

<https://openai.com/index/chatgpt/> I am trying to build a replication system between databases using java 25, springboot 4, jpa and kafka. I wold like to implemet an infrastructure for this replication mechanism. I need a base repository for all entities that handles replication using outbox pattern. All insert, update, delete operations made by this repository should be reflected to outbox table transparently. How can I achieve this?

Absolutely — you can build a **transparent, replication-ready base repository** in Spring Boot 4 / Java 25 that uses the **Outbox Pattern** to mirror all insert/update/delete operations to an **outbox table**, which you can then stream to **Apache Kafka** for replication.

Below is a **practical architecture + implementation guide** that will give you exactly that — a *base repository* that writes all persistence changes (CRUD) into an **outbox table** as part of the same JPA transaction, using Spring Data JPA + Spring Kafka.

📌 High-Level Architecture

1. Domain Repositories

Standard Spring Data JPA repositories extended with a *base interface*.

2. Outbox Table

A dedicated table that stores *domain change events* (INSERT / UPDATE / DELETE), usually with:

sql

```
id BIGINT AUTO_INCREMENT,  
aggregate_type VARCHAR,  
aggregate_id VARCHAR,  
operation VARCHAR,  
payload JSON,  
created_at TIMESTAMP
```

3. Outbox Writer / Event Publisher

- On every entity change, create an Outbox record.
- A *background poller* or *CDC (Debezium)* reads the outbox and publishes to Kafka.

4. Kafka Topics

Messages from outbox records are published to topics like `entity-changes`.

5. Consumer (Replication Service)

Other databases/services consume and apply the events for replication.

This pattern ensures **atomic persistence of data and events** in the same transaction (the key benefit of the *Transactional Outbox Pattern*). 

Step 1 — Define the Outbox Entity

java

```
@Entity
@Table(name = "outbox_event")
public class OutboxEvent {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String aggregateType;
    private String aggregateId;
    private String operation;      // CREATE/UPDATE/DELETE
    @Lob
    private String payload;        // JSON of entity state

    private LocalDateTime createdAt = LocalDateTime.now();
}
```

Step 2 — Create a Base Repository Interface

Define a base interface for all entities that should replicate:

java

```
@NoRepositoryBean
public interface ReplicatingRepository<T, ID> extends JpaRepository<T, ID> {
    // Optional custom methods, if needed
}
```

Now provide a custom implementation that **intercepts CRUD operations and writes to the Outbox**:

Step 3 — Custom Base Repository Implementation

java

```
public class ReplicatingRepositoryImpl<T, ID>
    extends SimpleJpaRepository<T, ID>
    implements ReplicatingRepository<T, ID> {

    private final EntityManager em;
    private final OutboxEventRepository outboxRepo;
    private final ObjectMapper objectMapper;
```

```
public ReplicatingRepositoryImpl(  
    JpaEntityInformation<T, ?> info,  
    EntityManager em,  
    OutboxEventRepository outboxRepo,  
    ObjectMapper objectMapper) {  
    super(info, em);  
    this.em = em;  
    this.outboxRepo = outboxRepo;  
    this.objectMapper = objectMapper;  
}  
  
@Override  
@Transactional  
public <S extends T> S save(S entity) {  
    boolean isNew = super.getEntityInformation().isNew(entity);  
    S saved = super.save(entity);  
    writeOutbox(saved, isNew ? "CREATE" : "UPDATE");  
    return saved;  
}  
  
@Override  
@Transactional  
public void delete(T entity) {  
    super.delete(entity);  
    writeOutbox(entity, "DELETE");  
}  
  
private void writeOutbox(T entity, String operation) {  
    try {  
        String payload = objectMapper.writeValueAsString(entity);  
        OutboxEvent event = new OutboxEvent();  
        event.setAggregateType(entity.getClass().getSimpleName());  
        event.setAggregateId(getId(entity).toString());  
        event.setOperation(operation);  
        event.setPayload(payload);  
        outboxRepo.save(event);  
    } catch (JsonProcessingException e) {  
        throw new RuntimeException("Failed to serialize entity", e);  
    }  
}  
  
private Object getId(T entity) {  
    return em.getEntityManagerFactory().getPersistenceUnitUtil().getIdentifier(entity);
```

```
}
```

```
}
```

This implementation **delegates to Spring Data JPA** and adds an additional step to write the outbox. This happens in the *same transaction*, guaranteeing consistency.

Medium

🔧 Step 4 — Enable Custom Repository Factory

Wire the custom implementation in Spring Boot:

java

```
@SpringBootApplication
@EnableJpaRepositories(
    basePackages = "com.yourorg",
    repositoryBaseClass = ReplicatingRepositoryImpl.class
)
public class Application {}
```

Now all your Spring Data repositories that extend `ReplicatingRepository` will automatically persist changes to the outbox table.

Example:

java

```
public interface PersonRepository
    extends ReplicatingRepository<Person, UUID> {}
```

🚚 Step 5 — Publish Outbox to Kafka

There are two common ways to push the outbox table into Kafka:

Option A — DB Poller / Scheduler

A scheduled job reads unsent outbox records, publishes them to Kafka using `KafkaTemplate`, then marks them as sent.

Option B — CDC (recommended for scale)

Use **Debezium (Kafka Connect)** to watch the outbox table and push changes to Kafka without application-level polling.

GitHub

Debezium will:

1. Read the outbox table's transaction log.
 2. Emit changes to Kafka topics.
 3. Make the system horizontally scalable and reliable.
-

□ Notes on Replication

Once Kafka events flow, other services (or replication consumers):

- Subscribe to topics.
- Deserialize event payloads.
- Apply create/update/delete operations to target DB.

This builds a **near-real-time replication mechanism** across databases.

🧠 Additional Tips & Best Practices

- Use **JSON serialization** (Jackson) for payloads — it's flexible for arbitrary entities
 - Consider adding **schema versioning** and **event metadata** (user, timestamp)
 - For high throughput, use **batch outbox publishing**
 - Monitor the outbox for backpressure and retention
 - Choose CDC for *exact-once semantics* and order guarantees in multi-instance environments
-

📌 Summary

- Define a base repository that persists to both the entity table and outbox table atomically
 - Use Spring Data's `repositoryBaseClass` to transparently add the outbox logic
 - Publish outbox records to Kafka (Debezium or Scheduler)
 - Build loosely coupled replication consumers
-

If you want **sample code** (GitHub-ready template) or help **generating schema migrations** for the outbox, just tell me what database you're using!

   Sources

Can this code be enriched? For example while converting to json we can

create json objects only include changed parts maybe?

Yes — you can *absolutely enrich your outbox events so they contain only the parts of the entity that actually changed*. That's a very useful optimization, especially for replication or event-sourcing systems where payload size and clarity matter.

Below are several ways to improve your JSON serialization strategy to only include *changed fields* or *meaningful deltas* instead of serializing the entire entity every time.

🧠 1) Track *Changed Fields* Before Serialization

By default, JPA/Hibernate dirty-checking will detect what fields have changed in the persistence context — but it doesn't expose a straightforward list of *which properties changed* out of the box unless you intercept internal events. Ekino ...

Option A — Use Hibernate Event Listeners

You can register a `PostUpdateEventListener` with Hibernate which gives you both *oldState* and *newState* for an entity's properties. Hibernate's internal API will let you derive which properties changed:

java

```
public class DirtyFieldListener implements PostUpdateEventListener {  
  
    @Override  
    public void onPostUpdate(PostUpdateEvent event) {  
        String[] propertyNames = event.getPersister().getPropertyNames();  
        Object[] oldState = event.getOldState();  
        Object[] newState = event.getState();  
  
        Map<String, Object> changes = new HashMap<>();  
        for (int i = 0; i < propertyNames.length; i++) {  
            if (!Objects.equals(oldState[i], newState[i])) {  
                changes.put(propertyNames[i], newState[i]);  
            }  
        }  
  
        // serialize "changes" to JSON for outbox  
    }  
}
```

Then register this listener in your configuration.

 The listener gives you **only the fields that actually changed**, rather than serializing the entire entity.

Ekino FR

2) Generate a *Delta JSON* Instead of Full JSON

Once you have a map of just changed fields:

```
java

Map<String, Object> delta = ...; // changes only

String payload = objectMapper.writeValueAsString(delta);
```

Instead of:

```
java

ObjectMapper.writeValueAsString(fullEntity);
```

This JSON will be significantly smaller and focused on *change context* — exactly what replication systems want.

3) Use Custom Jackson Serializers or Filters

If you prefer to keep control on the *serialization side*, you can:

Option B — Use Jackson property filtering

```
java

@JsonFilter("changedFieldsFilter")
public class YourEntity { ... }
```

Then register a filter:

```
java

FilterProvider filters = new SimpleFilterProvider()
    .addFilter("changedFieldsFilter",
        SimpleBeanPropertyFilter.filterOutAllExcept(changedFieldNames));
String json = objectMapper.writer(filters).writeValueAsString(entity);
```

This still results in a delta JSON. You drive the list of field names (`changedFieldNames`) from your dirty tracker.

🛠 4) Combine with @DynamicUpdate (Optional but Helpful)

If you annotate your entities with `@DynamicUpdate`, Hibernate will only generate SQL for modified columns — and you can piggyback on this for cleaner events too. [Baeldung on...](#)

java

```
@Entity  
@DynamicUpdate  
public class Person { ... }
```

This doesn't change JSON directly, but it aligns your persistence behavior with the notion of *only modified data matters*.

📌 Summary: How to Generate Minimal JSON

Approach	Captures Only Changed Fields	Complexity
Hibernate Event Listener (<code>PostUpdateEvent</code>)	✓	Medium
Jackson Custom Filter	✓ (with info from listener)	Low–Medium
Serializing full entity every time	✗	Easy

✓ Recommended Implementation

1. Detect changed fields with a Hibernate event listener
2. Build a map of only the changed fields
3. Serialize that map as your outbox payload
4. Optionally add metadata like old vs new values

If you want, I can give you a **complete sample project** wiring this into Spring Boot 4 + Kafka outbox, including *event publishing and CDC integration*.

 [e.](#) Sources

Can we detect generated sqls and send those sql statements instead of json payload?

Yes — you *can* capture the actual SQL statements Hibernate generates and send those into your outbox instead of a JSON payload, but it's *more complex* and comes with trade-offs. JPA/Hibernate doesn't expose SQL statements as first-class domain events, but there *are ways* to intercept or construct them before execution.

Here are the practical options:

🛠 1) Use Hibernate's `StatementInspector` to intercept generated SQL

Hibernate provides a built-in interface `StatementInspector` that allows you to inspect (and even modify) every SQL statement *before* it's executed. You can register an implementation in Spring Boot:

```
java

public class OutboxStatementInspector implements StatementInspector {
    @Override
    public String inspect(String sql) {
        // capture or transform SQL statement here
        outboxHandler(sql);
        return sql;
    }
}
```

Then register it in your application properties:

```
properties

spring.jpa.properties.hibernate.session_factory.statement_inspector=com.yourorg.OutboxStatementInsp
```

This gives you the raw SQL string Hibernate *is about to run* — including INSERT/UPDATE/DELETE — and lets you deposit it in the outbox. Vlad Mihalcea

Important:

- You'll still often see parameter placeholders (`?`). To resolve actual values you'd need a JDBC proxy or deeper binding interception.
- This *also sees every SQL*, including selects and metadata queries, so you need filtering logic.

💡 2) Use a JDBC proxy like P6Spy for full SQL + parameters

If your goal is to **capture *exact* SQL with parameters** (not just Hibernate's SQL template), then a JDBC proxy like **P6Spy** is a good fit.

Instead of trying to capture SQL inside Hibernate, you let P6Spy wrap the JDBC driver and capture every executed DML with real bound values. You can then:

1. Configure P6Spy to log all statements (with parameters). Baeldung on...
2. Process those logs in your app and write matching records to your outbox.

Example `application.properties`:

```
properties

spring.datasource.driver-class-name=com.p6spy.engine.spy.P6SpyDriver
spring.datasource.url=jdbc:p6spy:mysql://localhost/yourdb
```

P6Spy config lets you log or capture bind variables so you get executable SQL, not placeholders.

Baeldung on Kot...

👉 With this approach you're capturing **exact statements that hit the DB**, which you can replay elsewhere (e.g., other databases).

🧠 3) Generate SQL manually based on entity metadata

Instead of trying to *catch Hibernate's internal SQL*, you can programmatically **generate the equivalent SQL statements yourself** — e.g., using JPA metamodels:

- Based on entity class, table, and column mapping metadata
- Based on the values changed
- Emit `INSERT ...`, `UPDATE ... SET ...`, `DELETE ... WHERE ...`

This gives you *clean, replayable SQL* without needing to tap into Hibernate internals.

🟡 Downsides:

- Complex — you need full mapping metadata
- Might diverge from what Hibernate actually does
- Harder to support bulk/batch operations

📌 Trade-offs

Approach	Captures Full SQL	Captures Real Parameter Values	Easy to Implement	Works for All DML
StatementInspector	✓	✗ (placeholders)	Medium	✓
P6Spy (JDBC Proxy)	✓	✓	Medium	✓
Manual SQL generation	✓ (custom)	✓	Hard	✓

⚠ What You Cannot Easy Do

Hibernate does *not* provide an official event that hands you prepared SQL with bound values ready to replay — only the SQL template before binding. Vlad Mihalcea

So if you need replayable statements (with exact bind values), a JDBC-level interception layer like P6Spy or a custom JDBC wrapper is the only reliable path short of manual SQL construction.

✓ Recommended Strategy (for Replication)

1. Use JDBC interception (e.g., P6Spy) to capture exactly executed INSERT/UPDATE/DELETE statements with bind values.
2. Filter for DML only (ignore selects, metadata).
3. Write SQL + metadata to your outbox as the replication payload.
4. Create a consumer that replays those statements against the target database.

This gives you a **true SQL replication stream** rather than JSON event payloads.

❖ Notes

- Hibernate logging (e.g., `show_sql=true`) logs basic SQL but doesn't expose it programmatically and often uses `?` parameter placeholders. Mastertheboss
- Combining `StatementInspector` with a custom binder interceptor could be possible, but it's more work than using a JDBC proxy.

If you want, I can provide **sample code** for integrating P6Spy with Spring Boot and streaming those intercepted SQL statements to your outbox table. Just let me know!

Which approach do you suggest first or second?

For your use-case — capturing *actual SQL statements* for replication — there are **two main interception strategies** we discussed:

1. Hibernate's `StatementInspector`
2. **A JDBC proxy like `P6Spy` that captures executed SQL with bound parameters

Both are valid, but they fit *different goals* and complexity levels. Here's how I'd **recommend which to try first** based on your priorities:

FIRST — Try a JDBC Proxy (e.g., P6Spy)

 Best when you want real, executable SQL statements (including parameter values) for replication.

Why start here?

- It captures **exact statements as executed**, with bind parameters resolved — you don't just get templates with `?` placeholders. [p6spy.readthed...](#)
- It operates at the JDBC level, so **every insert/update/delete gets intercepted**, regardless of whether it's JPA/Hibernate or raw JDBC. [docs.jboss.org](#)
- Once set up, you can **transform and store full SQL statements** into your outbox — no extra work reconstructing queries or hooking JPA internals.

Challenges

- You must configure P6Spy as the JDBC driver or wrap your DataSource properly. [codingtechroom...](#)
- You may need output filtering so you only capture DML relevant for replication.

 When this is the best choice:

-  You want *actual executable SQL commands* for replaying on another DB.
-  You don't want to reverse-engineer how Hibernate constructs statements.
-  You want minimal dependence on internal Hibernate APIs.

SECOND — Hibernate's `StatementInspector`

 Good for lightweight SQL inspection before execution.

Why second?

- The `StatementInspector` interface lets you see the SQL Hibernate *generates* before it's prepared.
docs.hibernate.o... Vlad Mihalcea
- It's useful for **logging or short-circuiting statements** or if you want to *inspect or rewrite* SQL before execution.
- It requires **less external tooling** compared to P6Spy.

Limitations

⚠ The statements you get often contain **parameter placeholders** (?) — not the actual bound values — which means you *cannot replay them directly* without extra binding reconstruction. Medium

👉 Good for *debugging or analysis*, but not ideal if your goal is to generate **replayable replication SQL**.

📊 Quick Comparison

Approach	Captures Bound Values	Captures Every DML	Ease of Use	Best For
P6Spy / JDBC Proxy	✓	✓	Medium	Actual replication SQL
StatementInspector	✗ (placeholders only)	✓	Easy	SQL logging / inspection
Manual SQL generation	✓ (custom)	✓	Hard	Domain-specific custom replication

🟡 My Recommendation

👉 Start with a JDBC proxy like P6Spy

It gives you *true executable SQL with parameters*, which is exactly what you want for a replication system — *no need to reverse-engineer or reconstruct SQL later*. Capturing the actual commands your app runs makes replaying them on target databases *much simpler and reliable*.

Once that's working, you can *enrich or augment* your replication strategy with other tools (e.g., filtering only INSERT/UPDATE/DELETE, batching, etc.).

🚀 What to Do Next

If you want, I can provide:

- Example Spring Boot configuration for integrating P6Spy with your DataSource
- Example code for writing captured SQL into your outbox table
- Guidance on filtering/formatting SQL before sending to Kafka

Just tell me *which database* you're using (PostgreSQL, MySQL, etc.) and I'll tailor it!

    Sources

Give me this code example for all of them for PostgreSQL

Below is a concrete **Spring Boot + PostgreSQL** example showing how to capture/stream SQL statements using **all three approaches** you asked about (JDBC proxy = P6Spy, Hibernate `StatementInspector`, and programmatic Hibernate interceptor). You can adapt these into your outbox pattern to send the statements for replication.

Approach #1 — Use P6Spy (captures full SQL with parameters)

Goal: Get *actual* SQL as executed (with bound parameters) without heavy internal Hibernate hooks — this is usually the best starting point for replication.

1) Add dependency

xml

```
<dependency>
    <groupId>com.github.gavlyukovskiy</groupId>
    <artifactId>p6spy-spring-boot-starter</artifactId>
    <version>1.9.0</version>
</dependency>
```

(This starter wraps the JDBC driver and logs SQL including parameter values.)

Cloud-orient...

2) Configure application.properties

Use the P6Spy driver and PostgreSQL URL:

properties

```
spring.datasource.url=jdbc:p6spy:postgresql://localhost:5432/mydb
spring.datasource.username=postgres
```

```
spring.datasource.password=yourpass  
spring.datasource.driver-class-name=com.p6spy.engine.spy.P6SpyDriver
```

You'll see SQL prints like:

```
pgsql
```

```
insert into person (name, age) values ('Alice', 30)
```

...with real values, not placeholders.

Cloud-oriented ...

3) Customize logging (optional)

Create `spy.properties` under `src/main/resources`:

```
properties  
  
driverlist=org.postgresql.Driver  
appender=com.p6spy.engine.spy.appender.StdoutLogger  
logMessageFormat=com.p6spy.engine.spy.appender.SingleLineFormat
```

⚠️ P6Spy will now *intercept all JDBC calls*, including Hibernate-generated SQL. [php.cn](#)

4) Capture statements into your Outbox

Instead of just printing, you can implement a listener:

```
java  
  
@Bean  
public JdbcEventListener mySqlCaptureListener(OutboxRepo outboxRepo) {  
    return new JdbcEventListenerAdapter() {  
        @Override  
        public void onAfterAnyExecute(StatementInformation info, long time, SQLException e) {  
            String sql = info.getSqlWithValues(); // full SQL  
            outboxRepo.save(new OutboxEvent(sql));  
        }  
    };  
}
```

(Note: this requires wiring P6Spy event listeners — starter setups like `spring-boot-data-source-decorator` support this.) [GitHub](#)

Approach #2 — Hibernate StatementInspector

Goal: Inspect every SQL statement generated by Hibernate before execution.

Good for observation and filtering, but it won't include bound parameters by default.

Vlad Mihalcea

1) Define the inspector

```
java

public class SqlCaptureInspector implements StatementInspector {

    @Override
    public String inspect(String sql) {
        System.out.println("SQL about to execute: " + sql);
        // save to outbox if DML
        if (sql.trim().matches("(?i)^insert|update|delete).*")) {
            outboxRepo.save(new OutboxEvent(sql));
        }
        return sql;
    }
}
```

2) Enable in application.properties

```
properties
```

```
spring.jpa.properties.hibernate.session_factory.statement_inspector=com.yourorg.SqlCaptureInspector
```

Spring Boot will register the inspector with Hibernate, and `inspect(sql)` runs before each statement.

Stack Overflow

⚠ Parameters are not inlined — this gives templates like:

```
pgsql
```

```
INSERT INTO person (name, age) VALUES (?, ?)
```

...but this is often enough for replication if you replay them with your own parameter sourcing logic.

Hibernate

✓ Approach #3 — Programmatic Hibernate Interceptor (`EmptyInterceptor`)

Goal: Hook at the ORM level to inspect entity changes, and you can also capture the SQL (or construct it). This is closer to application logic than raw SQL.

1) Implement interceptor

```
java
```

```
public class SqlInterceptor extends EmptyInterceptor {  
  
    @Override  
    public String onPrepareStatement(String sql) {  
        // Runs right before Hibernate executes the statement  
        if (sql.matches("(?i)^insert|update|delete).*")) {  
            outboxRepo.save(new OutboxEvent(sql));  
        }  
        return super.onPrepareStatement(sql);  
    }  
}
```

2) Register it

In your config:

```
java
```

```
@Bean  
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds) {  
    LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();  
    em.setDataSource(ds);  
    em.setPackagesToScan("com.yourorg.entities");  
  
    Map<String, Object> props = new HashMap<>();  
    props.put("hibernate.ejb.interceptor", new SqlInterceptor());  
    em.setJpaPropertyMap(props);  
    return em;  
}
```

Unlike `StatementInspector`, this runs deeper in Hibernate and **sees the final SQL string used by JDBC prepared statement** — but still without inlined params unless you manually bind them.

🧠 Which One to Use First?

Approach	Actual SQL w/ params	Easy integration	Best First Step
P6Spy JDBC proxy	✓	🟡	⭐ Recommended
StatementInspector	⚠ placeholders	✓	Good for template capture

Approach	Actual SQL w/ params	Easy integration	Best First Step
Hibernate Interceptor	⚠ placeholders	🟡	Useful for ORM-level logic
<p>✓ P6Spy is the best first choice because it captures <i>real executed SQL with bound values</i> and requires the least Hibernate internal plumbing — perfect for replication outbox ingestion.</p>			

If you want, I can now show you exactly how to *stream those intercepted SQL statements directly into Kafka* as messages so your replication system can consume them — just let me know!

   Sources

Which approach do you prefer? Writing a base repository and writing json objects to outbox table or intercepting sqls and writing sqls to outbox table?

Both models you're considering are valid, but **one is almost always more pragmatic and reliable in real-world systems**. Here's a clear comparison and my recommendation based on typical replication and eventing goals:

Recommendation — Use a Base Repository + JSON Outbox First

⭐ Why this is usually the *better first choice*

✓ **Simplicity & clarity** — You explicitly record *what changed* in terms of business data (domain events), not noisy SQL syntax. This is easier for downstream consumers to interpret and act on.

✓ **Consistency with the pattern's intent** — The **Transactional Outbox Pattern** is designed to record domain events in the same transaction as your state change and then publish them later. This ensures **atomicity** — either both the state change and the event persist, or neither does.

✓ **Easier ordering & idempotency** — JSON payloads let you include metadata like entity ID, event type, timestamps, and version. Downstream consumers can more easily apply events in order and safely *replay* them without duplication.

Microsoft Learn

-  **Well-understood & supported** — Most tooling, frameworks, and best practices assume JSON or structured domain events in an outbox — not raw SQL. Libraries and platform patterns (e.g., **Debezium CDC**) read structured events much more reliably. devtechtools.org

Example insight

The pattern exists to avoid *dual-write* consistency issues (writing data and publishing events to Kafka in the same transaction), and writing structured JSON events directly into the outbox table is how most teams implement it. [Microsoft Learn](#)

When Would *SQL Statements* in Outbox Make Sense?

Capturing raw SQL statements might sound appealing, but it's usually *not* a first-order architectural choice unless you have specific needs:

Good reasons to choose SQL statements:

-  You need **exact replay** of DML on another database (e.g., for *physical replication*, not logical events).
-  Downstream systems expect *SQL APPLY semantics* rather than business events.

Why this is harder:

-  SQL doesn't inherently carry business meaning — replaying raw SQL often requires schema alignment and handling of bind variables.
 -  It ties your event system very tightly to your ORM/DML layer, which may change over time.
 -  Most outbox tooling (CDC, connectors) expects **events**, not SQL statements, meaning extra tooling work and complexity.
-

Which Pattern Maps to What Use Case

Approach	Best For	Downside
JSON domain events via base repository (recommended)	Application-level replication, microservices eventing, easy Kafka integration	Requires good event schema design
Intercept SQL & store raw statements	Physical replication or audit replay	Harder to maintain; raw SQL may vary by JPA/DB version

Practical Advice

Start with **JSON domain events**

- Write an Outbox entry containing structured event data whenever a business entity changes.
- Include fields like `eventType`, `aggregateId`, and a JSON payload of changed values.
- Use a CDC tool (e.g., Debezium) or a scheduled processor to push these events to Kafka. devtechtools.org

This gives you:

- Reliable transactional guarantees
- Clear, meaningful event data
- Easy routing/partitioning in Kafka
- Simplified replication semantics

Consider SQL interception only if:

- You explicitly need *replayable SQL*
- You have a legacy consumer that expects SQL text

Even then, it's often better to **derive the SQL you want from the structured events**, rather than capturing Hibernate/jdbc SQL directly.

🧠 Summary

👉 Preferred for most use cases:

→ *Base repository + JSON outbox events* — clean, structured, tool-friendly, and consistent with transactional outbox best practices. Medium

👉 Edge or specialized case:

→ *Capture SQL statements* — mostly for low-level replication or physical replay requirements.

If you want, I can help you design the **JSON event schema** and provide code examples showing how to *only include changed fields* in those events for efficient replication. Just let me know!

🔗  Sources

Can we use database wall logs for acquiring sql statements

Yes — you *can* use PostgreSQL's WAL (Write-Ahead Log) as a source of change data, but with some important caveats about *what* you get from WAL and how it differs from capturing SQL statements.

🧠 What PostgreSQL WAL *actually* contains

PostgreSQL's WAL is a **low-level log of physical or logical database changes**, designed for crash recovery and replication. It **does not store the SQL text that generated the changes** — it stores binary records that describe *which pages were modified and how* rather than the original `INSERT / UPDATE / DELETE` text.

newsletter.syste... +1

- WAL stores *physical changes* (e.g., "write 162 bytes to page X at offset Y"), not high-level SQL statements.
Stack Overflow
- You *cannot* read simple SQL text out of WAL in raw form; PostgreSQL doesn't log queries to WAL.
Stack Overflow
- Tools like `pg_waldump` can decode WAL for inspection, but they report low-level operations, not clean SQL statements.
Database Admin...

So if your goal is "get the SQL" — WAL by itself doesn't give this in a straightforward way.

💡 How WAL can be used effectively

While WAL doesn't contain SQL text, PostgreSQL **does support logical decoding** — a way to stream *logical change events* (high-level representations of data changes) using WAL as the source:

- ✓ **Logical decoding** allows you to extract a stream of operations like *inserts, updates, deletes* from the WAL.
- ✓ Plugins like `wal2json` or `pgoutput` decode WAL into JSON events representing row changes.
- ✓ This is exactly what CDC tools like **Debezium** use to get structured change events from PostgreSQL.

Google Cloud

This gives you a stream of events that *looks like change records*, not raw SQL text.

🔧 Practical options

✓ 1. Use logical replication / decoding

- Enable `wal_level = logical` in PostgreSQL.
- Create a *logical replication slot*.
- Use a decoding plugin (e.g., `wal2json`) that converts WAL to JSON change events (including table, row data, operation).
- Tools like **Debezium** can consume this stream and publish to Kafka.

This gives you **structured change events** you can feed into your replication mechanism — much closer to what you want than raw WAL bytes.

Google Cloud

📝 2. Use WAL only for physical replication

- PostgreSQL's WAL can be shipped and replayed to a standby database (binary replication).
- This keeps an exact copy of the database but *doesn't yield clean events* or SQL for external processing.

PostgreSQL

✗ Why WAL isn't ideal alone for SQL capture

People sometimes ask whether you can simply scan WAL and “get SQL statements” from it. The short answer: **not practically — WAL doesn’t store those statements in SQL text form**. Tools like `pg_waldump` show low-level storage changes, not replayable SQL.

A long-standing PostgreSQL community note explains that you *cannot* extract SQL from WAL — only logical changes via decoding.

Stack Overflow

💡 So how does this fit your replication needs?

Requirement	Suitable Approach
Capture actual SQL text	<i>Not directly from WAL</i> (use application-level logging or proxies like P6Spy)
Capture logical data changes	Use WAL + logical decoding (e.g., wal2json / Debezium)
Replicate to another DB	Use logical change events to apply updates downstream

🧠 Summary

✅ WAL is great for change-data capture when paired with **logical decoding** and a plugin that converts WAL into structured events.

❗ WAL does not contain the original SQL text — you cannot extract human-readable SQL directly from it.

Stack Overflow

💡 For replication, logical decoding + a CDC tool (like Debezium) is the recommended path if you want to avoid application-level outbox logic.

Google Cloud

If you want, I can show you how to set up **logical decoding in PostgreSQL + Debezium** so you can stream change events to Kafka — just let me know!

   Sources