



**Faculty
of Physics**

WARSAW UNIVERSITY OF TECHNOLOGY

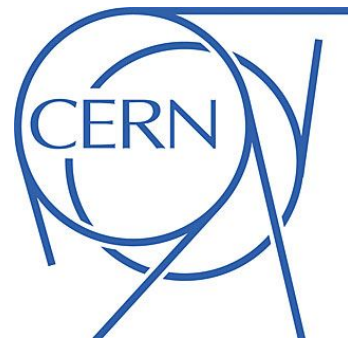
Event mixing in O2



ALICE

Maja Karwowska

O2 tutorials, 07.11.2023

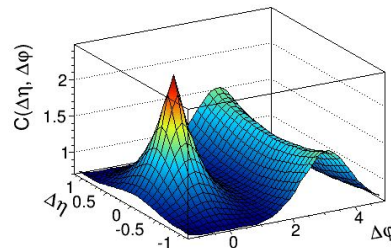


A novel algorithm of event mixing for ALICE Run 3

Angular and femtoscopic correlations: analyzing QGP initial state and thermalization mechanisms

Correlation function: $S(\Delta\eta, \Delta\phi) = \frac{d^2 N^{signal}}{\Delta\eta \Delta\phi}$ $B(\Delta\eta, \Delta\phi) = \frac{d^2 N^{mixed}}{\Delta\eta \Delta\phi}$

$$C(\Delta\eta, \Delta\phi) = \frac{N^{mixed}}{N^{signal}} \frac{S(\Delta\eta, \Delta\phi)}{B(\Delta\eta, \Delta\phi)}$$



Event mixing: pairs of tracks (V0s/cascades/...) from **2 different collisions** from the **same bin**, e.g., multiplicity and z-vertex intervals.

Run 2: sort collisions into **a vector of mixing buffers**, at the same time select pairs in a double loop

Run 3: many more collisions → **big** memory and time **overhead**

Idea: **lazy** generation (one at time) of combinations of elements, **without data copies**

mixed-event pairs: **binned combinations** of collisions + full track combinations

Universal – any n-tuple, any table.

How to implement combinations **effectively**?

combinations – pairs, triples, ... of elements from a table or different tables

Memory to store all tuples: **$O(n!)$** where n is the table size → **too much!**

→ **Lazy** generation – one tuple by one

iterator – refers to a certain row in a table

mMaxOffset: (5, 6, 5)

tables' sizes: 5, 6, 5

mCurrent:

(0, 0, 0) → (0, 0, 1) → ... → (0, 0, 4) → (0, 0, 5) → (0, 1, 0) → ... → (5, 6, 5)

reset of the last
iterator

end of the
combination

end of table

the last but one
iterator moved forward

Basic combination policies

Table sizes: (5, 6)

1. **Full:** (0, 0), ..., (0, 5), **(1, 0)**, ..., (1, 5), ..., **(4, 0)**, ..., (4, 5)
 - a. always reset an iterator to table begin
2. **Upper:** (0, 0), ..., (0, 5), **(1, 1)**, ..., (1, 5), ..., **(4, 4)**, (4, 5)
 - a. reset to the position of the left iterator
 - b. no repetitions of pairs like (0, 1) and (1, 0)
3. **Strictly upper:** (0, 1), ..., (0, 5), **(1, 2)**, ..., (1, 5), ..., **(3, 5)**
 - a. reset to the position of the left iterator + 1
 - b. max position: (table size - distance from the rightmost iterator) = (4, 6)
 - c. no repetitions of pairs like (0, 1) and (1, 0)
 - d. no repeated positions within a single tuple, e.g., (0, 0)

Block / binned combination policies

Tuples of elements sharing **a common value** in a specified column, e.g., **a bin number**.

Analogously to basic policies, we have full / upper / strictly upper block combinations.

Different tables:

CombinationsBlockUpperIndexPolicy

CombinationsBlockFullIndexPolicy

Tuples from the same table:

CombinationsBlockUpperSameIndexPolicy

CombinationsBlockFullIndexPolicy

CombinationsBlockStrictlyUpperIndexPolicy

Helper functions (shortcuts)

Accepts only **same type tables**, applies **block strictly upper** policy

type – e.g., `aod::Tracks`.
2 different track partitions
have the same C++ type!

```
selfCombinations(binningPolicy, categoryNeighbours, outsider, tables...)
```

```
selfPairCombinations(binningPolicy, categoryNeighbours, outsider, table)
```

```
selfTripleCombinations(binningPolicy, categoryNeighbours, outsider, table)
```

If tables are of the same type, applies **block strictly upper**, otherwise **block upper** policy

```
combinations(binningPolicy, categoryNeighbours, outsider, tables...)
```

If tables are the same, applies **strictly upper**, otherwise **upper** policy

```
combinations(tables...)
```

```
pairCombinations(table), tripleCombinations(table)
```

Applies **selected combination policy**

```
combinations(combinationPolicy)
```

Using combinations

See some examples in [tracksCombinations.cxx](#) tutorial and [O2 documentation](#)

```
void process(aod::Tracks const& tracks) {  
    for (auto& [track1, track2] : combinations(tracks, tracks)) { ... } // Strictly upper tracks  
}
```

tuple size deduced from the number of arguments

```
void process(Tracks const& tracks, V0s const& v0s) {  
    for (auto& [track, v0] : combinations(CombinationsFullIndexPolicy(tracks, v0s))) { ... }  
}
```

```
struct BinnedTrackCombinations {  
    std::vector<double> xBins{VARIABLE_WIDTH, -0.064, -0.062, -0.060, 0.066, 0.068, 0.070, 0.072};  
    std::vector<double> yBins{VARIABLE_WIDTH, -0.320, -0.301, -0.300, 0.330, 0.340, 0.350, 0.360};  
    ColumnBinningPolicy<aod::track::X, aod::track::Y> trackBinning{{xBins, yBins}, true};  
  
    void process(aod::Tracks const& tracks)  
    {  
        // Strictly upper tracks binned by x and y position  
        for (auto& [t0, t1] : selfCombinations(trackBinning, 5, -1, tracks, tracks)) { ... }  
    }  
};
```

Additional parameters for block combinations

```
selfCombinations(trackBinning, 5, -1, tracks, tracks)
```

Outsider: bin number that should be skipped, e.g., -1 that marks bin over- and underflow

For performance reasons, we do not want to combine tuples across the whole bin, but only in smaller bin segments ("**sliding windows**").

The window size is equal to the parameter **category neighbours** + 1.

Example

category neighbours: 4, sliding window size: 5

row numbers in a bin: 1, 3, 5, 6, 10, 13, 16, 19, 23, 26, 29, 34, 36, 38



strictly upper pairs: (1, 3), (1, 5), (1, 6), (1, 10), (3, 5), (3, 6), (3, 10), (3, 13), (5, 6), (5, 10), (5, 13), (5, 16)

To get the behavior without sliding windows, set category neighbours to a very high value.

Weighted combinations

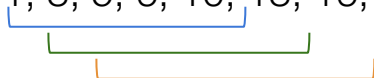
You might need to calculate **weights** for your event mixing. You can get useful variables:

- `currentWindowNeighbours()`
 - the number of other collisions to pair with
 - smaller if we are at the end of the sliding window or bin
- `bool isNewWindow()` – true only for the first pair from each sliding window

Example

category neighbours: 4, sliding window size: 5

row numbers in a bin: 1, 3, 5, 6, 10, 13, 16, 19, 23, 26, 29, 34, 36, 38



strictly upper pairs: (1, 3), (1, 5), (1, 6), (1, 10), (3, 5), (3, 6), (3, 10), (3, 13), (5, 6), (5, 10), (5, 13), (5, 16)

`currentWindowNeighbours()`: 4, 3, 2, 1, 4, 3, 2, 1, 4, 3, 2, 1

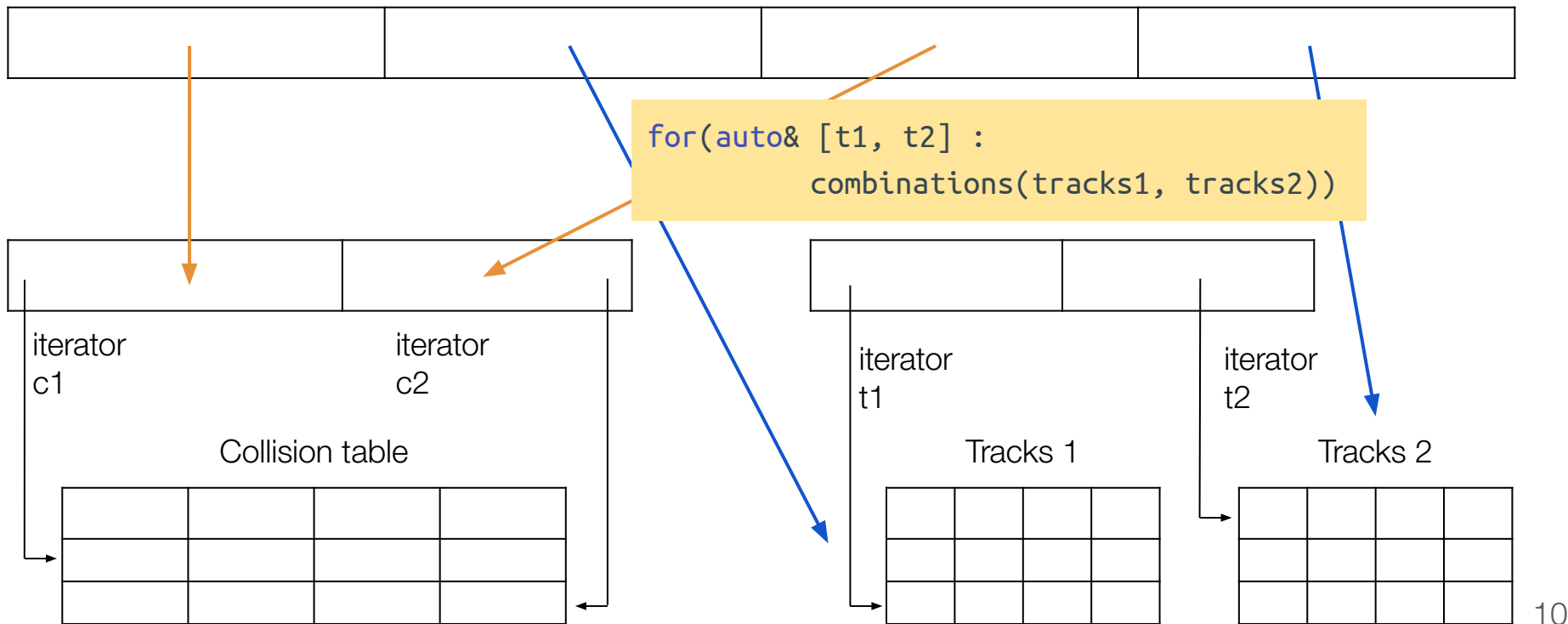
NOTE: Different behaviour for upper and full index policy, use with caution!

Code example: a [mixing test](#), real-life analysis: [Jan Fiete's correlations](#)

Event mixing

```
ColumnBinningPolicy<collision::PosX, collision::PosY> binning{{xBins, yBins}};  
SameKindPair<aod::Collisions, aod::Tracks> pair{binning};  
for (auto& [c1, tracks1, c2, tracks2] : pair) { ... }
```

tracks1 (tracks2)
contains only tracks
from the collision **c1 (c2)**



Using event mixing

See examples in the [eventMixing.cxx](#) tutorial described in O2 documentation [here](#) and [here](#).

BinningPolicy: array of bins, bool `ignoreOverflows` – if true, then under- and overflow values get bin -1

SameKindPair: binning policy, number of events to mix, bin number to ignore

```
struct MixedEvents {
    SliceCache cache;
    std::vector<double> xBins{VARIABLE_WIDTH, -0.064, -0.062, -0.060, 0.066, 0.068, 0.070, 0.072};
    std::vector<double> yBins{VARIABLE_WIDTH, -0.320, -0.301, -0.300, 0.330, 0.340, 0.350, 0.360};

    using BinningType = ColumnBinningPolicy<aod::collision::PosX, aod::collision::PosY>;
    BinningType binningOnPositions{{xBins, yBins}, true};
    SameKindPair<aod::Collisions, aod::Tracks, BinningType> pair{binningOnPositions, 5, -1, &cache};

    void process(aod::Collisions const& collisions, aod::Tracks const& tracks) {
        for (auto& [c1, tracks1, c2, tracks2] : pair) {
            // example of using tracks from mixing - iterate over all track pairs
            for (auto& [t1, t2] : combinations(CombinationsFullIndexPolicy(tracks1, tracks2))) { ... }
        }
    }
};
```

same order of
bins args!

Mixing types

SameKindPair: pairs of same associated tables, e.g., tracks

Pair: pairs of possibly different tables, e.g., tracks + V0s

SameKindTriple, Triple: analogously

```
struct MixedEventsTripleVariousKinds {
    SliceCache cache;
    BinningType binningOnPositions[{xBins, yBins, zBins}, true];
    Triple<aod::Collisions, aod::Tracks, aod::V0s, aod::Tracks, BinningType>
        triple{binningOnPositions, 5, -1, &cache};
    void process(aod::Collisions const& collisions, aod::Tracks const& tracks, aod::V0s const& v0s)
    {
        // tracks1 is an aod::Tracks table of tracks belonging to collision c1 (aod::Collision::iterator)
        // tracks2 is an aod::V0s table of V0s belonging to collision c2 (aod::Collision::iterator)
        // tracks3 is an aod::Tracks table of tracks belonging to collision c3 (aod::Collision::iterator)
        for (auto& [c1, tracks1, c2, tracks2, c3, tracks3] : triple) {
            for (auto& [t1, t2, t3] : combinations(CombinationsFullIndexPolicy(tracks1, tracks2, tracks3)))
                { ... }
        }
    }
};
```

Even more universality

Other mixed pairs than strictly upper:

```
using BinningType = ColumnBinningPolicy<aod::collision::PosX, aod::collision::PosY>;  
using GroupingPolicy = o2::soa::CombinationsBlockFullIndexPolicy<BinningType, int,  
                                                                    aod::Collisions, aod::Collisions>  
Pair<aod::Collisions, aod::Tracks, aod::Tracks, BinningType, int, GroupingPolicy>
```

This will repeat collision pairs like (0, 1) and (1, 0) – probably you won't use it in most cases.

Going beyond pair/tuples:

```
GroupedCombinationsGenerator(binningPolicy, categoryNeighbours, outsider, groupingTable,  
                             associatedTablesTuple)
```

No predefined binning policy for tuples bigger than triples – you need to write it yourself

Using dynamic columns

Most prominent example: mixing in z-vertex and **multiplicity V0M** bins.

Full code [here](#).

```
struct MixedEventsDynamicColumns {
    SliceCache cache;
    using aodCollisions = soa::Join<aod::Collisions, aod::Mults>;
    std::vector<double> zBins{7, -7, 7};
    std::vector<double> multBins{VARIABLE_WIDTH, 0, 5, 10, 20, 30, 40, 50, 100.1};
    using BinningType = ColumnBinningPolicy<aod::collision::PosZ,
                                             aod::mult::MultFV0M<aod::mult::MultFV0A, aod::mult::MultFV0C>;>;
    BinningType corrBinning{{zBins, multBins}, true};
    SameKindPair<aodCollisions, aod::Tracks, BinningType> pair{corrBinning, 5, -1, &cache};

    void process(aodCollisions& collisions, aod::Tracks const& tracks) {
        for (auto& [c1, tracks1, c2, tracks2] : pair) {
            for (auto& [t1, t2] : combinations(CombinationsFullIndexPolicy(tracks1, tracks2))) { ... }
        }
    }
};
```

Lambda binning policy

Sometimes binning parameters are **more complex** than a single column.
For example: multiplicity defined as `tracks.size()`.

Lambda policy: **user-defined calculation of bin numbers.**
Only inside `process()`

[Tutorial](#)
Real-life example:
[HF flow task](#)

```
auto getTracksSize =
    [&tracks](aod::Collision const& col) {
        auto associatedTracks = tracks.sliceByCached(o2::aod::track::collisionId, col.globalIndex());
        return associatedTracks.size();
    };
using BinningType = FlexibleBinningPolicy<std::tuple<decltype(getTracksSize)>,
                                           aod::collision::PosZ, decltype(getTracksSize)>;
BinningType binningWithLambda{{getTracksSize}, {axisVertex, axisMultiplicity}, true};

auto tracksTuple = std::make_tuple(tracks);
SameKindPair<aod::Collisions, aod::Tracks, BinningType> pair{binningWithLambda, 5, -1,
                                                            collisions, tracksTuple, &cache};
```



Thank you for your
attention!

Backup