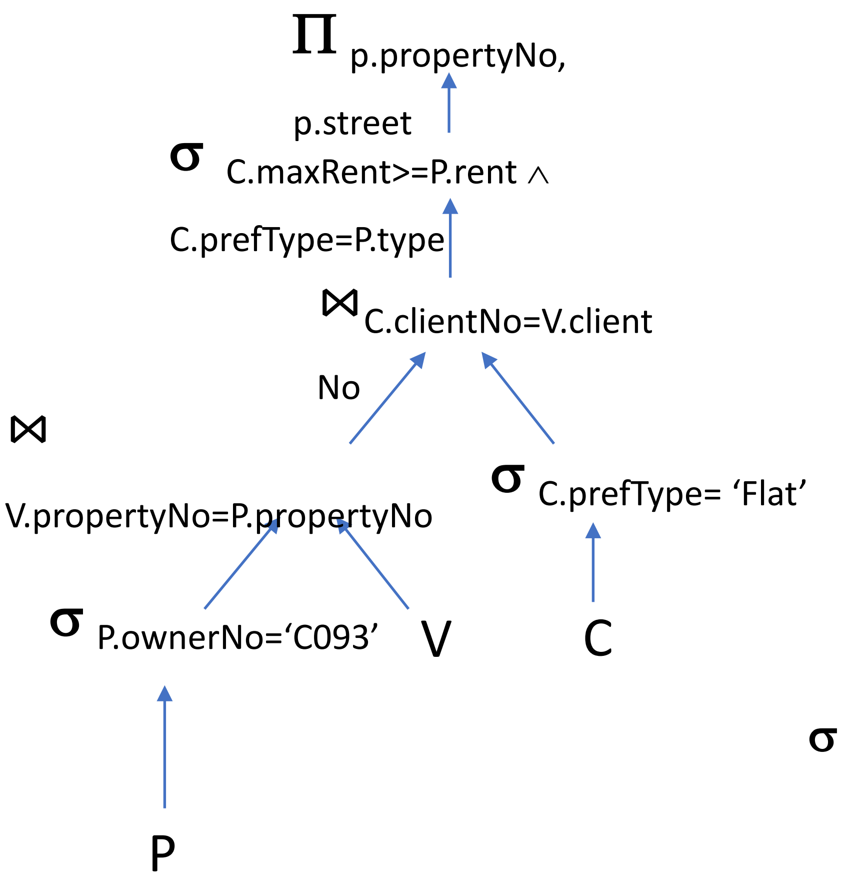


Relational algebra tree formed by pushing Selections down

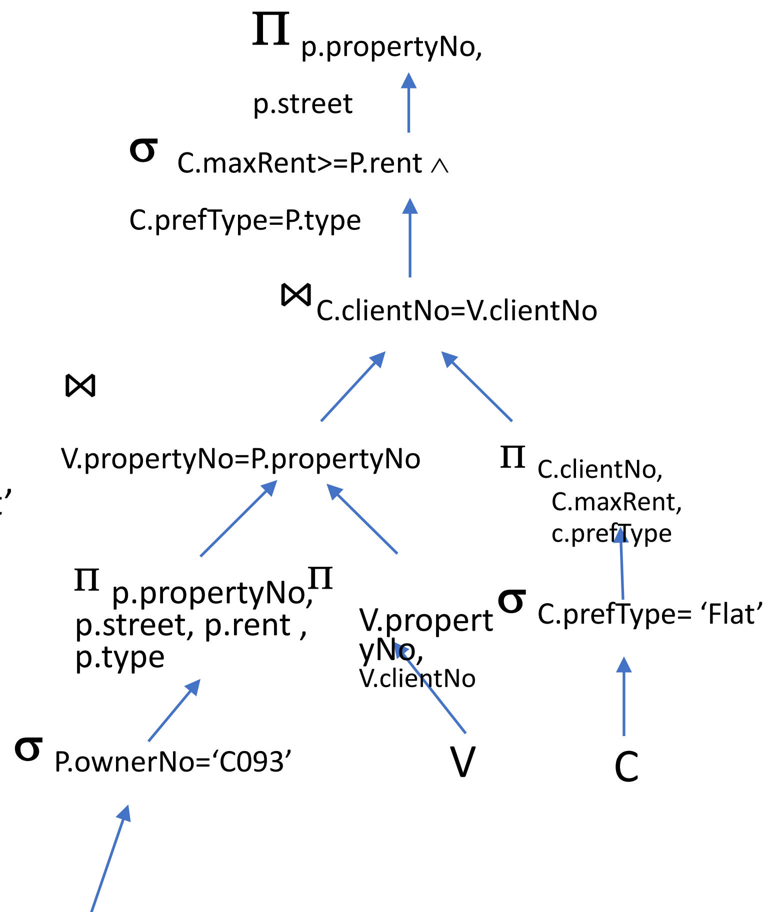


Relational algebra tree formed by changing Selection/Cartesian products to Equijoins;

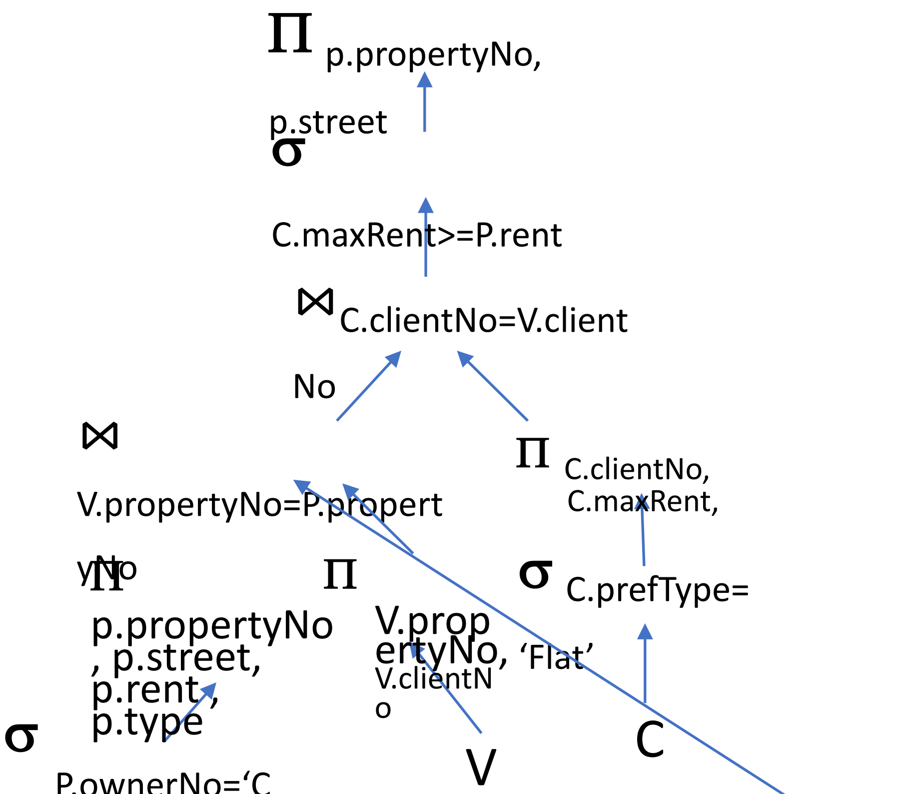
$$\Pi_{p.propertyNo,\ p.street}(\sigma_{c.prefType=\text{'Flat'} \wedge c.clientNo=v.clientNo \wedge v.propertyNo=p.propertyNo \wedge c.maxRent>=p.rent \wedge c.prefType=p.type \wedge p.ownerNo=\text{'C093'}}((c \times v) \times p))$$



Relational algebra tree formed using associativity of Equijoins;



Relational algebra tree formed by pushing Projections down;



Final reduced relational algebra tree formed by substituting c.prefType = 'Flat' in Selection on p.type and pushing resulting Selection down tree.

Heuristical Processing Strategies

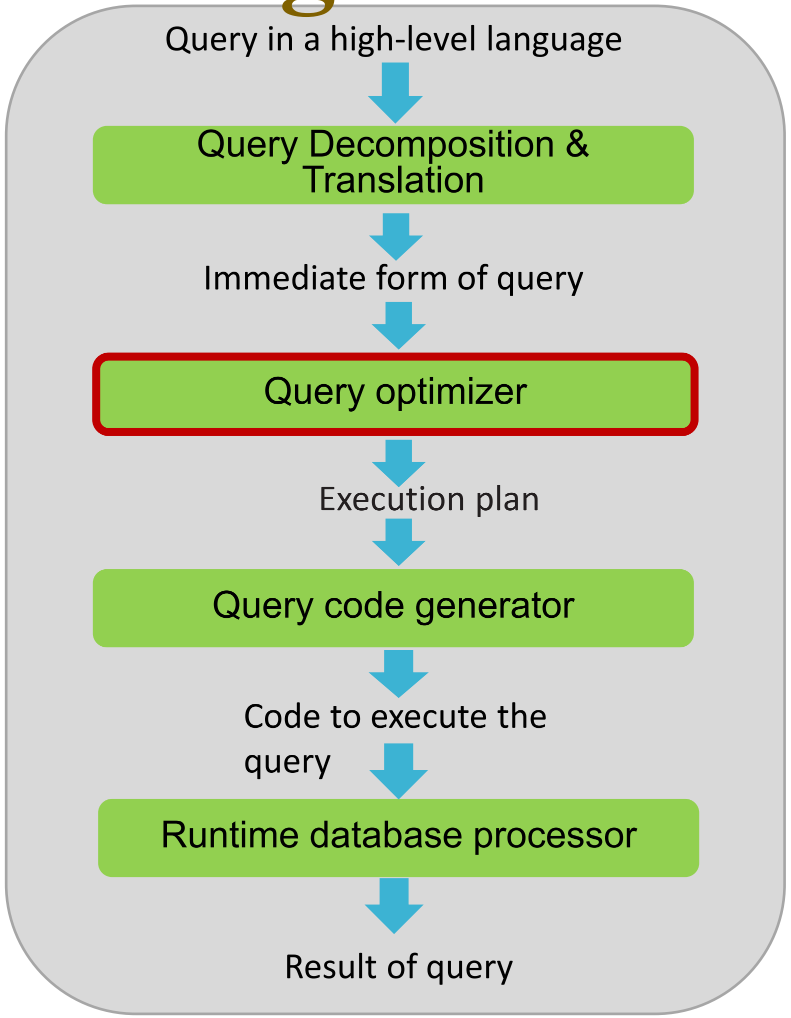
1. Perform Selection operations as early as possible.
   - ✓ Keep predicates on same relation together.
2. Combine Cartesian product with subsequent Selection whose predicate represents join condition into a Join operation.

$$\sigma_{R.a\ \theta\ S.b}(RXS) = R \bowtie_{R.a\ \theta\ S.b} (S)$$
$$(R \bowtie_{R.a\ \theta\ S.b} S) \bowtie_{S.c\ \theta\ T.d} T$$

3. Use associativity of binary operations to rearrange leaf nodes so leaf nodes with most restrictive Selection operations executed first.

4. Perform Projection as early as possible.
   - ✓ Keep projection attributes on same relation together.
5. Compute common expressions once.
   - ✓ If common expression appears more than once, and result not too large, store result and reuse it when required.
   - ✓ Useful when querying views, as same expression is used to construct view each time.

Query Processing



Aim:
As there are many equivalent transformations of the same high-level query, choose the one that minimizes resource usage.

There are two main techniques for query optimization.

- ✓ Heuristic rules
- ✓ **Systematically estimating**

Cost Estimation for the Relational Algebra Operations

- Many different ways of implementing Relational Algebra (RA) operations.
- Aim of Query Optimization(QO) is to choose most efficient one.
  - Use formulae that estimate costs for a number of options and select one with lowest cost.
- Consider only cost of disk access, which is usually dominant cost in QP.
- Many estimates are based on cardinality of the relation, so need to be able to estimate this.

Database Statistics

The success of estimating the size and cost of intermediate relational algebra operations depends on the amount and currency of the statistical information that the DBMS holds.

For each base relation R:

- ✓ nTuples(R)- the number of tuples (records) in relation R (that is, its cardinality).
- ✓ bFactor(R)- the blocking factor of R (that is, the number of tuples of R that fit into one block).
- ✓ nBlocks(R)- the number of blocks required to store R.

For each attribute A of base relation R:

- ✓ $nDistinct_A(R)$- the number of distinct values that appear for attribute A in relation R.
- ✓ $min_A(R)$, $max_A(R)$- the minimum and maximum possible values for the attribute A in relation R.
- ✓ $SC_A(R)$—the selection cardinality of attribute A in relation R.

For each multilevel index I on attribute set A:

- ✓ $nLevels_A(I)$—the number of levels in I.
- ✓ $nLfBlocks_A(I)$—the number of leaf blocks in I.

Selection Operation ($S = \sigma_p(R)$)

Predicate may be simple or composite.
Number of different implementations, depending on file structure, and whether attribute(s) involved are indexed/hashed.
Main strategies are:

1. Linear Search (Unordered file, no index): [nBlocks(R)/2], for equality condition on key attribute nBlocks(R), otherwise
2. Binary Search (Ordered file, no index): [$\log_2$ (nBlocks(R))], for equality condition on ordered attribute [$\log_2$ (nBlocks(R))] + [SC_A (R)/bFactor(R)] − 1, otherwise
3. Equality on hash key: 1, assuming no overflow
4. Equality condition on primary key: $nLevels_A$ (I) + 1
5. Inequality condition on primary key: $nLevels_A$ (I) + [nBlocks(R)/2]
6. Equality condition on clustering (secondary) index: $nLevels_A$ (I) + [SC_A (R)/bFactor(R)]
7. Equality condition on a non-clustering (secondary) index: $nLevels_A$ (I) + [SC_A (R)]
8. Inequality condition on a secondary B+-tree index: $nLevels_A$ (I) + [$nLfBlocks_A$ (I)/2 + nTuples(R)/2] $nLevels_A$ (I) + 1

Selection Operation ($S = \sigma_p(R)$)

Cost estimation for Selection operation:
We make the following assumptions about the Staff relation:

- There is a hash index with no overflow on the primary key attribute staffNo.
- There is a clustering index on the foreign key attribute branchNo.
- There is a B+-tree index on the salary attribute.
- The Staff relation has the following statistics stored in the system catalog:
  - nTuples(Staff) = 3000
  - bFactor(Staff) = 30 → nBlocks(Staff) = 100
  - $nDistinct_{branchNo}$(Staff) = 500 → $SC_{branchNo}$(Staff) = 6
  - $nDistinct_{position}$(Staff) = 10 → $SC_{position}$(Staff) = 300
  - $nDistinct_{salary}$(Staff) = 500 → SC_{salary}(Staff) = 6
  - min_{salary}(Staff) = 10,000 max_{salary}(Staff) = 50,000
  - $nLevels_{branchNo}$(I) = 2
  - $nLevels_{salary}$(I) = 2 $nLfBlocks_{salary}$(I) = 50

The estimated cost of a linear search on the key attribute staffNo is 50 blocks,
The cost of a linear search on a non-key attribute is 100 blocks.

Consider the following Selection operations:

S1: $\sigma_{staffNo='SG5'}$(Staff)

Equality condition on the primary key. the attribute staffNo is hashed, estimate the cost as 1 block. The estimated cardinality of the result relation is $SC_{staffNo}$(Staff) = 1.

S2: $\sigma_{position='manager'}$(Staff)

The attribute in the predicate is a non-key, non-indexed attribute, so we cannot improve on the linear search method, giving an estimated cost of 100 blocks. The estimated cardinality of the result relation is $SC_{position}$(Staff) = 300

Join Operation ($T = (R \bowtie_F S)$)

The most time-consuming operation to process.
The main strategies for implementing the Join operation.

- ✓ Block Nested Loop Join:

  nBlocks(R) 1 (nBlocks(R) * nBlocks(S)), if buffer has only one block for R and S
  nBlocks(R) 1 [nBlocks(S)*(nBlocks(R)/(nBuffer 2 2))], if (nBuffer 2 2) blocks for R
  nBlocks(R) 1 nBlocks(S), if all blocks of R can be read into database buffer

- ✓ Indexed Nested Loop Join:

  Depends on indexing method; for example:
  nBlocks(R) 1 nTuples(R)*(nLevelsA (I) 1 1), if join attribute A in S is the primary key
  Blocks(R) 1 nTuples(R)*(nLevelsA (I) 1 [SCA (R)/bFactor(R)]), for clustering index I on attribute A

- ✓ Sort-Merge Join:

  nBlocks(R)*[log2 (nBlocks(R)] 1 nBlocks(S)*[log2 (nBlocks(S)], for sorts
  nBlocks(R) 1 nBlocks(S), for merge

- ✓ Hash Join:

  3(nBlocks(R) 1 nBlocks(S)), if hash index is held in memory
  2(nBlocks(R) 1 nBlocks(S))*[lognBuffer–1 (nBlocks(S)) 2 1] 1 nBlocks(R) 1 nBlocks(S), otherwise

Projection Operation($S = \Pi_{A1, A2, \ldots, Am}(R)$)

To implement projection, need to:

- ✓ Remove attributes that are not required;
- ✓ Eliminate any duplicate tuples produced from previous step.

Estimating the cardinality of the Projection operation:

When the Projection contains a key attribute: the cardinality of the Projection is: nTuples(S) = nTuples(R)

If the Projection consists of a single non-key attribute ($S = \Pi_A(R)$), we can estimate the cardinality of the Projection as: nTuples(S) = $SC_A(R)$

Two main approaches to eliminating duplicates:

- ✓ Sorting;
- ✓ Hashing.

The Relational Algebra Set Operations($T = R \cup S$, $T = R \cap S$, $T = R - S$)

Implemented by

- ✓ sorting both relations on same attributes, and
- ✓ then scanning through each of sorted relations once to obtain desired result.

For all these operations, we could develop an algorithm using the sort–merge join algorithm as a basis.

The estimated cost in all cases is simply: nBlocks(R) + nBlocks(S) + nBlocks(R)*[$\log_2$ (nBlocks(R))]
+ nBlocks(S)*[$\log_2$ (nBlocks(S))]

Any Questions