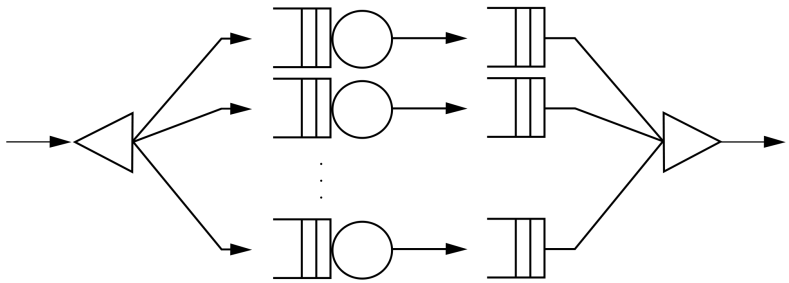
Large-Scale Merging of Histograms using Distributed In-Memory Computing

Jakob Blomer

CHEP 2015, Okinawa

Merging of Histograms

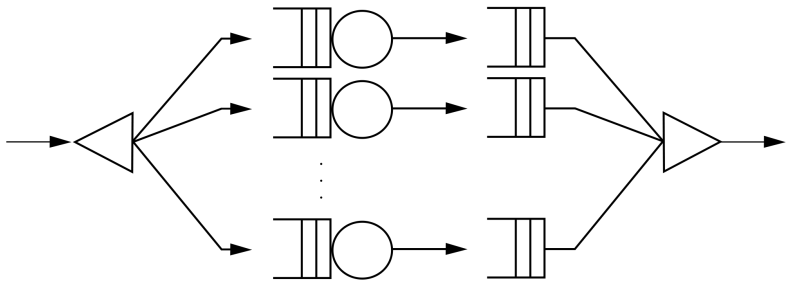
- HEP analysis tasks follow a fork-join workflow



- Task cannot run faster than initialization plus final merging (Amdahl's law)

Merging of Histograms

- HEP analysis tasks follow a fork-join workflow



- Task cannot run faster than initialization plus final merging (Amdahl's law)

Typical nasty case: output of thousands of histograms

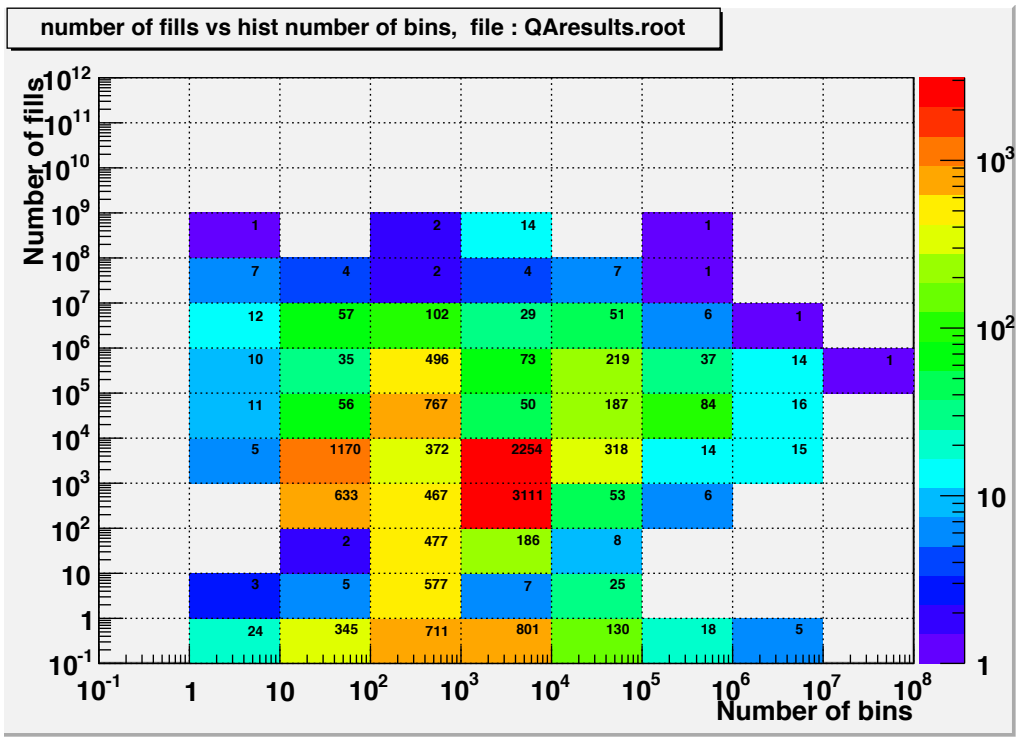
```
for all events do
  for all jets do
    for all particles with properties do
      value = Correlation(...)
      FillHistogram(histogram, value)
      ...

for all histograms
  Merge(histogram)
```

Each node produces gigabytes of data to be sent to the merger

Example of a Large Collection of Histograms

Histograms in ALICE data quality assurance:



Source: Brun

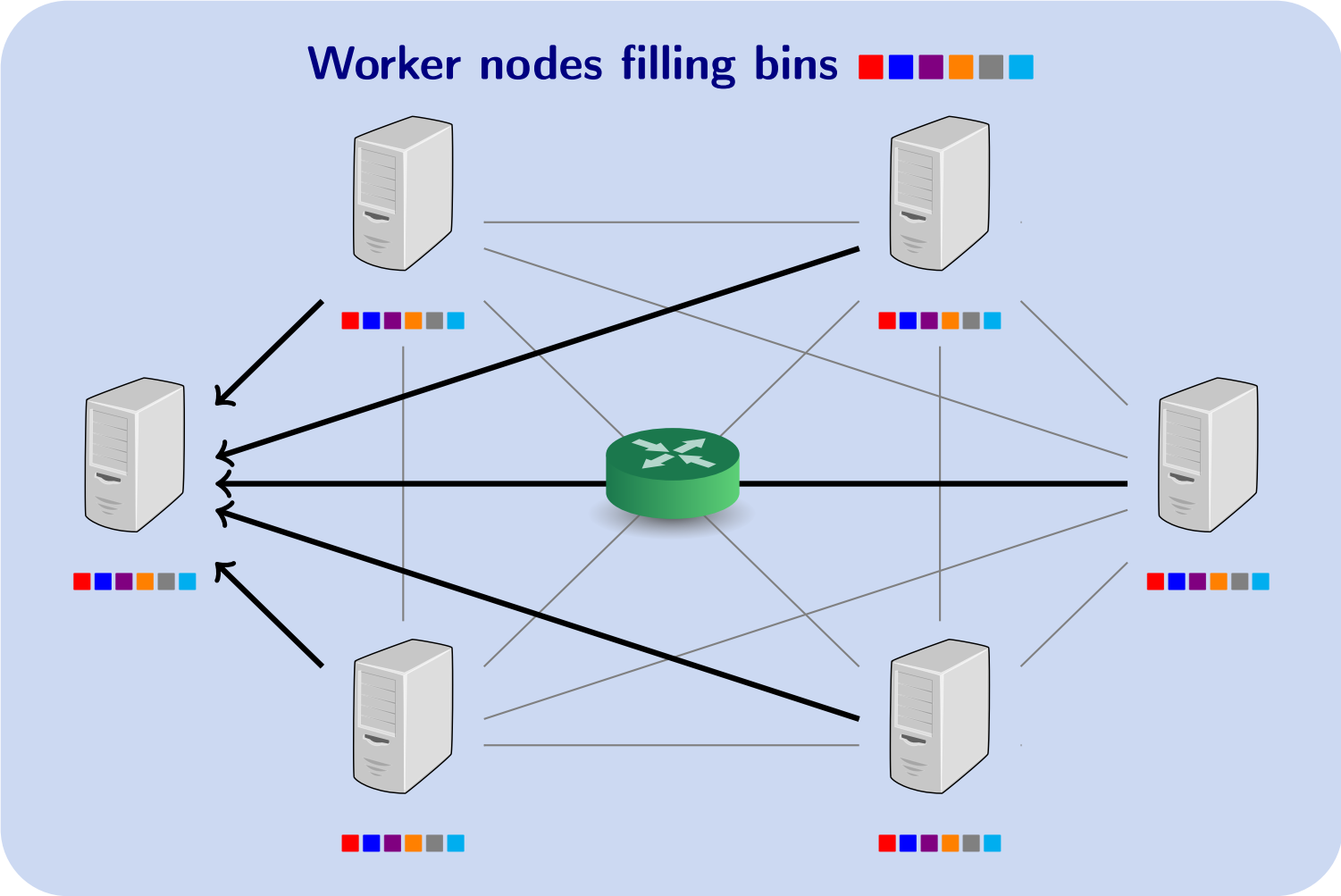
Scale per analysis worker:

- 10^2 to 10^4 histograms
- Up to 10^6 [3d] bins/histogram
- Millions of fills per second

Overall:

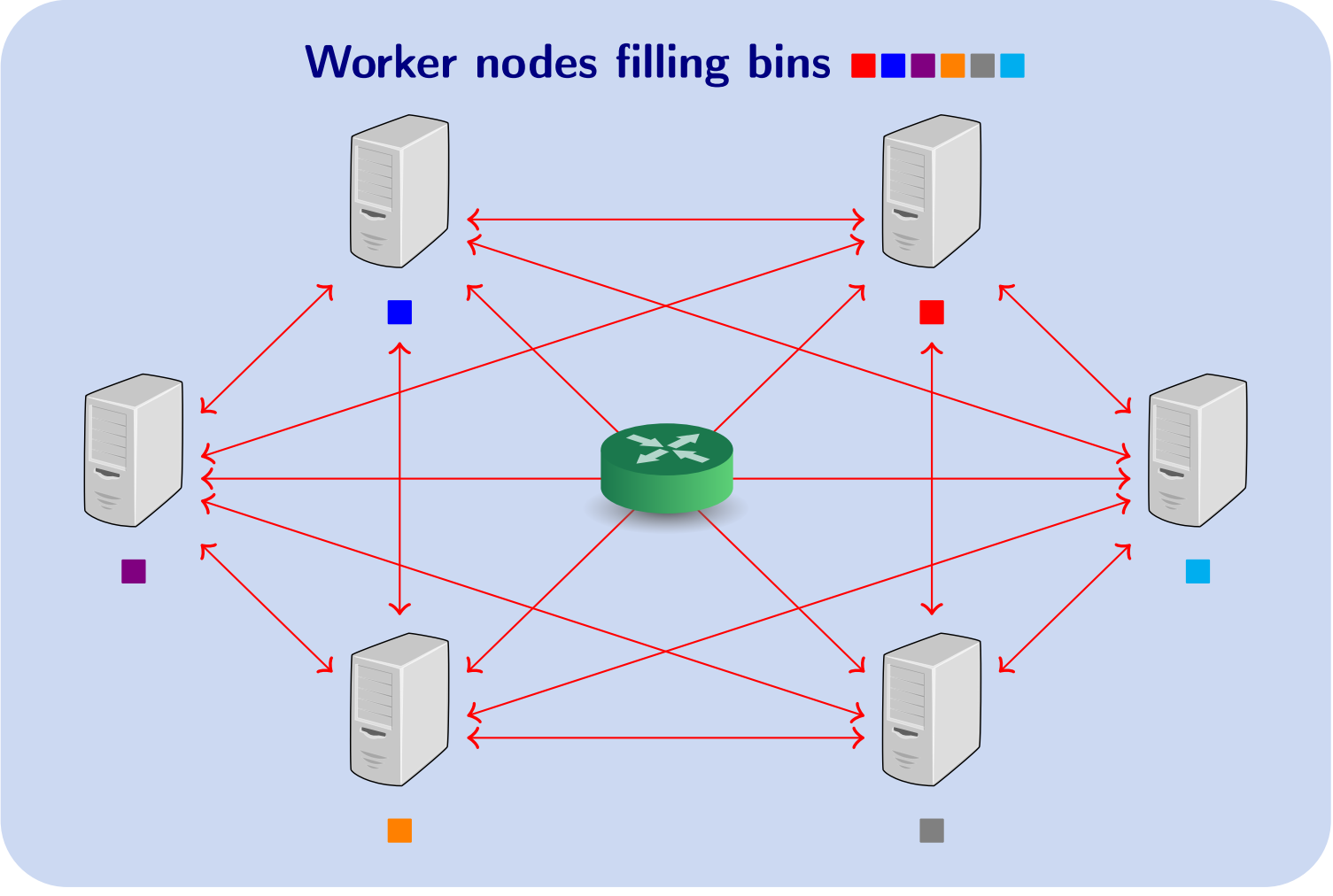
- ≈ 12 000 histograms
- ≈ 19 million non-zero bins
- Can take hours on the Grid to merge

Histogram Merging: A System's Perspective



Merging is asymmetric: n senders, 1 receiver

Histogram Merging: A Peer to Peer Approach



Instead: shuffle on a **distributed shared memory (key-value store)**

Chosen Distributed Key-Value Store: RAMCloud

- Developed since 2011 at Stanford University
- MIT license
- Aims at production grade software (e. g. fully unit-tested)
- ≈100 k lines of C++ code
- Easy to deploy: compiles on SL6, in 1–2 daemon processes.
- Highly performance-tuned:
  an order of magnitude smaller latency than other key-value stores
  → high throuput

We'll use it here out of its originally intended context (large web services)

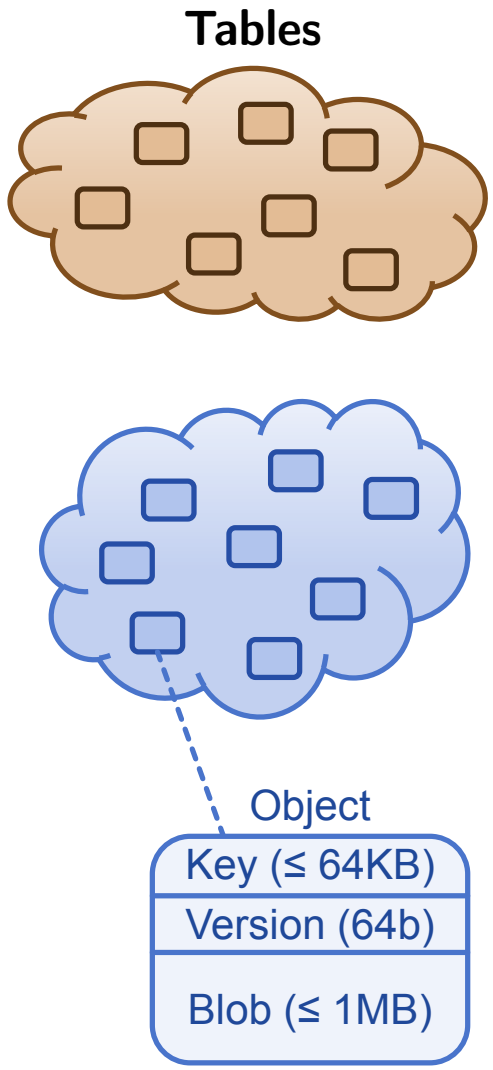
RAMCloud Data Model

Entities

- Table
- Object (row): Key + Value + Version
- Tablet: partition of a table (block of rows)

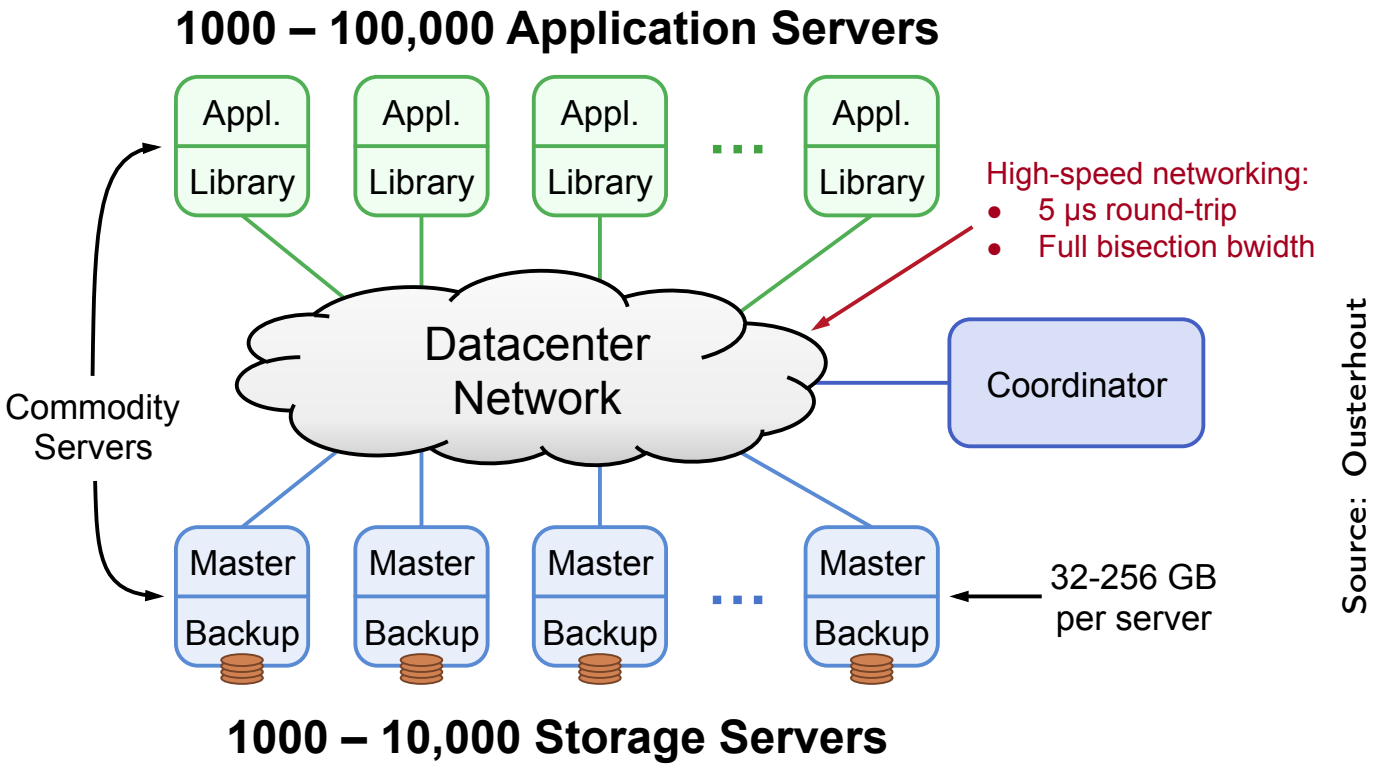
Operations

- `read(tableId, version)` → blob, version
- `write(tableId, key, value)` → version
- `delete(tableId, key)`
- `cwrite(tableId, key, value, version)` → version
  conditional write, simplifies concurrency control
- Atomic summation (on integers and doubles)
- Enumerate objects in a table
- Secondary Indices



Source: Ousterhout

RAMCloud – System Overview

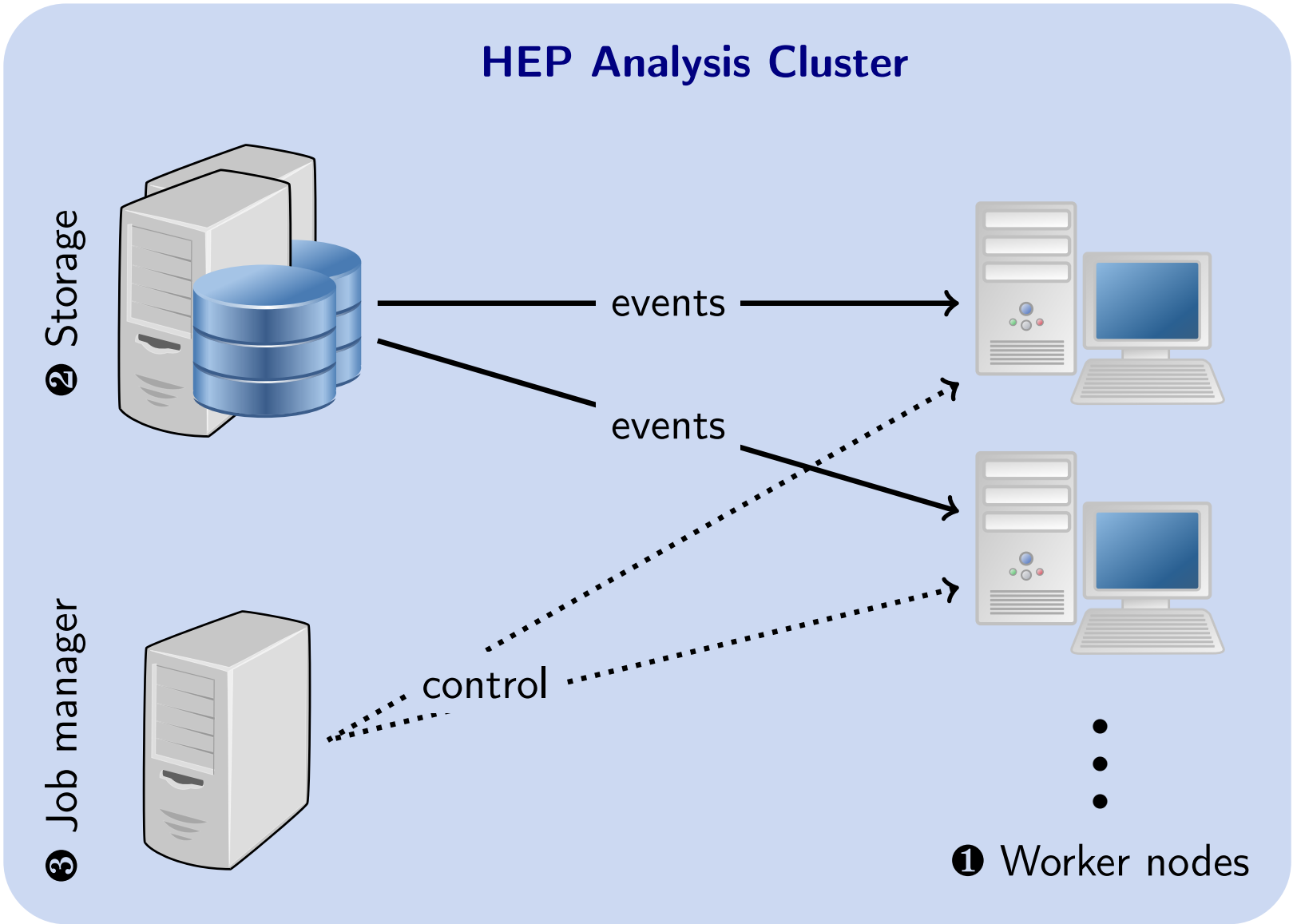


Key parameters:

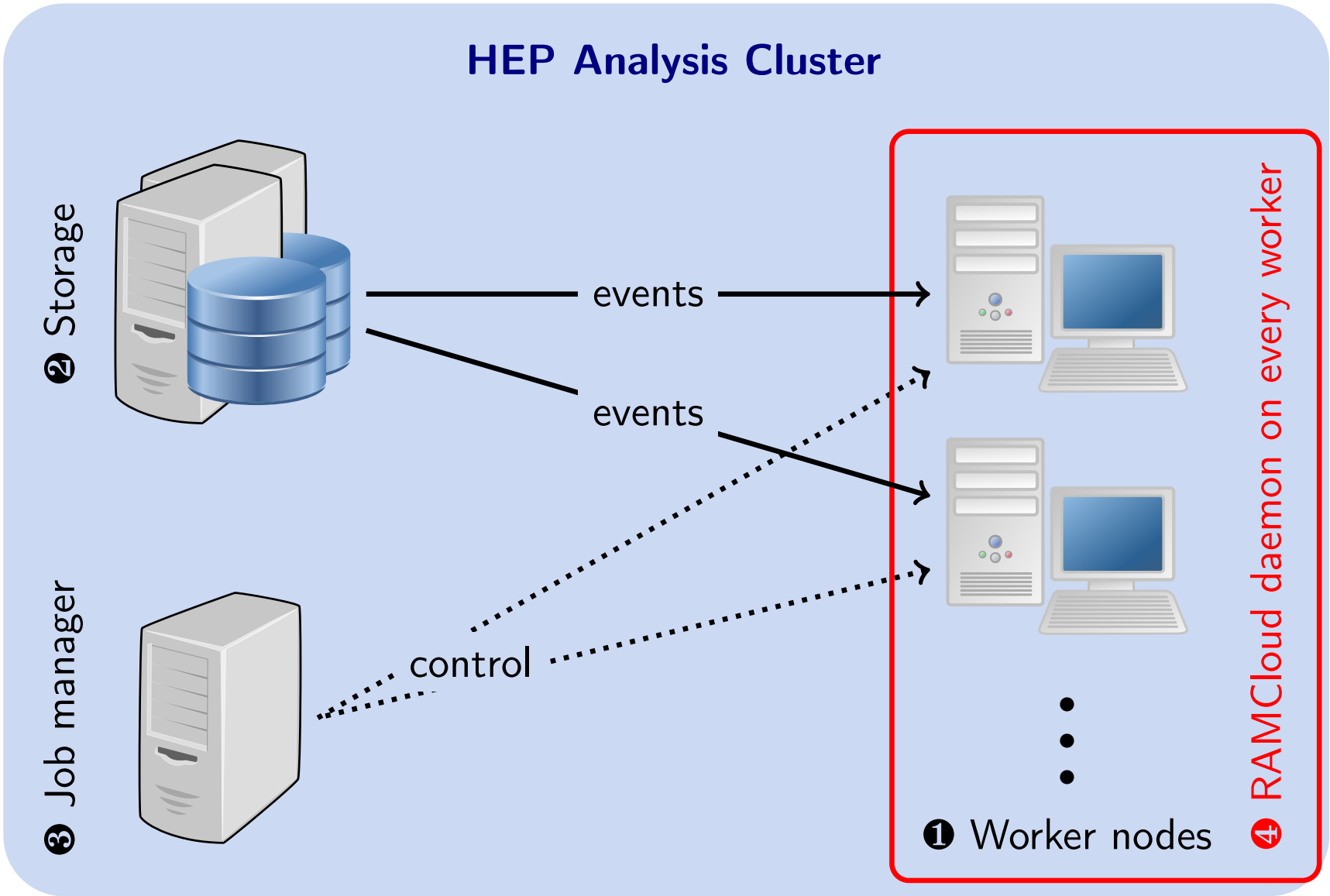
- Distributed, fast, and durable key-value store with atomic operations and secondary indexes
- All data guaranteed to be in **memory**, thus up to 1M ops/sec/server
- Reliable, *k* **replicas on disk** (buffered log, no disk write during store)
- Extra **low latency**: 5 µs to read, 15 µs to write (InfiniBand)

Publications: CACM'11 | SOSP'11 | HotOS'13 | FAST'14 | USENIX ATC 14

Worker Node Entanglement through RAMCloud



Worker Node Entanglement through RAMCloud



Histogram Merge Facility: Implementation

Histograms in RAMCloud

- Single table, evenly distributed over worker nodes
- Key:
  histogram ID $\oplus$ bin number
  (sparse, lazy bin creation)
- Value: 2 x IEEE-745 double
  (bin content and bin error)

Merging

1. "Merging": all nodes send the bin contents to other node's RAMCloud daemons; at the same time, all nodes receive bin contents and perform the additions.
2. "Readout": enumerate all bins on all nodes

ALICE QA Monitoring Benchmark

- $\approx$ 12 000 histograms, $\approx$ 19 million bins
- Simplified setup: only merging,
  same large ROOT file on every node
- No disk I/O

Hardware

- 29 physical nodes, Scientific Linux 6
- 8 core Xeon E5450, 16 GB RAM
- 1 GbE (10 GbE backplane)

Histogram Merge Facility: Implementation

Histograms in RAMCloud

- Single table, evenly distributed over worker nodes
- Key:
  histogram ID $\oplus$ bin number
  (sparse, lazy bin creation)
- Value: 2 x IEEE-745 double
  (bin content and bin error)

Merging

1. "Merging": all nodes send the bin contents to other node's RAMCloud daemons; at the same time, all nodes receive bin contents and perform the additions.
2. "Readout": enumerate all bins on all nodes

ALICE QA Monitoring Benchmark

- $\approx$ 12 000 histograms, $\approx$ 19 million bins
- Simplified setup: only merging, same large ROOT file on every node
- No disk I/O

Hardware

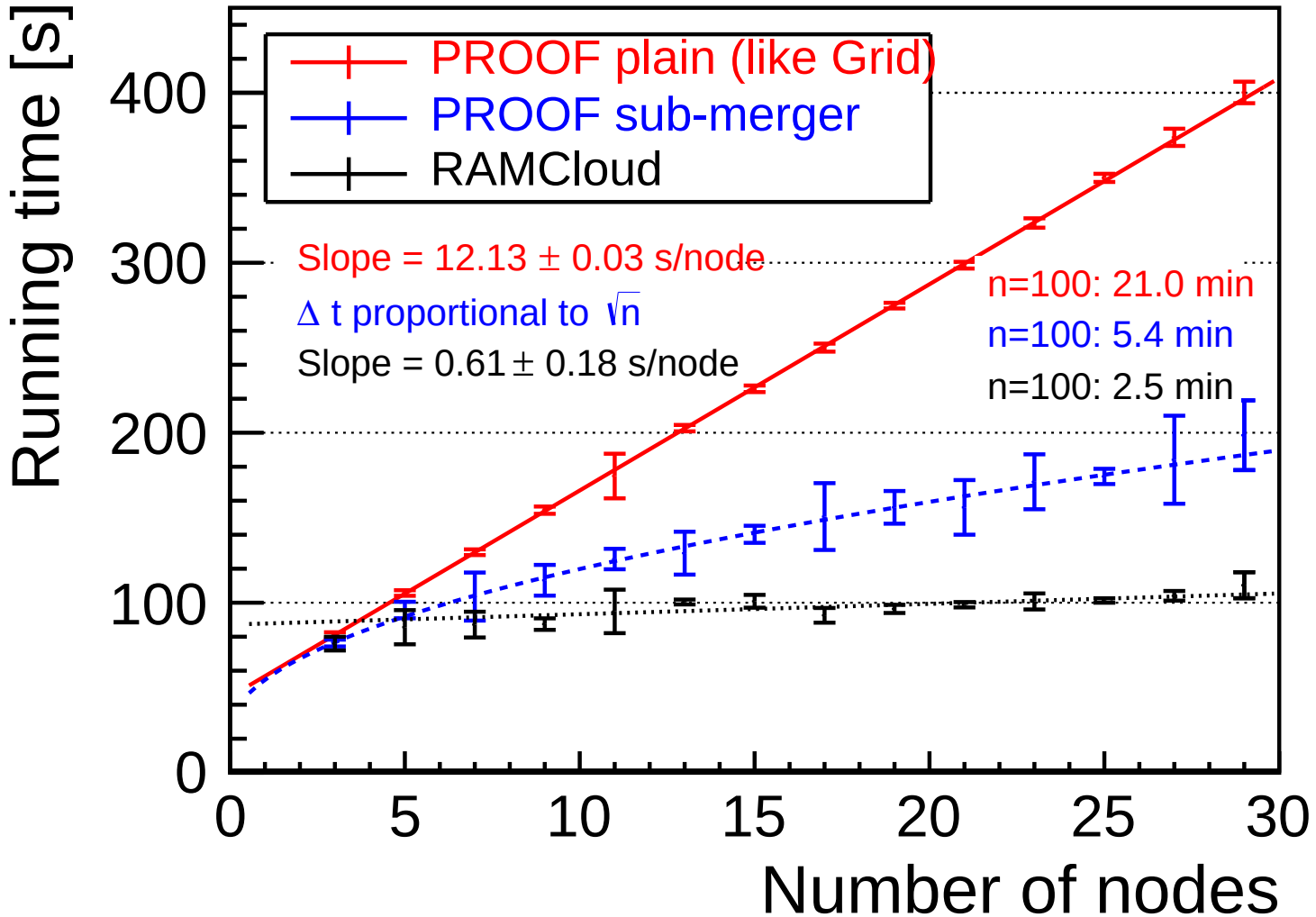
- 29 physical nodes, Scientific Linux 6
- 8 core Xeon E5450, 16 GB RAM
- 1 GbE (10 GbE backplane)

Results

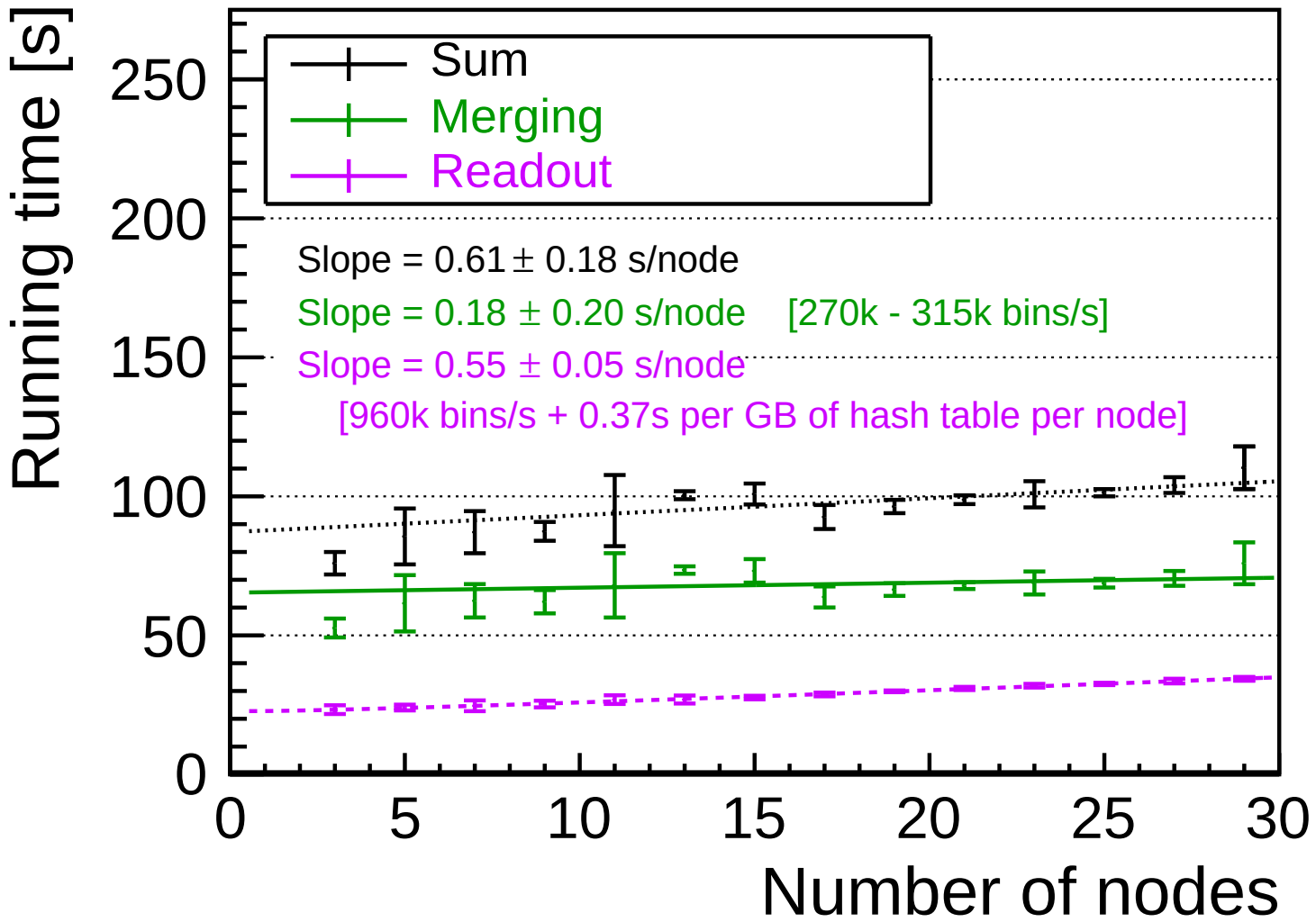
- Merging largely independent of number of workers
- Processes close to 300 k bins per second per node
- 400 % to 500 % load / node

Comparison with PROOF[1]

All PROOF numbers include 15 s transfer of results from master to client



[1] Parallel ROOT Facility (PROOF): `https://root.cern.ch/drupal/content/proof`


Running time [s]
250
200
150
100
50
0
0
5
10
15
20
25
30
Number of nodes
Sum
Merging
Readout
Slope = 0.61 ± 0.18 s/node
Slope = 0.18 ± 0.20 s/node [270k - 315k bins/s]
Slope = 0.55 ± 0.05 s/node
[960k bins/s + 0.37s per GB of hash table per node]

Conclusion

1. Distributed computing in HEP is not fully embarrassingly parallel
   Amdahl's law makes the non-parallel parts more and more visible

Conclusion

1. Distributed computing in HEP is not fully embarrassingly parallel
   Amdahl's law makes the non-parallel parts more and more visible

2. General purpose distributed storage tools become
   competitive to highly performance-tuned HEP tools

Conclusion

1. Distributed computing in HEP is not fully embarrassingly parallel
   Amdahl's law makes the non-parallel parts more and more visible

2. General purpose distributed storage tools become
   competitive to highly performance-tuned HEP tools

3. Opportunity to compose HEP system tools
   from larger building blocks of existing BigData software

Backup Slides

Distributed Key-Value Stores

A distributed key-value store typically

- is provided by a cluster of commodity servers
- has a simple dictionary interface:
  `PUT(<key>, <value>)` and `GET(<key>)`;
  sometimes with extensions such as
  atomic operations, transactions, secondary indexes, ...
  but no relational algebra, fuzzy queries, triggers/stored procedures
- scales horizontally: thousands or tens of thousands of nodes
- facilitates fault-tolerant storage (automatic data replication and recovery, no single point of failure)

Examples: Amazon Dynamo ▸ SOSP'07, Riak, Erlang Mnesia, redis, MemcacheDB, Ceph/RADOS ▸ Ceph'07, ...

Trend: from **ACID**[2] database systems to **BASE**[1] NoSQL systems

[2] ACID: atomic, consistent, isolated, durable; BASE: basic availability, soft-state, eventually consistent

Examples of Worker Node Entanglement

Worker Node Entanglement

- Software cache
- Mergable output
  - Histograms
  - Trees
- Auxiliary Data
  - Detector description
  - Physics tables
- Event buffers

Workbench Wishlist

1. Item store for *items* $< 10\,\mathrm{MB}$
2. Interface: `put` and `get` and "simple" arithmetic (e. g. `add`)
3. Collaboratively provided
   i. e. scales with the cluster size
   provided full-bisection bandwidth
4. Small overhead per node
   $< 10\,\%$ memory,
   $< 5\,\%$ of other resources
5. 100k – 1M ops/worker node
   a few operations per job per event