# Composite Index

Suppose that we have a table **boarding\_passes** with data like this

A screenshot of a computer

Description automatically generated

After taking a closer look, I see that the **seat\_no** has the highest cardinality and the **boarding\_no** column with the second position. Along with that it has 7.925.812 records, suitable for index testing.

We will go ahead on implementing the index like this under below

CREATE INDEX idx\_boarding\_passes\_boarding\_seat\_no ON bookings.boarding\_passes USING btree (seat\_no, boarding\_no);

Result without index

A screenshot of a computer program

Description automatically generated

With index

Both **seat\_no** and **boarding\_no columns**

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE seat\_no = 'A11' AND boarding\_no = 139;

A screen shot of a computer

Description automatically generated

With **seat\_no** column only

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE seat\_no = 'A11';

A screen shot of a computer

Description automatically generated

With **GROUP BY** and **WHERE** statement

EXPLAIN ANALYSE  
SELECT boarding\_no, *count*(\*) AS cnt  
FROM bookings.boarding\_passes  
WHERE seat\_no = '10A'  
GROUP BY boarding\_no

Without index

A screenshot of a computer

Description automatically generated

With index

A screenshot of a computer

Description automatically generated

Reverse condition statements

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE boarding\_no = 139  
 AND seat\_no = 'A11';

Still apply index

A screen shot of a computer

Description automatically generated

Only contains the second index column (**boarding\_no**)

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE boarding\_no = 139;

Does not apply index

A screenshot of a computer

Description automatically generated

Theory confirmation: Composite index columns with high cardinality (**seat\_no, boarding\_no**) prove to significantly improve the efficiency of the query while retrieving information based on indexed columns. Also, the query must include the first-most indexed column to work and continue with next indexed columns.

**Index Failure 01**

A screenshot of a computer

Description automatically generated

Trigram indexes

CREATE EXTENSION pg\_trgm;  
  
CREATE INDEX idx\_boarding\_passes\_seat\_no\_trigram ON bookings.boarding\_passes USING gin (seat\_no gin\_trgm\_ops);

A computer screen shot of a computer

Description automatically generated

Collate index

CREATE INDEX idx\_boarding\_passes\_seat\_no\_collate ON bookings.boarding\_passes USING btree (seat\_no collate "C");

A screenshot of a computer program

Description automatically generated

Theory confirmation: It does apply index when the character **“%”** doesn’t first appear in the compare string but for the two indexes **trigram** and **collate** it is much slower when compared **to seq scan (full table scan)**

**Index Failure 02**

CREATE INDEX idx\_boarding\_passes\_boarding\_seat\_no ON bookings.boarding\_passes USING btree (seat\_no, boarding\_no);

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE seat\_no = 'A11' OR boarding\_no = 139;

**A screenshot of a computer

Description automatically generated**

Theory confirmation: The query with **OR** condition will not apply index even though both **seat\_no** and **boarding\_no** have been indexed as the query will start to sequentially scan the table again.

**Index Failure 03**

CREATE INDEX idx\_boarding\_passes\_boarding\_no ON bookings.boarding\_passes USING btree (boarding\_no);

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE boarding\_no = '139';

Result

A screenshot of a computer

Description automatically generated

Theory confirmation: Passed in the string number force the type casting from string to int. Therefore, the index was successfully fully applied even though the **boarding\_no** column has the datatype of integer.

**Index Failure 04**

CREATE INDEX idx\_boarding\_passes\_boarding\_seat\_no ON bookings.boarding\_passes USING btree (seat\_no, boarding\_no);

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE *length*(seat\_no) = 2;

A screenshot of a computer

Description automatically generated

Theory confirmation: Using function on indexed column (**seat\_no**) won’t work.

**Index Failure 05**

CREATE INDEX idx\_boarding\_passes\_boarding\_no ON bookings.boarding\_passes USING btree (boarding\_no);

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE boarding\_no - 1 = 138;

Result  
A screenshot of a computer

Description automatically generated

Theory confirmation: Using expression on indexed column (**boarding\_no**) won’t apply index as well

However, when the index involves multiple columns, the index would work as normal but for the non-expression indexed column

CREATE INDEX idx\_boarding\_passes\_boarding\_no ON bookings.boarding\_passes USING btree (boarding\_no, seat\_no);

EXPLAIN ANALYSE  
SELECT \*  
FROM bookings.boarding\_passes  
WHERE boarding\_no - 1 = 138 AND seat\_no = 'A11';

Result

A screenshot of a computer program

Description automatically generated

**Index Clustering and Non-Clustered Index**

**Query before clustering index**

-- Test Query 1: Range scan on boarding\_no  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM boarding\_passes  
WHERE boarding\_no BETWEEN 1 AND 100  
ORDER BY boarding\_no;

A screenshot of a computer program

Description automatically generated

-- Test Query 2: Specific flight lookup  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM boarding\_passes  
WHERE flight\_id = 169878  
ORDER BY boarding\_no;

A screenshot of a computer program

Description automatically generated

Cluster index column **boarding\_no**

CREATE INDEX idx\_boarding\_passes\_boarding\_no  
 ON boarding\_passes USING btree (boarding\_no);

CLUSTER boarding\_passes USING idx\_boarding\_passes\_boarding\_no;

**Query after clustering index**

-- Test Query 1: Range scan on boarding\_no  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM boarding\_passes  
WHERE boarding\_no BETWEEN 1 AND 100  
ORDER BY boarding\_no;

A screenshot of a computer

Description automatically generated

-- Test Query 2: Specific flight lookup  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM boarding\_passes  
WHERE flight\_id = 169878  
ORDER BY boarding\_no;

A screenshot of a computer

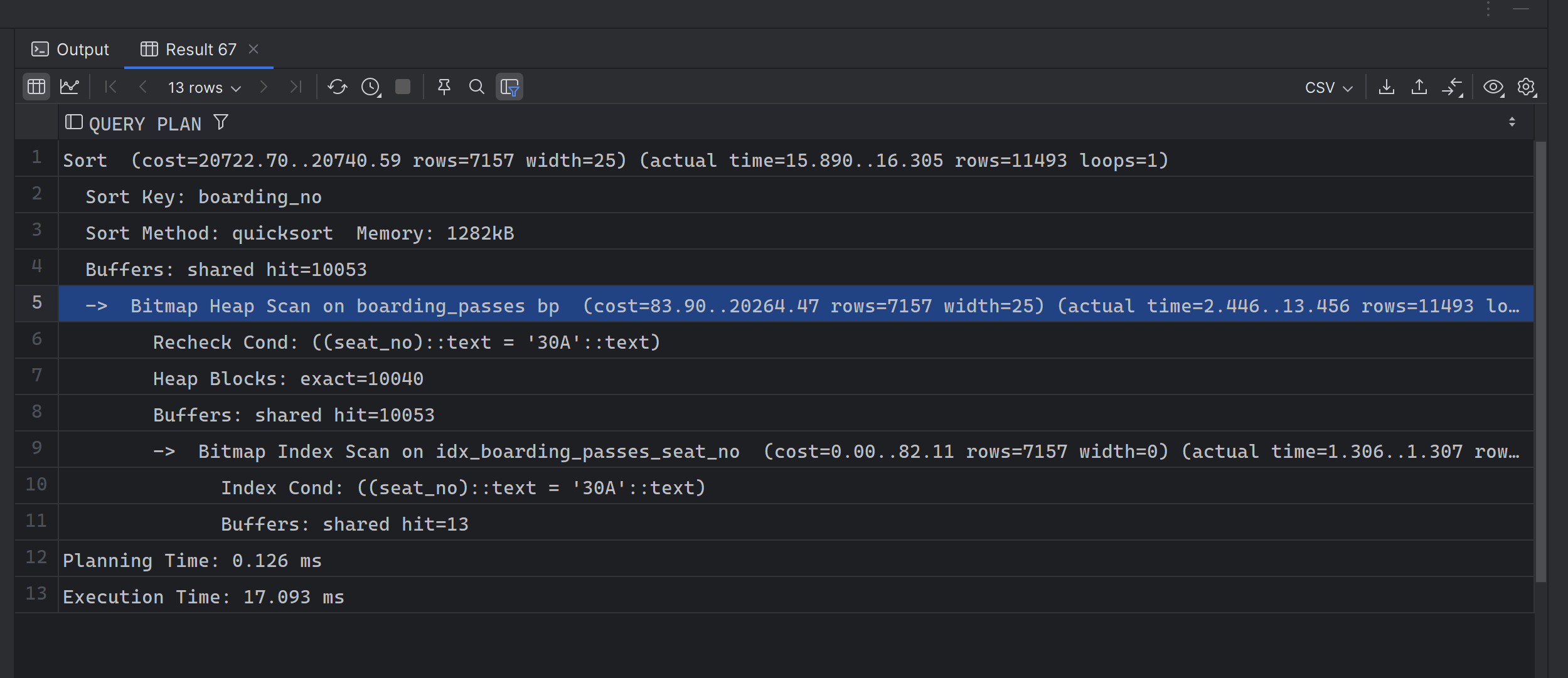
Description automatically generated

Conclusion: After cluster index on column **boarding\_no**, the number of IO on buffer memory has been reduced significantly as expected.

**Non-clustered index test case**

-- Test Query:  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT bp.\*  
FROM boarding\_passes bp  
WHERE seat\_no = '30A'  
ORDER BY boarding\_no;

Result



The result does show that for a query using secondary index (**seat\_no**) would take two read disks. One for non-clustered, the other if for the clustered index.