**❌ What is a Query Antipattern in Cosmos DB for PostgreSQL (Citus)?**

A **query antipattern** refers to an inefficient or suboptimal way of querying distributed data, leading to poor performance, high latency, or excessive resource consumption. These typically arise when queries violate the principles of distributed computing—particularly **colocation** and **distribution-awareness**.

**🧠 Analogy: "Asking the Wrong People"**

Imagine you’re in a company where data is stored in filing cabinets by department:

* Sales → Cabinet A
* Marketing → Cabinet B
* HR → Cabinet C

Now suppose you want employee sales records. Instead of asking only Cabinet A, you shout:  
“Everyone give me anything related to employees!”  
Now all departments start searching, even when they don’t have relevant data.

➡️ This is a query antipattern: **you didn’t target the right data sources**, so the system **broadcasts the query** everywhere, causing inefficiency.

**💥 Common Query Antipatterns**

| **Antipattern** | **Description** | **Fix** |
| --- | --- | --- |
| 🔁 **Cross-Shard Joins without Colocation** | Joining two distributed tables not colocated causes full data movement across nodes. | Use colocate\_with during table distribution. |
| 📦 **Querying without WHERE on Distribution Column** | Causes data scatter to all shards. | Always include a filter on the distribution column. |
| 🧮 **ORDER BY / LIMIT on distributed table without index** | Sorting globally across shards is expensive. | Use indexed fields or sort after coalescing. |
| 🧊 **Too many small shards (shard bloat)** | Too many shards make metadata overhead unbearable. | Use appropriate shard count. |
| 📢 **SELECT \* over massive distributed table** | Returns excessive data and causes bottlenecks. | Project only necessary columns. |

**🧪 Demo Example: Cross-Shard Join (Antipattern)**

sql

-- Assume both tables are distributed on different keys

SELECT u.login, e.event\_type

FROM payment\_users u

JOIN payment\_events e ON u.user\_id = e.user\_id;

**Problem**: If payment\_users is distributed on user\_id but payment\_events is on event\_id, this join will result in **data movement between shards**.

**✅ Better Version: Use Colocation**

-- Ensure both tables are distributed on 'user\_id' and colocated

SELECT create\_distributed\_table('payment\_users', 'user\_id');

SELECT create\_distributed\_table('payment\_events', 'user\_id', colocate\_with := 'payment\_users');

-- Now the same query becomes efficient

SELECT u.login, e.event\_type

FROM payment\_users u

JOIN payment\_events e ON u.user\_id = e.user\_id;



More AP Exmaples

**⚠️ Query Antipatterns (Based on Our Schema)**

**🔁 1. Cross-Shard Join Without Colocation**

*SELECT u.login, m.name*

*FROM payment\_users u*

*JOIN payment\_merchants m ON u.user\_id = m.merchant\_id;*

**Why it's bad**:

* payment\_users is distributed on user\_id, and payment\_merchants on merchant\_id.
* Join happens **on unrelated keys**, triggering **full data shuffle** across all shards.

**Fix**:

* Use reference tables for small lookup tables like payment\_merchants.
* Or **colocate** on the same key if meaningful.

**🔍 2. Full Table Scan Without WHERE on Distribution Column**

*SELECT \* FROM payment\_events WHERE event\_type = 'purchase';*

**Why it's bad**:

* No filter on user\_id (the distribution key).
* Triggers **query on all shards**.

**Fix**:

SELECT \* FROM payment\_events

WHERE user\_id = 123 AND event\_type = 'purchase';

**🔁 3. Group By Without Filtering**

sql

CopyEdit

SELECT user\_id, COUNT(\*) FROM payment\_events GROUP BY user\_id;

**Why it's bad**:

* Though user\_id is the distribution column, **no filtering** leads to **all-shards aggregation**.

**Fix**:

* Use WHERE user\_id = ... if scoped to user
* Or optimize with materialized rollup tables or pre-aggregations.

**❌ 4. Non-Colocated Join on Distributed Tables**

sql

CopyEdit

SELECT e.event\_type, m.name

FROM payment\_events e

JOIN payment\_merchants m ON e.merchant\_id = m.merchant\_id;

**Why it's bad**:

* payment\_merchants is **reference or differently distributed**
* Join on merchant\_id, which is not the distribution key of payment\_events.

**Fix**:

sql

CopyEdit

-- Make `payment\_merchants` a reference table

SELECT create\_reference\_table('payment\_merchants');

**⚖️ 5. Overusing SELECT \* in Distributed Tables**

sql

CopyEdit

SELECT \* FROM payment\_events WHERE user\_id = 123;

**Why it's bad**:

* Returns unnecessary columns (event\_details, created\_at, etc.)
* Wasteful in distributed context.

**Fix**:

sql

CopyEdit

SELECT event\_type FROM payment\_events WHERE user\_id = 123;

**🧱 6. Too Many Small Shards**

sql

CopyEdit

-- Implicitly creating with too many shards

SELECT create\_distributed\_table('payment\_users', 'user\_id', shards\_count := 10000);

**Why it's bad**:

* Overhead in metadata, query planning, and shard management.

**Fix**:

* Use a **sensible shard count** (e.g., 32, 64) based on data volume and node count.