A Two-layer Partitioning for Non-point Spatial Data

Dimitrios Tsitsigkos[1,3] Konstantinos Lampropoulos[3] Panagiotis Bouros[2]
Nikos Mamoulis[3] Manolis Terrovitis[1]

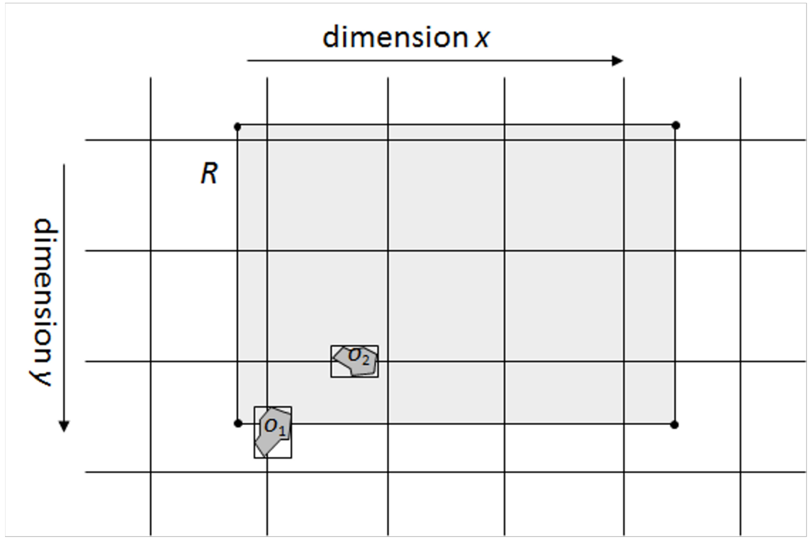
[1]Athena RC, Greece

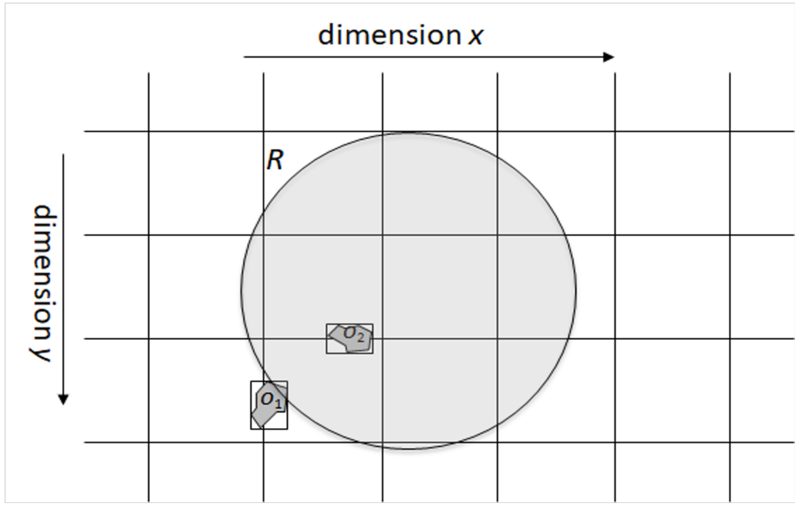
[2]Johannes Gutenberg University Mainz, Germany

[3]University Of Ioannina, Greece

Range Query

- Range query is a fundamental operation in managing spatial data:
  - Geographic Information Systems (e.g., management of huge meshes)
  - Neuroscience (e.g., building and indexing a spatial model of brain)
  - Location-based analytics (e.g., managing spatial influence region of mobile users in order to facilitate effective POI recomm)
- Retrieve all spatial objects which intersect with the area of R

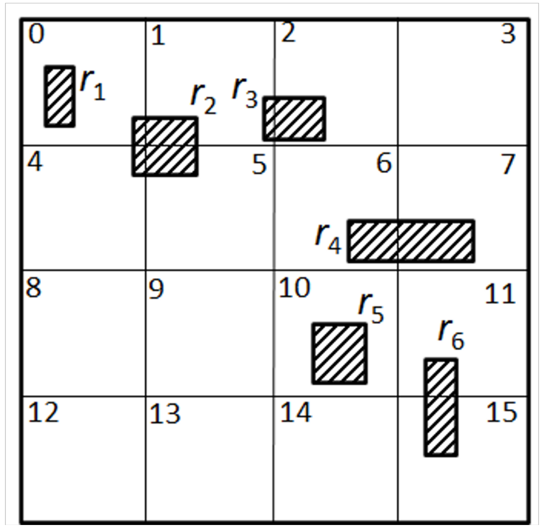




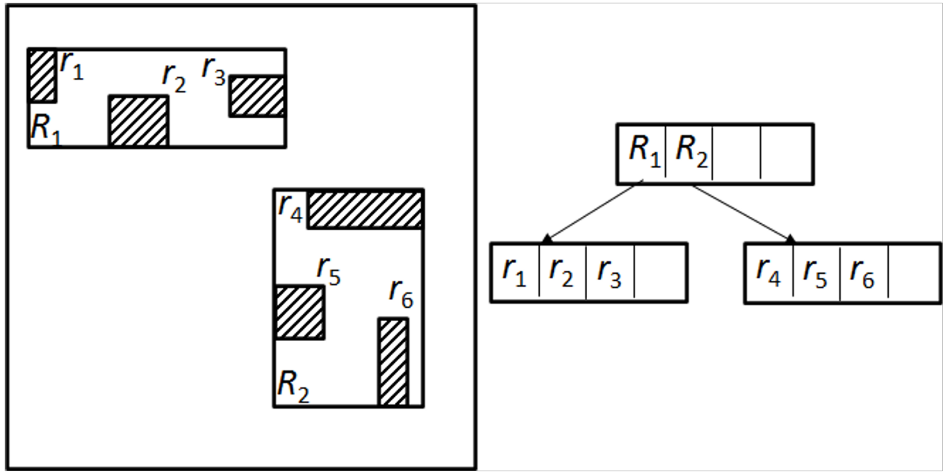
Motivation (1/2)

- Focus on non-point data

- Space-Oriented Partitioning:
  - Divide the space into spatially disjoint partitions
  - Data replication
  - Grid, quad-tree, etc.

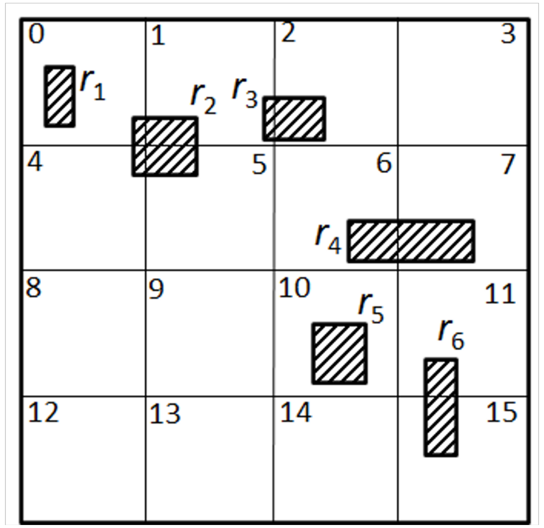


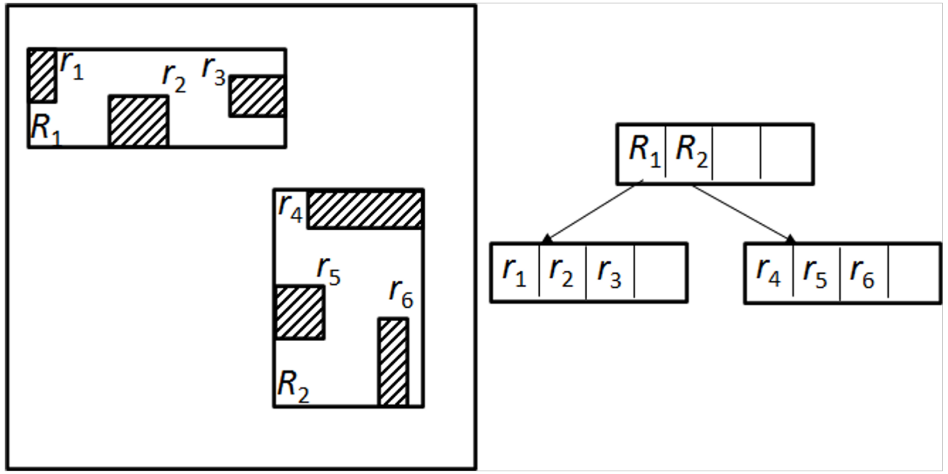
- Data-Oriented Partitioning:
  - No data replication
  - Balanced structure
  - R-tree family, etc.



Motivation (1/2)

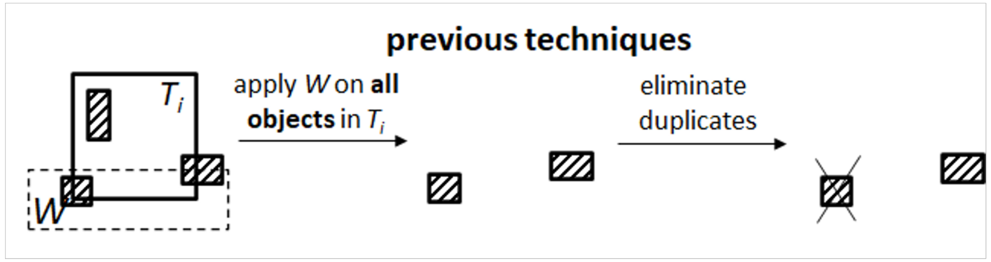
- Focus on non-point data
- Space-Oriented Partitioning:
  - Divide the space into spatially disjoint partitions
  - Data replication
  - Grid, quad-tree, etc.
- Data-Oriented Partitioning:
  - No data replication
  - Balanced structure
  - R-tree family, etc.

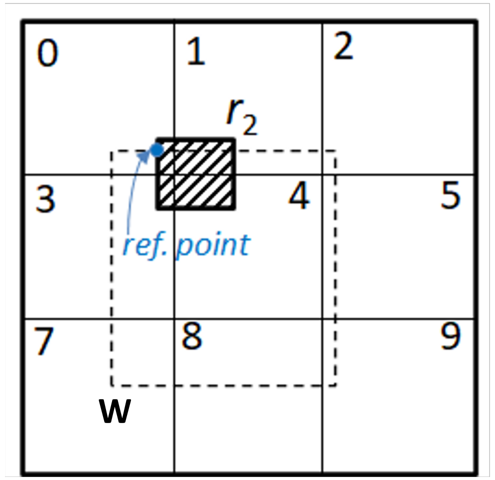


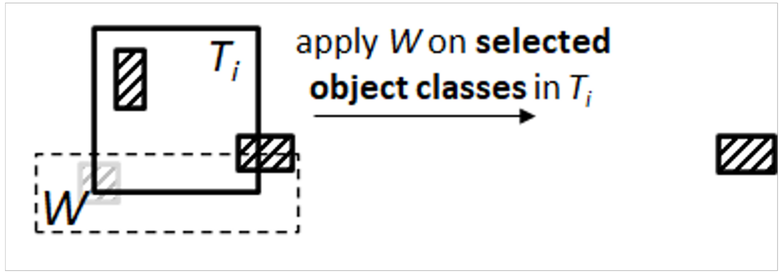


Motivation (2/2)

- Core challenge
  - Duplicated results
- Previous techniques
  - Find all results then duplicate elimination
  - Hashing, sorting
  - Reference point (Dittrich & Seeger, ICDE 2000)
- Our technique
  - Duplicate avoidance
  - Divide each tile into four classes that cannot generate duplicates

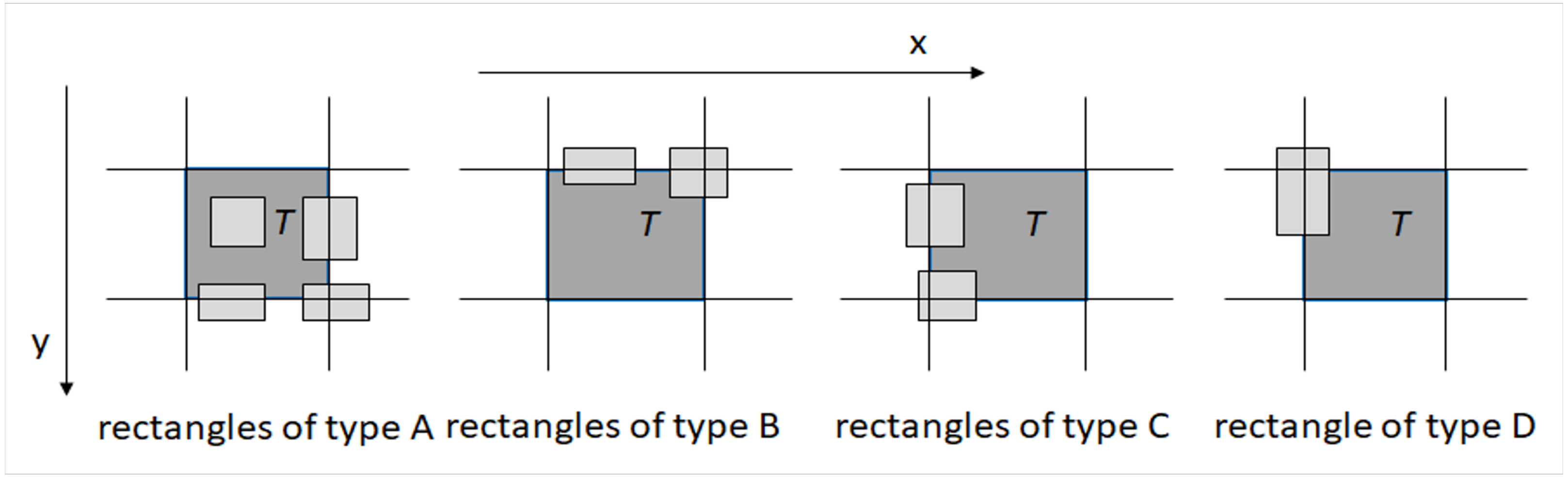




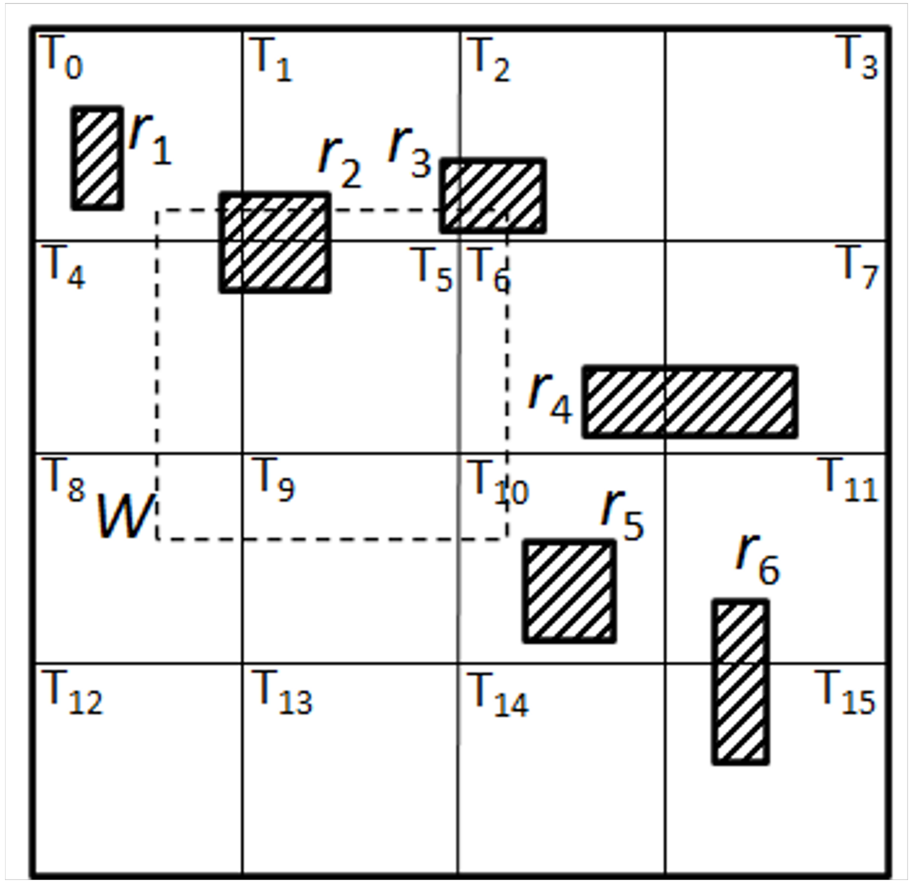


Two-layer Partitioning

- First layer
  - Standard Space-Oriented Partitioning
  - Divide the space into disjoint spatial partitions, called tiles
- Second layer
  - Additional in our approach
  - Divide each tile into four classes A, B, C, and D



Example



Tile	primary partitioning (standard)	secondary partitioning (our approach)
T_0	{r1,r2}	A= {r1, r2}
T_1	{r2,r3}	A = {r3}, C= {r2}
T_2	{r3}	C = {r3}
T_4	{r2}	B = {r2}
T_5	{r2}	D = {r2}
T_6	{r4}	A = {r4}
T_7	{r4}	C = {r4}
T_{10}	{r5}	A = {r5}
T_{11}	{r6}	A = {r6}
T_{15}	{r6}	B = {r6}

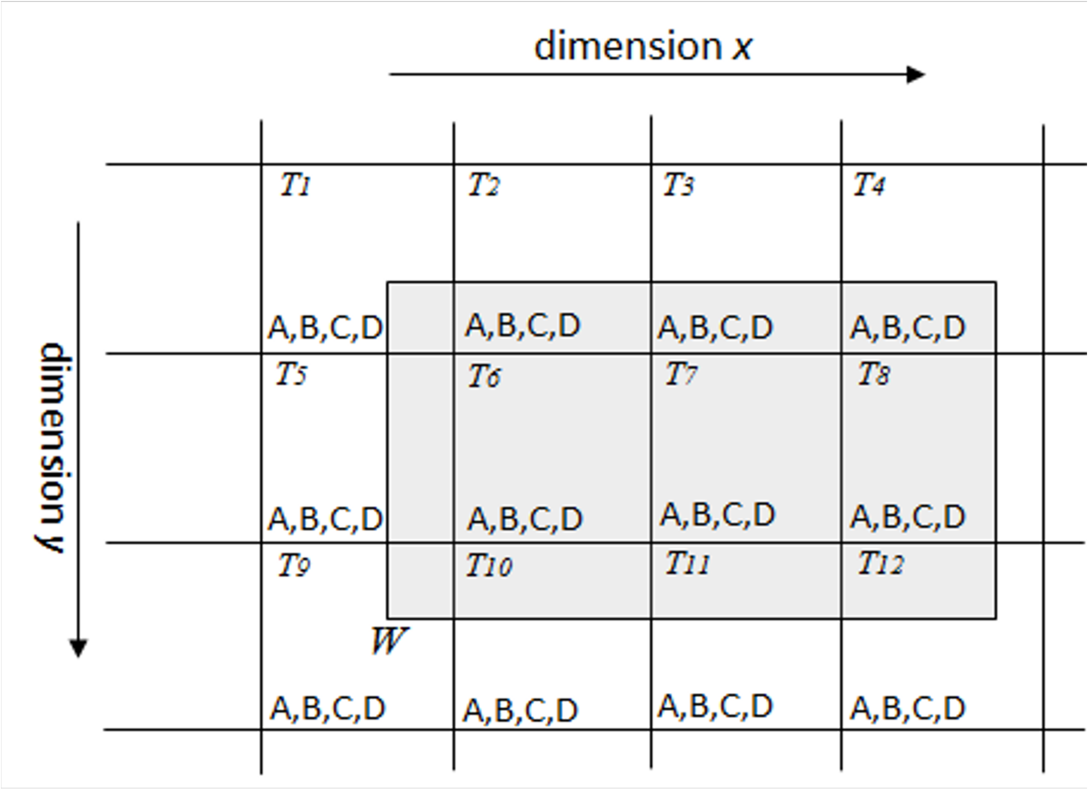
Query processing

- Identify the tiles relevant to the query
- For each relevant tile
  - Select relevant classes
  - For each relevant class
    - Identify intersecting rectangles

Query processing

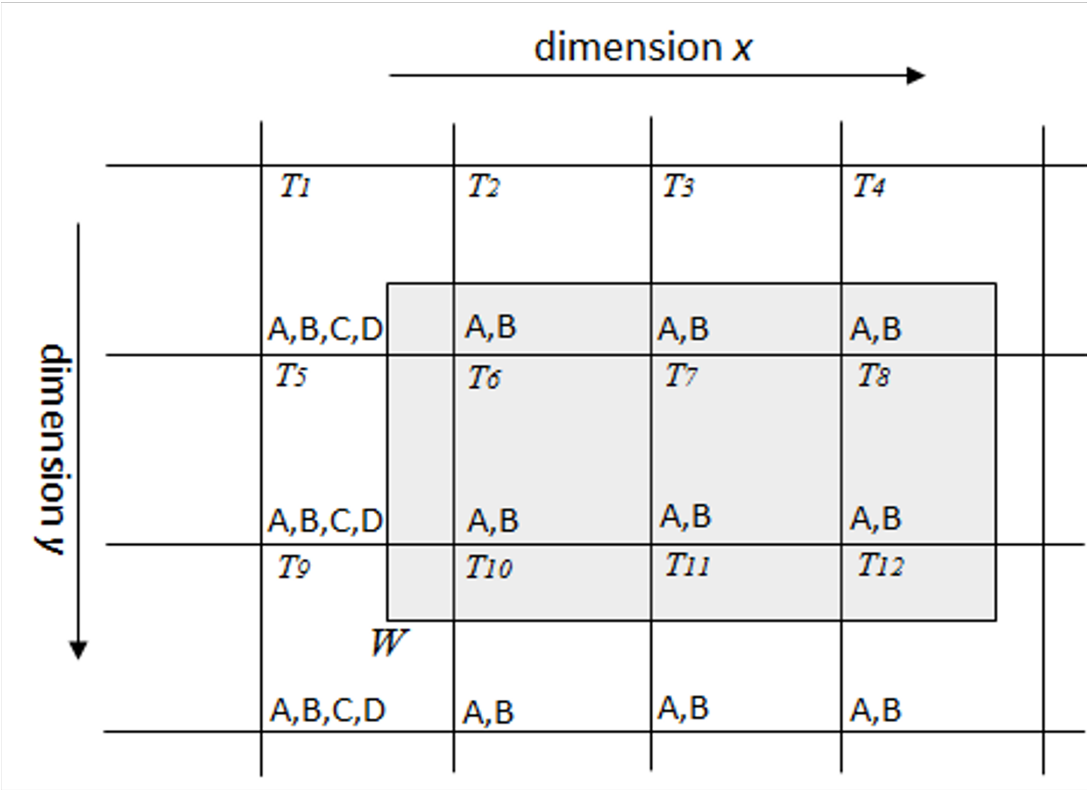
- Identify the tiles relevant to the query
- For each relevant tile
  - Select relevant classes
  - For each relevant class
    - Identify intersecting rectangles

Select Relevant Classes



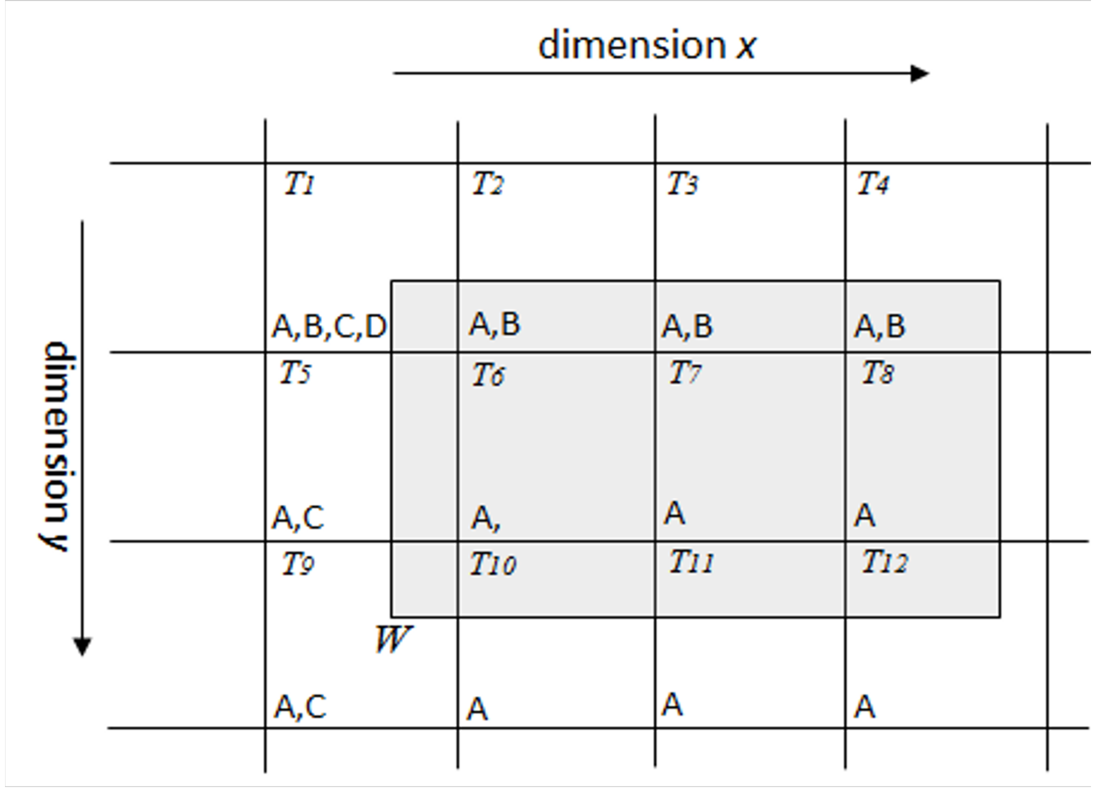
Select Relevant Classes

- **Observation 1**
  - Query W intersects tile T
  - W starts before T in dimension x
  - Disregard classes C and D
  - Examples: T2, T3, T4, T6, T7, T8, T10, T11, T12



Select Relevant Classes

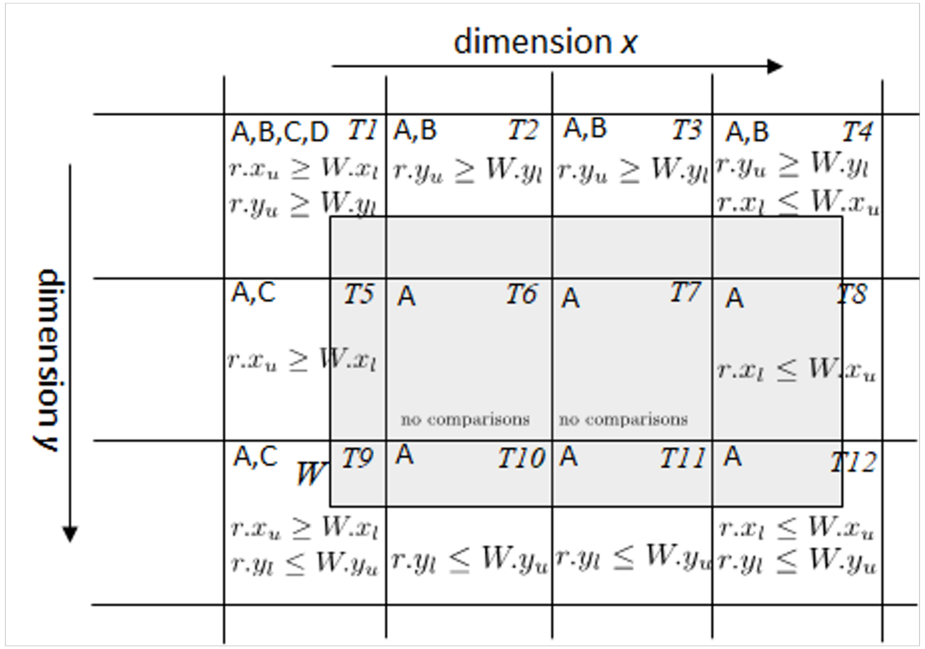
- **Observation 1**
  - Query W intersects tile T
  - W starts before T in dimension x
  - Disregard classes C and D
  - Examples: T2, T3, T4, T6, T7, T8, T10, T11, T12



- **Observation 2**
  - Query W intersects tile T
  - W starts before T in dimension y
  - Disregard classes B and D
  - Examples: T5, T6, T7, T8, T9, T10, T11, T12

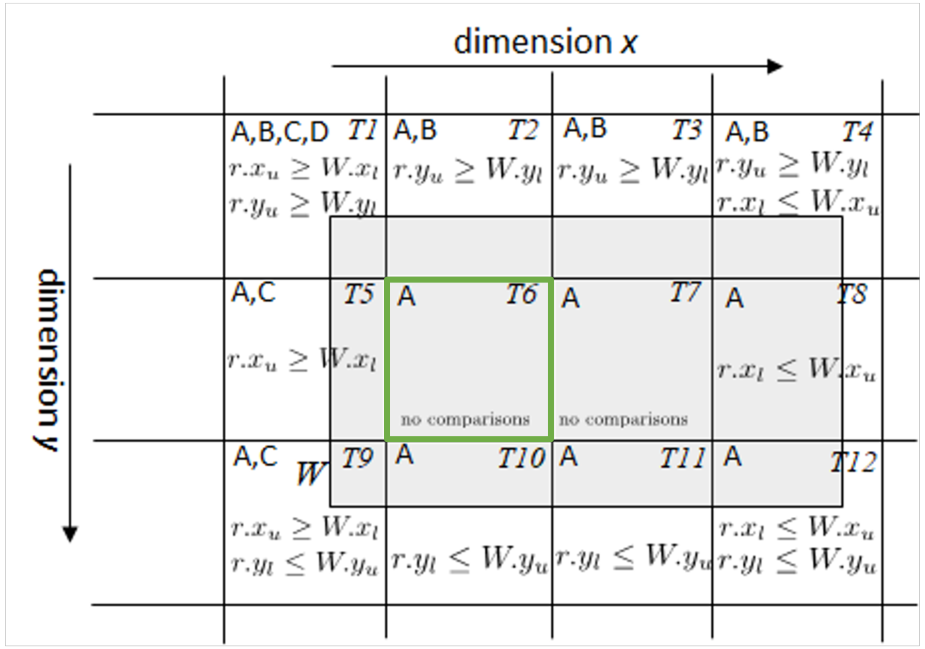
Identify Intersecting Rectangles

- **Intersection test**
  - Normally, four comparisons
  - Minimize the number of comparisons if W is bigger than tile size
- **Observation 1**
  - Tile covered by window in a dimension
  - No intersection test in this dimension
- **Observation 2**
  - Query W ends in a Tile T
  - W starts before T in dimension d
  - One comparison : rectangle.dl ≤ W.du
- **Observation 3**
  - Query W starts in a tile T
  - W ends after T in dimension d
  - One comparison: rectangle.du ≥ W.dl



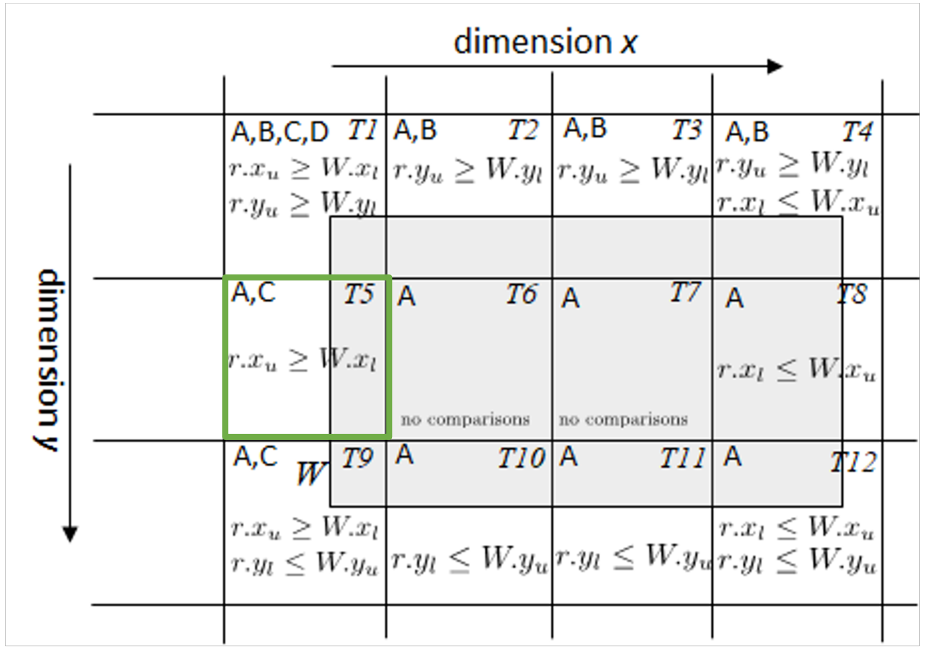
Identify Intersecting Rectangles

- **Intersection test**
  - Normally, four comparisons
  - Minimize the number of comparisons if W is bigger than tile size
- **Observation 1**
  - Tile covered by window in a dimension
  - No intersection test in this dimension
- **Observation 2**
  - Query W ends in a Tile T
  - W starts before T in dimension d
  - One comparison : rectangle.dl ≤ W.du
- **Observation 3**
  - Query W starts in a tile T
  - W ends after T in dimension d
  - One comparison: rectangle.du ≥ W.dl



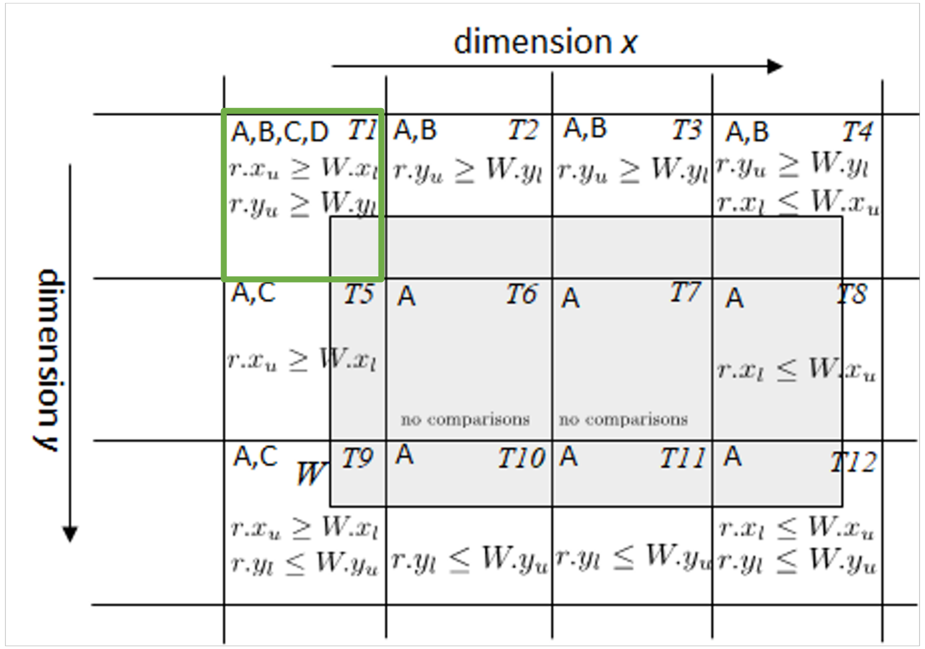
Identify Intersecting Rectangles

- **Intersection test**
  - Normally, four comparisons
  - Minimize the number of comparisons if W is bigger than tile size
- **Observation 1**
  - Tile covered by window in a dimension
  - No intersection test in this dimension
- **Observation 2**
  - Query W ends in a Tile T
  - W starts before T in dimension d
  - One comparison : rectangle.dl ≤ W.du
- **Observation 3**
  - Query W starts in a tile T
  - W ends after T in dimension d
  - One comparison: rectangle.du ≥ W.dl



Identify Intersecting Rectangles

- **Intersection test**
  - Normally, four comparisons
  - Minimize the number of comparisons if W is bigger than tile size
- **Observation 1**
  - Tile covered by window in a dimension
  - No intersection test in this dimension
- **Observation 2**
  - Query W ends in a Tile T
  - W starts before T in dimension d
  - One comparison : rectangle.dl ≤ W.du
- **Observation 3**
  - Query W starts in a tile T
  - W ends after T in dimension d
  - One comparison: rectangle.du ≥ W.dl

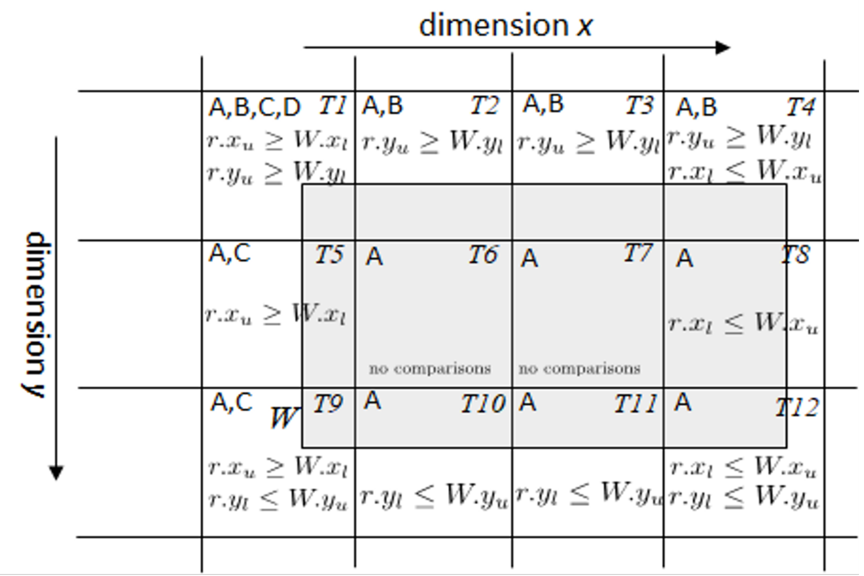


Storage Optimization

- Store the MBRs using the Decomposition Storage Model
  - Reduces the query cost
  - Improves data access locality
- Each table contains (coordinate, id) pairs
  - Sorted by coordinate
  - Used for queries where one endpoint is needed
- Not necessary to store all decompositions

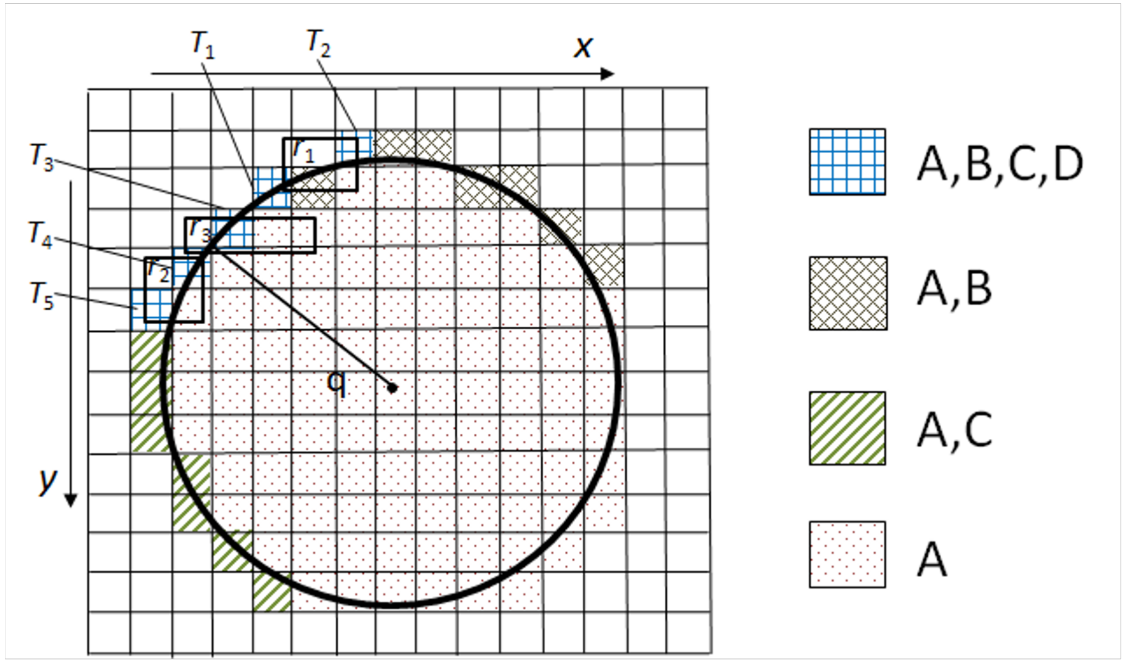
- Cons
  - Requires additional storage
  - Expensive to update

partition	required tables
T^A	$L_{xl}{}^A, L_{xu}{}^A, L_{yl}{}^A, L_{yu}{}^A$
T^B	$L_{xl}{}^B$, $L_{xu}{}^B$, $L_{yu}{}^B$
T^C	$L_{xu}{}^C$, $L_{yl}{}^C$, $L_{yu}{}^C$
T^D	$L_{xu}{}^D$, $L_{yu}{}^D$



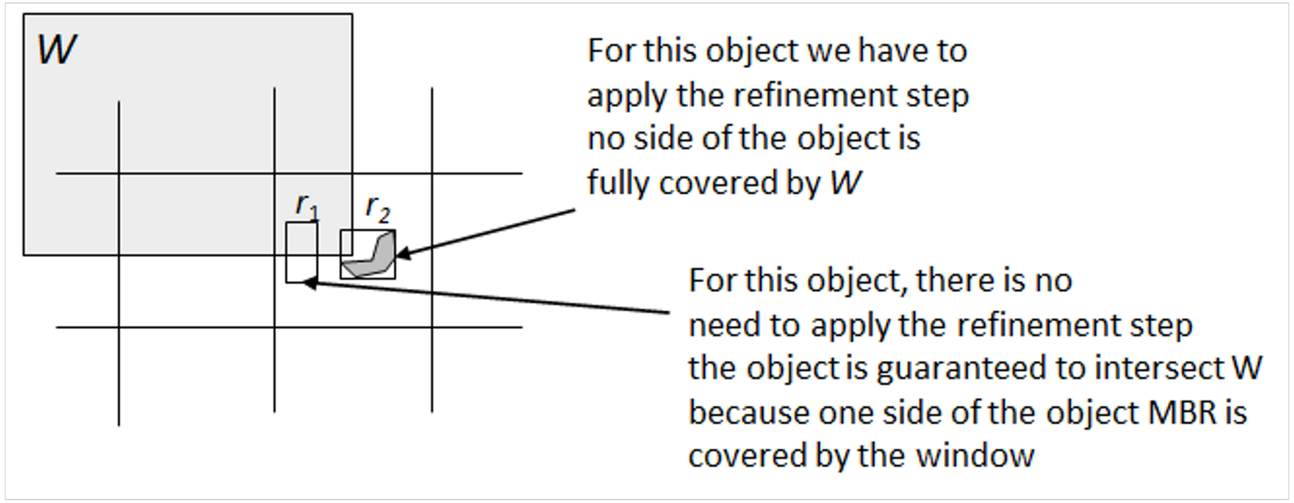
Disk Query

- If previous tile in x dimension intersects with the query -> A,C
- If previous tile in y dimension intersects with the query -> A,B
- if tile is full covered by query -> A
- If previous tiles does not intersect with the query in both dimensions -> A,B,C,D



Accelerate Refinement

- Secondary filtering before refinement
- **Observation**
  - At least one side of an MBR inside the query
  - Object does not need refinement step



Batch Query Processing

- Single-threaded and multi-threaded implementations
- Queries-based approach
  - Process each query independently
  - Assign queries to available threads in round robin
  - Cache-agnostic
- Tiles-based approach
  - For each tile
    - Find intersecting queries
    - Combine relevant classes for all queries
  - Process each tile independently
  - Assign tiles to available threads in round robin
  - Cache-conscious

Setup & Datasets

- Hardware:
  - Processor: dual Index(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20Ghz with 384 GBs of RAM
  - Hyper-threading enabled for batch processing, up to 40 threads
- Implementation:
  - Programming Language: C++
  - Operation System: CentOs Linux 7.6.1810

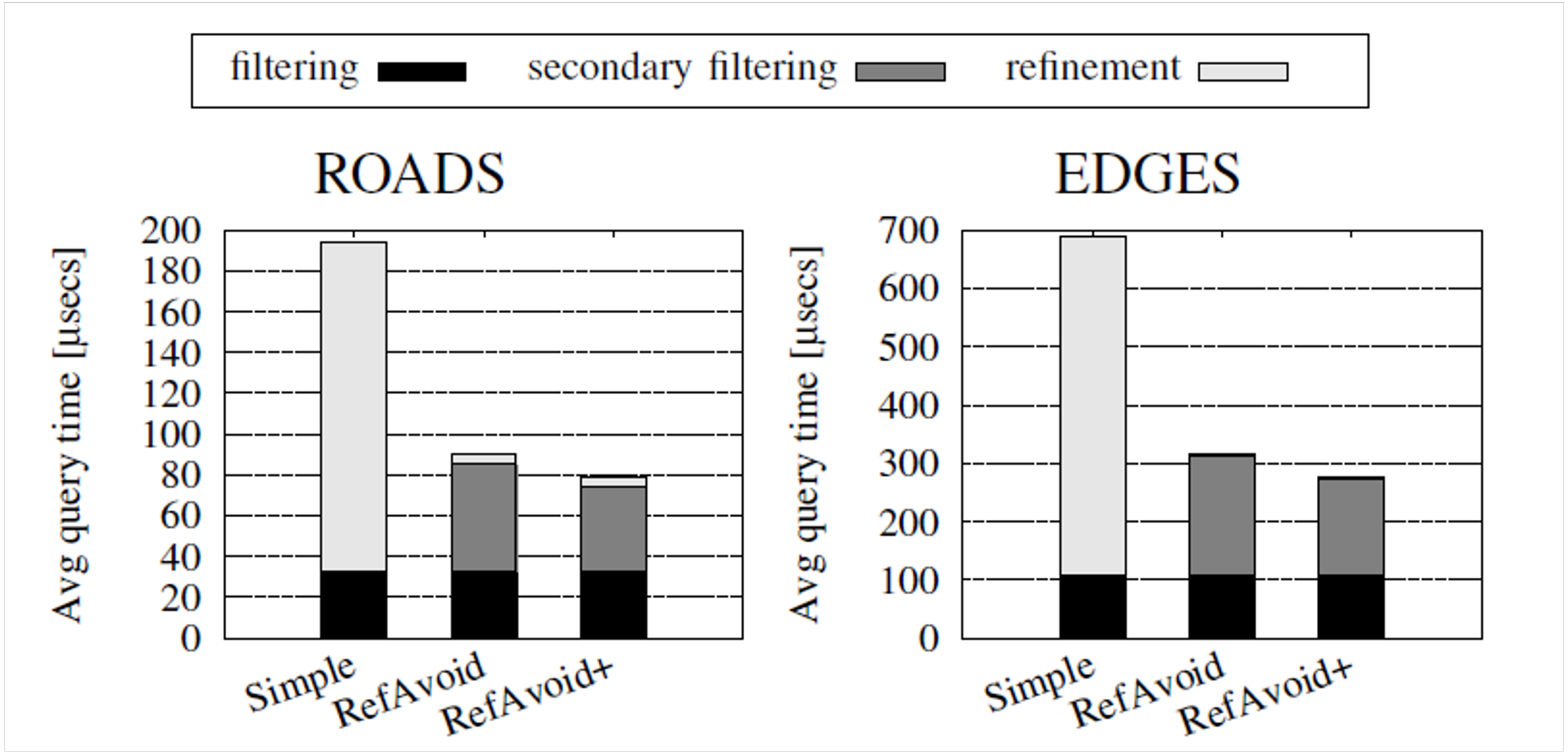
Synthetic dataset

parameter	values	Default
cardinality	1M, 5M, 10M, 50M, 100M	10M
area	$10^{-\infty}$, 10^{-14}, 10^{-12}, 10^{-8}, 10^{-6}	10^{-10}
distribution	Uniform or Zipfian (a = 1)	-

Real dataset

dataset	type	card.	avg. x-extend	avg. y-extend
ROADS	linestrings	20M	0.00001173	0.00000915
EDGES	polygons	70M	0.00000491	0.00000383
TIGER	mixed	98M	0.00000740	0.00000576

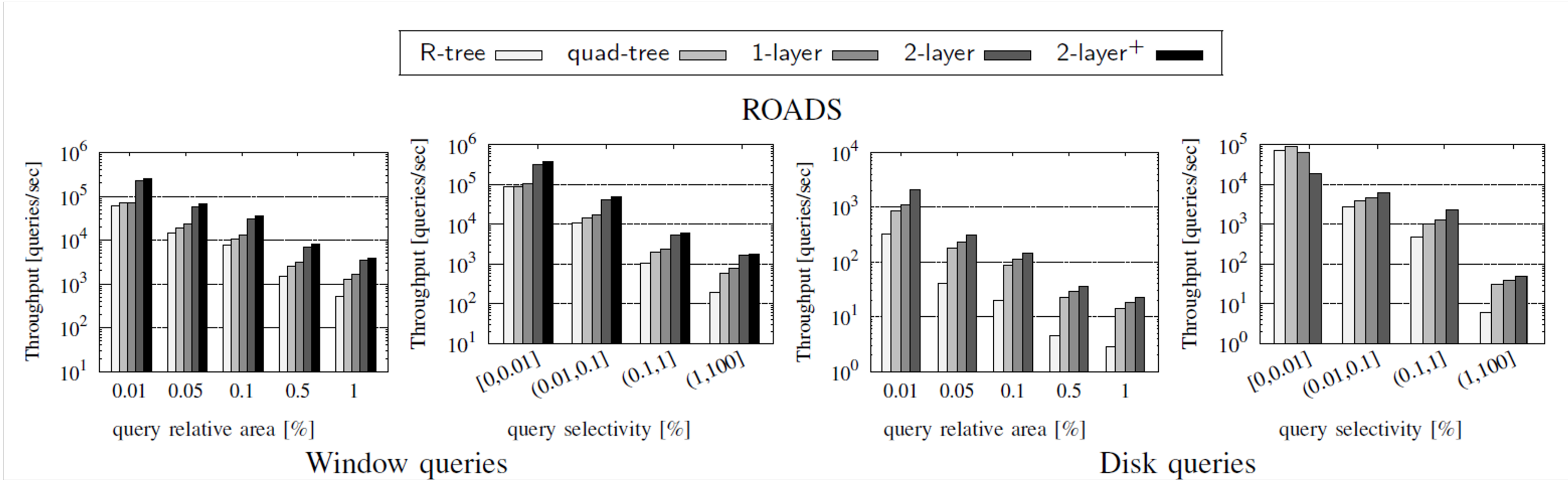
Filtering vs Refinement



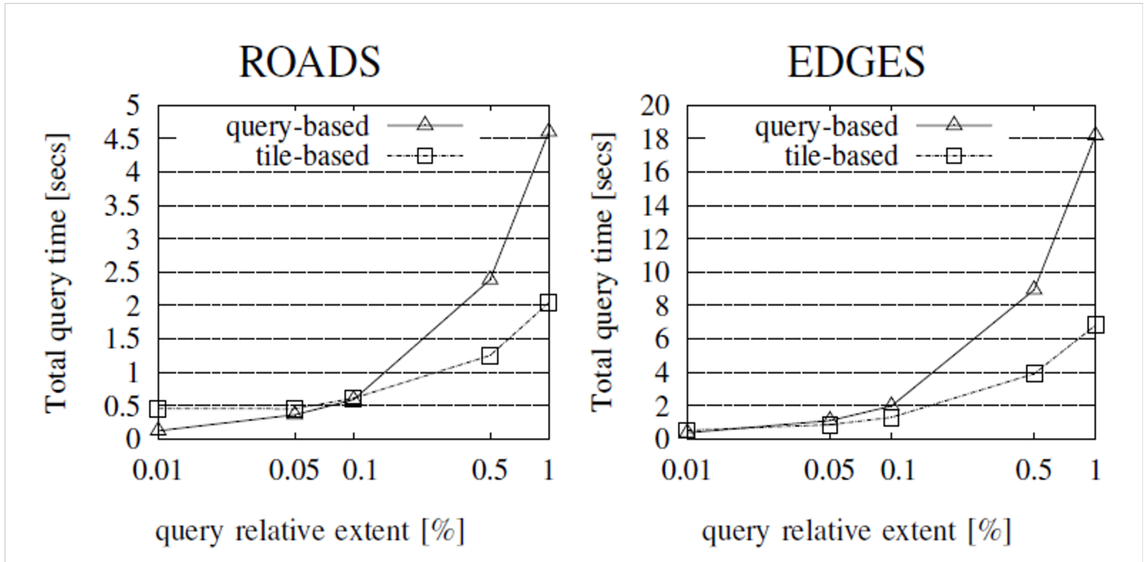
Compared Methods and their Throughput

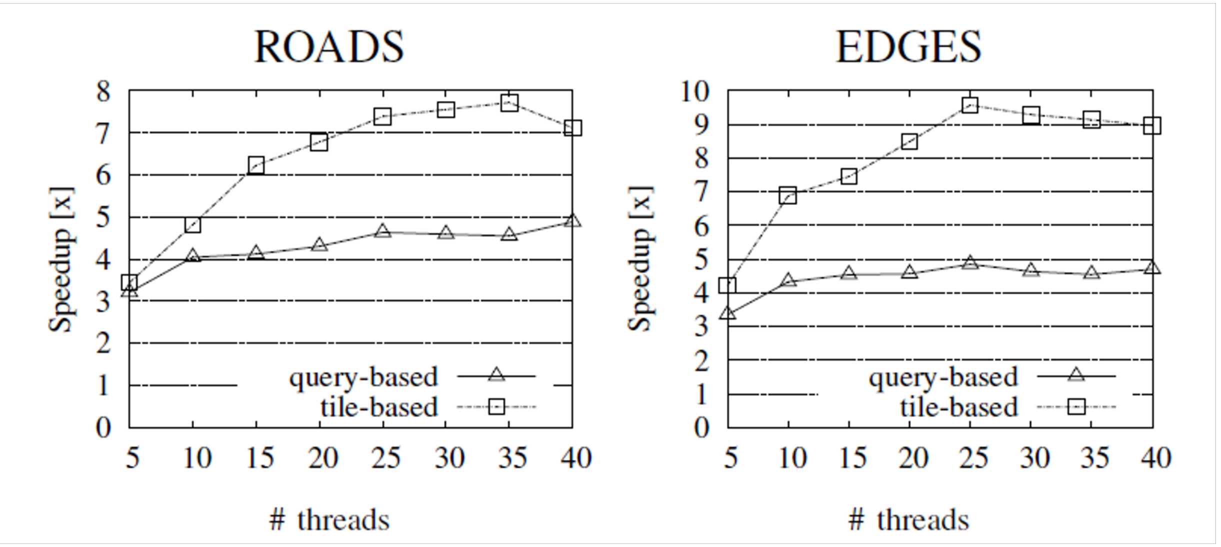
Type	index	throughput (queries/sec)	
		ROADS	EDGES
SOP	2-layer 2-layer$^+$ 1-layer quad-tree quad-tree, 2-layer	30981 36444 12597 10949 16883	9406 10855 4403 3640 5831
DOP	R-tree R*-tree BLOCK MXCIF quad-tree	7888 6415 < 1 8	2011 1610 < 1 2

Query Processing: Real Data



Batch query processing (window queries)





Conclusion

- Contributions
  - Our two-layer partitioning:
    - Easy to implement
    - Can be applied to any SOP index
    - Reduces the number of comparisons
    - Duplicate avoidance
  - Our secondary filtering technique avoids refinement step for the majority of the query results
  - Efficient processing of multiple queries in batch and in parallel

- Future work
  - Implementation of two-layer partitioning for 3D data
  - Apply two-layer partitioning in distributed data management
  - Consider other popular queries types, such as KNN and spatial join

Thank you for your attention

Questions

?