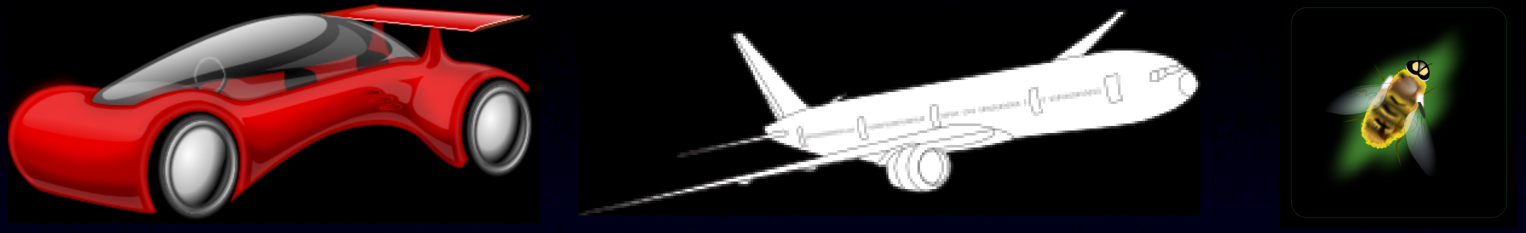
Indexing Moving Objects using Short-Lived Throwaway Indexes

SSTD 2009
Aalborg
July 10

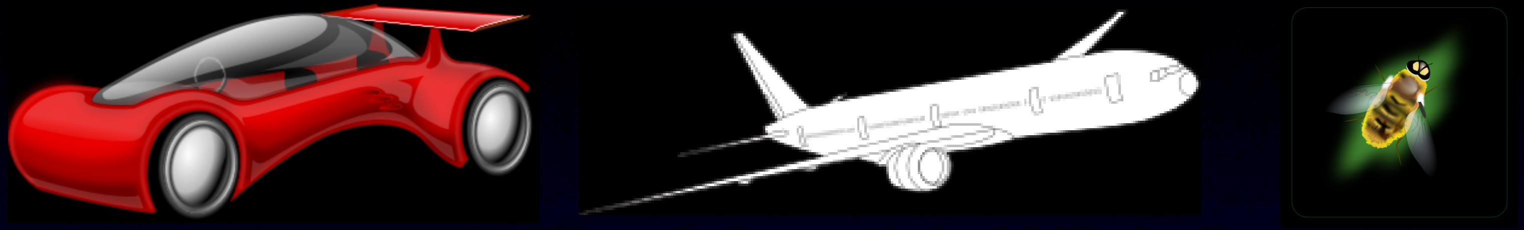
Jens Dittrich
Lukas Blunschi
Marcos Antonio Vaz Salles

Saarland University
ETH Zurich
Cornell University

Moving Objects Problem

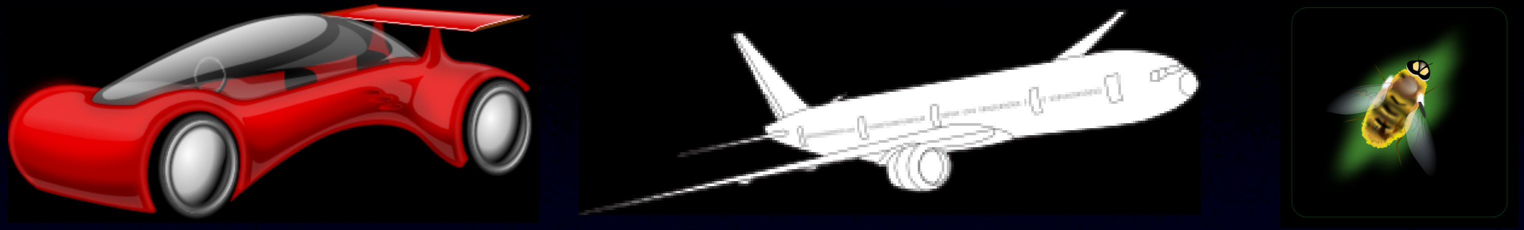


Moving Objects Problem



- Given
  - N moving objects, e.g., cars, planes, bees, particles, ...
  - space: 2D or 3D geometric

Moving Objects Problem



- Given
  - N moving objects, e.g., cars, planes, bees, particles, ...
  - space: 2D or 3D geometric
- Desired results:
  - moving objects within a range (query window)
    - as of now
    - or: in not too distant future

Why External Memory?

Why External Memory?

- 1st common assumption in existing work:
  „data does not fit into main memory“

Why External Memory?

- 1st common assumption in existing work: „data does not fit into main memory“
- but why?

Why External Memory?

- 1st common assumption in existing work: „data does not fit into main memory“
- but why?
- assume 4 bytes per dimension for current position and speed vector plus 4 bytes ID

Why External Memory?

- 1st common assumption in existing work: „data does not fit into main memory“
- but why?
- assume 4 bytes per dimension for current position and speed vector plus 4 bytes ID
- => 20 bytes per moving object

Why External Memory?

- 1st common assumption in existing work: „data does not fit into main memory“
- but why?
- assume 4 bytes per dimension for current position and speed vector plus 4 bytes ID
- => 20 bytes per moving object
- => 54 million moving objects per GB

Why External Memory?

- 1st common assumption in existing work: „data does not fit into main memory“
- but why?
- assume 4 bytes per dimension for current position and speed vector plus 4 bytes ID
- => 20 bytes per moving object
- => 54 million moving objects per GB
- and this is **ignoring** compression...

Why External Memory?

- 1st common assumption in existing work: „data does not fit into main memory“
- but why?
- assume 4 bytes per dimension for current position and speed vector plus 4 bytes ID
- => 20 bytes per moving object
- => 54 million moving objects per GB
- and this is **ignoring** compression...
- realistic: 100 million moving objects per GB

Why External Memory?

Why External Memory?

- current server hardware has at least 4 GB
  (we would call this a very small machine...)

Why External Memory?

- current server hardware has at least 4 GB
  (we would call this a very small machine...)
- 16GB main memory more common

Why External Memory?

- current server hardware has at least 4 GB
  (we would call this a very small machine...)
- 16GB main memory more common
- => 800 million moving objects in main memory

Example: infosys server

Example: infosys server

- one Server node X has:
  - Quad Core Xeon E5430, 2*6MB Cache, 2.66GHz
  - 16 GB Main Memory
  - 6 * 750 GB SATA, 7.2 rpm

Example: infosys server

- one Server node X has:
  - Quad Core Xeon E5430, 2*6MB Cache, 2.66GHz
  - 16 GB Main Memory
  - 6 * 750 GB SATA, 7.2 rpm
- 10*X = 29k € = 0.58 man years

Example: infosys server

- one Server node X has:
  - Quad Core Xeon E5430, 2*6MB Cache, 2.66GHz
  - 16 GB Main Memory
  - 6 * 750 GB SATA, 7.2 rpm
- 10*X = 29k € = 0.58 man years
- in total: 160 GB main memory

Example: infosys server

- one Server node X has:
  - Quad Core Xeon E5430, 2*6MB Cache, 2.66GHz
  - 16 GB Main Memory
  - 6 * 750 GB SATA, 7.2 rpm
- 10*X = 29k € = 0.58 man years
- in total: 160 GB main memory
- => 8 billion moving objects in main memory

Clouds/Farms/Grids

Clouds/Farms/Grids

- thousands of nodes

Clouds/Farms/Grids

- thousands of nodes
- Google, Yahoo, etc.

Clouds/Farms/Grids

- thousands of nodes
- Google, Yahoo, etc.
- clouds, map/reduce

Clouds/Farms/Grids

- thousands of nodes
- Google, Yahoo, etc.
- clouds, map/reduce
- => hundreds of billions of moving objects in main memory

Why Update?

Why Update?

- 2nd common assumption in existing work: „maintain index for incoming updates“

Why Update?

- 2nd common assumption in existing work: „maintain index for incoming updates“
- reason: cost to create an index from scratch is considered high

Why Update?

- 2nd common assumption in existing work: „maintain index for incoming updates“
- reason: cost to create an index from scratch is considered high
- therefore: maintain index

Why Update?

- 2nd common assumption in existing work: „maintain index for incoming updates“
- reason: cost to create an index from scratch is considered high
- therefore: maintain index
- but: index maintenance => random I/O

Why Update?

- 2nd common assumption in existing work: „maintain index for incoming updates“
- reason: cost to create an index from scratch is considered high
- therefore: maintain index
- but: index maintenance => random I/O
- random I/O => I/O bottleneck

Why Update?

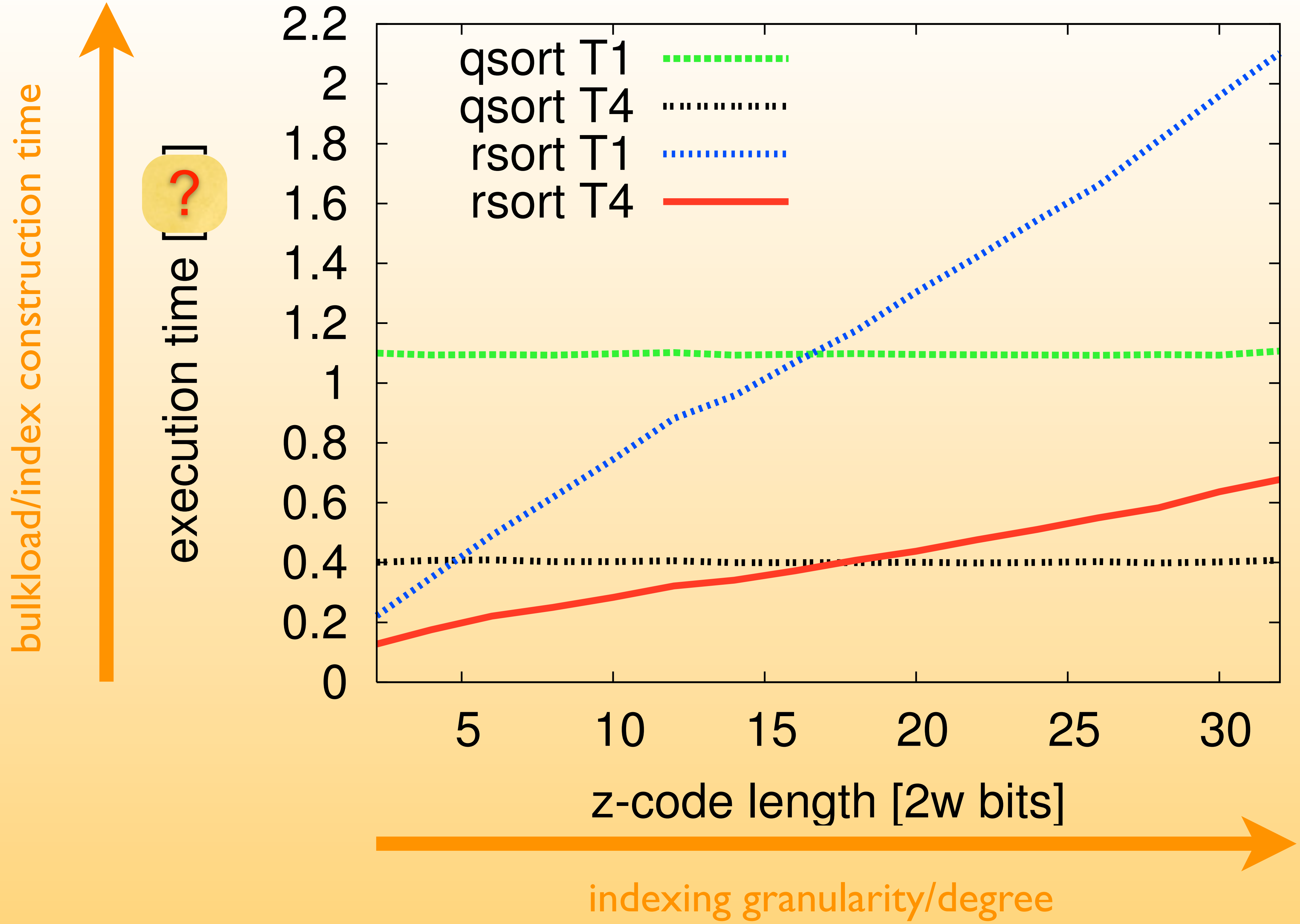
- 2nd common assumption in existing work: „maintain index for incoming updates“
- reason: cost to create an index from scratch is considered high
- therefore: maintain index
- but: index maintenance => random I/O
- random I/O => I/O bottle neck
- maaannnyyy tricks to improve this

Why Update?

- 2nd common assumption in existing work: „maintain index for incoming updates“
- reason: cost to create an index from scratch is considered high
- therefore: maintain index
- but: index maintenance => random I/O
- random I/O => I/O bottleneck
- maaannnyyy tricks to improve this
- but no real solution

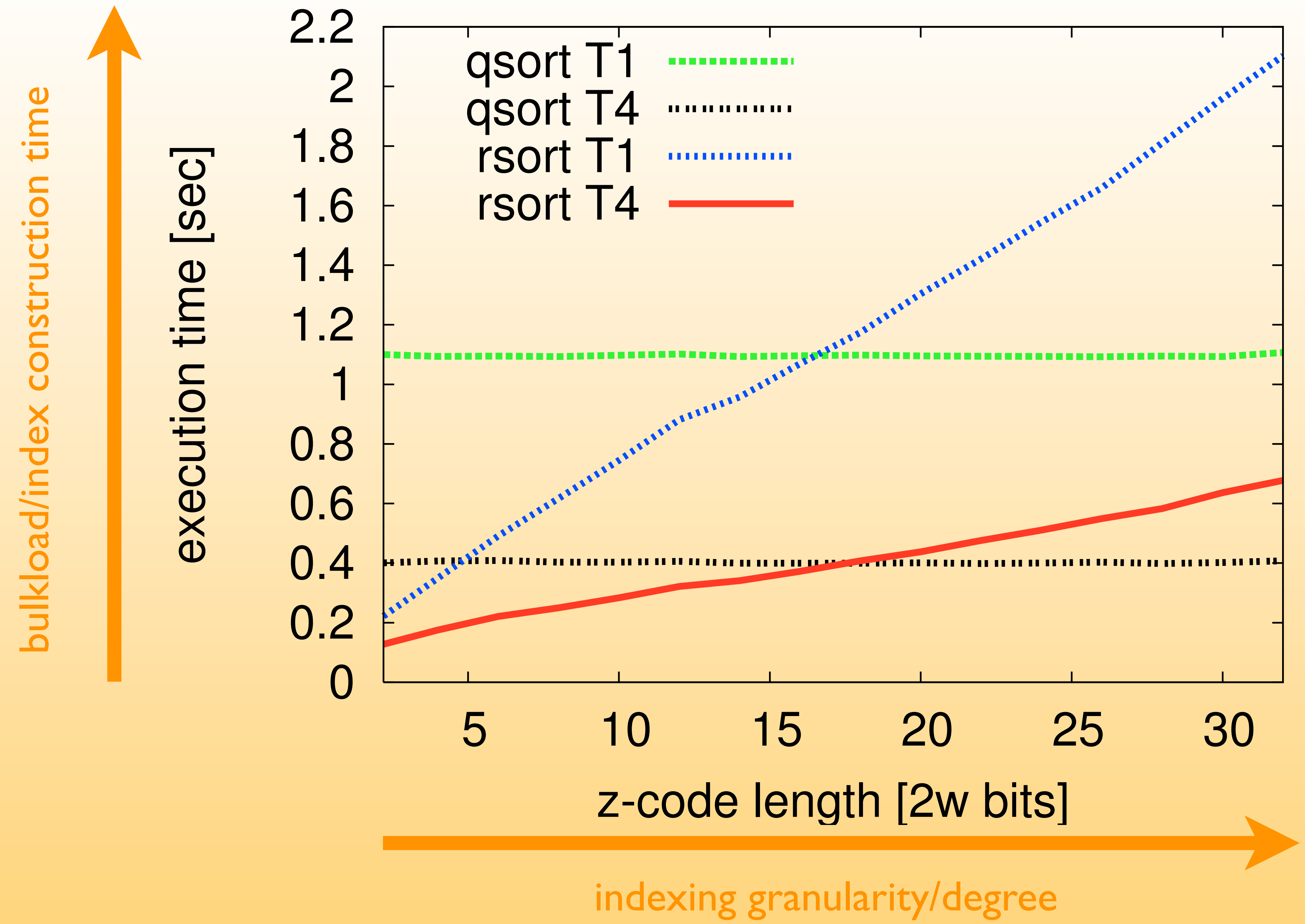
But wait: how long does it take to create an index in main memory?

But wait: how long does it take to create an index in main memory?



N=6.4M
Tx AMD cores
at 2.4 GHz

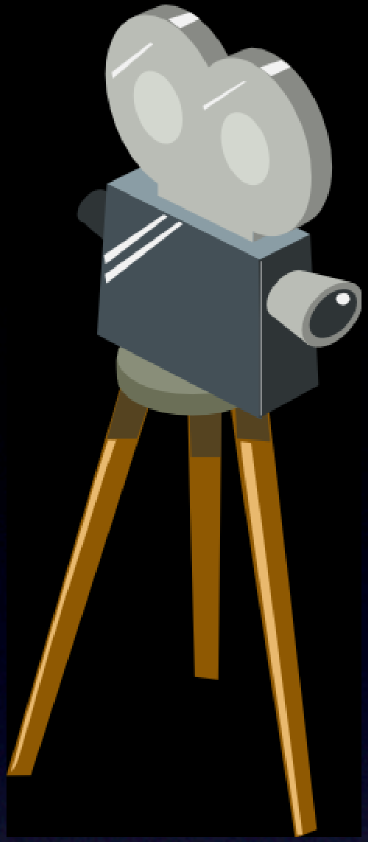
But wait: how long does it take to create an index in main memory?



N=6.4M
Tx AMD cores
at 2.4 GHz

Movies Analogy

Movies Analogy

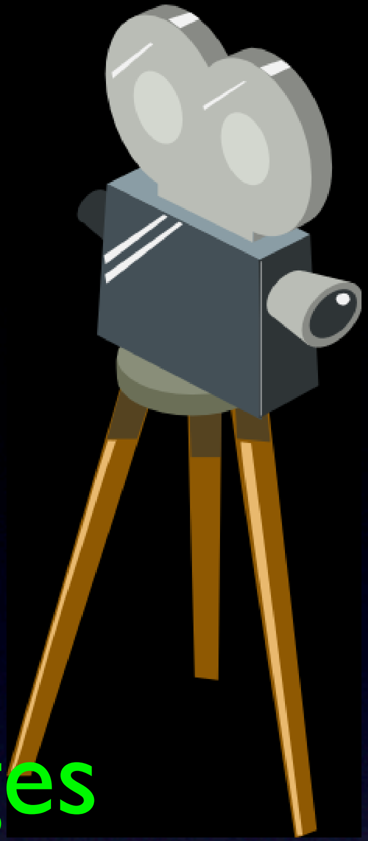


- moving picture capturing

Movies Analogy

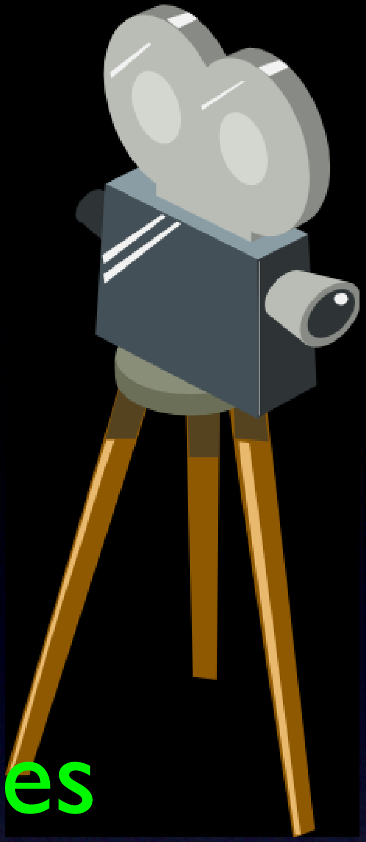
- moving picture capturing
- => so far technically impossible!

Movies Analogy



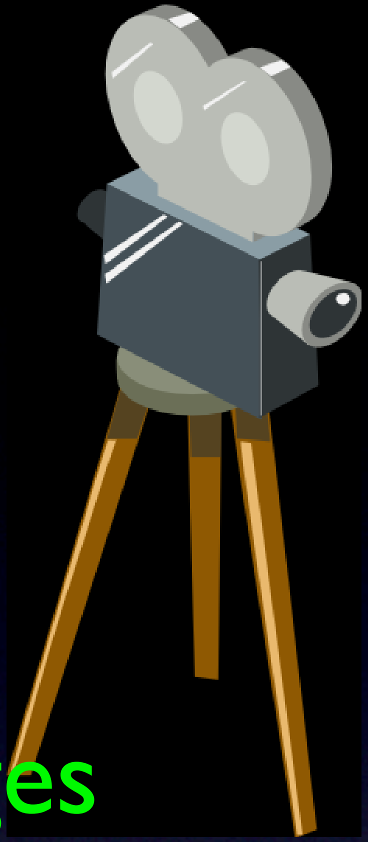
- moving picture capturing
- => so far technically impossible!
- movie camera shoots series of still images

Movies Analogy



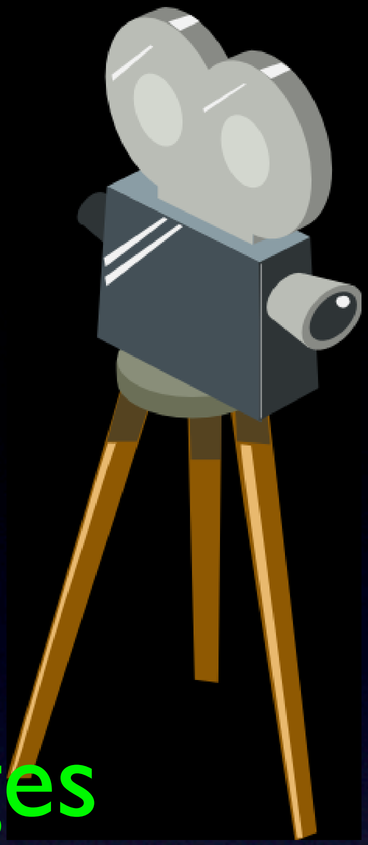
- moving picture capturing
- => so far technically impossible!
- movie camera shoots series of still images
- cinema shows series of still images

Movies Analogy



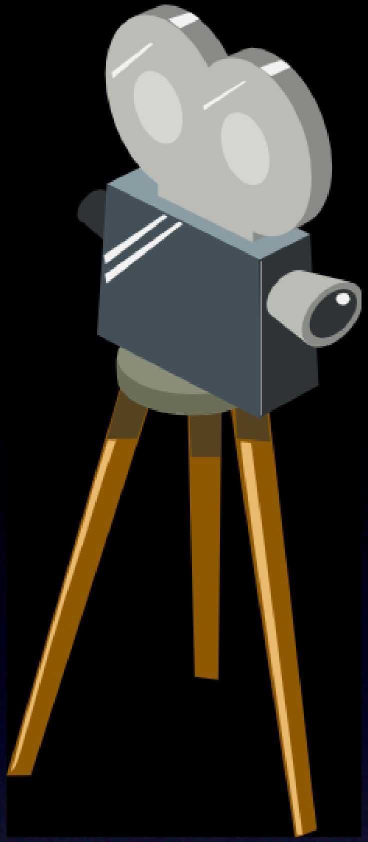
- moving picture capturing
- => so far technically impossible!
- movie camera shoots series of still images
- cinema shows series of still images
- inertia of human eye

Movies Analogy

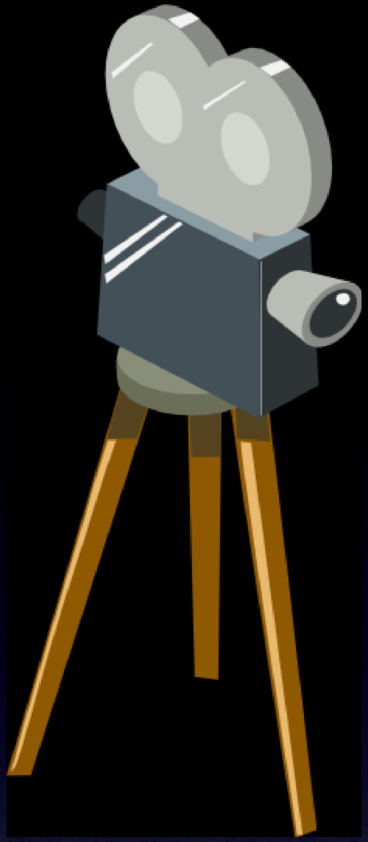


- moving picture capturing
- => so far technically impossible!
- movie camera shoots series of still images
- cinema shows series of still images
- inertia of human eye
- => human brain is tricked into believing that it sees a continuous movement

Movies Algorithm

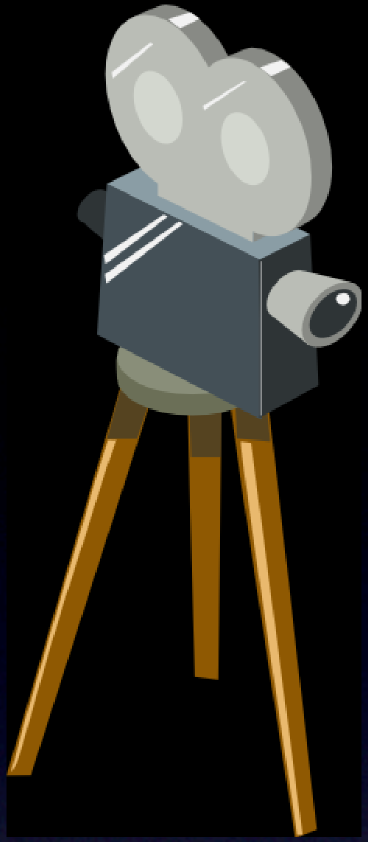


Movies Algorithm



- moving **index** capturing

Movies Algorithm

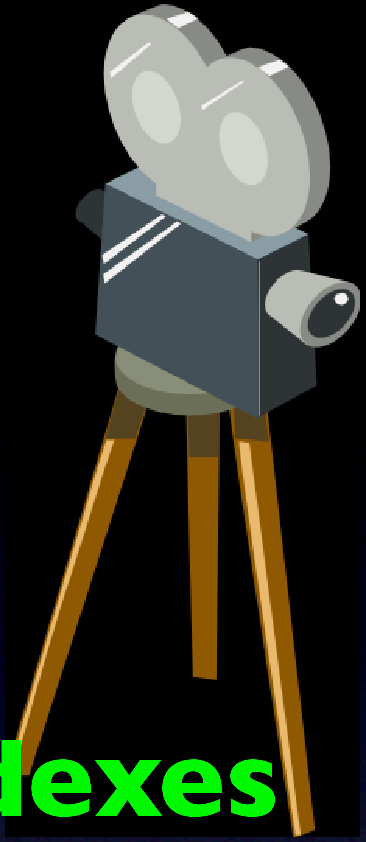


- moving **index** capturing
- =>so far not done!

Movies Algorithm

- moving **index** capturing
- =>so far not done!
- indexer shoots a quick series of still **indexes**

Movies Algorithm



- moving **index** capturing
- =>so far not done!
- indexer shoots a quick series of still **indexes**
- query processor shows series of still **indexes**

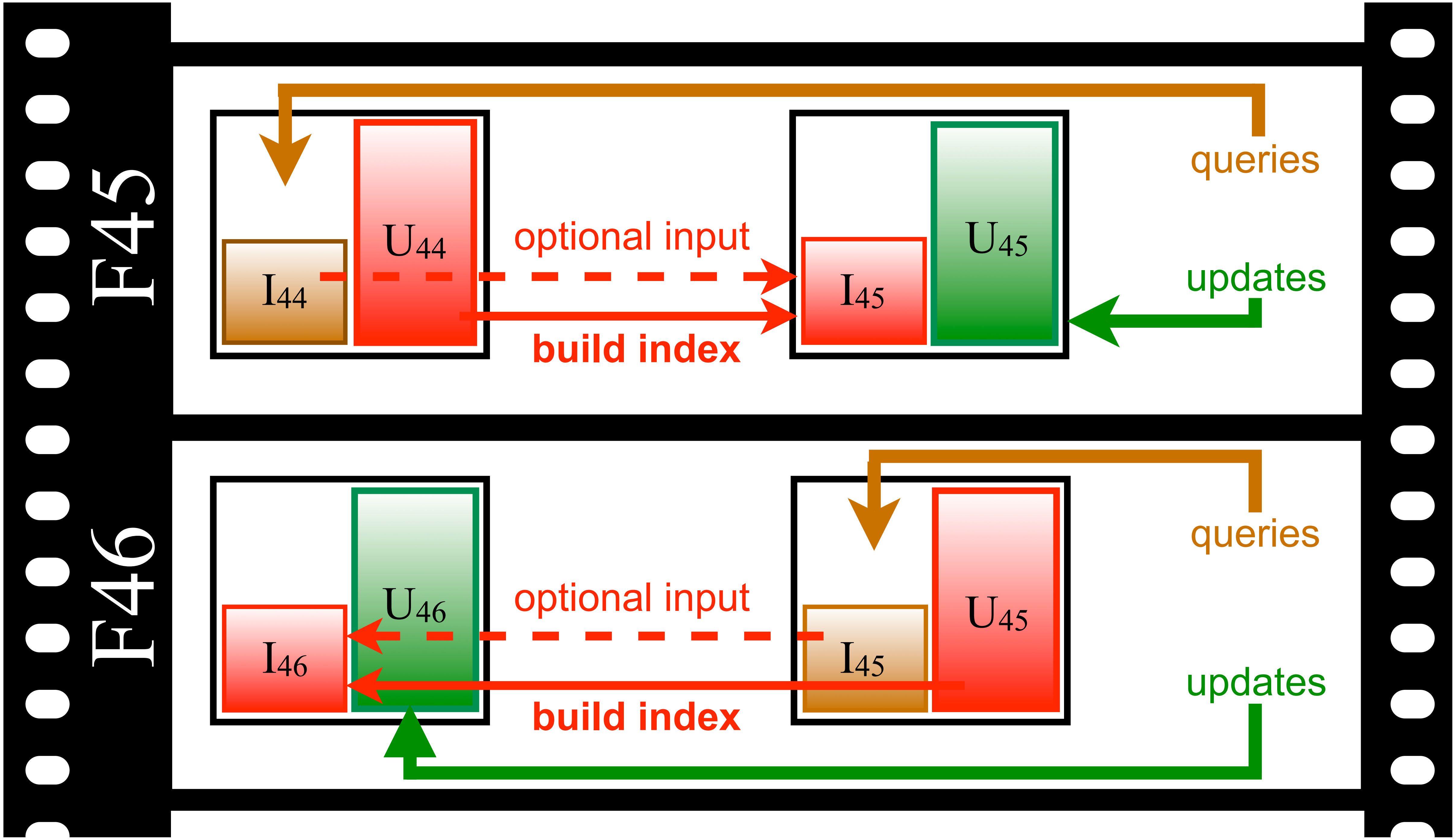
Movies Algorithm

- moving **index** capturing
- =>so far not done!
- indexer shoots a quick series of still **indexes**
- query processor shows series of still **indexes**
- inertia of object movement

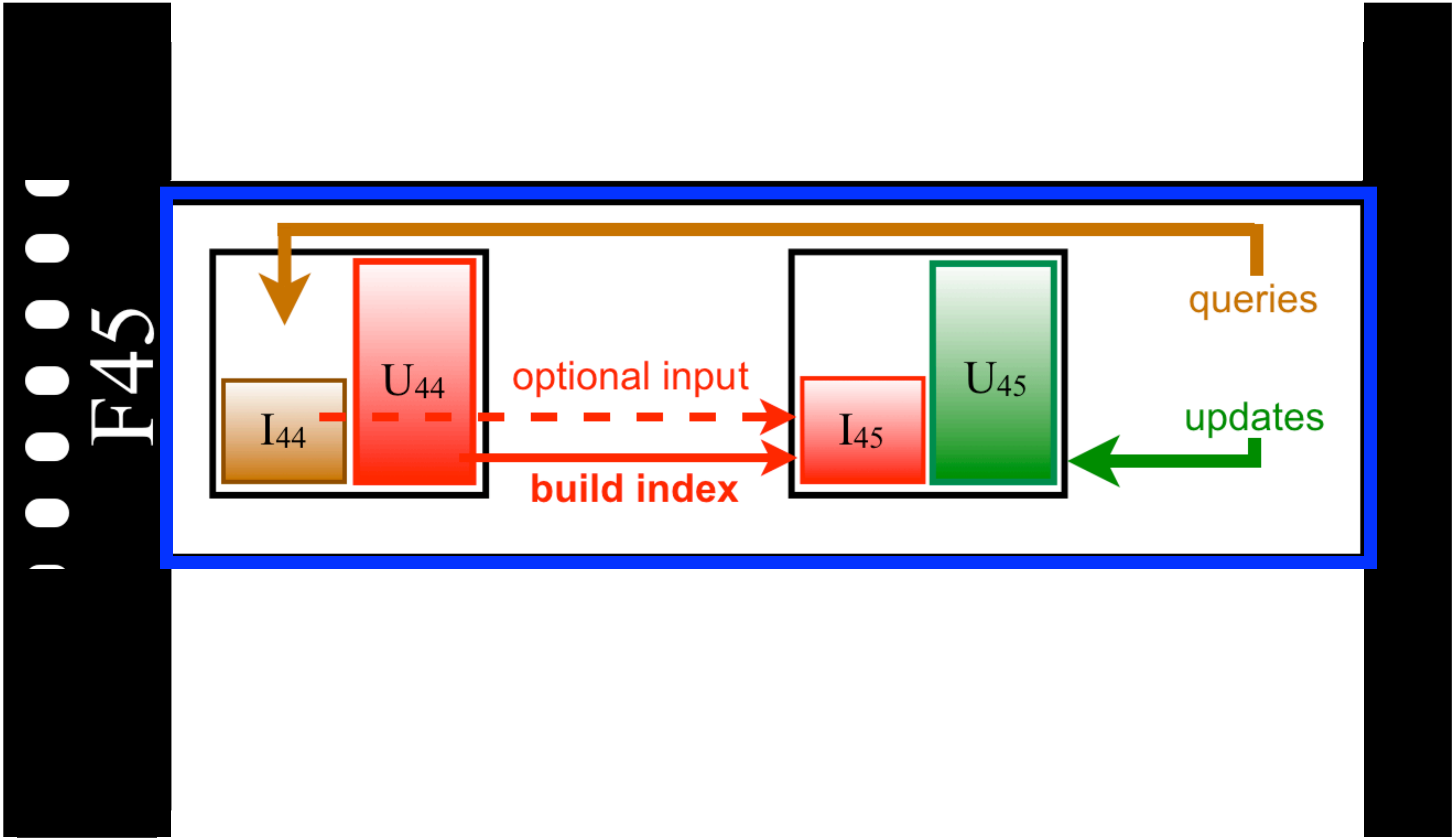
Movies Algorithm

- moving **index** capturing
- =>so far not done!
- indexer shoots a quick series of still **indexes**
- query processor shows series of still **indexes**
- inertia of object movement
- => applications are tricked into believing that they see a continuous movement

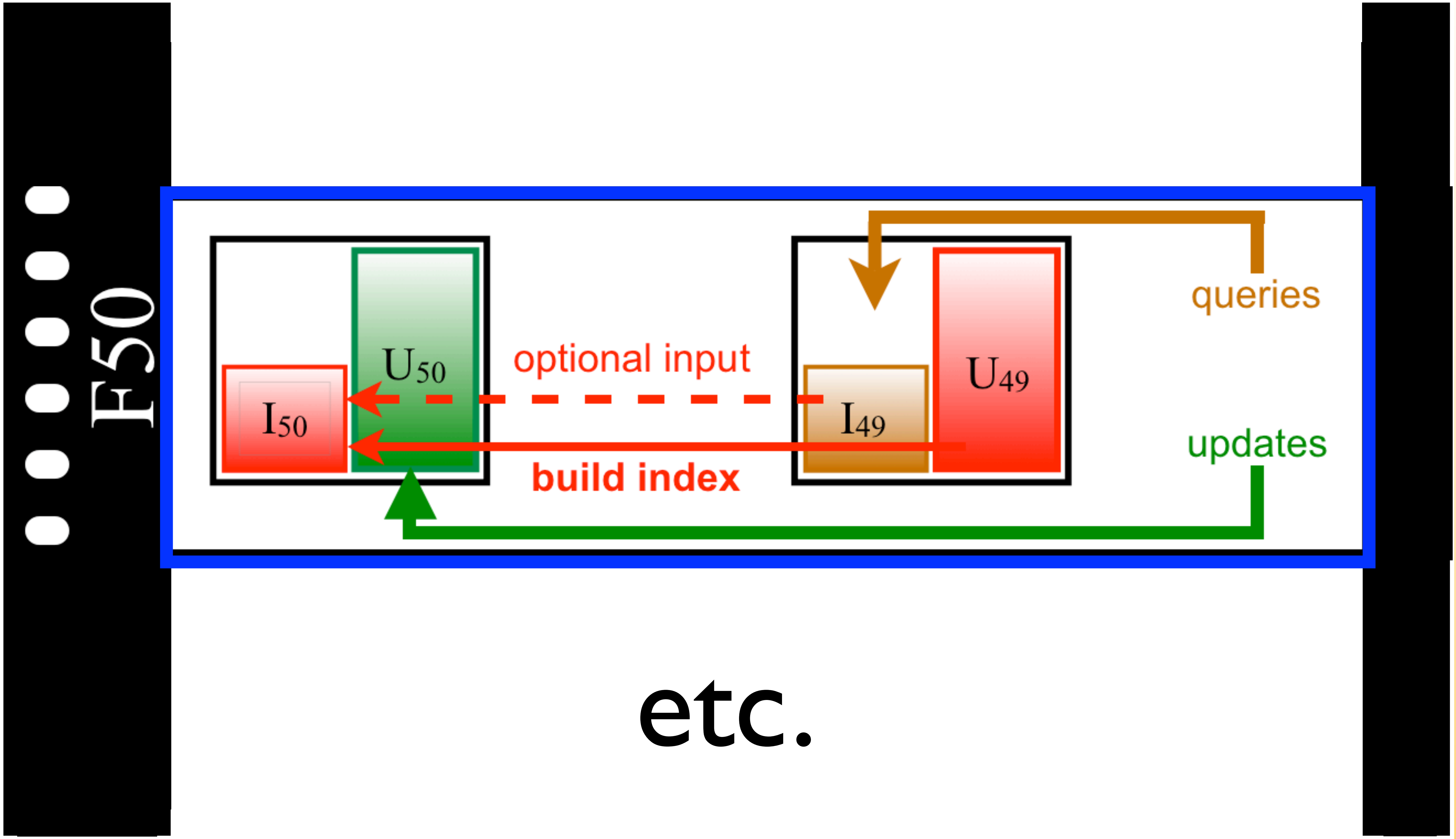
Main Algorithm



Main Algorithm



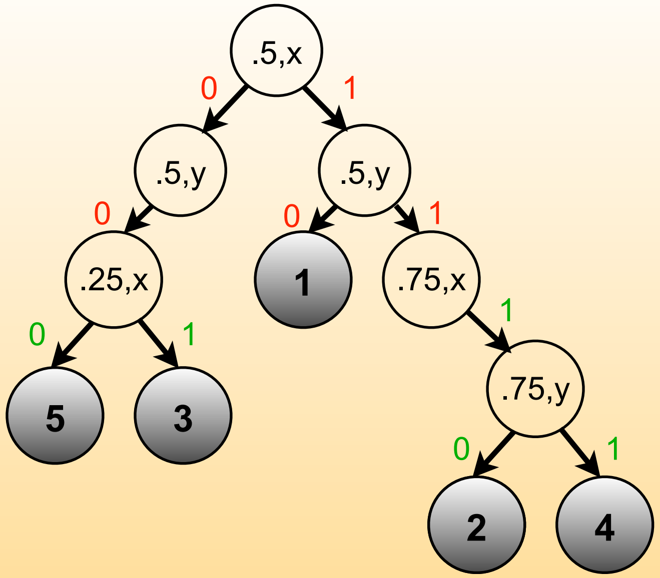
Main Algorithm



etc.

Indexing Strategy

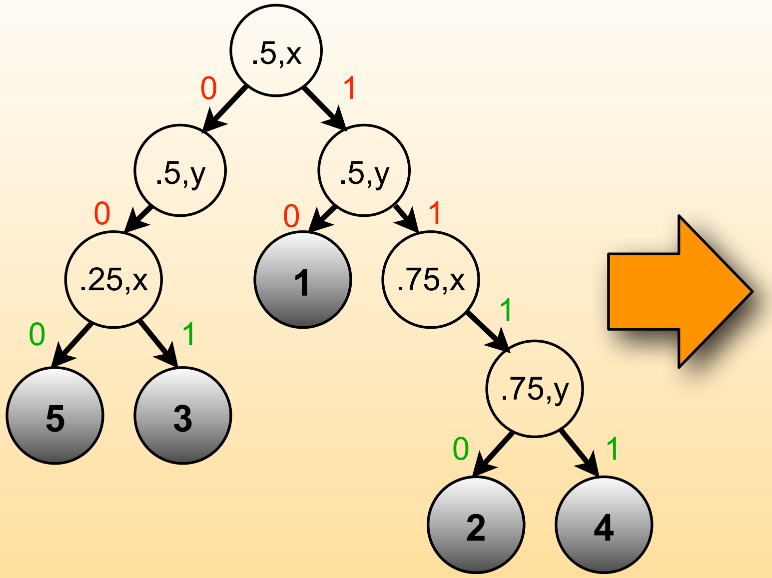
Indexing Strategy



(a) kd-trie

(a) any trie-partitioning

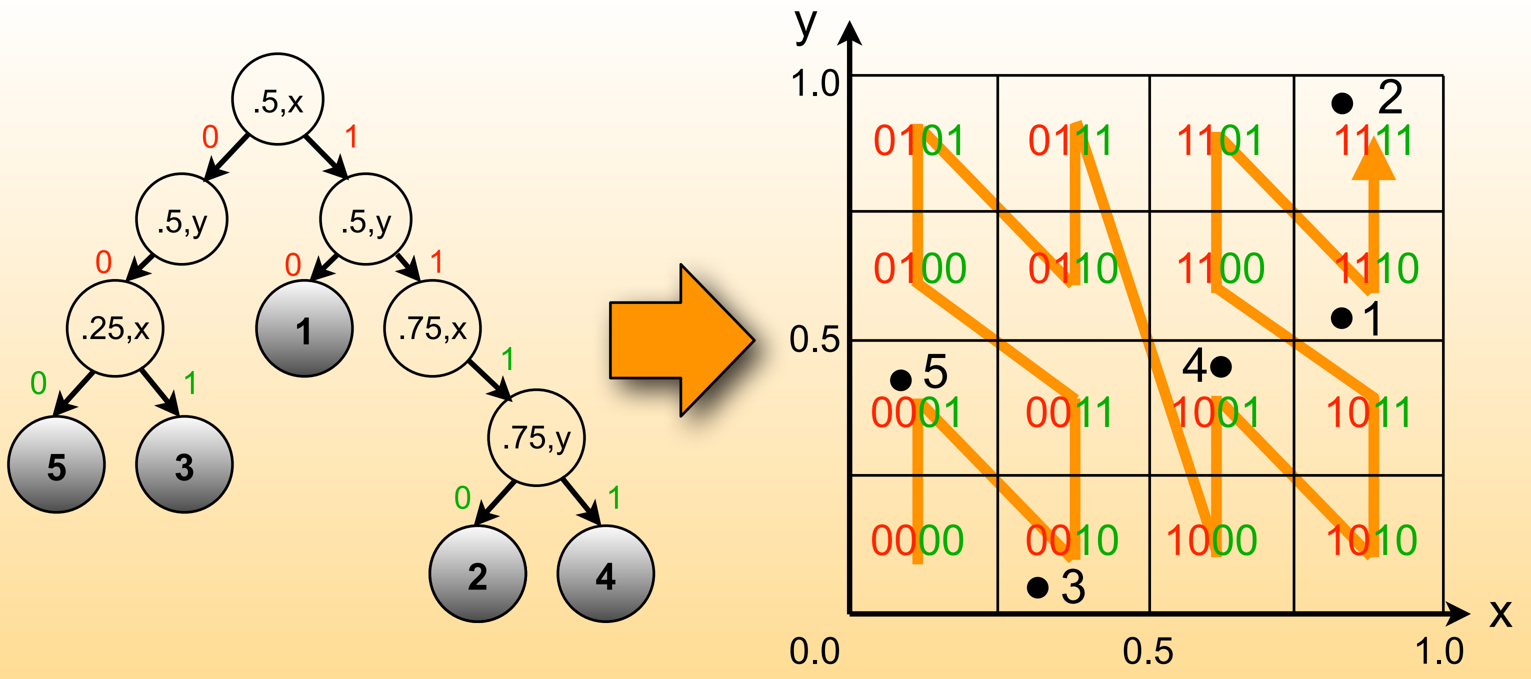
Indexing Strategy



(a) kd-trie

(a) any trie-partitioning

Indexing Strategy



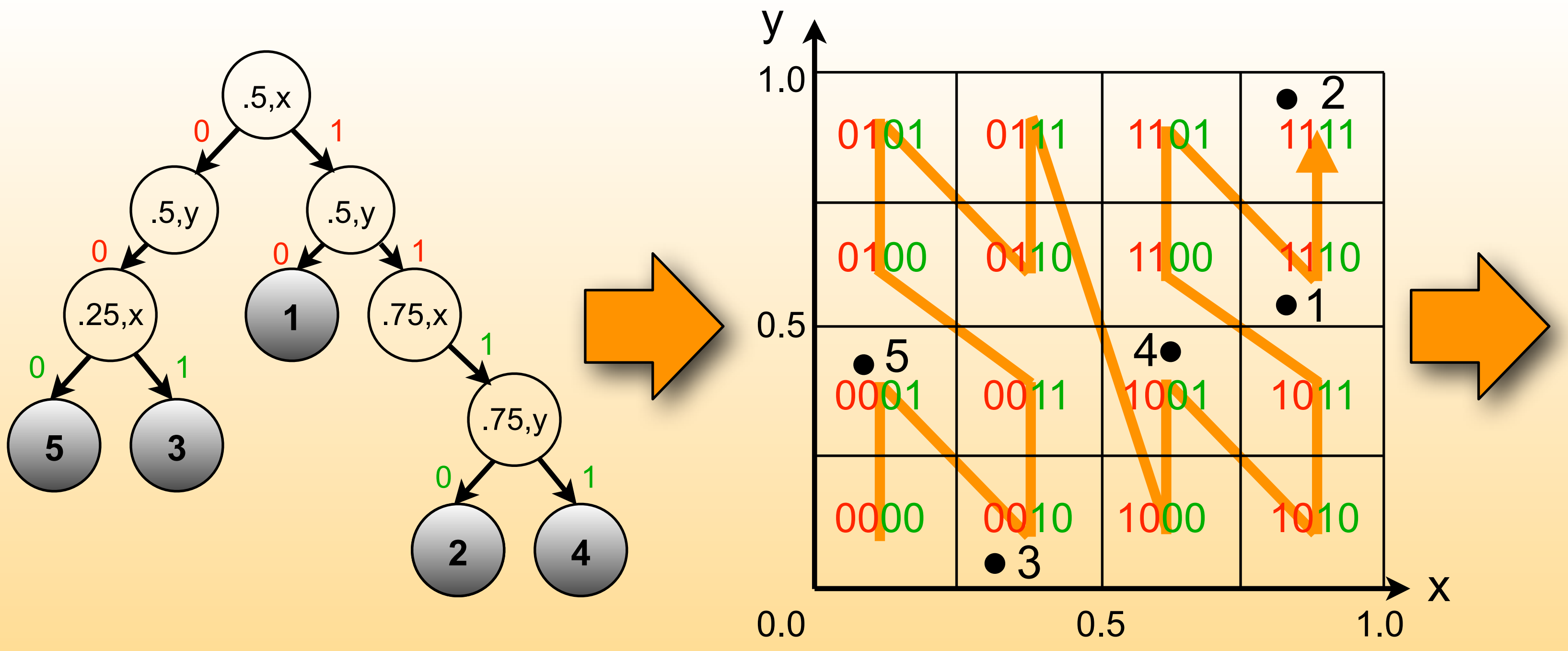
(a) kd-trie

(b) data and z-curve data partitioning

(a) any trie-partitioning

(b) mapped to any space-filling curve

Indexing Strategy



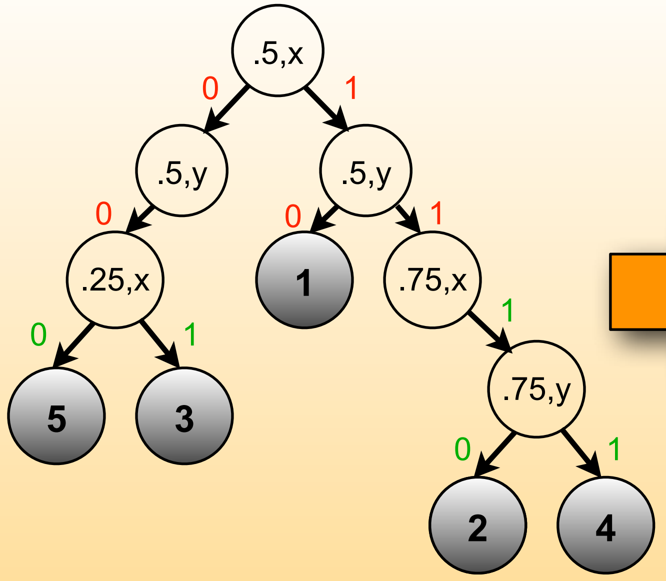
(a) kd-trie

(b) data and z-curve data partitioning

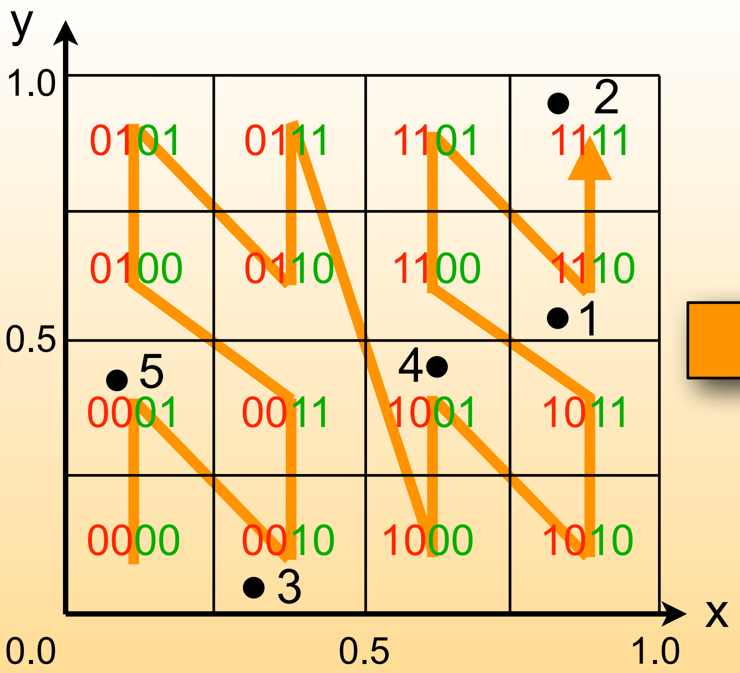
(a) any trie-partitioning

(b) mapped to any space-filling curve

Indexing Strategy



(a) kd-trie



(b) data and z-curve data partitioning

0001	10...11	1...1	0....00101
0010	01...01	0...1	0....00011
1001	11...11	0...0	0....00100
1110	00...10	1...1	0....00001
1111	10...11	1...0	0....00010
2w bits zcode	32-2w bits data suffix	5 bits $\vec{sv}$	27 bits OID

(c) index [w=2]

(a) any trie-partitioning

(b) mapped to any space-filling curve

(c) represented in compressed array

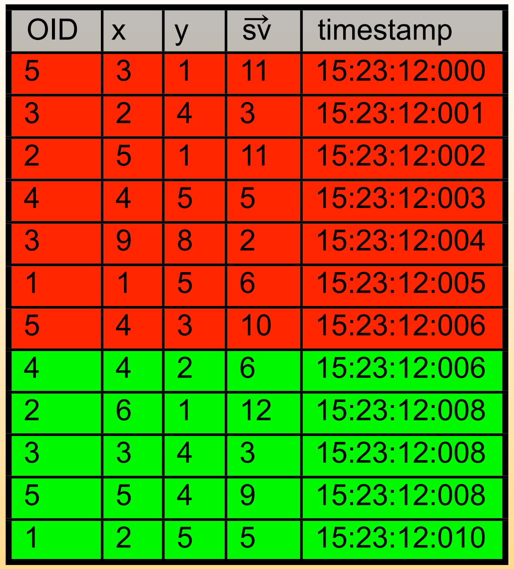
(or any other bulk-loaded tree structure)

How to Organize Update Buffers?

How to Organize Update Buffers?

(a) **Logged** MOVIES:

- log of updates
- pros: no latency for insert
- cons: possibly large



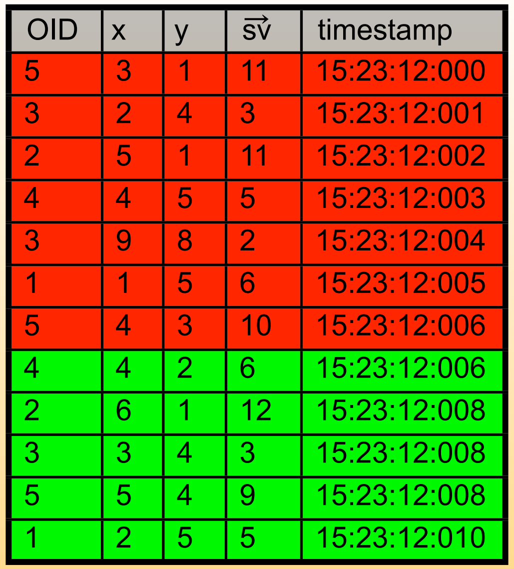
OID	x	y	$\vec{sv}$	timestamp
5	3	1	11	15:23:12:000
3	2	4	3	15:23:12:001
2	5	1	11	15:23:12:002
4	4	5	5	15:23:12:003
3	9	8	2	15:23:12:004
1	1	5	6	15:23:12:005
5	4	3	10	15:23:12:006
4	4	2	6	15:23:12:006
2	6	1	12	15:23:12:008
3	3	4	3	15:23:12:008
5	5	4	9	15:23:12:008
1	2	5	5	15:23:12:010

(a) log-buffer

How to Organize Update Buffers?

(a) **Logged** MOVIES:

- log of updates
- pros: no latency for insert
- cons: possibly large

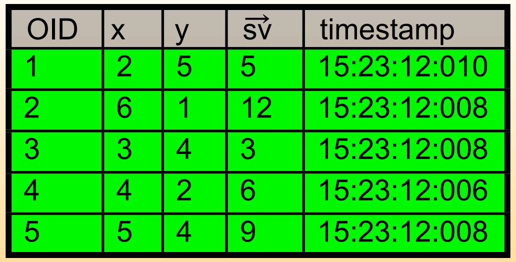


OID	x	y	$\vec{sv}$	timestamp
5	3	1	11	15:23:12:000
3	2	4	3	15:23:12:001
2	5	1	11	15:23:12:002
4	4	5	5	15:23:12:003
3	9	8	2	15:23:12:004
1	1	5	6	15:23:12:005
5	4	3	10	15:23:12:006
4	4	2	6	15:23:12:006
2	6	1	12	15:23:12:008
3	3	4	3	15:23:12:008
5	5	4	9	15:23:12:008
1	2	5	5	15:23:12:010

(a) log-buffer

(b) **Aggregated** MOVIES:

- keep most recent key for object
- cons: latency for insert
- pros: smaller



OID	x	y	$\vec{sv}$	timestamp
1	2	5	5	15:23:12:010
2	6	1	12	15:23:12:008
3	3	4	3	15:23:12:008
4	4	2	6	15:23:12:006
5	5	4	9	15:23:12:008

(b) aggregation buffer

Staleness

Staleness

- results delivered by a query may be slightly stale (=out-of-date)

Staleness

- results delivered by a query may be slightly stale (=out-of-date)
- assume current frame is F45

Staleness

- results delivered by a query may be slightly stale (=out-of-date)
- assume current frame is F45
- a result returned by index I44 may have been updated already, but...

Staleness

- results delivered by a query may be slightly stale (=out-of-date)
- assume current frame is F45
- a result returned by index I44 may have been updated already, but...
  - ...is currently used to build a new index => result will become available in F46

Staleness

- results delivered by a query may be slightly stale (=out-of-date)
- assume current frame is F45
- a result returned by index I44 may have been updated already, but...
    - ...is currently used to build a new index => result will become available in F46
    - ...is collected in current update buffer => result will become available in F47

Staleness

- results delivered by a query may be slightly stale (=out-of-date)
- assume current frame is F45
- a result returned by index I44 may have been updated already, but...
  - ...is currently used to build a new index => result will become available in F46
  - ...is collected in current update buffer => result will become available in F47
- Staleness $\leq 2 * t_{\text{Phase Time}}$

SAARLAND UNIVERSITY
COMPUTER SCIENCE

PI MOVIES

PI MOVIES

- =predictive indexing strategy

PI MOVIES

- =predictive indexing strategy
- build index for a single point in time t_{index}

PI MOVIES

- =predictive indexing strategy
- build index for a single point in time t_{index}
- pick t_{index} in near future

PI MOVIES

- =predictive indexing strategy
- build index for a single point in time t_{index}
- pick t_{index} in near future
- t_{index} chosen to minimize query enlargements (details see paper)

PI MOVIES

- =predictive indexing strategy
- build index for a single point in time t_{index}
- pick t_{index} in near future
- t_{index} chosen to minimize query enlargements (details see paper)
- pros: no timestamps in index required

PI MOVIES

- =predictive indexing strategy
- build index for a single point in time t_{index}
- pick t_{index} in near future
- t_{index} chosen to minimize query enlargements (details see paper)
- pros: no timestamps in index required
- pros: less storage space

PI MOVIES

- =predictive indexing strategy
- build index for a single point in time t_{index}
- pick t_{index} in near future
- t_{index} chosen to minimize query enlargements (details see paper)
- pros: no timestamps in index required
- pros: less storage space
- cons: CPU intensive as incoming updates need to be translated

NPI MOVIES

NPI MOVIES

- =non-predictive indexing strategy

NPI MOVIES

- =non-predictive indexing strategy
- index contains objects valid at different times

NPI MOVIES

- =non-predictive indexing strategy
- index contains objects valid at different times
- cons: larger query rewrite

NPI MOVIES

- =non-predictive indexing strategy
- index contains objects valid at different times
- cons: larger query rewrite
- cons: timestamps need to be stored

NPI MOVIES

- =non-predictive indexing strategy
- index contains objects valid at different times
- cons: larger query rewrite
- cons: timestamps need to be stored
- pros: less CPU intensive as incoming updates do not need to be translated

Experiments

Experiments

- largest road network ever used in experiments

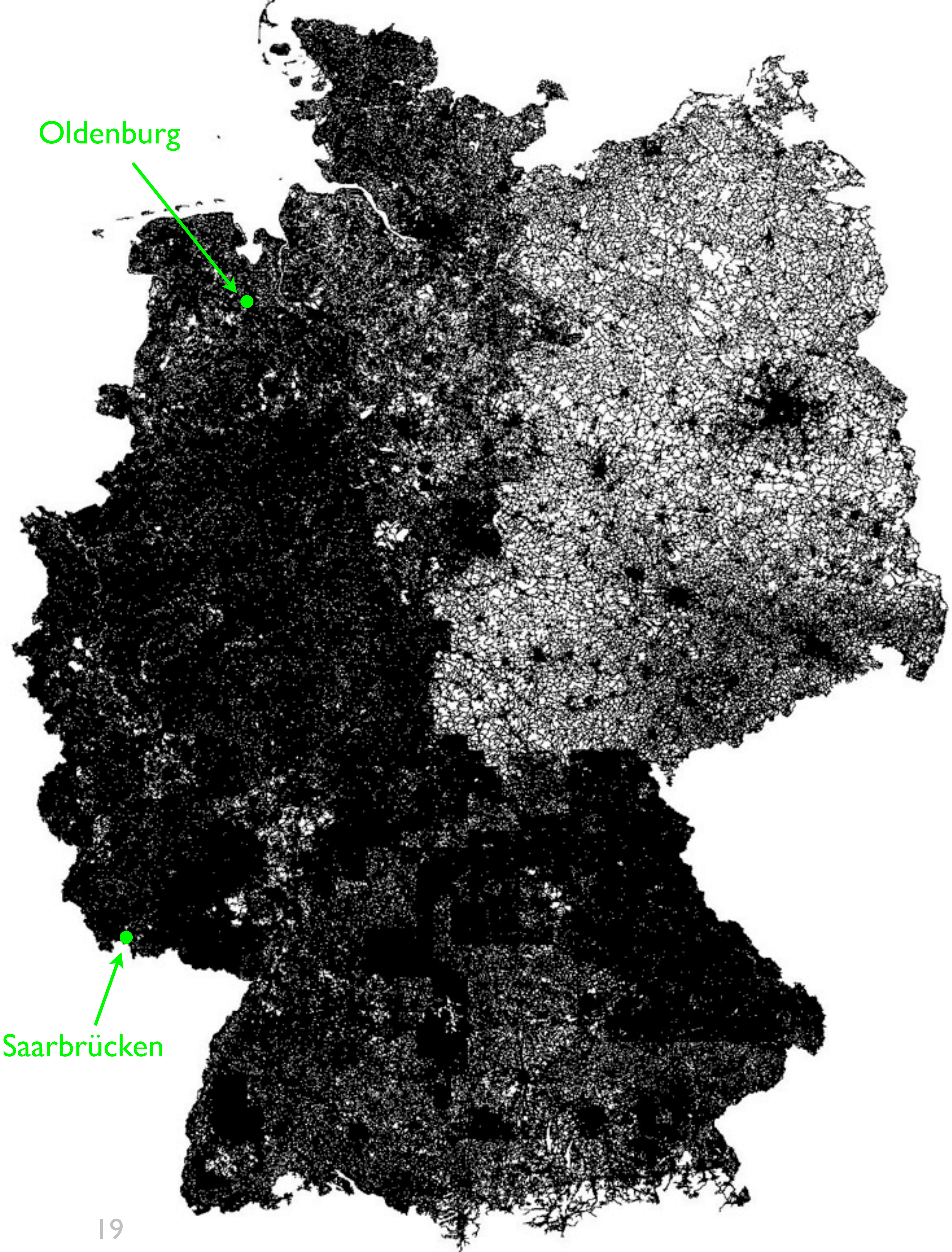
Experiments

- largest road network ever used in experiments
- up to 100 million moving objects

Experiments

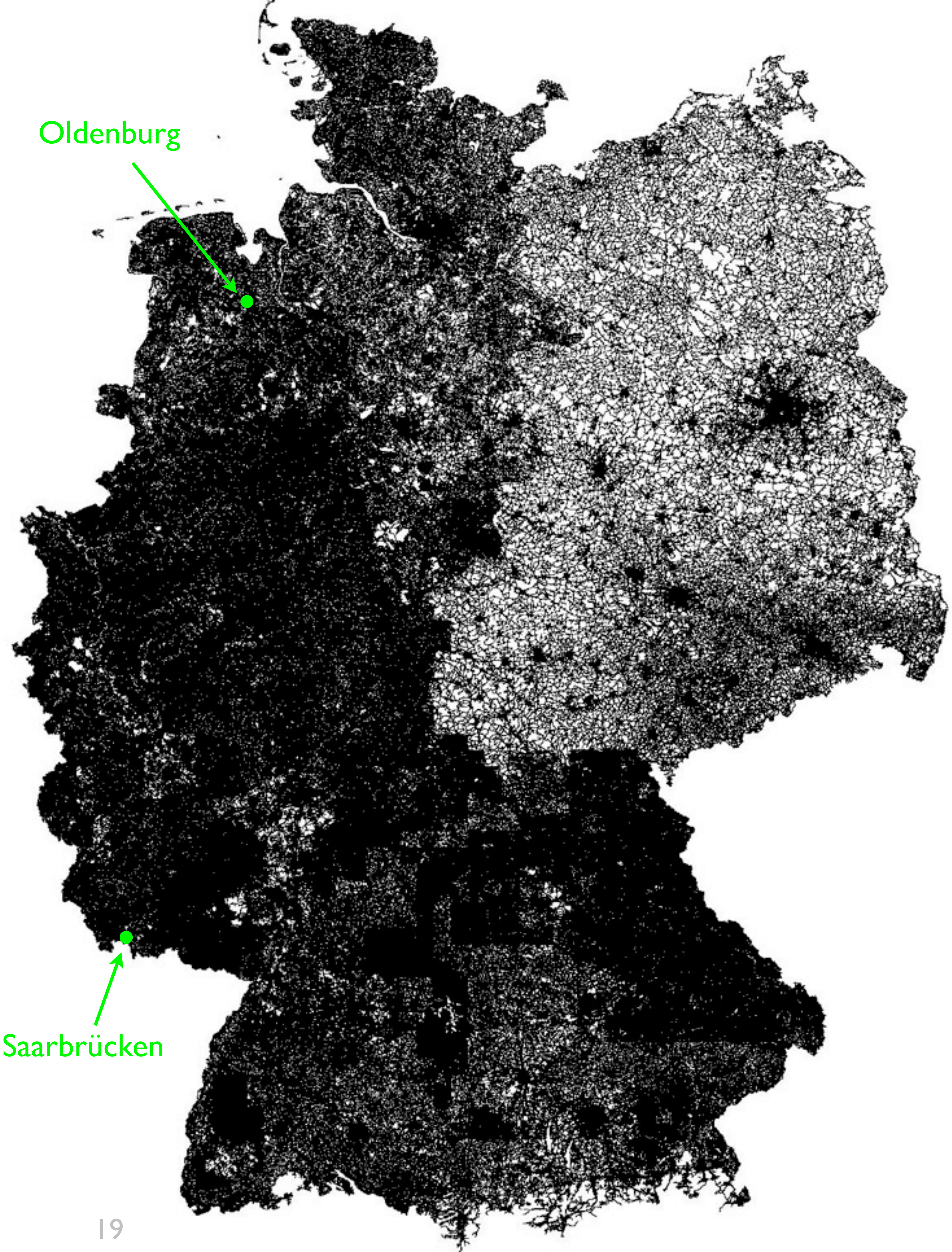
- largest road network ever used in experiments
- up to 100 million moving objects
- 6 nodes: each 2 $*$ Dual Core AMD Opteron at 2.4 GHz and 6GB main memory
  - 2 nodes used as data generators
  - up to 4 nodes used to index/query data

Data



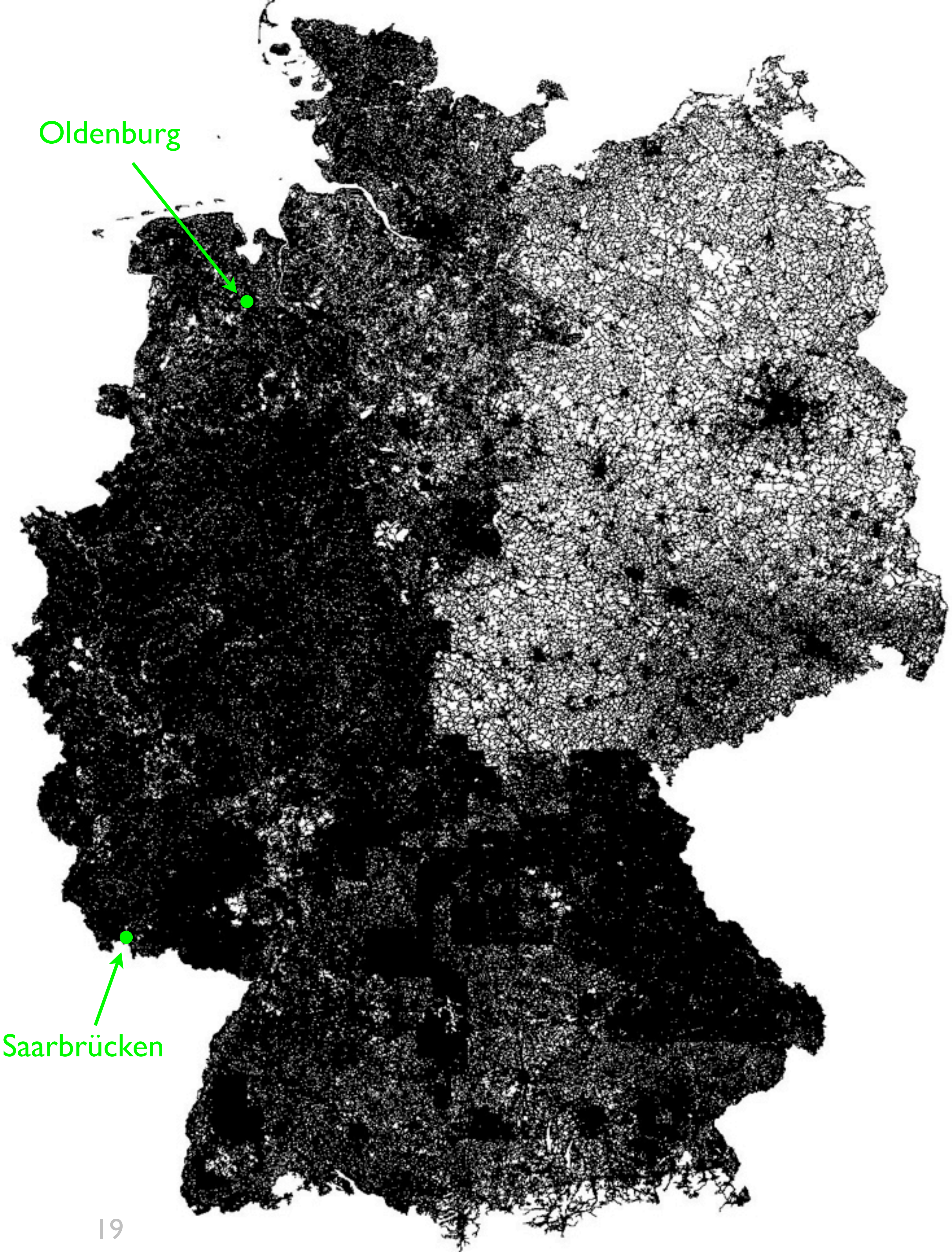
Data

- existing workload generators did not scale



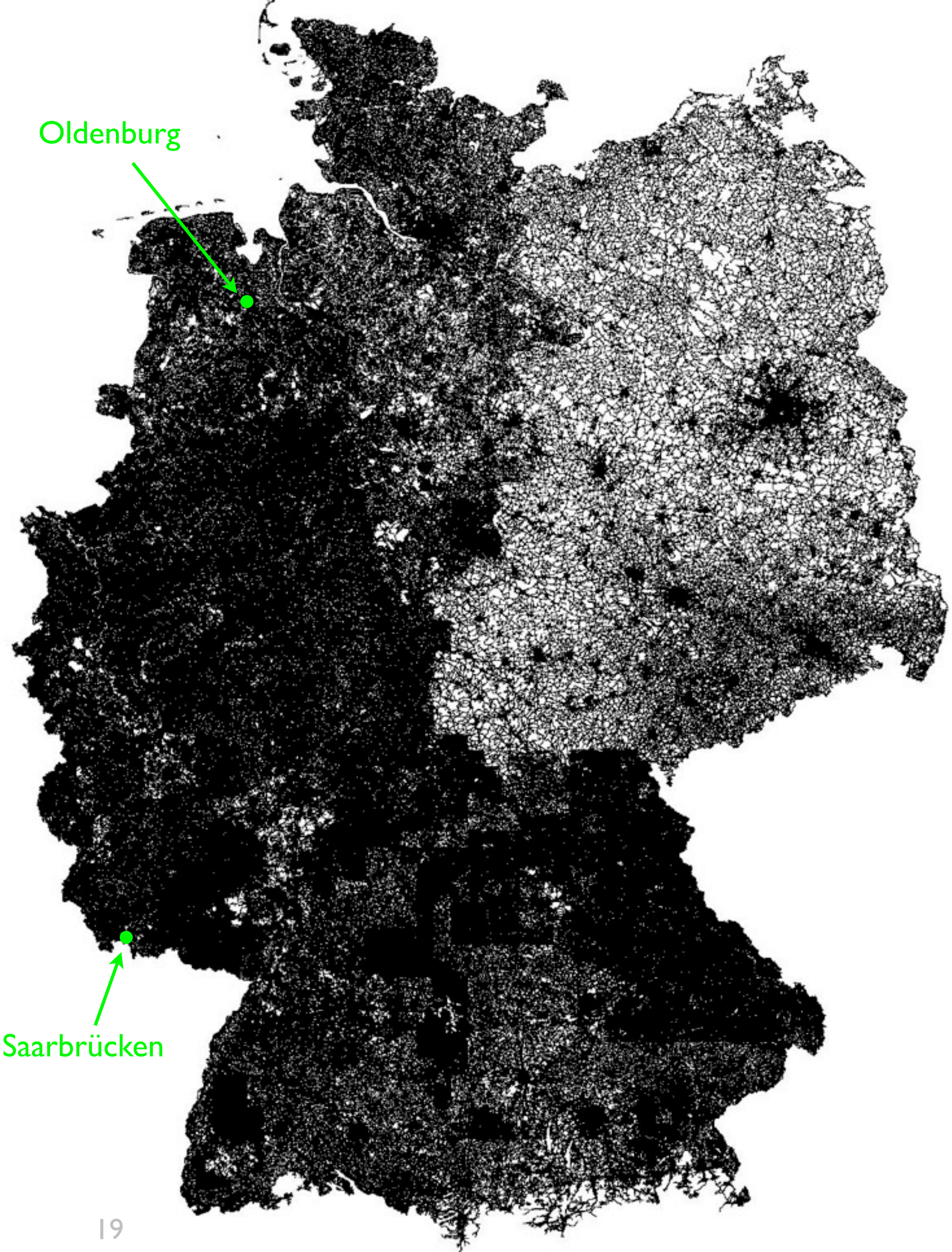
Data

- existing workload generators did not scale
- had to write our own: moto.sourceforge.net



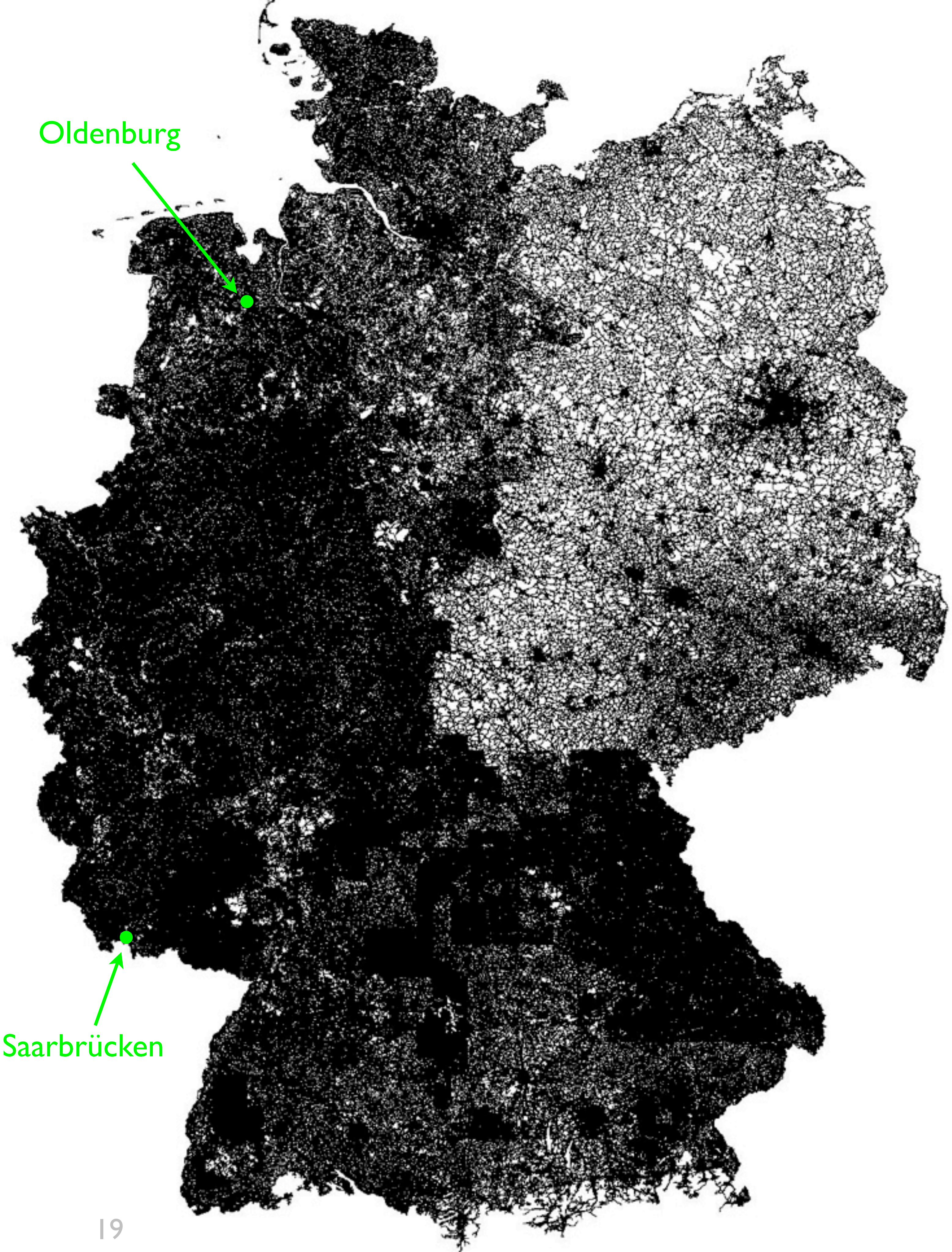
Data

- existing workload generators did not scale
- had to write our own: moto.sourceforge.net
- build on ideas from Brinkhoff generator



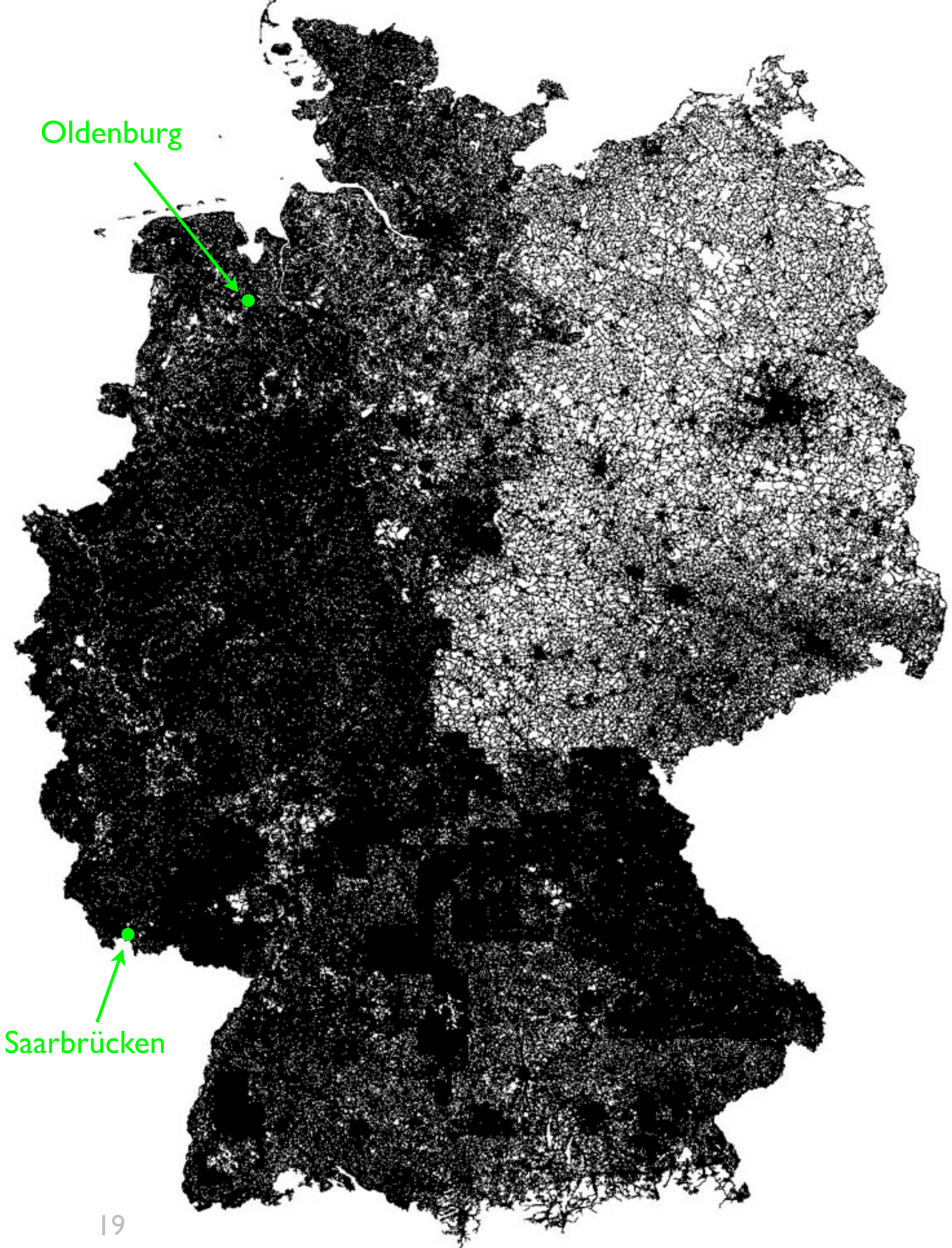
Data

- existing workload generators did not scale
- had to write our own: moto.sourceforge.net
- build on ideas from Brinkhoff generator
- 40 million nodes



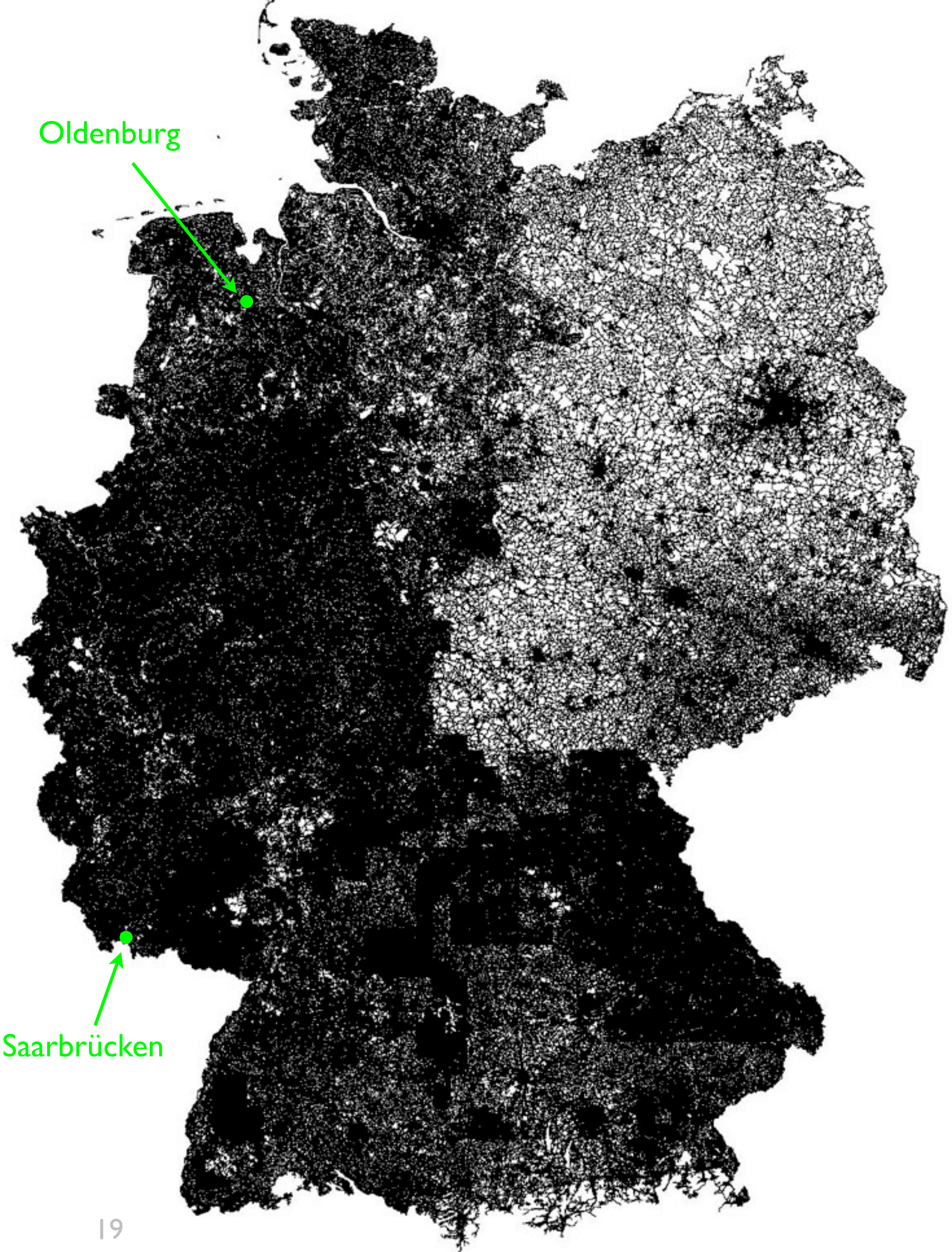
Data

- existing workload generators did not scale
- had to write our own: moto.sourceforge.net
- build on ideas from Brinkhoff generator
- 40 million nodes
- 40 million edges



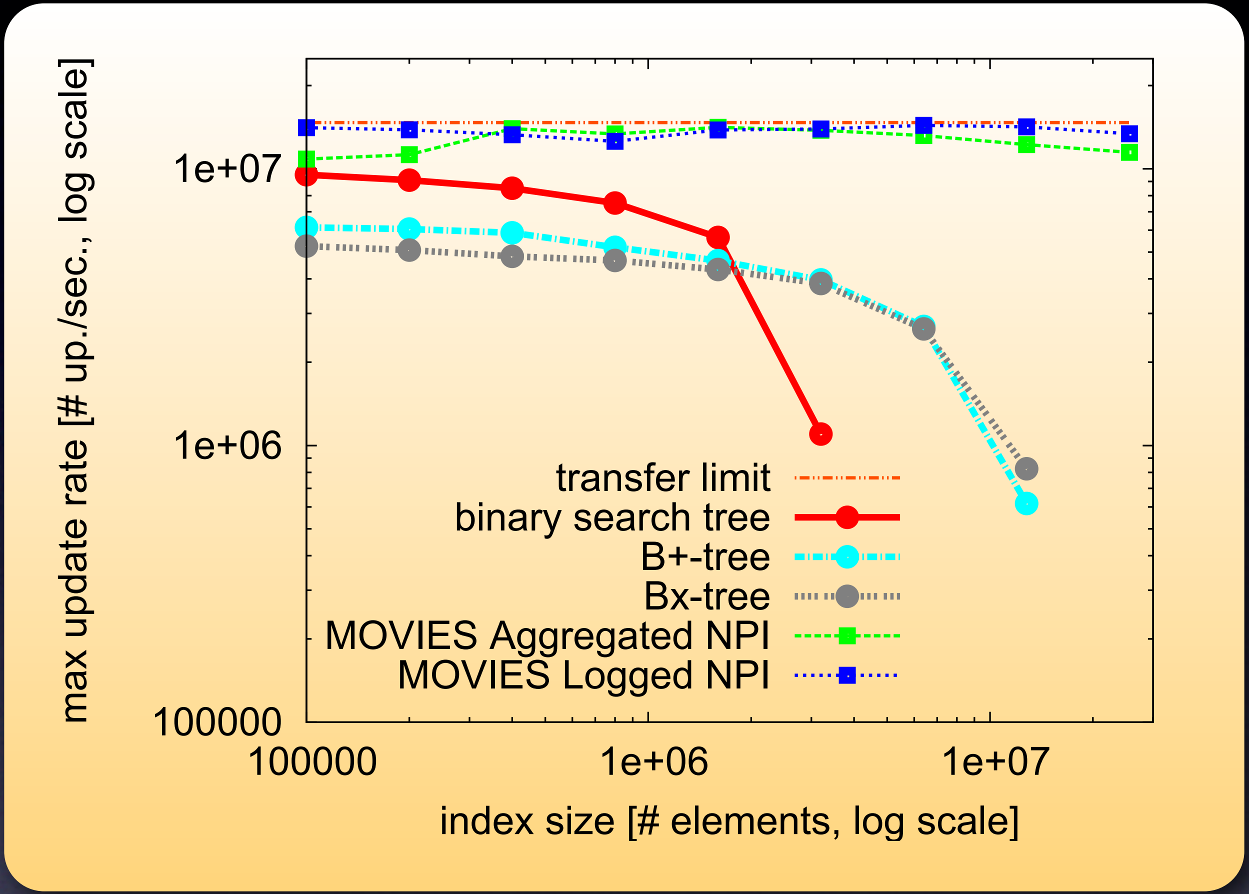
Data

- existing workload generators did not scale
- had to write our own: moto.sourceforge.net
- build on ideas from Brinkhoff generator
- 40 million nodes
- 40 million edges
- up to 100 million moving objects



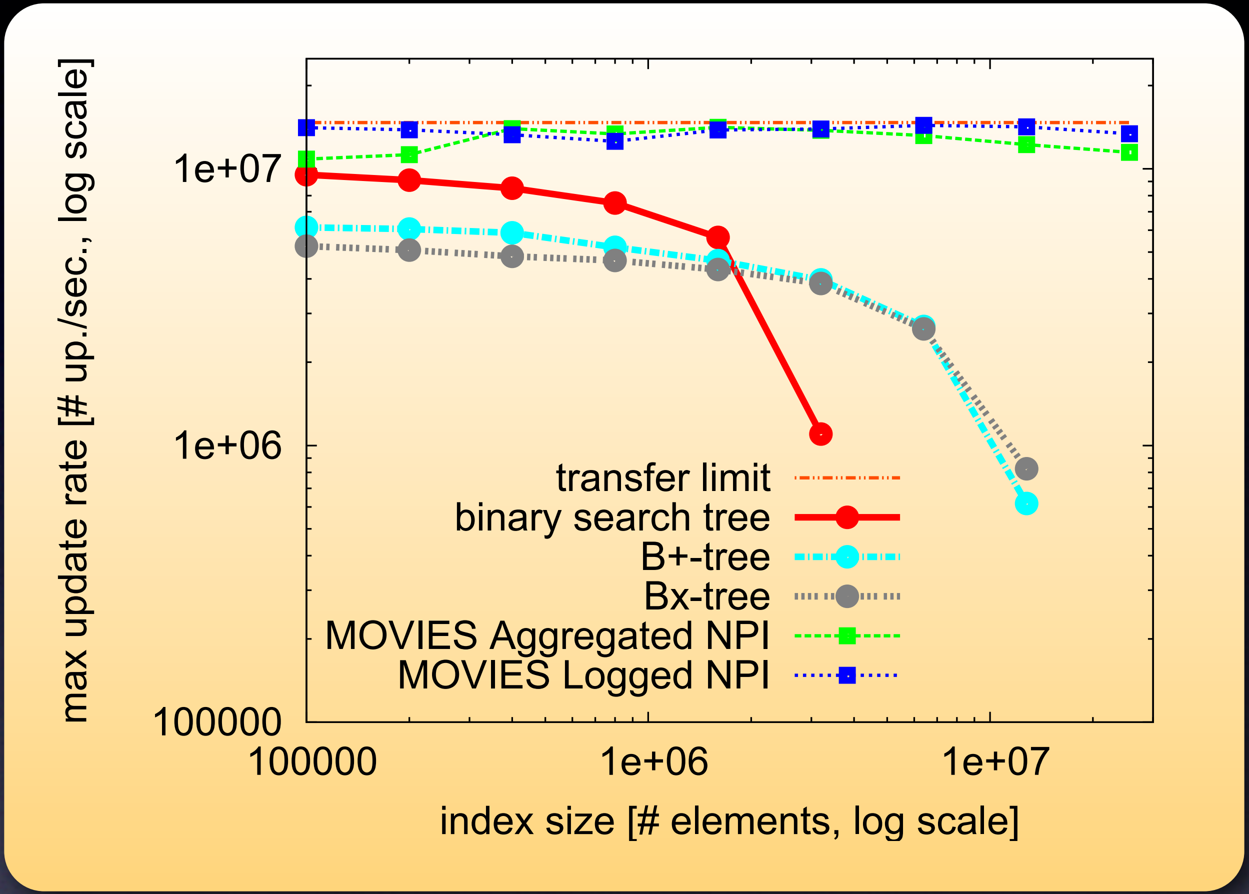
Scalability in Index Size

single node



Scalability in Index Size

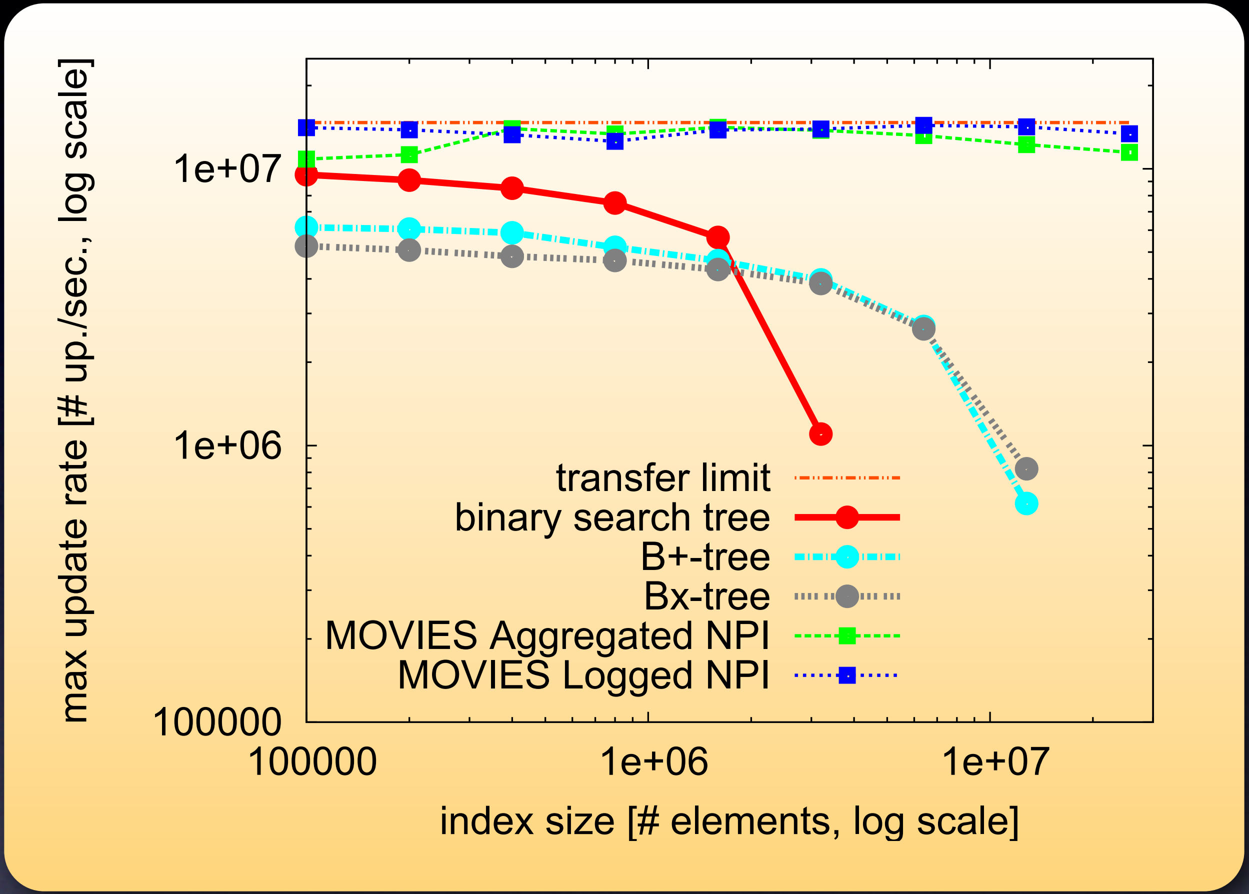
single node



- MOVIES outperforms B^x-tree by a factor >10

Scalability in Index Size

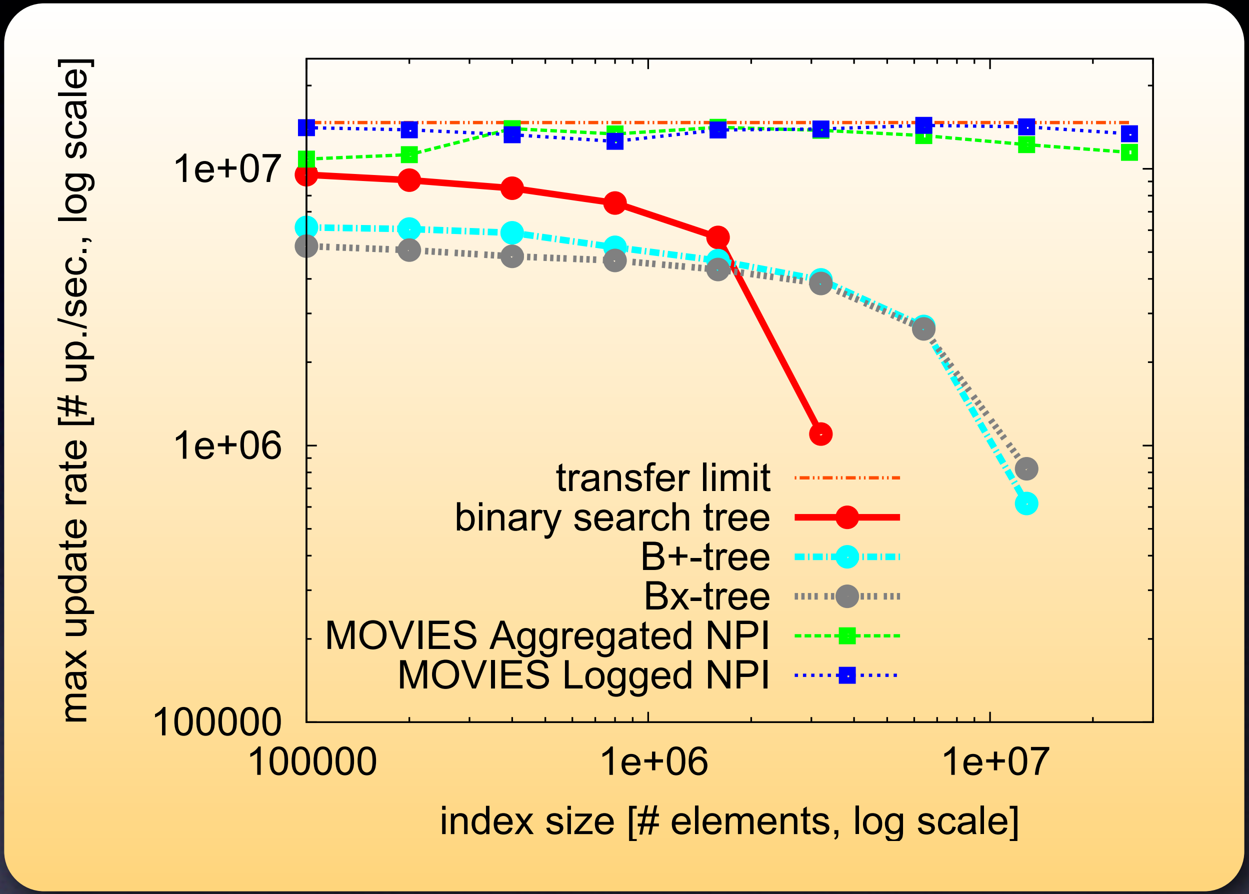
single node



- MOVIES outperforms B^x-tree by a factor >10
- BST B$^+$T, B^{x}T could not process largest dataset

Scalability in Index Size

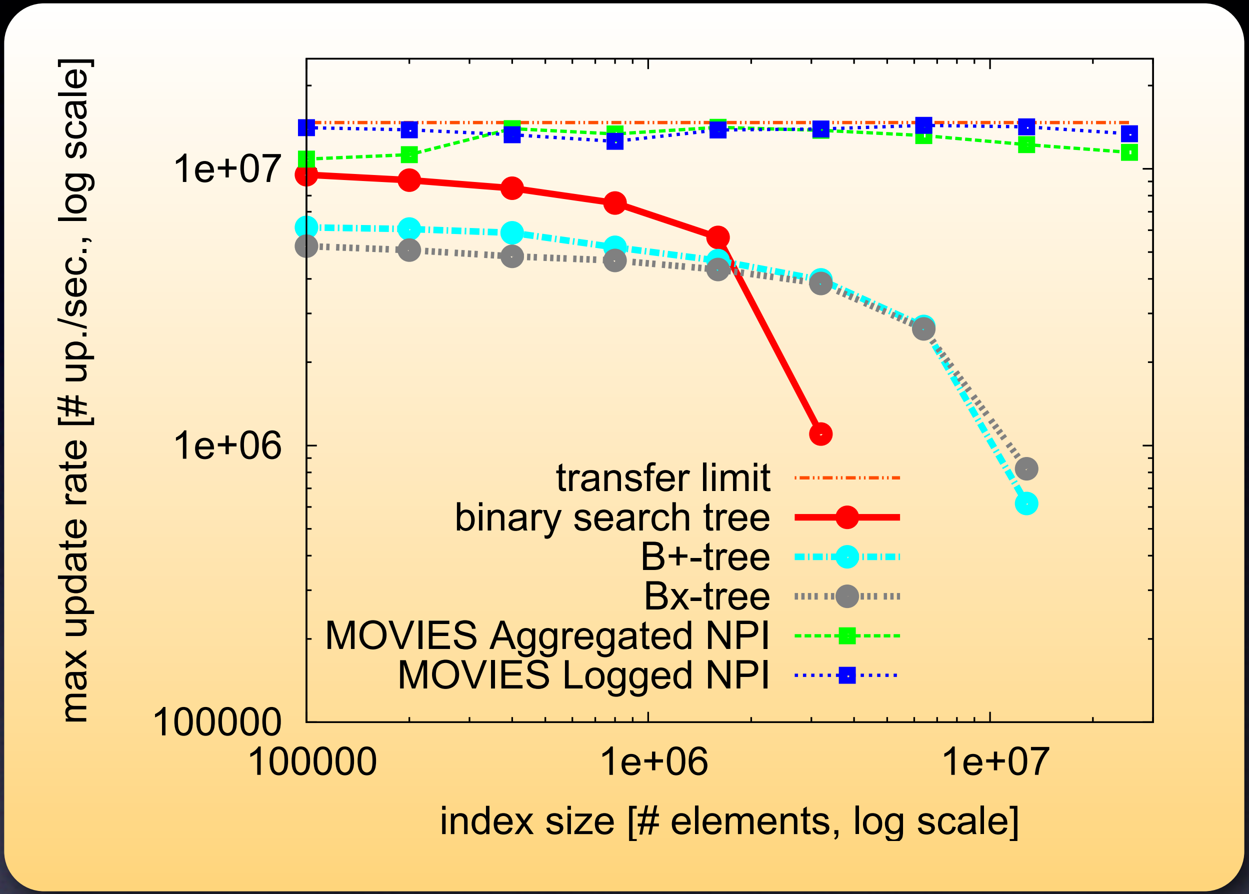
single node

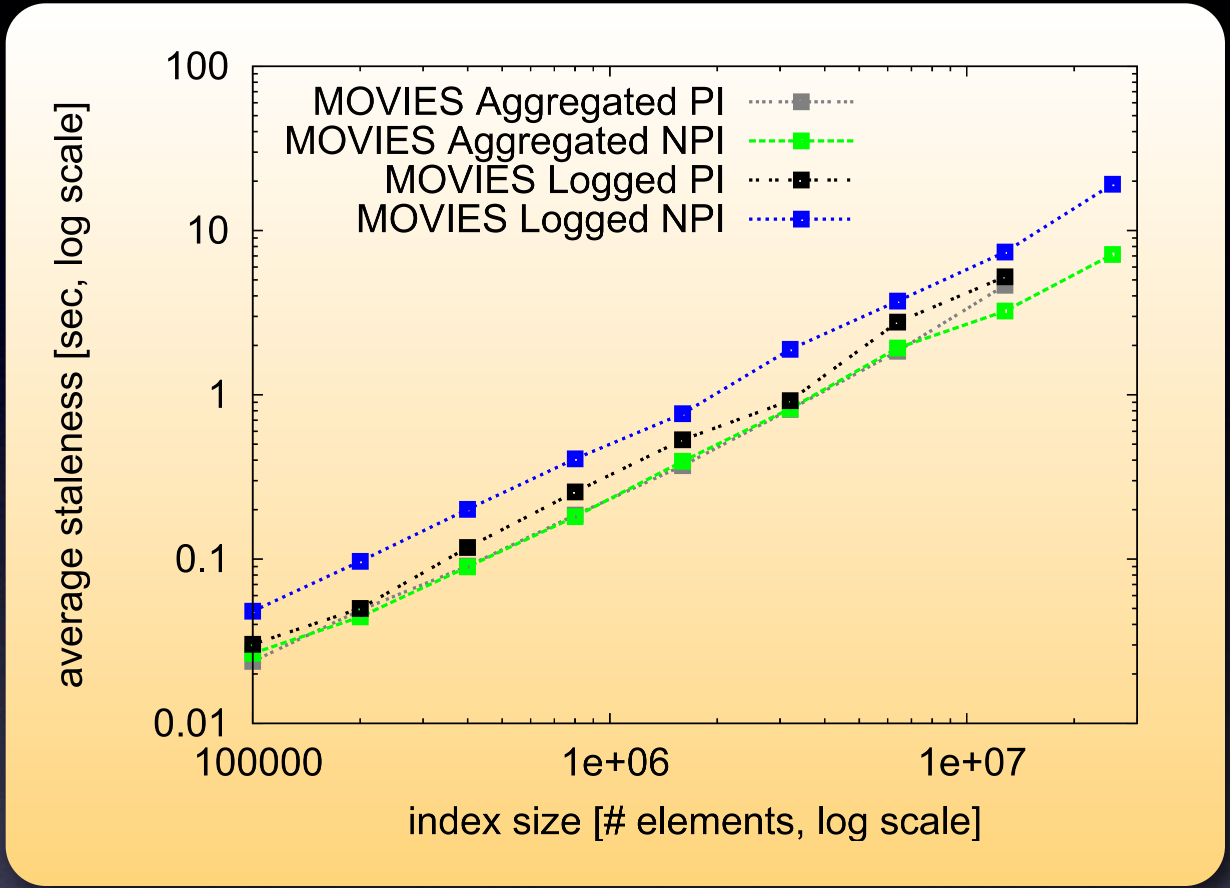


- MOVIES outperforms B^x-tree by a factor >10
- BST B$^+$T, B^{x}T could not process largest dataset
- MOVIES hits network bandwidth

Scalability in Index Size

single node

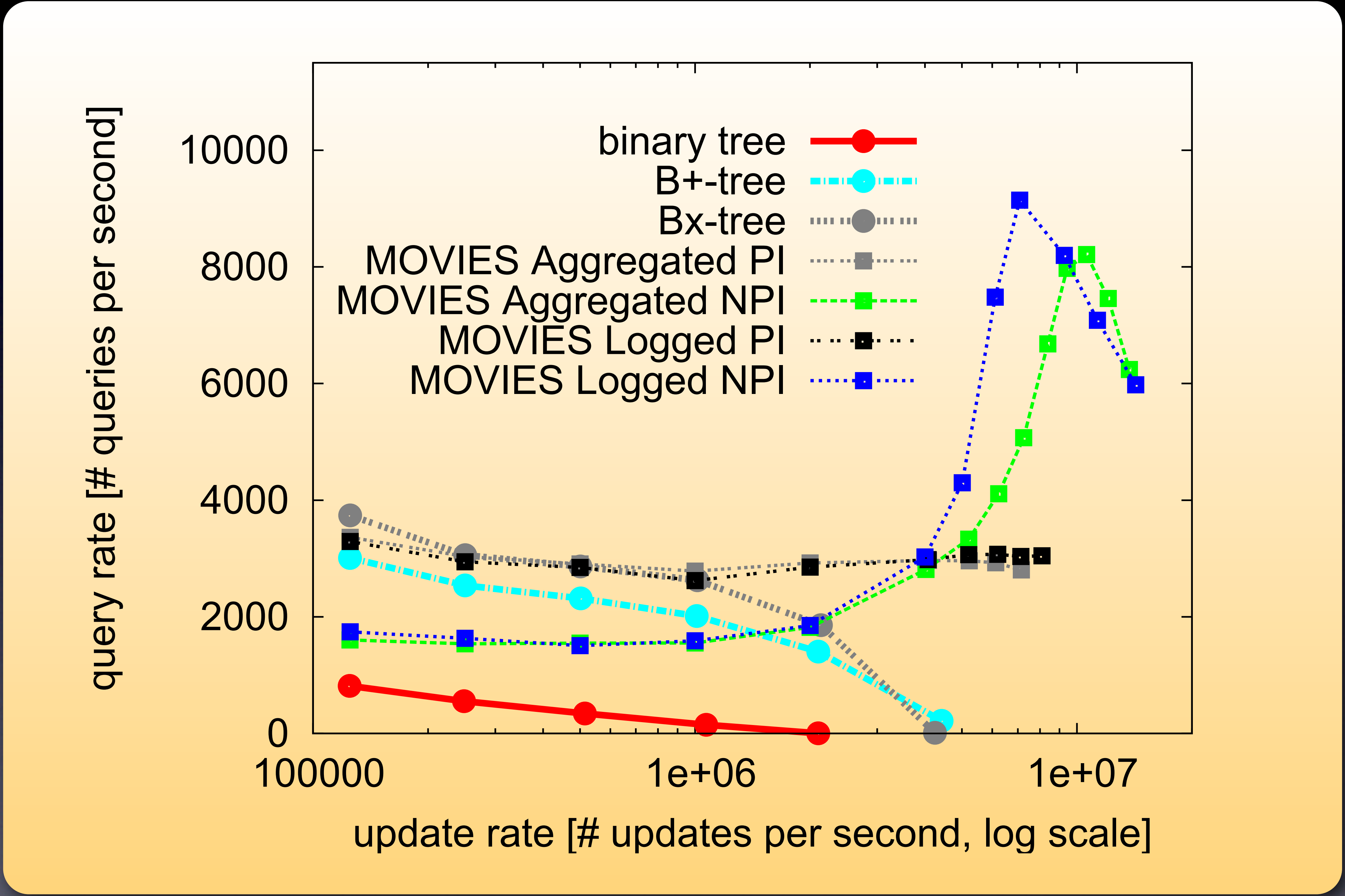




- MOVIES outperforms B^x-tree by a factor >10
- BST B^+T, B^xT could not process largest dataset
- MOVIES hits network bandwidth

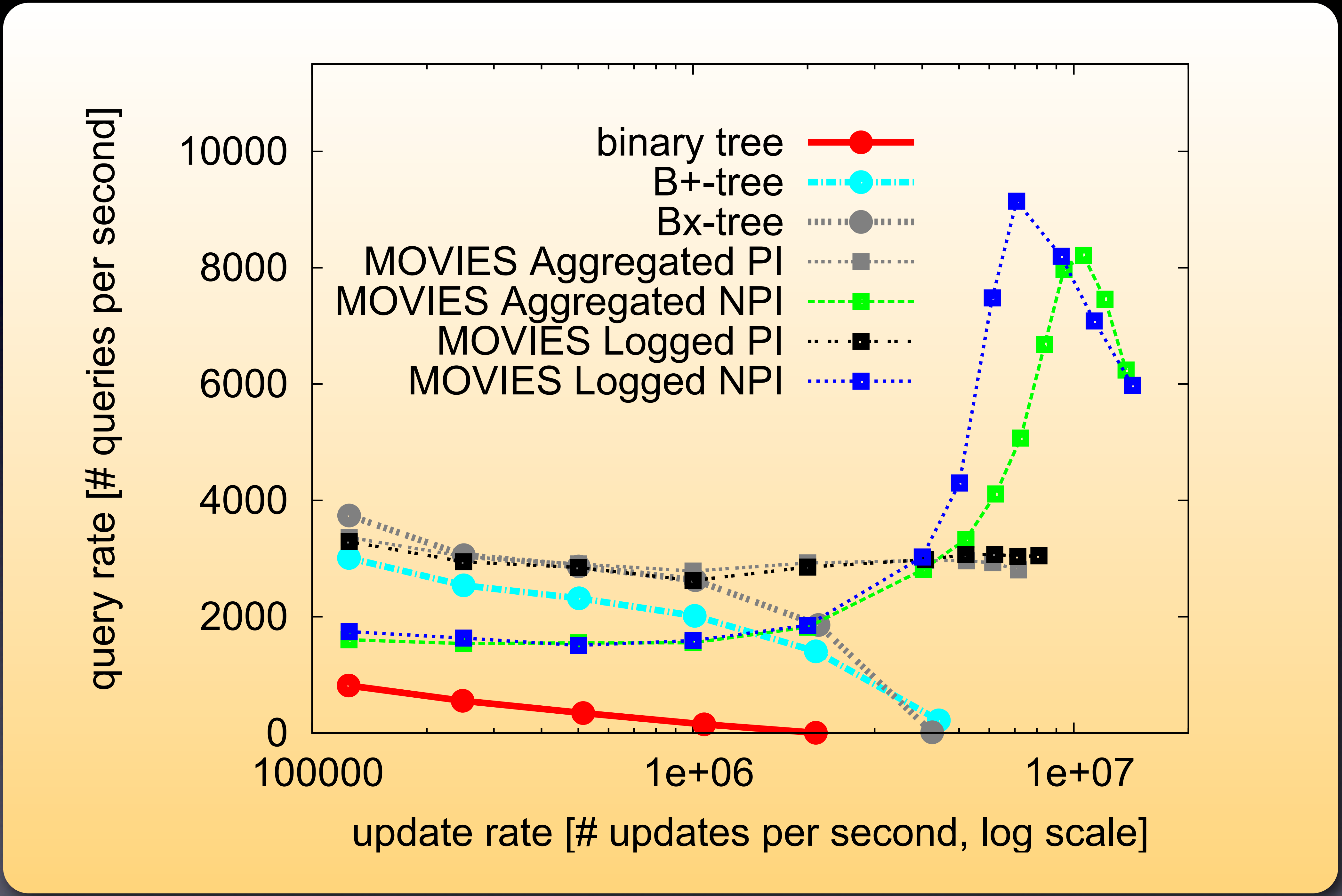
Scalability in Update Rate

single node



Scalability in Update Rate

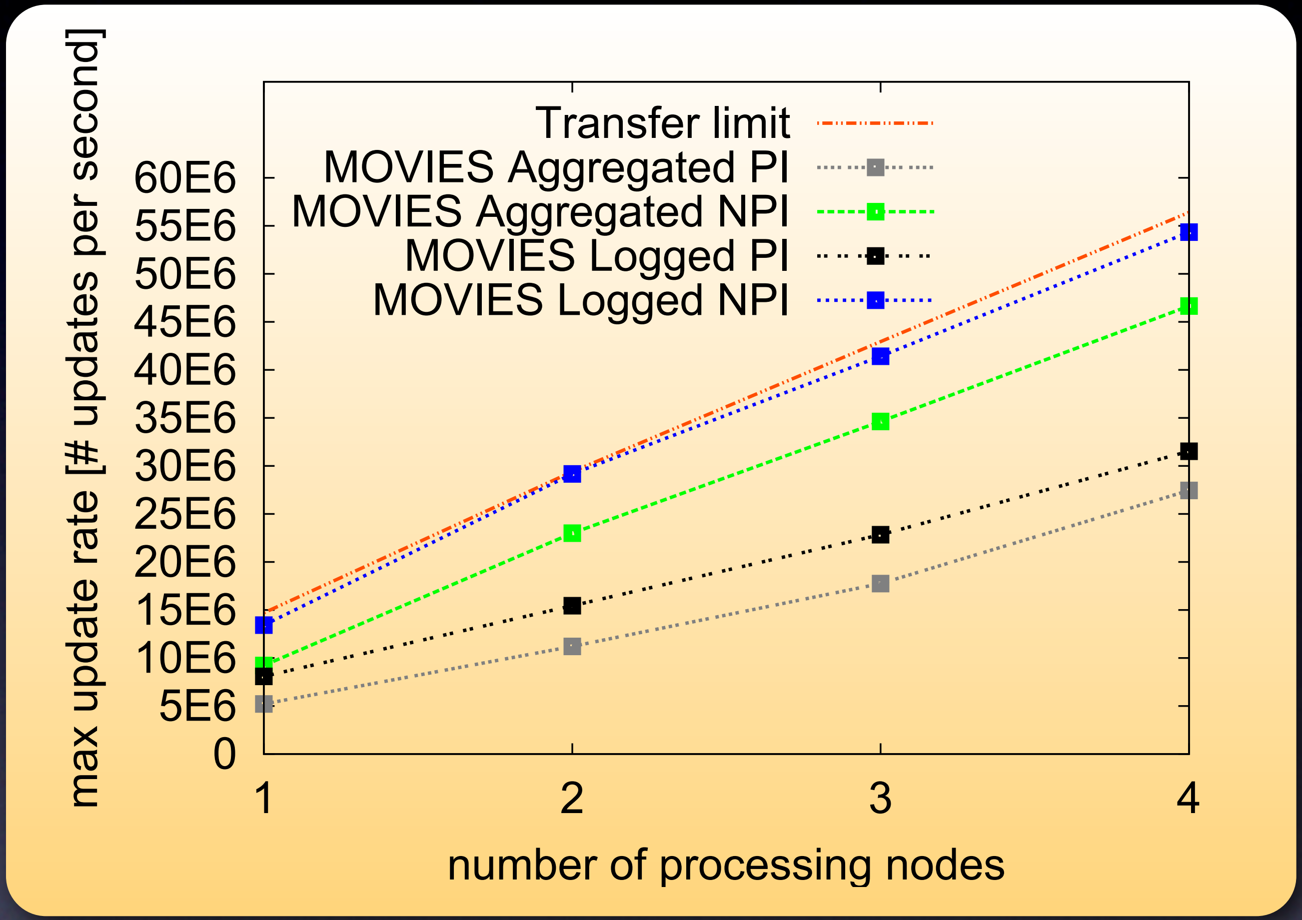
single node



- NPI MOVIES better than PI MOVIES (high up.rate)

Shared-Nothing Scale-Out

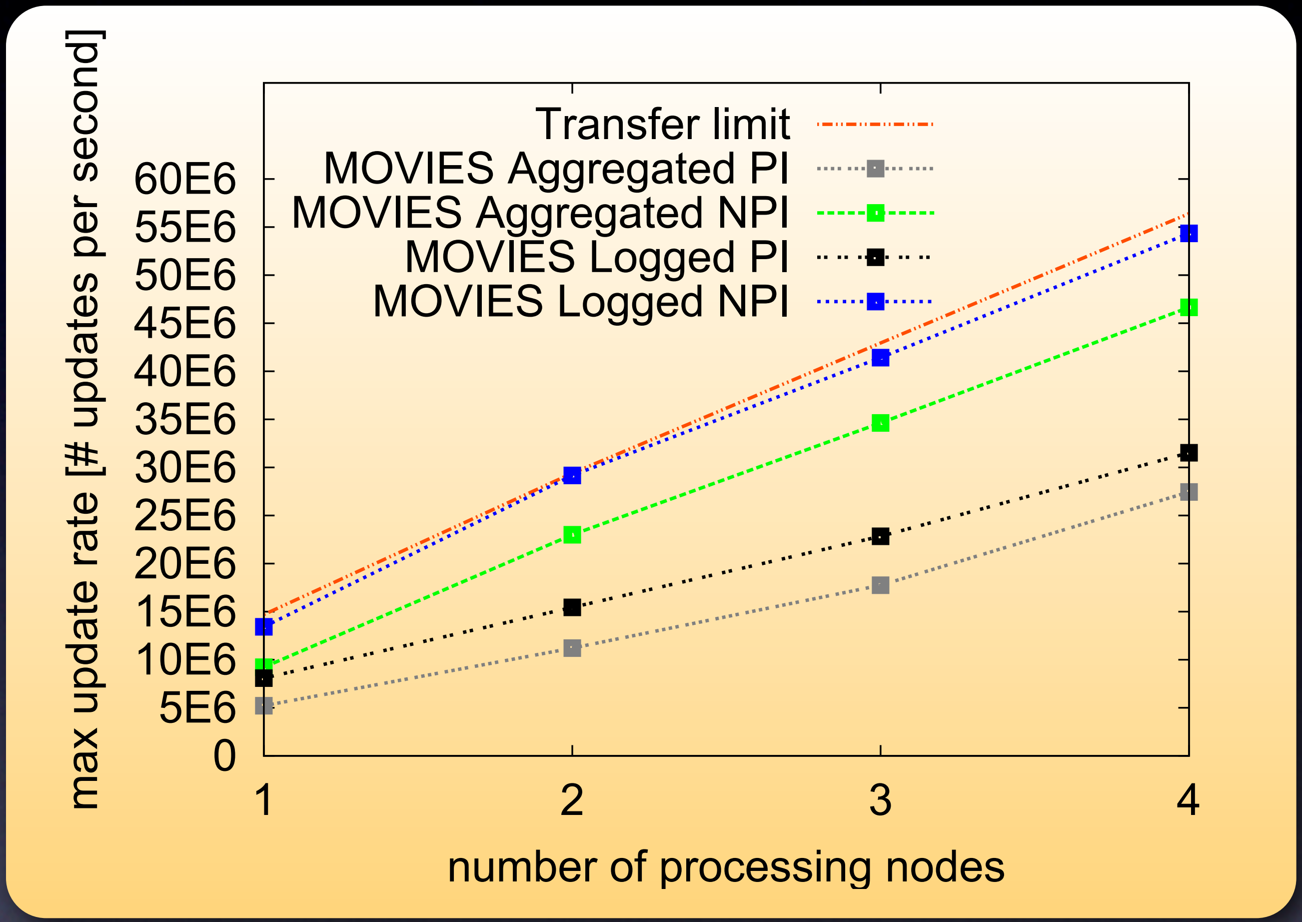
multiple nodes

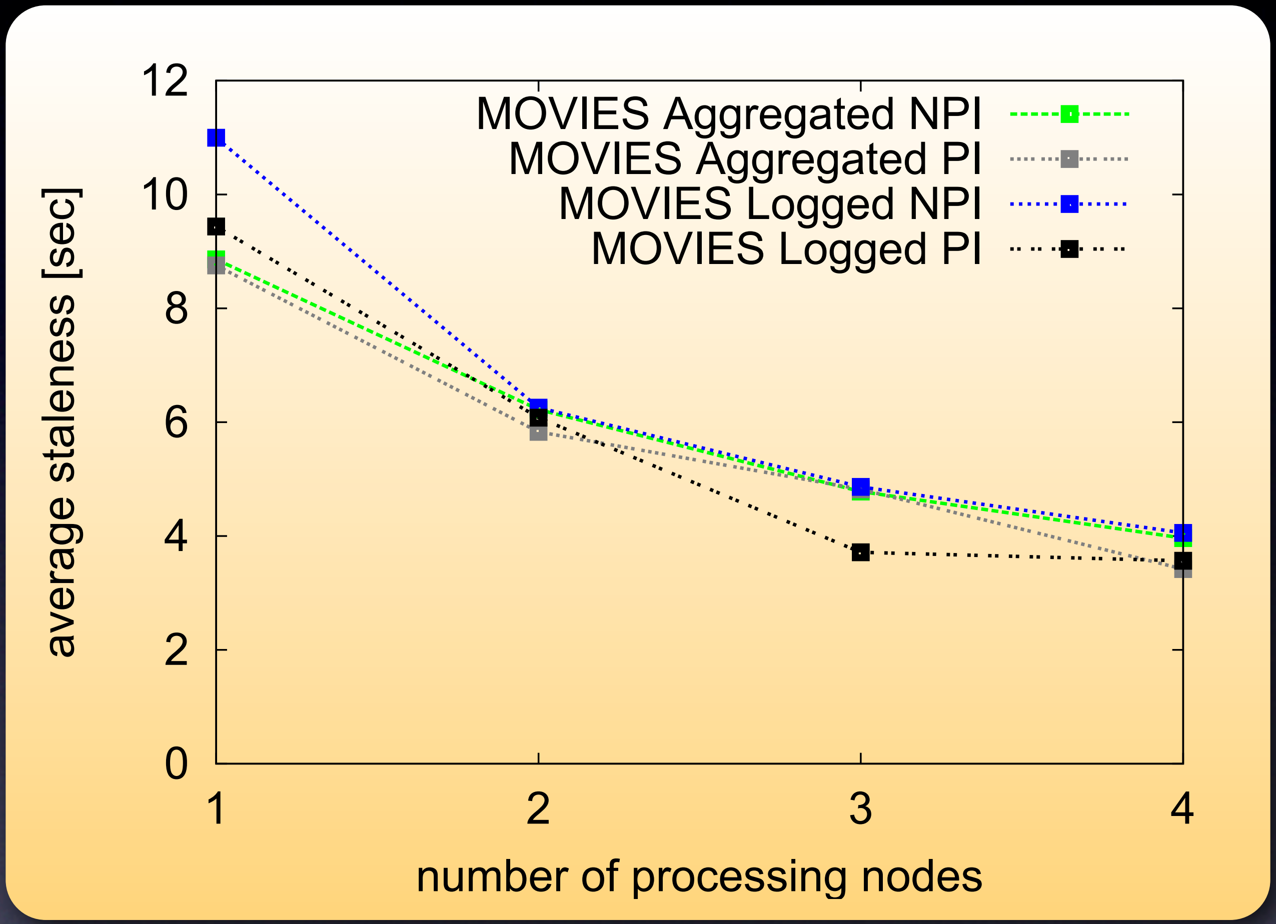


- N=25.8M
- special network setup for shared-nothing
- up to 2Gb/s bandwidth node2node

Shared-Nothing Scale-Out

multiple nodes

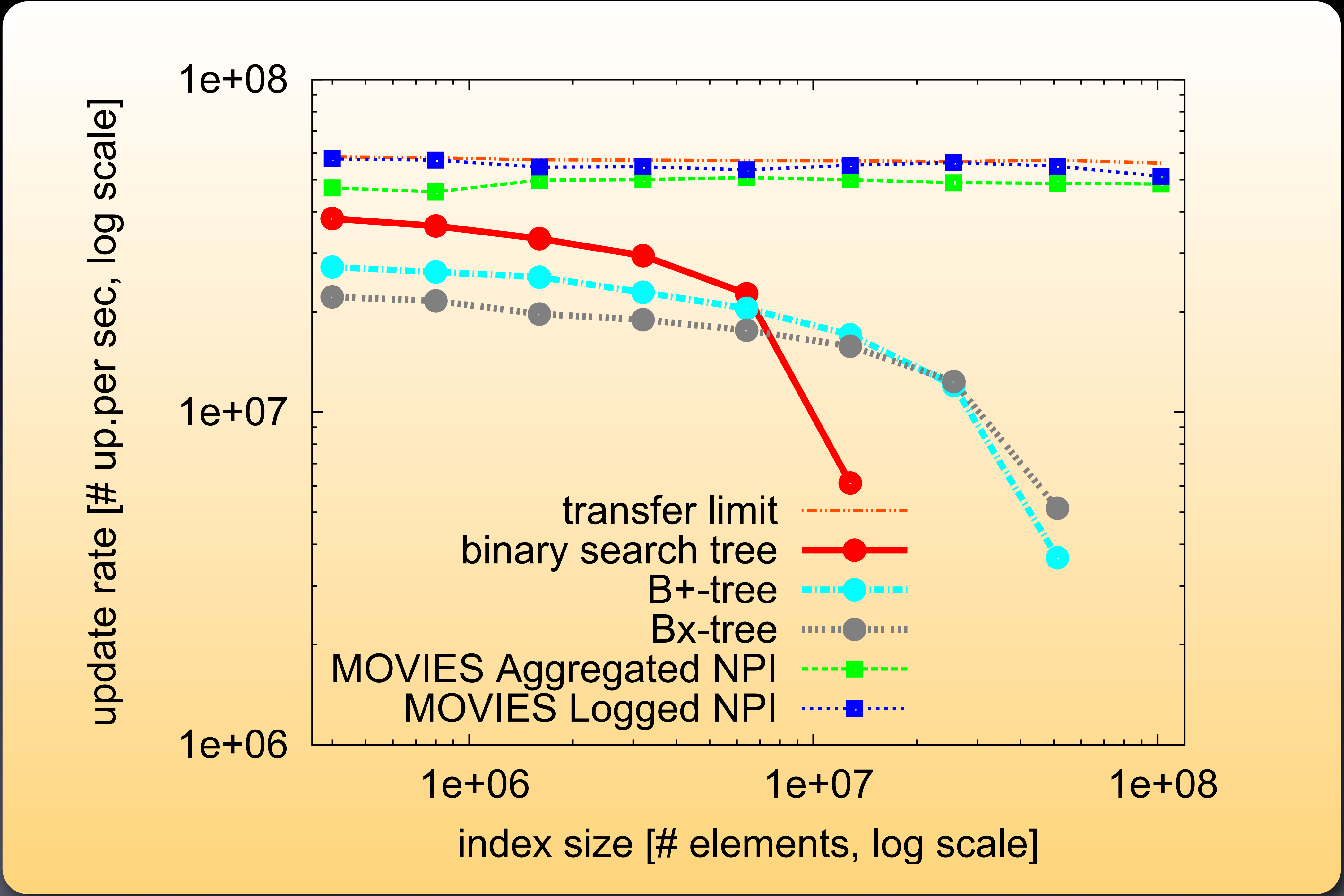




- N=25.8M
- special network setup for shared-nothing
- up to 2Gb/s bandwidth node2node

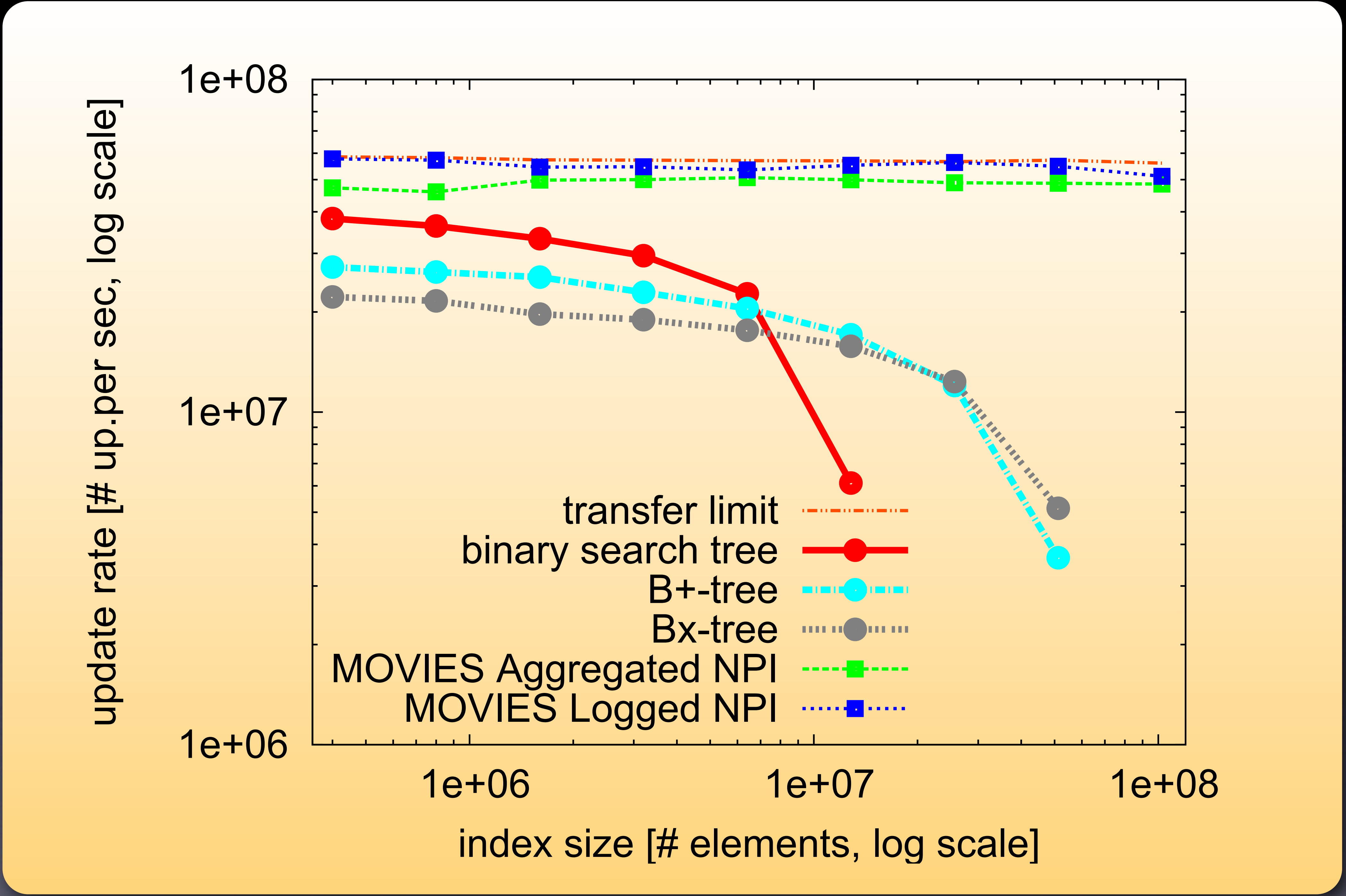
Scalability in Index Size

4 nodes



Scalability in Index Size

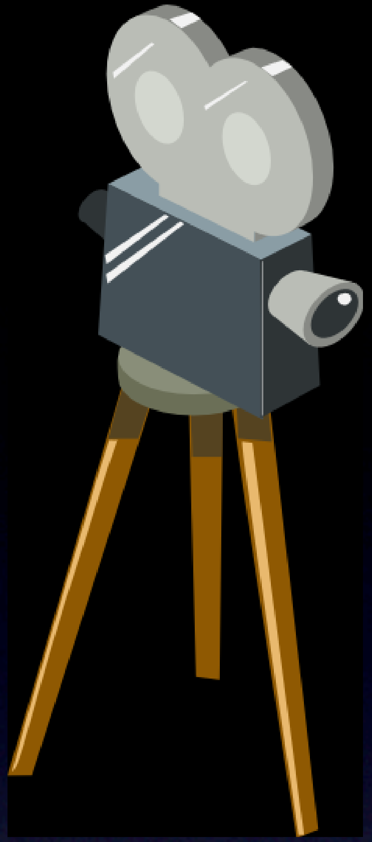
4 nodes



- up to **55 million updates per second!**

SAARLAND UNIVERSITY COMPUTER SCIENCE

Conclusions



Conclusions

- we question two assumptions done in almost all previous work
  1. „data does not fit into main memory“
  2. „maintain index for incoming updates“

Conclusions

- we question two assumptions done in almost all previous work
    1. „data does not fit into main memory“
    2. „maintain index for incoming updates“
- MOVIES builds a series of read-optimized main memory indexes

Conclusions

- we question two assumptions done in almost all previous work
    1. „data does not fit into main memory“
    2. „maintain index for incoming updates“
- MOVIES builds a series of read-optimized main memory indexes
- movie camera analogy

Conclusions

- we question two assumptions done in almost all previous work
    1. „data does not fit into main memory“
    2. „maintain index for incoming updates“
- MOVIES builds a series of read-optimized main memory indexes
- movie camera analogy
- also similarities to data warehousing

Conclusions

- we question two assumptions done in almost all previous work
  1. „data does not fit into main memory“
  2. „maintain index for incoming updates“
- MOVIES builds a series of read-optimized main memory indexes
- movie camera analogy
- also similarities to data warehousing
- but: create warehouse several times per second to minimize staleness

Conclusions

- we question two assumptions done in almost all previous work
    1. „data does not fit into main memory“
    2. „maintain index for incoming updates“
- MOVIES builds a series of read-optimized main memory indexes
- movie camera analogy
- also similarities to data warehousing
- but: create warehouse several times per second to minimize staleness
- simple yet very efficient

Conclusions

- we question two assumptions done in almost all previous work
    1. „data does not fit into main memory“
    2. „maintain index for incoming updates“
- MOVIES builds a series of read-optimized main memory indexes
- movie camera analogy
- also similarities to data warehousing
- but: create warehouse several times per second to minimize staleness
- simple yet very efficient
- outperforms existing techniques by orders of magnitude

Future Work

- investigate effects of staleness on quality
- other read-optimized indexes
- use cache-optimized indexes
- different merge strategies
- adaptive merge strategies based on workload
- MOVIES on flash
- application to general data streams

Thanks!