**Question 1: Data Normalization and Type Conversion**

// Convert string to number safely

function parseAmount(amount: string): number {

// Handle cases like: "$1,234.56", "1234.56", "", "N/A", null

if (!amount || amount === 'N/A' || amount === '') {

return 0; // Default to 0 for invalid/missing amounts to avoid breaking aggregations

}

// Remove currency symbols, commas, and trim whitespace

const cleaned = amount.replace(/[$,]/g, '').trim();

const parsed = parseFloat(cleaned);

return isNaN(parsed) ? 0 : parsed; // Return 0 if parsing fails

}

// Standardize gender values

function normalizeGender(gender: string): 'Male' | 'Female' | 'Other' {

// Handle cases like: "M", "male", "FEMALE", "f", "", "Non-binary"

const lower = gender.toLowerCase().trim();

if (['m', 'male'].includes(lower)) {

return 'Male';

} else if (['f', 'female'].includes(lower)) {

return 'Female';

} else {

return 'Other'; // Catch-all for non-binary, empty, or unknown

}

}

// Parse date safely

function parseDate(dateString: string): Date | null {

// Handle various date formats: "2023-12-15", "12/15/2023"

if (!dateString) {

return null;

}

// Try ISO format first

let date = new Date(dateString);

if (isNaN(date.getTime())) {

// Try MM/DD/YYYY

const parts = dateString.split('/');

if (parts.length === 3) {

date = new Date(`${parts[2]}-${parts[0]}-${parts[1]}`);

}

}

return isNaN(date.getTime()) ? null : date;

}

**Question 2: Data Validation and Filtering**

interface Transaction {

CustomerID: number;

TransactionDate: string;

TransactionType: string;

TransactionAmount: string;

AccountBalanceAfterTransaction: string;

Age: number;

// Other fields as needed

}

interface CleanedTransaction extends Transaction {

// Add parsed fields if needed

}

function validateAndFilter(data: Transaction[]): CleanedTransaction[] {

const validData: CleanedTransaction[] = [];

const customerAges = new Map<number, number>(); // Track consistent age per customer

const seenTransactionIDs = new Set<number>(); // Track duplicates

for (const row of data) {

const amount = parseAmount(row.TransactionAmount);

const balance = parseAmount(row.AccountBalanceAfterTransaction);

const age = row.Age;

// Validate amount for deposits

if (row.TransactionType === 'Deposit' && amount <= 0) {

continue; // Invalid deposit

}

// Balance non-negative

if (balance < 0) {

continue;

}

// Age range

if (age < 18 || age > 120) {

continue;

}

// Consistent age

if (customerAges.has(row.CustomerID)) {

if (customerAges.get(row.CustomerID) !== age) {

continue; // Inconsistent age

}

} else {

customerAges.set(row.CustomerID, age);

}

// No duplicate TransactionID (assuming TransactionID exists in data)

if (seenTransactionIDs.has(row.TransactionID)) {

continue;

}

seenTransactionIDs.add(row.TransactionID);

validData.push(row as CleanedTransaction);

}

return validData;

}

**Question 3: Edge Case Management**

**Different ages for customers**

function resolveAgeInconsistency(data: Transaction[]): Transaction[] {

const customerData = new Map<number, Transaction[]>();

data.forEach(row => {

if (!customerData.has(row.CustomerID)) customerData.set(row.CustomerID, []);

customerData.get(row.CustomerID)!.push(row);

});

return Array.from(customerData.values()).flatMap(group => {

const ages = group.map(r => r.Age);

const modeAge = ages.sort((a,b) =>

ages.filter(v => v===a).length - ages.filter(v => v===b).length

).pop();

return group.map(r => ({ ...r, Age: modeAge }));

});

}

**Balance Doesn’t Reconcile**

function reconcileBalances(data: Transaction[]): Transaction[] {

// Sort by CustomerID and TransactionDate

data.sort((a,b) => a.CustomerID - b.CustomerID || parseDate(a.TransactionDate)!.getTime() - parseDate(b.TransactionDate)!.getTime());

let prevBalance = 0;

return data.map((row, idx) => {

if (idx > 0 && data[idx-1].CustomerID === row.CustomerID) {

const expected = prevBalance + (row.TransactionType === 'Withdrawal' ? -parseAmount(row.TransactionAmount) : parseAmount(row.TransactionAmount));

if (expected !== parseAmount(row.AccountBalanceAfterTransaction)) {

row.AccountBalanceAfterTransaction = expected.toString(); // Correct

}

}

prevBalance = parseAmount(row.AccountBalanceAfterTransaction);

return row;

});

}

**Duplicate Transactions**

function removeDuplicates(data: Transaction[]): Transaction[] {

const seen = new Set<number>();

return data.filter(row => {

if (seen.has(row.TransactionID)) return false;

seen.add(row.TransactionID);

return true;

});

}

**Question 4: Business Insights and Aggregation**

// Calculate monthly transaction volume by branch

function getMonthlyVolumeByBranch(data: CleanedTransaction[]): Map<string, Map<string, number>> {

const volume = new Map<string, Map<string, number>>();

data.forEach(row => {

const date = parseDate(row.TransactionDate);

if (!date) return;

const month = `${date.getFullYear()}-${date.getMonth() + 1}`;

const branch = row.BranchID.toString(); // Assuming Branch ID in data

if (!volume.has(branch)) volume.set(branch, new Map());

const branchMap = volume.get(branch)!;

branchMap.set(month, (branchMap.get(month) || 0) + parseAmount(row.TransactionAmount));

});

return volume;

}

// Find customers with unusual spending patterns (e.g., >3x avg amount)

function detectAnomalousTransactions(data: CleanedTransaction[]): CleanedTransaction[] {

const customerAvgs = new Map<number, number>();

data.forEach(row => {

const amt = parseAmount(row.TransactionAmount);

const id = row.CustomerID;

customerAvgs.set(id, (customerAvgs.get(id) || 0) + amt);

});

// Compute avg per customer

data.forEach(row => customerAvgs.set(row.CustomerID, customerAvgs.get(row.CustomerID)! / data.filter(r => r.CustomerID === row.CustomerID).length));

return data.filter(row => parseAmount(row.TransactionAmount) > 3 \* (customerAvgs.get(row.CustomerID) || 0));

}

// Calculate customer lifetime value (sum of net transactions)

function calculateCustomerLTV(customerId: number, data: CleanedTransaction[]): number {

return data.filter(row => row.CustomerID === customerId)

.reduce((sum, row) => {

const amt = parseAmount(row.TransactionAmount);

return sum + (row.TransactionType === 'Deposit' ? amt : -amt);

}, 0);

}

**Question 5: Strategic Data Aggregation**

 Underperforming **branches**: Metrics: Total transaction volume, avg balance per account, anomaly count (from detectAnomalousTransactions). Approach: Aggregate by BranchID using getMonthlyVolumeByBranch; rank branches by volume descending; bottom 10% are underperforming. This identifies low-activity branches for closure or marketing.

 High**-Value Customer segments**: Segment by total transaction volume (sum amounts), age groups, account type. Approach: Group by CustomerID, compute sum(TransactionAmount) > 75th percentile; further segment by Age (e.g., 18-35, 36-55, 56+). Use calculateCustomerLTV for value; high segments get targeted offers.

 Seasonal **trends**: Approach: Aggregate transactions by month/year using date parsing; compute volume, avg amount. Visualize with line charts (x: month, y: volume) and heatmaps (months vs years). Insights: Peaks in Dec (holidays), dips in Feb; use for staffing/liquidity planning.

**Question 6: Performance Optimization for Large Datasets**

Bottlenecks in cleaning: Looping over 10M rows for parsing/validation (O(n) time), Maps/Sets for consistency checks (memory-intensive).

Optimizations: Use Web Workers for parallel processing; chunk data (process 100k rows at a time); efficient data structures like objects instead of Maps for faster access. Algorithms: Sort once for reconciliations instead of per-row checks.

Memory: Use streams (e.g., PapaParse for CSV) to avoid loading all data; garbage collect after chunks. In React, use virtualized lists (react-window) for rendering.

**Question 7: Visualization and Reporting**

Visualization 1: Branch Transaction Volume

* Chart: Bar chart
* X: Branch ID, Y: Total Volume
* Data: Sum TransactionAmount by BranchID (last 6 months)
* Insight: Highlights underperforming branches for resource allocation
* Code: aggregateByBranch(filterByDateRange(data, last6Months)) where filterByDateRange filters parsed dates, aggregateByBranch uses reduce to sum.

Visualization 2: Anomalous Transactions Scatter

* Chart: Scatter plot
* X: Transaction Date, Y: Amount
* Data: Output from detectAnomalousTransactions, filtered by date
* Insight: Spots fraud patterns over time
* Code: detectAnomalousTransactions(filterByDateRange(data, lastYear))

Visualization 3: Customer LTV Distribution

* Chart: Histogram
* X: LTV Bins, Y: Customer Count
* Data: Compute LTV per customer, bin into ranges
* Insight: Identifies high-value segments for retention strategies
* Code: customers.map(id => calculateCustomerLTV(id, data)).sort().reduce(binLTV)

Visualization 4: Seasonal Trends Line

* Chart: Line chart
* X: Month/Year, Y: Avg Transaction Volume
* Data: Monthly aggregates from getMonthlyVolumeByBranch (sum across branches)
* Insight: Predicts activity peaks for liquidity management
* Code: getMonthlyVolumeByBranch(data).reduce(mergeMonths)

Visualization 5: Age vs Balance Heatmap

* Chart: Heatmap
* X: Age Groups, Y: Account Type
* Data: Avg balance by age bin and account type
* Insight: Reveals demographic preferences for product targeting
* Code: groupByAgeAndType(data).map(computeAvgBalance)

**Question 8: Real-Time Data Architecture**

Challenges: Static CSV lacks streaming; reloading full dataset is inefficient for updates, causing latency and high memory use.

Design: Use Kafka/WebSockets for real-time feeds from backend (e.g., database events); frontend subscribes via Socket.io. Technologies: React with Redux-Saga for state, Apache Kafka for ingestion, MongoDB for storage.

Modifications: Make functions async/stream-aware (e.g., process in batches); add event listeners to update aggregates incrementally (e.g., update Maps on new data). Interfaces: Add timestamps, use observables for data flow.