Soft Deletes vs Hard Deletes

1. The Real Problem This Exists to Solve

When users delete data, you face a choice: permanently remove it (hard delete) or mark it as deleted while retaining the record (soft delete). This decision affects data recovery, audit trails, referential integrity, legal compliance, and system complexity.

Real production scenario:

- SaaS application with user-generated content
- User accidentally deletes important project
- **With hard delete:**
  - Project row removed from database
  - All foreign key relationships cascade delete
  - Related tasks, comments, files deleted
  - No recovery possible
  - User contacts support: "I need that project back!"
  - Support: "Sorry, it's permanently deleted"
  - User churns, writes angry review
- **With soft delete:**
  - Project marked as deleted_at = NOW()
  - Still in database, just hidden from queries
  - Foreign key relationships intact
  - User contacts support within 30 days
  - Support runs: UPDATE projects SET deleted_at = NULL WHERE id = X
  - Project restored with all related data
  - User happy, writes positive review

The fundamental problem: Deletion is often irreversible, but user intent isn't always final. Mistakes happen (fat-finger deletes, accidental bulk deletes, misunderstood UI). Hard deletes are instant and permanent. Soft deletes provide a safety net while introducing query complexity.

Without soft deletes (hard delete only):

- No undo/recovery mechanism
- Lost audit trail
- Compliance issues (GDPR right to access)
- Foreign key cascade complications
- Data loss from accidents

With soft deletes:

- Recoverable deletions (30-day grace period)
- Complete audit trail
- Referential integrity preserved
- Complexity in every query (WHERE deleted_at IS NULL)
- Storage overhead from "deleted" records

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Soft Delete Without Filtering Queries

```
// Incorrect: Add deleted_at but forget to filter it
app.get('/api/projects', async (req, res) => {
  // Missing: WHERE deleted_at IS NULL
  const projects = await db.query(
    'SELECT * FROM projects WHERE user_id = $1',
    [req.user.id]
  );

  res.json(projects.rows);
});

app.post('/api/projects/:id/delete', async (req, res) => {
  await db.query(
    'UPDATE projects SET deleted_at = NOW() WHERE id = $1',
    [req.params.id]
  );

  res.json({ success: true });
});
```

Why it seems reasonable:

- Added deleted_at column
- Soft delete updates the timestamp
- Data not actually removed

How it breaks:

```
// User deletes project
POST /api/projects/123/delete

// User lists projects
GET /api/projects
// Returns: [
//   { id: 123, name: "Deleted Project", deleted_at: "2024-01-15" },
//   { id: 124, name: "Active Project", deleted_at: null }
// ]

// "Deleted" project still appears in list!
```

Production symptoms:

- Deleted items still show in UI
- Counts include deleted records
- Deleted projects appear in search
- User confusion: "I deleted this, why is it still here?"
- Every single query needs manual filtering

❌ Incorrect Approach #2: Soft Delete Breaks Foreign Keys

```
// Incorrect: Foreign key references soft-deleted parent
// Schema:
// projects (id, name, deleted_at)
// tasks (id, project_id REFERENCES projects(id), name)

// Delete project (soft delete)
await db.query(
  'UPDATE projects SET deleted_at = NOW() WHERE id = $1',
  [projectId]
);

// Try to create task for deleted project
await db.query(
  'INSERT INTO tasks (project_id, name) VALUES ($1, $2)',
  [projectId, 'New Task']
);
// Success! Task created for "deleted" project

// Query tasks for active projects
const tasks = await db.query(
  `SELECT tasks.*
   FROM tasks
   JOIN projects ON tasks.project_id = projects.id
   WHERE projects.deleted_at IS NULL`
);
// Task orphaned: project_id points to deleted project
```

Why it seems reasonable:

- Foreign keys still valid (project exists)
- Referential integrity maintained
- Can restore project with all tasks

How it breaks:

- Can insert new records referencing "deleted" parents
- Child records orphaned when parent soft-deleted
- Cascade delete doesn't work with soft deletes
- Inconsistent state: active child, deleted parent

Production symptoms:

```
-- Orphaned tasks
SELECT COUNT(*) FROM tasks
WHERE project_id IN (
  SELECT id FROM projects WHERE deleted_at IS NOT NULL
);
-- Result: 1,247 orphaned tasks

-- User creates task for "deleted" project via cached project_id
-- Task appears in admin panel but not in UI (parent filtered out)
```

❌ Incorrect Approach #3: Unique Constraints Don't Work

```
// Incorrect: Unique constraint on soft-deleted records
// Schema:
// users (id, email UNIQUE, deleted_at)

// User: bob@example.com signs up
await db.query(
  'INSERT INTO users (email) VALUES ($1)',
  ['bob@example.com']
);

// Bob deletes account (soft delete)
await db.query(
  'UPDATE users SET deleted_at = NOW() WHERE email = $1',
  ['bob@example.com']
);

// Bob tries to sign up again with same email
await db.query(
  'INSERT INTO users (email) VALUES ($1)',
  ['bob@example.com']
);
// ERROR: duplicate key value violates unique constraint "users_email_key"

// Can't reuse email because soft-deleted record still has it!
```

Why it seems reasonable:

- Email must be unique for active users
- UNIQUE constraint enforces this
- Soft-deleted users kept for audit trail

How it breaks:

- Unique constraints apply to ALL rows (including soft-deleted)
- User can't reuse their email after deletion
- Can't have two soft-deleted users with same email
- Workarounds are hacky (append timestamp to deleted emails)

Production symptoms:

```
User: "I deleted my account and want to recreate it"
System: "Email already exists"
User: "But I deleted it!"
Support: "Your old account is still in our database..."

Database:
users table:
  id=1, email=bob@example.com, deleted_at=2024-01-01
  id=2, email=bob@example.com, deleted_at=NULL  ← Can't insert!
```

❌ Incorrect Approach #4: Inconsistent Soft Delete Handling

```
// Incorrect: Some models soft delete, others hard delete (inconsistent)
class UserService {
  async deleteUser(userId: number) {
    // Soft delete users
    await db.query(
      'UPDATE users SET deleted_at = NOW() WHERE id = $1',
      [userId]
    );
  }
}

class PostService {
  async deletePost(postId: number) {
    // Hard delete posts
    await db.query('DELETE FROM posts WHERE id = $1', [postId]);
  }
}

// User deletes account (soft delete)
await userService.deleteUser(123);

// Later, query user's posts
const posts = await db.query(
  'SELECT * FROM posts WHERE user_id = $1',
  [123]
);
// Returns posts with user_id = 123

// But user is soft-deleted!
// Inconsistency: deleted user has active posts
```

Why it seems reasonable:

- Different models have different requirements
- Users need soft delete for compliance
- Posts can be hard deleted

How it breaks:

- Inconsistent data model
- JOIN queries break (user soft-deleted, posts hard-deleted)
- Referential integrity issues
- Audit trail incomplete

Production symptoms:

```
-- Active posts for deleted users
SELECT COUNT(*) FROM posts
WHERE user_id IN (
  SELECT id FROM users WHERE deleted_at IS NOT NULL
);

-- Confusion: "Did we delete this user's data or not?"
-- GDPR compliance risk
```

❌ Incorrect Approach #5: No Cascade Soft Delete

```
// Incorrect: Delete parent but not children
await db.query(
  'UPDATE projects SET deleted_at = NOW() WHERE id = $1',
  [projectId]
);

// Tasks still active (not soft-deleted)
const tasks = await db.query(
  'SELECT * FROM tasks WHERE deleted_at IS NULL'
);
// Returns tasks for deleted project!
```

Why it seems reasonable:

- Deleted parent, that's enough
- Children queries will join and filter parent

How it breaks:

- Orphaned children still appear in searches
- Counts wrong (include deleted parent's children)
- Restore parent doesn't restore children
- Inconsistent state

3. Correct Mental Model (How It Actually Works)

Soft deletes mark records as deleted using a timestamp column. All queries must filter `WHERE deleted_at IS NULL`. Unique constraints require partial indexes. Cascade deletes require application-level logic.

The Soft Delete Pattern

```
-- Add deleted_at column
ALTER TABLE projects ADD COLUMN deleted_at TIMESTAMP DEFAULT NULL;

-- Soft delete
UPDATE projects SET deleted_at = NOW() WHERE id = 123;

-- Query active records
SELECT * FROM projects WHERE deleted_at IS NULL;

-- Restore
UPDATE projects SET deleted_at = NULL WHERE id = 123;
```

```
-- Permanent cleanup (later)
DELETE FROM projects WHERE deleted_at < NOW() - INTERVAL '30 days';
```

Query Filtering Strategies

Option 1: Manual filtering (every query)

```
SELECT * FROM projects WHERE deleted_at IS NULL;
```

Option 2: Database view

```
CREATE VIEW active_projects AS
SELECT * FROM projects WHERE deleted_at IS NULL;

SELECT * FROM active_projects;
```

Option 3: ORM global scope (Eloquent, TypeORM)

```
@Entity()
class Project {
  @DeleteDateColumn()
  deletedAt?: Date;
}

// Auto-filters deleted_at IS NULL
await projectRepo.find();

// Include soft-deleted
await projectRepo.find({ withDeleted: true });
```

Unique Constraints with Soft Deletes

```
-- Partial unique index (only active records)
CREATE UNIQUE INDEX users_email_unique
ON users (email)
WHERE deleted_at IS NULL;

-- Allows:
-- id=1, email=bob@example.com, deleted_at=NULL (active)
-- id=2, email=bob@example.com, deleted_at=2024-01-01 (deleted)
-- id=3, email=alice@example.com, deleted_at=NULL (active)
```

Cascade Soft Deletes

```
async function softDeleteProject(projectId: number) {
  await db.query('BEGIN');

  // Soft delete parent
  await db.query(
    'UPDATE projects SET deleted_at = NOW() WHERE id = $1',
    [projectId]
  );

  // Soft delete children
  await db.query(
    'UPDATE tasks SET deleted_at = NOW() WHERE project_id = $1',
    [projectId]
  );

  await db.query('COMMIT');
}
```

4. Correct Design & Algorithm

Strategy 1: Soft Delete with ORM Global Scope

```
// TypeORM entity
@Entity()
class Project {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @DeleteDateColumn()
  deletedAt?: Date;

  @OneToMany(() => Task, task => task.project)
  tasks: Task[];
}

// Auto-filters soft-deleted
const projects = await projectRepo.find();

// Include soft-deleted
const allProjects = await projectRepo.find({ withDeleted: true });

// Only soft-deleted
const deletedProjects = await projectRepo.find({
  withDeleted: true,
  where: { deletedAt: Not(IsNull()) }
});

// Soft delete
await projectRepo.softRemove(project);

// Hard delete
```

```
await projectRepo.remove(project);

// Restore
await projectRepo.recover(project);
```

Strategy 2: Hybrid (Soft Delete + Hard Delete After TTL)

```
-- Soft delete (immediate)
UPDATE projects SET deleted_at = NOW() WHERE id = ?;

-- Hard delete (cleanup job, runs daily)
DELETE FROM projects
WHERE deleted_at < NOW() - INTERVAL '30 days';
```

Strategy 3: Archive Table (Separate Storage)

```
-- Move to archive table
INSERT INTO projects_archive SELECT * FROM projects WHERE id = ?;
DELETE FROM projects WHERE id = ?;

-- Query includes archive
SELECT * FROM projects
UNION ALL
SELECT * FROM projects_archive;
```

5. Full Production-Grade Implementation

```
import { Entity, PrimaryGeneratedColumn, Column, DeleteDateColumn, Repository, Not,
IsNull } from 'typeorm';

/**
 * Project entity with soft delete
 */
@Entity('projects')
class Project {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  userId: number;

  @DeleteDateColumn()
  deletedAt?: Date;

  @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
  createdAt: Date;
}

/**
 * Task entity with soft delete
 */
@Entity('tasks')
class Task {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  title: string;

  @Column()
  projectId: number;

  @DeleteDateColumn()
  deletedAt?: Date;
}

/**
 * User entity with soft delete and unique email
 */
@Entity('users')
class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  email: string;

  @Column()
  name: string;

  @DeleteDateColumn()
  deletedAt?: Date;
}

/**
 * Service with cascade soft delete
 */
class ProjectService {
  constructor(
    private projectRepo: Repository<Project>,
    private taskRepo: Repository<Task>
  ) {}

  /**
   * Soft delete project and cascade to tasks
   */
  async deleteProject(projectId: number): Promise<void> {
    const project = await this.projectRepo.findOneBy({ id: projectId });

    if (!project) {
      throw new Error(`Project ${projectId} not found`);
    }

    // Soft delete project
    await this.projectRepo.softRemove(project);

    // Cascade soft delete to tasks
    const tasks = await this.taskRepo.find({
      where: { projectId },
      withDeleted: true, // Include already soft-deleted tasks
    });

    if (tasks.length > 0) {
      await this.taskRepo.softRemove(tasks);
    }
  }

  /**
   * Restore project and cascade to tasks
   */
  async restoreProject(projectId: number): Promise<void> {
    const project = await this.projectRepo.findOne({
      where: { id: projectId },
      withDeleted: true,
    });

    if (!project || !project.deletedAt) {
      throw new Error(`Soft-deleted project ${projectId} not found`);
    }

    // Restore project
    await this.projectRepo.recover(project);

    // Restore tasks
    const tasks = await this.taskRepo.find({
      where: { projectId },
      withDeleted: true,
    });

    if (tasks.length > 0) {
      await this.taskRepo.recover(tasks);
    }
  }

  /**
   * List active projects (auto-filtered)
   */
  async listActiveProjects(userId: number): Promise<Project[]> {
    return this.projectRepo.find({
      where: { userId },
      // deleted_at IS NULL automatically applied
    });
  }

  /**
   * List deleted projects (in recycle bin)
   */
  async listDeletedProjects(userId: number): Promise<Project[]> {
    return this.projectRepo.find({
      where: {
        userId,
        deletedAt: Not(IsNull()),
      },
      withDeleted: true,
    });
  }

  /**
   * Permanent delete (after grace period)
   */
  async permanentDelete(projectId: number): Promise<void> {
    // Remove tasks first (foreign key constraint)
    await this.taskRepo.delete({ projectId });

    // Remove project
    await this.projectRepo.delete({ id: projectId });
  }
}

/**
 * User service with unique email handling
 */
class UserService {
  constructor(private userRepo: Repository<User>) {}

  /**
   * Create user (checks for soft-deleted duplicates)
   */
  async createUser(email: string, name: string): Promise<User> {
    // Check if email exists (including soft-deleted)
    const existing = await this.userRepo.findOne({
      where: { email },
      withDeleted: true,
    });

    if (existing) {
      if (existing.deletedAt) {
        // Reactivate soft-deleted account
        existing.name = name;
        existing.deletedAt = null;
        return this.userRepo.save(existing);
```

```typescript
      } else {
        throw new Error(`Email ${email} already in use`);
      }
    }

    // Create new user
    const user = this.userRepo.create({ email, name });
    return this.userRepo.save(user);
  }

  /**
   * Soft delete user
   */
  async deleteUser(userId: number): Promise<void> {
    const user = await this.userRepo.findOneBy({ id: userId });

    if (!user) {
      throw new Error(`User ${userId} not found`);
    }

    await this.userRepo.softRemove(user);
  }

  /**
   * GDPR compliant hard delete (anonymize first)
   */
  async gdprDelete(userId: number): Promise<void> {
    const user = await this.userRepo.findOne({
      where: { id: userId },
      withDeleted: true,
    });

    if (!user) {
      throw new Error(`User ${userId} not found`);
    }

    // Anonymize before hard delete
    user.email = `deleted_${userId}@anonymized.local`;
    user.name = '[Deleted User]';
    await this.userRepo.save(user);

    // Now hard delete
    await this.userRepo.remove(user);
  }
}

/**
 * Cleanup job (cron)
 */
class CleanupService {
  constructor(
    private projectRepo: Repository<Project>,
    private taskRepo: Repository<Task>
  ) {}

  /**
   * Hard delete projects older than 30 days
   */
  async cleanupOldProjects(): Promise<number> {
    const thirtyDaysAgo = new Date();
    thirtyDaysAgo.setDate(thirtyDaysAgo.getDate() - 30);

    // Find projects to delete
    const oldProjects = await this.projectRepo.find({
      where: {
        deletedAt: Not(IsNull()),
      },
      withDeleted: true,
    });

    const toDelete = oldProjects.filter(
      p => p.deletedAt && p.deletedAt < thirtyDaysAgo
    );

    // Hard delete tasks first
    for (const project of toDelete) {
      await this.taskRepo.delete({ projectId: project.id });
    }

    // Hard delete projects
    const result = await this.projectRepo.remove(toDelete);

    return result.length;
  }
}

// Express API endpoints
app.post('/api/projects/:id/delete', async (req, res) => {
  try {
    await projectService.deleteProject(parseInt(req.params.id));
    res.json({ success: true, message: 'Project moved to recycle bin' });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

app.post('/api/projects/:id/restore', async (req, res) => {
  try {
    await projectService.restoreProject(parseInt(req.params.id));
    res.json({ success: true, message: 'Project restored' });
  } catch (error: any) {
    res.status(404).json({ error: error.message });
  }
});
```

```
app.get('/api/projects', async (req, res) => {
  const projects = await projectService.listActiveProjects(req.user.id);
  res.json(projects);
});

app.get('/api/projects/deleted', async (req, res) => {
  const projects = await projectService.listDeletedProjects(req.user.id);
  res.json(projects);
});

app.delete('/api/projects/:id/permanent', async (req, res) => {
  try {
    await projectService.permanentDelete(parseInt(req.params.id));
    res.json({ success: true, message: 'Project permanently deleted' });
  } catch (error: any) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * Raw SQL implementation (without ORM)
 */
class ProjectRepositorySQL {
  constructor(private db: Pool) {}

  async findActive(userId: number): Promise<Project[]> {
    const result = await this.db.query(
      'SELECT * FROM projects WHERE user_id = $1 AND deleted_at IS NULL ORDER BY
created_at DESC',
      [userId]
    );
    return result.rows;
  }

  async findDeleted(userId: number): Promise<Project[]> {
    const result = await this.db.query(
      'SELECT * FROM projects WHERE user_id = $1 AND deleted_at IS NOT NULL ORDER BY
deleted_at DESC',
      [userId]
    );
    return result.rows;
  }

  async softDelete(projectId: number): Promise<void> {
    await this.db.query('BEGIN');

    // Soft delete project
    await this.db.query(
      'UPDATE projects SET deleted_at = NOW() WHERE id = $1',
      [projectId]
    );
    // Cascade to tasks
    await this.db.query(
      'UPDATE tasks SET deleted_at = NOW() WHERE project_id = $1 AND deleted_at IS
NULL',
      [projectId]
    );

    await this.db.query('COMMIT');
  }

  async restore(projectId: number): Promise<void> {
    await this.db.query('BEGIN');

    // Restore project
    await this.db.query(
      'UPDATE projects SET deleted_at = NULL WHERE id = $1',
      [projectId]
    );

    // Restore tasks
    await this.db.query(
      'UPDATE tasks SET deleted_at = NULL WHERE project_id = $1',
      [projectId]
    );

    await this.db.query('COMMIT');
  }

  async hardDelete(projectId: number): Promise<void> {
    await this.db.query('BEGIN');

    await this.db.query('DELETE FROM tasks WHERE project_id = $1', [projectId]);
    await this.db.query('DELETE FROM projects WHERE id = $1', [projectId]);

    await this.db.query('COMMIT');
  }
}

/**
 * Migration: Add soft delete to existing table
 */
async function migrateSoftDelete() {
  await db.query('BEGIN');

  // Add deleted_at column
  await db.query('ALTER TABLE projects ADD COLUMN deleted_at TIMESTAMP DEFAULT
NULL');

  // Create index for performance
  await db.query('CREATE INDEX idx_projects_deleted_at ON projects(deleted_at)');
  // Partial unique index for email (only active users)
  await db.query(`
    CREATE UNIQUE INDEX users_email_unique
    ON users (email)
    WHERE deleted_at IS NULL
  `);

  await db.query('COMMIT');
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Recycle Bin (30-Day Grace Period)

```
// Delete moves to recycle bin
await projectService.deleteProject(projectId);

// User has 30 days to restore
await projectService.restoreProject(projectId);

// After 30 days, cleanup job hard deletes
await cleanupService.cleanupOldProjects();
```

Pattern 2: Audit Trail

```
// Soft delete preserves history
SELECT * FROM projects WHERE deleted_at IS NOT NULL;

// Can see who deleted what and when
SELECT id, name, deleted_at FROM projects
WHERE deleted_at BETWEEN '2024-01-01' AND '2024-01-31';
```

Pattern 3: GDPR Compliance (Right to Access)

```
// User requests all their data (including deleted)
const allData = await userRepo.find({
  where: { userId },
  withDeleted: true,
});

// Export includes soft-deleted records
```

7. Failure Modes & Edge Cases

Forgotten Filters

Problem: Queries forget `WHERE deleted_at IS NULL`, return deleted records.

Mitigation: Use ORM global scopes, database views, or query builders.

Unique Constraint Violations

Problem: Can't reuse email after soft delete.

Mitigation: Use partial unique index `WHERE deleted_at IS NULL`.

Cascade Complexity

Problem: Forgetting to soft-delete children.

Mitigation: Application-level cascade logic, test thoroughly.

Storage Bloat

Problem: Soft-deleted records accumulate, database grows.

Mitigation: Cleanup job to hard delete after 30-90 days.

8. Performance Characteristics & Tradeoffs

Query Performance

- **Soft Delete:** Every query must filter `deleted_at IS NULL` (index required)
- **Hard Delete:** No filtering needed, simpler queries

Storage

- **Soft Delete:** Grows indefinitely without cleanup
- **Hard Delete:** Minimal storage

Recovery

- **Soft Delete:** Instant recovery (UPDATE deleted_at = NULL)
- **Hard Delete:** No recovery (need backups)

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Forgetting to Filter Queries

Fix: Use ORM global scope or database view.

Mistake 2: No Partial Unique Index

Fix: `CREATE UNIQUE INDEX ON table (column) WHERE deleted_at IS NULL`.

Mistake 3: No Cascade Soft Delete

Fix: Soft delete parent AND children in transaction.

Mistake 4: No Cleanup Job

Fix: Cron job to hard delete after grace period.

Mistake 5: Inconsistent Soft Delete Usage

Fix: Use soft delete everywhere or nowhere (consistency).

10. When NOT to Use This (Anti-Patterns)

High-Volume Event Logs

Don't soft delete logs, archive to separate table instead.

Immutable Ledgers

Don't soft delete financial transactions, they should never be deleted.

Simple Applications

If no audit requirements and no recovery needed, hard delete is simpler.

11. Related Concepts (With Contrast)

Hard Delete

Difference: Soft delete marks deleted, hard delete removes row.

Archive Table

Difference: Soft delete keeps in same table, archive moves to separate table.

Event Sourcing

Related: Never delete, only append new events (immutable log).

12. Production Readiness Checklist

Database Schema

- [ ] Add `deleted_at TIMESTAMP DEFAULT NULL` to all tables
- [ ] Create index: `CREATE INDEX idx_table_deleted_at ON table(deleted_at)`
- [ ] Partial unique indexes for unique constraints
- [ ] Foreign keys handle soft-deleted parents

Application Code

- [ ] ORM global scope filters `deleted_at IS NULL`
- [ ] Soft delete method cascades to children
- [ ] Restore method cascades to children
- [ ] Queries explicitly handle deleted records

API Endpoints

- [ ] DELETE endpoint soft-deletes
- [ ] POST /restore endpoint restores
- [ ] GET /deleted endpoint lists recycle bin
- [ ] DELETE /permanent endpoint hard-deletes (admin only)

Cleanup & Maintenance

- [ ] Cron job hard-deletes after 30-90 days

- [ ] Monitor storage growth from soft-deleted records
- [ ] GDPR compliance: anonymize before hard delete
- [ ] Backup/restore includes soft-deleted records