



Institución Universitaria

Acreditada en Alta Calidad

ANDRES
MARTINEZ
GUTIERREZ

2 0 1 8 - 2

DATABASE ADMINISTRATION ADVANCED

AGENDA


④ Indexes

④ Explain Plan

Indexes

<https://use-the-index-luke.com/>

<http://bit.ly/2CZjZei>



```
1 SELECT * FROM COMUNAS
2
3 1    POPULAR
4 10   LA CANDELARIA
5 11   LAURELES ESTADIO
6 12   LA AMERICA
7 13   SAN JAVIER
8 14   POBLADO
9 15   GUAYABAL
10 16   BELÉN
11 2    SANTA CRUZ
12 3    MANRIQUE
13 4    ARANJUEZ
14 5    CASTILLA
15 50   PALMITAS
16 6    DOCE DE OCTUBRE
17 60   SAN CRISTOBAL
18 7    ROBLED0
19 70   ALTAVISTA
20 8    VILLA HERMOSA
21 80   SAN ANTONIO DE PRADO
22 9    BUENOS AIRES
23 90   SANTA ELENA
```



```
1 EXPLAIN PLAN FOR select * from comunas where nombre = 'CASTILLA'
```

```
2
```

```
3 SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

```
4
```

```
5 Plan hash value: 2814270609
```

```
6
```

```
7 -----
```

```
8 | Id      | Operation                      | Name      | Rows   | Bytes | Cost (%CPU)| Time       |
```

```
9 -----
```

```
10 |    0     | SELECT STATEMENT                |           |     1   |    14 |      3   (0)| 00:00:01   |
```

```
11 |*    1     | TABLE ACCESS FULL              | COMUNAS   |     1   |    14 |      3   (0)| 00:00:01   |
```

```
12 -----
```

```
13
```

```
14 Predicate Information (identified by operation id):
```

```
15 -----
```

```
16
```

```
17      1 - filter("NOMBRE"='CASTILLA')
```

How do databases index work?

- First of all... What is an index?
- The whole point of having an index is to speed up search queries by essentially **cutting down** the number of records/rows in a table that need to be examined.

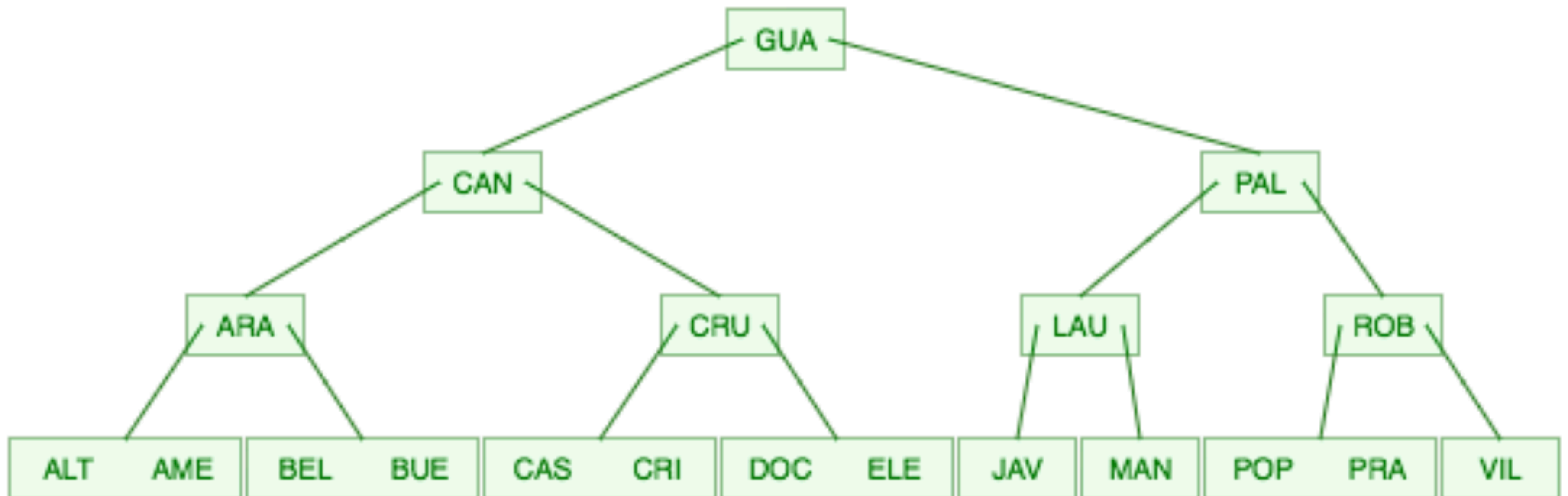
How do databases index work?

- The downside is that indexes make it slower to add rows or make updates to existing rows for that table
- Position of the blocks in the disk is random

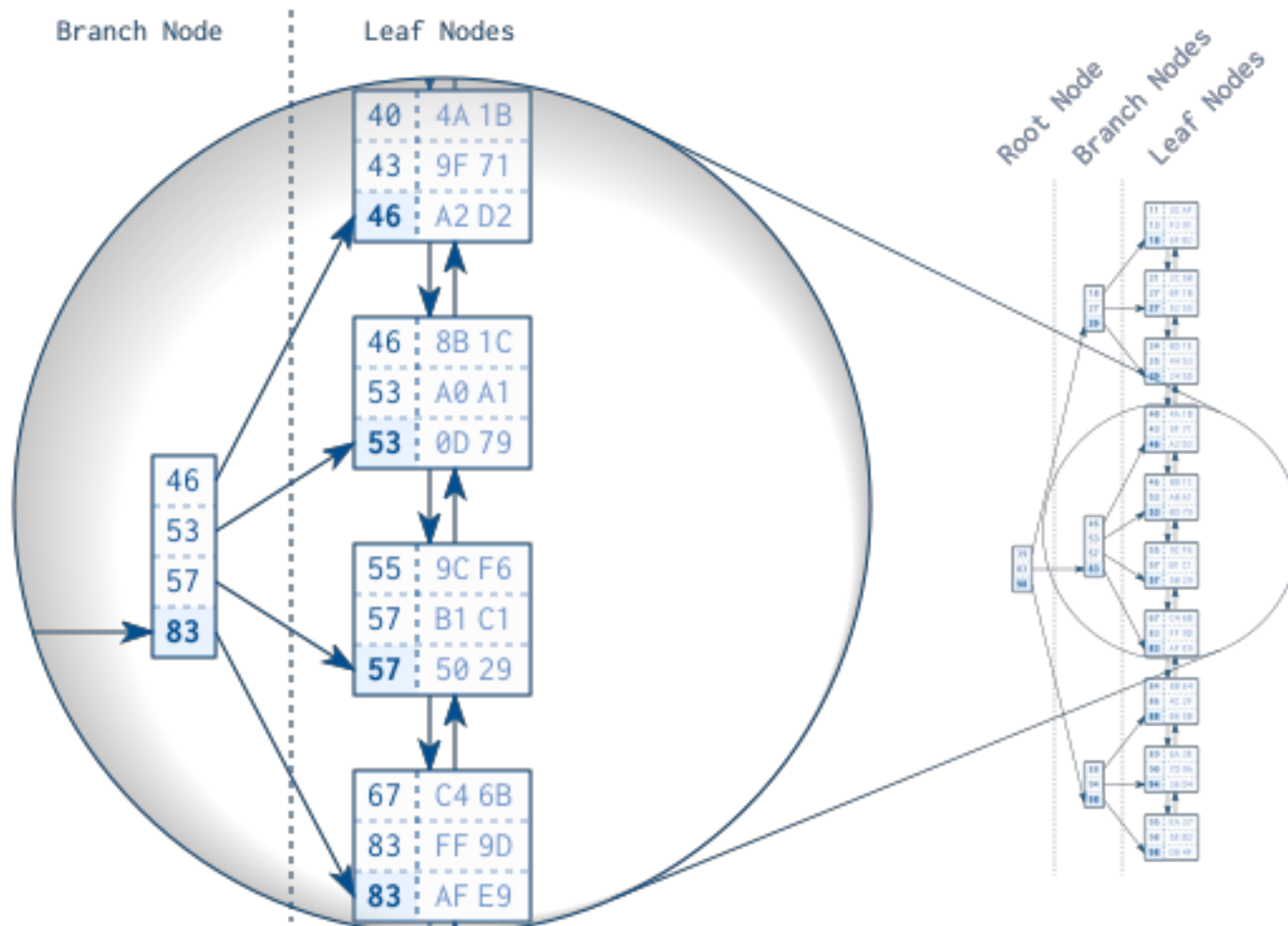
How do databases index work?

- Similar to a telephone directory with pages shuffled
- How to find a record between those pages?
- **“Balanced search tree / B-tree”**

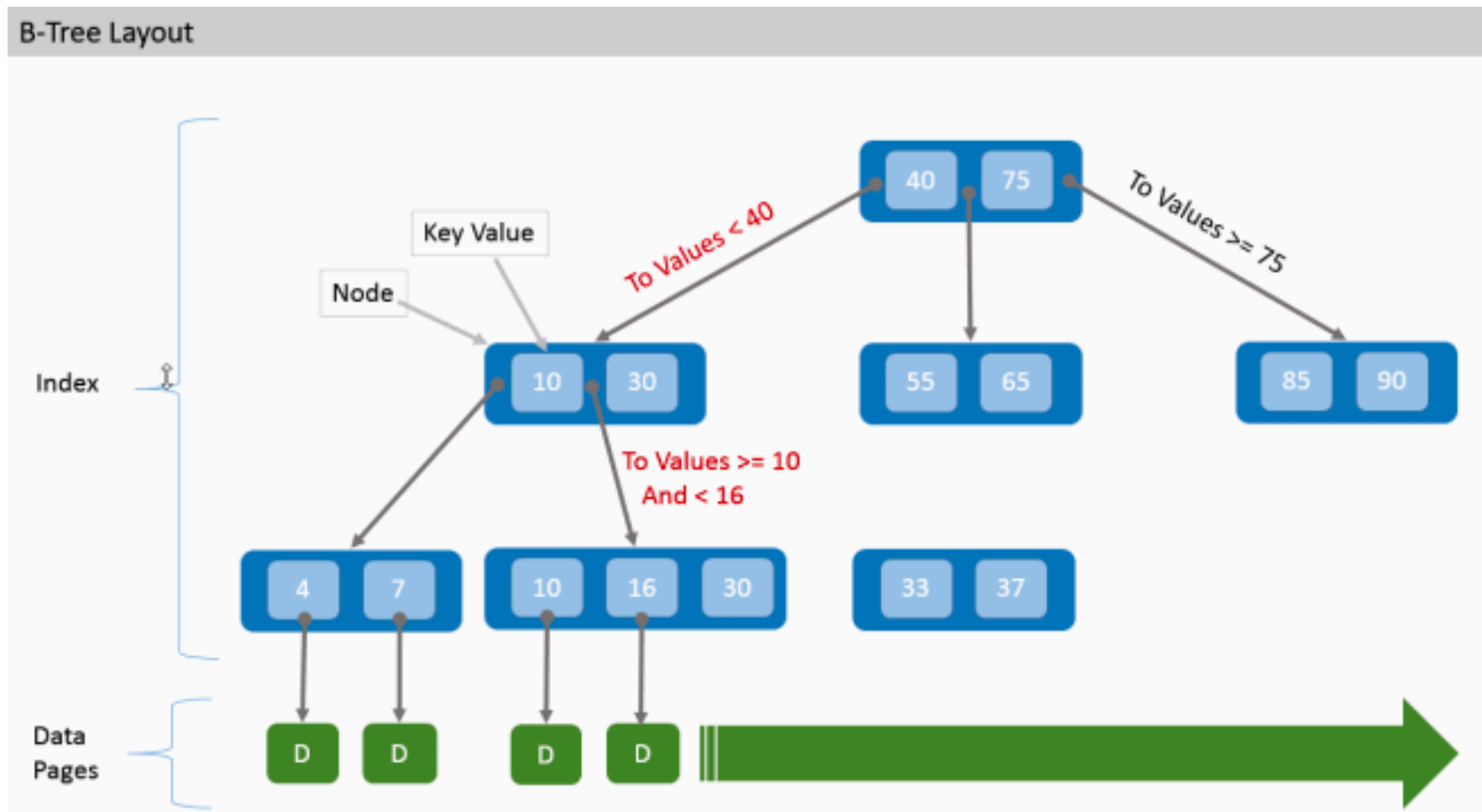
Balanced search tree (B-tree)



Balanced search tree (B-tree)



Balanced search tree (B-tree)



Balanced search tree (B-tree)

- Each branch node entry corresponds to the biggest value in the respective leaf node
- A branch layer is built up until all the leaf nodes are covered by a branch node.

Balanced search tree (B-tree)

- The next layer is built similarly, but on top of the first branch node level. (Until root node)
- The tree depth is equal at every position
- Distance between root node and leaf nodes is the same everywhere

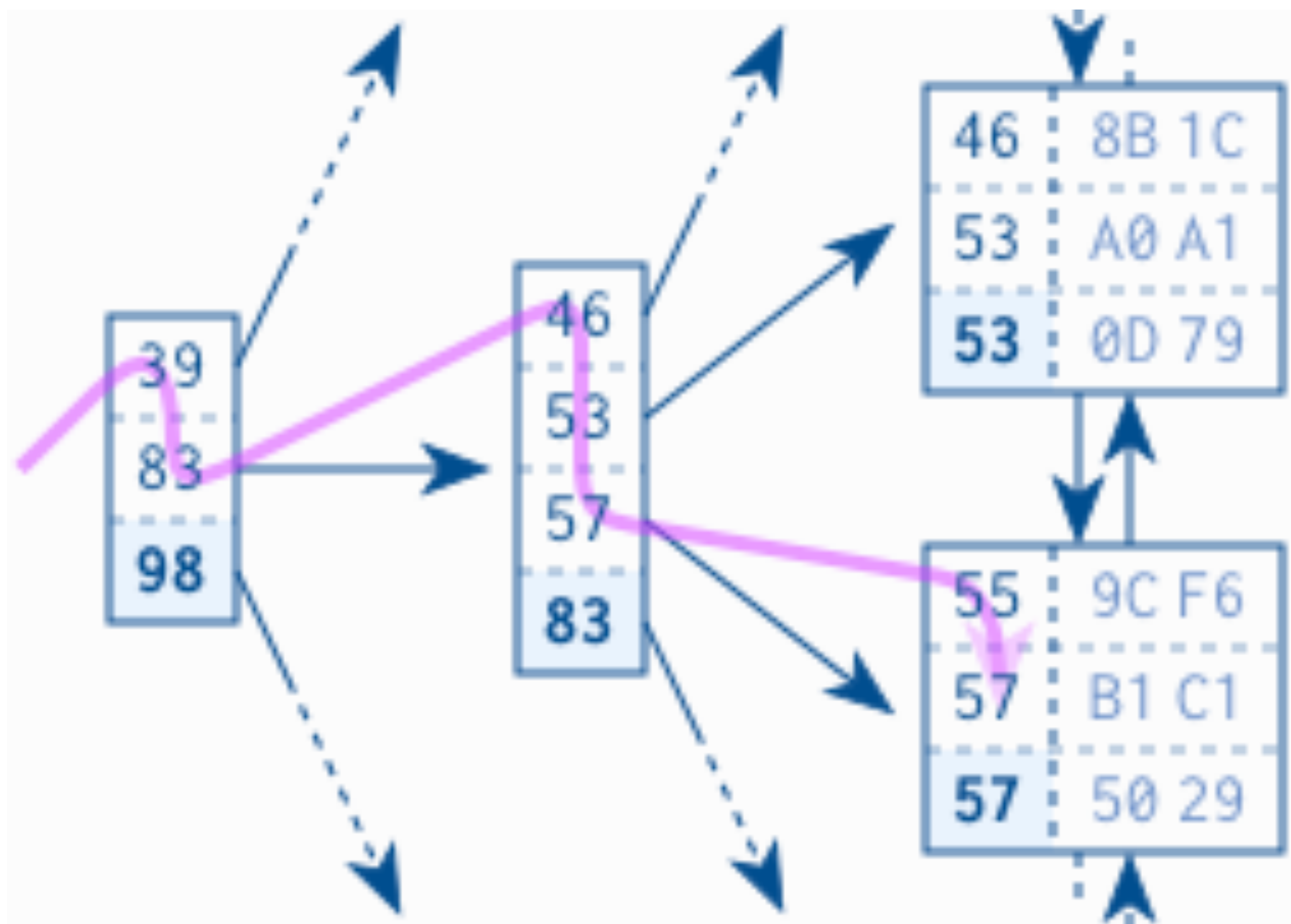
Balanced search tree (B-tree)

- Deletions and insertions in logarithmic time
- Can be sorted and is time efficient
- RDBMS choose which index but you can specify which one to use

Balanced search tree (B-tree)

- Once created the database maintains the index automatically (**insert, update, delete**)
- First power of indexing
- Real world indexes with millions of records have a tree depth of four or five

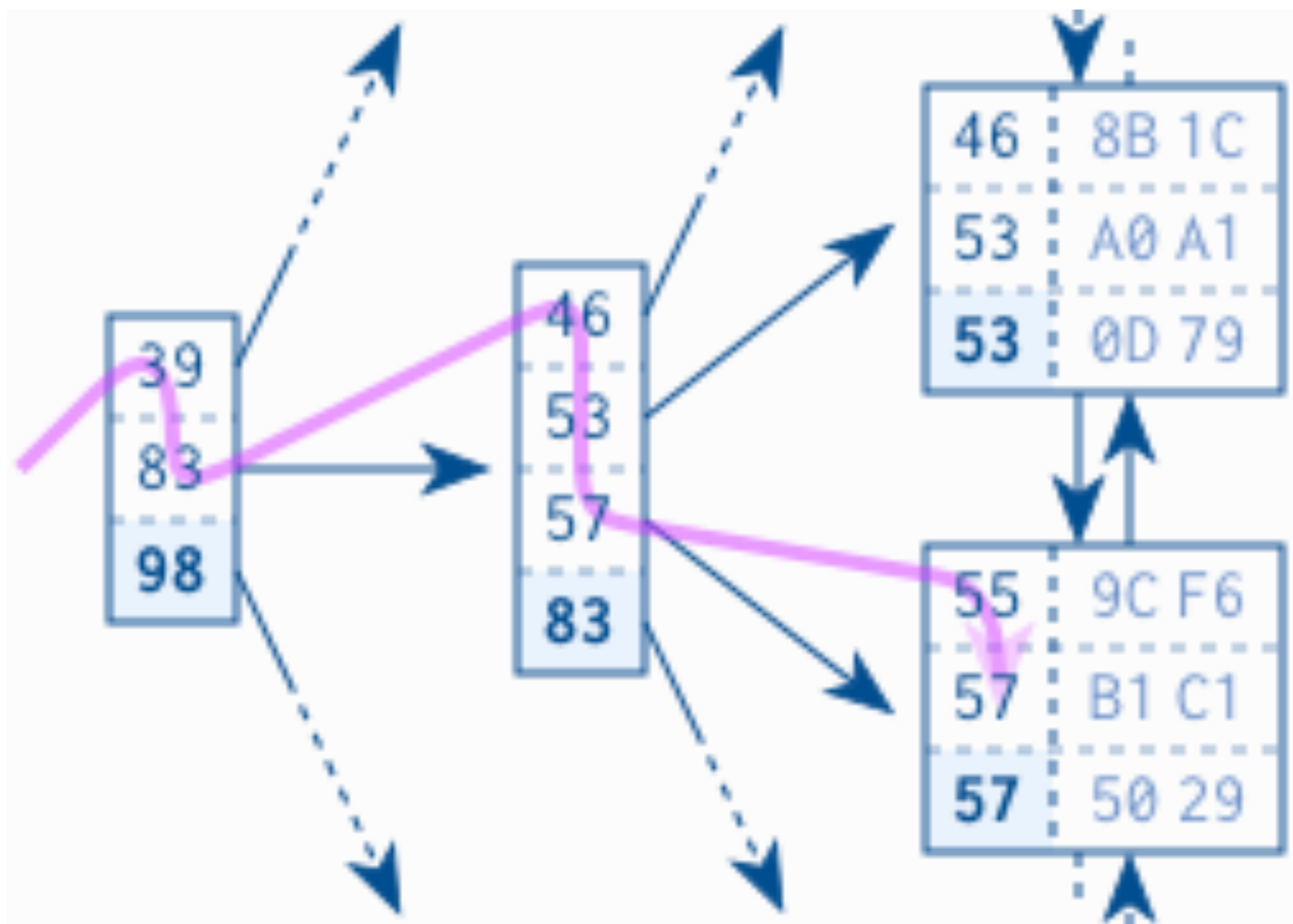
Balanced search tree (B-tree)



Slow Indexes: 1

- Rebuilding an index does not improve performance on the long run.
- The first ingredient for a slow index lookup is the leaf node chain (See id 57)
 - Index lookup not only needs to perform the tree traversal, it also needs to follow the leaf node chain.

Lookup leaf chain

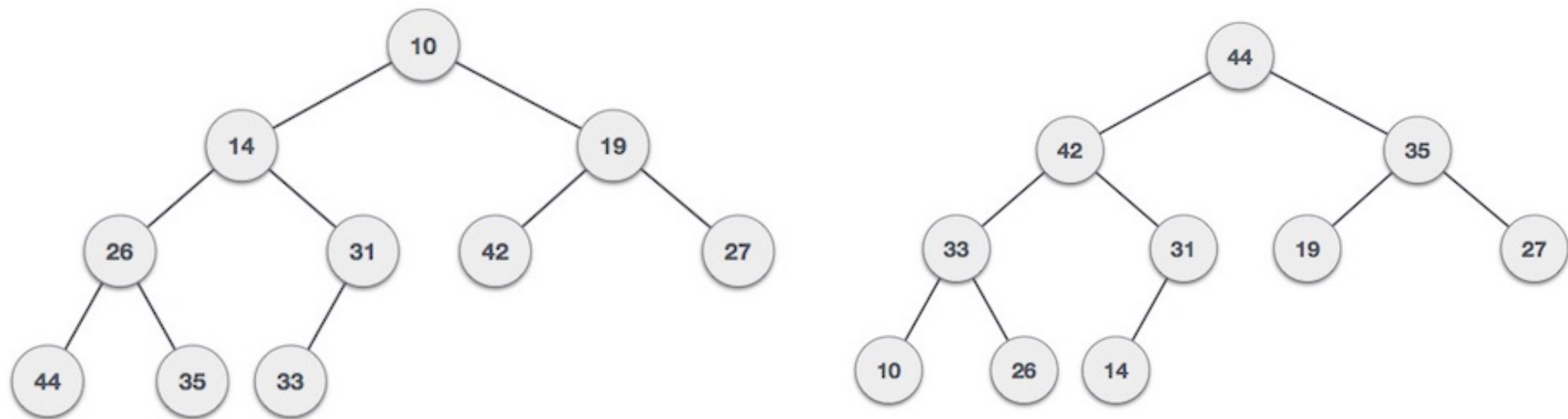


Slow Indexes: 2

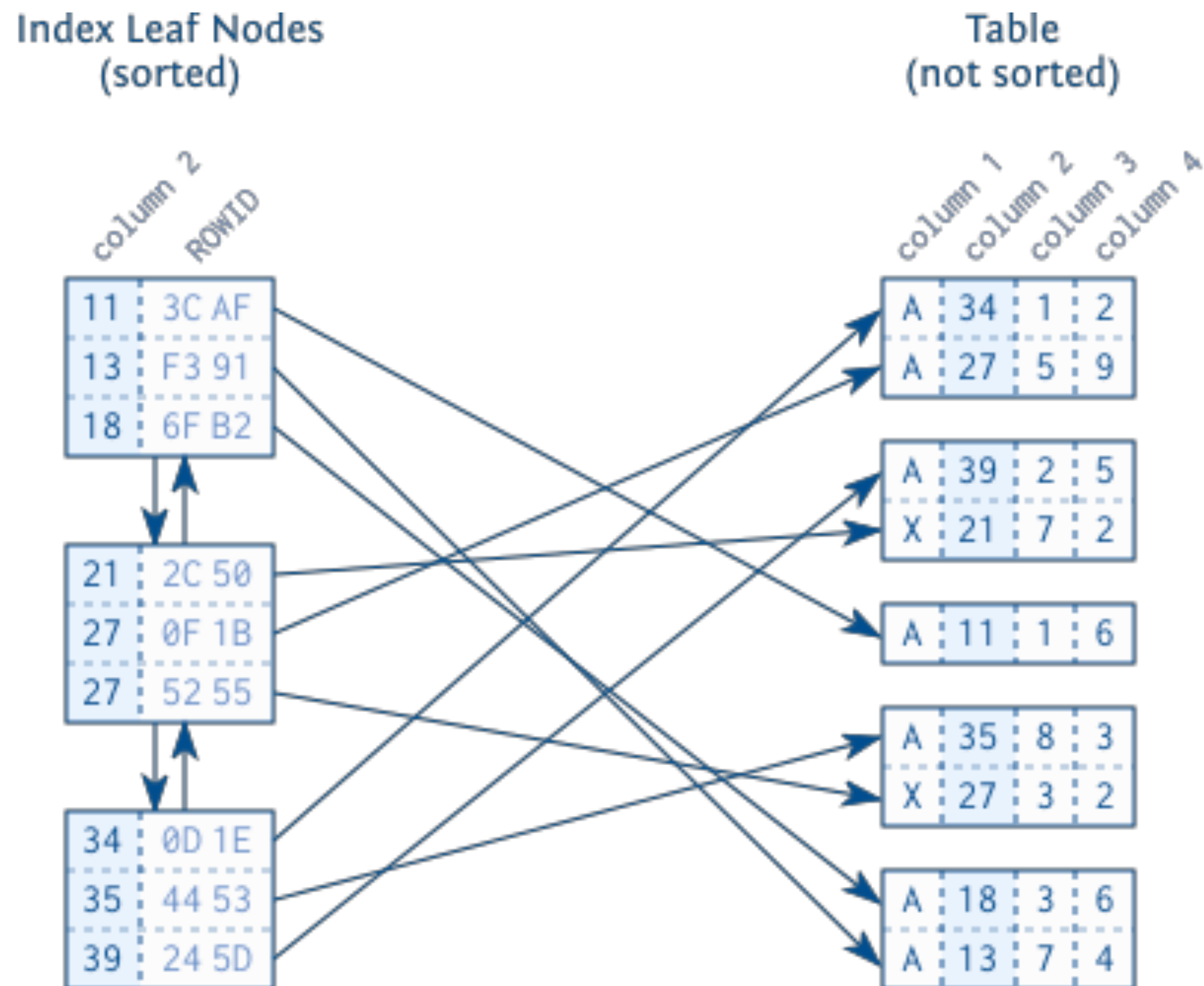
- The second ingredient for a slow index lookup is accessing the table.
- Each index entry consists of the indexed columns (key and table row)
- Table data is stored in a **heap** structure and is not sorted at all

Slow Indexes (Note)

- **Heap** is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly



Index Leaf Nodes and Corresponding Table Data



Index lookup

- An index lookup requires three steps: (1) the tree traversal; (2) following the leaf node chain; (3) fetching the table data.
- The tree traversal is the only step that has an upper bound for the number of accessed blocks—the index depth.
- The other two steps might need to access many blocks—they cause a slow index lookup.

Oracle's Operations

- **1. Index and table access**
- **2. Joins**
- **3. Sorting and grouping**

Oracle's Operations

1. Index table and access

- **INDEX UNIQUE SCAN**
- **INDEX RANGE SCAN**
- **INDEX FULL SCAN**
- **INDEX FAST FULL SCAN**
- **TABLE ACCESS BY INDEX ROWID**
- **TABLE ACCESS FULL**

Oracle's Operations

1. Index table and access

- **INDEX UNIQUE SCAN:** It performs the B-tree traversal only. The database uses this operation if a unique constraint ensures that the search criteria will match no more than one entry

Oracle's Operations

1. Index table and access

- **INDEX RANGE SCAN:** The INDEX RANGE SCAN performs the B-tree traversal and follows the leaf node chain to find all matching entries.

Oracle's Operations

1. Index table and access

- **INDEX FULL SCAN:** Reads the entire index—all rows—in index order. Depending on various system statistics, the database might perform this operation if it needs all rows in index

Oracle's Operations

1. Index table and access

- TABLE ACCESS BY INDEX ROWID:

Retrieves a row from the table using the ROWID retrieved from the preceding index lookup.

Oracle's Operations

1. Index table and access

- **TABLE ACCESS FULL:** This is also known as full table scan. Reads the entire table—all rows and columns—as stored on the disk. It is still one of the most expensive operations. Besides high IO rates, a full table scan must inspect all table rows so it can also consume a considerable amount of CPU time.

Oracle's Operations

2. Joins

- **NESTED LOOPS JOIN**
- **HASH JOIN**
- **MERGE JOIN**

Oracle's Operations

2. Joins

- **NESTED LOOPS JOIN:** Joins two tables by fetching the result from one table and querying the other table for each row from the first.
- <https://use-the-index-luke.com/sql/join/nested-loops-join-n1-problem>

Oracle's Operations

2. Joins

- **HASH JOIN:** The hash join loads the candidate records from one side of the join into a hash table that is then probed for each row from the other side of the join.
- <https://use-the-index-luke.com/sql/join/hash-join-partial-objects>

Oracle's Operations

2. Joins

- **MERGE JOIN:** The merge join combines two sorted lists like a zipper. Both sides of the join must be presorted.
- <https://use-the-index-luke.com/sql/join/sort-merge-join>

Oracle's Operations

3. Sorting and grouping

- **SORT ORDER BY**
- **SORT ORDER BY STOPKEY**
- **SORT GROUP BY**
- **SORT GROUP BY NOSORT**
- **HASH GROUP BY**

Oracle's Operations

3. Sorting and grouping

- **`SORT ORDER BY`**: Sorts the result according to the order by clause. This operation needs large amounts of memory to materialize the intermediate result

Oracle's Operations

3. Sorting and grouping

- **`SORT ORDER BY STOPKEY`**: Sorts a subset of the result according to the order by clause. Used for top-N queries if pipelined execution is not possible

Oracle's Operations

3. Sorting and grouping

- **`SORT GROUP BY`**: Sorts the result set on the group by columns and aggregates the sorted result in a second step. This operation needs large amounts of memory to materialize the intermediate result set

Oracle's Operations

3. Sorting and grouping

- **`SORT GROUP BY NOSORT`**: Aggregates a presorted set according the group by clause. This operation does not buffer the intermediate result: it is executed in a pipelined manner.

Oracle's Operations

3. Sorting and grouping

- **HASH GROUP BY:** Groups the result using a hash table. This operation needs large amounts of memory to materialize the intermediate result set (not pipelined). The output is not ordered in any meaningful way.

Primary Key

- **INDEX UNIQUE SCAN**—the operation that only traverses the index tree
- Logarithmic scalability of the index
- After that performs **TABLE ACCESS BY INDEX ROWID** operation.
- Ingredients of a slow query are not present with an **INDEX UNIQUE SCAN**.

Concatenated Indexes

- Still room for manual refinements if the key consists of multiple columns
- The column order of a concatenated index (also known as multi-column, composite or combined index) has great impact on its usability so it must be chosen carefully

Concatenated Indexes



```
1 CREATE UNIQUE INDEX
2   employee_pk ON employees (employee_id, subsidiary_id)
3
4 SELECT first_name, last_name
5   FROM employees
6  WHERE employee_id    = 123
7         AND subsidiary_id = 30
8
9 SELECT first_name, last_name
10  FROM employees
11  WHERE subsidiary_id = 20
```

Concatenated Indexes

What happens when using only one of the key columns?

- The database does not use the index.
- Instead it performs a **TABLE ACCESS FULL**.
- The database reads the entire table and evaluates every row against the where clause

Concatenated Indexes

What happens when using only one of the key columns?

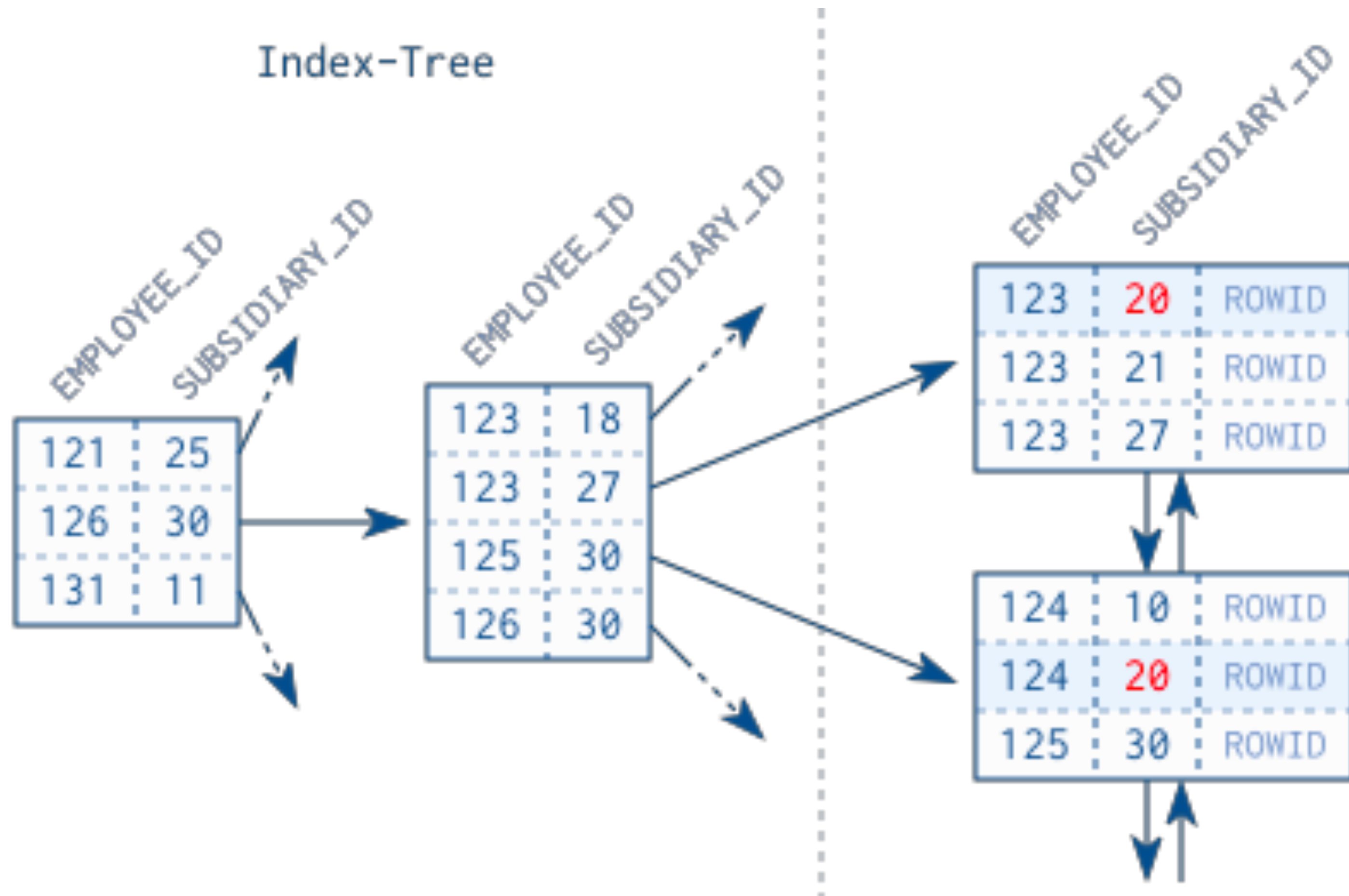
- It causes serious performance problems in production.
- The database does not use the index because it cannot use single columns from a concatenated index arbitrarily.

Concatenated Indexes

What happens when using only one of the key columns?

- The first column is the primary sort criterion and the second column determines the order only if two entries have the same value in the first column and so on.

Concatenated Indexes



Concatenated Indexes



```
1 CREATE UNIQUE INDEX
2   employee_pk ON employees (employee_id, subsidiary_id)
3
4 SELECT first_name, last_name
5   FROM employees
6  WHERE employee_id    = 123
7         AND subsidiary_id = 30
8
9 SELECT first_name, last_name
10  FROM employees
11  WHERE subsidiary_id = 20
```


Concatenated Indexes Solution?

- We could, of course, add another index on SUBSIDIARY_ID to improve query speed
- It is like a telephone directory: you don't need to know the first name to search by last name. **The trick is to reverse the index column order so that the SUBSIDIARY_ID is in the first position**

Concatenated Indexes Solution?

- **The most important consideration when defining a concatenated index is how to choose the column order so it can be used as often as possible.**



CREATE UNIQUE INDEX EMPLOYEES_I

```
1 CREATE UNIQUE INDEX EMPLOYEES_PK
2   ON EMPLOYEES (SUBSIDIARY_ID, EMPLOYEE_ID)
```

Concatenated Indexes

- The single-index solution is preferable.
- It saves storage space
- Maintenance overhead for the second index.
- The fewer indexes a table has, the better the insert, delete and update performance.

Indexes

**Using an index does
not automatically
mean a statement is
executed in the best
way possible.**

Explain Plan

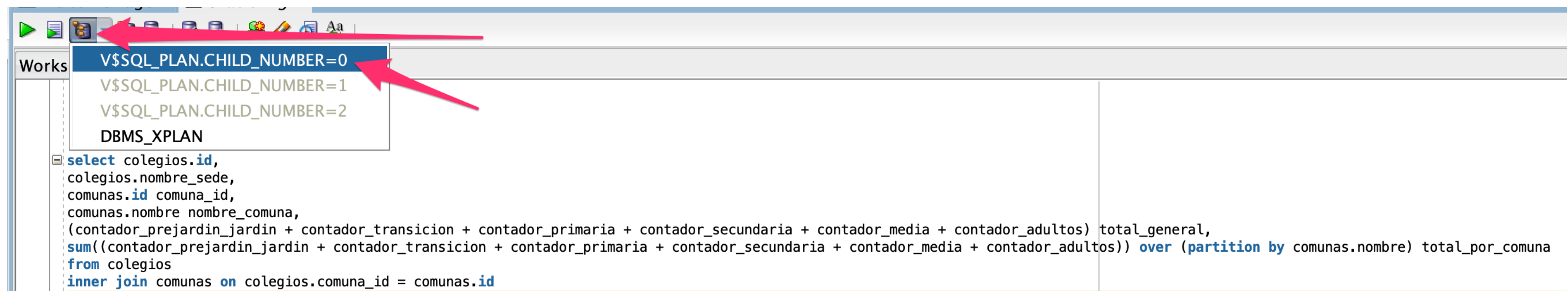
[https://docs.oracle.com/cd/B19306_01/server.102/b14211/
ex_plan.htm#i18300](https://docs.oracle.com/cd/B19306_01/server.102/b14211/ex_plan.htm#i18300)

Rows	The predicted number of rows supplied to its parent by one execution of this operation
Bytes	The predicted number of bytes (rows x expected row size) supplied to its parent by one execution of this operation
Cost	The prediction of resources required for one execution of this operation – including the resources required for every execution of every descendent of this operation that would be needed for this operation to complete once. The figure in parentheses (%CPU) is the fraction of the resource usage that can be attributed to CPU activity. As we will see below, there are two ways of interpreting the cost.
Time	The predicted elapsed time (hours:minutes:seconds) required to execute this operation just once. As with cost, the time for a single execution of the operation includes the time spent in all the executions of all the child operations needed to complete one execution of this operation.



```
1 -- Identifying Statements for EXPLAIN PLAN
2
3 EXPLAIN PLAN
4   SET STATEMENT_ID = 'st1' FOR
5 SELECT last_name FROM employees;
6
7 -- Specifying Different Tables for EXPLAIN PLAN
8
9 EXPLAIN PLAN
10    SET STATEMENT_ID = 'st1'
11    INTO my_plan_table
12  FOR
13 SELECT last_name FROM employees;
14
15 -- Displaying PLAN_TABLE Output
16
17 SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
18
19 SELECT PLAN_TABLE_OUTPUT
20    FROM TABLE(DBMS_XPLAN.DISPLAY('MY_PLAN_TABLE', 'st1', 'TYPICAL'));
21
```

Explain plan through Sql Developer



Script Output x Explain Plan x Query Result x				
SQL 0.121 seconds				
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			815	79
WINDOW		SORT	815	79
HASH JOIN			815	11
Access Predicates				
COLEGIOS.COMUNA_ID=COMUNAS.ID				
TABLE ACCESS	COMUNAS	FULL	21	2
TABLE ACCESS	COLEGIOS	FULL	815	8

Regular explain plan

```
Script Output x Explain Plan x Query Result x
SQL | All Rows Fetched: 32 in 0.021 seconds

PLAN_TABLE_OUTPUT
1 SQL_ID 6nwq9248rg7t8, child number 0
2 -----
3 select colegios.id, colegios.nombre_sede, comunas.id comuna_id,
4 comunas.nombre nombre_comuna, (contador_prejardin_jardin +
5 contador_transicion + contador_primaria + contador_secundaria +
6 contador_media + contador_adultos) total_general,
7 sum((contador_prejardin_jardin + contador_transicion +
8 contador_primaria + contador_secundaria + contador_media +
9 contador_adultos)) over (partition by comunas.nombre) total_por_comuna
10 from colegios inner join comunas on colegios.comuna_id = comunas.id
11
12 Plan hash value: 1638564981
13
14 -----
15 | Id | Operation | Name | Rows | Bytes | TempSpc | Cost (%CPU) | Time |
16 -----
17 | 0 | SELECT STATEMENT | | | | | 79 (100) | |
18 | 1 | WINDOW SORT | | | 815 | 298K | 336K | 79 (3) | 00:00:01 |
19 |* 2 | HASH JOIN | | | 815 | 298K | | 11 (10) | 00:00:01 |
20 | 3 | TABLE ACCESS FULL | COMUNAS | 21 | 2982 | | 2 (0) | 00:00:01 |
21 | 4 | TABLE ACCESS FULL | COLEGIOS | 815 | 185K | | 8 (0) | 00:00:01 |
22 -----
23
24 Predicate Information (identified by operation id):
25 -----
26
27 2 - access("COLEGIOS"."COMUNA_ID"="COMUNAS"."ID")
```

Explain plan with Postgresql

sp6 on postgres@sp6_dev

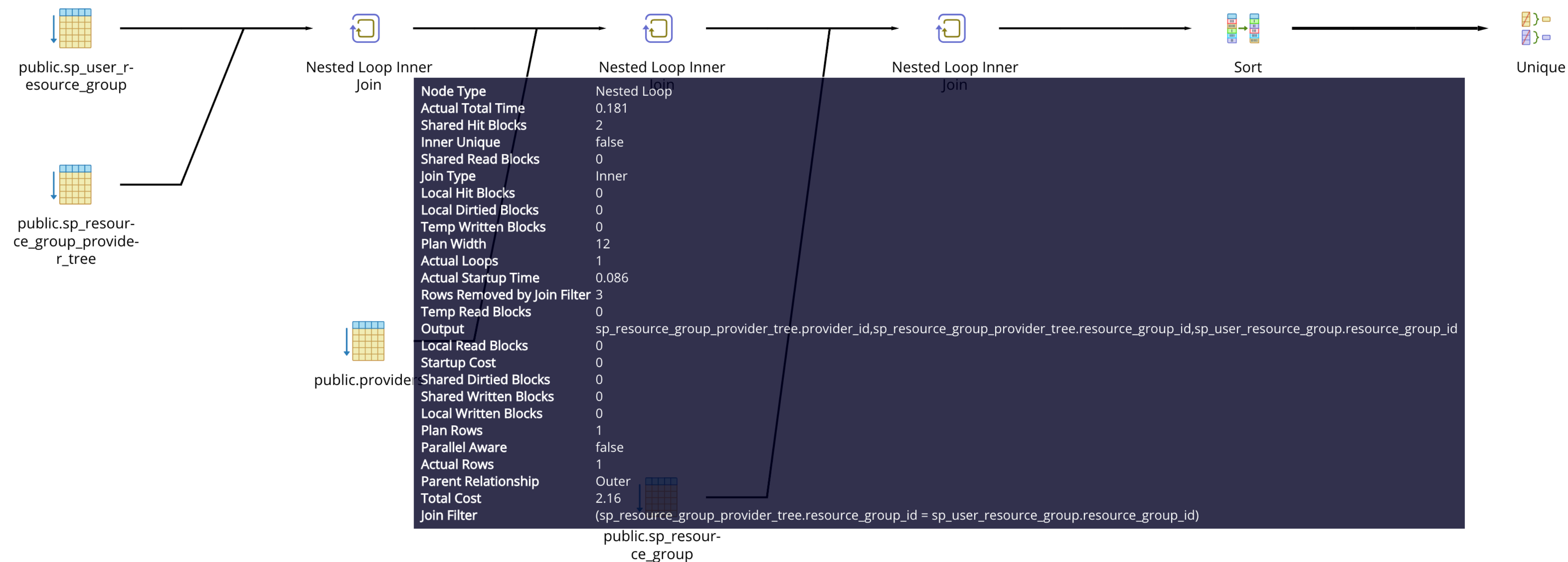
```

1 explain (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)
2 SELECT DISTINCT providers.* FROM providers
3 INNER JOIN sp_resource_group_provider_tree ON sp_resource_group_provider_tree.provider_id = providers.provider_id
4 INNER JOIN sp_resource_group ON sp_resource_group.resource_group_id = sp_resource_group_provider_tree.resource_group_id
5 INNER JOIN sp_user_resource_group ON sp_user_resource_group.resource_group_id = sp_resource_group.resource_group_id
6 WHERE sp_user_resource_group.user_id = 14

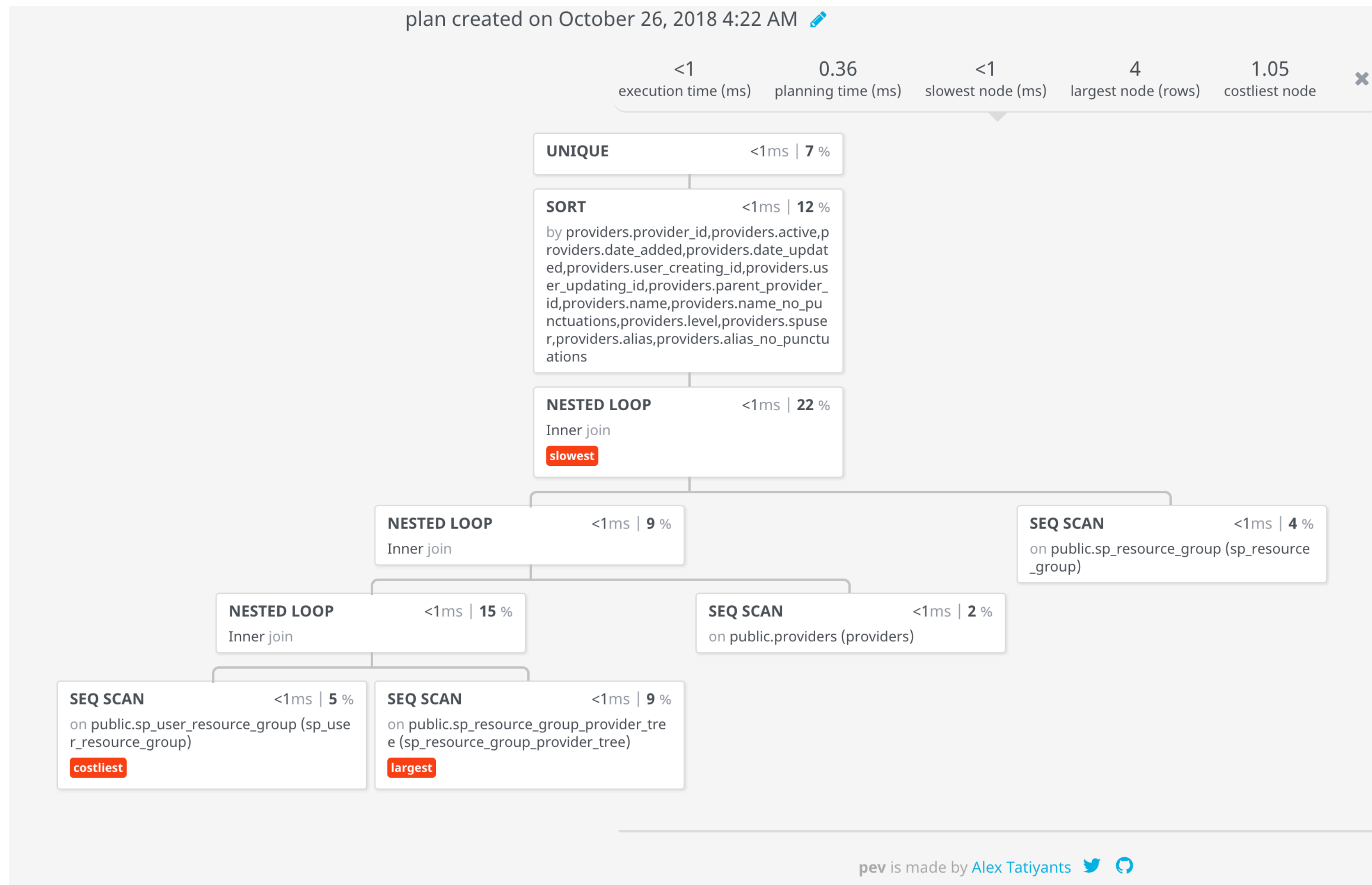
```

Data Output Explain Messages Query History

Q X Q



Explain plan with Postgresql



<http://tatiyants.com/postgres-query-plan-visualization/>

Let's practice

1. Find out the differences between each query
2. Create a View for each query
3. Create a explain plan for each query
4. Compare



```
1 -- Traiga el nombre del barrio y el número de colegios ubicados
2 -- en cada barrio de aquellas instituciones ubicadas en la comuna
3 -- de buenos aires ordenado por el número de colegios de mayor a menor.
4
5 -- Query 1
6 select barrio,
7 count(barrio) NUMERO_COLEGIOS from colegios
8 inner join comunas ON colegios.comuna_id = comunas.id
9 where comunas.NOMBRE = 'BUENOS AIRES'
10 group by barrio
11 order by NUMERO_COLEGIOS desc, barrio;
12
13 -- Query 2
14 select distinct barrio,
15 count(*) over (partition by barrio) as numero_colegios
16 from colegios
17 inner join comunas ON colegios.comuna_id = comunas.id
18 where comunas.NOMBRE = 'BUENOS AIRES'
19 order by numero_colegios desc, barrio;
```




```
1 -- Traiga los registros junto con el nombre de su comuna, para cada registro deberá calcularse el
  total de los estudiantes según los contadores. También deberá traer el total de estudiantes agrupados
  por comuna.
2
3 -- Query 1
4 SELECT DISTINCT COLEGIOS.ID,
5 COLEGIOS.NOMBRE_SEDE,
6 COLEGIOS.COMUNA_ID,
7 COMUNAS.NOMBRE AS NOMBRE_COMUNA,
8 (CONTADOR_PREJARDIN_JARDIN + CONTADOR_TRANSICION +
9  CONTADOR_PRIMARIA + CONTADOR_SECUNDARIA + CONTADOR_MEDIA + CONTADOR_ADULTOS) AS TOTAL_GENERAL,
10 SUM(CONTADOR_PREJARDIN_JARDIN + CONTADOR_TRANSICION + CONTADOR_PRIMARIA + CONTADOR_SECUNDARIA +
  CONTADOR_MEDIA + CONTADOR_ADULTOS)
11  OVER (PARTITION BY COLEGIOS.COMUNA_ID) TOTAL_POR_COMUNA
12 FROM COLEGIOS
13 JOIN JOIN COMUNAS ON COLEGIOS.COMUNA_ID = COMUNAS.ID ORDER BY NOMBRE_COMUNA, NOMBRE_SEDE;
14
15 -- Query 2
16 SELECT COLEGIOS.ID,
17 COLEGIOS.NOMBRE_SEDE,
18 COMUNAS.ID COMUNA_ID,
19 COMUNAS.NOMBRE NOMBRE_COMUNA,
20 (CONTADOR_PREJARDIN_JARDIN + CONTADOR_TRANSICION +
21  CONTADOR_PRIMARIA + CONTADOR_SECUNDARIA + CONTADOR_MEDIA + CONTADOR_ADULTOS) TOTAL_GENERAL,
22 SUM((CONTADOR_PREJARDIN_JARDIN + CONTADOR_TRANSICION + CONTADOR_PRIMARIA + CONTADOR_SECUNDARIA +
  CONTADOR_MEDIA + CONTADOR_ADULTOS))
23  OVER (PARTITION BY COMUNAS.NOMBRE) TOTAL_POR_COMUNA
24 FROM COLEGIOS
25 INNER JOIN COMUNAS ON COLEGIOS.COMUNA_ID = COMUNAS.ID ORDER BY NOMBRE_COMUNA, NOMBRE_SEDE;
```

Take an integer n ($n \geq 0$) and a digit d ($0 \leq d \leq 9$) as an integer.

Square all numbers k ($0 \leq k \leq n$) between 0 and n .

Count the numbers of digits d used in the writing of all the k^2

For instance:

$n = 10$, $d = 1$, the k^2 are 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

We are using the digit 1 in 1, 16, 81, 100. The total count is then 4.

$n = 25$, $d=1$ there are 11 digits `1` for the squares of numbers between 0 and 25.

Thank you!