Cache-Efficient Top-k Aggregation over High Cardinality Large Datasets

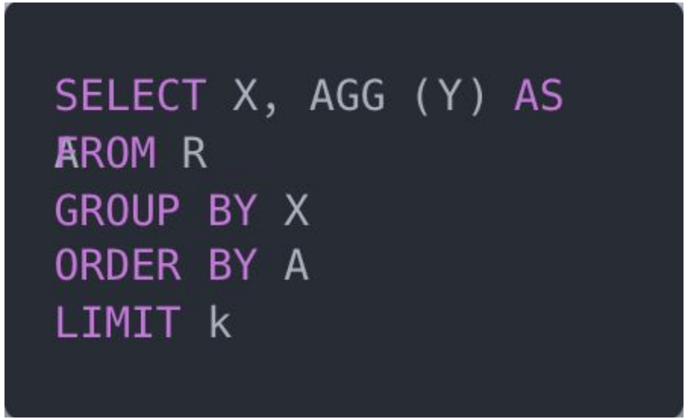
Microsoft Research

4th July, 2025

Presented by: Yash Bhisikar

Problem Statement

Given a table R, consisting of dimension attribute X and measure attribute Y, we want to maximize the throughput of top-k aggregate queries with the following template:



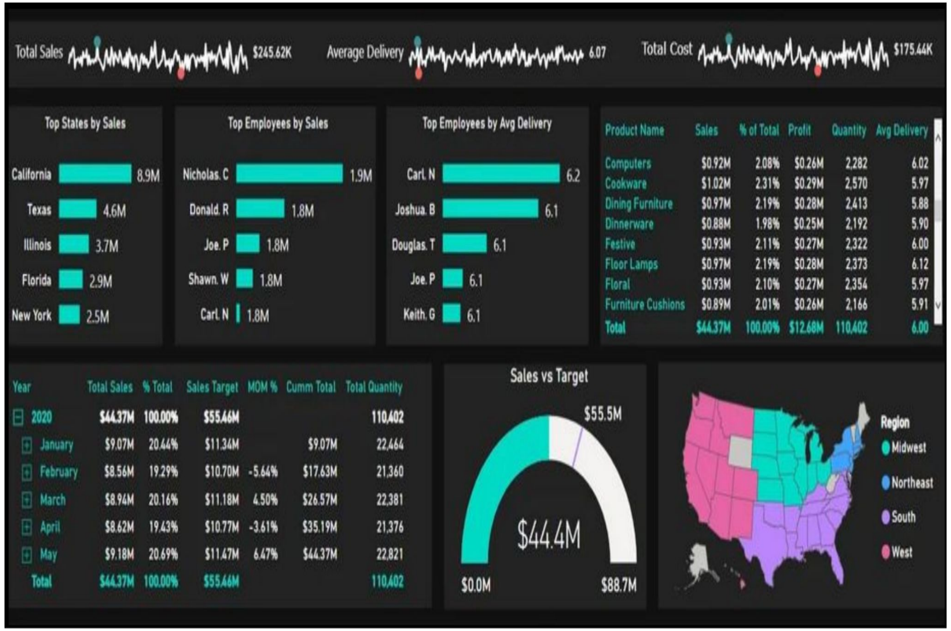
```
SELECT X, AGG (Y) AS
FROM R
GROUP BY X
ORDER BY A
LIMIT k
```

Primary focus is on monotonic aggregation functions such as ***COUNT****(*)*, ***MAX****(Y)*, ***MIN****(Y)*, and ***SUM****(Y)* with Y ≥ 0. Assume a multi-core main memory system

Why is this a problem?

Standard methods compute exact aggregates for all groups before selecting the top k.

This is highly inefficient for large datasets with high cardinality, as the massive volume of intermediate results overwhelms CPU caches, leading to excessive and slow data movement between the cache and main memory.



Product Name	Sales	% of Total	Profit	Quantity	Avg Delivery
Computers	$0.92M	2.08%	$0.26M	2,282	6.02
Cookware	$1.02M	2.31%	$0.29M	2,570	5.97
Dining Furniture	$0.97M	2.19%	$0.28M	2,413	5.88
Dinnerware	$0.88M	1.98%	$0.25M	2,192	5.90
Festive	$0.93M	2.11%	$0.27M	2,322	6.00
Floor Lamps	$0.97M	2.19%	$0.28M	2,373	6.12
Floral	$0.93M	2.10%	$0.27M	2,354	5.97
Furniture Cushions	$0.89M	2.01%	$0.26M	2,166	5.91
Total	$44.37M	100.00%	$12.68M	110,402	6.00

Year	Total Sales	% Total	Sales Target	MOM %	Cumm Total	Total Quantity
⊟ 2020	$44.37M	100.00%	$55.46M			110,402
⊞ January	$9.07M	20.44%	$11.34M		$9.07M	22,464
⊞ February	$8.56M	19.29%	$10.70M	-5.64%	$17.63M	21,360
⊞ March	$8.94M	20.16%	$11.18M	4.50%	$26.57M	22,381
⊞ April	$8.62M	19.43%	$10.77M	-3.61%	$35.19M	21,376
⊞ May	$9.18M	20.69%	$11.47M	6.47%	$44.37M	22,821
Total	$44.37M	100.00%	$55.46M			110,402

Problem Setup and Assumptions

- Target Scenarios
  - High cardinality: Millions of unique groups
  - Large datasets: Hundreds of millions of records
  - Top-k queries: k << M (top-k value much smaller than total groups)
- Dataset Characteristics
  - Assumption: N >> M >> C
  - N: Dataset size (number of records), M: Number of unique groups, C: Cache capacity
- Aggregation Functions
  - Primary focus: Monotonic functions (COUNT, SUM, MAX, MIN)
  - Secondary: Non-monotonic functions (AVG)
- Data Access Pattern
  - No indexing or pre-partitioning on grouping attributes - works with raw data

Core Idea

Instead of full aggregation, the framework(called Zippy) leverages the inherent **skew** found in real-world data. It uses an adaptive, multi-pass algorithm to:

1. Quickly identify a small set of *candidate groups* that are likely to be in the top-k results.
2. Perform exact, cache-efficient aggregation for only these candidates.
3. Cheaply prune the vast majority of non-candidate groups using efficient hashing and partitioning, without needing to fully aggregate them.

This approach drastically reduces data movement and unnecessary computation.

The Algorithm

Three main phases

1. **Candidate Selection:** A small sample of the data is analyzed to validate data skew and identify an initial set of *candidate groups* for the **Fine-grained Aggregates (FA)** cache.
2. **Multi-Pass Processing:** The data is processed in passes. In each pass:
   - **Fine-grained Aggregates (FA):** A cache-resident hash table is used to compute exact aggregates for the candidate groups with high efficiency.
   - **Coarse-grained Aggregates (CA):** Non-candidate groups are handled by a partitioning mechanism that only computes lightweight statistics (sum, count, etc.) for each partition. This avoids expensive exact aggregation for the vast majority of groups.
3. **Merge & Prune:** After each pass, results are merged across cores. A `topKBound` is calculated (the k-th highest value seen so far). Any partition in the CA whose statistics guarantee it cannot contain a top-k result is pruned from subsequent passes.

Algorithm Details I - Cache-Resident Structures

FA: Fine-grained Aggregates

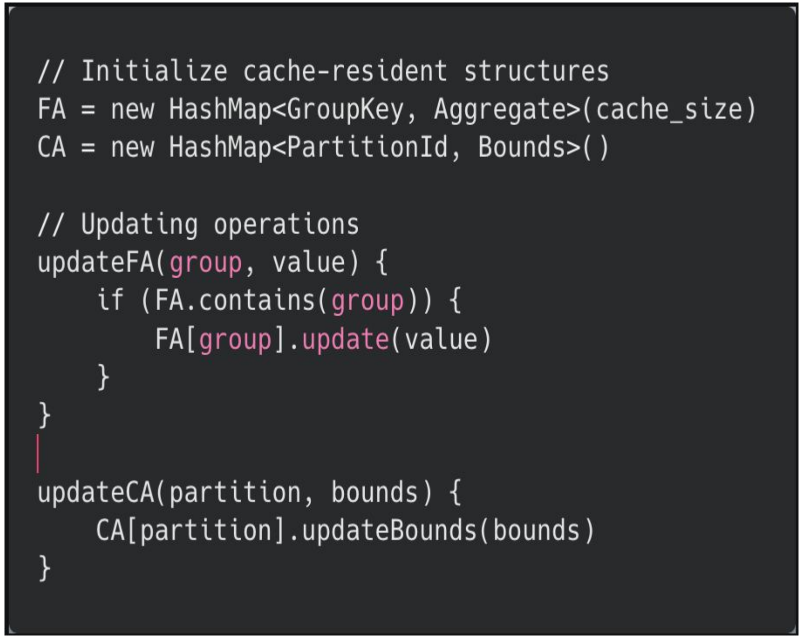
- Stores exact aggregates for candidate groups
- Maintained in CPU cache for fast access
- Size limited by cache capacity
- Updated incrementally as data is processed

CA: Coarse-grained Aggregates

- Partition-level statistics for pruning
- Maintains bounds for each partition
- Much smaller memory footprint
- Enables early termination decisions

Cache Efficiency Benefits

- Hot data stays in cache
- Only process promising candidates
- Structures size based on available cache



```
// Initialize cache-resident structures
FA = new HashMap<GroupKey, Aggregate>(cache_size)
CA = new HashMap<PartitionId, Bounds>()

// Updating operations
updateFA(group, value) {
    if (FA.contains(group)) {
        FA[group].update(value)
    }
}

updateCA(partition, bounds) {
    CA[partition].updateBounds(bounds)
}
```

Algorithm Details II - Candidate Sampling

Uniform Sampling: Each core scans its input data and selects a small, random sample of tuples. The sample size is statistically determined to be representative. This step is extremely fast.

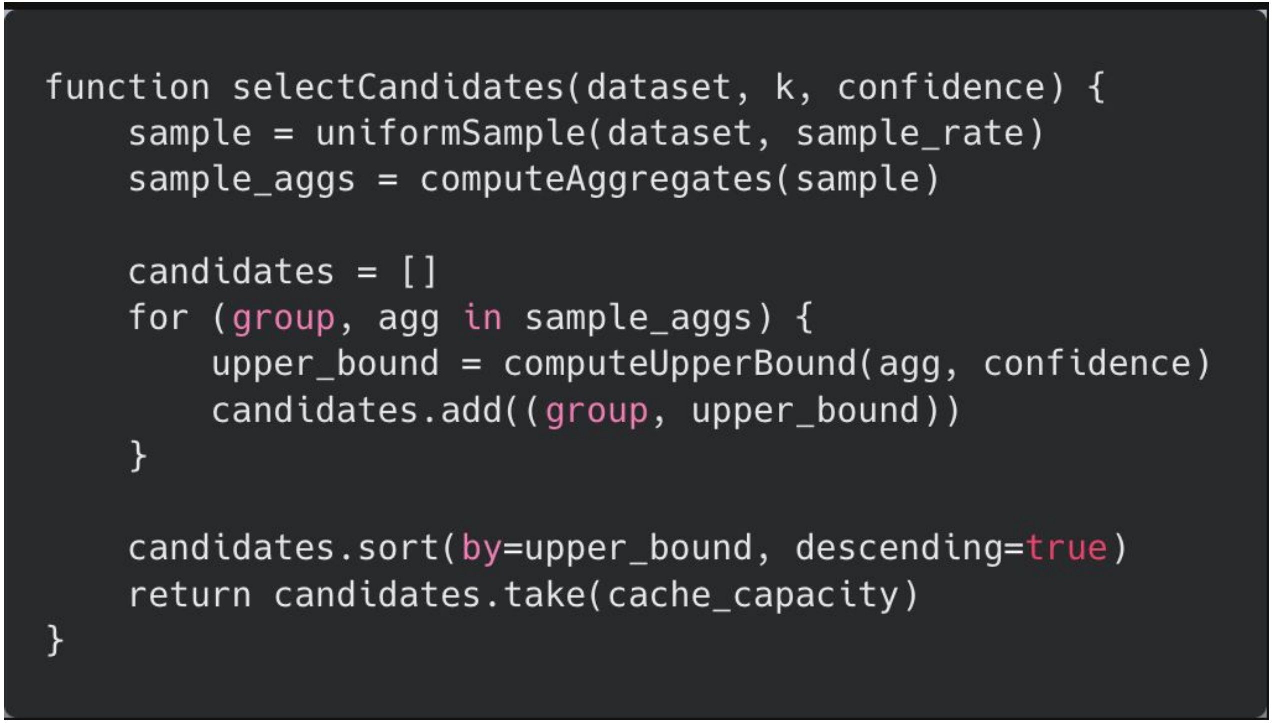
Skew Validation via Confidence Intervals:

- Aggregates are computed on the sample data.
- For each group in the sample, a confidence interval is calculated for its true aggregate value (e.g., using Hoeffding's inequality for SUM/COUNT).
- The algorithm checks if the distribution is sufficiently skewed. If the number of potential candidates (groups whose lower-bound estimate is high) is too large to fit in the cache, Zippy determines optimization is not feasible and reverts to a standard aggregation method.

Identifying FA Groups:

- If the skew is validated, groups with high lower-bound estimates are selected as candidates for the **Fine-grained Aggregates (FA)** cache.
- To maximize efficiency, any remaining space in the FA cache is filled with "heavy hitters" (groups with the highest frequency in the sample).

Algorithm Details II - Candidate Sampling



```
function selectCandidates(dataset, k, confidence) {
    sample = uniformSample(dataset, sample_rate)
    sample_aggs = computeAggregates(sample)

    candidates = []
    for (group, agg in sample_aggs) {
        upper_bound = computeUpperBound(agg, confidence)
        candidates.add((group, upper_bound))
    }

    candidates.sort(by=upper_bound, descending=true)
    return candidates.take(cache_capacity)
}
```

Algorithm Details III - Adaptive Partitioning and Pruning

For each partition of data, the algorithm makes an adaptive choice:

- **Perform Exact Aggregation:** If a partition has very few distinct groups or exhibits high data locality (many occurrences of the same group are close together), it's more efficient to compute exact aggregates for all groups within it directly.
- **Perform Partitioning (for non-candidate groups):** If exact aggregation is not viable, the algorithm partitions the data for non-candidate groups. It adaptively chooses between:
    - **Logical Partitioning:** This is the default and cheapest option. Only partition-level statistics are maintained in a hash table. Tuples are **not** physically moved. This allows for rapid pruning if the statistics are sufficient.
    - **Physical Partitioning:** If a logical partition could not be pruned in a previous pass, the algorithm escalates to physical partitioning, where tuples are actually moved into new, co-located memory regions. This prepares the data for more efficient processing in later passes.

Algorithm Details IV - Merge and Prune

- At the end of each pass, a global synchronization occurs.
  - A `topKBound` is established, representing the current value of the k-th item in the result set.
  - The key step is **pruning**: any partition whose coarse-grained statistics (e.g., the total sum of the partition) prove it cannot possibly contain a group with an aggregate value greater than `topKBound` is discarded entirely from all future processing.
- Pruning Mechanism:
  - Maintain upper bounds for each partition
  - Compare bounds with current k-th largest aggregate
  - Prune partitions whose upper bound < k-th largest
  - Process remaining partitions in order of potential

More details in the paper!

1. Parallelization
2. Rolling Top-k queries
3. Tuning the thresholds and ablation studies
4. Results

Thank You