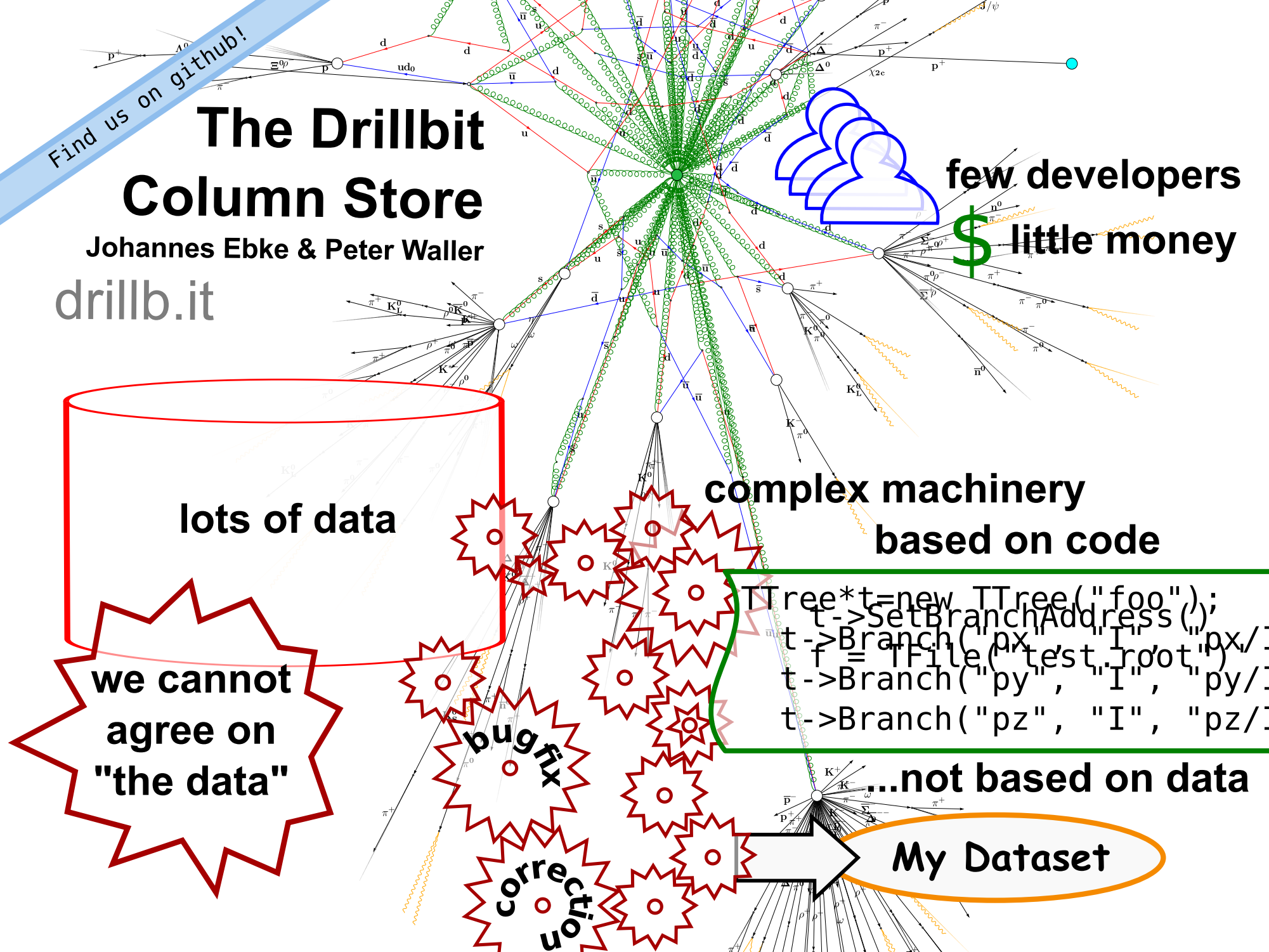
The Drillbit Column Store

Johannes Ebke[TNG] & Peter Waller[SW]

drillb.it

TNG TECHNOLOGY CONSULTING

ScraperWiki

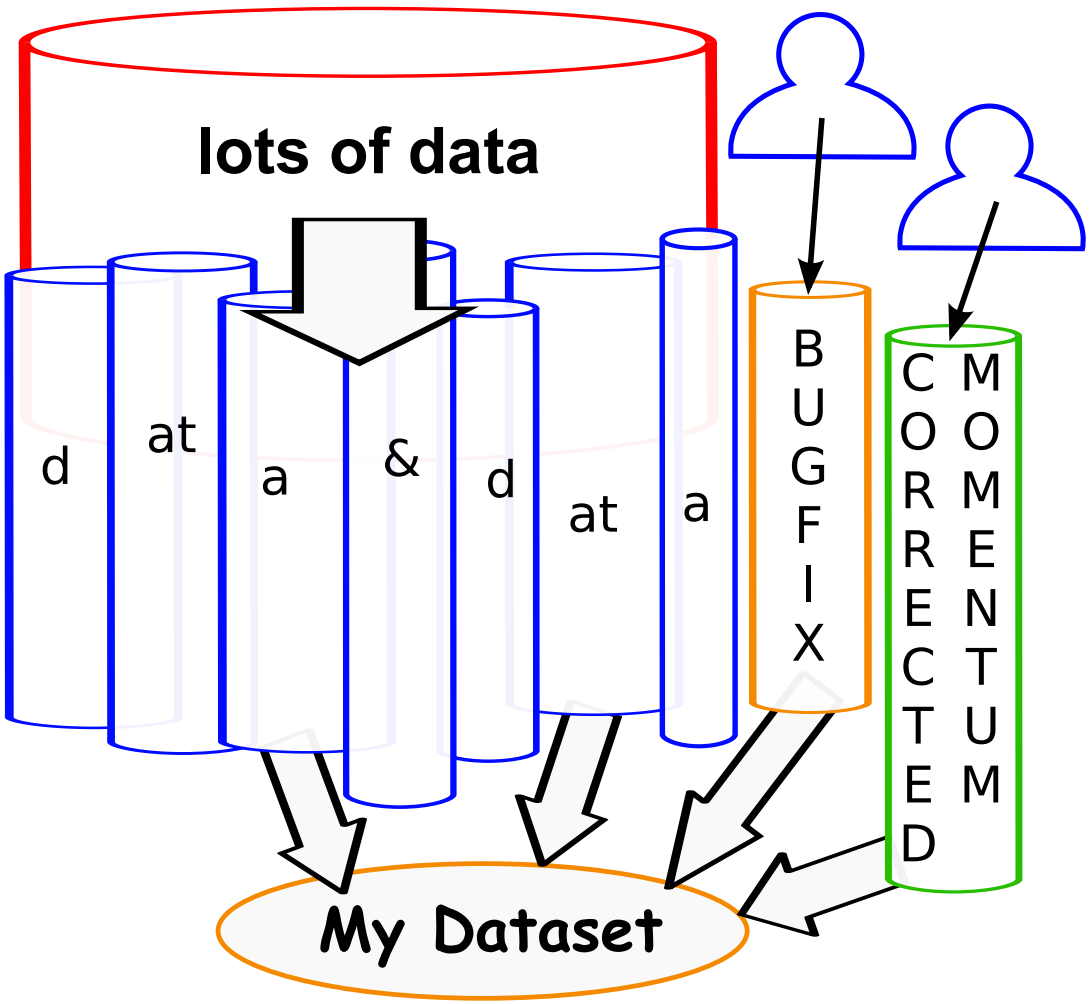

Find us on github!
The Drillbit Column Store
Johannes Ebke & Peter Waller
drillb.it
few developers
$ little money
lots of data
complex machinery
based on code
TTree*t=new TTree("foo");
t->SetBranchAddress()
t->Branch("px", "I", "px/I
f = TFile("test.root")
t->Branch("py", "I", "py/I
t->Branch("pz", "I", "pz/I
we cannot agree on "the data"
bug fix
correction
...not based on data
My Dataset

Find us on github!

The Drillbit Column Store

Johannes Ebke & Peter Waller

drillb.it



- Split large tuples into columns,
- One or more columns per variable

- Efficiently combine columns into tuples on request

- Add new columns instead of reprocessing or running manual fixes

- Reduce the immense pressure on developers to get (almost) everything right in the first go in the (re)processing

Core Requirements for Drillbit

Significant concerns:

- Speed and scalability:
  “more data analysed faster, in more ways, by more physicists”

Core Requirements for Drillbit

Significant concerns:

- Speed and scalability:
  “more data analysed faster, in more ways, by more physicists”
- Provide guarantees for original dataset equality:
  “which data are in your analysis?”

Core Requirements for Drillbit

Significant concerns:

- Speed and scalability:
  “more data analysed faster, in more ways, by more physicists”
- Provide guarantees for original dataset equality:
  “which data are in your analysis?”
- Provide reproducibility:
  “how was this data obtained from raw data?”

Core Requirements for Drillbit

Significant concerns:

- Speed and scalability:
  “more data analysed faster, in more ways, by more physicists”
- Provide guarantees for original dataset equality:
  “which data are in your analysis?”
- Provide reproducibility:
  “how was this data obtained from raw data?”
- Organize the Chaos of “last-minute” corrections:
  “shareable, versioned columns”

Core Requirements for Drillbit

Significant concerns:

- Speed and scalability:
  "more data analysed faster, in more ways, by more physicists"
- Provide guarantees for original dataset equality:
  "which data are in your analysis?"
- Provide reproducibility:
  "how was this data obtained from raw data?"
- Organize the Chaos of "last-minute" corrections:
  "shareable, versioned columns"

This Talk

Low-level techniques for columnar data handling

Poster

Poster "A novel dynamic event data model using the Drillbit column store" deals more with guarantees and reproducibility

Outline

1. Motivation / Why would we benefit from yet another data format?
2. Inspiration / How can we build it?
3. Validation / How good is it?

Outline

Technical Basis

Inspiration by a Google Paper:

Dremel: Interactive Analysis of Web-Scale Datasets

@ `http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R29.pdf`

Technical Basis

Inspiration by a Google Paper:

Dremel: Interactive Analysis of Web-Scale Datasets

@ `http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R29.pdf`

- Impressive performance numbers: $\mathbf{24 \times 10^9}$ nested records with 530 fields each in a datacenter with 2900 nodes lets aggregation queries run in $\approx$ **2 seconds**.
- Scales to petabytes of data.

Technical Basis

Inspiration by a Google Paper:

Dremel: Interactive Analysis of Web-Scale Datasets

@ `http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R29.pdf`

- Impressive performance numbers: $\mathbf{24 \times 10^9}$ nested records with 530 fields each in a datacenter with 2900 nodes lets aggregation queries run in $\approx$ **2 seconds**.
- Scales to petabytes of data.

How do they do it?

- Speed: columnar data layout + parallel processing of data
- Structure: novel columnar storage representation for nested records

Columnar Storage Representation

Example of a data structure to be saved:

```
repeated message event {
 int32 run_number;
 float32 total_energy;
 repeated message electrons {
   double pt, eta, phi;
   repeated double hits_timing;
...
```

Columnar Storage Representation

Example of a data structure to be saved:

```
repeated message event {
 int32 run_number;
 float32 total_energy;
 repeated message electrons {
   double pt, eta, phi;
   repeated double hits_timing;
...
```

Resulting columns:

```
event.run_number
event.total_energy
event.electrons.pt
event.electrons.eta
event.electrons.phi
event.electrons.hits_timing
```

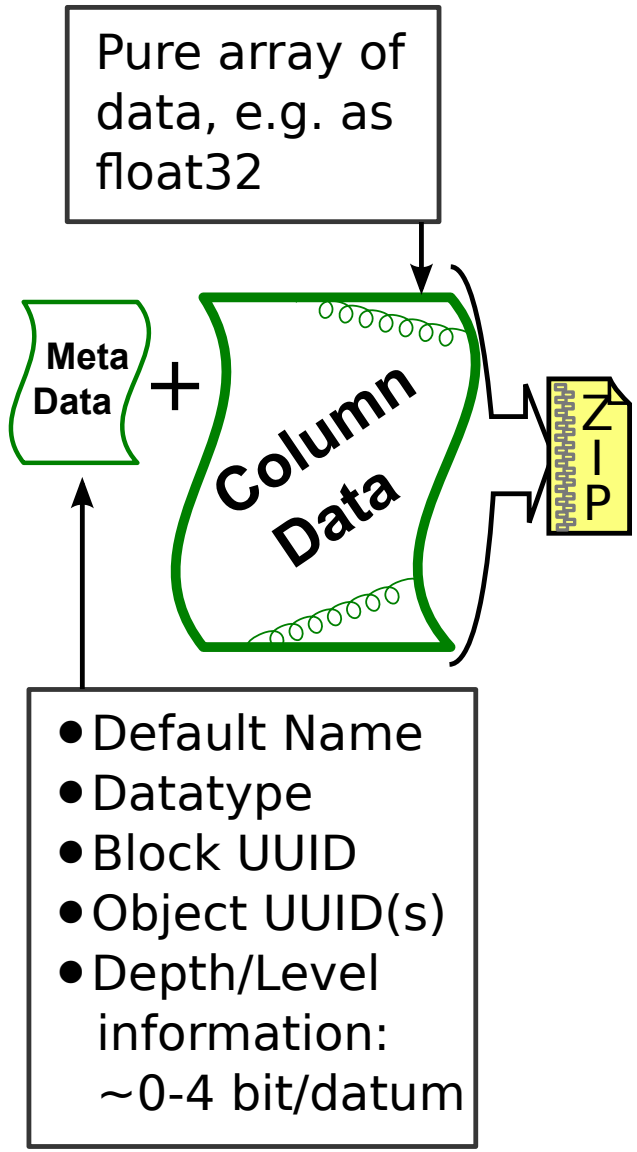
Columnar Storage Representation

Example of a data structure to be saved:

```
repeated message event {
 int32 run_number;
 float32 total_energy;
 repeated message electrons {
   double pt, eta, phi;
   repeated double hits_timing;
...
```

Resulting columns:

```
event.run_number
event.total_energy
event.electrons.pt
event.electrons.eta
event.electrons.phi
event.electrons.hits_timing
```



Outline

Evaluation on real data

Results of initial implementation of the Dremel Algorithm:

Evaluation on real data

Results of initial implementation of the Dremel Algorithm:

- Gain of -20% **in size** with identical compression on average over the set of all ATLAS ntuples

Evaluation on real data

Results of initial implementation of the Dremel Algorithm:

- Gain of -20% **in size** with identical compression on average over the set of all ATLAS ntuples
- Processing speed with a TTree-compatible interface on par with native TTree

Evaluation on real data

Results of initial implementation of the Dremel Algorithm:

- Gain of -20% **in size** with identical compression
  on average over the set of all ATLAS ntuples
- Processing speed with a TTree-compatible interface
  on par with native TTree
- Skimming is easy, Slimming and thinning are literally trivial

Evaluation on real data

Results of initial implementation of the Dremel Algorithm:

- Gain of -20% **in size** with identical compression
  on average over the set of all ATLAS ntuples
- Processing speed with a TTree-compatible interface
  on par with native TTree
- Skimming is easy, Slimming and thinning are literally trivial

Current Status:

- ROOT $\rightarrow$ Columns $\rightarrow$ ROOT round-trip for **all basic data types**
  and their (nested) std::vectors
- Implemented in $\approx$ 2**kLOC C++**
- `TTree * t = DrillbitTree(files);` // functional in `TSelector`

Evaluation on real data

Results of initial implementation of the Dremel Algorithm:

- Gain of -20% **in size** with identical compression
  on average over the set of all ATLAS ntuples
- Processing speed with a TTree-compatible interface
  on par with native TTree
- Skimming is easy, Slimming and thinning are literally trivial

Current Status:

- ROOT $\rightarrow$ Columns $\rightarrow$ ROOT round-trip for **all basic data types**
  and their (nested) std::vectors
- Implemented in $\approx$ 2**kLOC C++**
- `TTree * t = DrillbitTree(files);` // functional in `TSelector`

Other possible gains:

- Possibility to do cache-efficient computations on subset of variables
- Many opportunities to use multiple cores

Ideal World - A Columnar Event Data Model

Ideal World - A Columnar Event Data Model

Ideal World - A Columnar Event Data Model

- ♡ **Version tree of event objects** and variables combined with links to tools that calculate these

Ideal World - A Columnar Event Data Model

- **Version tree of event objects** and variables combined with links to tools that calculate these
- Variables and last-minute corrections on objects are **shared as data columns** instead of having to reprocess the whole data every time

Ideal World - A Columnar Event Data Model

- ♡ **Version tree of event objects** and variables combined with links to tools that calculate these
- Variables and last-minute corrections on objects are **shared as data columns** instead of having to reprocess the whole data every time
- Example tree entry that could be shared between analysts:
  - `hww/muon@r2 := (perf_muon@r2321 || isolation := HWWMuonIsolationTool@v00-02-03(perf_muon@r2321))`
  - `ebke/muon/isolation@r2 := git://../my_tool@r2312(hww/muon@r2, event_info/vertices@r2)`

Ideal World - A Columnar Event Data Model

- ♡ **Version tree of event objects** and variables combined with links to tools that calculate these
- Variables and last-minute corrections on objects are **shared as data columns** instead of having to reprocess the whole data every time
- Example tree entry that could be shared between analysts:
  - `hww/muon@r2 := (perf_muon@r2321 || isolation := HWWMuonIsolationTool@v00-02-03(perf_muon@r2321))`
  - `ebke/muon/isolation@r2 := git://../my_tool@r2312(hww/muon@r2, event_info/vertices@r2)`
- ↓ `get_data --variables hww@r42 --datasets hww@r42`
  - → retrieves ∼few GB ntuple to analyse

Challenges Ahead

Back to reality - possible short-term actions:

- Use columnar datasets as a transparent intermediate storage format from which to generate custom ntuples

Challenges Ahead

Back to reality - possible short-term actions:

- Use columnar datasets as a transparent intermediate storage format from which to generate custom ntuples
- Consider setting up a columnar dataset pool for an analysis group, e.g. publishing corrections as new columns so not everyone has to run the correction code

Challenges Ahead

Back to reality - possible short-term actions:

- Use columnar datasets as a transparent intermediate storage format from which to generate custom ntuples
- Consider setting up a columnar dataset pool for an analysis group, e.g. publishing corrections as new columns so not everyone has to run the correction code
- Use columnar storage in new experiment software or analysis projects

Challenges Ahead

Back to reality - possible short-term actions:

- Use columnar datasets as a transparent intermediate storage format from which to generate custom ntuples
- Consider setting up a columnar dataset pool for an analysis group, e.g. publishing corrections as new columns so not everyone has to run the correction code
- Use columnar storage in new experiment software or analysis projects
- Set up data versioning for reproducibility

Summary

Summary

Summary:

- `central data + distributed code` might not be as good as `distributed data + distributed code`

Summary

Summary:

- `central data + distributed code` might not be as good as `distributed data + distributed code`
- The Dremel data representation published by Google is great for our use case, and would provide a solid and simple foundation for a strong and future-proof architecture

Summary

Summary:

- `central data + distributed code` might not be as good as `distributed data + distributed code`
- The Dremel data representation published by Google is great for our use case, and would provide a solid and simple foundation for a strong and future-proof architecture
- We believe collaboration on version-controlled columns has the potential to make physicists less unhappy

Summary

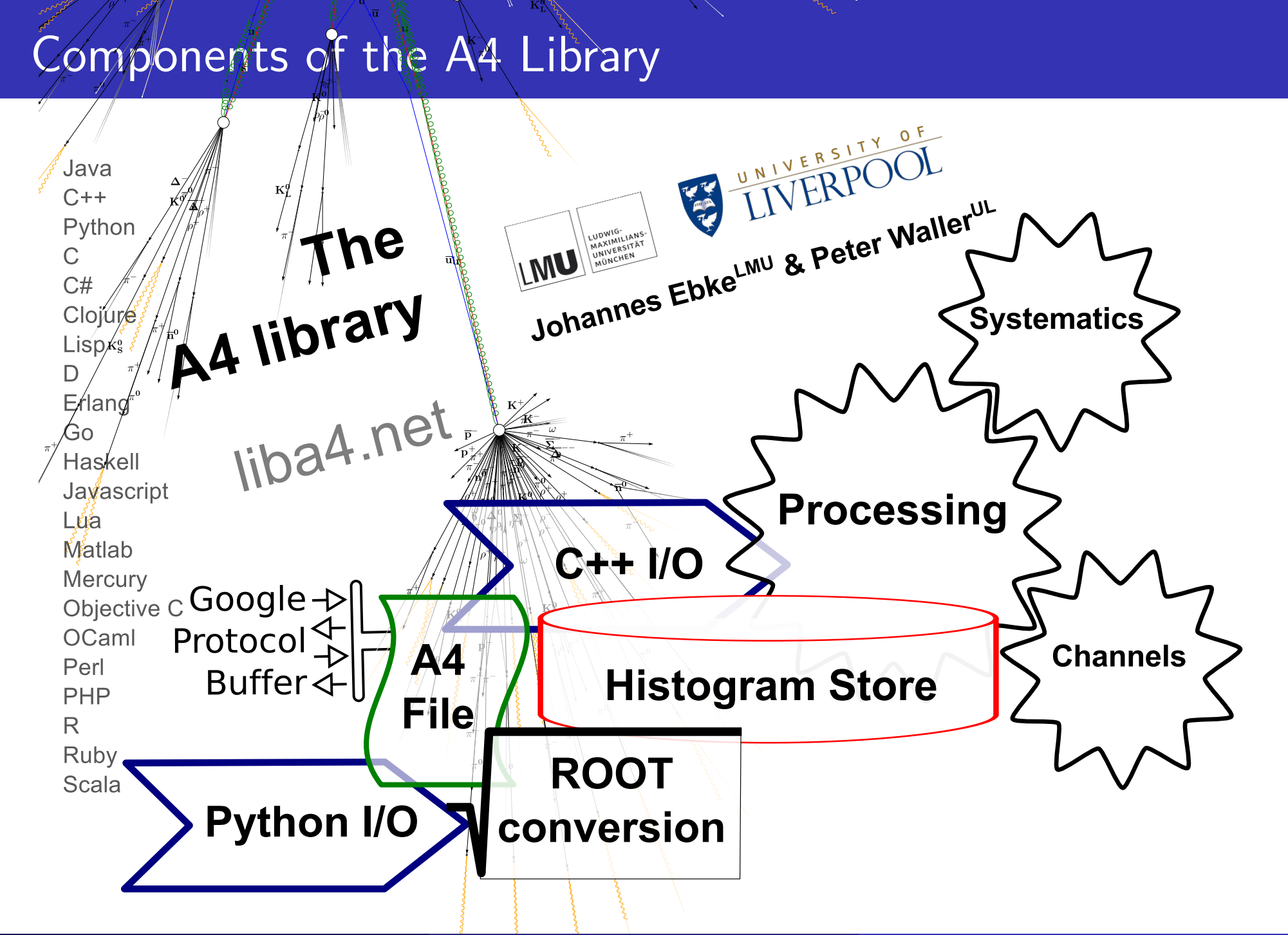
Thank you for your attention!
Questions or Comments welcome.

Summary:

- `central data + distributed code` might not be as good as `distributed data + distributed code`
- The Dremel data representation published by Google is great for our use case, and would provide a solid and simple foundation for a strong and future-proof architecture
- We believe collaboration on version-controlled columns has the potential to make physicists less unhappy

Bonus Slides

Components of the A4 Library



Protocol Buffer Message Format

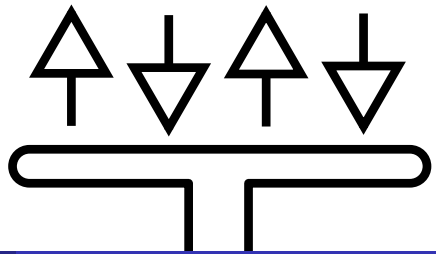
```
message Lepton {
 optional double pt = 1;
 optional double eta = 2;
 optional double phi = 3;
 optional int32 charge = 4;
}
message PhysicsEvent {
 optional int32 run_number = 1;
 optional int32 event_number = 2;
 repeated Lepton electrons = 5;
 repeated Lepton muons = 6;
}
```

Example of a protobuf message definition for a Physics event

```
PhysicsEvent e;
e.set_run_number(105200);
Lepton * l = e.add_muons();
l->set_pt(23.2);
l->pt() == 23.2;
```

Example of C++ code using the compiled version of the message definition

⇒ Separation of Data structure definition and Serialization Code



The A4 file and IO Library

- Protobuf messages are not self-describing

The A4 file and IO Library

- Protobuf messages are not self-describing
  – but the `descriptions` can be stored in protobuf messages!

The A4 file and IO Library

- Protobuf messages are not self-describing
  – but the `descriptions` can be stored in protobuf messages!

$\Rightarrow$ A4 files contain `descriptions` of all contained messages and are **self-contained**.

The A4 file and IO Library

- Protobuf messages are not self-describing
  – but the `descriptions` can be stored in protobuf messages!

$\Rightarrow$ A4 files contain `descriptions` of all contained messages and are **self-contained**.

- Native support for **metadata messages**, so there can be separation of histograms by sample in one file

The A4 file and IO Library

- Protobuf messages are not self-describing
  – but the `descriptions` can be stored in protobuf messages!

$\Rightarrow$ A4 files contain `descriptions` of all contained messages and are **self-contained**.

- Native support for **metadata messages**, so there can be separation of histograms by sample in one file

$\oplus$ Can be concatenated with "cat" - merging made easy.

The A4 file and IO Library

- Protobuf messages are not self-describing
  – but the `descriptions` can be stored in protobuf messages!

$\Rightarrow$ A4 files contain `descriptions` of all contained messages and are **self-contained**.

- Native support for **metadata messages**, so there can be separation of histograms by sample in one file

$\oplus$ Can be concatenated with "cat" - merging made easy.

- Can be up to **6 times faster** than ROOT IO*

*conditions apply: You have to use $>$ 40% of the event data..

- Conversion from and to ROOT trees / histograms included

J.E., Peter Waller, 2012 *J. Phys.: Conf. Ser.* **396** 022012;
`http://arxiv.org/abs/1208.1600`

A4 Histograms and Object Store

Small selection of cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling

```
S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);
```

A4 Histograms and Object Store

Small selection of cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling

  ```
  S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);
  ```

- "Code Locality" for histograms and cutflows: Easy to modify existing analyses and to reuse histogram definitions in functions:

A4 Histograms and Object Store

Small selection of cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling

```
S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);
```

- "Code Locality" for histograms and cutflows: Easy to modify existing analyses and to reuse histogram definitions in functions:

```
void plot(ObjectStore FS, ALorentzVector v) {
  FS.T<H1>("m")("Mass")(500,0,100,"m [GeV]").fill(v.m());
  FS.T<H1>("pt")("p_T")(500,0,100,"p_T [GeV]").fill(v.pt());
}
```

A4 Histograms and Object Store

Small selection of cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling

```
S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);
```

- "Code Locality" for histograms and cutflows: Easy to modify existing analyses and to reuse histogram definitions in functions:

```
void plot(ObjectStore FS, ALorentzVector v) {
  FS.T<H1>("m")("Mass")(500,0,100,"m [GeV]").fill(v.m());
  FS.T<H1>("pt")("p_T")(500,0,100,"p_T [GeV]").fill(v.pt());
}
S.T<Cutflow>("cf").passed("initial"); // create cutflow
plot(S("initial/muon0_"), mu0); // gives initial/muon0_pt
```

A4 Histograms and Object Store

Small selection of cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling

```
S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);
```

- "Code Locality" for histograms and cutflows: Easy to modify existing analyses and to reuse histogram definitions in functions:

```
void plot(ObjectStore FS, ALorentzVector v) {
  FS.T<H1>("m")("Mass")(500,0,100,"m [GeV]").fill(v.m());
  FS.T<H1>("pt")("p_T")(500,0,100,"p_T [GeV]").fill(v.pt());
}
S.T<Cutflow>("cf").passed("initial"); // create cutflow
plot(S("initial/muon0_"), mu0); // gives initial/muon0_pt

if (mu0.pt() < 20*GeV) return; // Cut
```

A4 Histograms and Object Store

Small selection of cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling

```
S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);
```

- "Code Locality" for histograms and cutflows: Easy to modify existing analyses and to reuse histogram definitions in functions:

```
void plot(ObjectStore FS, ALorentzVector v) {
  FS.T<H1>("m")("Mass")(500,0,100,"m [GeV]").fill(v.m());
  FS.T<H1>("pt")("p_T")(500,0,100,"p_T [GeV]").fill(v.pt());
}
S.T<Cutflow>("cf").passed("initial"); // create cutflow
plot(S("initial/muon0_"), mu0); // gives initial/muon0_pt

if (mu0.pt() < 20*GeV) return; // Cut

S.T<Cutflow>("cf").passed("20GeV");
plot(S("cut1/muon0_"), mu0);
```

A4 Histograms and Object Store II

```
S.T<H1>("el_", 1, "_pt")(500,0,100,"p_T [GeV]").fill(pt);
```

A4 Histograms and Object Store II

```
S.T<H1>("el_", 1, "_pt")(500,0,100,"p_T [GeV]").fill(pt);
```

Technical details:

- Histogram pointers stored in a **custom hierarchical hash table**

A4 Histograms and Object Store II

```
S.T<H1>("el_", 1, "_pt")(500,0,100,"p_T [GeV]").fill(pt);
```

Technical details:

- Histogram pointers stored in a **custom hierarchical hash table**
- **C++11 variadic templates** allow very fast "pseudo-concatenation" of strings and numbers

A4 Histograms and Object Store II

```
S.T<H1>("el_", 1, "_pt")(500,0,100,"p_T [GeV]").fill(pt);
```

Technical details:

- Histogram pointers stored in a **custom hierarchical hash table**
- **C++11 variadic templates** allow very fast "pseudo-concatenation" of strings and numbers
- A "const char pointer constancy check" allows safe lookups by pointer value with **no string comparisons in the event loop**

A4 Histograms and Object Store II

```
S.T<H1>("el_", 1, "_pt")(500,0,100,"p_T [GeV]").fill(pt);
```

Technical details:

- Histogram pointers stored in a **custom hierarchical hash table**
- **C++11 variadic templates** allow very fast "pseudo-concatenation" of strings and numbers
- A "const char pointer constancy check" allows safe lookups by pointer value with **no string comparisons in the event loop**
- One-line initialization is skipped by a single `jmp` instruction!

A4 Processing and CLI

A4 Processing scaffold:

- Scaffold to quickly make useable analysis executables
- Handles command line arguments
- Provides multithreading
- Makes metadata available at analysis time
- Automatically does the re-running for the `systematic` function

! Due for a redesign soon to enable in-program analysis composition

A4 command line tools:

- a4copy, a4diff, a4dump, a4info, a4merge, a42root, a4results2root, a4reweight...

A4 Histograms and Object Store

Cool things that are possible:

- One-line histogram & cutflow storage, initialization, and filling
- "Code Locality" for histograms and cutflows: Easy to modify existing analyses and to reuse histogram definitions in functions
- Support for single-line `systematic` uncertainty evaluation

Technical details:

- Histogram pointers stored in a **custom hierarchical hash table**
- **C++11 variadic templates** allow very fast "pseudo-concatenation" of strings and numbers
- A "const char pointer constancy check" allows safe lookups by pointer value with **no string comparisons in the event loop**
- One-line initialization is skipped by a single `jmp` instruction!

Example Histogram code

```
S.set_weight(event_weight); // Set event weight for everything
// create, store and fill a histogram with title and axis label
S.T<H1>("m_ee")("ee-Mass")(500,0,100,"m_ee").fill(m);

// Define and use a function for plotting many histograms
void plot(ObjectStore FS, ALorentzVector v) {
      FS.T<H1>("m")("Mass")(500,0,100,"m [GeV]").fill(v.m());
      FS.T<H1>("pt")("p_T")(500,0,100,"p_T [GeV]").fill(v.pt());
}
plot(S("initial/muon0_"), mu0); // gives initial/muon0_pt
plot(S("initial/muon", 1, "_"), mu1);
plot(S("initial/dimuon_"), mu0 + mu1);

S.T<Cutflow>("cf").passed("initial"); // create cutflow
if (mu0.pt() < 20*GeV) return; // Cut
S.T<Cutflow>("cf").passed("20GeV");
plot(S("cut1/muon0_"), mu0);

if (systematic("mu_scale_up")) mu0 *= 1.1;
```

Summary and Conclusion

The A4 Library:

- A4 files are **fast self-contained protobuf containers**
- The A4 ObjectStore is a fast way to generate a lot of histograms using few lines of code
- A4 processing helps to make an analysis a nice unix command line tool

Summary and Conclusion

The A4 Library:

- A4 files are **fast self-contained protobuf containers**
- The A4 ObjectStore is a fast way to generate a lot of histograms using few lines of code
- A4 processing helps to make an analysis a nice unix command line tool

Development:

- Implementing Google Dremel columnar splitting to A4 files
- Conceptual development of **columnar store EDM and analysis**

Summary and Conclusion

The A4 Library:

- A4 files are **fast self-contained protobuf containers**
- The A4 ObjectStore is a fast way to generate a lot of histograms using few lines of code
- A4 processing helps to make an analysis a nice unix command line tool

Development:

- Implementing Google Dremel columnar splitting to A4 files
- Conceptual development of **columnar store EDM and analysis**

$\Rightarrow$ A lot of challenging problems are ahead

Summary and Conclusion

The A4 Library:

- A4 files are **fast self-contained protobuf containers**
- The A4 ObjectStore is a fast way to generate a lot of histograms using few lines of code
- A4 processing helps to make an analysis a nice unix command line tool

Development:

- Implementing Google Dremel columnar splitting to A4 files
- Conceptual development of **columnar store EDM and analysis**

$\Rightarrow$ A lot of challenging problems are ahead

Thank you for your attention