Accelerating queries of set data types with GIN, GiST, and custom indexing extensions

Markus Nullmeier

Zentrum für Astronomie der Universität Heidelberg
Astronomisches Rechen-Institut

`mnullmei@ari.uni.heidelberg.de`

Sets

- Come up as a model of various real-world data
- Not available as such in PostgreSQL, but
  - Use the keys of JSONB or hstore as set 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 for fast queries

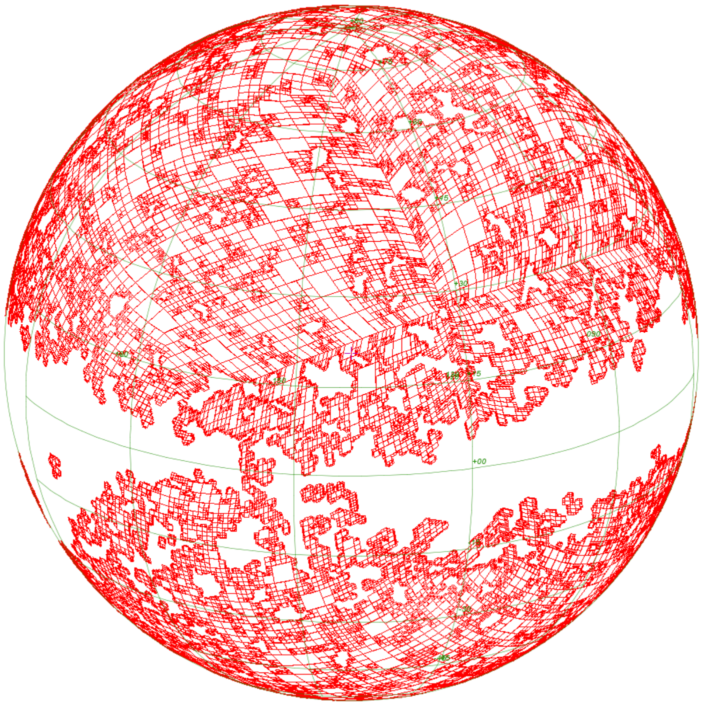
- Typical techniques
  - “inverted file” = inverted index (see RUM talk), with:
    - elements as keys, sets as indexed columns
    - Very good for accelerating single-element overlap
    - Available for intarray, JSONB, hstore
  - RD-Trees
    - Useful for superset queries
    - Available for intarray via GiST

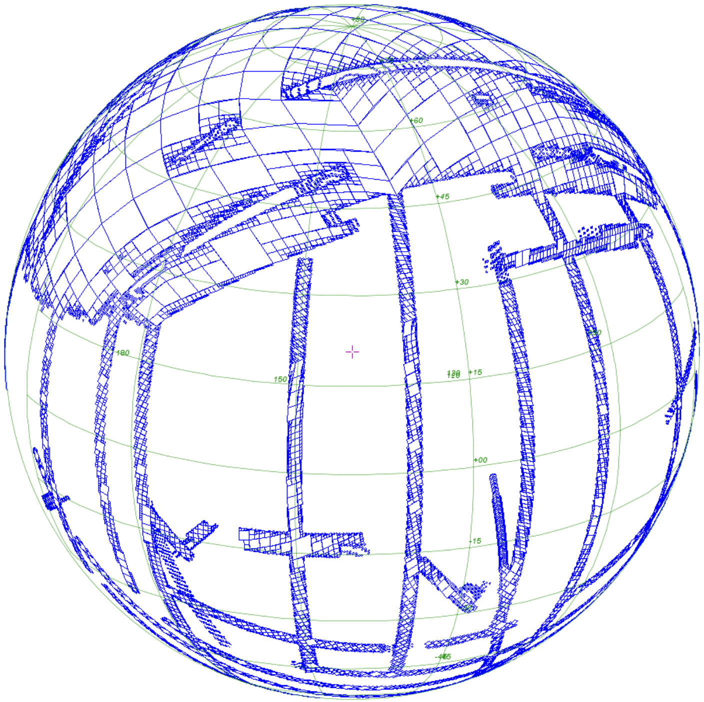
Evaluation

- Built-in or ‘contrib’ features sufficient for most uses
  - Small to medium-sized sets
  - Index support is there
- Limitations
  - All set operations must load the whole set from disk
    - AFAIK, being worked on for JSONB
  - May be inefficient for domain-specific set types

A use case from astronomy

- **Sky coverage of astronomical surveys**



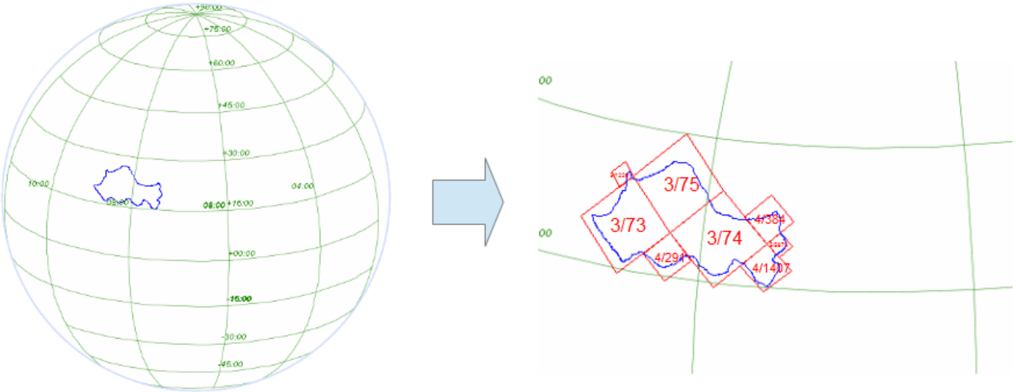


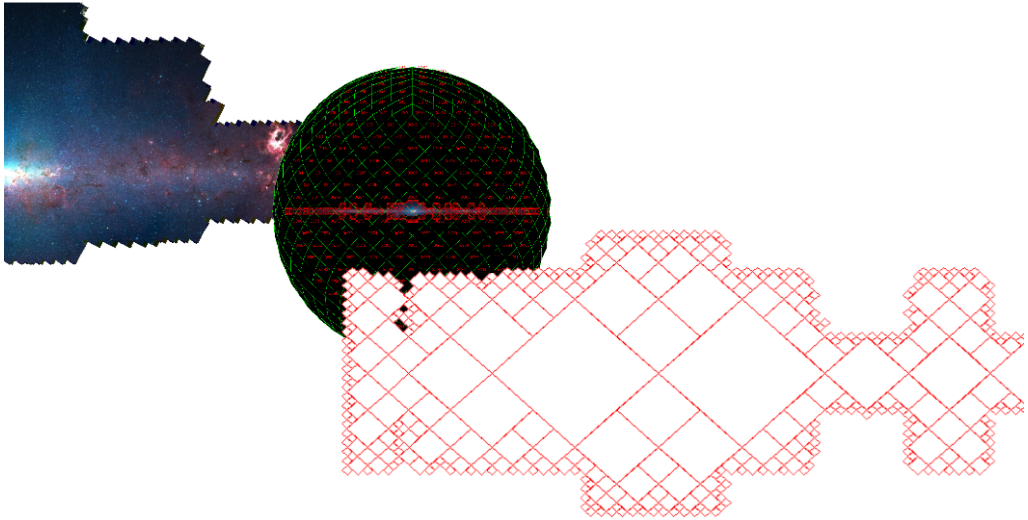
Use case: details (I)

- 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**
  - **Nearby sky elements encoded as interval of 2 numbers**





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
- Write once means:
  - No space wasted, tree is nicely balanced
  - No penalty for full sequential access

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
  - $\Rightarrow$ implementation with GIN impossible
- RUM to the rescue! 
  - usable as installable index extension
  - PostgreSQL license
  - must be somewhat modified...

Project “OUZO”

- As of yet undisclosed inverted acronym
- 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
- Generic for any kind of interval key type
  - and entry type
- Nearing completion

OUZO: key changes to RUM

- Insertion to the index must split the intervals-as-keys
  - of the inserted sky map
  - and all preexisting keys
- B-tree scan requires ‘lower bound’ search
  - For insertion and for queries
- Additional support functions for the operator class

Insertion interval split example

- 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

Concurrency of index insertion

- At most 3 intervals must be changed at the same time
  - locking stops other backends to modifying entry tree
- ‘Long’ intervals are inserted on step at a time
  - Release locks after each elementary step
  - Should give decent concurrency
- Not tested yet

‘lower bound’ search for RUM

- Return exact match of start of interval or next higher
  - RUM only uses exact match so far
  - Existing implementation ‘almost’ gives lower bounds
- Allows much code reuse
  - RUM features C-style object orientation for its B-trees
  - Re-implement 2 methods: ‘find in tree / leaf page’

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 pointers to boundaries

Thank you for listening!

Questions?

Markus Nullmeier

Zentrum für Astronomie der Universität Heidelberg
Astronomisches Rechen-Institut

`mnullmei@ari.uni.heidelberg.de`