Particle Physics, 10 000 times faster

Jim Pivarski

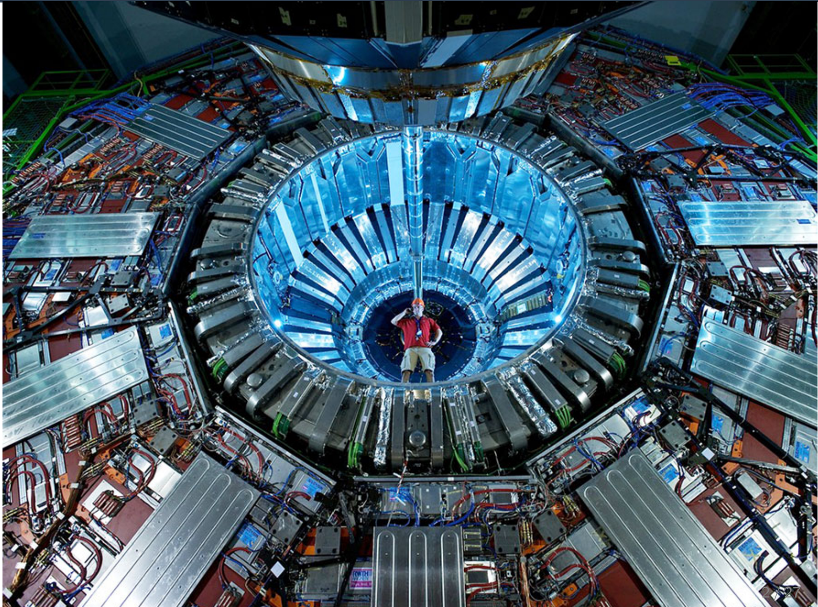
Princeton University – DIANA

September 30, 2017

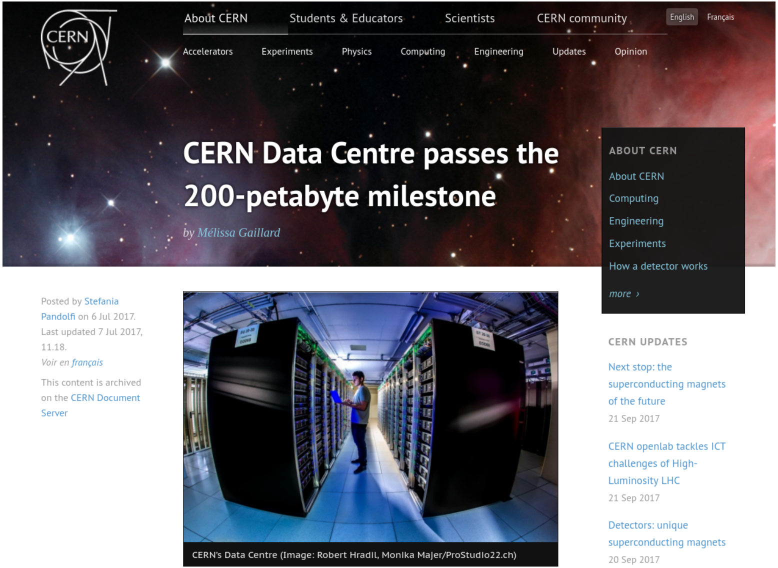
Particle physics: the most industrial field of academia

The goals are academic: to explore strange new phenomena; to seek out new particles and new interactions. . .

The scale is industrial: billion dollar hardware, planning on decadal time-scales, millions of lines of code. . .



It's big data...


About CERN | Students & Educators | Scientists | CERN community | English | Français

Accelerators | Experiments | Physics | Computing | Engineering | Updates | Opinion

CERN Data Centre passes the 200-petabyte milestone

by Mélissa Gaillard

Posted by Stefania Pandolfi on 6 Jul 2017. Last updated 7 Jul 2017, 11.18.

Voir en français

This content is archived on the CERN Document Server

CERN's Data Centre (Image: Robert Hradil, Monika Majer/ProStudio22.ch)

ABOUT CERN

- About CERN
- Computing
- Engineering
- Experiments
- How a detector works
- *more ›*

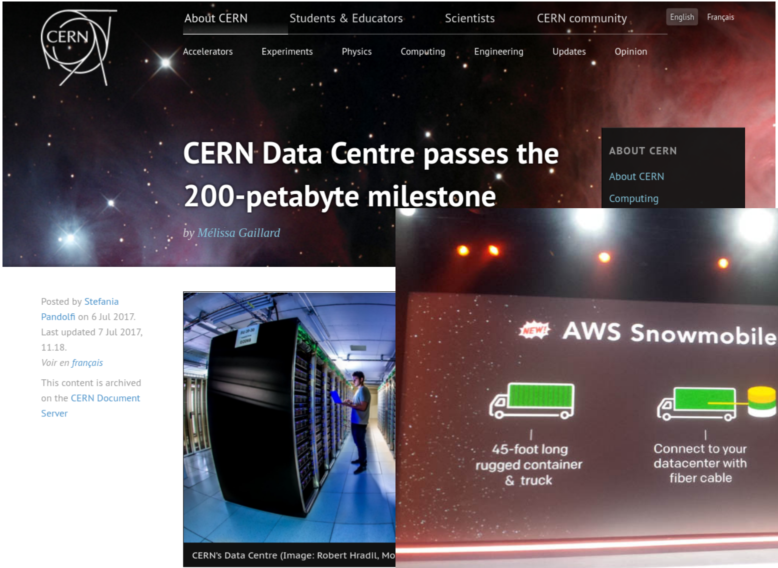
CERN UPDATES

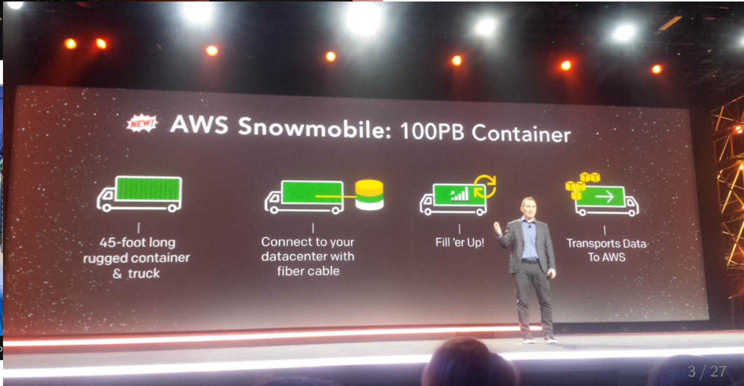
Next stop: the superconducting magnets of the future
21 Sep 2017

CERN openlab tackles ICT challenges of High-Luminosity LHC
21 Sep 2017

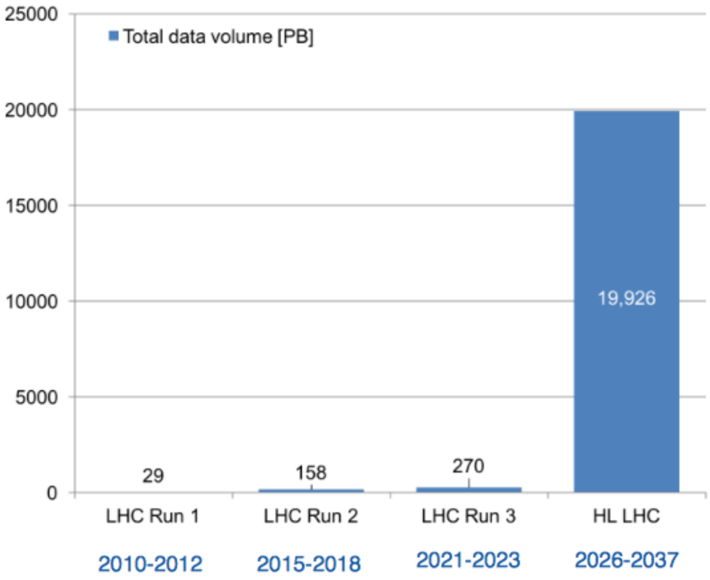
Detectors: unique superconducting magnets
20 Sep 2017

It's big data... but not *really* big

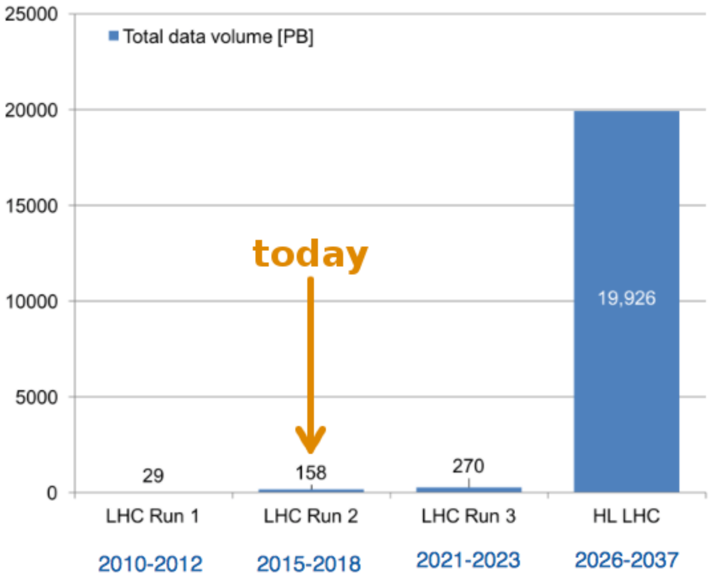




On the third hand, it will be getting bigger. . .

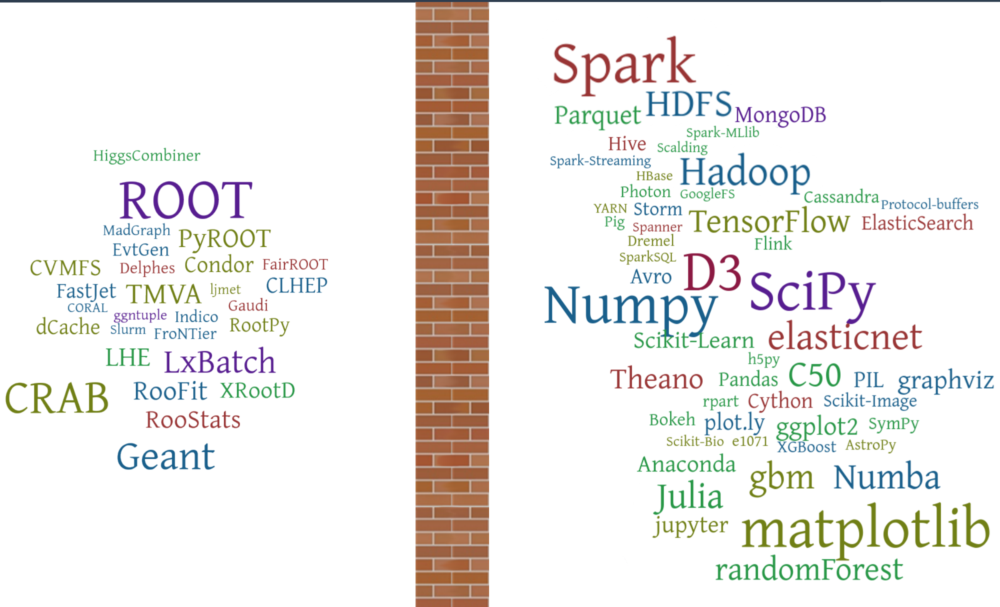


On the third hand, it will be getting bigger. . .

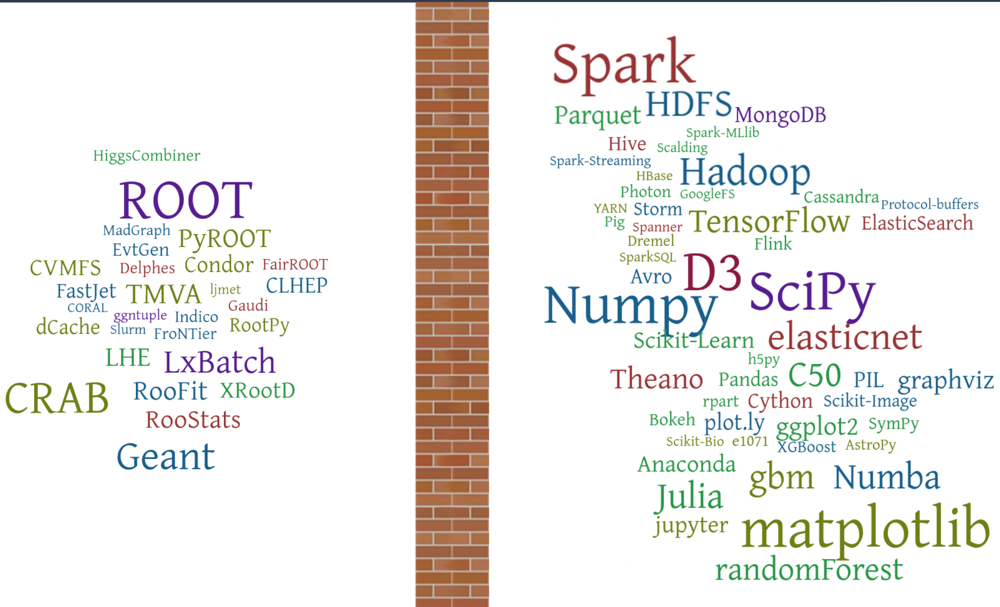


Our software developed outside the big data ecosystem

dianahep

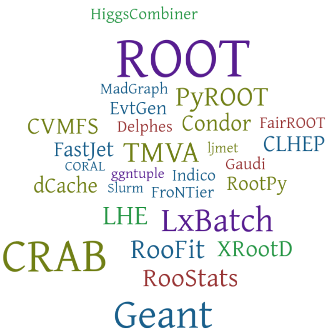


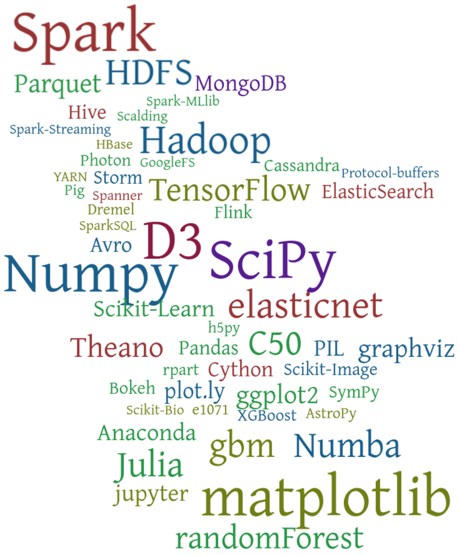
Our software developed ~~outside~~ before the big data ecosystem



Our software developed ~~outside~~ before the big data ecosystem

It's my job to try to find ways to bridge the divide.





The obstacles are not just *accidental*— artifacts of technology choice (e.g. C++ in particle physics and Java in the Hadoop/Spark world).

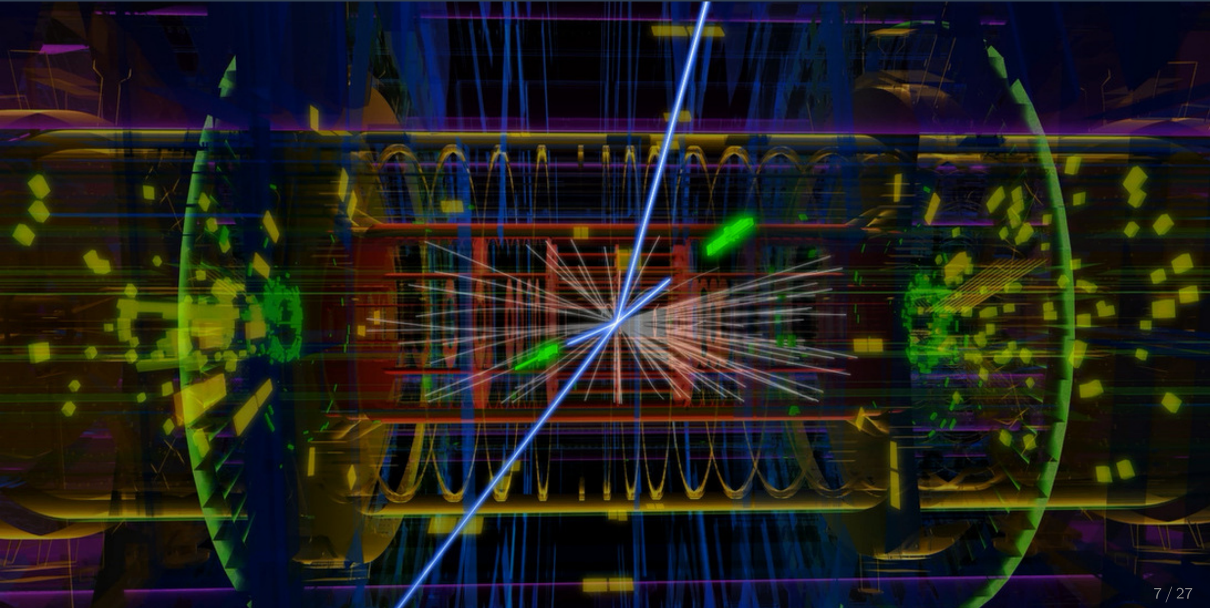
There are also *essential* qualities that current big data systems don't offer.

The obstacles are not just *accidental*— artifacts of technology choice (e.g. C++ in particle physics and Java in the Hadoop/Spark world).

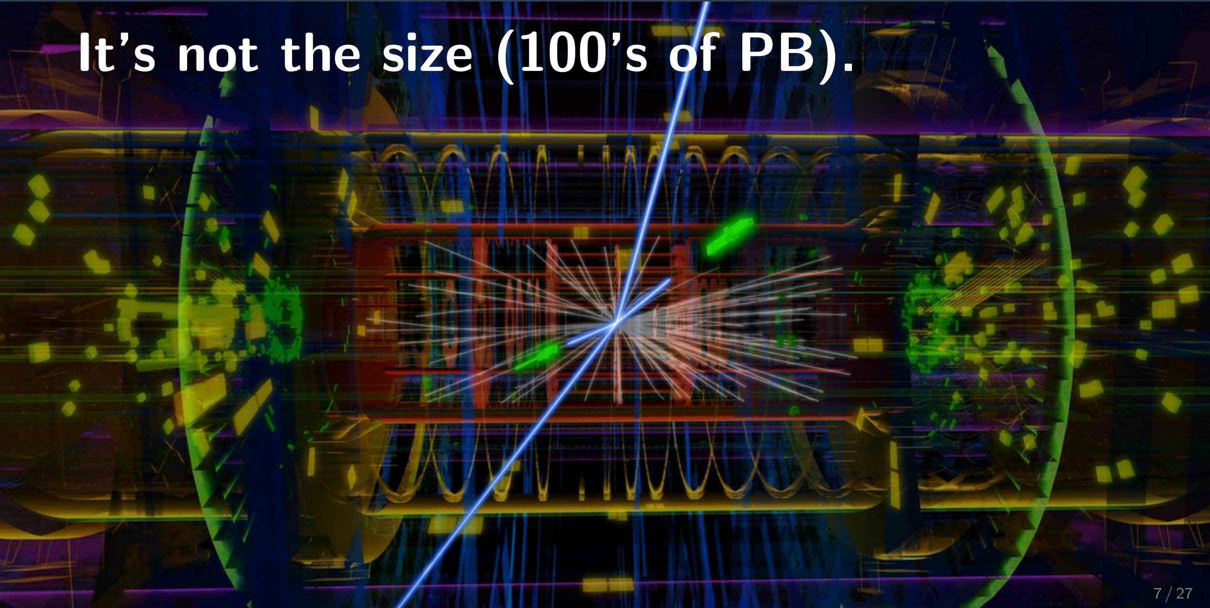
There are also *essential* qualities that current big data systems don't offer.

This represents an opportunity on both sides: alien civilizations that evolved on different planets can learn a lot from each other!

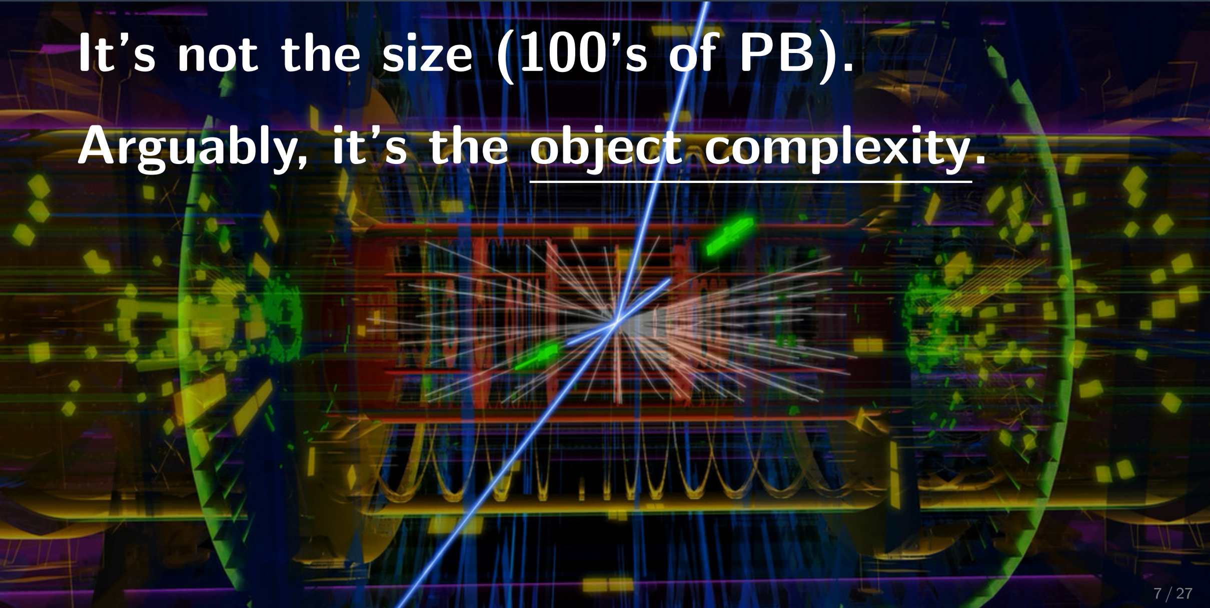
So, what is unique about particle physics data?



It’s not the size (100’s of PB).



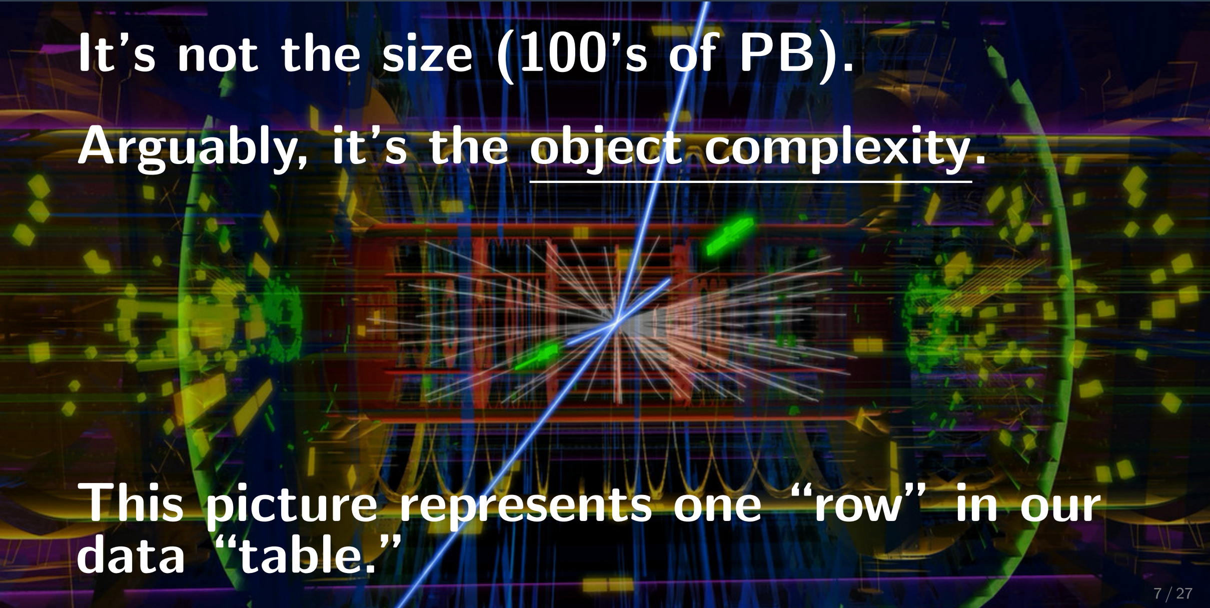
diana hep



It’s not the size (100’s of PB).

Arguably, it’s the <u>object complexity</u>.

So, what is unique about particle physics data?



It's not the size (100's of PB).

Arguably, it's the object complexity.

This picture represents one "row" in our data "table."

Why are "row" and "table" in quotation marks?

Why are “row” and “table” in quotation marks?

Because our data are stored in files, not databases.

Why are “row” and “table” in quotation marks?

Because our data are stored in files, not databases.

- We don’t benefit from indexing, query planning, or high-level query languages.

Why are “row” and “table” in quotation marks?

Because our data are stored in files, not databases.

- We don’t benefit from indexing, query planning, or high-level query languages.
- Every data pull is a custom C++ program, accessing lists of files, taking months while the user chases down failures and stragglers.

Why are “row” and “table” in quotation marks?

Because our data are stored in files, not databases.

- We don’t benefit from indexing, query planning, or high-level query languages.
- Every data pull is a custom C++ program, accessing lists of files, taking months while the user chases down failures and stragglers.
- But if our data *were* in a conventional (relational or NoSQL) database, the first thing we’d have to do is extract it and work with files again.

Why are “row” and “table” in quotation marks?

Because our data are stored in files, not databases.

- We don’t benefit from indexing, query planning, or high-level query languages.
- Every data pull is a custom C++ program, accessing lists of files, taking months while the user chases down failures and stragglers.
- But if our data *were* in a conventional (relational or NoSQL) database, the first thing we’d have to do is extract it and work with files again.

Why?

Our data are deeply nested and cross-linked

Our data are deeply nested and cross-linked

- Not a problem nowadays, as Drill, Parquet, and Arrow can explode nested structures into columns for fast, sparse access.

  (We've been doing it since the 90's.)

Our data are deeply nested and cross-linked

- Not a problem nowadays, as Drill, Parquet, and Arrow can explode nested structures into columns for fast, sparse access.

  (We've been doing it since the 90's.)

- Cross-links (pointers) could be supported by a graph database.

  (Though list indexes work well enough for our large number of small graphs.)

Our data are deeply nested and cross-linked	The operations we perform make intensive use of that structure

- Not a problem nowadays, as Drill, Parquet, and Arrow can explode nested structures into columns for fast, sparse access.

  (We've been doing it since the 90's.)

- Cross-links (pointers) could be supported by a graph database.

  (Though list indexes work well enough for our large number of small graphs.)

Our data are deeply nested and cross-linked

- Not a problem nowadays, as Drill, Parquet, and Arrow can explode nested structures into columns for fast, sparse access.

  (We've been doing it since the 90's.)

- Cross-links (pointers) could be supported by a graph database.

  (Though list indexes work well enough for our large number of small graphs.)

The operations we perform make intensive use of that structure

- We frequently need to search sublists under constraints, optimize pairings, iterate over combinatorics. . .

Our data are deeply nested and cross-linked

- Not a problem nowadays, as Drill, Parquet, and Arrow can explode nested structures into columns for fast, sparse access.

  (We've been doing it since the 90's.)

- Cross-links (pointers) could be supported by a graph database.

  (Though list indexes work well enough for our large number of small graphs.)

The operations we perform make intensive use of that structure

- We frequently need to search sublists under constraints, optimize pairings, iterate over combinatorics...
- Even the simplest particle physics search criteria would require explodes, tags, and joins in SQL.

Our data are deeply nested and cross-linked

- Not a problem nowadays, as Drill, Parquet, and Arrow can explode nested structures into columns for fast, sparse access.

  (We've been doing it since the 90's.)

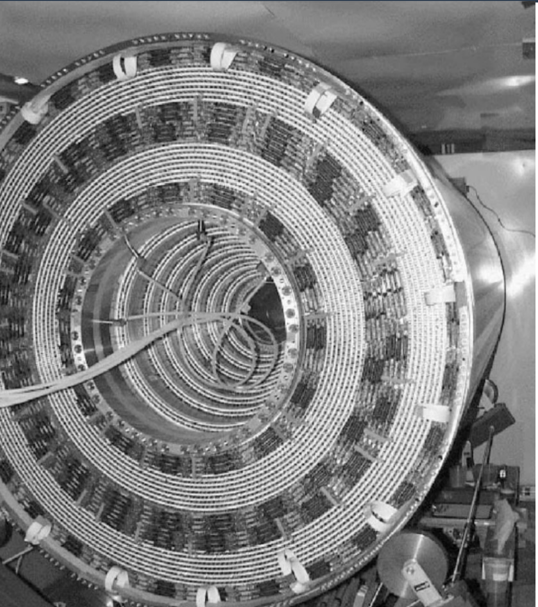
- Cross-links (pointers) could be supported by a graph database.

  (Though list indexes work well enough for our large number of small graphs.)

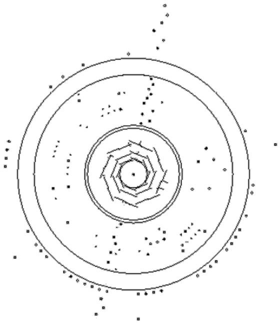
The operations we perform make intensive use of that structure

- We frequently need to search sublists under constraints, optimize pairings, iterate over combinatorics. . .
- Even the simplest particle physics search criteria would require explodes, tags, and joins in SQL.

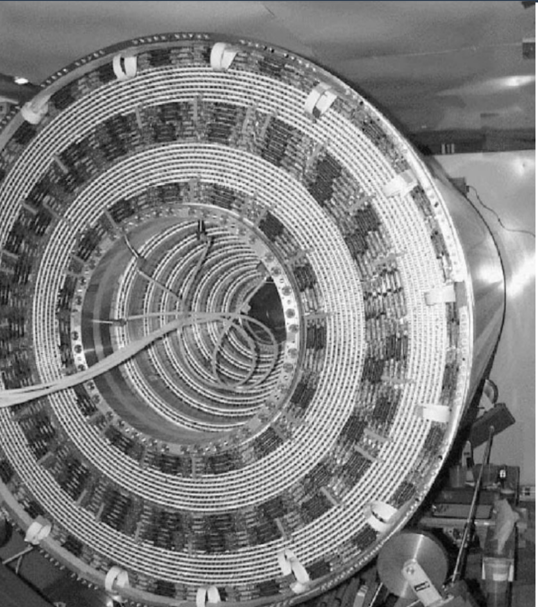
To give a sense of the problem, I'll walk through the steps of an analysis.



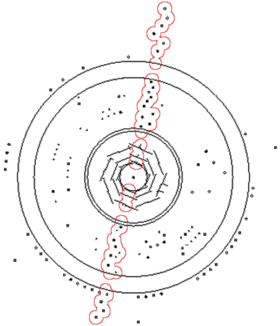
Can you see the particle tracks?



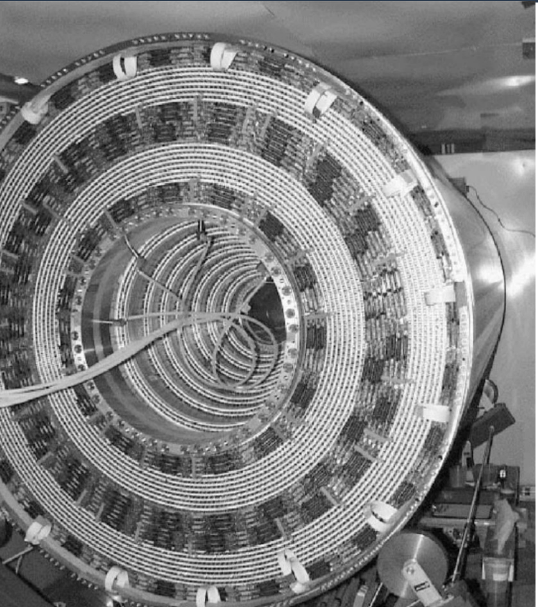
From raw signals to tracks

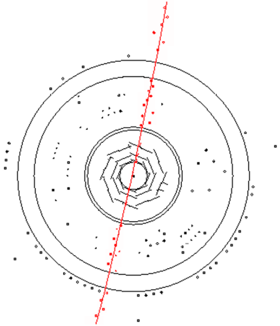


How about now?

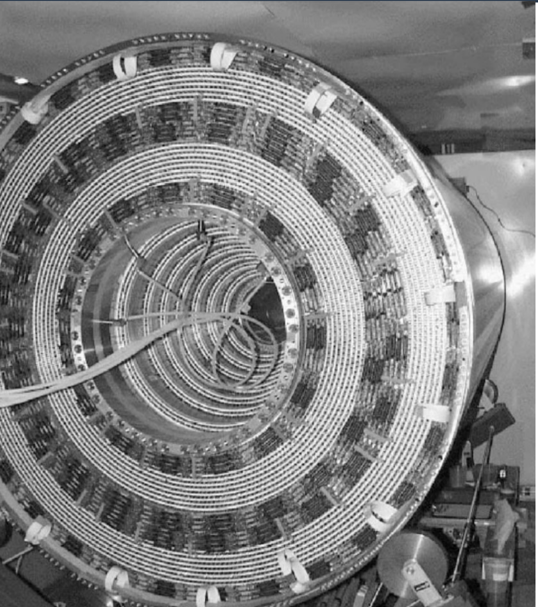


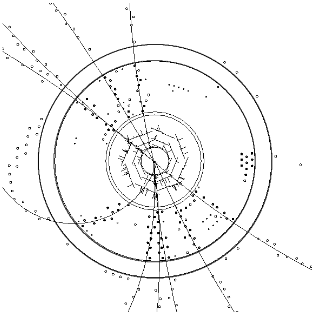
From raw signals to tracks



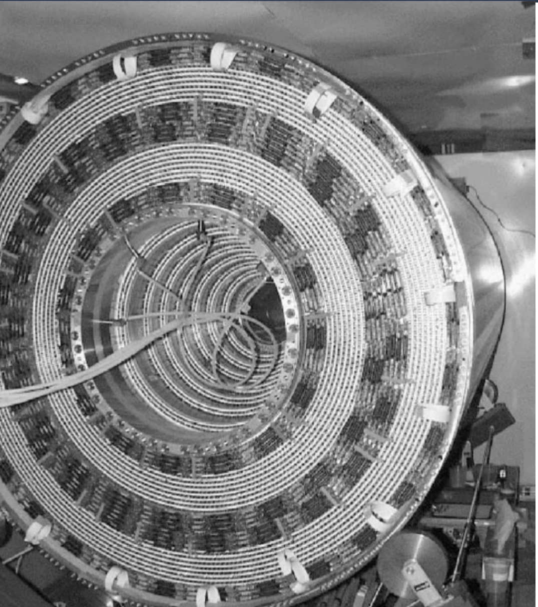


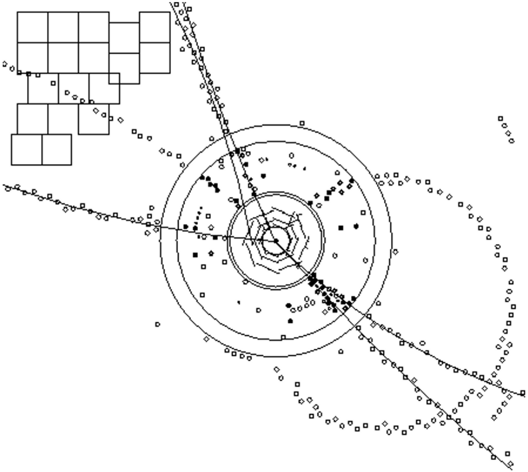
From raw signals to tracks

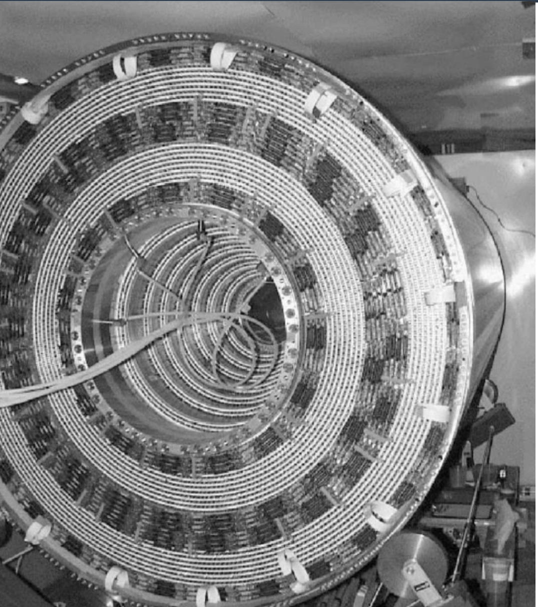




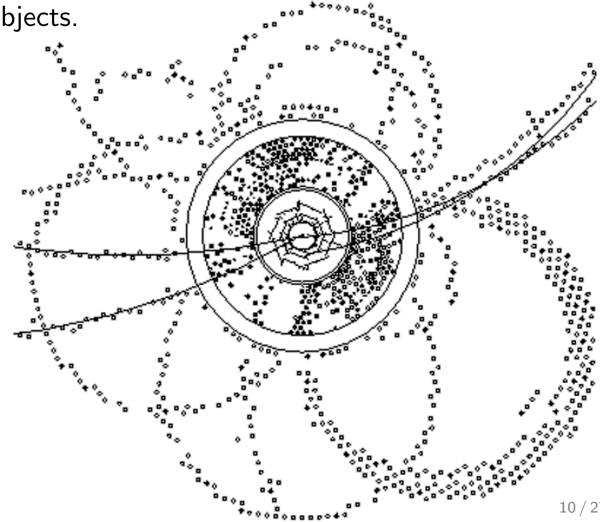
From raw signals to tracks





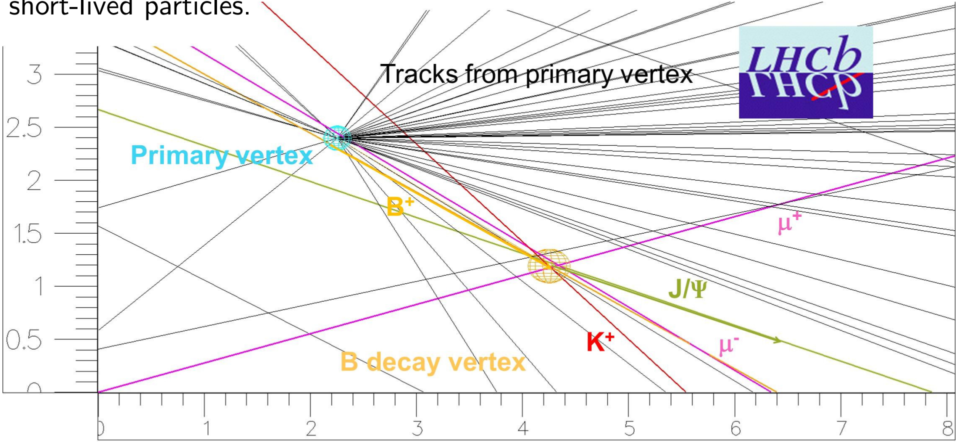


Raw data could have been a (sparsely filled) table, but tracks are an arbitrary-length list of objects.



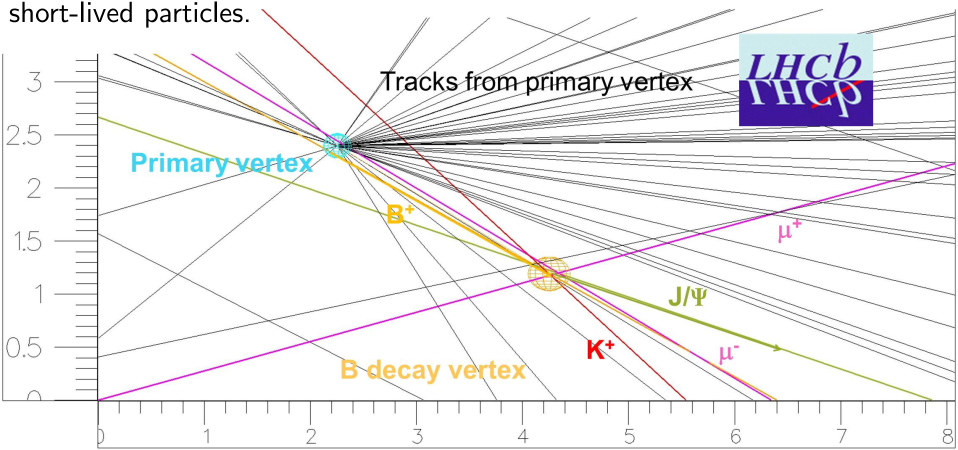
From tracks to particles

Tracks are long-lived particles (on the nanosecond scale) that came from the decay of very short-lived particles.



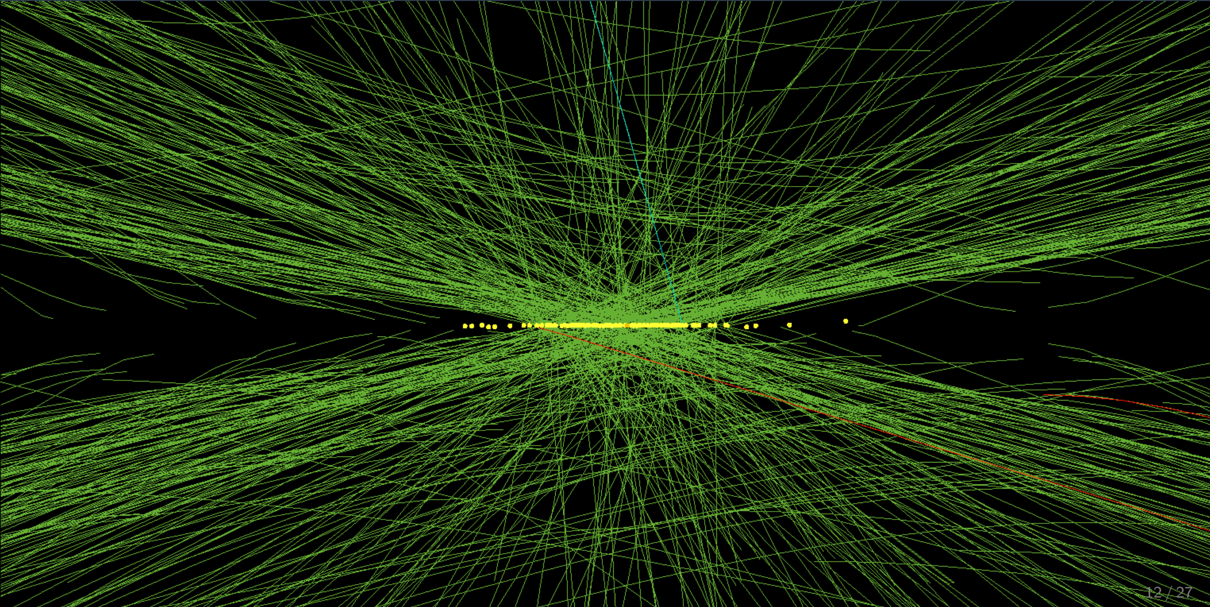
From tracks to particles

Tracks are long-lived particles (on the nanosecond scale) that came from the decay of very short-lived particles.



Tracks have structured associations with one another, and those associations are not certain: flexibility has to be carried through to the final analysis.

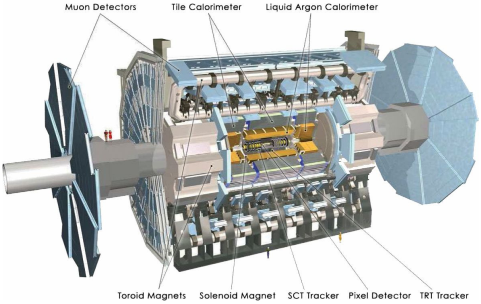
And there are a lot of combinations to consider. . .

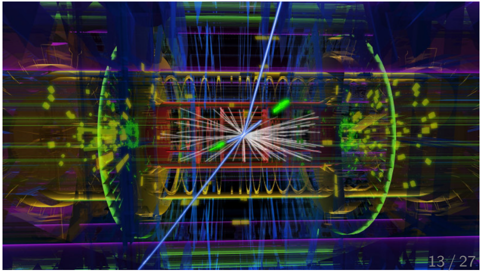


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

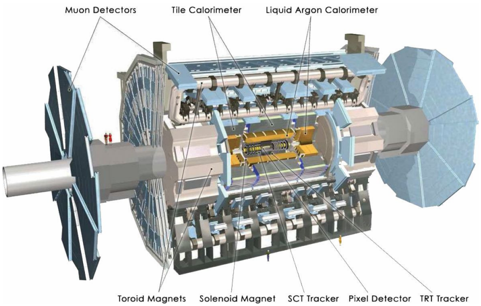
$$H \to ZZ \to e^+e^-\mu^+\mu^-$$

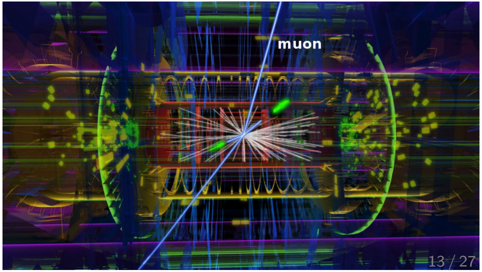




Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \to ZZ \to e^+e^-\mu^+\mu^-$$

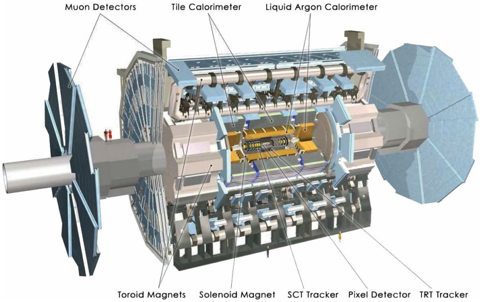


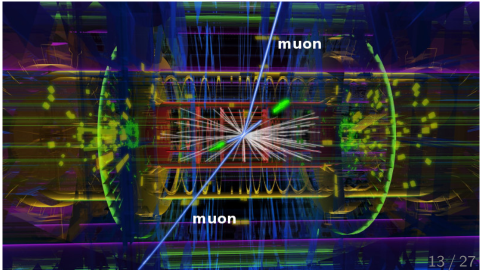


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \rightarrow ZZ \rightarrow e^+ e^- \mu^+ \mu^-$$

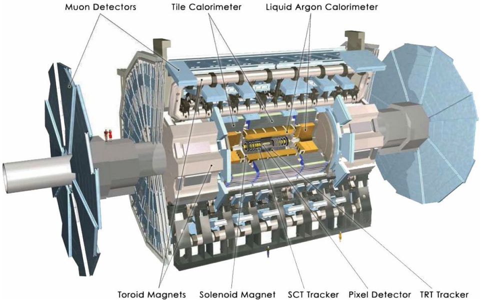


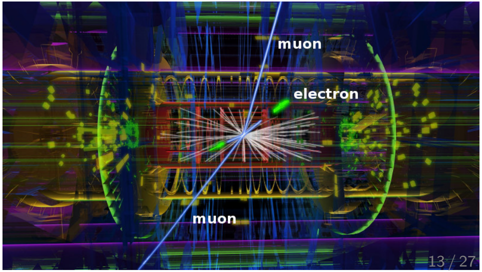


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \to ZZ \to e^+e^-\mu^+\mu^-$$

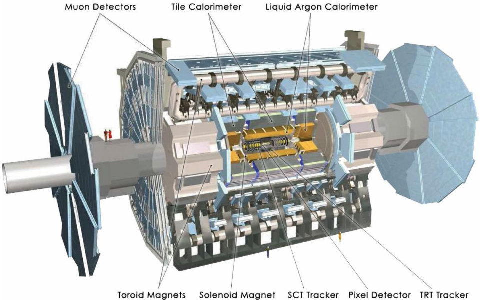


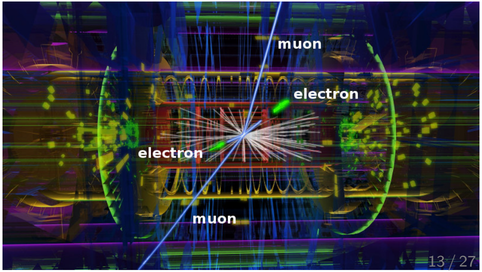


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \rightarrow ZZ \rightarrow e^+e^-\mu^+\mu^-$$

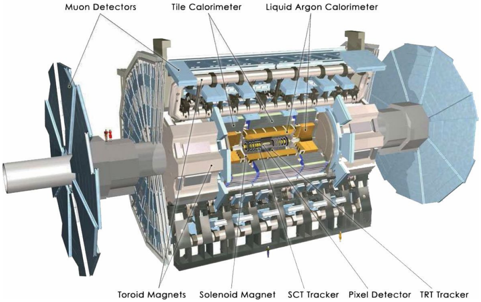


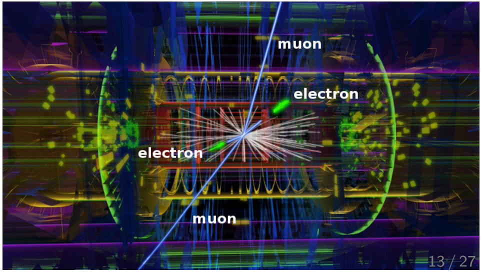


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \to ZZ \to \underbrace{e^+ e^-}\mu^+ \mu^-$$

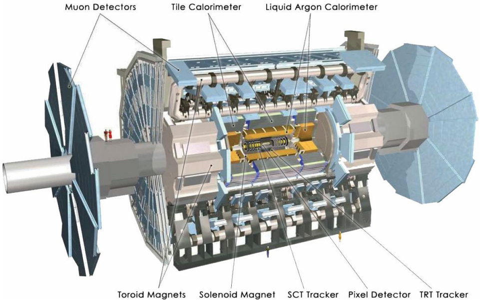


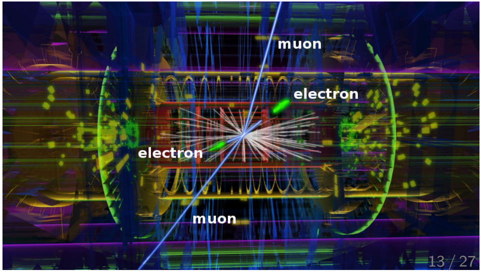


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \to ZZ \to \underbrace{e^+e^-}\,\underbrace{\mu^+\mu^-}$$

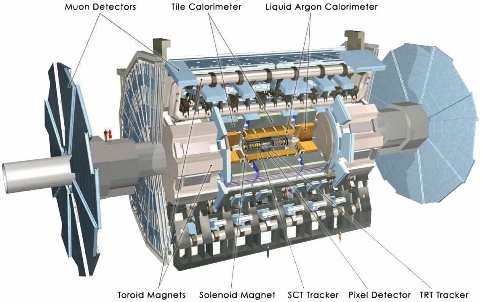


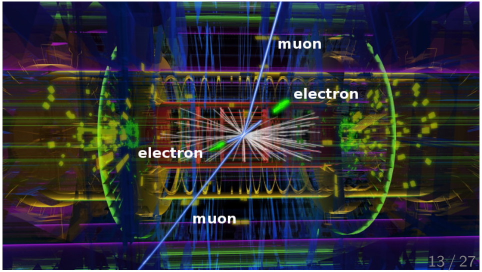


From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

$$H \to ZZ \to \underbrace{\underbrace{e^+ e^-} \quad \underbrace{\mu^+ \mu^-}}_{\text{compute mass of progenitor}}$$

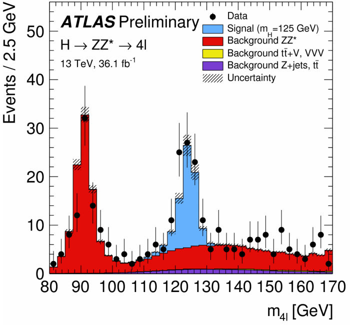


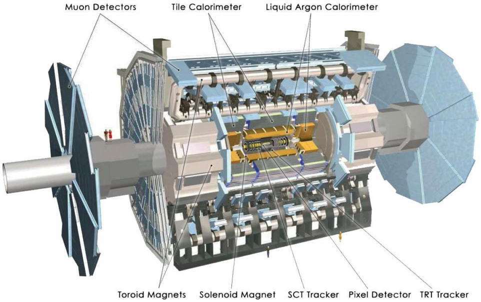


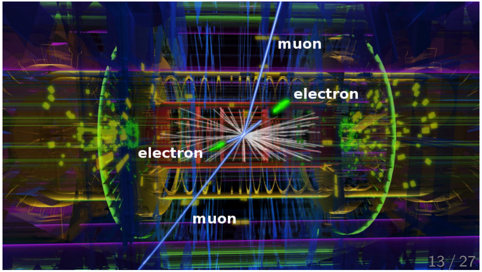
From particles to discovery

Suppose there's a particle called "Higgs" that would decay into two "Z bosons," each of which decays into two electrons or two muons.

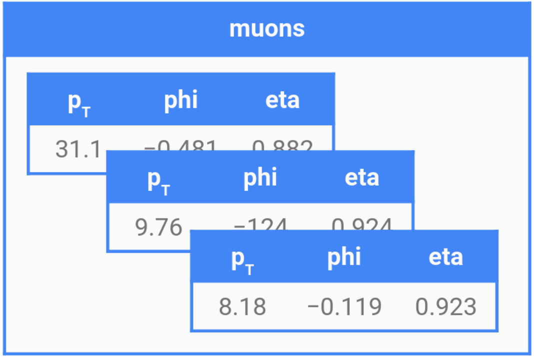
$$H \to ZZ \to \underbrace{\underbrace{e^+e^-}\quad\underbrace{\mu^+\mu^-}}_{\text{compute mass of progenitor}}$$







Objects versus flat tables



mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

To try different associations between particles, between data from different detectors, in many different combinations. . .

. . . it's easier to write these as *algorithms over objects!*

Modern SQL can *represent* that

```sql
CREATE TYPE PARTICLE FROM
  STRUCT<pt: FLOAT,
         eta: FLOAT,
         phi: FLOAT
         charge: INT>;

CREATE TABLE events (
  eventid   INT,
  electrons ARRAY<PARTICLE>,
  muons     ARRAY<PARTICLE>,
  UNIQUE KEY eventid
);
```

Modern SQL can *represent* that

```sql
CREATE TYPE PARTICLE FROM
  STRUCT<pt: FLOAT,
         eta: FLOAT,
         phi: FLOAT
         charge: INT>;

CREATE TABLE events (
  eventid   INT,
  electrons ARRAY<PARTICLE>,
  muons     ARRAY<PARTICLE>,
  UNIQUE KEY eventid
);
```

But to do the Higgs search, you'd have to

1. explode the `electrons` array into a table,
2. explode the `muons` array into a table,
3. do an outer join of the `electrons` table on itself, subject to the constraints that they have the same `eventid` and opposite `charge`,
4. filter for those close to the Z mass,
5. do the same for the `muons` table,
6. do a join of *those* two tables to compute H masses,
7. group-by to make a histogram.

Modern SQL can *represent* that

```
CREATE TYPE PARTICLE FROM
  STRUCT<pt: FLOAT,
         eta: FLOAT,
         phi: FLOAT,
         charge: INT>;

CREATE TABLE events (
  eventid   INT,
  electrons ARRAY<PARTICLE>,
  muons     ARRAY<PARTICLE>,
  UNIQUE KEY eventid
);
```

But to do the Higgs search, you'd have to

1. explode the `electrons` array into a table,
2. explode the `muons` array into a table,
3. do an outer join of the `electrons` table on itself, subject to the constraints that they have the same `eventid` and opposite `charge`,
4. filter for those close to the Z mass,
5. do the same for the `muons` table,
6. do a join of *those* two tables to compute H masses,
7. group-by to make a histogram.

This is in no way easier than writing a nested for loop!

We can get the best of both worlds by adding first-class functions to SQL.

Last year, I started developing Femtocode: a declarative query language with a functional, object-oriented syntax.

We can get the best of both worlds by adding first-class functions to SQL.

Last year, I started developing Femtocode: a declarative query language with a functional, object-oriented syntax.

```
dataset.histogram(90, 80, 170, flatten({event =>
    electrons = event.tracks.filter(
        e => 0.9 < e.calorimeterEnergy / e.trackMomentum < 1.1)
    muons = event.tracks.filter(m => m.outerHits > 4)

    def goodz(p1, p2):
        p1.charge * p2.charge < 0 and 60 < mass(p1, p2) < 120

    ez = electrons.distinctpairs.filter(goodz)
    mz =     muons.distinctpairs.filter(goodz)

    table(ez, mz).map((e1, e2), (m1, m2) => mass(e1, e2, m1, m2))
}))
```

Why the language is great and I won't be talking about it

Why the language is great and I won't be talking about it

By shrink-wrapping the language around our problem, we could add some nice features:

- automatically vectorize calculations across objects
- 100% compile-time error checking with dependent types

Why the language is great and I won't be talking about it

By shrink-wrapping the language around our problem, we could add some nice features:

- automatically vectorize calculations across objects
- 100% compile-time error checking with dependent types

In the past year, other projects started adding functional programming to query languages:

- SparkSQL 3.0's `TRANSFORM` keyword
- DataFun: Michael Arntzenius's talk in US Regency AB!

Why the language is great and I won't be talking about it

By shrink-wrapping the language around our problem, we could add some nice features:

- automatically vectorize calculations across objects
- 100% compile-time error checking with dependent types

In the past year, other projects started adding functional programming to query languages:

- SparkSQL 3.0's `TRANSFORM` keyword
- DataFun: Michael Arntzenius's talk in US Regency AB!

Meanwhile, we discovered that Femtocode's internal data representation is orders of magnitude faster to scan than our current methods.

Why the language is great and I won't be talking about it

By shrink-wrapping the language around our problem, we could add some nice features:

- automatically vectorize calculations across objects
- 100% compile-time error checking with dependent types

In the past year, other projects started adding functional programming to query languages:

- SparkSQL 3.0's `TRANSFORM` keyword
- DataFun: Michael Arntzenius's talk in US Regency AB!

Meanwhile, we discovered that Femtocode's internal data representation is orders of magnitude faster to scan than our current methods.

This is the key issue.

Why the language is great and I won't be talking about it

By shrink-wrapping the language around our problem, we could add some nice features:

- automatically vectorize calculations across objects
- 100% compile-time error checking with dependent types

In the past year, other projects started adding functional programming to query languages:

- SparkSQL 3.0's `TRANSFORM` keyword
- DataFun: Michael Arntzenius's talk in US Regency AB!

Meanwhile, we discovered that Femtocode's internal data representation is orders of magnitude faster to scan than our current methods.

This is the key issue.

We can apply the new data representation on its own, without introducing a new language.

Iteration over single attribute in many objects

Such as (single-threaded):

```
for (i = 0;  i < numEvents;  i++)
    for (j = 0;  j < events[i].numTracks;  j++)
        fill_histogram(events[i].tracks[j].trackMomentum);
```

Four orders of magnitude between how we currently access data
and how we <u>could</u> access data!

0.018 MHz   our current framework

250 MHz   minimal loop over flattened `trackMomentum` array

Iteration over single attribute in many objects

Such as (single-threaded):

```
for (i = 0;  i < numEvents;  i++)
    for (j = 0;  j < events[i].numTracks;  j++)
        fill_histogram(events[i].tracks[j].trackMomentum);
```

Four orders of magnitude between how we currently access data
and how we could access data!

0.018 MHz our current framework

31 MHz allocate `std::vector<double>` on stack for each event

250 MHz minimal loop over flattened `trackMomentum` array

Iteration over single attribute in many objects

Such as (single-threaded):

```
for (i = 0;  i < numEvents;  i++)
    for (j = 0;  j < events[i].numTracks;  j++)
        fill_histogram(events[i].tracks[j].trackMomentum);
```

Four orders of magnitude between how we currently access data and how we could access data!

0.018 MHz	our current framework
12 MHz	allocate `std::vector<double*>` on heap for each event; then delete
31 MHz	allocate `std::vector<double>` on stack for each event
250 MHz	minimal loop over flattened `trackMomentum` array

Iteration over single attribute in many objects

Such as (single-threaded):

```
for (i = 0;  i < numEvents;  i++)
    for (j = 0;  j < events[i].numTracks;  j++)
        fill_histogram(events[i].tracks[j].trackMomentum);
```

Four orders of magnitude between how we currently access data
and how we could access data!

0.018 MHz	our current framework
2.8 MHz	deserialize into `std::vectors` of single-attribute `Track` instances
12 MHz	allocate `std::vector<double*>` on heap for each event; then delete
31 MHz	allocate `std::vector<double>` on stack for each event
250 MHz	minimal loop over flattened `trackMomentum` array

Iteration over single attribute in many objects

dianahep

Such as (single-threaded):

```
for (i = 0;  i < numEvents;  i++)
    for (j = 0;  j < events[i].numTracks;  j++)
        fill_histogram(events[i].tracks[j].trackMomentum);
```

Four orders of magnitude between how we currently access data
and how we <u>could</u> access data!

0.018 MHz	our current framework
0.029 MHz	deserialize into `Track` instances with all 95 track attributes
2.8 MHz	deserialize into `std::vectors` of single-attribute `Track` instances
12 MHz	allocate `std::vector<double*>` on heap for each event; then delete
31 MHz	allocate `std::vector<double>` on stack for each event
250 MHz	minimal loop over flattened `trackMomentum` array

To be fair...

The current framework is never used to fill only one histogram.

It's usually used to extract a subset of events and attributes for the physicist to analyze locally (laptop, university cluster, national lab).

To be fair...

The current framework is never used to fill only one histogram.

It's usually used to extract a subset of events and attributes for the physicist to analyze locally (laptop, university cluster, national lab).

1. These jobs take weeks or months[1].

[1] one analyst claimed 1.5 years for a single data pull!

To be fair...

The current framework is never used to fill only one histogram.

It's usually used to extract a subset of events and attributes for the physicist to analyze locally (laptop, university cluster, national lab).

1. These jobs take weeks or months[1].
2. The data analyst[2] has to manage sets of files and chase down failed jobs.

[1] one analyst claimed 1.5 years for a single data pull!
[2] usually the youngest graduate student

To be fair...

The current framework is never used to fill only one histogram.

It's usually used to extract a subset of events and attributes for the physicist to analyze locally (laptop, university cluster, national lab).

1. These jobs take weeks or months[1].
2. The data analyst[2] has to manage sets of files and chase down failed jobs.
3. Repeating the process is so time-consuming that analysis groups hedge their bets by requesting more data than they're sure they'll need.

[1] one analyst claimed 1.5 years for a single data pull!
[2] usually the youngest graduate student

To be fair...

The current framework is never used to fill only one histogram.

It's usually used to extract a subset of events and attributes for the physicist to analyze locally (laptop, university cluster, national lab).

1. These jobs take weeks or months[1].
2. The data analyst[2] has to manage sets of files and chase down failed jobs.
3. Repeating the process is so time-consuming that analysis groups hedge their bets by requesting more data than they're sure they'll need.
4. So the process is slower and the downloaded dataset is bigger.

[1] one analyst claimed 1.5 years for a single data pull!
[2] usually the youngest graduate student

To be fair...

The current framework is never used to fill only one histogram.

It's usually used to extract a subset of events and attributes for the physicist to analyze locally (laptop, university cluster, national lab).

1. These jobs take weeks or months[1].
2. The data analyst[2] has to manage sets of files and chase down failed jobs.
3. Repeating the process is so time-consuming that analysis groups hedge their bets by requesting more data than they're sure they'll need.
4. So the process is slower and the downloaded dataset is bigger.
5. `GOTO #1`.

[1] one analyst claimed 1.5 years for a single data pull!
[2] usually the youngest graduate student

So this is really about a change in behavior:

(1)	(2)
big download, work locally	small operations on a shared resource

So this is really about a change in behavior:

(1)	(2)
big download, work locally	small operations on a shared resource

For the new style of analysis workflow to compete,

So this is really about a change in behavior:

(1)	(2)
big download, work locally	small operations on a shared resource

For the new style of analysis workflow to compete,

- responses must be rapid enough for end-user analysis (seconds per plot)

So this is really about a change in behavior:

(1)	(2)
big download, work locally	small operations on a shared resource

For the new style of analysis workflow to compete,

- responses must be rapid enough for end-user analysis (seconds per plot)
- the interface must allow for algorithms on nested objects.

Key idea: <u>leave</u> the data in columns!

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

```
[0,                                       3,   3,    4] (outer list offsets)
[ 0,              4,   4,               6,   7] (inner list offsets)
[   a, b, c, d,          e, f,             g    ] (attribute data)
```

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

```
[0,                               3,   3,    4] (outer list offsets)
[ 0,             4,   4,             6,   7] (inner list offsets)
[   a, b, c, d,         e, f,          g    ] (attribute data)
```

when the user writes

```
for outer in lists:
   for inner in outer:
      for char in inner:
         print(char)
```

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

```
[0,                                3,   3,    4]  (outer list offsets)
[ 0,             4,   4,                6,    7]  (inner list offsets)
[   a, b, c, d,          e, f,             g   ]  (attribute data)
```

<u>when the user writes</u>

```
for outer in lists:
    for inner in outer:
        for char in inner:
            print(char)
```

<u>we shouldn't create lists and sublists...</u>

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

```
[0,                          3,   3,    4]  (outer list offsets)
[ 0,              4,   4,          6,   7]  (inner list offsets)
[   a, b, c, d,         e, f,       g    ]  (attribute data)
```

when the user writes

```
for outer in lists:
    for inner in outer:
        for char in inner:
            print(char)
```

we should instead execute

```
for (i = 0; i < 3; i++)
    for (j = outer[i]; j < outer[i+1]; j++)
        for (k = inner[j]; k < inner[j+1]; k++)
            print(data[k]);
```

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

```
[0,                           3,   3,    4]  (outer list offsets)
[ 0,           4,   4,            6,    7]   (inner list offsets)
[   a, b, c, d,       e, f,          g    ]  (attribute data)
```

when the user writes

```
for outer in lists:
   for inner in outer:
      for char in inner:
         print(char)
```

or even (special case of exhaustive nested loops)

```
for (k = 0; k < inner[outer[3]]; k++)
   print(data[k]);
```

Leave the data in columns!

We've always *stored* the data as exploded columns (similar to Apache Parquet), but we also shouldn't spend time materializing them as objects.

Suppose that `[[a, b, c, d], [], [e, f]], [], [[g]]` is stored as

```
[0,                                  3,   3,    4] (outer list offsets)
[ 0,                4,   4,               6,    7] (inner list offsets)
[   a, b, c, d,            e, f,             g   ] (attribute data)
```

when the user writes

```
for outer in lists:
    for inner in outer:
        for char in inner:
            print(char)
```

or even (special case of exhaustive nested loops)

```
for (k = 0; k < inner[outer[3]]; k++)
    print(data[k]);
```

The data representation is Apache Arrow; the code transformation can be automated.

Measurements in a real system

I'm using Python as a stepping-stone toward Femtocode. By transforming Python object references, we can turn it into nothing but arrays and number-crunching.

Measurements in a real system

I'm using Python as a stepping-stone toward Femtocode. By transforming Python object references, we can turn it into nothing but arrays and number-crunching.

Numba, a Python-to-LLVM compiler, is particularly good at optimizing this.

Measurements in a real system

I'm using Python as a stepping-stone toward Femtocode. By transforming Python object references, we can turn it into nothing but arrays and number-crunching.

Numba, a Python-to-LLVM compiler, is particularly good at optimizing this.

```
# objects in Python code
def dimuon(event):
    n = len(event.muons)
    for i in range(n):
        for j in range(i+1, n):
            m1 = event.muons[i]
            m2 = event.muons[j]
            mass = sqrt(2*m1.pt*m2.pt*(
                cosh(m1.eta - m2.eta) -
                cos(m1.phi - m2.phi)))
            fill_histogram(mass)

# translated to array references
plur.compile.run(arrays, dimuon)
```

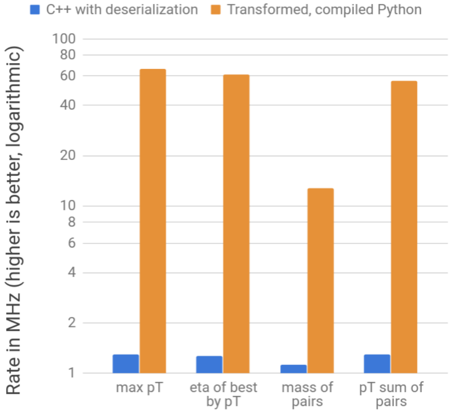
Measurements in a real system

I'm using Python as a stepping-stone toward Femtocode. By transforming Python object references, we can turn it into nothing but arrays and number-crunching.

Numba, a Python-to-LLVM compiler, is particularly good at optimizing this.

```
# objects in Python code
def dimuon(event):
    n = len(event.muons)
    for i in range(n):
        for j in range(i+1, n):
            m1 = event.muons[i]
            m2 = event.muons[j]
            mass = sqrt(2*m1.pt*m2.pt*(
                cosh(m1.eta - m2.eta) -
                cos(m1.phi - m2.phi)))
            fill_histogram(mass)

# translated to array references
plur.compile.run(arrays, dimuon)
```



General code transformation for all types is hard

General code transformation for all types is hard

Concentrate on the *minimal set* of type generators:

General code transformation for all types is hard

Concentrate on the *minimal set* of type generators:

Primitives: fixed-width numbers, booleans, characters.

Lists: arbitrary-length lists of another type.

Unions: set of possible types; runtime object is exactly one possibility.

Records: package of several named, typed fields; runtime object has all nested subfields.

General code transformation for all types is hard

Concentrate on the *minimal set* of type generators:

Primitives: fixed-width numbers, booleans, characters.

Lists: arbitrary-length lists of another type.

Unions: set of possible types; runtime object is exactly one possibility.

Records: package of several named, typed fields; runtime object has all nested subfields.

Other common types (such as strings) can be constructed from these (arbitrary-length list of characters, for instance).

General code transformation for all types is hard

Concentrate on the *minimal set* of type generators:

Primitives: fixed-width numbers, booleans, characters.

Lists: arbitrary-length lists of another type.

Unions: set of possible types; runtime object is exactly one possibility.

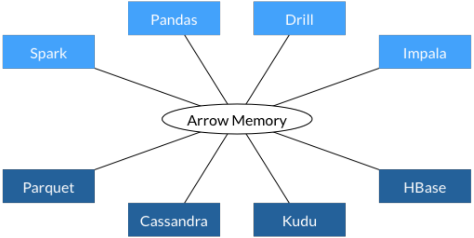
Records: package of several named, typed fields; runtime object has all nested subfields.

Other common types (such as strings) can be constructed from these (arbitrary-length list of characters, for instance).

https://github.com/diana-hep/plur

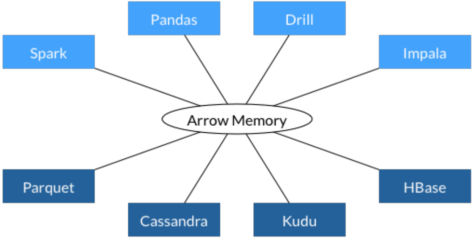
Relationship to Apache Arrow

The way that Primitives, Lists, (sparse) Unions, and Records are represented are a subset of the Apache Arrow specification, so in principle this ought to make Python— with arbitrarily nested loops— fast on Arrow dataframes.



Relationship to Apache Arrow

The way that Primitives, Lists, (sparse) Unions, and Records are represented are a subset of the Apache Arrow specification, so in principle this ought to make Python— with arbitrarily nested loops— fast on Arrow dataframes.

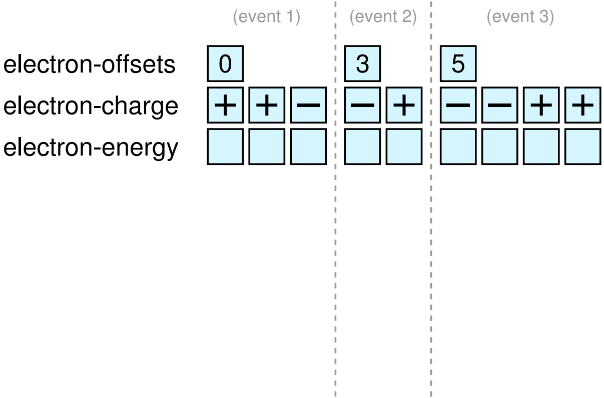


Is anyone else interested in that?

Last thought: manage the data in columns, too!

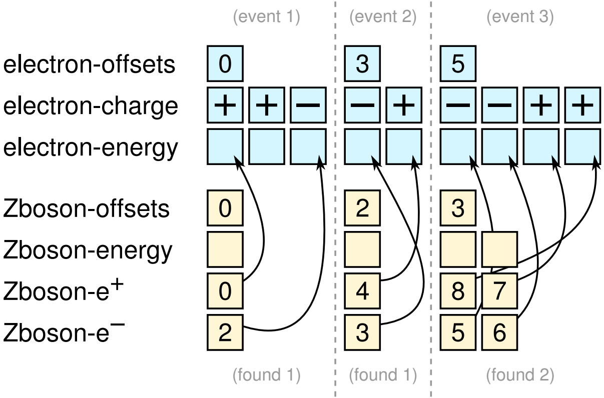
Last thought: manage the data in columns, too!

One reason users copy data is to enrich it with derived features:



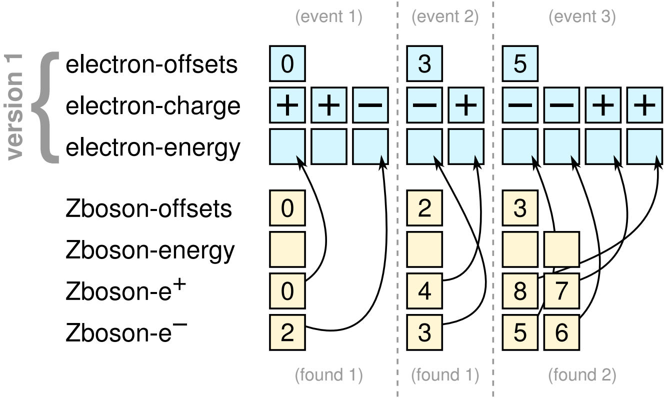
Last thought: manage the data in columns, too!

One reason users copy data is to enrich it with derived features:



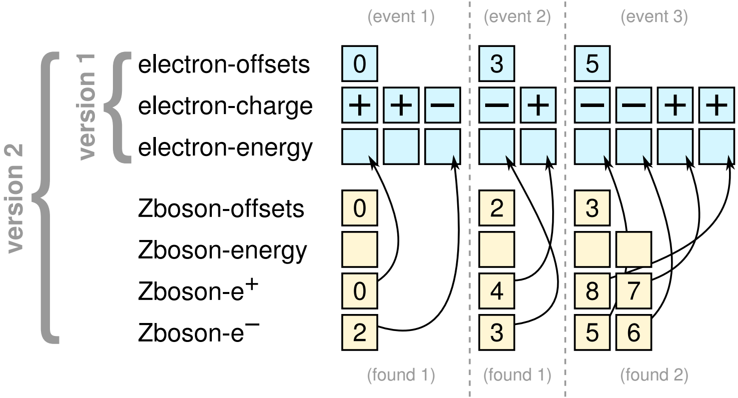
Last thought: manage the data in columns, too!

One reason users copy data is to enrich it with derived features:



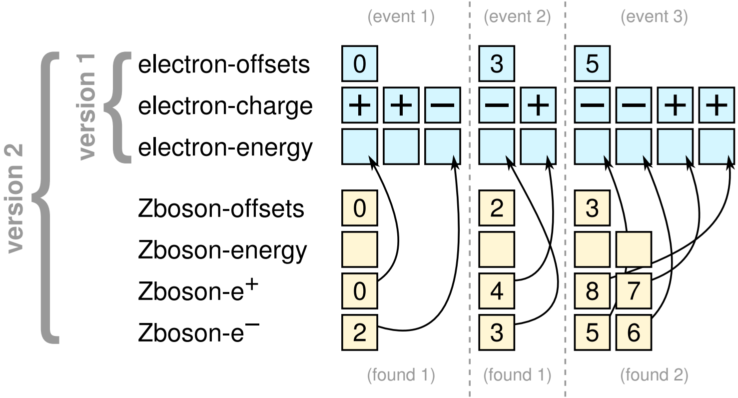
Last thought: manage the data in columns, too!

One reason users copy data is to enrich it with derived features:



Last thought: manage the data in columns, too!

One reason users copy data is to enrich it with derived features:



If the data are addressed as individual columns, rather than files, users can change the structure of the data by adding new columns, *without copying.*

I hope it was interesting to learn about data issues in particle physics.

But I'm really interested in hearing back from you: do you have suggestions or do you think these tools could be useful in your work?

If it would help but needs to be more mature, are you interested in collaborating?

`pivarski@fnal.gov`