Lecture 22: Columnar Storage & Compression

Logistics

- Project final reports due on **Nov 17**
- In-class presentations on **Nov 21**

Recap

- Database and Unix
- DDL Operator
- Query Parsing
- Query Compilation

Lecture Overview

- Columnar Storage
- Compression
- Compressed Columnar Storage

Columnar Storage

Weather Analysis

- A weather dataset with columns for timestamp, temperature, humidity, and wind speed.
- **Query:** Find average temperate between timestamp 100 and timestamp 150

Row Storage

- Find average temperate between timestamp 5 and timestamp 15

PAGE 1			
Timestamp 1	Temperature 1	Humidity 1	Windspeed 1
Timestamp 2	Temperature 2	Humidity 2	Windspeed 2
…	…	…	…
Timestamp 10	Temperature 10	Humidity 10	Windspeed 10

PAGE 2			
Timestamp 11	Temperature 11	Humidity 11	Windspeed 11
…	…	…	…
Timestamp 20	Temperature 20	Humidity 20	Windspeed 20

Why Columnar Storage?

- **Row Storage vs. Columnar Storage**: Row storage stores all fields of a record together, while columnar storage organizes data by columns.
- **Benefits**: Columnar storage is ideal for analytical queries, which often access only a subset of columns.
- **Example Use Case**: A weather dataset with columns for timestamp, temperature, humidity, and wind speed.

Columnar Storage

- Find average temperate between timestamp 5 and timestamp 15

PAGE 1
Timestamp 1
Timestamp 2
…
Timestamp 80

PAGE 2
Temperature 1
Temperature 2
…
Temperature 80

PAGE 3
Humidity 1
Humidity 2
…
Humidity 80

PAGE 4
Windspeed 1
Windspeed 2
…
Windspeed 80

Row vs Columnar Storage

FEATURE	ROW STORAGE	COLUMNAR STORAGE
Data Organization	All attributes of a record stored together	Each attribute stored in separate files
Ideal Use Case	Transactional workloads (OLTP), where entire records are accessed frequently.	Analytical workloads (OLAP), which often require only a subset of columns
Access Pattern	Efficient for accessing full rows	Efficient for accessing specific columns
Compression	Limited compression potential	High compression potential due to similar data types within columns

Generating Synthetic Weather Data

- This schema will store four columns in a table with 1 million rows.
- **Page-based Storage**: Organizes data in 4096-byte pages to optimize read/write efficiency.

```
const int NUM_COLS = 4;  // Timestamp, Temperature, Humidity, Wind Speed
const int NUM_ROWS = 1000000;  // Example data size
const int PAGE_SIZE = 4096;  // Data stored in pages of 4096 bytes
```

Generating Synthetic Weather Data

- Temperature, humidity, and wind speed are randomly generated.
- Uses sequential timestamps to simulate real-world weather data.

```
void generateData(std::vector<std::vector<int>>& data) {
    std::normal_distribution<> temp_change(-0.5, 5);
    std::uniform_int_distribution<> humidity_dis(0, 100);
    std::uniform_int_distribution<> wind_speed_dis(0, 100);
    ...
}
```

Generated Weather Data

Timestamp	Temperature	Humidity	Wind Speed
1609459200	40	74	12
1609459202	37	84	14
1609459207	35	6	31
1609459210	35	90	59

Row Storage

- Single file storing rows (tuples) sequentially.

```
std::ofstream rowFile("row_storage.dat", std::ios::binary);
for (const auto& row : data) {
    rowFile.write(reinterpret_cast<const char*>(row.data()),
                  NUM_COLS * sizeof(int));
}
```

Columnar Storage

- Separate files for each column in the table in binary format.

```
std::vector<std::ofstream> colFiles(NUM_COLS);
for (int i = 0; i < NUM_COLS; ++i) {
    colFiles[i].open("column_storage_" + std::to_string(i) + ".dat",
                     std::ios::binary);
}
```

Querying Data with Row Storage

- Scans all rows and filters by timestamp, accumulating temperature values within the range.

```
double queryAverageTemperatureRowStorage(int startTimestamp, int endTimestamp) {
    std::ifstream rowFile("row_storage.dat", std::ios::binary);
    int sumTemperatures = 0, count = 0;
    while (rowFile.read(reinterpret_cast<char*>(row.data()), NUM_COLS * sizeof(int))) {
        if (row[0] >= startTimestamp && row[0] <= endTimestamp) {
            sumTemperatures += row[1];
            count++;
        }
    }
    return count > 0 ? static_cast<double>(sumTemperatures) / count : 0.0;
}
```

Querying Data with Row Storage

- **Limitations**: Processes entire rows, even though only one column (temperature) is needed.

```
double queryAverageTemperatureRowStorage(int startTimestamp, int endTimestamp) {
    std::ifstream rowFile("row_storage.dat", std::ios::binary);
    int sumTemperatures = 0, count = 0;
    while (rowFile.read(reinterpret_cast<char*>(row.data()), NUM_COLS * sizeof(int))) {
        if (row[0] >= startTimestamp && row[0] <= endTimestamp) {
            sumTemperatures += row[1];
            count++;
        }
    }
    return count > 0 ? static_cast<double>(sumTemperatures) / count : 0.0;
}
```

Querying Data with Columnar Storage

- Only the timestamp and temperature files are accessed, making the query faster than row storage.

```
long long queryColumnarStorage(int& filterPagesRead, int& aggregatePagesRead) {
    std::ifstream filterFile("column_storage_" + std::to_string(FILTER_COLUMN) + ".dat",
                             std::ios::binary);
    std::ifstream aggregateFile("column_storage_" + std::to_string(AGGREGATE_COLUMN) + ".dat",
                             std::ios::binary);
    long long sum = 0;
    std::unordered_map<int, std::vector<int>> pageOffsetMap;
    ..
}
```

Querying Data with Columnar Storage

- Figure out the qualifying tuples by going over the temperature file.

```
// Read the filter column and collect row offsets for qualifying rows
    for (int startRow = 0; startRow < NUM_ROWS; startRow += INTS_PER_PAGE) {
        auto filterPage = readColumnPage(filterFile, startRow, filterPagesRead);
        for (size_t i = 0; i < filterPage.size(); ++i) {
            //std::cout << filterPage[i] << "\n";
            if (filterPage[i] > FILTER_THRESHOLD) {
                pageOffsetMap[startRow / INTS_PER_PAGE].push_back(i);
            }
        }
    }
```

Querying Data with Columnar Storage

- Aggregate the temperature column using the collected tuple offset

```
// Read the aggregate column using the collected row offsets
    for (const auto& [pageIndex, offsets] : pageOffsetMap) {
        auto aggregatePage = readColumnPage(aggregateFile, pageIndex * INTS_PER_PAGE,
aggregatePagesRead);
        for (const auto& rowIndex : offsets) {
            sum += aggregatePage[rowIndex];
        }
    }
```

Illustrative Results

- Fewer pages are read from disk with columnar storage

```
Filter Selectivity: 0.003416
Row Storage Query Result: 7358
Row Storage Query Time: 0.112099 seconds
Columnar Storage Query Result: 7358
Columnar Storage Query Time: 0.00361763 seconds

Pages Read:
Total Row Storage Pages Read: 3907
Filter Pages Read: 489
Aggregate Pages Read: 473
Total Columnar Storage Pages Read: 962
```

Compression

Compression in Databases

- **Storage Reduction**: Compression reduces the space required for large tables.
- **Performance Gains**: Reduces I/O time by storing smaller, compressed data on disk.

Compression and Columnar Storage

- **Row Storage**: Mixed data types make it challenging to compress entire rows effectively, as each row contains diverse data (e.g., timestamps, names, quantities).
- **Columnar Storage**: Compression can target the specific patterns within each column, such as delta encoding for timestamps or bit-packing for small integer ranges.

Compression and Columnar Storage

- **Uniform Data Types per Column**: Each column contains only one data type (e.g., integers for temperature, strings for product IDs), which often exhibit similar values or ranges.
- **Increased Data Redundancy**: Since similar values are often grouped together in columns (e.g., temperatures in small ranges, or product IDs with repeated entries), compression algorithms are highly effective.
- **Reduced I/O for Query Performance**: Columnar compression enables more data to fit in memory, reducing I/O operations and speeding up query performance, especially for analytical workloads that access only a subset of columns.

Compression and Columnar Storage

Timestamp
1609459200
1609459202
1609459207
1609459210

Timestamp
1609459200
+2
+5
+3
+2
+5
+3
+5

Compression Algorithms

ALGORITHM	IDEAL USE CASE
Bit-Packing	Small-range numeric values
Huffman Coding	Frequent categorical strings
Byte Dictionary	Repeated values, fewer than 256

Bit Packing

- Reduce storage by only using the minimum number of bits needed to represent values.
- Suitable for columns with small ranges, like temperature in weather data.
- **Example**: If temperatures are between 0 and 15, only 4 bits are needed instead of 32 (standard int).

Bit Packing

- The temperatures (35, 40, 50, 55) fit within a 6-bit range, allowing values up to 63.
- Each temperature is represented by a 6-bit binary code.

Temperature	Compressed	Packed Compressed
35	100011	10001110
40	101000	10001100
45	110010	10110111
50	110111	

Compressing

```
std::vector<uint8_t> bitPack(const std::vector<int>& values, int bitWidth) {
    int numBits = values.size() * bitWidth;
    int numBytes = (numBits + BITS_PER_BYTE - 1) / BITS_PER_BYTE;
    std::vector<uint8_t> packedData(numBytes, 0);
    int bitPos = 0;
    for (int value : values) {
        for (int b = 0; b < bitWidth; ++b) {
            if (value & (1 << b)) {
                packedData[bitPos / BITS_PER_BYTE] |= (1 << (bitPos % BITS_PER_BYTE));
            }
            ++bitPos;
        }
    }
    return packedData;
}
```

Decompressing

- Extract each value by reading specific bit ranges from the packed data.
- Use bitwise operations to unpack each value.
- Read 6-bit chunks to reconstruct each temperature.

```
int extractValue(const std::vector<uint8_t>& packedData, int bitWidth, int index) {
    int value = 0;
    int startBit = index * bitWidth;
    for (int b = 0; b < bitWidth; ++b) {
        if (packedData[startBit / BITS_PER_BYTE] & (1 << (startBit % BITS_PER_BYTE))) {
            value |= (1 << b);
        }
        ++startBit;
    }
    return value;
}
```

Huffman Coding

- **Concept**: Assigns shorter codes to frequently occurring values and longer codes to less frequent ones.
- Suitable for columns with high-frequency categorical data, like product IDs.
- **Example**: Product IDs that frequently appear in a sales table can be represented with shorter binary codes.

Huffman Coding

- **Shorter codes** are assigned to more frequent IDs (e.g., 123456 is coded as 0).
- **Longer codes** are assigned to less frequent IDs (e.g., 456789 is coded as 111).

Product ID	Frequency
111111	100
222222	60
333333	30
444444	10

Compressed
0
10
110
111

Compressing

```
void buildHuffmanCodes(const std::unordered_map<std::string, int>& freq_map) {
    std::priority_queue<HuffmanNode*, std::vector<HuffmanNode*>, CompareNode> minHeap;
    for (const auto& pair : freq_map) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }
    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top(); minHeap.pop();
        HuffmanNode* right = minHeap.top(); minHeap.pop();
        HuffmanNode* newNode = new HuffmanNode("", left->freq + right->freq);
        newNode->left = left; newNode->right = right;
        minHeap.push(newNode);
    }
    root = minHeap.top();
    buildCodes(root, "");
}
```

Byte Dictionary Encoding

- **Concept**: Assigns unique byte values (0-255) to frequently occurring strings or values, storing each as a single byte.
- Efficient for columns with many repeated values, such as product IDs or categories.
- **Example**: Each product ID is assigned a unique byte, reducing storage size for frequent IDs.

Byte Dictionary Encoding

- Assigns unique byte values (0-255) to frequently occurring strings or values, storing each as a single byte.

Product ID
111111
222222
333333
444444

Compressed
0x01
0x02
0x03
0x04

Compressing

```
for (const auto& record : table) {
    if (dict.find(record.product_id) != dict.end()) {
        encoded_data.push_back(dict[record.product_id]);
    } else {
        encoded_data.push_back(static_cast<unsigned char>(dict_size));
        reverse_dict[static_cast<unsigned char>(dict_size)] = record.product_id;
        dict_size++;
    }
}
```

Decoding Byte Dictionary Data

- **Reverse Dictionary**: Each byte value in encoded_data maps back to the original product ID using reverse_dict.
- **Efficiency**: Enables efficient decompression for queries by looking up values.

Compressed
0x01
0x02
0x03
0x04

Product ID
111111
222222
333333
444444

Illustrative Results

```
Total quantity sold: 412712, Total sales: $2.26383e+07
Uncompressed query time                  : 7.79113 milliseconds
Huffman compressed query time            : 2.24537 milliseconds
Byte Dictionary compressed query time: 0.707792 milliseconds
```

Compressed Columnar Storage

Weather Dataset: Delta Encoding

- For weather data (timestamps, temperature, humidity, wind speed), compression can save space due to the narrow range of values.
- **Concept**: Stores differences between consecutive values instead of full values.
- Timestamps often increase incrementally, so delta encoding can represent each as an offset from the previous timestamp.

```
std::vector<int> deltaTimestamps(timestamps.size());
deltaTimestamps[0] = timestamps[0];
std::adjacent_difference(timestamps.begin(), timestamps.end(), deltaTimestamps.begin());
```

Zero-Based Adjustment of Temperature Values

- Small temperature variations around a baseline allow shifting to start from zero.
- Subtract minimum temperature, creating "zero-based indexing" to reduce range.

```
int minTemp = *std::min_element(temperatures.begin(), temperatures.end());
std::transform(temperatures.begin(), temperatures.end(), adjustedTemperatures.begin(),
               [minTemp](int temp) { return temp - minTemp; });
```

Bit-Packing

- Use only as many bits as necessary to store adjusted temperatures
- 4-bit width is sufficient for temperature data with small ranges.

```
std::vector<uint8_t> bitPack(const std::vector<int>& values, int bitWidth);
auto packedTemperatures = bitPack(adjustedTemperatures, TEMPERATURE_BIT_WIDTH);
```

Bit-Packing

- Use only as many bits as necessary to store adjusted temperatures
- 4-bit width is sufficient for temperature data with small ranges.

```
std::vector<uint8_t> bitPack(const std::vector<int>& values, int bitWidth);
auto packedTemperatures = bitPack(adjustedTemperatures, TEMPERATURE_BIT_WIDTH);
```

Storing Compressed Data

- **Delta Timestamps**: Stored in a separate binary file.
- **Bit-Packed Temperatures**: Saved as a compact binary file, reducing disk space.

```
std::ofstream deltaFile("delta_encoded_timestamps.dat", std::ios::binary);
std::ofstream tempFile("packed_temperatures.dat", std::ios::binary);
tempFile.write(reinterpret_cast<const char*>(packedTemperatures.data()), packedTemperatures.size());
```

Querying Compressed Columnar Data

- **Bit-Packed Decoding**: Extracts packed values using bitwise operations, reconstructing temperature data within a range.
- **Delta Decoding**: To retrieve timestamps, add each delta back to a cumulative sum.

```
size_t startBitPos = startOffset * TEMPERATURE_BIT_WIDTH;
size_t endBitPos = (endOffset + 1) * TEMPERATURE_BIT_WIDTH;

Delta Decoding:   [1609459200, +2, +5, ...] ->
                  [1609459200, 1609459202, 1609459207, ...]
```

Illustrative Results

```
Average Temperature (Row Storage)     : 36.9554°C
Query Time (Row Storage)              : 17.57 milliseconds
Query Time (Columnar Storage)         :  5.13 milliseconds
Query time on compressed data         :  4.08 milliseconds
```

Conclusion

- Columnar Storage
- Compression
- Compressed Columnar Storage