Adding a New Column to a 100-Million-Row Oracle Table

Production-safe approaches • Oracle 19c • Performance tuning

A video-ready technical guide with demo scripts, decision points, operational risks, validation, and interview questions.

1. Introduction

Imagine an Oracle table contains 100 million rows and a new business requirement asks us to add a column. The first question is not simply which ALTER TABLE statement to run. We must establish whether the column can remain NULL, needs a constant default, must be populated from existing data, or must be NOT NULL. Those requirements have very different performance and availability implications.

The objective is to minimize unnecessary table rewrites, long-running transactions, redo and undo generation, blocking, archive-log growth, and impact on application workloads.

2. Learning objectives

- Choose an appropriate column-add strategy for a very large table.
- Understand metadata-only DDL versus physically updating every row.
- Plan a controlled, restartable backfill when values must be calculated.
- Monitor locking, redo, undo, archive logs, execution time, and Data Guard lag.
- Validate data and constraints before completing the change.

3. Decision matrix

Requirement	Starting approach	Key consideration
Nullable column only	ALTER TABLE ADD column	Usually metadata-oriented; DDL still needs a lock.
Constant default	ALTER TABLE ADD column DEFAULT value	Oracle can optimize eligible constant defaults; verify datatype/version/table conditions.
Value derived for each row	Add nullable column, then controlled backfill	The backfill physically changes rows and generates redo/undo.
NOT NULL column	Plan default/backfill/validation in phases	Existing rows must satisfy the constraint; online application compatibility matters.
Strict availability	Expand-and-migrate rollout	Coordinate old/new application versions and concurrent writes.

4. Scenario A — Add a nullable column

Use this when the new field can be NULL for existing records and the application can populate it later.

```
ALTER TABLE EMPLOYEE ADD (
    NEW_COLUMN VARCHAR2(100)
);
```

For a conventional Oracle heap table, adding a nullable column without a default is generally a metadata operation rather than a rewrite of all 100 million rows. However, Oracle must obtain the required DDL lock. The statement can wait behind active transactions, so schedule and test it appropriately.

5. Scenario B — Add a constant default

```
ALTER TABLE EMPLOYEE ADD (
    STATUS VARCHAR2(10) DEFAULT 'ACTIVE'
);
```

Oracle 11g and later introduced optimizations that allow eligible columns with constant defaults to be added without physically updating every existing row. Oracle can represent the default through metadata for rows that pre-date the column. This is not equivalent to executing UPDATE against every row.

Important: behavior depends on the Oracle release, datatype, default expression, table organization, and other features. Validate the exact DDL in a representative non-production environment. A volatile or row-dependent expression is not the same as a constant literal default.

6. Scenario C — Every existing row needs a calculated value

When the value must be derived from other columns or business rules, adding the column is only the first step. The backfill is the expensive operation because it modifies rows. Avoid one uncontrolled transaction over all 100 million rows unless the impact and rollback requirements have been explicitly assessed.

```
ALTER TABLE EMPLOYEE ADD (
    NEW_COLUMN NUMBER
);
```

Example business rule for demonstration only: set NEW_COLUMN equal to EMP_ID. Replace it with the actual rule.

```
UPDATE EMPLOYEE
SET NEW_COLUMN = EMP_ID
WHERE NEW_COLUMN IS NULL;

COMMIT;
```

The statement above illustrates the rule, but it is not the recommended production execution pattern for 100 million rows: it may scan and modify a very large portion of the table in one transaction, generating substantial redo and undo.

7. Controlled batch backfill pattern

Prefer deterministic batches using an indexed, stable key (for example, a numeric primary key) or a prepared worklist. The following is a template for a table with a numeric EMP_ID primary key. Choose batch boundaries based on the actual key distribution and workload. It deliberately uses key ranges rather than repeatedly selecting arbitrary rows.

```
-- Example only: determine actual key bounds first.
SELECT MIN(EMP_ID), MAX(EMP_ID), COUNT(*)
FROM EMPLOYEE;

-- Run one bounded batch at a time.
UPDATE EMPLOYEE
SET NEW_COLUMN = EMP_ID
WHERE EMP_ID >= :low_key
  AND EMP_ID <  :high_key
  AND NEW_COLUMN IS NULL;

COMMIT;
```

A controller script or scheduler should advance :low_key and :high_key through non-overlapping ranges. Record each completed range in a control table so the process can restart safely. If keys are sparse, use measured batch sizes or a worklist. Do not assume a fixed key range contains a fixed number of rows.

Tune batch size from measured elapsed time, redo/undo pressure, blocking, and application latency. Small batches reduce the duration of each transaction but can increase commit overhead; large batches can create long transactions and increase rollback and resource pressure.

8. Restartability and concurrent application writes

- Use an explicit migration/control table to track completed key ranges, start time, end time, row count, and status.
- Use an idempotent predicate such as NEW_COLUMN IS NULL only when NULL genuinely means 'not yet processed'.
- Define how inserts and updates that happen during the backfill will populate the new column.
- Deploy application changes in a compatible order: add the column first, update writers, backfill historical rows, validate, then enforce constraints if needed.
- If business rules can change while the job runs, capture a consistent rule/version or design reconciliation logic.

9. Monitoring before, during, and after

Before the change

- Confirm table size, partitioning, indexes, constraints, triggers, replication, and application dependencies.
- Check free space, undo capacity/retention, archive-log destination capacity, and standby apply performance where applicable.
- Review active transactions and schedule DDL to reduce lock waits.
- Capture baseline application latency and database resource metrics.

During the change

- Track elapsed time and rows processed per batch.
- Monitor blocking sessions, CPU, I/O, redo generation, undo usage, archive-log growth, and tablespace capacity.
- If Data Guard is used, monitor transport/apply lag and ensure the change does not create an unacceptable backlog.
- Pause or reduce batch rate if the application experiences material degradation.

Useful SQL checks

```
-- Table and segment metadata
SELECT owner, table_name, num_rows, last_analyzed
FROM all_tables
WHERE owner = UPPER(:owner)
  AND table_name = 'EMPLOYEE';

-- Segment size (requires appropriate dictionary access)
SELECT owner, segment_name, segment_type,
       ROUND(bytes / 1024 / 1024 / 1024, 2) AS size_gb
FROM dba_segments
WHERE owner = UPPER(:owner)
  AND segment_name = 'EMPLOYEE';
```

```
-- Active sessions (privileges required)
SELECT sid, serial#, username, status, event,
       blocking_session, sql_id
FROM v$session
WHERE username IS NOT NULL
  AND status = 'ACTIVE';
```

Dictionary statistics can be stale or approximate; NUM_ROWS is not guaranteed to be an exact live row count. Use only views available to the account and follow local security policy.

10. Validation after backfill

```
-- Count remaining rows that have not been populated
SELECT COUNT(*) AS rows_missing_value
FROM EMPLOYEE
WHERE NEW_COLUMN IS NULL;

-- Validate the derivation rule on a sample or full scan if appropriate
SELECT EMP_ID, NEW_COLUMN
FROM EMPLOYEE
WHERE NEW_COLUMN <> EMP_ID
  AND ROWNUM <= 100;

-- Check the column definition
SELECT column_name, data_type, nullable, data_default
FROM user_tab_columns
WHERE table_name = 'EMPLOYEE'
  AND column_name = 'NEW_COLUMN';
```

If NULL is a valid business value, a NULL count alone cannot prove completeness. Track the expected population set and reconcile processed rows against that set. A full-table validation query may itself be expensive, so plan it.

11. Adding NOT NULL or an index later

Do not combine all operations into one rushed deployment. First establish the data and application behavior, then validate constraints and indexes as separate steps.

```
-- Only after every applicable row has a valid value:
ALTER TABLE EMPLOYEE MODIFY (
    NEW_COLUMN NOT NULL
);

-- Example index, only if workload and access patterns justify it:
CREATE INDEX EMPLOYEE_NEWCOL_IX
    ON EMPLOYEE (NEW_COLUMN);
```

Constraint validation and index creation can consume significant I/O, CPU, space, and time. Indexing a low-selectivity column may not help queries. Evaluate online index options, licensing/edition restrictions, and operational policy before use.

12. Expand-and-migrate approach for strict uptime

1. Expand: add the new nullable column without breaking the current application.

2. Deploy compatible writers so new and changed rows receive the new value.
3. Backfill historical data in restartable batches.
4. Reconcile rows changed during the backfill and validate business rules.
5. Add constraints/indexes if justified and supported by the rollout plan.
6. Contract: remove compatibility code only after all application instances and consumers use the new model.

13. Common mistakes to avoid

- Assuming every ALTER TABLE operation is instant or never needs a lock.
- Using one giant UPDATE without measuring redo, undo, archive logs, rollback time, and application impact.
- Running a batch query that rescans the whole table on every iteration.
- Committing per row rather than per controlled batch.
- Creating an index without confirming query patterns and selectivity.
- Enforcing NOT NULL before validating all relevant rows and concurrent-write behavior.
- Ignoring standby lag, backup windows, downstream replication, and operational rollback/recovery plans.

14. Demo walkthrough for a video

7. Introduce the 100-million-row problem and explain why the business requirement matters.
8. Compare nullable-column DDL with a constant-default DDL.
9. Explain metadata-only optimization and its conditions.
10. Demonstrate why a derived-value backfill is a separate, data-changing operation.
11. Show a key-range batch, commit strategy, restartability, and monitoring.
12. Finish with validation, constraints/indexes, and production rollout recommendations.

15. Interview questions and answers

Q1. Does adding a nullable column rewrite 100 million rows?

Usually not for a conventional table with a simple nullable column and no default; it is generally metadata-oriented. DDL locking and table-specific conditions still matter.

Q2. Why can adding a constant default be fast?

Oracle can use metadata-based optimizations for eligible constant defaults instead of physically updating all pre-existing rows.

Q3. Why is updating a derived column expensive?

It changes table blocks and generates redo and undo, can cause I/O and archive-log growth, and may create long-running transactions.

Q4. Why use batches?

They bound transaction size, make progress observable, reduce the scope of rollback, and allow the job to be paused or restarted.

Q5. Should every batch commit?

Commit at a controlled batch boundary, not per row. Batch size must balance resource pressure and commit overhead.

Q6. Is parallel DML always faster?

No. It can increase CPU, I/O, redo, contention, and impact on other workloads. Test it and confirm the operational constraints.

Q7. Can we add NOT NULL immediately?

Only if the existing-row semantics and Oracle's constraint/default behavior make it valid. For derived values, populate and validate first.

Q8. What is the safest approach for a zero/low-downtime deployment?

Use an expand-and-migrate rollout, keep old and new application versions compatible, backfill in controlled batches, reconcile concurrent changes, then enforce the final model.

16. Final summary

For a 100-million-row Oracle table, the best method is driven by the new column's semantics. If a nullable column is enough, a simple ALTER TABLE ADD is usually the lowest-impact choice. If a constant default is required, use Oracle's eligible default-column optimization after testing. If every row needs a calculated value, separate the metadata change from a controlled, restartable backfill. Production success depends on locking strategy, workload monitoring, concurrent-write handling, and validation—not just SQL syntax.

Suggested closing line: “Before changing a 100-million-row table, identify whether the task is metadata-only or a real data migration. That decision drives the performance strategy.”

17. Production checklist

- ☐ Requirement and null/default semantics confirmed
- ☐ Oracle version and table configuration verified
- ☐ DDL lock timing tested
- ☐ Application compatibility and concurrent writes planned
- ☐ Undo, redo, archive-log, storage, and standby capacity checked
- ☐ Batch boundaries and restart control prepared
- ☐ Monitoring thresholds and pause criteria agreed
- ☐ Validation and reconciliation queries reviewed
- ☐ Constraint/index operations scheduled separately
- ☐ Recovery and rollback strategy approved