Why Drop Foreign Keys in Scalable Systems?

Foreign Keys (FK) enforce referential integrity at the database level, they require **extra checks**, **locking**, and **synchronous coordination**, which introduces **latency**, **blocking**, and **complexity** — all of which hurt **horizontal scalability** and **write throughput**.

Let's break this down technically and practically

1. Foreign Keys Introduce Synchronous Checks on Every Write

Example:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY
);

CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INTEGER REFERENCES users(id)
);
```

Now when you run:

```
INSERT INTO orders (user_id) VALUES (123);
```

PostgreSQL must:

- **Look up users.id = 123** to confirm the user exists (even if indexed).
- **Lock users during insert** to avoid race conditions with concurrent deletes.

On millions of writes per second, this creates:

- Index pressure on users.id

- Contention on user rows
- Cascading rollback complexity on failure

2. FKs Cause Lock Contention (Especially in Concurrent Inserts/Deletes)

Scenario:

1. You're inserting into orders rapidly (write-heavy).
2. Another process tries to delete a user in users.

You get **RowExclusiveLock** → **ShareRowExclusiveLock** blocking. PostgreSQL waits until inserts finish to delete the user, or vice versa.

In a high-throughput system:

- This causes **delays**, **timeouts**, or even **deadlocks**.
- Write amplification from enforcement grows fast.

3. FK Enforcement Slows Bulk Inserts & Batch Jobs

Inserting 10 million rows into a child table with FK:

- Each row requires FK validation — that's **10 million index lookups**.
- PostgreSQL's planner has to **confirm consistency**, which **thrashes cache and I/O**.
- In batch jobs, it's better to **defer validation** or do it at app level.

4. Foreign Keys Don’t Work Across Shards or Microservices

If you're scaling to:

- **Distributed databases (e.g., Citus)**
- **Multi-region setups**
- **Service-oriented architectures**

👉 You **can’t enforce FKs across databases or services**.

Example:

- User lives in users_eu (Europe) shard.
- Order lives in orders_asia shard.
- PostgreSQL cannot enforce the constraint → **FKs are disabled in Citus**.

5. Cascading Deletes Are Dangerous at Scale

```
ON DELETE CASCADE

Imagine deleting a user cascades to delete:
- 1,000 orders
- 10,000 order items
- 50,000 audit logs
```

On **DELETE FROM users WHERE id = X** can become **hundreds of thousands of operations**, locking several tables.

At scale:

- You don't want to **trigger multi-table writes** implicitly.
- Safer to **emit a deletion event** and process deletes asynchronously.

6. How FKs Affect CRUD Performance

Operation	How FKs Slow It Down
INSERT	Lookup in parent table, lock acquisition, index contentior
UPDATE FK field	Must check parent validity and update index, can block concurrent reads
DELETE parent	Check for matching child rows, or cascade, or block unti child rows are gone
BULK INSERT	Each row = FK lookup + index hit = slow + RAM/CPU heav

Why Move Validation to the App?

In a **horizontally scalable app**:

- The **database focuses only on raw reads/writes** (dumb storage).
- All **integrity logic lives in the app layer or event bus**.
- Faster writes, async processing, resilience to spikes.

App-level checks:

```
func CreateOrder(order Order) error {
    if !userRepo.Exists(order.UserID) {
        return errors.New("invalid user")
    }
    return orderRepo.Insert(order)
}
```

✔️ No FK, but same logic
✔️ Can use **Redis cache** to check fast
✔️ Better suited to scale-out systems (Kubernetes, sharded databases, CQRS)

When to Drop FKs:

Situation	Drop FKs?	Why
Small monolith app (e.g., blog)	***No***	FKs add safety and aren't bottlene
Mid-size app with some traffic	***Maybe***	Depends on write load and latenc
High-scale system (10K+ QPS)	***Yes***	FKs limit horizontal scaling
Distributed DB or microservices	***Yes***	FKs don’t work cross-db
Using Citus, Yugabyte, CockroachDB	***Must***	Most don’t allow distributed FKs

Real-World Application work on No FKs

- **GitHub**: App-level data integrity for many FK-like relationships.
- **Facebook**: No FKs; uses validation layers and async jobs.
- **Google**: Services own their own DBs; rely on IDs, events, and async validation.

Conclusion

Foreign keys are **great for correctness**, but **not designed for high throughput or distributed systems**. As you scale, **drop FKs**, **handle validation at the app or service layer**, and **use async/event-driven patterns** to keep things fast and safe.