Pagination Under Mutation

1. The Real Problem This Exists to Solve

When paginating through a dataset that's being actively modified (insertions, deletions, updates), offset-based pagination produces inconsistent results: duplicate items across pages, missing items, or incorrect page counts. This breaks user experience and data integrity.

Real production scenario:

- Social media feed with offset pagination
- User viewing page 1 (items 0-19)
- While user views page 1, 5 new posts published
- User clicks "Next" to page 2
- **Offset-based pagination:** OFFSET 20 LIMIT 20
  - Original items 20-39 now at positions 25-44
  - User sees items 25-44 (missing items 20-24)
  - 5 items never seen by user
- Alternative: User on page 3, 10 items deleted from page 1
  - User clicks page 4: sees items already seen on page 3 (duplicates)
- **Business impact:**
  - Users miss important content
  - Duplicate content annoys users
  - Analytics corrupted (items viewed reported multiple times)
  - Perceived as "buggy" feed

The fundamental problem: Offset-based pagination assumes static dataset. When data changes between page loads, the offset no longer points to the same logical position, causing skips and duplicates.

Without pagination mutation handling:

- Offset points to physical position (row number)
- Insertions/deletions shift all subsequent offsets
- Page 2 doesn't follow page 1 logically
- User experience breaks
- No consistency guarantees

With cursor-based pagination:

- Cursor points to specific item (logical position)
- New items don't affect cursor position
- Page 2 always follows page 1
- Consistent view despite mutations
- Predictable behavior

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Offset-Based Pagination with Active Mutations

```
// Incorrect: Classic offset pagination on changing data
app.get('/api/posts', async (req, res) => {
```

```
  const page = parseInt(req.query.page) || 1;
  const pageSize = 20;
  const offset = (page - 1) * pageSize;

  const posts = await db.query(
    'SELECT * FROM posts ORDER BY created_at DESC LIMIT $1 OFFSET $2',
    [pageSize, offset]
  );

  const total = await db.query('SELECT COUNT(*) FROM posts');

  res.json({
    posts: posts.rows,
    page,
    pageSize,
    total: total.rows[0].count,
    totalPages: Math.ceil(total.rows[0].count / pageSize),
  });
});
```

Why it seems reasonable:

- Simple implementation
- Standard SQL pattern
- Easy page jumping (go to page 5 directly)
- Familiar to users (page numbers)

How it breaks:

```
Initial state (20 posts):
  Page 1: Posts 1-20 (OFFSET 0)
  Page 2: Posts 21-40 (OFFSET 20)

New post inserted at top:
  Page 1: Posts 1-20 (but actually original 1-19 + new post)
  Page 2: Posts 21-40 (OFFSET 20 = original posts 20-39)

Result: Original post 20 appears on BOTH pages (duplicate)

Or if 5 posts deleted from page 1:
  Page 1: Posts 1-20 (now showing original 1-15 + 6-20)
  Page 2: OFFSET 20 (now points to original post 25)

Result: Original posts 21-24 NEVER shown (skipped)
```

Production symptoms:

- User reports: "I saw the same post twice on different pages"
- User reports: "Some posts are missing from my feed"
- Analytics show same post viewed multiple times
- Page count changes between requests (confusing)
- Pagination controls show incorrect total pages

❌ Incorrect Approach #2: Snapshot Isolation with Stale Counts

```
// Incorrect: Use transaction snapshot but count still changes
app.get('/api/posts', async (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const pageSize = 20;
  const offset = (page - 1) * pageSize;

  await db.query('BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ');

  // Snapshot of data
  const posts = await db.query(
    'SELECT * FROM posts ORDER BY created_at DESC LIMIT $1 OFFSET $2',
    [pageSize, offset]
  );

  // But total count is from snapshot (doesn't match current)
  const total = await db.query('SELECT COUNT(*) FROM posts');

  await db.query('COMMIT');

  res.json({
    posts: posts.rows,
    total: total.rows[0].count,  // Stale count
  });
});
```

Why it seems reasonable:

- Consistent snapshot within transaction
- Data doesn't change during pagination
- REPEATABLE READ ensures consistency

How it breaks:

- Transaction only lasts for single request
- Each page request is new transaction
- Page 2 request sees different snapshot than page 1
- Can't maintain transaction across page loads
- Still get skips/duplicates between pages
- Count is from snapshot, doesn't match current state

Production symptoms:

- "Page 10 of 5" displayed (total pages decreased)
- Click last page → empty results
- Pagination controls inaccurate

❌ Incorrect Approach #3: Cursor-Based Without Handling Deletes

```
// Incorrect: Cursor pagination but doesn't handle deleted cursors
app.get('/api/posts', async (req, res) => {
```

```
  const cursor = req.query.cursor;
  const pageSize = 20;

  let query = 'SELECT * FROM posts WHERE created_at < $1 ORDER BY created_at DESC
LIMIT $2';
  const posts = await db.query(query, [cursor, pageSize]);

  res.json({
    posts: posts.rows,
    nextCursor: posts.rows[posts.rows.length - 1]?.created_at,
  });
});

// Client code
async function loadNextPage() {
  const response = await fetch(`/api/posts?cursor=${lastPostDate}`);
  // What if post with lastPostDate was deleted?
  // Query returns empty because cursor doesn't exist
}
```

Why it seems reasonable:

- Uses cursor (better than offset)
- Handles insertions correctly
- Simple implementation

How it breaks:

- If item at cursor position is deleted, query breaks
- WHERE created_at < might skip items
- Cursor points to non-existent item
- Can't continue pagination
- User stuck (can't load next page)

Production symptoms:

- User clicks "Load more" → no results
- Even though more posts exist
- Pagination appears broken
- Lost cursor position

❌ Incorrect Approach #4: Keyset Pagination with Non-Unique Sorting

```
// Incorrect: Cursor on non-unique field (created_at)
app.get('/api/posts', async (req, res) => {
  const cursorDate = req.query.cursor;

  const posts = await db.query(
    'SELECT * FROM posts WHERE created_at < $1 ORDER BY created_at DESC LIMIT 20',
    [cursorDate]
  );

  res.json({
```

```
    posts: posts.rows,
    nextCursor: posts.rows[posts.rows.length - 1]?.created_at,
  });
});
```

Why it seems reasonable:

- Cursor-based pagination
- Sorted by date (natural order)
- Simple WHERE clause

How it breaks:

- Multiple posts can have same created_at (millisecond precision)
- cursor = 2024-01-15T10:30:00.123Z
- If 5 posts have this exact timestamp
- WHERE created_at < cursor might skip or duplicate them
- Non-deterministic ordering for ties

Production symptoms:

- Posts with same timestamp appear on multiple pages
- Or posts with same timestamp skipped entirely
- Inconsistent results when refreshing

3. Correct Mental Model (How It Actually Works)

Pagination under mutation requires cursor-based pagination with unique, stable ordering keys that aren't affected by insertions/deletions elsewhere in the dataset.

The Cursor Model

```
Cursor = Unique identifier of last item on page

Page 1:
  Query: SELECT * FROM posts ORDER BY created_at DESC, id DESC LIMIT 20
  Results: Post IDs [100, 99, 98, ..., 81]
  Cursor: (created_at=2024-01-15T10:00:00, id=81)

New posts inserted (IDs 101-105):

Page 2:
  Query: SELECT * FROM posts
         WHERE (created_at, id) < (2024-01-15T10:00:00, 81)
         ORDER BY created_at DESC, id DESC LIMIT 20
  Results: Post IDs [80, 79, 78, ..., 61]

Result: No duplicates, no skips
  - New posts 101-105 would appear BEFORE cursor (not after)
  - Page 2 continues exactly from where page 1 ended
```

The Stable Ordering Guarantee

```
Requirements for cursor field:
  1. Stable: Value doesn't change after creation
  2. Unique: No two items have same value (or use composite)
  3. Sortable: Can ORDER BY this field
  4. Immutable: Field doesn't get updated

Good cursor fields:
  - id (auto-increment primary key)
  - created_at + id (composite for tie-breaking)
  - uuid (if sortable by time)

Bad cursor fields:
  - updated_at (changes on edit)
  - like_count (changes over time)
  - username (can be changed)
```

The Composite Cursor Pattern

```
Single field cursor (problems):
  created_at: "2024-01-15T10:00:00.123Z"
  Multiple posts can have same timestamp → ambiguity

Composite cursor (solution):
  (created_at: "2024-01-15T10:00:00.123Z", id: 42)
  Globally unique
  Deterministic ordering

SQL query with composite cursor:
  WHERE (created_at, id) < (cursor_date, cursor_id)
  ORDER BY created_at DESC, id DESC

This ensures:
  - All posts with same created_at are ordered by id
  - No ambiguity
  - No duplicates or skips
```

4. Correct Design & Algorithm

Strategy 1: Composite Cursor Pagination

```sql
-- Page 1 (no cursor)
SELECT id, created_at, title
FROM posts
ORDER BY created_at DESC, id DESC
LIMIT 20;

-- Page 2 (with cursor from last item of page 1)
SELECT id, created_at, title
FROM posts
WHERE (created_at, id) < ('2024-01-15 10:00:00', 42)
```

```sql
ORDER BY created_at DESC, id DESC
LIMIT 20;

-- Index required
CREATE INDEX idx_posts_pagination ON posts(created_at DESC, id DESC);
```

Strategy 2: Encoded Cursor

```
Cursor format:
  Base64({ "created_at": "2024-01-15T10:00:00Z", "id": 42 })

Benefits:
  - Opaque to client (can't manipulate)
  - Can include multiple fields
  - Can evolve format without breaking clients
```

Strategy 3: Bi-Directional Pagination

```
Next page:
  WHERE (created_at, id) < (cursor_date, cursor_id)
  ORDER BY created_at DESC, id DESC

Previous page:
  WHERE (created_at, id) > (cursor_date, cursor_id)
  ORDER BY created_at ASC, id ASC
  (then reverse results in application)
```

5. Full Production-Grade Implementation

```typescript
interface CursorData {
  created_at: string;
  id: number;
}

interface PaginationResult<T> {
  items: T[];
  nextCursor: string | null;
  prevCursor: string | null;
  hasMore: boolean;
}

class CursorPagination {
  /**
   * Encode cursor data to opaque string
   */
  static encodeCursor(data: CursorData): string {
    return Buffer.from(JSON.stringify(data)).toString('base64');
  }
```

```
/**
 * Decode cursor string to data
 */
static decodeCursor(cursor: string): CursorData {
  try {
    return JSON.parse(Buffer.from(cursor, 'base64').toString('utf-8'));
  } catch {
    throw new Error('Invalid cursor');
  }
}

/**
 * Paginate posts with cursor
 */
static async paginate(
  db: Database,
  cursor: string | null,
  limit: number = 20,
  direction: 'forward' | 'backward' = 'forward'
): Promise<PaginationResult<Post>> {
  // Fetch one extra to check if there are more results
  const fetchLimit = limit + 1;

  let query: string;
  let params: any[];

  if (!cursor) {
    // First page
    query = `
      SELECT id, title, content, created_at, author_id
      FROM posts
      ORDER BY created_at DESC, id DESC
      LIMIT $1
    `;
    params = [fetchLimit];
  } else {
    const cursorData = this.decodeCursor(cursor);

    if (direction === 'forward') {
      // Next page
      query = `
        SELECT id, title, content, created_at, author_id
        FROM posts
        WHERE (created_at, id) < ($1, $2)
        ORDER BY created_at DESC, id DESC
        LIMIT $3
      `;
      params = [cursorData.created_at, cursorData.id, fetchLimit];
    } else {
      // Previous page
      query = `
        SELECT id, title, content, created_at, author_id
```

```
        FROM posts
        WHERE (created_at, id) > ($1, $2)
        ORDER BY created_at ASC, id ASC
        LIMIT $3
      `;
      params = [cursorData.created_at, cursorData.id, fetchLimit];
    }
  }

  const result = await db.query(query, params);
  let items = result.rows;

  // Check if there are more results
  const hasMore = items.length > limit;
  if (hasMore) {
    items = items.slice(0, limit);
  }

  // Reverse results if backward pagination
  if (direction === 'backward') {
    items = items.reverse();
  }

  // Generate cursors
  const nextCursor = hasMore && items.length > 0
    ? this.encodeCursor({
        created_at: items[items.length - 1].created_at,
        id: items[items.length - 1].id,
      })
    : null;

  const prevCursor = items.length > 0
    ? this.encodeCursor({
        created_at: items[0].created_at,
        id: items[0].id,
      })
    : null;

  return {
    items,
    nextCursor,
    prevCursor,
    hasMore,
  };
}

/**
 * Paginate with filters
 */
static async paginateWithFilters(
  db: Database,
  filters: PostFilters,
```

```
  cursor: string | null,
  limit: number = 20
): Promise<PaginationResult<Post>> {
  const fetchLimit = limit + 1;

  // Build WHERE clause for filters
  const whereClauses: string[] = [];
  const params: any[] = [];
  let paramIndex = 1;

  // Apply filters
  if (filters.authorId) {
    whereClauses.push(`author_id = $${paramIndex++}`);
    params.push(filters.authorId);
  }

  if (filters.tag) {
    whereClauses.push(`tags @> $${paramIndex++}`);
    params.push([filters.tag]);
  }

  // Apply cursor
  if (cursor) {
    const cursorData = this.decodeCursor(cursor);
    whereClauses.push(`(created_at, id) < ($${paramIndex++}, $${paramIndex++})`);
    params.push(cursorData.created_at, cursorData.id);
  }

  const whereClause = whereClauses.length > 0
    ? `WHERE ${whereClauses.join(' AND ')}`
    : '';

  params.push(fetchLimit);

  const query = `
    SELECT id, title, content, created_at, author_id, tags
    FROM posts
    ${whereClause}
    ORDER BY created_at DESC, id DESC
    LIMIT $${paramIndex}
  `;

  const result = await db.query(query, params);
  let items = result.rows;

  const hasMore = items.length > limit;
  if (hasMore) {
    items = items.slice(0, limit);
  }

  const nextCursor = hasMore && items.length > 0
    ? this.encodeCursor({
```

```
          created_at: items[items.length - 1].created_at,
          id: items[items.length - 1].id,
        })
      : null;

    return {
      items,
      nextCursor,
      prevCursor: null,
      hasMore,
    };
  }
}

interface Post {
  id: number;
  title: string;
  content: string;
  created_at: string;
  author_id: number;
  tags?: string[];
}

interface PostFilters {
  authorId?: number;
  tag?: string;
}

// Express API
app.get('/api/posts', async (req, res) => {
  try {
    const cursor = req.query.cursor as string | null;
    const limit = Math.min(parseInt(req.query.limit as string) || 20, 100);
    const direction = (req.query.direction as 'forward' | 'backward') || 'forward';

    const result = await CursorPagination.paginate(db, cursor, limit, direction);

    res.json(result);
  } catch (error) {
    if (error.message === 'Invalid cursor') {
      res.status(400).json({ error: 'Invalid cursor' });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});

// With filters
app.get('/api/posts/by-author/:authorId', async (req, res) => {
  const filters: PostFilters = {
    authorId: parseInt(req.params.authorId),
    tag: req.query.tag as string,
```

```
  };

  const cursor = req.query.cursor as string | null;
  const limit = Math.min(parseInt(req.query.limit as string) || 20, 100);

  const result = await CursorPagination.paginateWithFilters(
    db,
    filters,
    cursor,
    limit
  );

  res.json(result);
});

// Client implementation
class PostsClient {
  private cursor: string | null = null;
  private hasMore = true;

  async loadNextPage(): Promise<Post[]> {
    if (!this.hasMore) {
      return [];
    }

    const url = this.cursor
      ? `/api/posts?cursor=${encodeURIComponent(this.cursor)}&limit=20`
      : '/api/posts?limit=20';

    const response = await fetch(url);
    const data = await response.json();

    this.cursor = data.nextCursor;
    this.hasMore = data.hasMore;

    return data.items;
  }

  async loadPreviousPage(prevCursor: string): Promise<Post[]> {
    const url = `/api/posts?
cursor=${encodeURIComponent(prevCursor)}&direction=backward&limit=20`;
    const response = await fetch(url);
    const data = await response.json();

    return data.items;
  }

  reset(): void {
    this.cursor = null;
    this.hasMore = true;
  }
}
```

```
// React infinite scroll example
function InfiniteFeed() {
  const [posts, setPosts] = useState<Post[]>([]);
  const [cursor, setCursor] = useState<string | null>(null);
  const [loading, setLoading] = useState(false);
  const [hasMore, setHasMore] = useState(true);

  const loadMore = async () => {
    if (loading || !hasMore) return;

    setLoading(true);
    try {
      const url = cursor
        ? `/api/posts?cursor=${cursor}&limit=20`
        : '/api/posts?limit=20';

      const response = await fetch(url);
      const data = await response.json();

      setPosts(prev => [...prev, ...data.items]);
      setCursor(data.nextCursor);
      setHasMore(data.hasMore);
    } catch (error) {
      console.error('Failed to load posts:', error);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => {
    loadMore();
  }, []);

  return (
    <div>
      {posts.map(post => (
        <PostCard key={post.id} post={post} />
      ))}
      {hasMore && (
        <button onClick={loadMore} disabled={loading}>
          {loading ? 'Loading...' : 'Load More'}
        </button>
      )}
    </div>
  );
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Social Media Feed

```
// Infinite scroll feed with cursor pagination
const feed = await CursorPagination.paginate(db, cursor, 20);
// New posts added at top don't affect pagination
// Deleted posts don't break cursor
```

Pattern 2: Real-Time Chat Messages

```
// Load older messages
const messages = await CursorPagination.paginate(
  db,
  cursor,
  50,
  'backward' // Load messages before cursor
);
```

Pattern 3: Search Results with Live Updates

```
// Search results that change over time
const results = await CursorPagination.paginateWithFilters(
  db,
  { query: searchTerm },
  cursor,
  20
);
```

7. Failure Modes & Edge Cases

Cursor Points to Deleted Item

Solution: Cursor contains item data, not just reference. Query uses >, < operators, not =.

Sorting Field Updated

Problem: If sorting by updated_at and item gets updated, position changes.

Mitigation: Use created_at (immutable) or snapshot pagination.

Bulk Deletions

Problem: Deleting all items on current page.

Mitigation: Cursor still valid, moves to next available items.

8. Performance Characteristics & Tradeoffs

Query Performance

- Composite index required: (created_at DESC, id DESC)
- WHERE (created_at, id) < (...) uses index efficiently
- O(log n) for each page load

No Total Count

Trade-off: Can't show "Page X of Y" or total results. Benefit: No expensive COUNT(*) query, scales to billions of rows.

No Random Page Access

Trade-off: Can't jump to page 10 directly. Benefit: Consistent results, no skips/duplicates.

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

Mistake 1: Single-Field Cursor on Non-Unique Column

Fix: Always include unique ID in composite cursor.

Mistake 2: Exposing Raw Database IDs in Cursor

Fix: Encode cursor as opaque base64 string.

Mistake 3: Not Creating Composite Index

Fix: CREATE INDEX ON posts(created_at DESC, id DESC).

Mistake 4: Forgetting LIMIT + 1

Fix: Fetch one extra row to detect hasMore.

Mistake 5: Using Updated Fields for Sorting

Fix: Sort by immutable fields (created_at, id).

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

Static Datasets

If data never changes, offset pagination is simpler.

Need Page Numbers

If UI requires page numbers (1, 2, 3...), offset necessary.

Small Datasets

For <1000 items, offset pagination overhead negligible.

11. Related Concepts (With Contrast)

Offset Pagination

Difference: Offset uses physical position (row number). Cursor uses logical position (item identity).

Keyset Pagination

Same concept: Cursor pagination is keyset pagination.

Infinite Scroll

Uses cursor: Infinite scroll typically implemented with cursor pagination.

12. Production Readiness Checklist

Database

- [ ] Composite index on (sort_field, id)
- [ ] Tested with millions of rows
- [ ] Query execution plan verified

API

- [ ] Cursor encoding/decoding implemented
- [ ] Invalid cursor handling
- [ ] Limit validation (max 100)
- [ ] Backward pagination support

Client

- [ ] Cursor stored between requests
- [ ] hasMore flag respected
- [ ] Loading states
- [ ] Error handling

Testing

- [ ] Test with concurrent insertions
- [ ] Test with concurrent deletions
- [ ] Test cursor on deleted item
- [ ] Load test with 1M+ rows