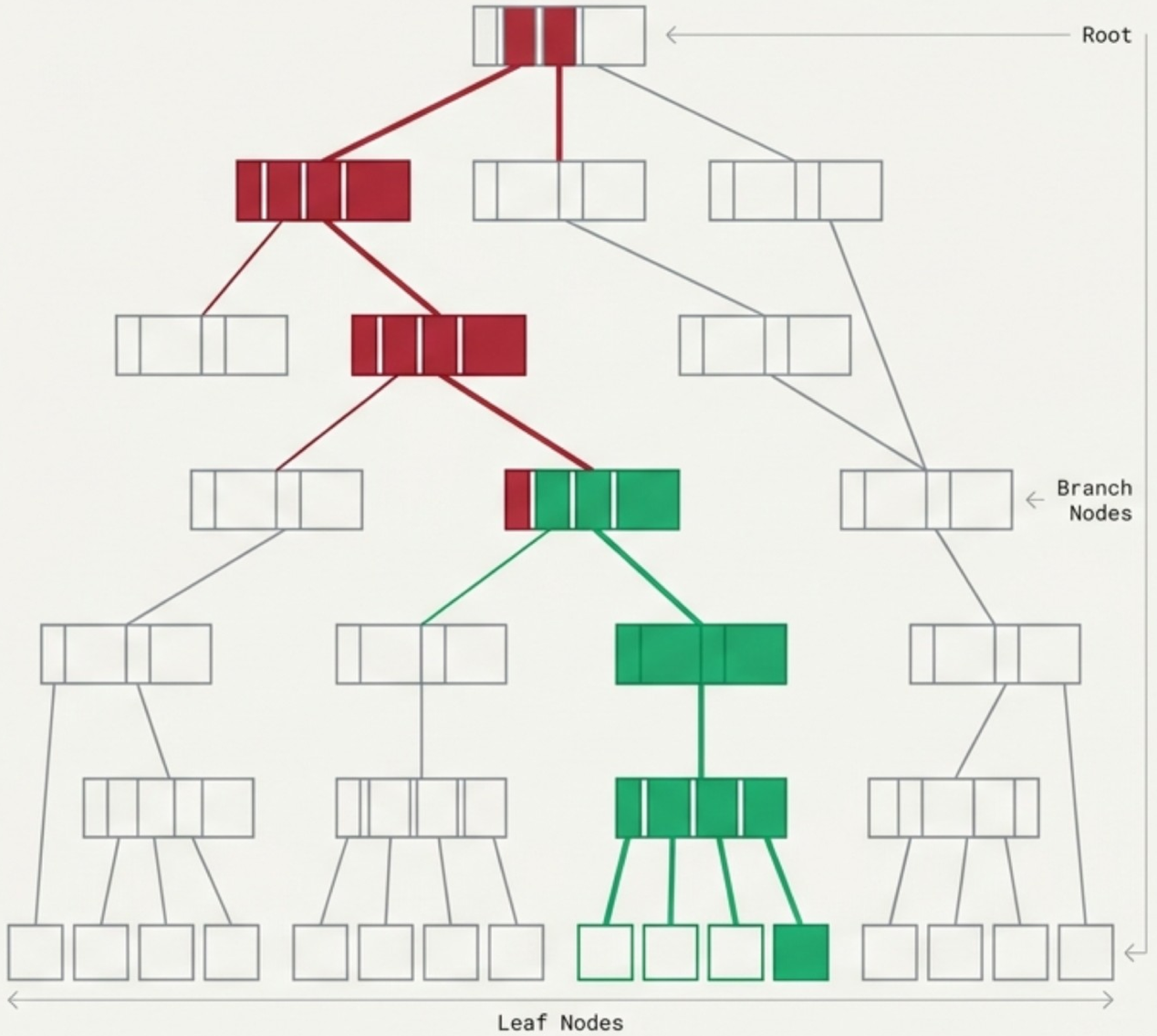
Oracle Composite Indexes

The Golden Rule of Column Order.



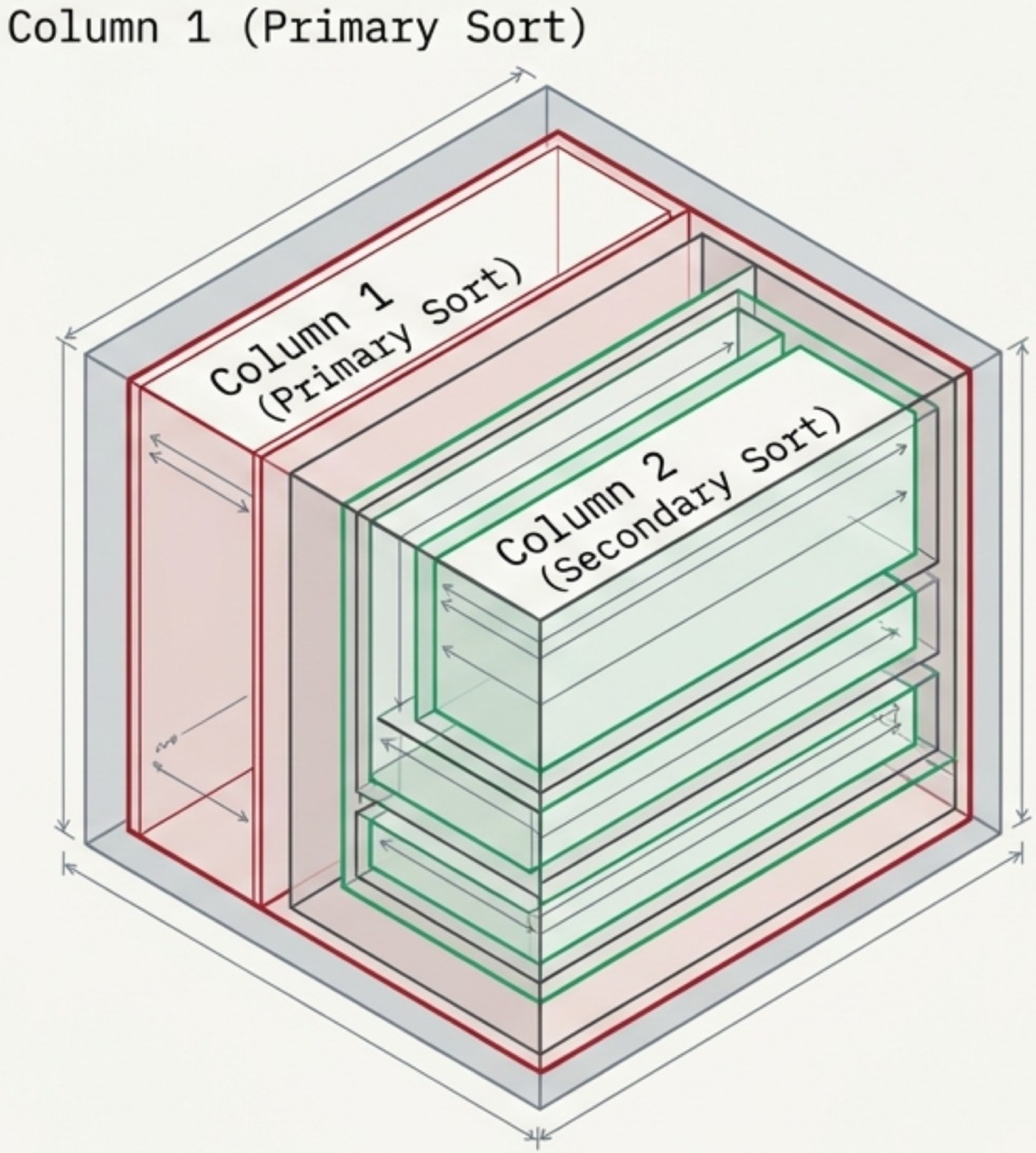
Physical Organization Dictates Efficiency

The Mechanism:

When creating a multi-column index, Oracle strictly sorts the data by the **first column**, then the second.

The Consequence:

Column order is not a logical preference; it is a physical layout that completely dictates the Optimizer's efficiency.



THE GOLDEN RULE OF INDEXING

EQUALITY (=)

goes before

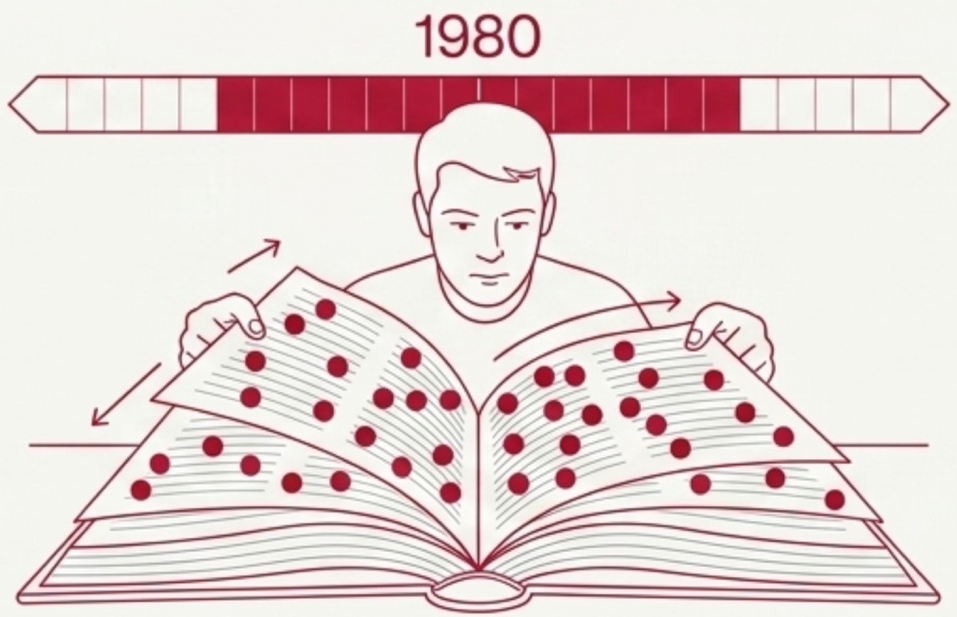
RANGE (>, <, BETWEEN, LIKE)

The Phone Book Analogy: Searching for ‘Smith, born in 1980’

Sort by DOB → Last Name

Wide Scan

TECHNICAL NIBE



TECHNICAL IMAL

Sort by Last Name → DOB

Direct Seek

VECTOR VIEW

TECHNICALLIMAL

Mechanics: Navigation vs. Inspection

Access Predicate = Navigation

Tells Oracle exactly where to start and stop traversing the B-Tree.

Filter Predicate = Inspection

Oracle must inspect the physical row and discard wasted work.

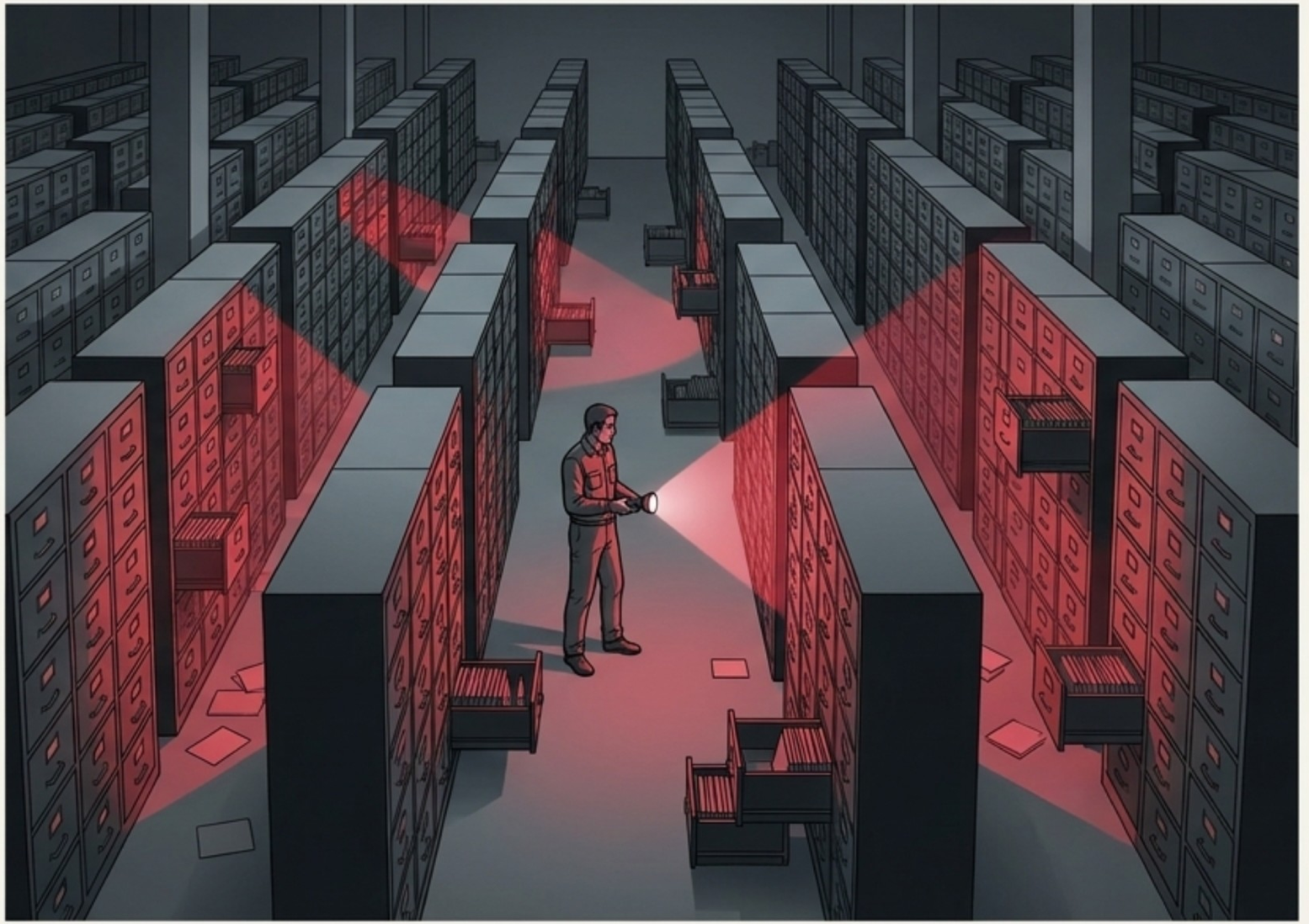
Scenario A: Range First Creates a Labyrinth

```
Index: (order_date, status)
```

```
Query:
order_date > '01-JAN-2023'
AND status = 'ACTIVE'
```

The Physics: Oracle jumps to the date boundary and scans forward. ‘ACTIVE' statuses are scattered randomly randomly across the date range.

The Result: Status checks become scattered, causing massive unnecessary reads.



The Labyrinth of Data (Range First)

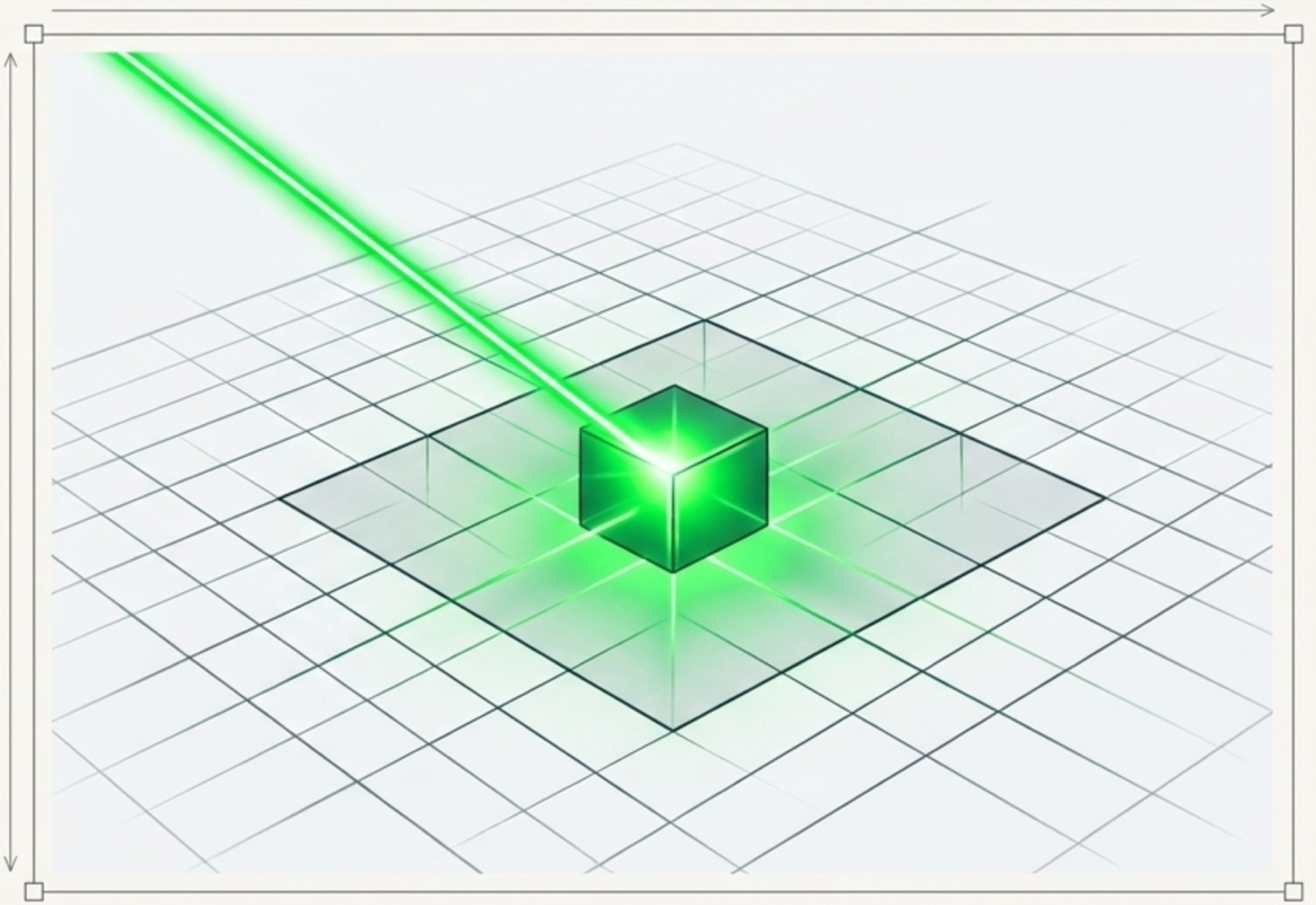
Scenario B: Equality First Creates a Laser

```
Index: (status, order_date)
```

```
Query:
order_date > '01-JAN-2023'
AND status = 'ACTIVE'
```

The Physics: Oracle jumps **directly** to the 'ACTIVE' section. Within that bucket, data is **perfectly sorted** by date.

The Result: The scan is **highly focused**. Both columns are used for access, ensuring **exact reads**.



The Quantum Filter (Equality First)

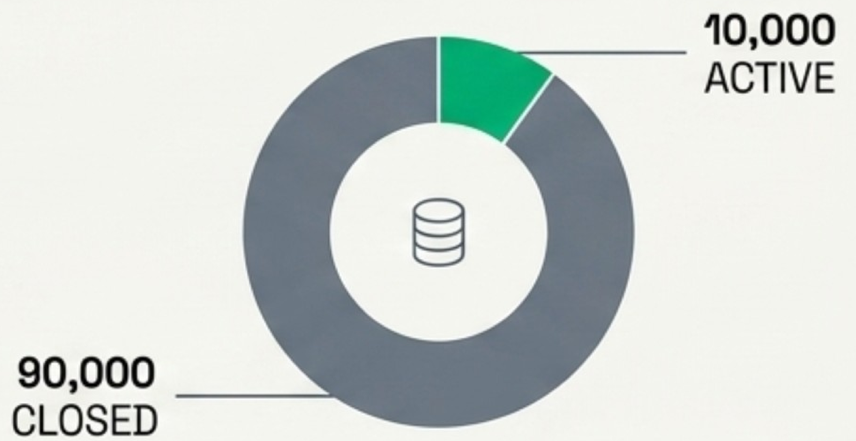
The Proving Ground: SQL Setup Overview

Total Volume

100,000

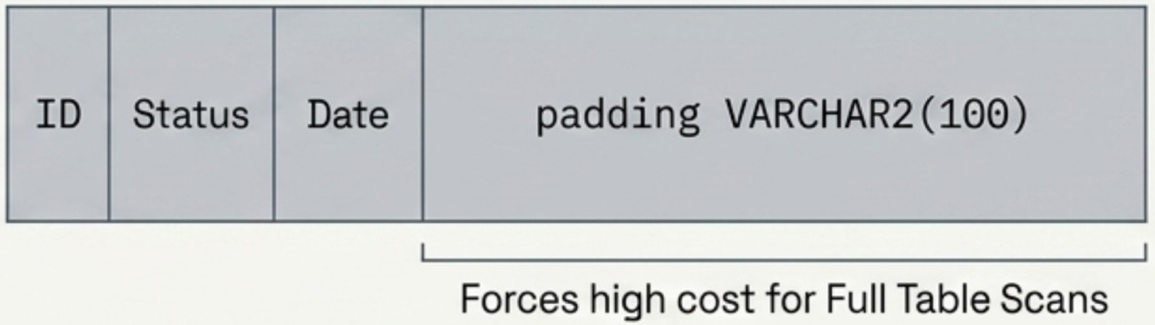
Total Rows

Distribution



Timeline Spread

Row Width



Execution Plan: The Cost of Scattered Reads

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ORDER_DATE">SYSDATE@!-30)
       filter("STATUS"='ACTIVE')
```

The status is merely a filter applied after the read. This results in heavy wasted effort inspecting rows, or a complete fallback to a Full Table Scan.

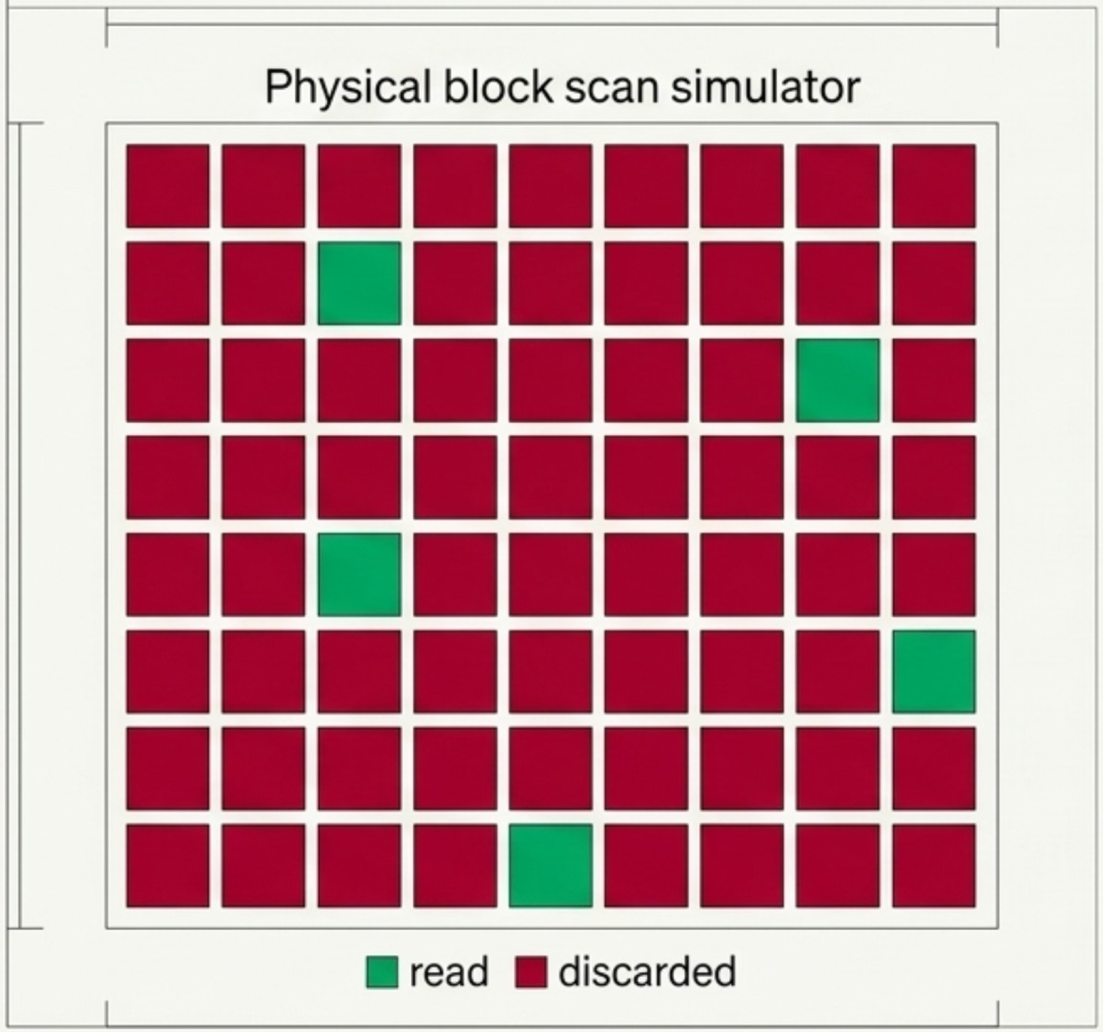
Execution Plan: The Combined Access Predicate

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("STATUS"='ACTIVE' AND "ORDER_DATE">
         SYSDATE@!!-30)
```

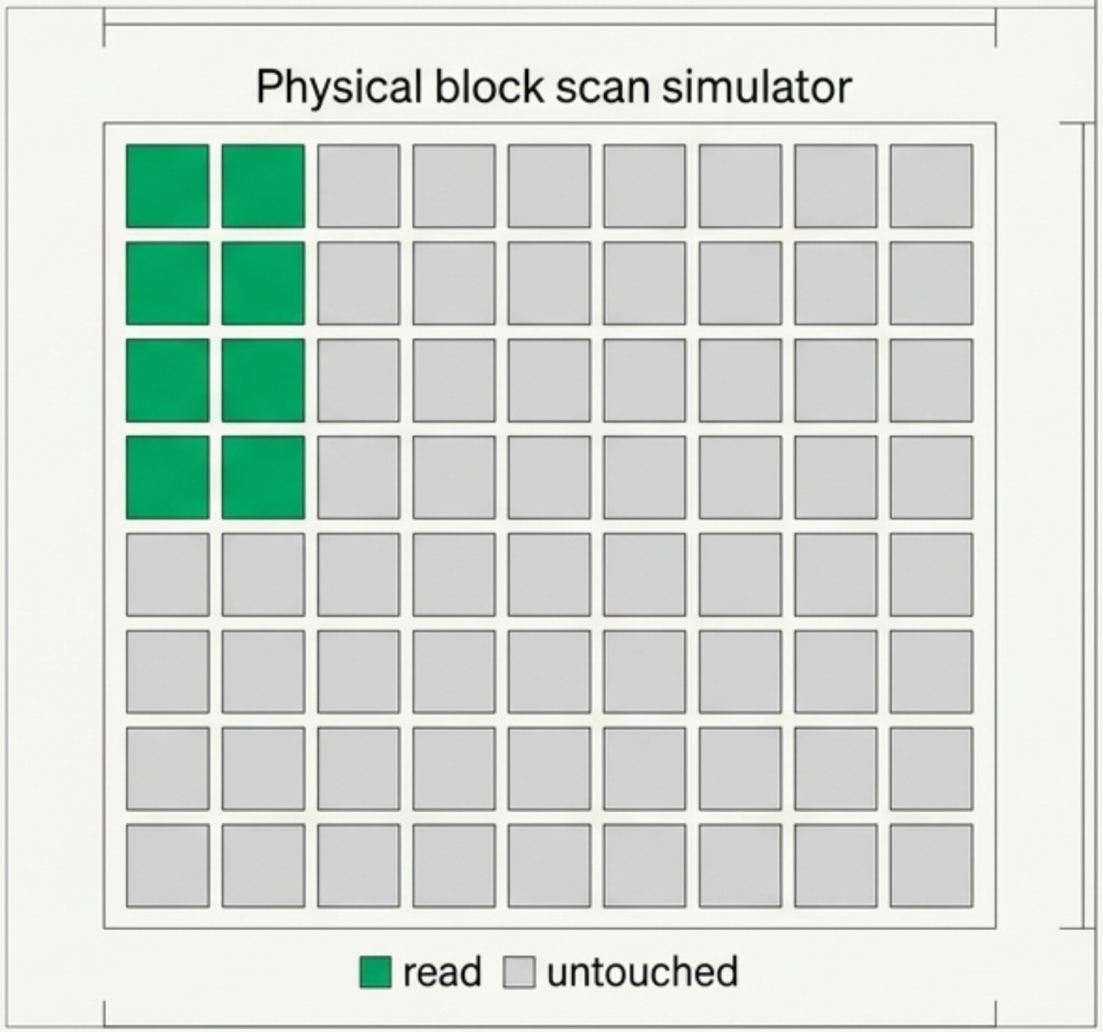
Because Equality came first, the Optimizer combined BOTH conditions into the ACCESS predicate. Navigation goes straight to the target. Zero wasted reads.

The Physical Reality: Block Behavior

Index Range Scan (Range First)



Index Range Scan (Equality First)



The Optimizer's Choice: Comparison Matrix

	Range First (Date, Status)	Equality First (Status, Date)
Primary Sort Strategy	Wide chronological spread	Strict category grouping
Predicate Classification	Access + Filter	Combined Access
Physical Block Behavior	High scatter / discard rate	Direct pinpoint reads
Fallback Risk	High risk of Full Table Scan	Highly stable index usage

The Index Design Checklist

1. Map the Predicates

Identify which columns in your query use = and which use ranges (>, <, BETWEEN).

2. Prioritize Equality

Force your WHERE clause equalities to the very front of your composite index definition.

3. Verify the XPLAN

Use `gather_plan_statistics` and check `Predicate Information`. If a targeted column is trapped as a filter, check your column order.