SQL DML Cheat Sheet: SQLite | PostgreSQL | MySQL | SQL Server

1. INSERT

DB	Basic Syntax	Multi-row / SELECT Variant	Upsert / Conflict Handling	Notes & Gotchas
SQLite	`INSERT INTO table (col1, col2, ...) VALUES (v1, v2, ...);`	`INSERT INTO table (cols...) SELECT ... FROM other_table;`	`INSERT OR REPLACE INTO ...` or since SQLite 3.24+: `INSERT … ON CONFLICT … DO UPDATE`	`REPLACE` deletes then reinserts (may reset `ROWID`, trigger cascades). Be careful with side-effects and foreign keys.
PostgreSQL	`INSERT INTO table (cols...) VALUES (...);`	`INSERT INTO table (cols...) SELECT ...;`	`INSERT … ON CONFLICT (unique_col) DO UPDATE …` or `DO NOTHING`	Powerful `RETURNING` clause to fetch inserted rows.
MySQL / MariaDB	`INSERT INTO table (cols...) VALUES (...);`	`INSERT INTO table (cols...) SELECT ...;`	`INSERT … ON DUPLICATE KEY UPDATE …` or `REPLACE INTO …`	`ON DUPLICATE KEY` works for unique/PK conflicts. `REPLACE` is delete + insert (watch foreign keys).
SQL Server (T-SQL)	`INSERT INTO table (cols...) VALUES (...);`	`INSERT INTO table (cols...) SELECT ...;`	Use `MERGE` (WHEN MATCHED / WHEN NOT MATCHED) for “upsert” logic	Use `OUTPUT` to return inserted rows. Be aware of known quirks/bugs in `MERGE`.

Tips for INSERT in general: - Always specify the column list explicitly; don’t rely on implicit ordering. - Omitting columns means you must supply *all* columns in table order. - For large inserts, use batched inserts to avoid memory/log issues. - Be mindful of default values, computed columns, identity/autoincrement semantics. - In migrations, test differences in defaults, nullability, unique constraints, trigger behavior across DBs.

---

2. UPDATE

DB	Basic Syntax	Update with JOINS / References to other tables	Returning / Output	Notes & Gotchas
SQLite	`UPDATE table SET col1 = expr1, col2 = expr2 WHERE condition;`	Doesn't support `UPDATE … JOIN` syntax; use correlated subqueries or `WHERE EXISTS`	Newer SQLite versions support `RETURNING` (3.35)	Omitting `WHERE` updates all rows. Be cautious when referencing same-table subqueries.
PostgreSQL	`UPDATE table SET col1 = expr1, col2 = expr2 WHERE condition;`	Supports `UPDATE … FROM other_table … WHERE …` join syntax	Supports `RETURNING *` for post-update results	Under MVCC, update = create new tuple + mark old as dead. Watch out for rule system rewrites.
MySQL / MariaDB	`UPDATE table SET col1 = expr1, col2 = expr2 WHERE condition;`	Supports `UPDATE t1 JOIN t2 ON … SET t1.col = t2.col2 WHERE …`	No built-in `RETURNING` in older versions; limited newer support	Be cautious with ambiguous joins, order of evaluation, and missing WHERE clauses.

DB	Basic Syntax	Update with JOINS / References to other tables	Returning / Output	Notes & Gotchas
SQL Server (T-SQL)	`UPDATE t SET col1 = expr1, col2 = expr2 FROM table t JOIN other_table o ON … WHERE …`	Yes: `UPDATE … FROM` join syntax is standard in T-SQL	Use `OUTPUT inserted.*, deleted.*`	`OUTPUT` is powerful but has restrictions (e.g. can't always be used with views). Be careful about cascading updates, triggers, and identity columns.

General Update Gotchas: - Always include `WHERE` (or join filter) to avoid updating all rows. - Be aware of side-effects via triggers or cascading foreign key `ON UPDATE`. - In expressions like `SET col = col + 1`, understand how DB snapshots values (most DBs compute RHS from original value, not sequentially). - Null semantics: `col = NULL` is never true; use `IS NULL`. - Concurrent updates: consider locking, isolation levels, "lost updates". - Updating unique or primary key columns can break constraints.

---

3. DELETE

DB	Basic Syntax	Delete with JOINS / Multi-table support	Returning / Output	Notes & Gotchas
SQLite	`DELETE FROM table WHERE condition;`	Doesn't support `DELETE … JOIN` syntax; use `WHERE EXISTS` or subqueries	Newer SQLite versions support `RETURNING` (3.35)	Omitting `WHERE` deletes all rows. Foreign key cascades may delete related rows.

DB	Basic Syntax	Delete with JOINS / Multi-table support	Returning / Output	Notes & Gotchas
PostgreSQL	`DELETE FROM table WHERE condition;`	Supports `DELETE … USING other_table` syntax: `DELETE FROM t1 USING t2 WHERE t1.col = t2.col AND …`	`RETURNING *` allowed	On large tables, many dead tuples may accumulate; VACUUM may be needed.
MySQL / MariaDB	`DELETE FROM table WHERE condition;`	Supports `DELETE t1 FROM t1 JOIN t2 ON … WHERE …`	No built-in return before newer versions	Deleting many rows in one transaction can bloat logs; use chunked deletes.
SQL Server (T-SQL)	`DELETE FROM table WHERE condition;`	Supports `DELETE t1 FROM t1 JOIN t2 ON … WHERE …`	Use `OUTPUT deleted.*`	Watch cascading deletes, triggers, lock escalation. `TRUNCATE TABLE` is different (no row-by-row, faster, bypasses triggers, cannot be rolled back in some contexts).

General Delete Gotchas: - Always use `WHERE` unless deleting *all* rows intentionally. - Cascading deletes from foreign keys may trigger large deletion chains. - Deleting a large number of rows in one go may hit transaction log limits; break into smaller batches. - In MVCC engines, delete leaves “dead” row versions/tuples; cleanup (vacuum / compaction) is necessary. - Use `RETURNING` / `OUTPUT` to see exactly what rows were deleted, but consider performance and side effect costs.

---

4. Summary & Best Practices

- Basic DML (INSERT / UPDATE / DELETE) is broadly portable, but **extensions** (joins in updates, returning/outputs, upsert/merge) vary significantly across engines.
- To write **portable** SQL, stick close to the common subset: `INSERT` with column lists, `UPDATE` & `DELETE` with simple `WHERE`, avoid engine-specific features unless wrapped behind abstraction.
- Use `RETURNING` / `OUTPUT` to avoid extra round trips when supported — but guard such usage behind dialect checks.
- Be cautious with `MERGE` / `UPSERT` / `REPLACE` statements — they often carry subtle behavior (side effects, ordering, constraint resolution).
- Before executing destructive or bulk operations, always test your `WHERE` filters via `SELECT`.
- For heavy workloads, batch updates/deletes, monitor transaction log usage, handles locking contention, and manage version cleanup (in MVCC systems).
- Be aware of hidden effects: triggers, cascades, implicit commits, side-effects. Always understand the full chain of dependencies.
- Treat DML in migrations with care. Build a comprehensive test suite to validate behavior across DB platforms, nulls, constraint violations, edge boundary cases.
