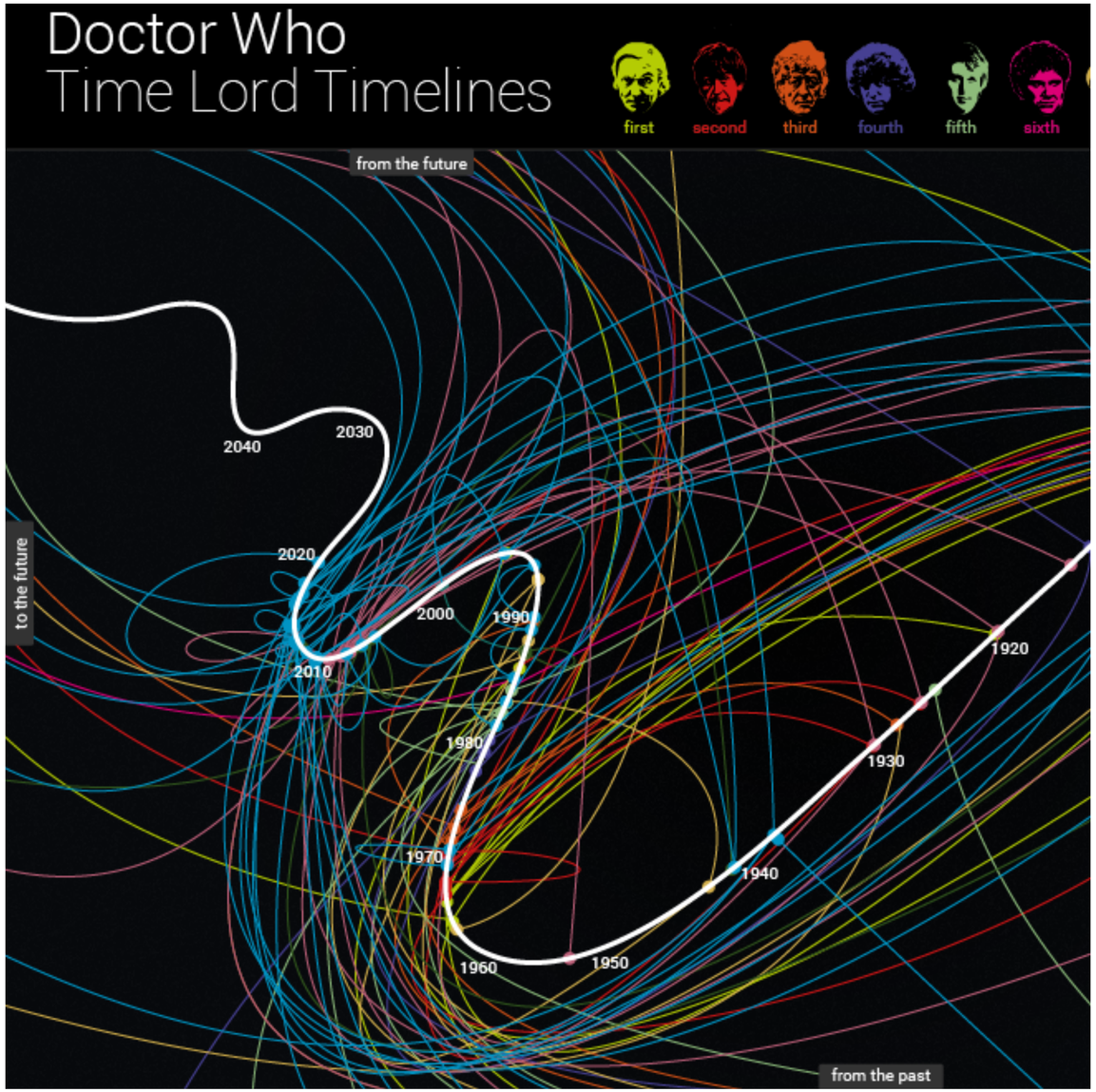
Time Flies Like a Duck

Richard Wesley

DuckDB Labs

Temporal Joins



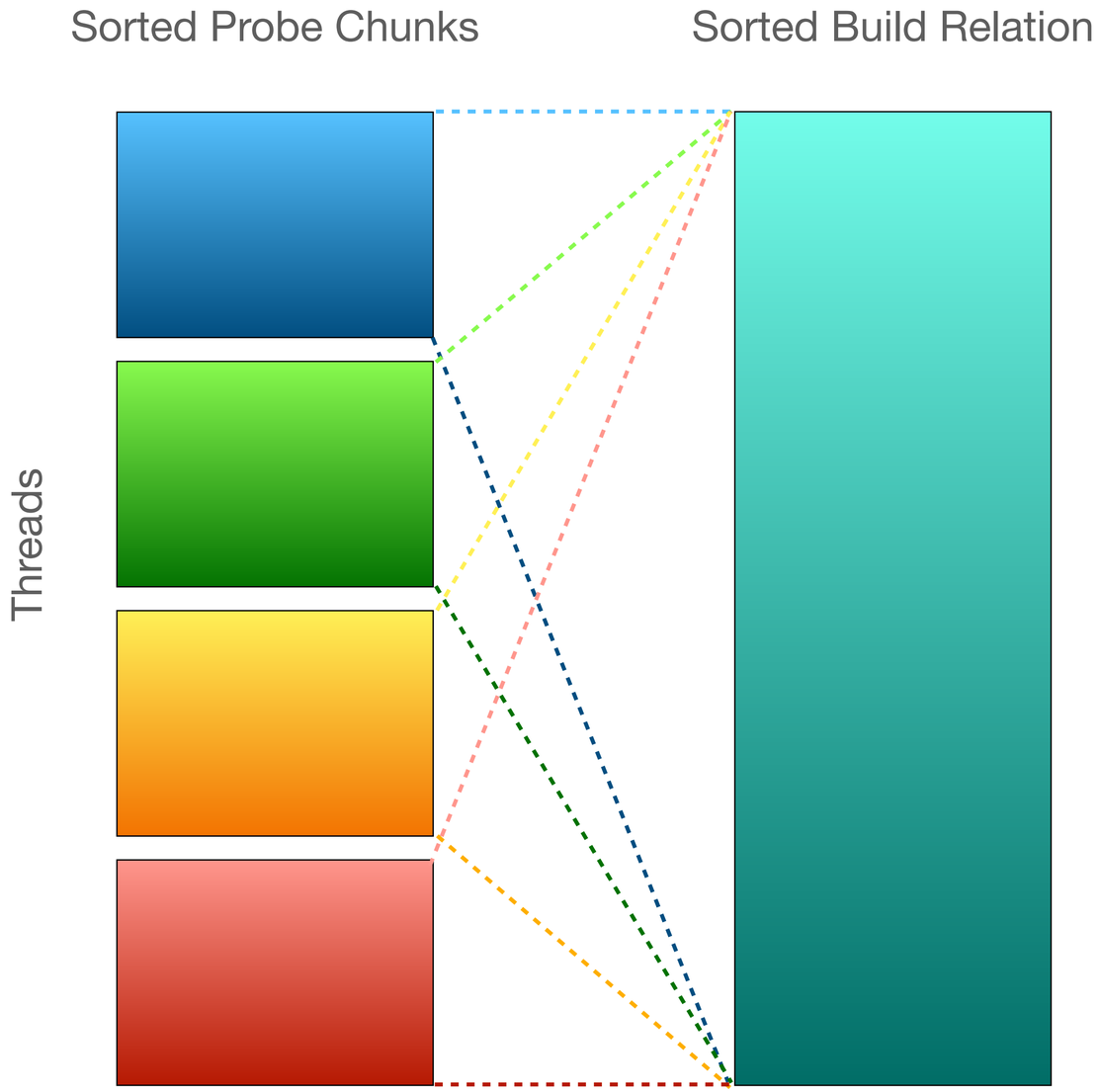
BBC Time Line Graph for Dr Who

Temporal Joins use non-equality predicates and require appropriate algorithms

Single Inequality Joins

- Piecewise Merge Join
  - Sort build side
  - **Sort probe chunks**
  - **Thread-local** merge join
  - Apply equality predicates

- Planning
  - Equalities not all selective!
  - **Cardinality** estimation?



Piecewise Merge Join

Image text alternative

Range Joins

callsign	craft	begin	end
Starbuck	2794NC	3004-05-04 13:22:12	3004-05-04 15:05:49
Apollo	2794NC	3004-05-04 10:00:00	3004-05-04 18:19:12
Boomer	312	3004-05-04 13:33:52	3004-05-05 19:12:21
Husker	N7242C	3008-03-20 08:14:37	3008-03-20 10:21:15
…	…	…	…

battle	begin	end
Fall of the Colonies	3004-05-04 13:21:45	3004-05-05 02:47:16
Red Moon	3004-05-28 07:55:27	3004-05-28 08:12:19
Tylium Asteroid	3004-06-09 09:00:00	3004-06-09 11:14:29
Resurrection Ship	3004-10-28 22:00:00	3004-10-28 23:47:05
…	…	…

callsign	battle	begin	end
Starbuck	Fall of the Colonies	3004-05-04 13:22:12	3004-05-04 15:05:49
Apollo	Fall of the Colonies	3004-05-04 13:22:12	3004-05-04 18:19:12
Boomer	Fall of the Colonies	3004-05-04 13:33:52	3004-05-05 02:47:16
…	…	…	…

- Use cases:
  - Period **overlap**
  - Joint **state tables**
  - Often only one matching row
- IEJoin
  - Any two inequalities
  - Best **general purpose algorithm**
- Similar Planning Issues
  - Cardinality estimation?

https://duckdb.org/2022/05/27/iejoin.html

AsOf Joins

- Find **closest timestamp**
  - Value "**as of**" this time
- Joins **two time series**
  - (But works for any type)

- Use cases
  - Stock price as of timestamp
  - Utility time-of-use tariffs
  - Event table lookup

Stock Ticker Chyron

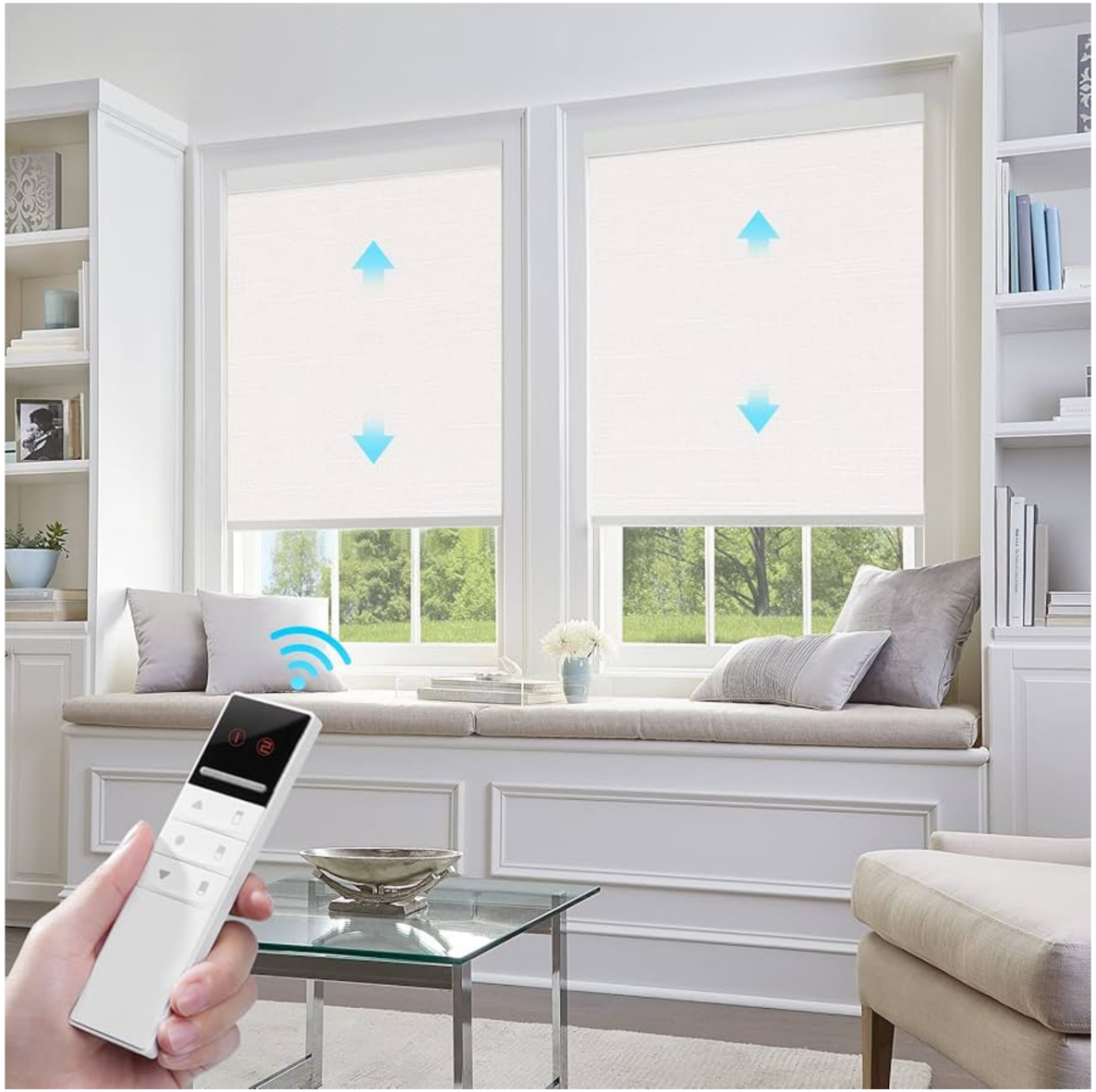
Positional Joins

- Join on **row number**
  - Hard to express in SQL
  - Common with DataFrames

- Two (or more!) inputs
  - **Streams** all table scans
  - Other inputs materialised
  - Outer join for unequal lengths
  - Only **serial execution**

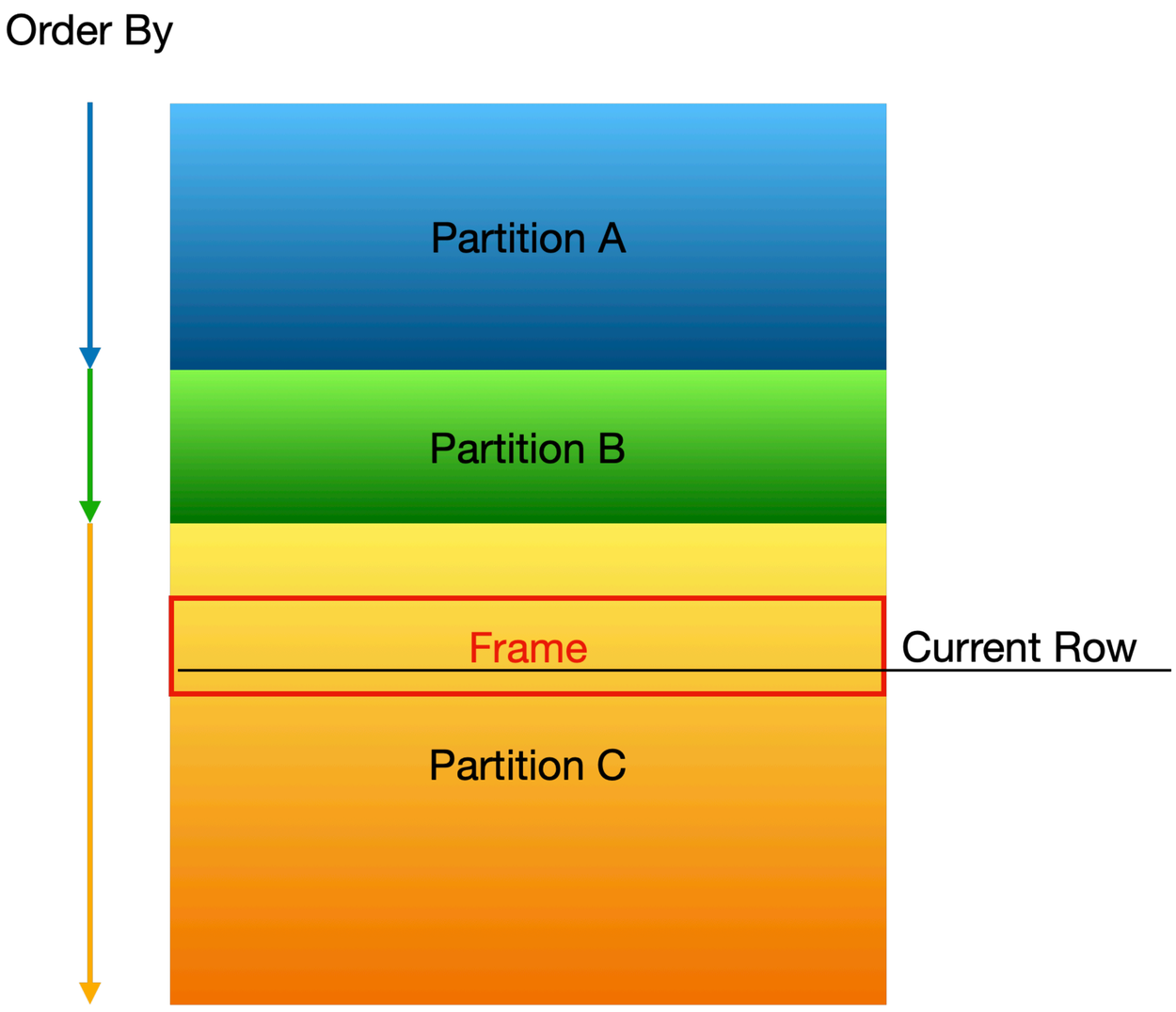
Grid of Rubber Ducks

The Window Operators



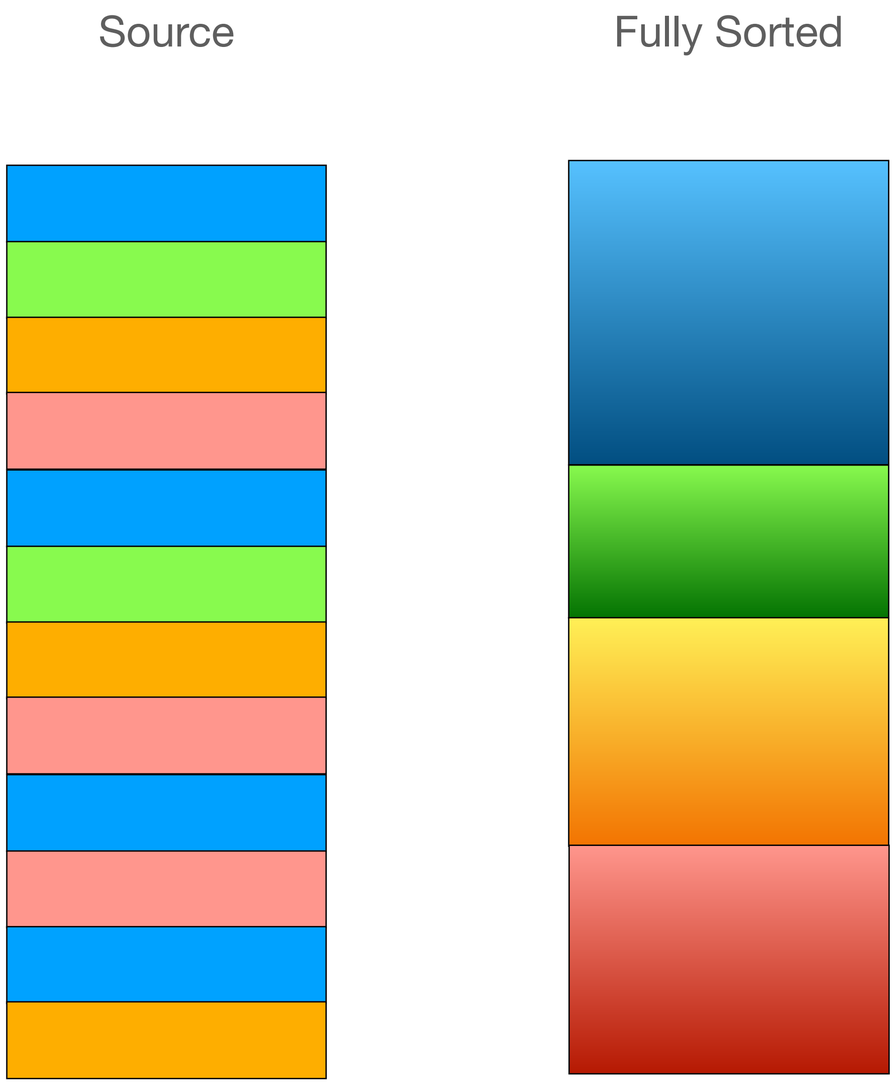
The Windowing Model

- **`PARTITION BY`**
  - Independent blocks of rows
- **`ORDER BY`**
  - Sort the partitions
- **`ROWS`/`RANGE`/`GROUPS BETWEEN`**
  - Distance from the current row
  - `EXCLUDE` around current row



Traditional Window Operators

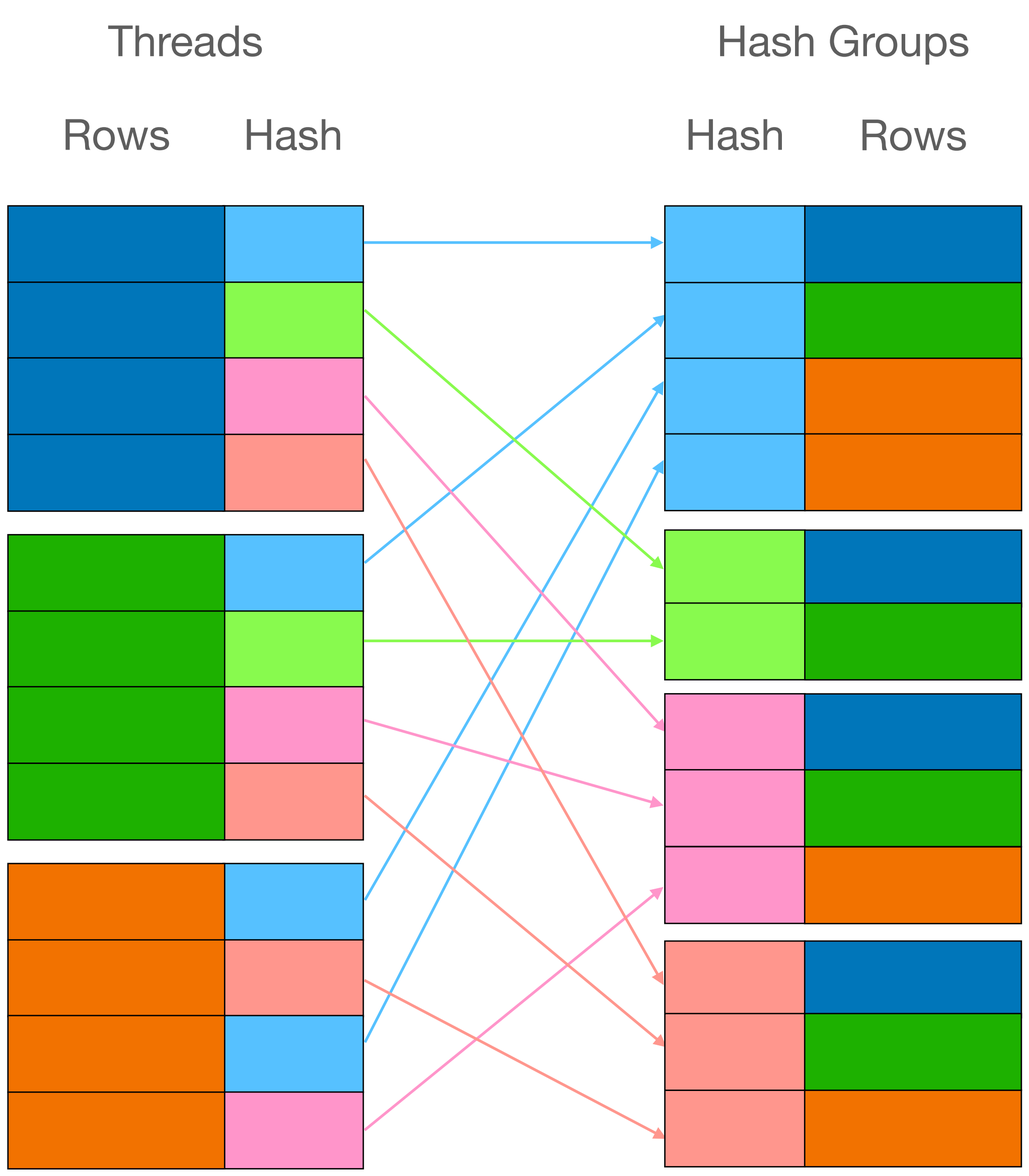
- **Materialise everything**
- Sort on **Order AND Partition**
- **Multiple functions**
  - Group by full sorting spec
- Sorting spec evaluation order?
  - NP-hard (Cao et al.)



Fully Sorted Window Relation

Cao et al., Optimization of Analytic Window Functions, VLDB 2012

Hash Grouping



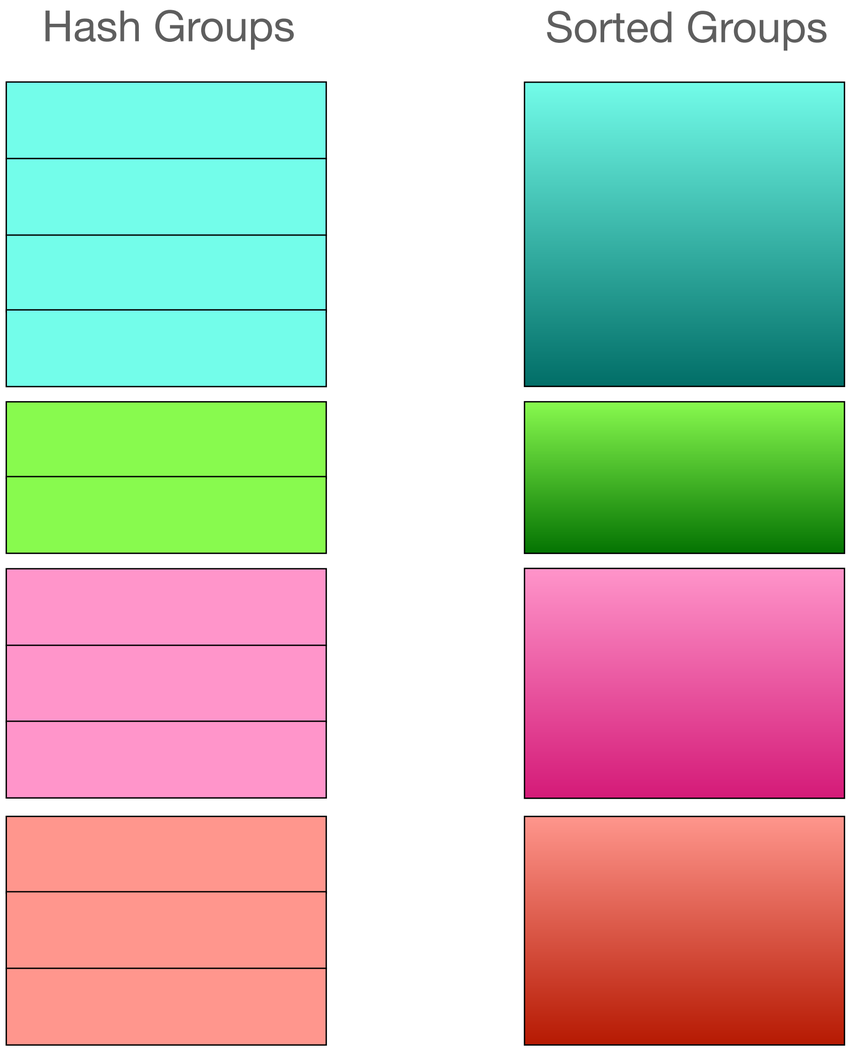
- **Hash incoming chunks**
  - Lies et al., VLDB 2015
  - Up to 128 hash groups
- Only **hash the partition keys**
  - Reduces sorting size
  - `N * log (N/p)`
- Hash collisions?
  - **Multiple partitions** per group

Lies et al., Efficient Processing of Window Functions in Analytical SQL Queries, VLDB 2015

Sorting

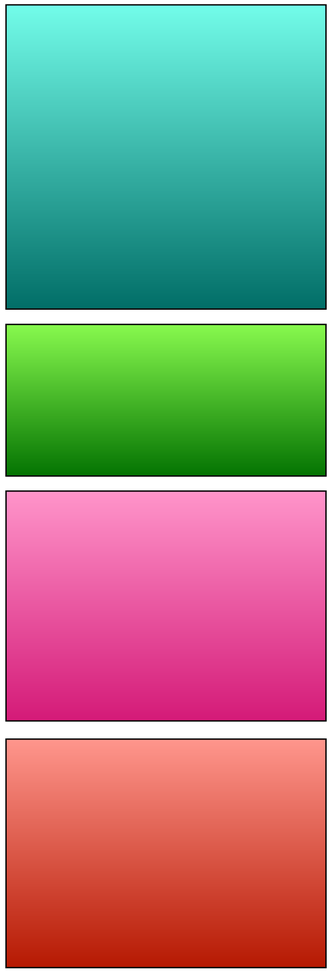
- Set up `Tasks` in `Finalize`
  - Uses **latest sorting library**
  - Same code as `ORDER BY`
  - Sorts include partition keys

- Sorts **not memory limited**
  - Can spill to disk
  - **Results also paged**



Sorted Hash Groups

Boundary Masks



Sorted Groups	Masks: Partition	Masks: Order	
	1	1	
	0	0	← Duplicate / Peer
	0	1	
	1	1	← New Partition
	1	1	← New Hash Group
	0	1	
	1	1	
	0	0	
	0	1	
	1	1	
	1	1	
	0	0	

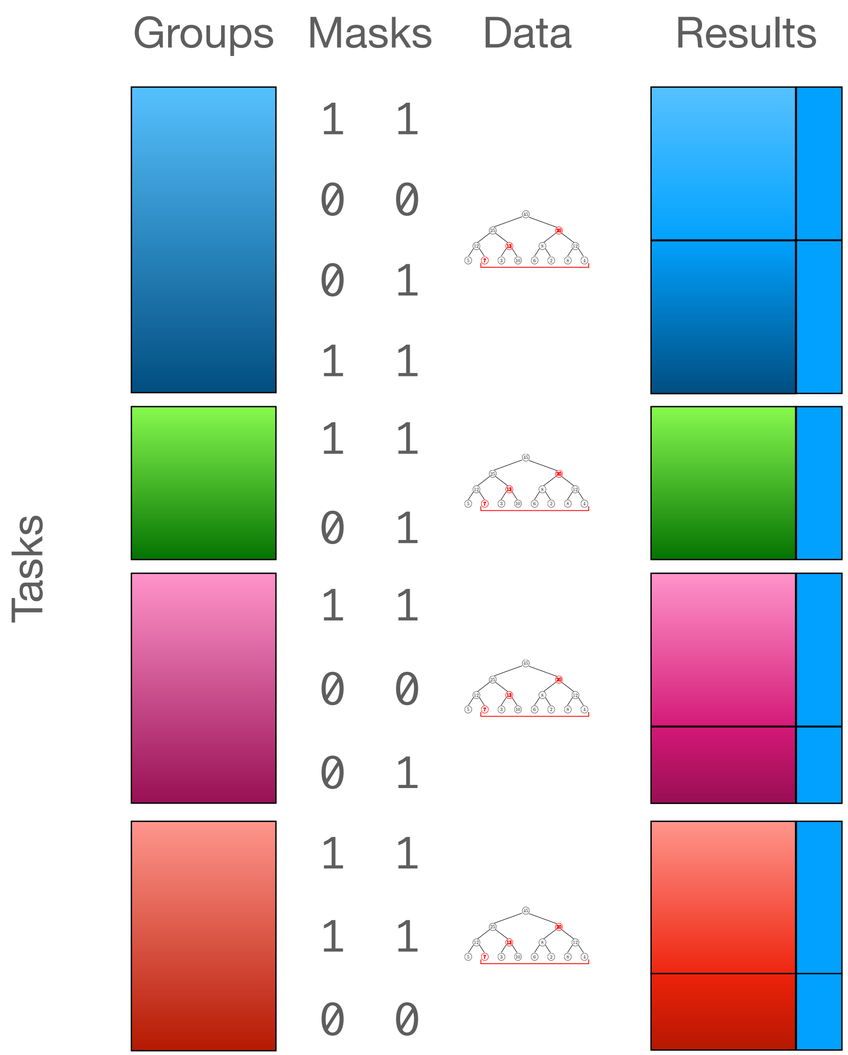
- Need to **track boundaries**
  - Multiple partitions per group
  - Multiple sorts (prefixes)
  - **Peers** for `RANK` and `RANGE`

- Build masks in parallel
  - Use **sort keys**
  - New sort code provides them

Windowing Resource Management

- Minimise memory footprint
  - **One partition at a time**
  - Use all threads
  - Size scheduling to largest
  - Smaller partitions can share

- Do **everything in parallel**
  - Build masks
  - Build **acceleration** structures
  - Evaluate results



https://duckdb.org/2025/02/14/window-flying.html

Source Task Diagram

Streamed Windowing

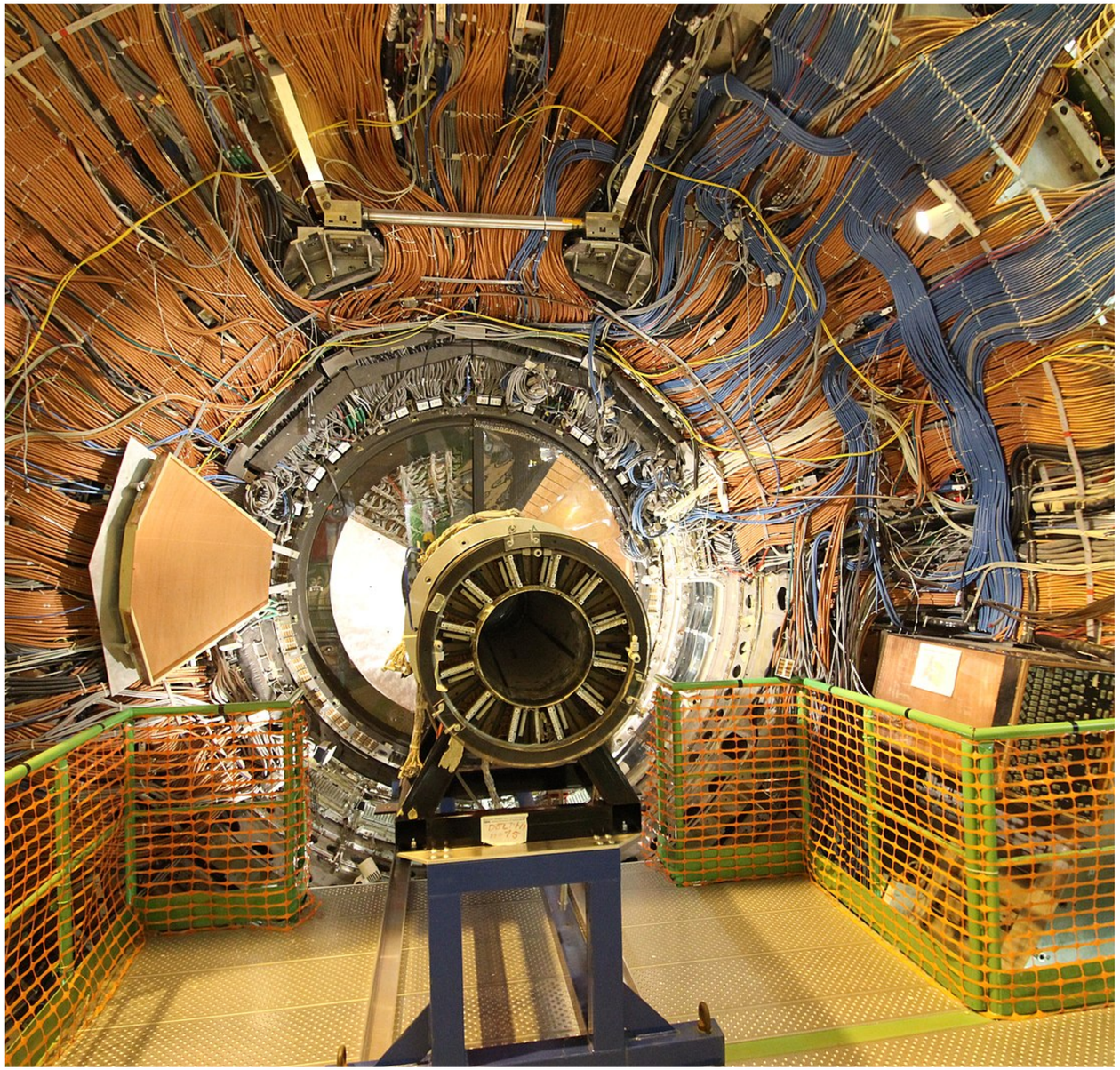
- Can we **stream evaluation**?
  - Uses "natural order"
  - **`OVER(ROWS...)`**
  - No `IGNORE NULLS`
  - "Simple" non-aggregates
  - "**Running total**" aggregates

- Recent functionality
  - `FILTER` and `DISTINCT`
  - `LEAD` and `LAG` (< 2048)

https://duckdb.org/2025/02/14/window-flying.html

Window Functions

Aggregation Accelerators



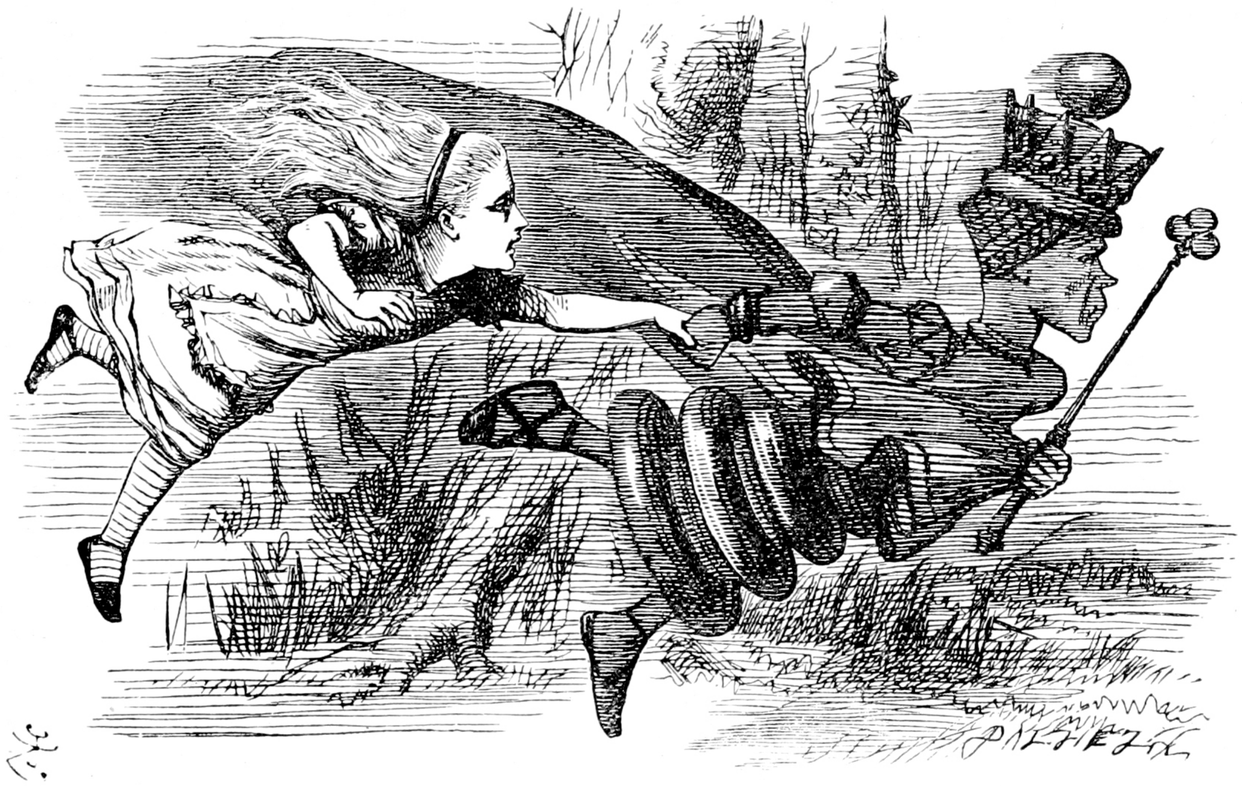
- Na ve evaluation is s l o w!
  - Independent row evaluation
  - No history reuse

- Accelerators
  - **Single Value**
  - **Segment Trees**
  - **Merge Sort Trees**
  - Custom Window APIs
  - Na ve (for testing)

Single Value

- Unsorted frames
  - No `ORDER BY`
  - Frame is **entire partition**
  - Only **one value**

- We detect this
  - Only **compute it once**
  - Copy to all rows
  - Often constant vector



Alice and the Red Queen, John Tenniel, Public Domain

Segment Trees

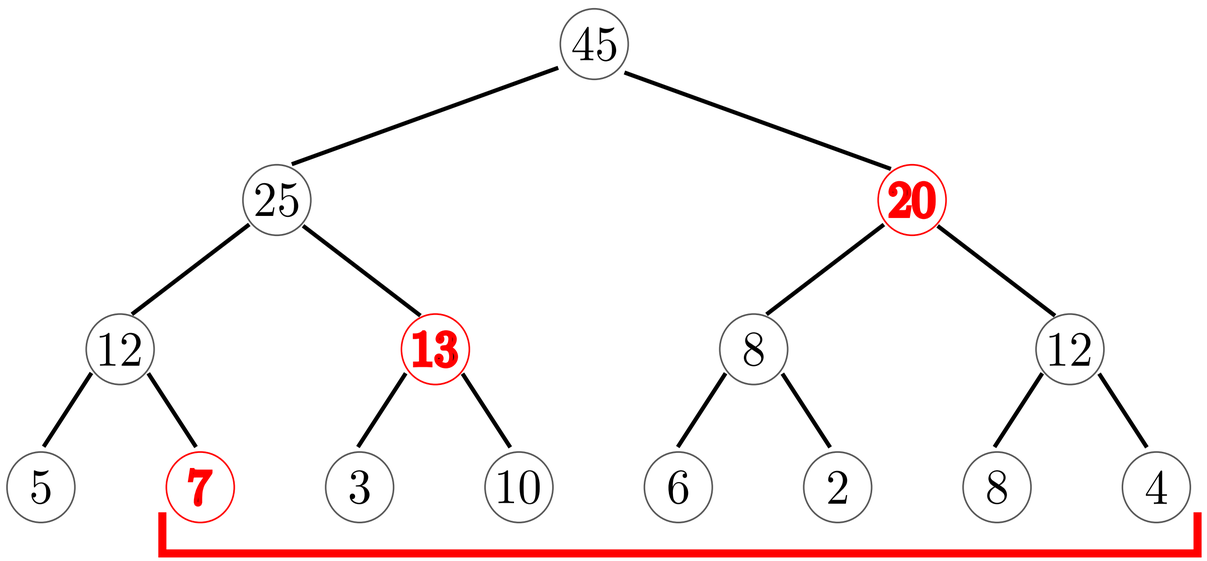


Figure 5: Segment Tree for `sum` aggregation. Only the red nodes (7, 13, 20) have to be aggregated to compute the sum of 7, 3, 10, 6, 2, 8, 4

- **Small data structure**
  - Can be built in parallel
- **Read-only evaluation**
  - Values at bottom of tree
  - Higher nodes are states
  - Evaluation builds new states
- Handles **any aggregation**
  - `SUM`, etc.
  - Can't handle `DISTINCT`

Merge Sort Trees

- Window has one frame order

- Function wants another:
  - **`DISTINCT`** aggregates
  - **`quantile/mad`**
  - **`rank`** by a different order

- Doubly ordered tree!
  - **Built and queried in parallel**

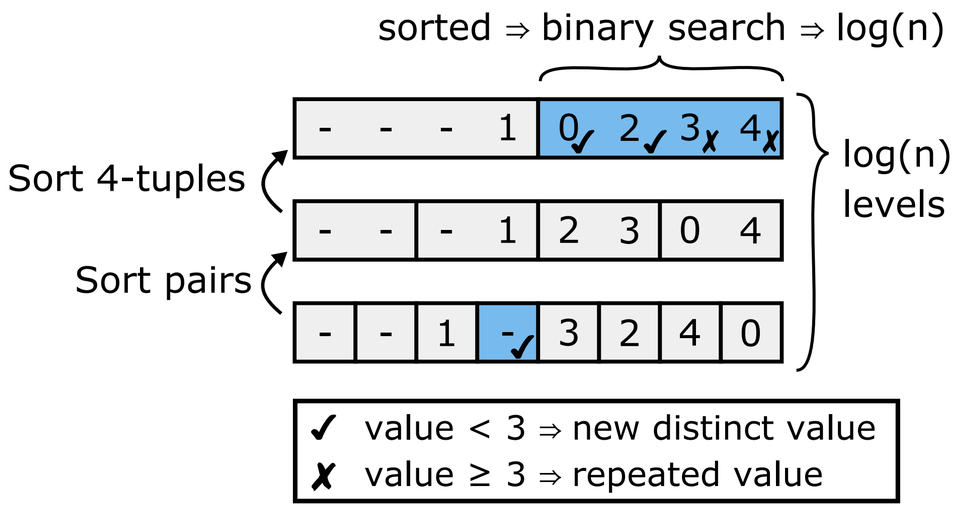
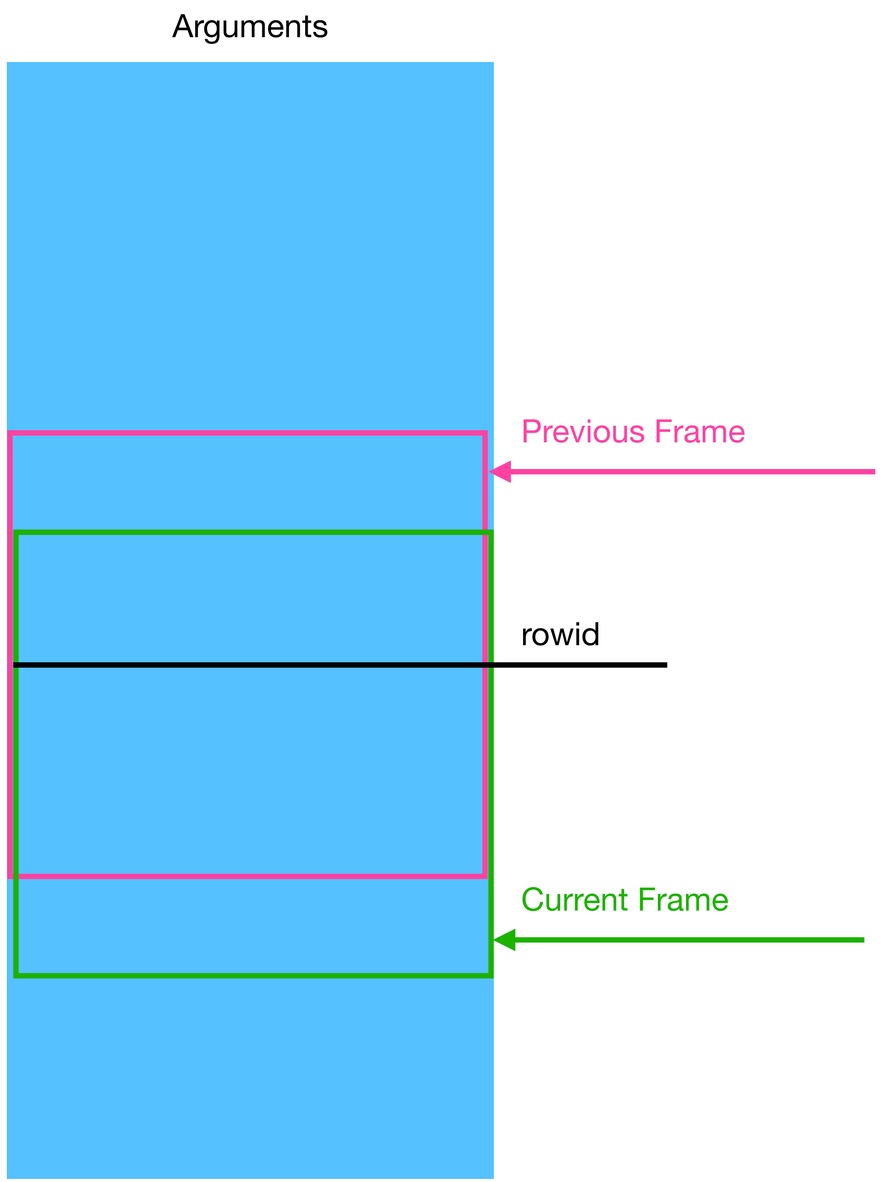


Figure 2: A merge sort tree improves query time to $O(n(\log n)^2)$ by utilizing a tree of sorted lists.

https://duckdb.org/2025/02/10/window-catchup.html

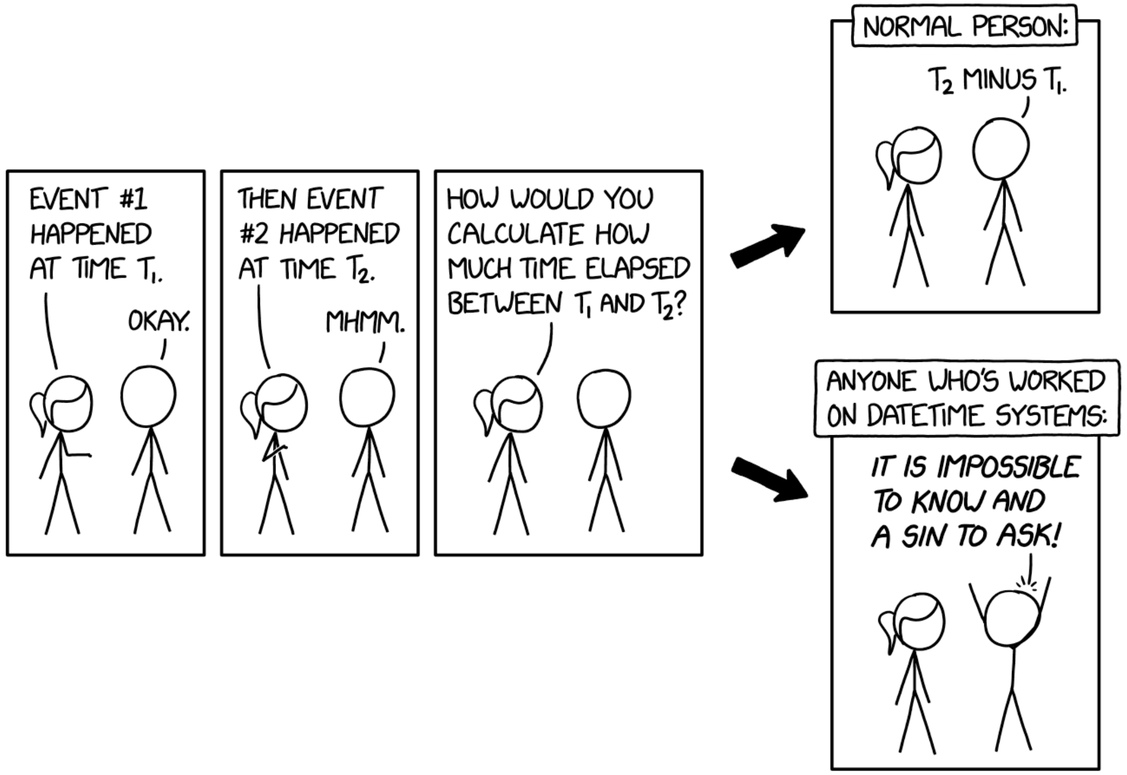
Basic Merge Sort Tree, from Vogelsgesang et al.

Custom Window APIs



- Segment trees **big and slow**
  - `count(*)`
  - `quantile`, `mode`, `mad`
- Optional **`window` API**
  - All the input values
  - Persistent local state
  - Current `rowid` and bounds
- Optional **`window_init`** API
  - Persistent global state

Temporal Functions



People assume that time is a strict progression of cause to effect, but actually from a non-linear, non-subjective viewpoint it's more like a big ball of wibbly-wobbly timey-wimey stuff.

– Doctor Who: Blink

https://xkcd.com/2867/

Binning Support

- Regular Timestamps
  - Fast **Gregorian-UTC library**
- Timestamp With Time Zone
  - **ICU extension** (preloaded)
  - Set TimeZone (and Calendar)
- Binning **can be slow** (esp. ICU)
  - DatePart part lists => structs
  - Pre-build calendar table

Binned Produce

https://duckdb.org/2022/01/06/time-zones.html

Temporal Aggregation

- Useful Aggregates
  - `arg_max / arg_min(value, ordering)`
  - (Also `arg_min_n / arg_max_n`
  - Value at min/max of ordering
  - **Avoids windowing and sorting**!
- Argument Sorting
  - `agg(args ORDER BY ordering)`
  - Example: `string_agg`

https://duckdb.org/docs/sql/aggregates

References

- Cao et al., *Optimization of Analytic Window Functions,* VLDB 2012
- Khayyat et al., *Lightning Fast and Space Efficient Inequality Joins*, VLDB 2015
- Lies et al., *Efficient Processing of Window Functions in Analytical SQL Queries*, VLDB 2015
- Wesley & Xu, *Incremental Computation of Common Windowed Holistic Aggregates*, VLDB 2016
- Kohn et al., *Building Advanced SQL Analytics From Low-Level Plan Operators*, SIGMOD 2021
- Vogelsgesang et al., *Efficient Evaluation of Arbitrarily-Framed Holistic SQL Aggregates and Window Functions*, SIGMOD 2022
- Bača, *Window Function Expression: Let the Self-join Enter*, VLDB 2024

Question Time

Inquisitive Whio