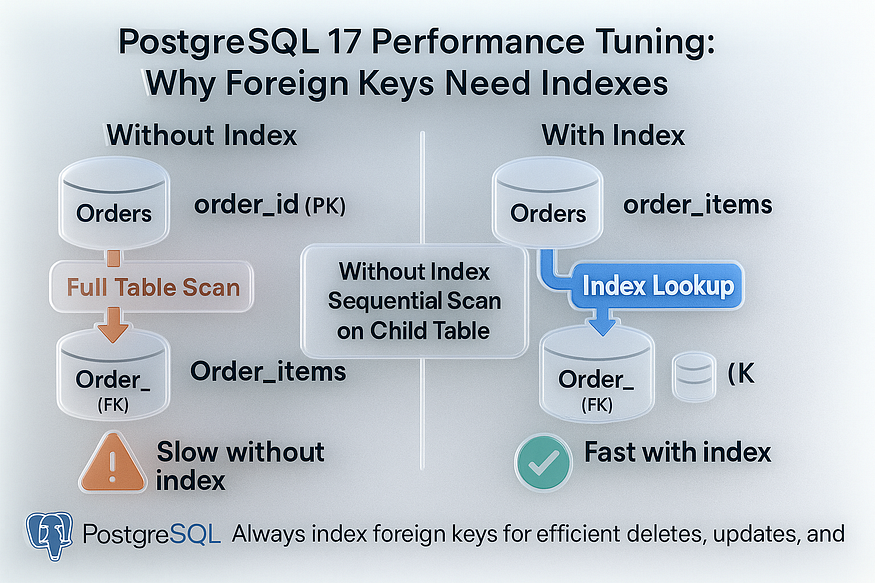
# **16 -PostgreSQL 17 Performance Tuning: Why Foreign Keys Need Indexes**



When designing relational databases, parent-child relationships are common. A classic case is the ****Orders**** table (parent) and the ****Items**** table (child). Each order can have many items, so the child table typically has far more rows than the parent.

👉 In such cases, it’s always a good practice to create an index on the ****foreign key column**** in the child table.

There are two main reasons:

1. ****Faster joins**** → Queries joining parent and child tables almost always join on the foreign key. An index dramatically speeds this up.
2. ****Faster parent changes**** → When deleting or updating parent rows, PostgreSQL must check child rows for constraint enforcement. Without an index, these checks can be painfully slow.

Let’s build a demo to see the impact.

## **Step 1: Create Parent and Child Tables**

-- Parent table: Orders  
CREATE TABLE orders (  
 order\_no SERIAL PRIMARY KEY,  
 order\_date DATE DEFAULT CURRENT\_DATE  
);

postgres=# -- Parent table: Orders  
CREATE TABLE orders (  
 order\_no SERIAL PRIMARY KEY,  
 order\_date DATE DEFAULT CURRENT\_DATE  
);  
CREATE TABLE  
postgres=#

-- Child table: Items  
CREATE TABLE items (  
 item\_no SERIAL PRIMARY KEY,  
 order\_no INT,  
 product\_name TEXT,  
 description TEXT,  
 created\_at TIMESTAMP DEFAULT now(),  
 CONSTRAINT fk\_order FOREIGN KEY (order\_no)  
 REFERENCES orders(order\_no)  
 ON UPDATE CASCADE  
 ON DELETE CASCADE  
);

postgres=# -- Child table: Items  
CREATE TABLE items (  
 item\_no SERIAL PRIMARY KEY,  
 order\_no INT,  
 product\_name TEXT,  
 description TEXT,  
 created\_at TIMESTAMP DEFAULT now(),  
 CONSTRAINT fk\_order FOREIGN KEY (order\_no)  
 REFERENCES orders(order\_no)  
 ON UPDATE CASCADE  
 ON DELETE CASCADE  
);  
CREATE TABLE  
postgres=#

## **Step 2: Populate Tables with Data**

We’ll insert ****1 million rows**** into orders and****3 million rows**** into items (4 items per order).

-- Insert 1M orders  
INSERT INTO orders (order\_no)  
SELECT g FROM generate\_series(1, 1000000) g;

postgres=# -- Insert 1M orders  
INSERT INTO orders (order\_no)  
SELECT g FROM generate\_series(1, 1000000) g;  
INSERT 0 1000000  
postgres=#

-- Insert 3M items (4 per order)  
INSERT INTO items (order\_no, product\_name, description)  
SELECT (g/4)+1, 'Product\_' || g, 'Desc\_' || g  
FROM generate\_series(1, 3000000) g;

postgres=# -- Insert 3M items (4 per order)  
INSERT INTO items (order\_no, product\_name, description)  
SELECT (g/4)+1, 'Product\_' || g, 'Desc\_' || g  
FROM generate\_series(1, 3000000) g;  
INSERT 0 3000000  
postgres=#

👉 Now:

* orders has 1M rows.
* items has 3M rows linked by the foreign key.

## **Step 3: Query Without Index on Foreign Key**

Let’s try a join query:

\timing  
SELECT \*   
FROM orders o  
JOIN items i ON o.order\_no = i.order\_no  
WHERE o.order\_no = 120;

postgres=# \timing  
Timing is on.  
postgres=#  
postgres=# SELECT \*  
FROM orders o  
JOIN items i ON o.order\_no = i.order\_no  
WHERE o.order\_no = 120;  
 order\_no | order\_date | item\_no | order\_no | product\_name | description | created\_at  
----------+------------+----------+----------+--------------+-------------+----------------------------  
 120 | 2025-09-02 | 12000476 | 120 | Product\_476 | Desc\_476 | 2025-09-02 17:16:37.825555  
 120 | 2025-09-02 | 12000477 | 120 | Product\_477 | Desc\_477 | 2025-09-02 17:16:37.825555  
 120 | 2025-09-02 | 12000478 | 120 | Product\_478 | Desc\_478 | 2025-09-02 17:16:37.825555  
 120 | 2025-09-02 | 12000479 | 120 | Product\_479 | Desc\_479 | 2025-09-02 17:16:37.825555  
(4 rows)  
  
Time: 194.021 ms  
postgres=#

****Result:****

* PostgreSQL performs a ****sequential scan**** on the items table (4 million rows) to find just 4 matches.
* Execution is slow.

Check the plan:

EXPLAIN ANALYZE  
SELECT \*   
FROM orders o  
JOIN items i ON o.order\_no = i.order\_no  
WHERE o.order\_no = 120;

postgres=# EXPLAIN ANALYZE  
SELECT \*  
FROM orders o  
JOIN items i ON o.order\_no = i.order\_no  
WHERE o.order\_no = 120;  
 QUERY PLAN  
-------------------------------------------------------------------------------------------------------------------------------  
 Nested Loop (cost=1000.42..44661.88 rows=4 width=51) (actual time=0.249..198.825 rows=4 loops=1)  
 -> Index Scan using orders\_pkey on orders o (cost=0.42..8.44 rows=1 width=8) (actual time=0.016..0.020 rows=1 loops=1)  
 Index Cond: (order\_no = 120)  
 -> Gather (cost=1000.00..44653.40 rows=4 width=43) (actual time=0.230..198.794 rows=4 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 -> Parallel Seq Scan on items i (cost=0.00..43653.00 rows=2 width=43) (actual time=122.458..187.599 rows=1 loops=3)  
 Filter: (order\_no = 120)  
 Rows Removed by Filter: 999999  
 Planning Time: 0.105 ms  
 Execution Time: 198.853 ms  
(11 rows)  
  
Time: 200.196 ms  
postgres=#

👉 You’ll see a ****Seq Scan on items****, which is inefficient.

## **Step 4: Add Index on the Foreign Key Column**

Now add an index:

CREATE INDEX idx\_items\_order\_no ON items(order\_no);

postgres=# CREATE INDEX idx\_items\_order\_no ON items(order\_no);  
CREATE INDEX  
postgres=#

## **Step 5: Query With Index**

Run the same query again:

EXPLAIN ANALYZE  
SELECT \*   
FROM orders o  
JOIN items i ON o.order\_no = i.order\_no  
WHERE o.order\_no = 120;

postgres=# EXPLAIN ANALYZE  
SELECT \*  
FROM orders o  
JOIN items i ON o.order\_no = i.order\_no  
WHERE o.order\_no = 120;  
 QUERY PLAN  
-----------------------------------------------------------------------------------------------------------------------------------  
 Nested Loop (cost=0.85..16.98 rows=4 width=51) (actual time=0.030..0.041 rows=4 loops=1)  
 -> Index Scan using orders\_pkey on orders o (cost=0.42..8.44 rows=1 width=8) (actual time=0.017..0.018 rows=1 loops=1)  
 Index Cond: (order\_no = 120)  
 -> Index Scan using idx\_items\_order\_no on items i (cost=0.43..8.50 rows=4 width=43) (actual time=0.008..0.011 rows=4 loops=1)  
 Index Cond: (order\_no = 120)  
 Planning Time: 0.091 ms  
 Execution Time: 0.060 ms  
(7 rows)  
  
Time: 0.442 ms  
postgres=#

****Result:****

* PostgreSQL switches to an ****Index Scan on items****.
* The query time drops dramatically.

👉 Instead of scanning 3M rows, it jumps directly to the 4 matching rows.

## **Step 6: Parent Delete Performance**

Indexes also help when deleting from the parent:

DELETE FROM orders WHERE order\_no = 500;

postgres=# DELETE FROM orders WHERE order\_no = 500;  
DELETE 1  
Time: 4.355 ms  
postgres=#

* Without an index on items.order\_no, PostgreSQL must scan the entire items table to check for child rows.
* With the index, it can instantly find and remove matching rows.

## **Key Takeaways**

* Always ****index foreign key columns**** in child tables.
* Benefits:
* ****Joins are faster**** (avoid sequential scans).
* ****Parent updates/deletes are efficient**** (quick referential checks).
* Without indexes, queries and deletes on large parent-child datasets can degrade to ****full-table scans****, crippling performance.

✅ In PostgreSQL 17, just as in earlier versions, indexing foreign keys is one of the simplest yet most impactful tuning steps for real-world workloads.