1. What are Idempotent Producers in Kafka?

Answer:

Idempotent producers in Kafka guarantee that even if the same message is sent multiple times due to retries or failures, it's written **only once** to the topic.

It works by assigning a **Producer ID (PID)** and using **monotonically increasing sequence numbers** per partition. Kafka brokers track these to detect duplicates.

To enable it:

properties
CopyEdit

```
enable.idempotence=true
```

Trade-offs:

- Requires `acks=all`
- Limits `max.in.flight.requests.per.connection` to 5 or fewer to preserve order
- Only prevents duplicates **from the producer side**, not end-to-end

---

2. Explain Kafka Delivery Semantics

Answer:

Semantics	What It Means	Config
At-most-once	Messages may be lost but never duplicated	`acks=0`, no retries
At-least-once	Messages will be delivered, but might be duplicated	`acks=1 or all`, retries enabled, no idempotence
Exactly-once	Message is delivered **once** and **only once**	`acks=all` + `enable.idempotence=true` + Kafka transactions

Trade-offs:

- **At-most-once** = fastest, least reliable
- **At-least-once** = safe, but duplicates must be handled downstream
- **Exactly-once** = safest, but adds complexity and overhead

---

3. Kafka Transactions – How to Achieve Exactly-Once Processing

Answer:

Kafka transactions allow you to write to **multiple partitions/topics** and **commit consumer offsets** atomically.

Use this to ensure:

- Messages consumed from Topic A
- Processed
- Written to Topic B and offset committed

Either all these succeed or all are aborted.

Required Config:

properties
CopyEdit

```properties
enable.idempotence=true
transactional.id=your-producer-id
```

Producer Code Flow:

java
CopyEdit

```java
producer.initTransactions();
producer.beginTransaction();
producer.send(...);
producer.sendOffsetsToTransaction(offsets, consumerGroup);
producer.commitTransaction(); // or abortTransaction()
```

Trade-offs:

- Works only **Kafka-to-Kafka**
- Adds **latency and coordination complexity**
- You must handle **retries, fencing, and restarts** carefully

---

4. Producer Retries & Ordering

Answer:

Retries help prevent message loss, but if not configured correctly, they may **break ordering or cause duplicates**.

To maintain order **with retries**, make sure:

- `enable.idempotence=true`
- `max.in.flight.requests.per.connection <= 5` (or 1 for strict ordering)
- `acks=all`

These ensure no message reordering or duplication during retries.

Trade-off:
Higher reliability comes at the cost of **higher latency** and **reduced throughput** if in-flight requests are throttled.

---

5. Exactly-Once Kafka + External DB – Is It Possible?

Answer:

You can't achieve **true exactly-once semantics** across Kafka and an external DB without a 2-phase commit (which Kafka does not support).

So instead, you implement **at-least-once with idempotent logic**, like:

- Consume message
- Write to DB (idempotent insert)
- Only **after success**, commit offset

- On failure, don't commit offset → retry on restart

Outbox Pattern is a better alternative:

- App writes to DB + "outbox" table in the same local transaction
- A CDC tool (like Debezium) reads that outbox and produces to Kafka

Trade-off:
Adds design complexity but ensures **data consistency** between Kafka and DB.

---

6. Atomic Writes to Multiple Topics

Answer:

If you need to **produce messages to multiple Kafka topics atomically**, use Kafka Transactions.

With a transactional producer, you can:

- Send message to Topic A
- Send another to Topic B
- Either **commit** both or **abort** both

Benefit: Ensures **no partial writes** even if a crash occurs mid-way.

Trade-off: Adds a bit of latency and failure handling complexity.

---

7. Real-World Use Case – Exactly-Once Billing Event

Q: How would you guarantee exactly-once processing for billing events from Kafka to DB?

Answer:

In this case, I'd:

1. Use **idempotent DB writes** — e.g., based on event ID, skip duplicates
2. Consume from Kafka **without auto-commit**

3. After successful DB write, manually commit Kafka offset

4. Use a **deduplicating key** in the DB (like a `txn_id`) to avoid double billing

If I need Kafka-to-Kafka delivery (like billing audit topic), I'll use **Kafka transactions** to produce to multiple topics and commit offsets atomically.

Trade-offs:

- No real EOS with DB unless using Outbox/CDC
- Retry logic + deduplication must be solid to prevent real money loss