C++ MAPREDUCE

SINGLE NODE EDITION

November, 2019

Roman Gershman, romange@gmail.com

Thanks to Adi Solodnik for making these slides great.

github.com/romange/gaia

UBIMO

https://www.ubimo.com/about/

Mobile and Digital Out Of Home DSP

- Billions of records are processed per day
- Dozens of pipelines of various complexity

Location-based marketing Intelligence Platform

- Many more pipelines.

Ubimo MR development history

2015 First single-node C++ MR (on AWS)

2016 Dozens of pipelines deployed in prod

2017 We switched to GCP, cloud economy changed
Developed MR2 - distributed version

2019 Adopted flow building principles from Java-based frameworks, improved our infrastructure - single node MR3 over GAIA (open sourced)

Prelude – 20 years ago

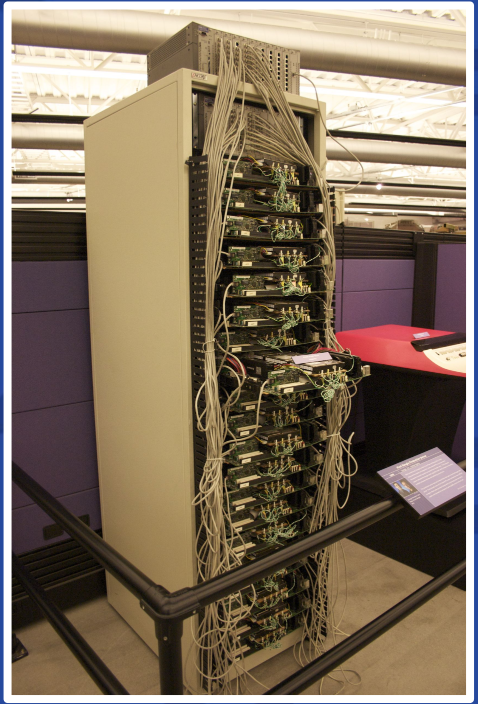
- Data center real-estate economy – rent per area
- Renters got cooling and electricity included.

Most companies

Google – 1998

- Bought cheap unreliable hardware
- Stripped what they could
- Fully utilized their rental capacity
- Beat data-center hosting company at their game

Used economics to their advantage



Google 2002-2004

Problem - reliability at scale:

- Distributed files system - GFS
  "It provides **fault tolerance while running on inexpensive commodity hardware**, and it delivers high aggregate performance to a large number of clients…"

- Mapreduce paper
  "The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, **handling machine failures**, and managing the required inter-machine communication"

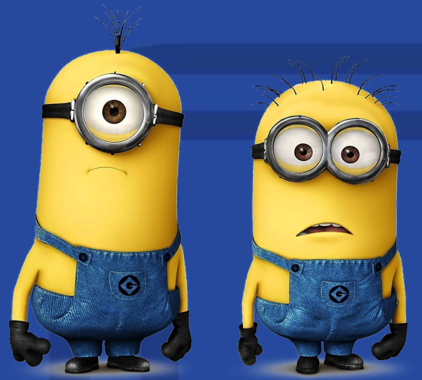
To summarize

- Google used datacenter economy to their advantage
- Used unreliable, weak (1-2 cores) machines
- Butterfly effect - creation of fault tolerant & distributed systems suited for cheap hardware

What's more expensive?

96
1-core
machines

Single
96-core
machine



What's more powerful?

96
1-core
machines

Single
96-core
machine

Design Goals

Reduce I/O usage

Best Value per dollar

Word Count Benchmark

100 beam 1-cpu nodes

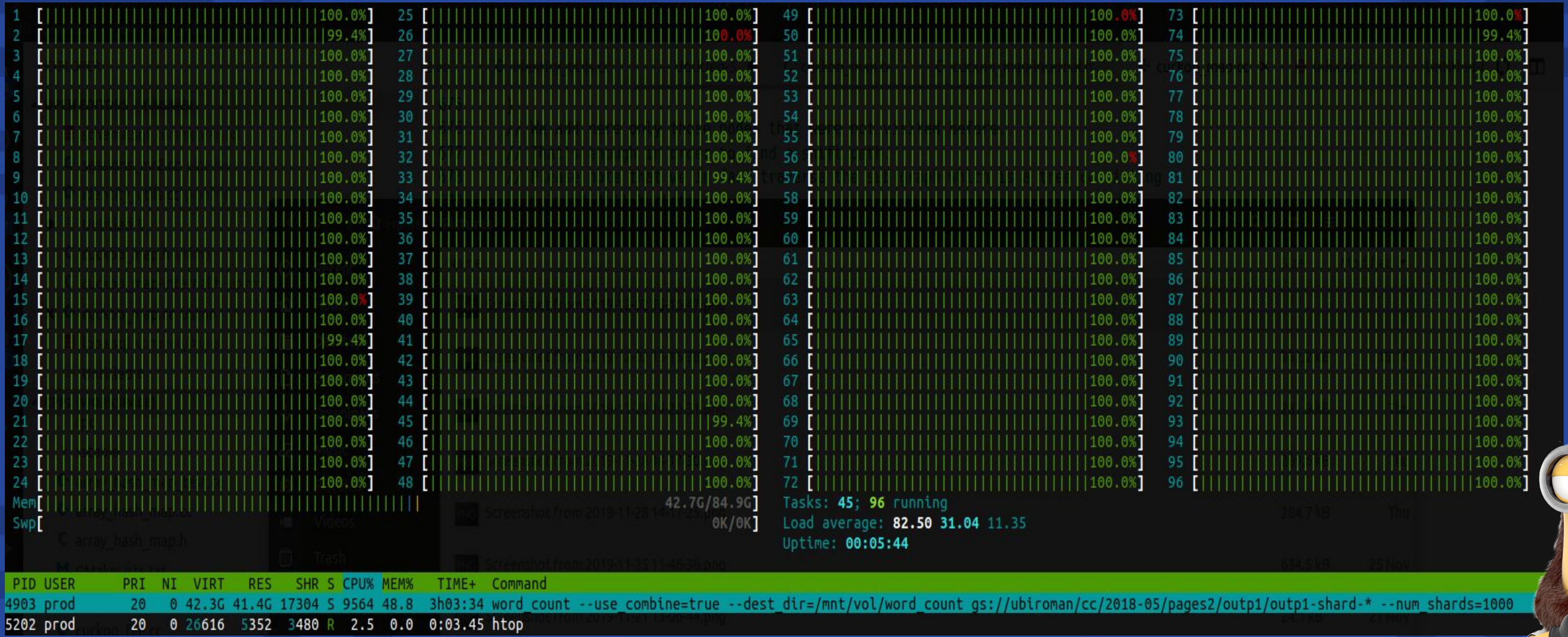
96-core (*preemptable) node

- **The task:** process web pages, count words frequency.
- **Dataset: 351GB** of gzipped pages. Compiled from CommonCrawl sample (18.5 billion lines of text).
- **2 test setups**: with combiner (original) and without.

Word Count Benchmark

	100 beam 1-cpu nodes	96-core (*preemptable) node
Combiner *(original)*	56 minutes/675 cents	18 minutes/25 cents
No Combiner	160 minutes/2032 cents	29 minutes/34 cents

Word count in action



Mapreduce as glorified Join/GroupBy

- Not a novel algorithm.
- A paradigm, computational framework. Brilliant engineering solution.
- Fits for big-data problems, with repeatable, parallelizable computations without lots of inter-dependencies.
- Prerequisite: design your problem as map and reduce.
  If it fits - implement your “map” and “reduce” operators and run the pipeline.

WordCount mapper

For each word : WebDoc

Write(shard_id = hash(word) % N, word, 1)

```
// GAIA

class WordSplitter {
 public:
   WordSplitter() : re_("(\\pL+)") {}

   void Do(string line, DoContext<WordCount>* cntx) {
     string word;
     while (RE2::FindAndConsume(&line, *re_, &word)) {
       cntx->Write(WordCount{word, 1});
     }
   }
}
```

WordCount reducer

For each word, stream<values> : Shard

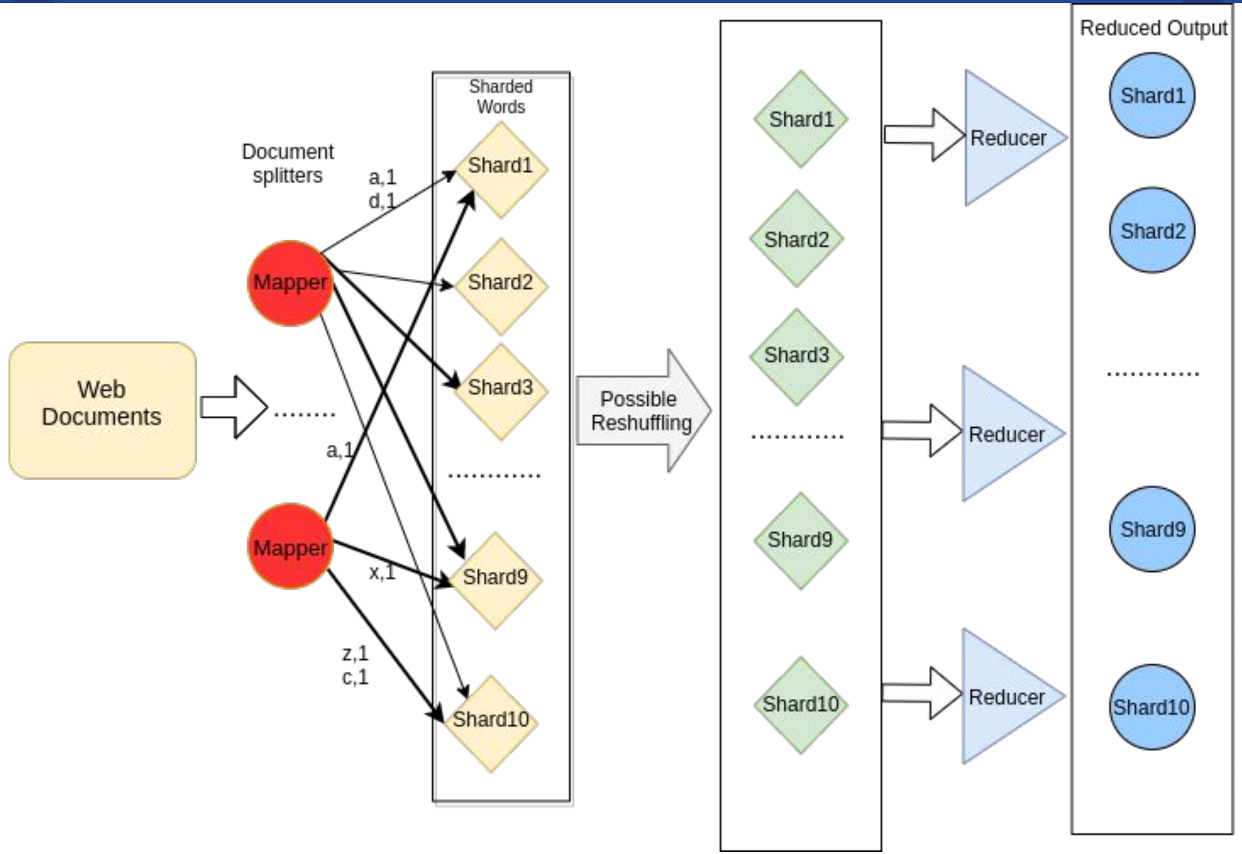
Total_sum = Sum(stream)

Output(word, total_sum)

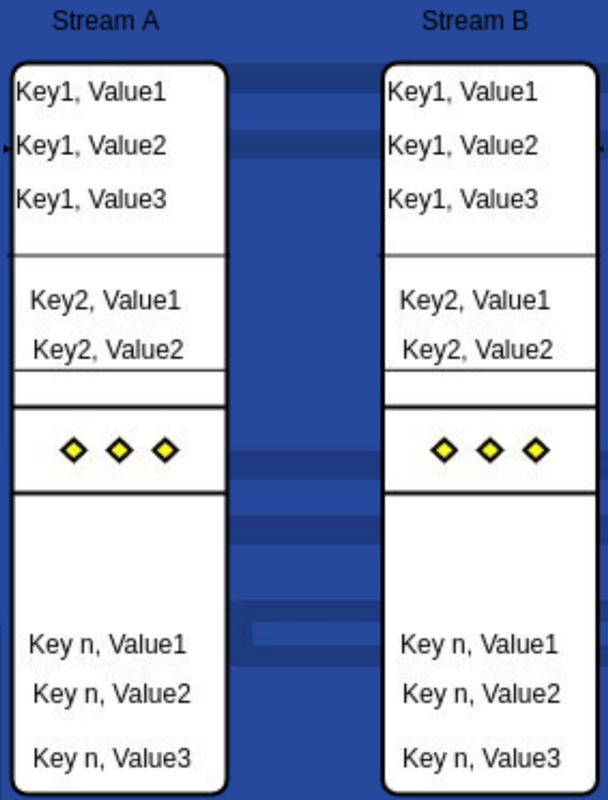
```
class WordGroupBy {
public:
  void OnWordCount(WordCount wc, DoContext<WordCount>* context) {
    word_table_[wc.string] += wc.count;
  }
  // We hold the whole shard in memory
  void OnShardFinish(DoContext<WordCount>* cntx) { word_table_.Flush(cntx); }
  }
  private:
   WordCountTable word_table_;
  };
};
```

```
  void OnWordCount(string key, Stream<uint64_t> counts,
                   DoContext<string>* context) {
    uint64_t sum = 0;
    while (!counts.empty()) { sum += counts.Next(); }
    context->Write(std::format("{}:{}", key, sum));
  }
```

WordCount graph



Shuffling step: Merging micro-shards by key



Mapreduce Anatomy

- Mapping phase
  - Transform each record
  - Horizontally scalable
  - Independent
  - **Mapper output is partitioned into K micro-shards files**
  - ∗Possible combining
- Shuffling
  - Gather mapper output from multiple machines/workers.
  - Reshuffle and merge into K shards, possibly sort them
  - Possibly distribute into Reducer workers for further processing.
- Reducer phase
  - Load shard *I* (possibly from several sources).
  - Iterate and join per common key.
  - Apply Reduce/Join/GroupBy and output.

Bind Everything

```
// Mapper phase
PTable<WordCount> intermediate_table =
    pipeline->ReadText("inp1", inputs).Map<WordSplitter>("word_splitter", db);
intermediate_table.Write("word_interim", pb::WireFormat::TXT)
    .WithModNSharding(FLAGS_num_shards,
                      [](const WordCount& wc) { return base::Fingerprint(wc.word); })
    .AndCompress(pb::Output::ZSTD, FLAGS_compress_level);

// GroupBy phase
PTable<WordCount> word_counts = pipeline->Join<WordGroupBy>(
    "group_by", {intermediate_table.BindWith(&WordGroupBy::OnWordCount)});
word_counts.Write("wordcounts", pb::WireFormat::TXT)
    .AndCompress(pb::Output::ZSTD, FLAGS_compress_level);
```

Classic approach: multiple IO Passes per stream

- Mapping Phase: Input Read + Write (partitioning)
- **Shuffling: Partition Read + Sort + Write merge-sorted shards.**
- Reduce Phase: Streams Read + Write (output)

Total: 3 I/O passes:reads & writes.

GAIA Philosophy

- Less I/O usage - more performance
  - Provides weaker guarantees, less resilient to hw failures.
  - Requires more control from a pipeline developer
- Fully utilize all the CPU and RAM of a single node.
- WYBWYR: What You Build is What You Run
  - No pipeline optimizer.
  - Shard processing is pushed to pipeline user-code.
  - No shuffle phase: 2 I/O passes.

What now?

- Try it!
- Needs better documentation, so ask questions and I will add as much as possible.
- Has few examples
- Needs traction with the community.

Bonus question

What is common between this lecture and the next one?

Appendix mrgrep

```
class Grepper {
 public:
  Grepper(string reg_exp) : re_(reg_exp) {
   CHECK(re_.ok());
  }

  void Do(string val, DoContext<string>* context) {
   if (RE2::PartialMatch(val, re_)) {
     auto* raw = context->raw();
     cout << raw->input_file_name() << ":" << raw->input_pos() << " " << val << endl;
   }
  }
 private:
  RE2 re_;
};
```

https://github.com/romange/gaia/blob/master/examples/mrgrep.cc

Appendix mrgrep

```
int main(int argc, char** argv) {
  PipelineMain pm(&argc, &argv);
  auto inputs = ...

  Pipeline* pipeline = pm.pipeline();
  StringTable st = pipeline->ReadText("read_input", inputs);
  StringTable no_output = st.Map<Grepper>("grep", FLAGS_e);
  no_output.Write("null", pb::WireFormat::TXT);

  LocalRunner* runner = pm.StartLocalRunner("/tmp/");
  pipeline->Run(runner);
  return 0;
}
```