useTableOperations Hook - Complete Usage Guide

Introduction

`useTableOperations` is a React hook that provides access to all data and operations from `DataProviderNew`. It allows child components to interact with the data table's state, filters, sorting, pagination, and sidebar operations.

Basic Setup

```
import { useTableOperations } from '../contexts/TableOperationsContext';

function MyComponent() {
  const {
    // Destructure what you need
    filteredData,
    openDrawerWithData,
    // ... other properties and functions
  } = useTableOperations();

  // Your component logic
}
```

Important: Components using `useTableOperations` must be children of `DataProviderNew` (or `DataProvider` with `useOrchestrationLayer={true}`).

Data Properties

Raw Data

`rawData`

The sorted table data (after all processing including filtering, sorting, etc.). This is the final processed data before pagination.

Note: Despite the name, `rawData` in the context actually represents the fully processed and sorted data, not the original unfiltered data. For the original data before user filters, use `filteredData` or check the data source directly.

```
const { rawData } = useTableOperations();
// Returns: Array of all rows after filtering and sorting
```

Processed Data

`filteredData`

Data after applying all user filters.

```
const { filteredData } = useTableOperations();
// Use this for custom filtering or display
const highPriorityItems = filteredData.filter(row => row.priority === 'Hi
```

`groupedData`

Data grouped by `outerGroupField` and optionally `innerGroupField`.

```
const { groupedData } = useTableOperations();
// Structure: Array with __isGroupRow__ and __groupRows__ properties
groupedData.forEach(group => {
  if (group.__isGroupRow__) {
    console.log('Group:', group.__groupKey__);
    console.log('Rows:', group.__groupRows__);
  }
});
```

`sortedData`

Data after applying sorting.

```
const { sortedData } = useTableOperations();
// Use for total count calculations
const totalRecords = sortedData.length;
```

`paginatedData`

Current page of data (after filtering, grouping, sorting, and pagination).

```
const { paginatedData } = useTableOperations();
// This is what's displayed in the table
```

Metadata

`columns`

Array of all column names.

```
const { columns } = useTableOperations();
columns.forEach(col => console.log(col));
```

`columnTypes`

Object mapping column names to their types: `'string' | 'number' | 'date' | 'boolean'`.

```
const { columnTypes } = useTableOperations();
const type = columnTypes['sales']; // 'number'
```

`filterOptions`

Available filter values for each column (for multiselect columns).

```
const { filterOptions } = useTableOperations();
const salesTeamOptions = filterOptions['salesTeam']; // Array of options
```

Configuration Flags

`enableSort`, `enableFilter`, `enableSummation`, `enableGrouping`

Boolean flags indicating which features are enabled.

```
const { enableSort, enableFilter } = useTableOperations();
if (enableFilter) {
```

```
  // Show filter controls
}
```

`textFilterColumns`

Array of columns that use text filtering.

```
const { textFilterColumns } = useTableOperations();
```

Grouping Configuration

`outerGroupField`

Field name used for outer grouping.

```
const { outerGroupField } = useTableOperations();
// e.g., 'region'
```

`innerGroupField`

Field name used for inner grouping.

```
const { innerGroupField } = useTableOperations();
// e.g., 'quarter'
```

Styling Configuration

`redFields`, `greenFields`

Arrays of column names that should be styled red or green.

```
const { redFields, greenFields } = useTableOperations();
const isRed = redFields.includes('loss');
const isGreen = greenFields.includes('profit');
```

Percentage Columns

`hasPercentageColumns`

Boolean indicating if percentage columns are configured.

```
const { hasPercentageColumns } = useTableOperations();
```

`percentageColumns`

Array of percentage column configurations.

```
const { percentageColumns } = useTableOperations();
percentageColumns.forEach(pc => {
  console.log(pc.columnName, pc.targetField, pc.valueField);
});
```

`percentageColumnNames`

Array of percentage column names.

```
const { percentageColumnNames } = useTableOperations();
```

Additional Configuration

`multiselectColumns`

Array of column names that use multiselect filtering.

```
const { multiselectColumns } = useTableOperations();
// Check if a column uses multiselect
const isMultiselect = multiselectColumns.includes('region');
```

`enableDivideBy1Lakh`

Boolean flag indicating if numeric values should be divided by 1 lakh (100,000) for display.

```
const { enableDivideBy1Lakh } = useTableOperations();
if (enableDivideBy1Lakh) {
  // Values are displayed divided by 100,000
}
```

`enableReport`

Boolean flag indicating if report mode is enabled.

```
const { enableReport } = useTableOperations();
if (enableReport) {
  // Report features are available
}
```

reportData

Object containing report data when report mode is active. Structure depends on the report configuration.

```
const { reportData, enableReport } = useTableOperations();
if (enableReport && reportData) {
  // Access report data
  const reportTableData = reportData.tableData;
}
```

columnGroupBy

String indicating column grouping mode. Can be `'values'` or a date column name.

```
const { columnGroupBy } = useTableOperations();
// 'values' or date column name like 'date'
```

State Objects

filters

Current filter state object.

```
const { filters } = useTableOperations();
// Structure: { columnName: { value: any, matchMode: string } }
const salesFilter = filters['sales'];
```

sortMeta

Current sort configuration.

```
const { sortMeta } = useTableOperations();
// Array: [{ field: 'sales', order: 1 }] // 1 = asc, -1 = desc
```

pagination

Current pagination state.

```
const { pagination } = useTableOperations();
// { first: 0, rows: 10 }
```

expandedRows

Currently expanded rows (for grouped data).

```
const { expandedRows } = useTableOperations();
```

visibleColumns

Array of currently visible column names.

```
const { visibleColumns } = useTableOperations();
```

Search and Sort Configuration

clientSave

Boolean indicating if client-side save is enabled for the current query.

```
const { clientSave } = useTableOperations();
if (clientSave) {
  // Client-side save is enabled
}
```

searchFields

Array of field paths used for search functionality. These are the fields that will be searched when using the search term.

```
const { searchFields } = useTableOperations();
// e.g., ['name', 'email', 'address.city']
```

`sortFields`

Array of field paths used for sort functionality. These are the fields that can be used for sorting.

```
const { sortFields } = useTableOperations();
// e.g., ['sales', 'date', 'region']
```

`searchTerm`

Current search term string applied to the data.

```
const { searchTerm } = useTableOperations();
// Current search term, e.g., 'John Doe'
```

`sortConfig`

Current sort configuration object. Structure: `{field: string, direction: "asc" | "desc"}`.

Note: This is different from `sortMeta`. `sortConfig` is used for server-side or advanced sorting, while `sortMeta` is used for client-side table sorting.

```
const { sortConfig } = useTableOperations();
// { field: 'sales', direction: 'asc' } or null
```

Drawer/Sidebar State

`drawerVisible`

Boolean indicating if sidebar is open.

```
const { drawerVisible } = useTableOperations();
```

`drawerData`

Data currently displayed in the sidebar.

```
const { drawerData } = useTableOperations();
```

drawerTabs

Array of drawer tab configurations.

```
const { drawerTabs } = useTableOperations();
drawerTabs.forEach(tab => {
  console.log(tab.id, tab.name, tab.outerGroup, tab.innerGroup);
});
```

activeDrawerTabIndex

Index of currently active drawer tab.

```
const { activeDrawerTabIndex } = useTableOperations();
```

clickedDrawerValues

Values that triggered the drawer open.

```
const { clickedDrawerValues } = useTableOperations();
// { outerValue: 'North', innerValue: 'Q1' }
```

Calculations

sums

Object with sum calculations for numeric columns.

```
const { sums } = useTableOperations();
const totalSales = sums['sales']; // Sum of all sales values
```

Filter Operations

updateFilter(column, value)

Update filter for a specific column.

```
const { updateFilter } = useTableOperations();

// Text filter
updateFilter('name', 'John');

// Number filter (supports operators: >, <, >=, <=, =, <>)
updateFilter('sales', '>1000');
updateFilter('sales', '100<>500'); // Range

// Date filter
updateFilter('date', { start: new Date('2024-01-01'), end: new Date('2024
```

```
// Boolean filter
updateFilter('isActive', true);

// Multiselect filter
updateFilter('region', ['North', 'South']);
```

clearFilter(column)

Clear filter for a specific column.

```
const { clearFilter } = useTableOperations();
clearFilter('sales');
```

clearAllFilters()

Clear all active filters.

```
const { clearAllFilters } = useTableOperations();
clearAllFilters();
```

Example: Custom Filter Component

```
function CustomFilter() {
  const { updateFilter, clearFilter, filters } = useTableOperations();
```

```
  return (
    <div>
      <input
        value={filters.sales?.value || ''}
        onChange={(e) => updateFilter('sales', e.target.value)}
        placeholder="Filter sales (e.g., >1000)"
      />
      <button onClick={() => clearFilter('sales')}>Clear</button>
    </div>
  );
}
```

Sort Operations

updateSort(sortMeta)

Update sort configuration.

```
const { updateSort } = useTableOperations();

// Single column sort
updateSort([{ field: 'sales', order: 1 }]); // 1 = ascending, -1 = descer

// Multi-column sort
updateSort([
  { field: 'region', order: 1 },
  { field: 'sales', order: -1 }
]);

// Clear sort
updateSort([]);
```

Example: Sort Control

```
function SortControl() {
  const { updateSort, sortMeta } = useTableOperations();

  const handleSort = (field) => {
    const currentSort = sortMeta.find(s => s.field === field);
    const newOrder = currentSort?.order === 1 ? -1 : 1;
```

```
    updateSort([{ field, order: newOrder }]);
  };

  return (
    <button onClick={() => handleSort('sales')}>
      Sort by Sales {sortMeta.find(s => s.field === 'sales')?.order === 1
    </button>
  );
}
```

Search and Sort Operations

setSearchTerm(term)

Update the search term to filter data across search fields.

```
const { setSearchTerm, searchTerm } = useTableOperations();

// Set search term
setSearchTerm('John Doe');

// Clear search
setSearchTerm('');
```

Example: Search Input Component

```
function SearchInput() {
  const { searchTerm, setSearchTerm } = useTableOperations();

  return (
    <input
      type="text"
      value={searchTerm}
      onChange={(e) => setSearchTerm(e.target.value)}
      placeholder="Search..."
    />
  );
}
```

setSortConfig(config)

Update the sort configuration for advanced/server-side sorting.

```
const { setSortConfig, sortConfig } = useTableOperations();

// Set sort configuration
setSortConfig({ field: 'sales', direction: 'asc' });

// Clear sort configuration
setSortConfig(null);
```

Example: Advanced Sort Control

```
function AdvancedSortControl() {
  const { setSortConfig, sortConfig, sortFields } = useTableOperations();

  const handleSortChange = (field, direction) => {
    setSortConfig({ field, direction });
  };

  return (
    <div>
      {sortFields?.map(field => (
        <div key={field}>
          <button onClick={() => handleSortChange(field, 'asc')}>
            Sort {field} ↑
          </button>
          <button onClick={() => handleSortChange(field, 'desc')}>
            Sort {field} ↓
          </button>
        </div>
      ))}
    </div>
  );
}
```

Note: The difference between `sortConfig` and `sortMeta`:

- `sortConfig` - Used for server-side or advanced sorting with field paths and direction strings
- `sortMeta` - Used for client-side table sorting with field names and numeric order values

---

Pagination Operations

`updatePagination(first, rows)`

Update pagination state.

```
const { updatePagination } = useTableOperations();

// Go to first page
updatePagination(0, 10);

// Go to page 3 (if rows per page is 10)
updatePagination(20, 10);

// Change rows per page
updatePagination(0, 25);
```

Example: Custom Pagination

```
function CustomPagination() {
  const { pagination, updatePagination, sortedData } = useTableOperations
  const totalPages = Math.ceil(sortedData.length / pagination.rows);
  const currentPage = Math.floor(pagination.first / pagination.rows) + 1;

  return (
    <div>
      <button onClick={() => updatePagination(0, pagination.rows)}>First<
      <button onClick={() => updatePagination(pagination.first - paginati
        Previous
      </button>
      <span>Page {currentPage} of {totalPages}</span>
      <button onClick={() => updatePagination(pagination.first + paginati
        Next
      </button>
    </div>
  );
}
```

Column Visibility Operations

`updateVisibleColumns(columns)`

Update which columns are visible.

```
const { updateVisibleColumns, columns } = useTableOperations();

// Show only specific columns
updateVisibleColumns(['name', 'sales', 'region']);

// Show all columns
updateVisibleColumns(columns);
```

Example: Column Toggle

```
function ColumnToggle() {
  const { visibleColumns, updateVisibleColumns, columns } = useTableOpera

  const toggleColumn = (col) => {
    if (visibleColumns.includes(col)) {
      updateVisibleColumns(visibleColumns.filter(c => c !== col));
    } else {
      updateVisibleColumns([...visibleColumns, col]);
    }
  };

  return (
    <div>
      {columns.map(col => (
        <label key={col}>
          <input
            type="checkbox"
            checked={visibleColumns.includes(col)}
            onChange={() => toggleColumn(col)}
          />
          {col}
        </label>
      ))}
    </div>
  );
}
```

Row Expansion Operations

updateExpandedRows(rows)

Update which grouped rows are expanded.

```
const { updateExpandedRows, expandedRows } = useTableOperations();

// Expand specific row
updateExpandedRows({ 'North': true });

// Toggle expansion
const newExpanded = { ...expandedRows, 'North': !expandedRows['North'] };
updateExpandedRows(newExpanded);
```

Sidebar/Drawer Operations

openDrawerWithData(data, outerValue, innerValue)

Open sidebar with custom filtered data.

```
const { openDrawerWithData, filteredData } = useTableOperations();

// Open with custom filtered data
const customData = filteredData.filter(row => row.priority === 'High');
openDrawerWithData(customData, 'High Priority', null);
```

openDrawerForOuterGroup(value)

Filter by outer group field and open sidebar.

```
const { openDrawerForOuterGroup } = useTableOperations();

// Open drawer for specific outer group
openDrawerForOuterGroup('North');
```

openDrawerForInnerGroup(outerValue, innerValue)

Filter by both outer and inner group fields and open sidebar.

```
const { openDrawerForInnerGroup } = useTableOperations();

// Open drawer for specific outer and inner group combination
openDrawerForInnerGroup('North', 'Q1');
```

closeDrawer()

Close the sidebar.

```
const { closeDrawer } = useTableOperations();
closeDrawer();
```

addDrawerTab()

Add a new tab to the drawer.

```
const { addDrawerTab } = useTableOperations();
addDrawerTab();
```

removeDrawerTab(tabId)

Remove a drawer tab.

```
const { removeDrawerTab, drawerTabs } = useTableOperations();
removeDrawerTab(drawerTabs[0].id);
```

updateDrawerTab(tabId, updates)

Update a drawer tab configuration.

```
const { updateDrawerTab, drawerTabs } = useTableOperations();
updateDrawerTab(drawerTabs[0].id, {
  name: 'Custom Tab',
  outerGroup: 'North',
  innerGroup: 'Q1'
});
```

`setActiveDrawerTabIndex(index)`

Set the active drawer tab.

```
const { setActiveDrawerTabIndex } = useTableOperations();
setActiveDrawerTabIndex(1); // Switch to second tab
```

Example: Summary Card with Drawer

```
function SummaryCard({ title, filterValue }) {
  const {
    filteredData,
    openDrawerWithData,
    outerGroupField
  } = useTableOperations();

  const cardData = filteredData.filter(row => {
    const value = getDataValue(row, outerGroupField);
    return String(value) === String(filterValue);
  });

  const handleClick = () => {
    openDrawerWithData(cardData, filterValue, null);
  };

  return (
    <div onClick={handleClick} className="card">
      <h3>{title}</h3>
      <p>Count: {cardData.length}</p>
    </div>
  );
}
```

Export Operations

`exportToXLSX()`

Export current data to Excel file.

```
const { exportToXLSX, sortedData } = useTableOperations();

function ExportButton() {
  return (
    <button
      onClick={exportToXLSX}
      disabled={!sortedData || sortedData.length === 0}
    >
      Export to Excel
    </button>
  );
}
```

Utility Functions

formatDateValue(value)

Format a date value for display.

```
const { formatDateValue } = useTableOperations();
const formatted = formatDateValue(new Date()); // "Jan 15, 2024"
const formattedWithTime = formatDateValue('2024-01-15T10:30:00'); // "Jar
```

formatHeaderName(key)

Format a column name for display.

```
const { formatHeaderName } = useTableOperations();
const header = formatHeaderName('sales_amount'); // "Sales Amount"
```

isTruthyBoolean(value)

Check if a value is truthy (handles true, 1, '1').

```
const { isTruthyBoolean } = useTableOperations();
isTruthyBoolean(true); // true
isTruthyBoolean(1); // true
```

```
isTruthyBoolean('1'); // true
isTruthyBoolean(false); // false
```

isPercentageColumn(column)

Check if a column is a percentage column.

```
const { isPercentageColumn } = useTableOperations();
if (isPercentageColumn('conversion_rate')) {
  // Handle percentage column
}
```

getPercentageColumnValue(row, column)

Get the calculated percentage value for a row.

```
const { getPercentageColumnValue, paginatedData } = useTableOperations();
const percentage = getPercentageColumnValue(paginatedData[0], 'conversior
```

getPercentageColumnSortFunction(column)

Get the sort function for a percentage column.

```
const { getPercentageColumnSortFunction } = useTableOperations();
const sortFn = getPercentageColumnSortFunction('conversion_rate');
```

parseNumericFilter(filterValue)

Parse a numeric filter string into a filter object.

```
const { parseNumericFilter } = useTableOperations();
const parsed = parseNumericFilter('>1000');
// { type: 'gt', value: 1000 }
const parsedRange = parseNumericFilter('100<>500');
// { type: 'range', min: 100, max: 500 }
```

applyNumericFilter(cellValue, parsedFilter)

Apply a parsed numeric filter to a cell value.

```
const { applyNumericFilter, parseNumericFilter } = useTableOperations();
const filter = parseNumericFilter('>1000');
const matches = applyNumericFilter(1500, filter); // true
```

`applyDateFilter(cellValue, dateRange)`

Apply a date range filter to a cell value.

```
const { applyDateFilter } = useTableOperations();
const dateRange = { start: new Date('2024-01-01'), end: new Date('2024-12
const matches = applyDateFilter(new Date('2024-06-15'), dateRange); // tr
```

`isNumericValue(value)`

Check if a value is numeric.

```
const { isNumericValue } = useTableOperations();
isNumericValue(100); // true
isNumericValue('100'); // true
isNumericValue('abc'); // false
```

Complete Component Examples

Example 1: Summary Card Component

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';
import { getDataValue } from '../utils/dataAccessUtils';
import { isNil, sumBy } from 'lodash';

export default function SummaryCard({
  title,
  filterField,
  filterValue,
  showCount = true,
  showSum = false,
```

```
  sumField = null
}) {
  const {
    filteredData,
    openDrawerWithData,
    formatHeaderName
  } = useTableOperations();

  const cardData = React.useMemo(() => {
    if (!filteredData || !Array.isArray(filteredData)) return [];
    return filteredData.filter(row => {
      if (!filterField || isNil(filterValue)) return true;
      const rowValue = getDataValue(row, filterField);
      if (isNil(rowValue) && isNil(filterValue)) return true;
      if (isNil(rowValue) || isNil(filterValue)) return false;
      return String(rowValue) === String(filterValue);
    });
  }, [filteredData, filterField, filterValue]);

  const count = cardData.length;
  const sum = showSum && sumField
    ? sumBy(cardData, row => {
        const value = getDataValue(row, sumField);
        return typeof value === 'number' ? value : 0;
      })
    : null;

  const handleClick = () => {
    openDrawerWithData(cardData, filterValue ? String(filterValue) : null
  };

  return (
    <div onClick={handleClick} className="cursor-pointer card">
      <h3>{title}</h3>
      {showCount && <p>Count: {count}</p>}
      {showSum && sum !== null && <p>Total: {sum.toLocaleString()}</p>}
    </div>
  );
}
```

Example 2: Filter Control Component

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';
import { InputText } from 'primereact/inputtext';

export default function FilterControl({ column }) {
  const {
    updateFilter,
    clearFilter,
    filters,
    formatHeaderName
  } = useTableOperations();

  const currentFilter = filters[column]?.value || '';

  return (
    <div>
      <label>{formatHeaderName(column)}</label>
      <InputText
        value={currentFilter}
        onChange={(e) => updateFilter(column, e.target.value)}
        placeholder={`Filter ${formatHeaderName(column)}...`}
      />
      {currentFilter && (
        <button onClick={() => clearFilter(column)}>Clear</button>
      )}
    </div>
  );
}
```

Example 3: Export Button Component

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';
import { Button } from 'primereact/button';

export default function ExportButton() {
  const { exportToXLSX, sortedData } = useTableOperations();

  return (
    <Button
      label="Export to Excel"
      icon="pi pi-file-excel"
```

```
      onClick={exportToXLSX}
      disabled={!sortedData || sortedData.length === 0}
    />
  );
}
```

Example 4: Group Navigation Component

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';
import { getDataValue } from '../utils/dataAccessUtils';
import { uniq } from 'lodash';

export default function GroupNavigation() {
  const {
    filteredData,
    outerGroupField,
    openDrawerForOuterGroup
  } = useTableOperations();

  const groups = React.useMemo(() => {
    if (!outerGroupField || !filteredData) return [];
    return uniq(
      filteredData.map(row => getDataValue(row, outerGroupField))
    ).filter(Boolean);
  }, [filteredData, outerGroupField]);

  return (
    <div className="flex gap-2">
      {groups.map(group => (
        <button
          key={group}
          onClick={() => openDrawerForOuterGroup(group)}
          className="px-4 py-2 bg-blue-500 text-white rounded"
        >
          {group}
        </button>
      ))}
    </div>
  );
}
```

Example 5: Data Visualization Component

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';
import { getDataValue } from '../utils/dataAccessUtils';
import { sumBy, groupBy } from 'lodash';

export default function SalesChart() {
  const { filteredData, formatHeaderName } = useTableOperations();

  const chartData = React.useMemo(() => {
    if (!filteredData) return [];
    const grouped = groupBy(filteredData, row => getDataValue(row, 'regic
    return Object.entries(grouped).map(([region, rows]) => ({
      region,
      sales: sumBy(rows, row => getDataValue(row, 'sales') || 0)
    }));
  }, [filteredData]);

  return (
    <div>
      <h3>Sales by Region</h3>
      {chartData.map(({ region, sales }) => (
        <div key={region}>
          <span>{region}: </span>
          <span>{sales.toLocaleString()}</span>
        </div>
      ))}
    </div>
  );
}
```

Example 6: Multiselect Column Checker

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';

export default function ColumnTypeIndicator({ column }) {
  const { multiselectColumns, textFilterColumns } = useTableOperations();

  const isMultiselect = multiselectColumns.includes(column);
```

```
  const isTextFilter = textFilterColumns.includes(column);

  return (
    <div>
      {isMultiselect && <span className="badge">Multiselect</span>}
      {isTextFilter && <span className="badge">Text Filter</span>}
    </div>
  );
}
```

Example 7: Search and Sort Component

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';
import { InputText } from 'primereact/inputtext';
import { Dropdown } from 'primereact/dropdown';

export default function SearchAndSortControl() {
  const {
    searchTerm,
    setSearchTerm,
    sortConfig,
    setSortConfig,
    searchFields,
    sortFields
  } = useTableOperations();

  const sortOptions = sortFields?.map(field => ({
    label: field,
    value: { field, direction: 'asc' }
  })) || [];

  return (
    <div className="flex gap-2">
      <InputText
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
        placeholder="Search..."
      />
      <Dropdown
        value={sortConfig}
        onChange={(e) => setSortConfig(e.value)}
```

```
        options={[
          { label: 'None', value: null },
          ...sortOptions
        ]}
        placeholder="Sort by..."
      />
    </div>
  );
}
```

Example 8: Report Data Display

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';

export default function ReportSummary() {
  const { enableReport, reportData, columnGroupBy } = useTableOperations(

  if (!enableReport || !reportData) {
    return <div>Report mode is not enabled</div>;
  }

  return (
    <div>
      <h3>Report Summary</h3>
      <p>Grouping by: {columnGroupBy}</p>
      {reportData.tableData && (
        <p>Total records: {reportData.tableData.length}</p>
      )}
    </div>
  );
}
```

Example 9: Value Formatter with Divide by 1 Lakh

```
'use client';
import { useTableOperations } from '../contexts/TableOperationsContext';

export default function FormattedValue({ value, column }) {
  const { enableDivideBy1Lakh, isNumericValue } = useTableOperations();
```

```
  const formatValue = (val) => {
    if (!isNumericValue(val)) return val;

    const numValue = typeof val === 'string' ? parseFloat(val) : val;

    if (enableDivideBy1Lakh) {
      return (numValue / 100000).toFixed(2) + ' L';
    }

    return numValue.toLocaleString();
  };

  return <span>{formatValue(value)}</span>;
}
```

Integration Guide

Adding Children to DataProvider

```
import DataProvider from './components/DataProvider';
import DataTableNew from './components/DataTableNew';
import SummaryCard from './components/SummaryCard';
import GroupNavigation from './components/GroupNavigation';

export default function DataTablePage() {
  return (
    <DataProvider
      useOrchestrationLayer={true}
      // ... other props
    >
      {/* Custom components - can use useTableOperations */}
      <div className="grid grid-cols-3 gap-4 p-4">
        <SummaryCard
          title="High Priority"
          filterField="priority"
          filterValue="High"
          showCount={true}
        />
        <SummaryCard
          title="Active Sales"
          filterField="status"
```

```
            filterValue="active"
            showCount={true}
            showSum={true}
            sumField="sales"
          />
        </div>

        <GroupNavigation />

        {/* Data table */}
        <DataTableNew
          // ... props
        />
      </DataProvider>
    );
  }
```

Component Hierarchy

```
DataProvider (useOrchestrationLayer={true})
  ├── TableOperationsContext.Provider
  │   ├── CustomComponent1 (uses useTableOperations)
  │   ├── CustomComponent2 (uses useTableOperations)
  │   ├── DataTableNew (uses useTableOperations)
  │   └── ... other children
  └── Sidebar (managed by DataProvider)
```

Common Patterns

Pattern 1: Filtering Data for Custom Display

```
  const { filteredData, openDrawerWithData } = useTableOperations();

  // Filter by custom criteria
  const customData = filteredData.filter(row => {
    return row.status === 'active' && row.sales > 1000;
  });

  // Display or open in drawer
  openDrawerWithData(customData, 'Active High Sales', null);
```

Pattern 2: Opening Sidebar with Filtered Data

```
// Option 1: Use helper functions
const { openDrawerForOuterGroup } = useTableOperations();
openDrawerForOuterGroup('North');

// Option 2: Custom filtering
const { filteredData, openDrawerWithData } = useTableOperations();
const filtered = filteredData.filter(row => row.region === 'North');
openDrawerWithData(filtered, 'North', null);
```

Pattern 3: Accessing Grouped Data

```
const { groupedData, outerGroupField } = useTableOperations();

groupedData.forEach(group => {
  if (group.__isGroupRow__) {
    const groupKey = group.__groupKey__;
    const innerRows = group.__groupRows__;
    // Process grouped data
  }
});
```

Pattern 4: Working with Percentage Columns

```
const {
  hasPercentageColumns,
  percentageColumns,
  getPercentageColumnValue,
  isPercentageColumn
} = useTableOperations();

if (hasPercentageColumns) {
  percentageColumns.forEach(pc => {
    const value = getPercentageColumnValue(row, pc.columnName);
    // Use percentage value
  });
}
```

Pattern 5: Handling Pagination in Custom Components

```
const {
  pagination,
  updatePagination,
  sortedData
} = useTableOperations();

const totalPages = Math.ceil(sortedData.length / pagination.rows);
const currentPage = Math.floor(pagination.first / pagination.rows) + 1;

// Navigate
const goToPage = (page) => {
  updatePagination((page - 1) * pagination.rows, pagination.rows);
};
```

Pattern 6: Using Search and Sort Together

```
const {
  searchTerm,
  setSearchTerm,
  sortConfig,
  setSortConfig,
  filters,
  updateFilter
} = useTableOperations();

// Combine search, sort, and filters
const applySearch = (term) => {
  setSearchTerm(term);
};

const applySort = (field, direction) => {
  setSortConfig({ field, direction });
};

// Search works across searchFields, filters work on specific columns
```

Pattern 7: Working with Multiselect Columns

```
const {
  multiselectColumns,
  filterOptions,
  updateFilter
} = useTableOperations();

// Check if column is multiselect
const isMultiselect = multiselectColumns.includes('region');

if (isMultiselect) {
  // Get available options
  const options = filterOptions['region'] || [];

  // Update filter with array of selected values
  updateFilter('region', ['North', 'South']);
}
```

Pattern 8: Handling Report Mode

```
const {
  enableReport,
  reportData,
  columnGroupBy,
  filteredData
} = useTableOperations();

if (enableReport && reportData) {
  // Use report data
  const data = reportData.tableData || filteredData;
  const groupingMode = columnGroupBy; // 'values' or date column name
} else {
  // Use regular filtered data
  const data = filteredData;
}
```

Best Practices

1. **Destructure only what you need** - Don't destructure everything, only what your component uses.
2. **Use memoization for computed values** - Use `useMemo` for filtered/computed data.

3. **Check for data existence** - Always check if arrays exist before using them.

4. **Use appropriate data source**:

    - `filteredData` - For custom filtering
    - `paginatedData` - For current page display
    - `sortedData` - For total counts
    - `groupedData` - For grouped operations
    - `rawData` - For fully processed data (note: it's actually sorted data, not raw)

5. **Handle loading states** - Check if data is available before rendering.

6. **Use utility functions** - Leverage `formatDateValue`, `formatHeaderName`, etc. for consistency.

7. **Understanding search vs filters**:

    - `searchTerm` - Global search across multiple fields (defined in `searchFields`). Use for quick text search.
    - `filters` - Column-specific filters with operators. Use for precise filtering on specific columns.
    - Both can be used together for comprehensive filtering.

8. **Understanding sortConfig vs sortMeta**:

    - `sortConfig` - Used for server-side or advanced sorting with field paths (`{field: string, direction: "asc" | "desc"}`). Use when working with nested fields or server-side sorting.
    - `sortMeta` - Used for client-side table sorting with field names and numeric order (`[{field: string, order: 1 | -1}]`). Use for standard table column sorting.

9. **Working with report data**:

    - Always check `enableReport` before accessing `reportData`.
    - `reportData` structure depends on report configuration and may include `tableData` and other report-specific properties.
    - Use `columnGroupBy` to understand how data is grouped in report mode.

10. **Multiselect columns**:

    - Use `multiselectColumns` to check if a column supports multiselect filtering.
    - Multiselect columns use array values in filters, while text filter columns use string values.

Error Handling

If `useTableOperations` is called outside of `DataProviderNew`, it will throw an error:

```
Error: useTableOperations must be used within DataProviderNew
```

Always ensure your component is a child of `DataProviderNew` (or `DataProvider` with `useOrchestrationLayer={true}`).