OUZO for indexing sets

Accelerating queries to sets with GIN, GiST, and custom indexing extensions

Markus Nullmeier

Zentrum für Astronomie der Universität Heidelberg
Astronomisches Rechen-Institut

`mnullmei@ari.uni.heidelberg.de`
`https://github.com/mnullmei`

Sets

- Come up as a model of various real-world data
- Not available as such in PostgreSQL, but
  - Use keys of JSONB / Hstore as elements:

```
SELECT '{"elem1": 1, "elem2": 2, "elem3": 1}'::json;
```

  - Use sorted arrays:

```
SELECT '{3,11,17,29}';
```

Some PostgreSQL set operations

```
create extension intarray;
```

- **Overlap** `SELECT '{5,17,23}'::int[] && '{3,11,17,29}'::int[];`
- **Subset** `SELECT '{17,23}'::int[] && '{3,23,29}'::int[];`
- **Union** `SELECT '{}'::int[] | '{1,3,5}'::int[];`
- **Intersection** `SELECT '{2}'::int[] & '{1,2,3}'::int[];`

Indexing sets

- Typical techniques
  - "inverted file" = inverted index
    - elements as keys, sets as indexed columns
    - Very good for single-element search
    - In PG: available for intarray, JSONB, hstore
  - RD-Trees
    - Useful for superset queries
    - Available for intarray via GiST

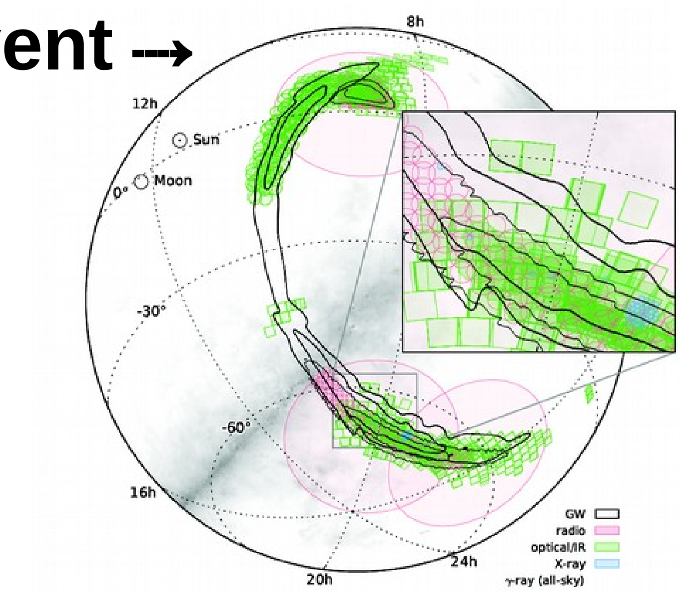
Evaluation

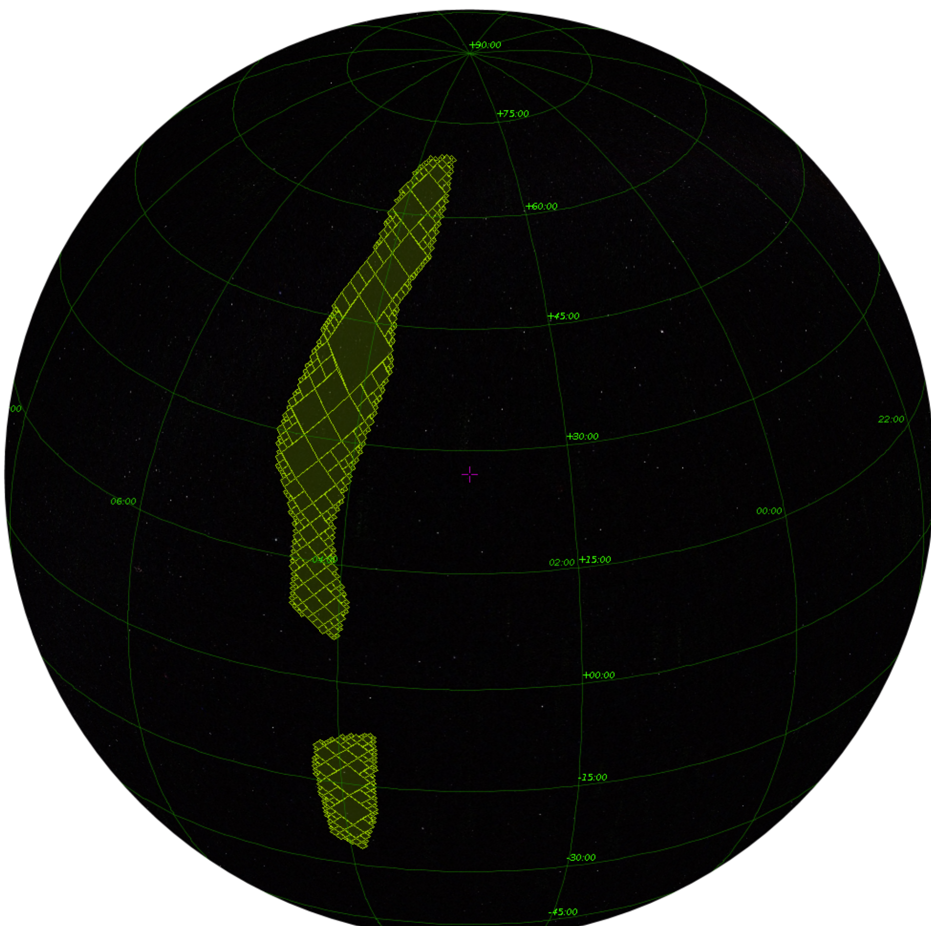
- PG's built-in / contrib features are sufficient for most uses
  - Small to medium-sized sets
  - Index support is there
- Limitations
  - Any set operation must load the whole set from disk / buffers
    - not necessarily so: `PG_DETOAST_DATUM_SLICE`
  - May be inefficient for domain-specific set types

A use case from astronomy

- Sky coverage of astronomical surveys

gravitational wave event ⇢



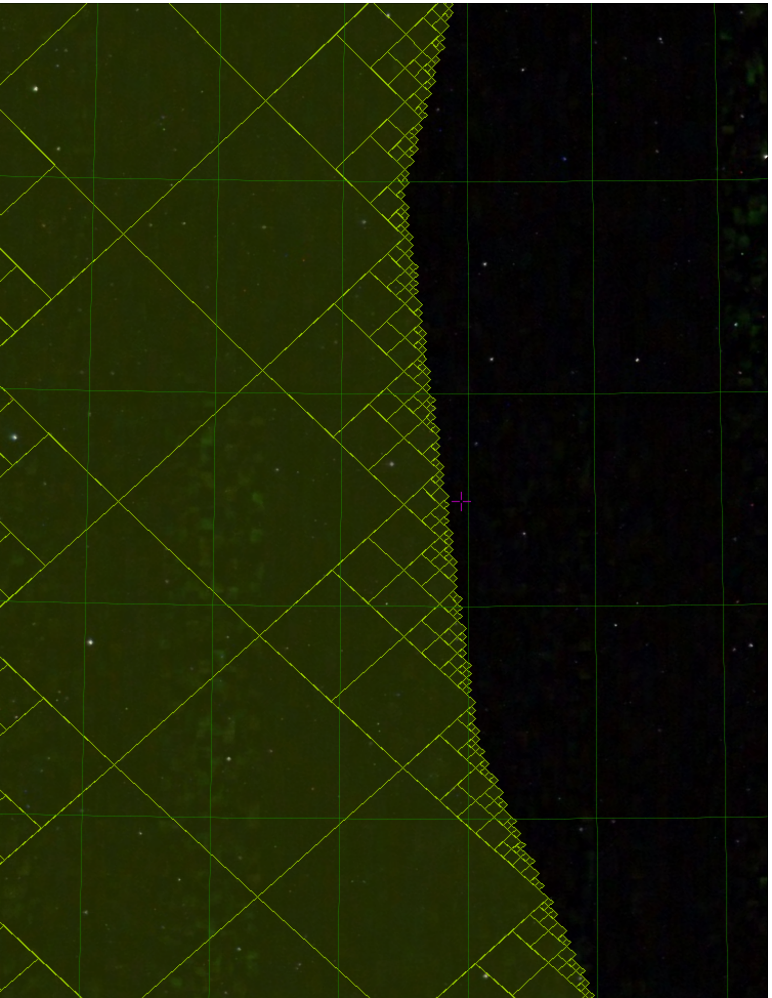


← Multi-Order Coverage

= set of sphere elements
of different orders
→ Set of integer intervals

A use case from astronomy

- Sky coverage



1 diamond element
= 1 integer interval

1 set
= 1 list of intervals

```
{[2, 6) [17, 30) [33, 40)
 [123, 124) [332, 438), ...}
```

Use case: details (I)

- Sky coverage sets may very detailed, i. e., large
- Fast response times for public data required
- Domain-specific standard (IVOA MOC, Healpix-based)
  - “multi-order coverage”
- Many astronomical on-line databases use PostgreSQL

Use case: details (II)

- **Run-length compression for spatial locality**
  - Any **large sky element**, consisting of a large number of elements at the finest resolution ***is encoded as an interval of 2 integers***

Custom data type

`{[2, 6) [17, 30) [33, 40) [123, 124) [332, 438)}`

- Set of intervals of integers
  - = boundaries at finest level of resolution
  - Non-overlapping
  - Stored in sorted order
- Typical operations
  - Subset for single numbers (points) or sets
  - Set overlap

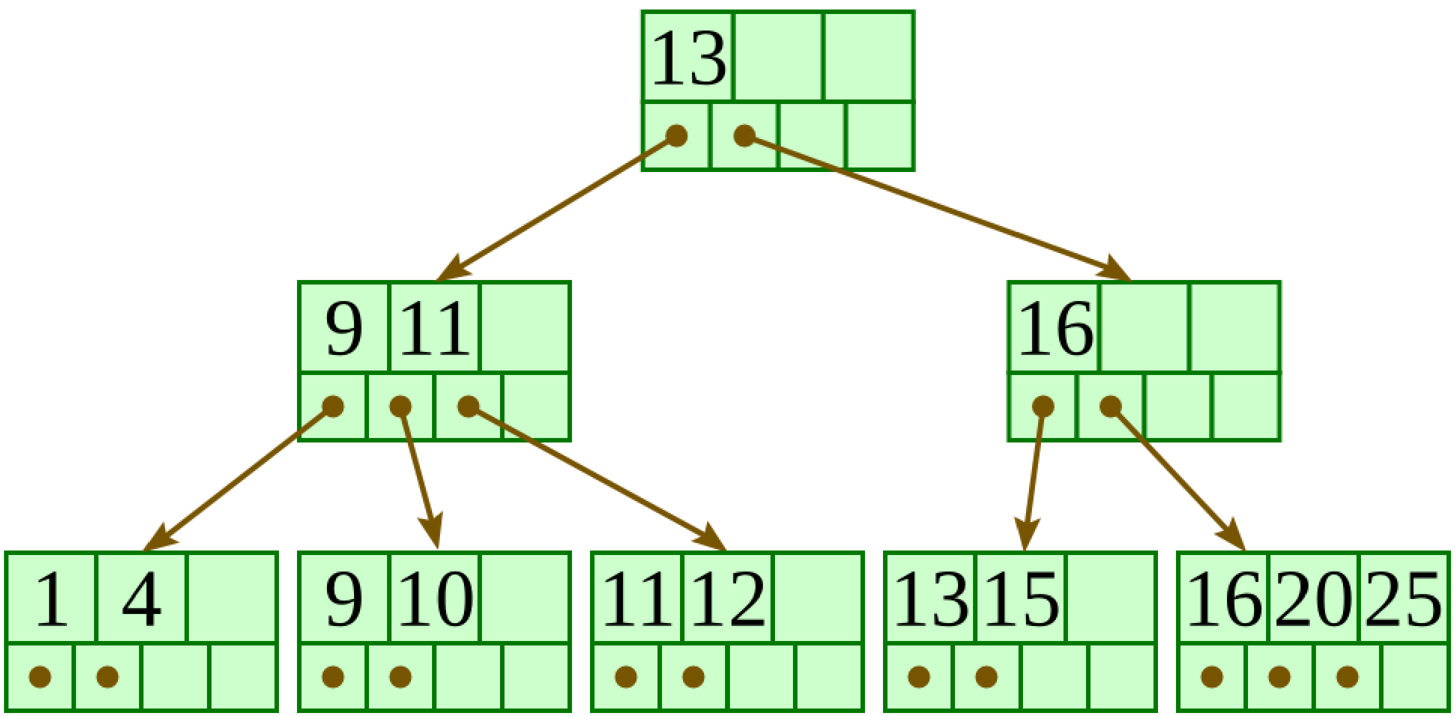
Make sequential scan fast

- Loading a whole sky map just for one point is inefficient
- Use sliced access of on-disk “TOAST” data
- Serialise each sky map B-tree-like
  - read-only
  - Page size = TOAST fragment size, 1996 bytes
- Write once means:
  - No space wasted, tree is nicely balanced
  - No penalty for full sequential access

Make sequential scan fast

- Searching one point on sufficiently fast machine in 17K objects: 75ms
- On-Disk serialisation of a single interval set as B-tree

`{[2, 6) [17, 30) [33, 40) [123, 124) [332, 438)}`



Still not fast enough?

- **Ordinary, element-wise inverted indexes impossible**
- **...but using intervals as keys would do the trick**

sorted intervals	sets of pointers to sky maps
[17, 30)	{ obj7, obj11 }
[843, 2577)	{ obj2, **obj108**, obj109 }
[5756, 9433)	{ **obj108**, obj732, obj11030 }
...	...

Sky map indexing

- Intervals-as-keys
  - must not overlap, else inefficient
  - $\Rightarrow$ indexing with GIN impossible
- RUM to the rescue!
  - GIN descendant with various improvements
  - usable as installable index extension
  - PostgreSQL license
    `https://github.com/postgrespro/rum`
  - must be somewhat modified...

Project “OUZO”

- “Often Useful Zermelo* Ordering”
- Index access method for set of intervals
- Generic for any kind of interval key type
  - and associated set type

*Ernst Friedrich Ferdinand Zermelo (1871-1953),
founder of axiomatic set theory

Project “OUZO”

- Relatively high-level extension of RUM
  - Complete reuse of concurrent B-tree code
    - for entry tree as well as for posting trees
  - Will be backward compatible

OUZO: key changes to RUM

- Insertion to the index must split the intervals-as-keys
  - of the inserted set (sky map)
  - and all overlapping keys already in the index
- B-tree insertion requires ‘lower bound’ search
- Additional support functions for the operator class

Insertion interval split

- **To insert: interval [96,128) of obj108**
- **Index before:**

[32, 128)	{ obj7, obj11 }

- **Index after insertion:**

[32, 96)	{ obj7, obj11 }
[96, 128)	{ obj7, obj11, **obj108** }

- **One of 13 possible cases**

‘lower bound’ search for RUM

- Return exact match of start of interval or next higher
  - RUM mostly only uses exact match so far
  - Existing implementation ‘almost’ gives lower bounds for searches
- Allows much code reuse
  - RUM features C-style object orientation for its B-trees
  - Re-implement ‘find in leaf page’ method

New ‘SQL’ support functions

- Specified in ‘`create operator class`’ DDL instruction
  - makes indexes usable for specific data types
- `internal get_left_boundary(interval)`
- `internal get_right_boundary(interval)`
- `int compare_boundaries(internal, internal)`
- `interval make_interval(internal, internal)`
  - ‘`internal`’: basically opaque pointer to boundary

Concurrency of index insertion

- At most 3 intervals must be changed at the same time
  - other backends modifying entry tree do not wait too long
- 'Long' intervals are inserted on step at a time
  - Must release locks after each insertion elementary step
  - *Should* give decent concurrency

Thank you for listening!

Questions?

Markus Nullmeier

Zentrum für Astronomie der Universität Heidelberg
Astronomisches Rechen-Institut

`mnullmei@ari.uni.heidelberg.de`
`https://github.com/mnullmei`