~~Inequality Joins in Pandas with Pyjanitor~~

Presented by Samuel Oranyeli

Efficient Range Joins in Pandas

Presented by Samuel Oranyeli

About Me

- Data Engineer
- Open source contributor

Contents

- Equi joins vs non-equi joins
- What are range joins
- Why are range joins slow?
- A faster option - conditional_join
- Performance

References

- https://select.dev/posts/snowflake-range-join-optimization
- https://www.vertica.com/blog/what-is-a-range-join-and-why-is-it-so-fastba-p223413/
- https://duckdb.org/2022/05/27/iejoin.html
- https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/AnalyzingData/Queries/Joins/RangeJoins.htm
- Dathan, B et al.
- https://db.in.tum.de/~reif/papers/p2535-reif.pdf
- https://stackoverflow.com/q/24480031/7175713
- Zuhair Khayyat et al.

Equi joins vs non-equi joins

```
select
  ...
from orders
join customers
  on orders.customer_id=customers.id -- example equi-join condition
```

```
select
  ...
from orders
inner join customers
  on orders.customer_id=customers.id  -- equi-join
  and orders.created_at > customers.one_year_anniversary_date -- non-equi join
```

```
orders.merge(customers, on = 'customer_id', how = 'inner')
```

```
(orders
 .merge(
     customers,
     on = 'customer_id',
     how = 'inner')
 .loc[lambda df: df.created_at > df.one_year_anniversary_date]
)
```

What are range joins?

Range Joins

Point in interval join

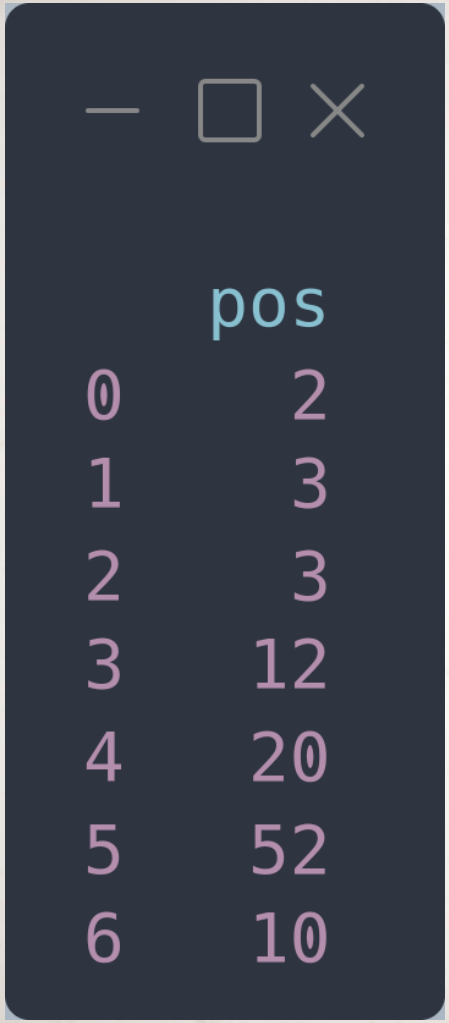
```
SELECT owner_id, COUNT(*)
FROM clicks JOIN ip_owners
ON clicks.dest_ip
BETWEEN ip_start AND ip_end
GROUP BY owner_id;
```

Overlap join

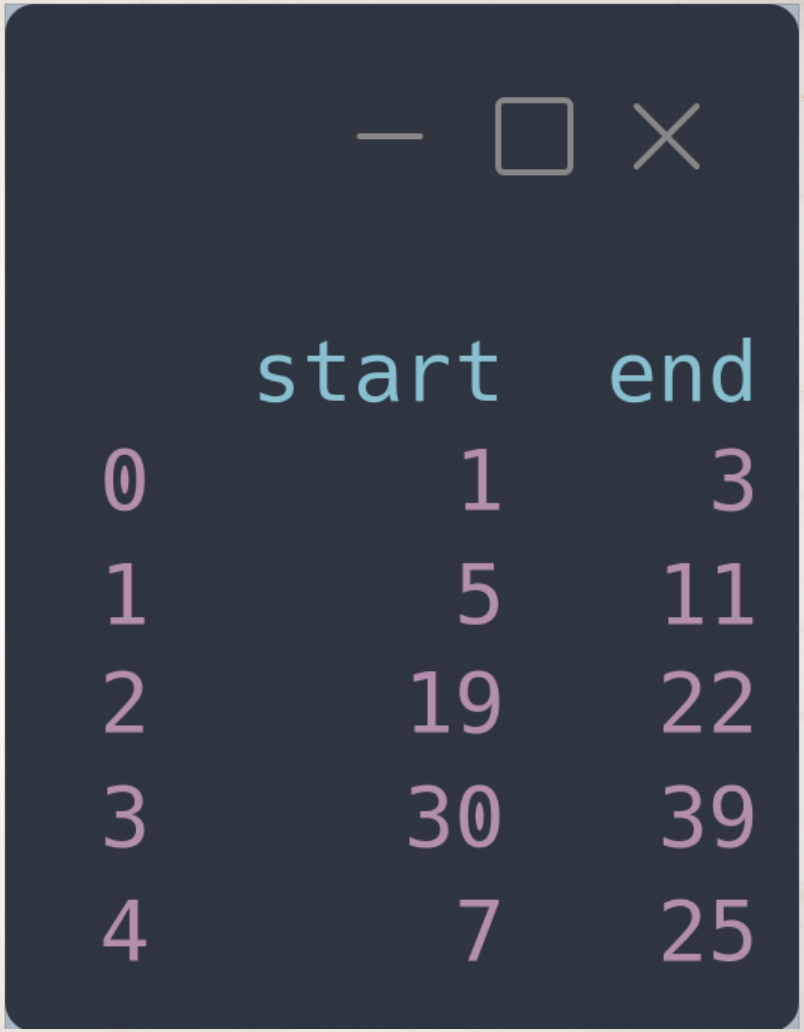
```
SELECT
    r.id,
    s.id
FROM events r
JOIN events s
    ON r.start <= s.end
    AND r.end >= s.start
    AND r.id <> s.id
```

Why are range joins slow?

- Range joins in pandas are usually computed via a cartesian join.
- For relatively large datasets, this becomes expensive memory wise, and is generally slow.
- IntervalIndex is an excellent, performant option, but is limited to scenarios where the intervals do not overlap.



	pos
0	2
1	3
2	3
3	12
4	20
5	52
6	10



	start	end
0	1	3
1	5	11
2	19	22
3	30	39
4	7	25

```
(df1
 .merge(df2, how='cross')
 .loc[lambda df: df.pos.between(df.start, df.end)]
)
```

	pos	start	end
0	2	1	3
5	3	1	3
10	3	1	3
19	12	7	25
22	20	19	22
24	20	7	25
31	10	5	11
34	10	7	25

Return rows where df1.pos
is between df2.start and df2.end

df1

	pos
0	2
1	3
2	3
3	12
4	20
5	52
6	10

df2

	start	end
0	1	3
1	5	11
2	19	22
3	30	39
4	7	25

cartesian product of df1 and df2

	pos	start	end
0	2	1	3
1	2	5	11
2	2	19	22
3	2	30	39
4	2	7	25
5	3	1	3
6	3	5	11
7	3	19	22
8	3	30	39
9	3	7	25
10	3	1	3
11	3	5	11
12	3	19	22
13	3	30	39
14	3	7	25
15	12	1	3
16	12	5	11
17	12	19	22
18	12	30	39
19	12	7	25
20	20	1	3
21	20	5	11
22	20	19	22
23	20	30	39
24	20	7	25
25	52	1	3
26	52	5	11
27	52	19	22
28	52	30	39
29	52	7	25
30	10	1	3
31	10	5	11
32	10	19	22
33	10	30	39
34	10	7	25

range condition applied after cartesian join

	pos	start	end
0	2	1	3
5	3	1	3
10	3	1	3
19	12	7	25
22	20	19	22
24	20	7	25
31	10	5	11
34	10	7	25

IntervalIndex wont be efficient here, since the intervals overlap.

Source: there must be a better way gif

conditional_join

```
conditional_join(
    df,
    right,
     *args,
    use_numba,
    df_columns,
    right_columns)
```

**args*

(left_column, right_column , comparator)

pip install pyjanitor

import janitor

```
(df1
 .conditional_join(
      df2,
      ('pos', 'start', '>='),
      ('pos', 'end', '<='))
)
```

	pos	start	end
0	2	1	3
1	3	1	3
2	3	1	3
3	12	7	25
4	20	7	25
5	20	19	22
6	10	5	11
7	10	7	25

Return rows where df1.pos
is between df2.start and df2.end

df1

	pos
0	2
1	3
2	3
3	12
4	20
5	52
6	10

df2

	start	end
0	1	3
1	5	11
2	19	22
3	30	39
4	7	25

- sort start in ascending order
- get furthest positions where pos is >= start
- get earliest position where pos is <= end

	start	end	cummax
0	1	3	3
1	5	11	11
4	7	25	25
2	19	22	25
3	30	39	39

reduced search space

	pos	bottom	top
0	2	1	0
1	3	1	0
2	3	1	0
3	12	3	2
4	20	4	2
~~5~~	~~52~~	~~4~~	~~4~~
6	10	3	1

top should be less than bottom

Do actual comparison

pos	start	end
2	1	3
3	1	3
12	7	25
20	19	22
20	7	25
10	5	11
10	7	25

final output

	pos	start	end
0	2	1	3
1	3	1	3
2	3	1	3
3	20	19	22
4	12	7	25
5	20	7	25
6	10	5	11
7	10	7	25


Sorting + Binary Search

Performance

```
In [6]: events.shape
Out[6]: (29999, 6)

In [7]: events.head()
Out[7]:
   id     name  audience               start     sponsor                 end
0   1  Event 1      1178 2022-11-19 10:00:00   Sponsor 2 2022-11-19 10:15:00
1   2  Event 2      1446 2015-09-27 15:00:00  Sponsor 11 2015-09-27 15:11:00
2   3  Event 3      2261 2019-11-12 18:00:00  Sponsor 10 2019-11-12 18:53:00
3   4  Event 4      1471 2019-12-24 22:00:00   Sponsor 6 2019-12-24 22:11:00
4   5  Event 5      2605 2028-06-20 12:00:00   Sponsor 8 2028-06-20 12:31:00
```

Source: Duck DB blog post

```
In [22]: (events
    ...: .conditional_join(
    ...:     events,
    ...:     ('start', 'end', '<='),
    ...:     ('end', 'start', '>='),
    ...:     ('id', 'id', '!='),
    ...:     use_numba = False,
    ...:     df_columns = ['id', 'start', 'end'],
    ...:     right_columns = ['id', 'start', 'end'])
    ...: )
Out[22]:
       left                                          right
         id               start                 end     id               start                 end
0        10 1993-11-27 12:00:00 1993-11-27 12:37:00   2345 1993-11-27 10:00:00 1993-11-27 12:00:00
1        15 1993-04-04 16:00:00 1993-04-04 18:00:00  11178 1993-04-04 17:00:00 1993-04-04 17:22:00
2        17 2030-10-25 07:00:00 2030-10-25 07:27:00  19605 2030-10-25 06:00:00 2030-10-25 08:00:00
3        26 2005-10-04 17:00:00 2005-10-04 17:18:00   8218 2005-10-04 17:00:00 2005-10-04 17:27:00
4        35 2024-05-02 15:00:00 2024-05-02 15:35:00   6916 2024-05-02 15:00:00 2024-05-02 15:36:00
...     ...                 ...                 ...    ...                 ...                 ...
3697  29966 2000-08-26 11:00:00 2000-08-26 13:00:00  29375 2000-08-26 13:00:00 2000-08-26 13:53:00
3698  29971 2018-05-18 04:00:00 2018-05-18 04:18:00  24173 2018-05-18 04:00:00 2018-05-18 04:36:00
3699  29978 1992-06-07 22:00:00 1992-06-07 22:23:00    981 1992-06-07 22:00:00 1992-06-07 22:30:00
3700  29984 2025-06-05 03:00:00 2025-06-05 03:17:00  19051 2025-06-05 01:00:00 2025-06-05 03:00:00
3701  29995 2016-09-04 14:00:00 2016-09-04 14:32:00  12296 2016-09-04 14:00:00 2016-09-04 14:50:00

[3702 rows x 6 columns]
```

```
In [23]: %%timeit
    ...: (events
    ...: .conditional_join(
    ...:     events,
    ...:     ('start', 'end', '<='),
    ...:     ('end', 'start', '>='),
    ...:     ('id', 'id', '!='),
    ...:     use_numba = False,
    ...:     df_columns = ['id', 'start', 'end'],
    ...:     right_columns = ['id', 'start', 'end'])
    ...: )
    ...:
    ...:
18.7 ms ± 719 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [15]: eventss = events.sample(5000)

In [16]: (eventss
    ...: .merge(eventss, how='cross')
    ...: .loc[lambda df: df.start_x.le(df.end_y) & df.end_x.ge(df.start_y),
    ...:      ['id_x','start_x', 'end_x', 'id_y','start_y','end_y']]
    ...: )
Out[16]:
           id_x             start_x               end_x   id_y             start_y               end_y
0          4479 2005-04-26 03:00:00 2005-04-26 03:42:00   4479 2005-04-26 03:00:00 2005-04-26 03:42:00
5001      22513 2012-12-17 20:00:00 2012-12-17 20:44:00  22513 2012-12-17 20:00:00 2012-12-17 20:44:00
10002     27926 2003-12-25 04:00:00 2003-12-25 04:41:00  27926 2003-12-25 04:00:00 2003-12-25 04:41:00
15003     12258 2010-07-15 18:00:00 2010-07-15 18:26:00  12258 2010-07-15 18:00:00 2010-07-15 18:26:00
20004     16255 2013-09-26 07:00:00 2013-09-26 09:00:00  16255 2013-09-26 07:00:00 2013-09-26 09:00:00
...         ...                 ...                 ...    ...                 ...                 ...
24979995  29724 1995-11-06 19:00:00 1995-11-06 19:07:00  29724 1995-11-06 19:00:00 1995-11-06 19:07:00
24984996   7677 2028-02-20 01:00:00 2028-02-20 01:47:00   7677 2028-02-20 01:00:00 2028-02-20 01:47:00
24989997    439 2001-04-14 13:00:00 2001-04-14 13:10:00    439 2001-04-14 13:00:00 2001-04-14 13:10:00
24994998  22785 2011-06-23 16:00:00 2011-06-23 18:00:00  22785 2011-06-23 16:00:00 2011-06-23 18:00:00
24999999  28536 2031-10-13 02:00:00 2031-10-13 02:50:00  28536 2031-10-13 02:00:00 2031-10-13 02:50:00

[5082 rows x 6 columns]
```

```
In [17]: %%timeit
    ...: (eventss
    ...: .merge(eventss, how='cross')
    ...: .loc[lambda df: df.start_x.le(df.end_y) & df.end_x.ge(df.start_y),
    ...:      ['id_x','start_x', 'end_x', 'id_y','start_y','end_y']]
    ...: )
    ...:
    ...:
2.07 s ± 101 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Performance

```
In [3]: flights.shape
Out[3]: (127426, 6)

In [4]: flights.head()
Out[4]:
             takeoff             landing  orig  dest               start                 end
0 2021-11-27 07:15:00 2021-11-27 08:55:00     0     0 2021-11-27 09:40:00 2021-11-27 11:55:00
1 2021-11-27 20:05:00 2021-11-28 00:50:00     1     1 2021-11-28 01:35:00 2021-11-28 03:50:00
2 2021-11-27 21:00:00 2021-11-27 21:35:00     2     2 2021-11-27 22:20:00 2021-11-28 00:35:00
3 2021-11-27 21:15:00 2021-11-27 22:25:00     0     2 2021-11-27 23:10:00 2021-11-28 01:25:00
4 2021-11-26 11:40:00 2021-11-26 14:45:00     3     3 2021-11-26 15:30:00 2021-11-26 17:45:00
```

```
%%timeit
# classic pandas merge and filter
outt = (flights
        .merge(flights, left_on='dest', right_on='orig')
        .loc[lambda f: f.takeoff_y.between(f.start_x, f.end_x) & (f.orig_x != f.orig_y),
             ['start_x', 'takeoff_y', 'start_y', 'dest_x', 'orig_y']
             ]
        )
8.24 s ± 304 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
# use indices from Pandas internal merge function
# compute the non-equi joins
# then create the dataframe
outer = (flights
 .conditional_join(
     flights,
     ('end', 'takeoff' ,'>='),
     ('start', 'takeoff', '<='),
     ('orig','orig','!='),
    ('dest', 'orig', '=='),
    df_columns = ['start', 'end', 'dest'],
     right_columns = ['takeoff', 'orig'],
     use_numba=False)
)
1.24 s ± 6.72 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
# there can be scenarios
# where executing the non-equi joims
# before the equi-join is more performant
out = (flights
 .conditional_join(
     flights,
     ('end', 'takeoff' ,'>='),
    ('start', 'takeoff', '<='),
     ('orig','orig','!='),
     ('dest', 'orig', '=='),
     df_columns = ['start', 'end', 'dest'],
     right_columns = ['takeoff', 'orig'],
     force=True,
     use_numba=False)
)
329 ms ± 1.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
# works great if the keys in the equi-join
# are heavily duplicated
# this numba version uses sorting+binary search
# instead of a hash join
out = (flights
 .conditional_join(
     flights,
     ('end', 'takeoff' ,'>='),
    ('start', 'takeoff', '<='),
     ('orig','orig','!='),
     ('dest', 'orig', '=='),
     df_columns = ['start', 'end', 'dest'],
     right_columns = ['takeoff', 'orig'],
     use_numba=True)
)
65.9 ms ± 2.65 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# duplicated keys
In [20]: flights.value_counts(subset='orig')
Out[20]:
orig
3      6257
0      5452
25     3535
44     3332
54     2847
       ...
178      15
148      15
195      15
153      13
51       11
Name: count, Length: 201, dtype: int64

In [21]: flights.value_counts(subset='dest')
Out[21]:
dest
2      6778
27     5426
31     3286
16     3189
37     2882
       ...
123      14
107      12
115      11
199      11
196       7
Name: count, Length: 201, dtype: int64
```


THANK
YOU