Performance tuning analysis

> Okay multiple threads will never update the same row each row is an individual update and data from one row is not depend on data from another row threads or process updating these rows will never compete for same row since every row represent distinct trade record

Thanks for the clarification — that reduces risk of row-level lock contention significantly. Given your reconciliation pipeline uses:

- **10 JVM processes**
- Each running **10 threads**
- Using `JdbcTemplate.batchUpdate()`
- Targeting **independent rows**
- With a WHERE clause on `trade_id` + `trade_date` (both indexed)

Let's assess **potential DB2 contention risks** and throughput improvement strategies under these constraints.

✅ Lock Contention Risk (Minimal)

You're safe **row-lock wise** because:

- No two threads update the same row.
- There are no dependencies between rows.
- Indexes on `trade_id` and `trade_date` make lookups efficient.

However, **page-level or table-level locking** in DB2 can still slow you down under pressure. Especially:

🚨 Possible Sources of Contention:

1. **Table-level or page-level locks** (esp. in row-organized tables):
   - If many rows fall into the same DB2 page/block due to ordering or clustering, page locks may serialize writes.
2. **Index contention**:
   - Even with separate rows, if multiple updates affect the same index pages (like `status`, `match_id`), writes to index B-Trees can serialize.
3. **Log writer bottlenecks**:

- All 10x10 threads pushing updates = 100 parallel updates → DB2's transaction log can become a shared bottleneck.

4. **Lock escalation**:
   - If DB2 starts acquiring too many row locks, it may escalate to table locks unless properly tuned (`LOCKLIST`, `MAXLOCKS`).
5. **Autocommit or oversized transaction batches**:
   - `batchUpdate()` groups updates, but uncommitted rows can block others if transaction boundaries aren't well tuned.

🧠 Recommendations

1. Switch to Insert-Only `recon_status` for Primary Path

Even if updates are still needed for rare events (cancels, corrections), make inserts the default path for high-volume flows:

- **Inserts avoid row locking, page locking (mostly), and index write-back pressure.**
- Better for **horizontal scaling** (you can log trade changes in parallel across partitions or sharded tables).

2. Use Updates Sparingly (for Edge Cases Only)

For `cancel` and `amend` types:

- Update via a secondary thread pool.
- Or log as a new row (immutable event log style) and **resolve the latest state at query time** or in downstream rollup.

3. Optimize `batchUpdate()`

- Batch 5000 is decent, but test with 1000–2000 size for better commit granularity (trade-off: throughput vs rollback risk).
- Explicitly wrap batch updates in transactions:

```java
transactionTemplate.execute(status -> {
  jdbcTemplate.batchUpdate(...);
  return null;
});
```

4. Avoid Hot Index Pages

- Randomize insert/update order to avoid clustering.
- Use a secondary index structure (like UUID or partitioned ID) to distribute writes.

5. Parallelize Across Table Partitions (if not already)

- If not yet, parallelize based on `trade_date` or broker code into multiple tables or partitions.
- Each JVM/thread can then target isolated DB2 partitions = **zero interference**.

🔬 Benchmark: Insert vs Update

> ❓**Which is faster: insert (11 columns) vs update (2 columns)?**

🚀 Insert is usually faster *if*:

- There are fewer indexes to update.
- Logging and page allocation aren't bottlenecks.
- No unique key conflicts.

⚙️ Update may be faster *if*:

- You're updating 1–2 columns.
- Indexes are stable.
- Row is already cached in buffer pool.

🔬 **BUT**: With your workload of **10M+ parallel operations**, inserts tend to scale better due to less locking + write-ahead logging overhead.

🔄 Option: Hybrid Table Design

If insert-only isn't viable for all cases:

Consider:

```sql
CREATE TABLE recon_status (
  trade_id VARCHAR(50),
  trade_date DATE,
  op_type CHAR(1),        -- 'I' = insert, 'C' = cancel, 'A' = amend
  match_id VARCHAR(50),
```

```
  status VARCHAR(20),
  -- other 6-7 fields
  updated_ts TIMESTAMP,
  PRIMARY KEY (trade_id, trade_date, op_type)
)
```

- Log multiple versions per trade with op_type.
- Your reconciliation view selects latest entry per trade.
- Enables **event sourcing-style processing** without hard update.

🧾 Summary:

- ✅ You're safe from row lock contention — great setup.
- ⚠️ May still hit page locks, log writer bottlenecks, or index contention.
- 📈 Inserts scale better under concurrency — consider `recon_status` insert-only as primary path.
- 🧪 Tune batch size, commit frequency, and index design.
- 🛠️ Hybrid model (insert for most, update for rare) is valid, but control update flow strictly.

Would you like:

- Sample schema + query plan for `recon_status`?
- Benchmarks for insert vs update performance using sample data?
- Lock monitoring tips (`MON_LOCKWAITS`, `SNAPLOCK`) in DB2?

Let me know how deep you want to go next.