Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

Chair for Database Systems

School of Computation, Information and Technology

Technical University of Munich

28.04.2026

Lab 2 is underway

- Questions?

- Next Week: Assignment 2 Deadline
  - Cloud Setup: Microsoft Azure
  - Assignment 3 Introduction

Lab 1 Recap - Idempotency Race

Suppose the following scenario:

t=0 - Worker A picks partition P, starts
t=15 - Coordinator times out A, requeues P
t =16 - Worker B picks up P
t=17 - Worker A's network finally delivers PARTITION_RESULT for p
t=30 - Worker B partition result arrives, counted normally

```
void complete_partition(WorkQueue& q, const string& url,
                        const map& counts) {
    if (q.in_progress.erase(url) == 0) return;   // late result, drop it
    for (auto& [k, v] : counts) q.global_counts[k] += v;
    q.completed_count++;
}
```

Lab 1 Recap - Top-K is not only sorting

```
sort() + resize(k)          →  O(n log n)
partial_sort / nth_element →  O(n log k)   ← the right idiom for top-k
```

Nits:

Deterministic tiebreaker:

- Without it, two zones with count=5000 rank in unordered_map iteration order different on every run. Add a key-asc tiebreaker.

`resize(k)` without bounds check:

- If `counts.size() < k`, you pad with default-constructed pairs.
- Use `min(k, counts.size())`.

Lab 1 Recap - Shutdown Handshake

- Ungrateful exits make the coordinator unable to distinguish "worker finished cleanly" from "worker crashed mid-batch." Both look identical at the socket layer (close → recv returns nullopt). The handshake is what restores that signal.

- WORKER_DONE -> SHUTDOWN Exchange Omitted
  - Works because of TCP cascading
  - Coordinator recv_message returns nullopt on close => requeue_worker_partitions
  - **Real cost**: coordinator can't tell clean exit from crash. Phase 2 of Lab 2 needs this distinction.

Lab 1 Recap - Requeue Granularity

- Split - some requeue the timed-out partition, others yank all worker’s partitions.
  - Per-partition: minimum blast radius, but may keep assigning more work to the slow worker
  - Per-worker: fail-fast, but wastes work if worker was just briefly stalled
  - No right answer. Per-worker is what Spark does (executor-level failure); per-partition is what Hadoop did (task-level retry). Both ship.

Lab 1 Recap - Your highlights

- Bottleneck Analysis - Mert: profiler appendix in report
- Exception-Safe Worker - Jingyi: try/catch over the connection
- Full Idempotency - Nourallah: find + worker_id match
- LaTeX Report - Filipe: with charts and ideal-vs-measured speedup

The shuffle primitive

Data Shuffling

- **Goal**: Enable **scale-out** for all operations
  - Difficult to achieve without robust shuffling in place
- Tasks
  - Can repartition data across the cluster
  - Hash-based manner (e.g., GROUP BY keys)
  - Range-based (sort or distribution on sorted columns)
  - Random (simple load balancing to avoid skew)
- The design space for shuffle is huge

Data Shuffling Taxonomy

Dimension	Examples
Deployment	In-service / Separate shuffle service
Topology	N:N / N:M / Multi-level
Storage	Main memory / Local disk / Networked Storage
Partitioning	Static / Dynamic / Overpartitioned
Execution Mode	Streaming across stages / Blocking
Fault Tolerance	Can restart within query / Need to restart whole query
Networking	Implicit (distributed file system) / Explicit (TCP, …) / Specialized (RDMA)
Data Layout	Row-based / Column-based / Compressed / Uncompressed

Hash Shuffle (naive)

Machine

Executor

map

map

Executor

map

map

Local Disk

M*R

秦言夏语

Src: https://www.alibabacloud.com/blog/learning-about-distributed-systems---part-17-shuffle_600222

Hash Shuffle (naïve)

- Each mapper writes one file per reducer
- Total files = M × R (mappers × reducers)
- Large jobs → 100s of millions of files

→ Massive pressure on disks & network

Hash Shuffle (consolidated)

Machine

Executor

map

map

Executor

map

map

Local Disk

P*R

Src: https://www.alibabacloud.com/blog/learning-about-distributed-systems---part-17-shuffle_600222

Consolidated Hash Shuffle

- Only concurrent map tasks need output files
- Executors share shuffle files across maps
- Total files reduce to P × R
- P = number of concurrent tasks on executor
- Introduced in Spark (0.8.1)

Benefit: File count no longer scales with number of mappers.
Limitation: Still depends on number of reducers (R).

Sort Shuffle

Src: https://www.alibabacloud.com/blog/learning-about-distributed-systems---part-17-shuffle_600222

Sort Shuffle (default in Spark)

- Mapper writes one data file + one index file (2 files)
- Total files = 2 × M, independent of R

Process:

- Collect records
- Sort by partitionId (and key)
- Spill sorted segments
- Merge into final sorted file
- Create index for reducers

Allows reducers to read only their partitions.

Sort Shuffle

Machine

Executor

map sort&spill

map sort&spill

Executor

map sort&spill

map sort&spill

merge index

merge index

merge index

merge index

Local Disk

2M

秦言夏语

Src: https://www.alibabacloud.com/blog/learning-about-distributed-systems---part-17-shuffle_600222

Sort Shuffle (default in Spark)

- Mapper writes one data file + one index file (2 files)
- Total files = 2 × M, independent of R

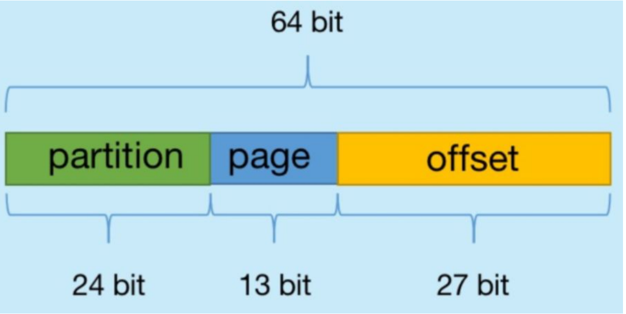
Process:

- Collect records
- Sort by partitionId (and key)
- Spill sorted segments
- Merge into final sorted file
- Create index for reducers

Allows reducers to read only their partitions.

Further Shuffle Optimizations

- **Bypass Merge Sort Shuffle**
  - Full sorting is expensive.
  - If the number of partitions (reducers) is small, Spark can bypass sorting entirely.
  - Uses hash partitioning, then performs a lightweight merge to produce a single data file.
- **Tungsten Sort Shuffle (Spark-specific)**
  - Addressing CPU bottlenecks
  - Part of Spark’s Tungsten Project (memory + CPU efficiency).
  - Stores shuffle data in off-heap binary format → avoids JVM object overhead.
  - Sorts serialized records directly (no deserialization).
  - Uses 8-byte pointer sorting → more cache-friendly, fewer CPU cycles.



SQL Example - Distributed Pre-Aggregations

```
SELECT passenger_count, MAX(max_fare)
FROM tripdata
GROUP BY passenger_count
```

How this executes in a distributed system

- Each worker node reads a subset of the table (each has different partitions).
- Each worker computes a partial aggregation:
  - GROUP BY passenger_count
  - MAX(max_fare) over its local data.
- Partial results are shuffled: rows with the same passenger_count meet on the same node.

SQL Example - Pre-Aggregations

```
SELECT passenger_count, MAX(max_fare)
FROM tripdata
GROUP BY passenger_count
```

Logical Distributed Query Plan

```
HashAggregate (FINAL): groupBy=[passenger_count],
aggr=[MAX(max_fare)]
     ↑ Exchange (Shuffle)
HashAggregate (PARTIAL): groupBy=[passenger_count],
aggr=[MAX(max_fare)]
     ↑
Scan: tripdata.parquet (distributed across workers)
```

SQL Example - Pre-Aggregations

tripdata (part0) → Worker 0 (Hash Aggregate) → hash_agg (part0)

tripdata (part1) → Worker 1 (Hash Aggregate) → hash_agg (part1)

tripdata (part2) → Worker 2 (Hash Aggregate) → hash_agg (part2)

Shuffle

Worker (hash_agg, final result)

SQL Example - Pre-Aggregations

tripdata (part0) → Worker 0 (Hash Aggregate) → hash_agg (part0)

tripdata (part1) → Worker 1 (Hash Aggregate) → hash_agg (part1)

tripdata (part2) → Worker 2 (Hash Aggregate) → hash_agg (part2)

Worker (hash_agg, final result)

Shuffle

NOTE: Pre-aggregation is only worthwhile when key cardinality is low; if almost no duplicates exist, local pre-aggregation provides no significant benefit.

The four patterns of distributed tasks

#	Pattern	Communication	Cost	Lab
1	Embarrassingly Parallel	None	T ≈ N/P	HTTP fetch in Lab 1
2	Reduce / fold	All workers → coordinator (one shot)	T ≈ N/P + |distinct keys|	Lab 1 (complete_partition merge)
3	Shuffle-aggregate	Workers → workers, hash-partitioned	T ≈ N/P + (combined size)/P	Lab 2
4	Shuffle-join (New)	Both inputs → workers, co-partitioned	Compute and Shuffle T ≈ (N+M)/P + (N+M)/P	Lab 3

Embarrassingly parallel operators: do not change the partitioning scheme of the data.
Reduce / fold: A single worker (the coordinator) sees every key.
Shuffle-aggregate: All records with the same key must be co-located on the same worker.
Shuffle-join: Matching rows from multiple sources must be co-located on the same worker.

Case Study: Database Operators

Every parallel/distributed database (Spark, Snowflake, BigQuery, ClickHouse, Umbra) decomposes a query into exactly these four shapes.

#	Pattern	Operators
1	Embarrassingly Parallel	Scan, Filter, Projection
2	Reduce / fold	Scalar aggregates (COUNT, SUM, MIN/MAX), TOP-K / LIMIT, central ORDER BY
3	Shuffle-aggregate	GROUP BY, COUNT(DISTINCT)
4	Shuffle-join (New)	Join

Three partitioning primitives (DeWitt & Gray, 1992)

Key	What it controls	Locality	Skew
Round-robin	Hand out the next chunk	None	By construction
Hash	partition = h(key) mod P	On equality	If keys are skewed
Range	partition = bucket(key)	On equality + ordering	Often skewed without histograms

- Lab 1 work queue is round-robin: locality was not important
- Lab 2/3 shuffle is hash: we want locality on the agg. / join key

Three partitioning keys

Key	What it controls	Set by
Input partition key	How data is split before workers see it	The data (file boundaries)
Shuffle key	How data is redistributed among workers	The system (the query plan)
Aggregation key	What the user actually grouped by	The user (the query)

Three keys (Lab 2)

Each worker reads its files and builds a partial hash map of (PU, DO)

- **Input partition key** = month(pickup_datetime) + per-month sharding

Shuffle: redistribute by hash of the pair

- **Shuffle key** = (PULocationID, DOLocationID)

Per-shard merge: each shard owner aggregates its pile

- **Aggregation key** = (PULocationID, DOLocationID)

In Lab 2:

- Shuffle key = Aggregation key

In Lab 3:

- Shuffle key ≠ Aggregation key
  - Stay tuned

Assignment 2

Deadline: Session 4 **(05.05.2026), 2 weeks.**

Next session (05.05):

- Focus on shuffle-join operators
- Azure Cloud Hands-On Session
- Further material:
  - Graefe, Volcano: An Extensible and Parallel Query Processing System
  - Alibaba Cloud - Learning about Distributed Systems (Shuffle)

Thank you for your attention!