

Managing Intervals Efficiently in Object-Relational Databases

The Relational Interval Tree (patent pending)

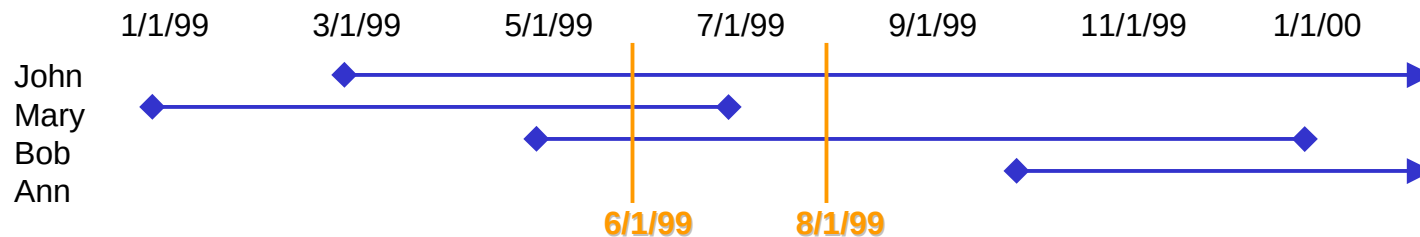
Hans-Peter Kriegel, Marco Pötke, Thomas Seidl

University of Munich, Germany

VLDB 2000, Cairo

Interval Data

- Who worked for us between 6/1/99 and 8/1/99?



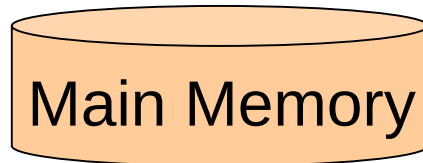
- Extended objects occur in many applications:
 - Temporal databases (valid time, transaction time)
 - Constraint databases (interval constraints)
 - Scientific databases (fuzzy measurements)
 - Spatial applications (space-filling curves)

Content

1. Related Work
2. The Relational Interval Tree
3. Query Processing
4. Experimental Evaluation

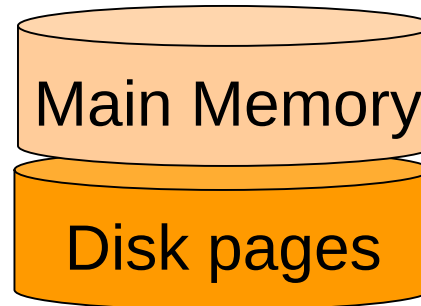
Structures to Store Intervals

Main-Memory
'No DBMS'



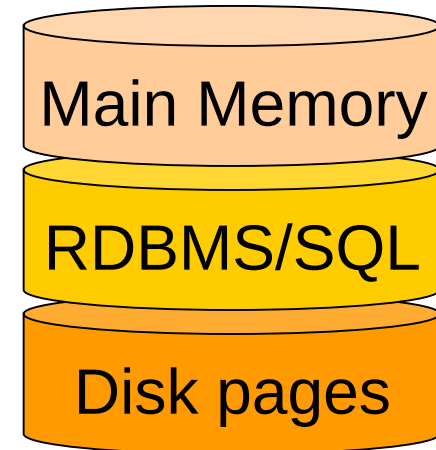
– No block structure

Secondary Storage
'Extend the DBMS'



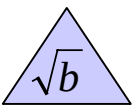
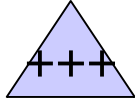
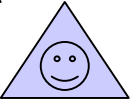
+ Persistent storage
+ Efficient solutions
– Complex to implement
(ACID, ...)
– Intrusive modifications

Relational Storage
'Use RDBMS as it is'

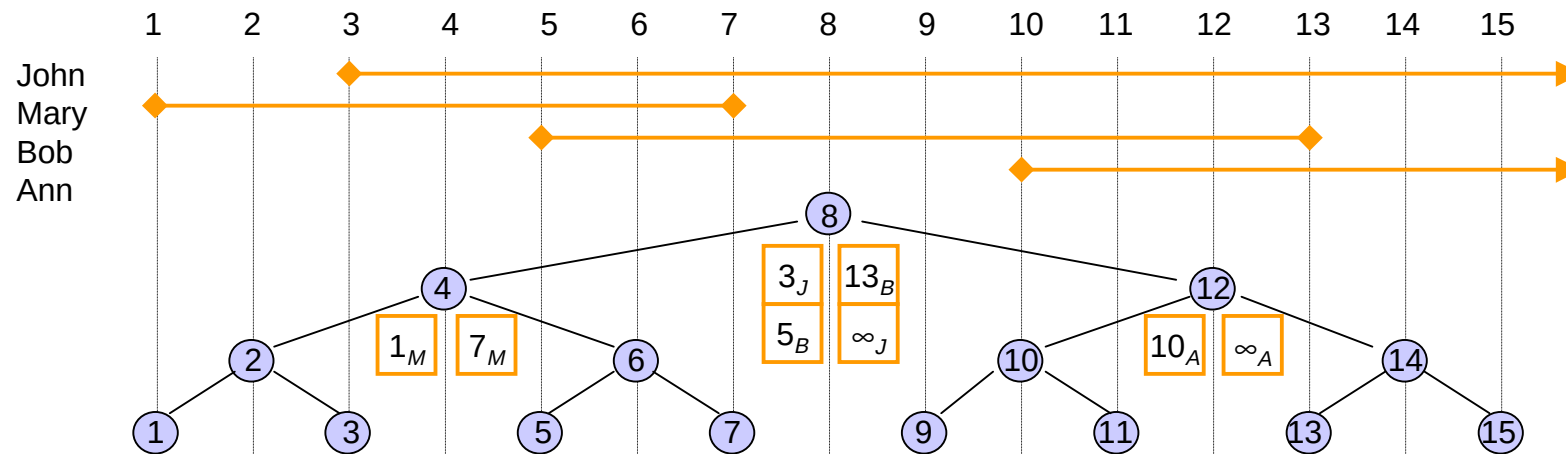


+ Industrial strength
(Robust, ACID, ...)
+ Easy to implement

Motivation

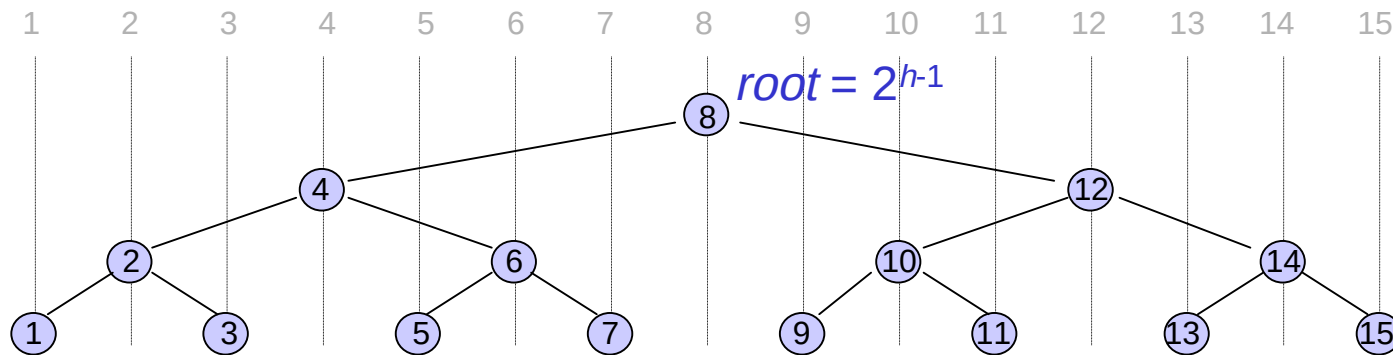
- Making Edelsbrunner's interval tree persistent
 - Ext.Mem.Int.Tree [AV'96]: **Pagination by increased fan-out** 
 - Interval B-tree [AT'95]: **Augmentation of B+-trees** 
 - RI-Tree [new]: **Use built-in relational indexes as-they-are** 
- Design goals for the Relational Interval Tree
 - Relational integration: **Supported by any off-the-shelf RDBMS**
 - Performance: **Fully dynamic and efficient data structure**
 - Basic variant: **Support of integer domains**

Main-Memory Interval Tree



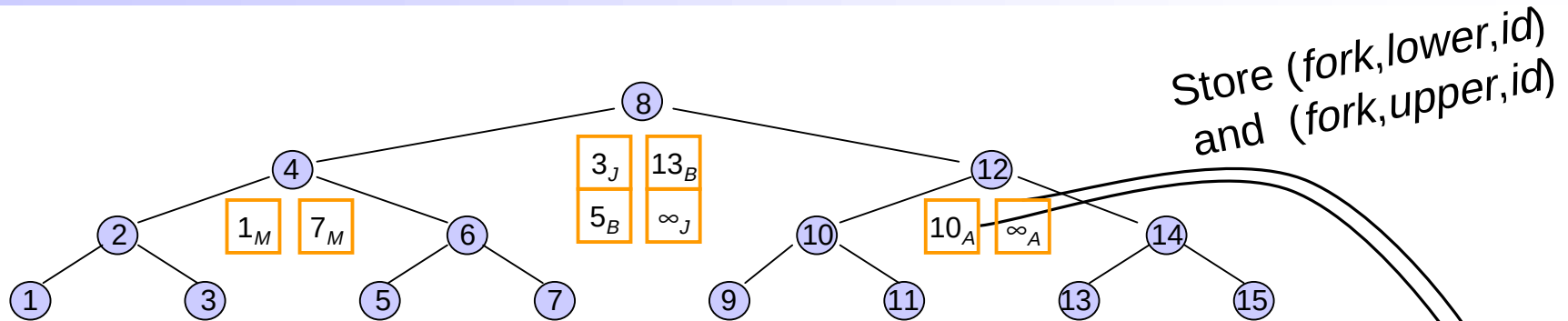
- Primary structure (backbone): Binary tree with labeled nodes
- Secondary structure: Local lists of interval bounds
- No redundancy: Each interval is registered at its fork node only
- Query Processing: Traverse tree to report resulting intervals

Virtual Primary Structure



- Fixed backbone: Entire integer domain $[1 \dots 2^h - 1]$
- Virtualization: Backbone is not materialized
- Purely arithmetic traversal: No I/Os (cf. Linear Quadtree)
- $O(1)$ storage space: Parameter $root = 2^{h-1}$

Stored Secondary Structure



- Two relational indexes collect the local lists of interval bounds:

lowerIndex (fork, lower, id):

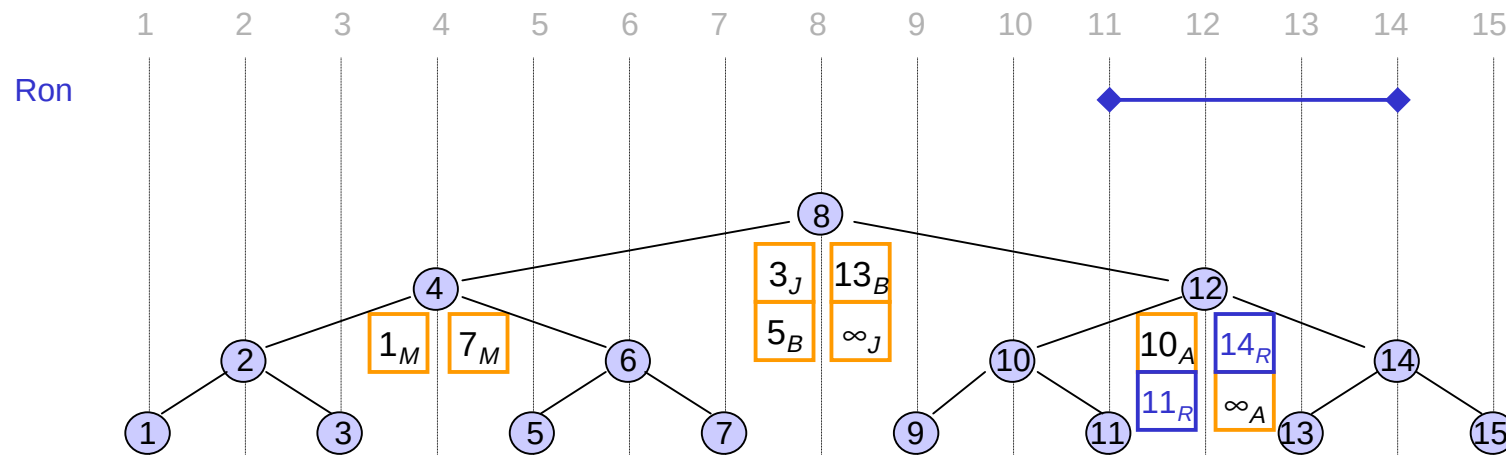
4,1,Mary	8,3,John	8,5,Bob	12,10,Ann
----------	----------	---------	-----------

upperIndex (fork, upper, id):

4,7,Mary	8,13,Bob	8,∞,John	12,∞,Ann
----------	----------	----------	----------

- Supported by any RDBMS: No modification of built-in B+-trees
- Storage space for n intervals: $O(n/b)$ disk blocks of size b

Performing Updates



- Compute fork node: Arithmetic traversal, no I/O required
- Update relational indexes:

lowerIndex:

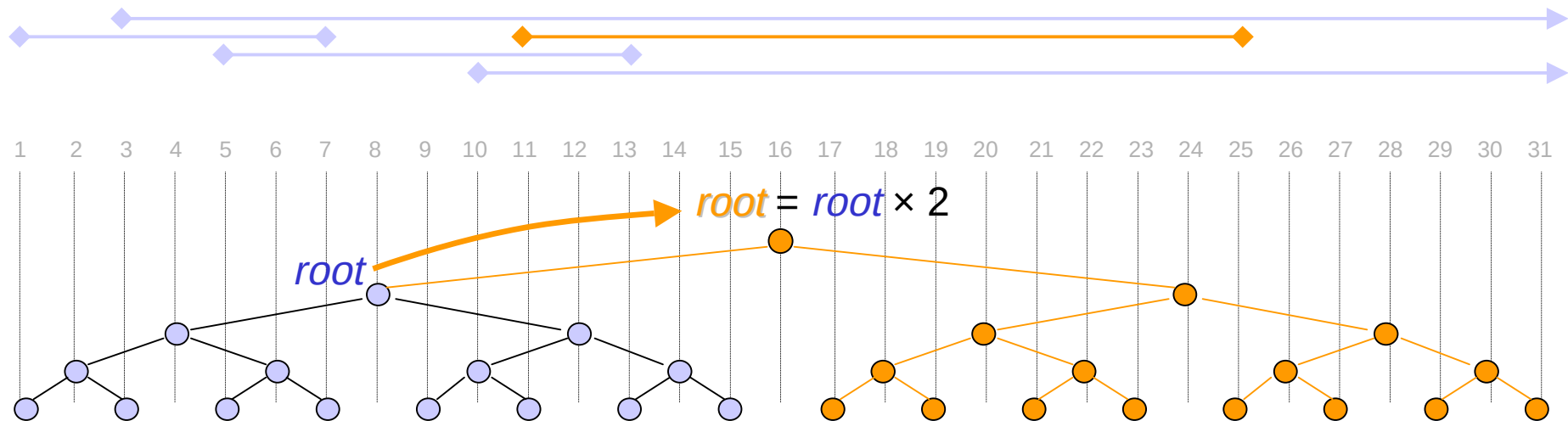
4,1,Mary	8,3,John	8,5,Bob	12,11,Ron	12,10,Ann
----------	----------	---------	-----------	-----------

upperIndex:

4,7,Mary	8,13,Bob	8,infinity,John	12,14,Ron	12,infinity,Ann
----------	----------	-----------------	-----------	-----------------

- I/O complexity: $O(\log_b n)$ I/Os for updating the B+-trees

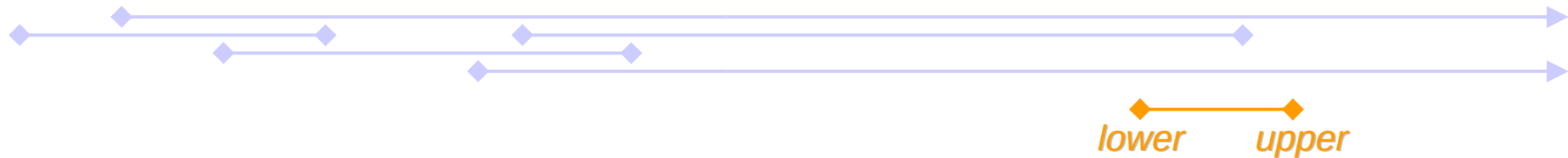
Expansion of the Data Space



- No index reorganization required
- The tree height h is kept minimal:

$$h = O \left(\log_2 \left(\frac{\max\{upper\} - \min\{lower\}}{\min\{upper - lower + 1\}} \right) \right)$$

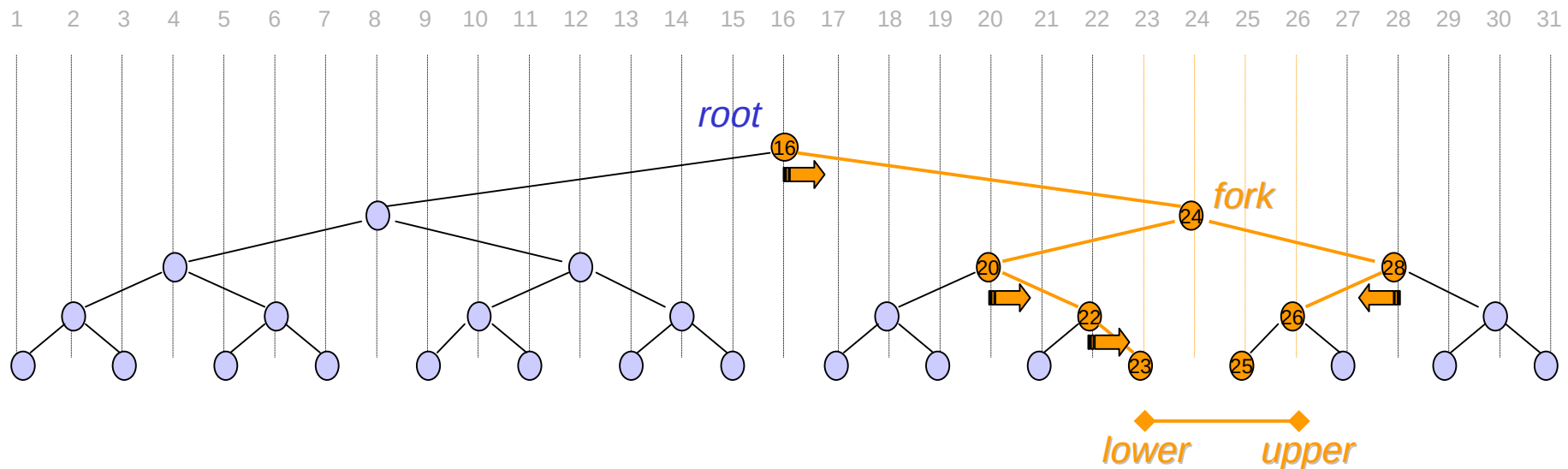
Query Processing



Two steps to process an interval query

1. Transform interval query into a set of range queries
 - The generated queries are collected in transient tables (no I/Os)
2. Perform a single SQL query
 - Join the transient query tables with the relational indexes

Generating Range Queries



- Prepare two transient tables to collect the query nodes
- Arithmetically descend from *root* to *lower* and to *upper*:
 - At nodes left of *lower*: report entries i with $i.upper \geq lower$ (16,20,22)
 - At nodes right of *upper*: report entries i with $i.lower \leq upper$ (28)
 - For nodes between *lower* and *upper*: report all entries (23 - 26)

Single SQL Query

- Join query tables with the indexes:

```
SELECT id FROM upperIndex u, leftQueries left  
  WHERE u.node = left.node AND u.upper >= lower
```

UNION ALL

```
SELECT id FROM lowerIndex l, rightQueries right  
  WHERE l.node = right.node AND l.lower <= upper
```

UNION ALL

```
SELECT id FROM lowerIndex /*or upperIndex*/ i  
  WHERE i.node BETWEEN lower AND upper
```

- No duplicates are produced ---> **UNION ALL**
- Blocked output of index range scans is guaranteed

Analysis

- I/O complexity of query processing

Assume block size b and tree height h :

- No I/Os for arithmetic traversal and transient query mgmt.
- At most $2 \times h$ B+-tree lookups
- $O(\log_b n)$ I/Os for a single B+-tree lookup
- $O(r/b)$ I/Os to report r results (blocked output)
- Overall I/O Complexity:

$$O(h \log_b n + r/b)$$

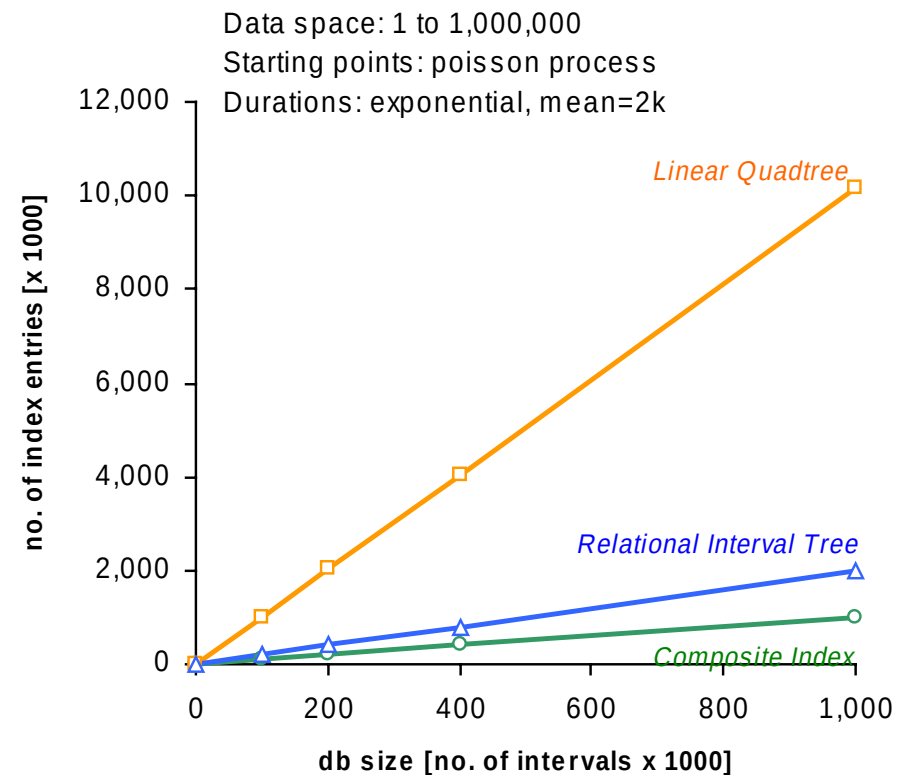
- Experimental evaluation on a commercial RDBMS

Experimental Evaluation

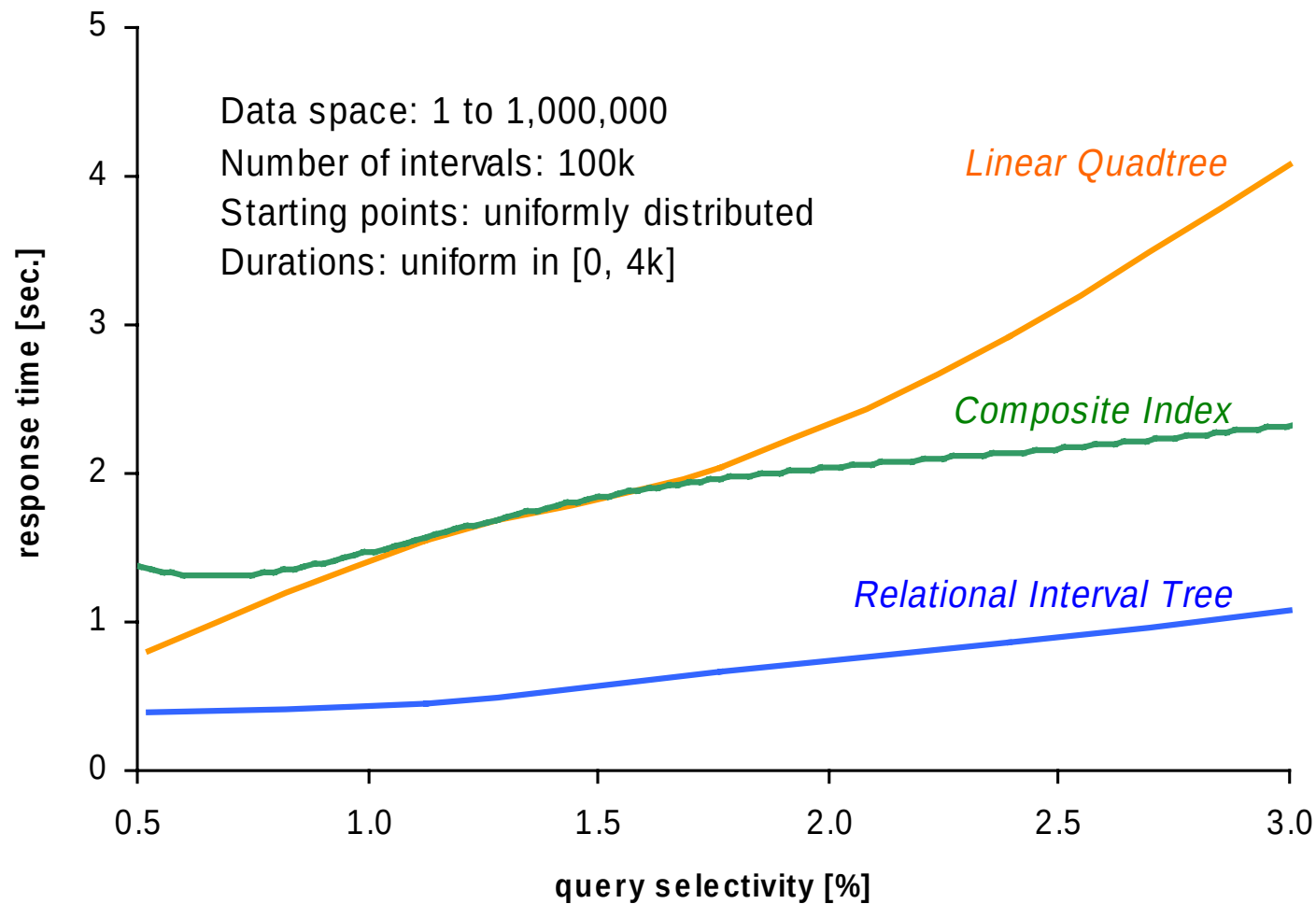
Competitors (Relational Storage):

- **Linear Quadtree**
 - Hybrid tiling (Oracle, IBM, ESRI)
 - Here: optimized fixed tiling
- **Composite Index**
 - MAP21 (Nascimento, Dunham '99)
 - Interval-spatial-transformation (Goh, Lu, Ooi, Tan '96)
 - Here: index on (upper, lower)
- **Relational Interval Tree** (new)

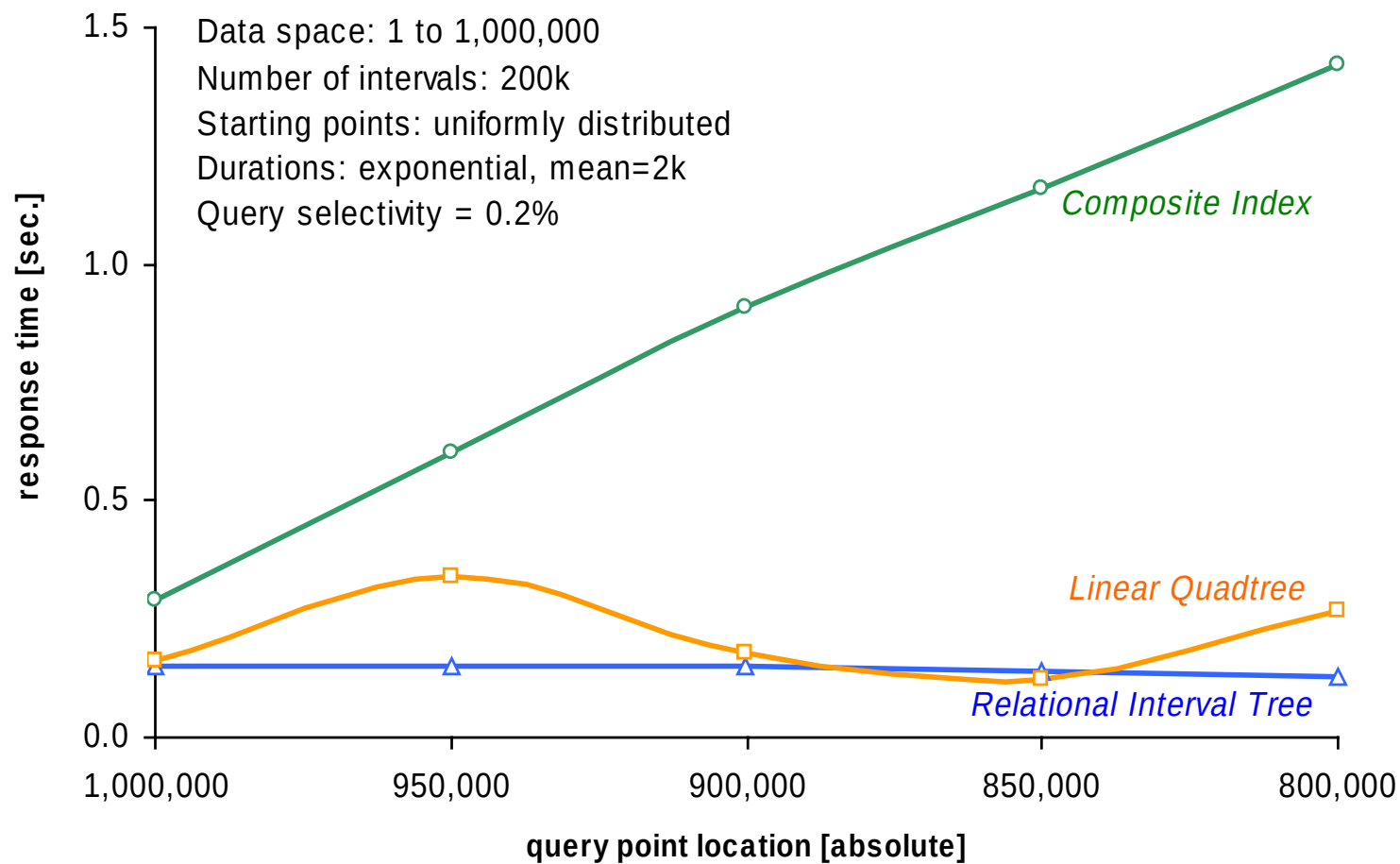
Space Requirement



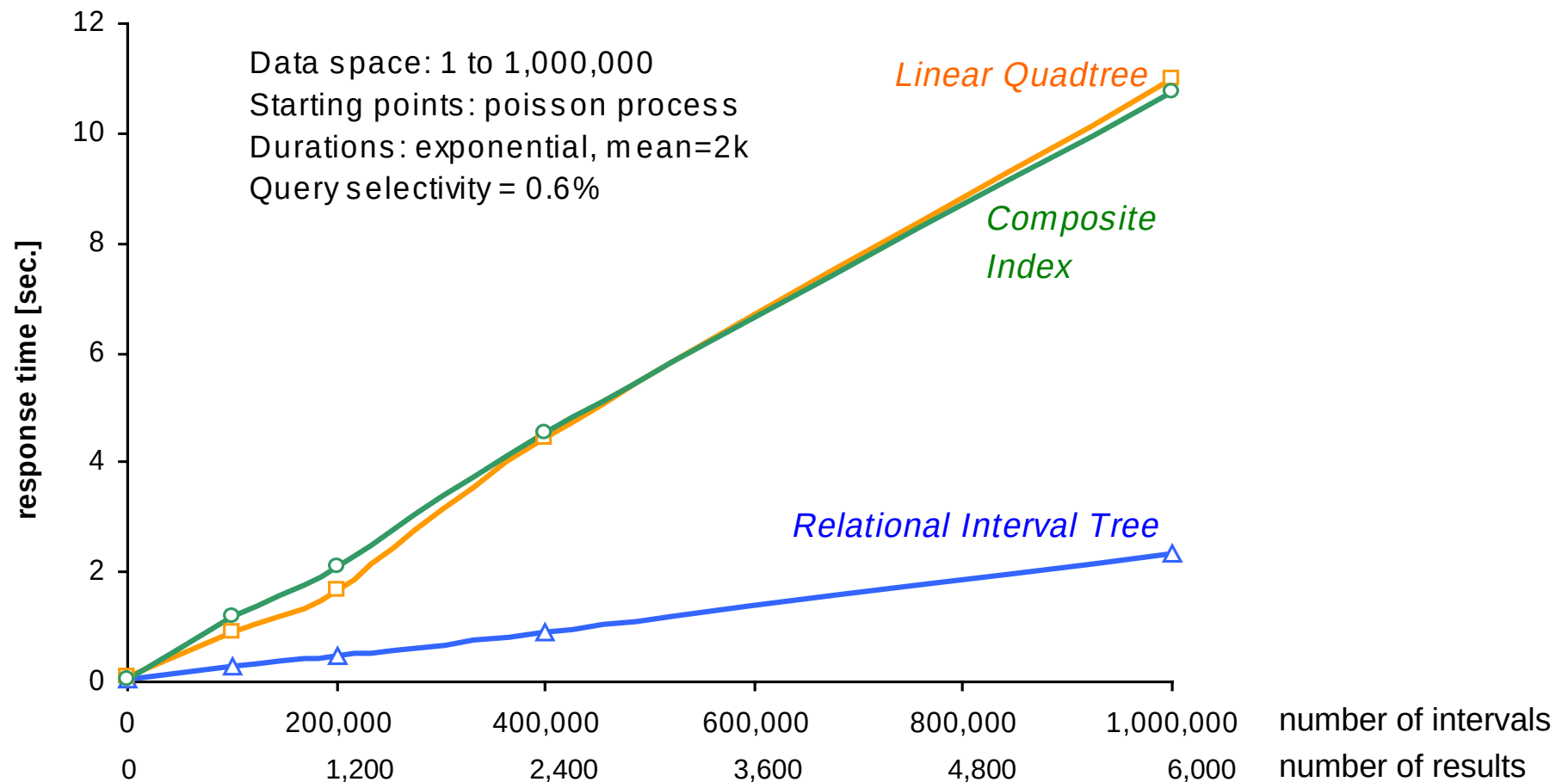
Varying Query Selectivity



Varying Query Location



Scale-Up for Large Databases



Conclusions

- Relational Integration
 - SQL interface, no intrusive modification
 - Ready for object-relational wrapping
- Performance
 - Optimal space/update/query complexity
 - Very good experimental performance
- Extensibility
 - Dynamic expansion of the data space
 - 2D and 3D spatial data (interval sequences)

Related Work

Main-Memory 'No RDBMS'

- Interval Tree (Edelsbrunner '80)
- Priority Search Tree (McCreight '85)
- Segment Tree (Bentley)
- Interval Skip List (Hanson '90)

Secondary Storage 'Extend RDBMS'

- Time Index (Elmasri, Wu, Kim '90)
- Interval B-tree (Ang, Tan '95)
- Interval B+-tree (Bozkaya, Özsoyoglu '98)
- TP-Index (Shen, Ooi, Lu '94)
- Temporal Grid File (Lee, Tseng '98)
- Ext. Mem. Interval Tree (Arge, Vitter '96)
- External Segment Tree (Blankenagel, Güting '94)
- Path Caching (Ramaswamy, Subramanian '94)
- Segment R-tree (Kolovson, Stonebraker '91)

Relational Storage 'Use RDBMS as-is'

- Linear Quadtree (Samet '90)
- D-Order (Goh, Lu, Ooi, Tan '96)
- MAP21 (Nascimento, Dunham '99)
- Window List (Ramaswamy '97)