Cooperative Scans:
Dynamic Bandwidth Sharing in a DBMS

Marcin Zukowski

Sandor Heman, Niels Nes, Peter Boncz

CWI, Amsterdam

VLDB 2007

Outline

- Scans in a DBMS
- Cooperative Scans
- Benchmarks
- DSM version

Scans in a DBMS

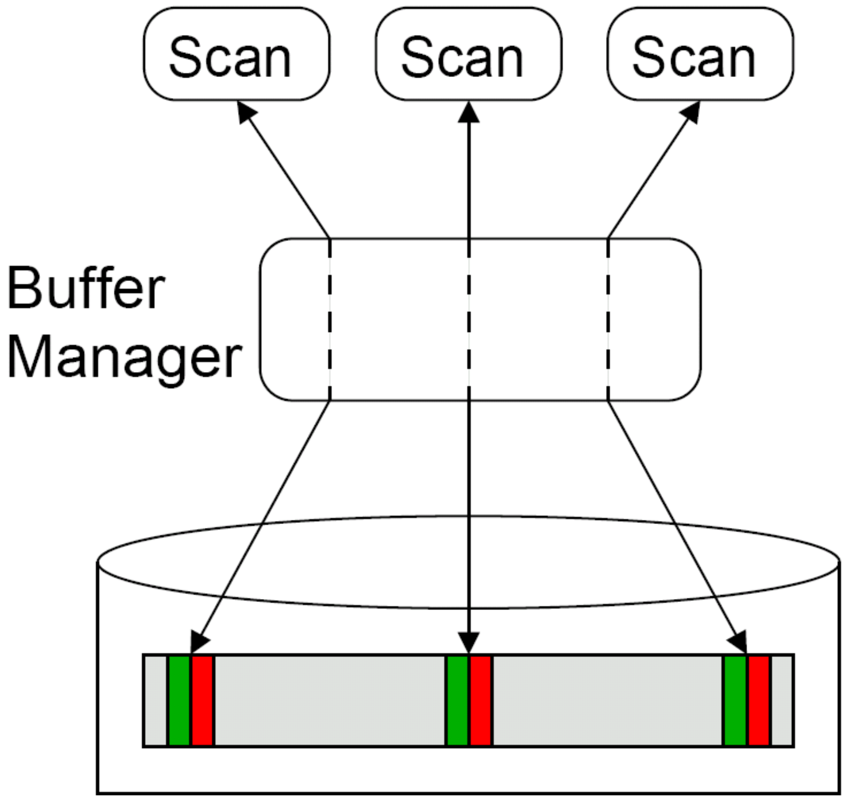
- Scan-based processing:
  - Large queries
  - Clustered indices
  - No useful indices
- Types of scans:
  - Full-table scans
  - Range-scans
  - Multi-range scans

Single scan optimizations

- Compression
  - Reduce data volume
- Column storage
  - Don’t read unnecessary columns
- Per-block statistics
  - Don’t read unnecessary ranges of rows

Concurrent scans

- Multiple queries scanning the same table
  - Different start times
  - Different scan ranges
- Compete for disk access and buffer space
- FCFS request scheduling: poor latency

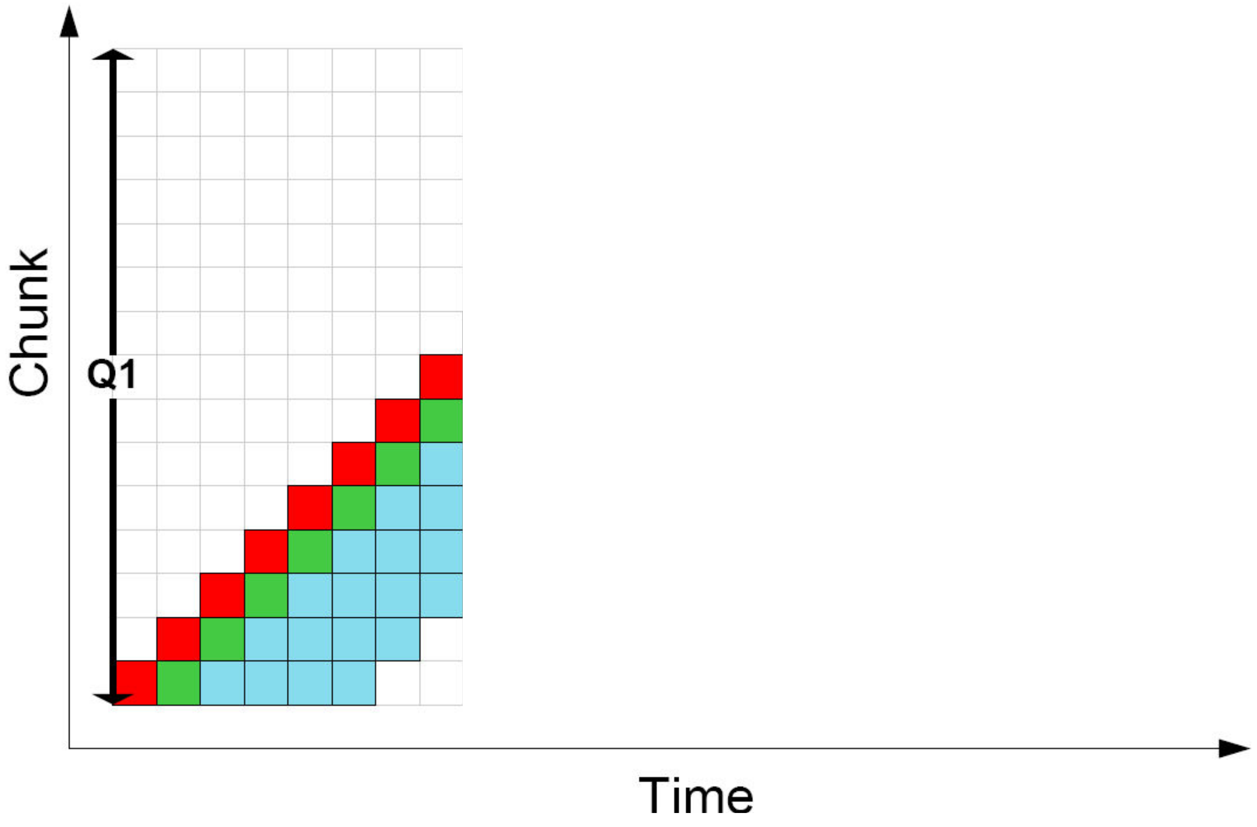


Chunks

- Pages clustered on disk
- Large I/O units
- Amortizes random-seek with large-reads
- Result: “random” system bandwidth close to sequential

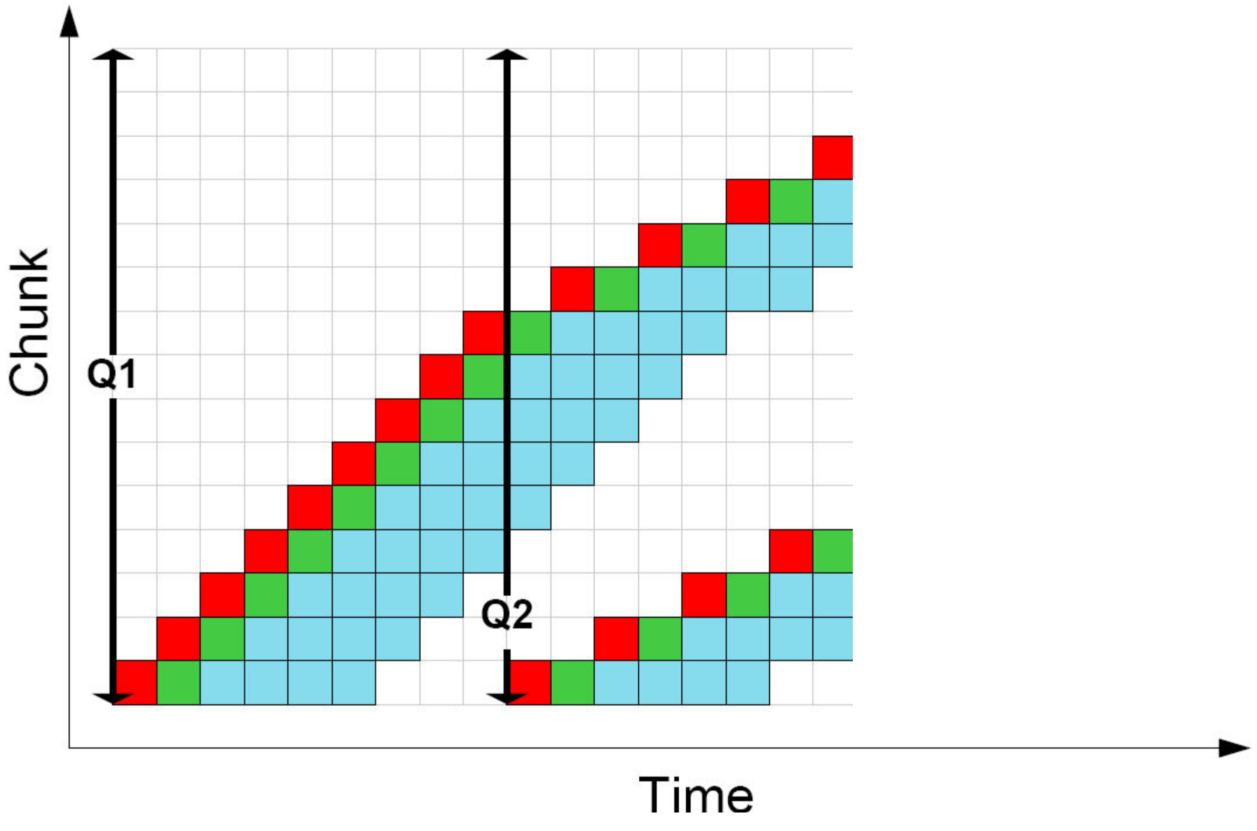
"Normal" policy

- Strictly sequential read order



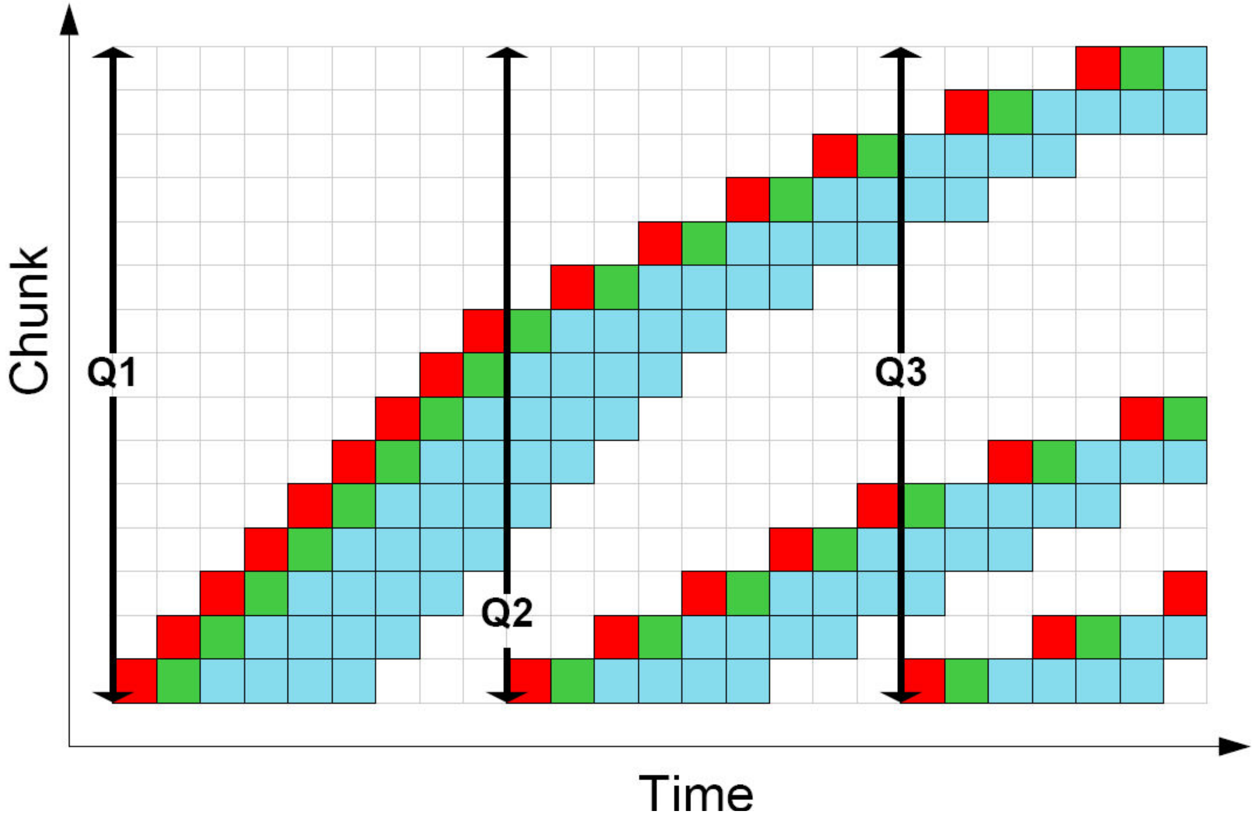
“Normal” policy

- Strictly sequential read order



“Normal” policy

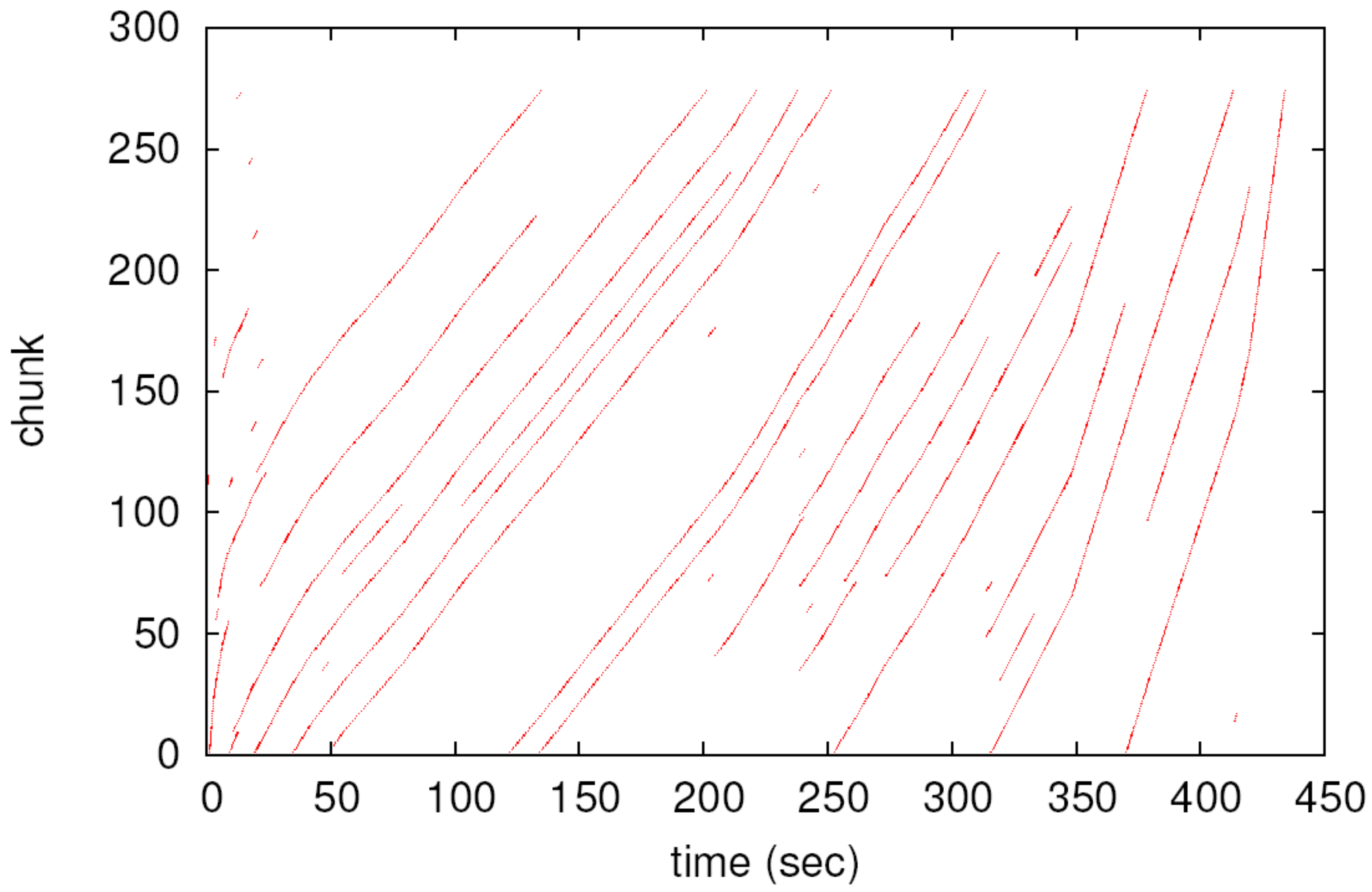
- Strictly sequential read order



“Normal” performance

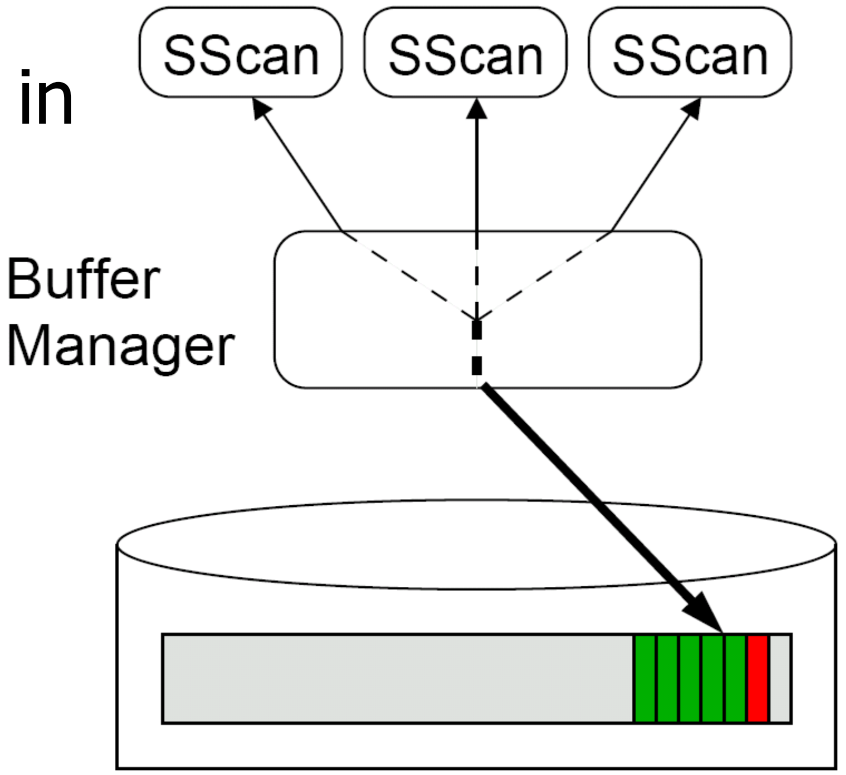
- Data read many times – poor sharing
- Many scans fight for bandwidth
- Both latency and throughput bad

“Normal” in real life



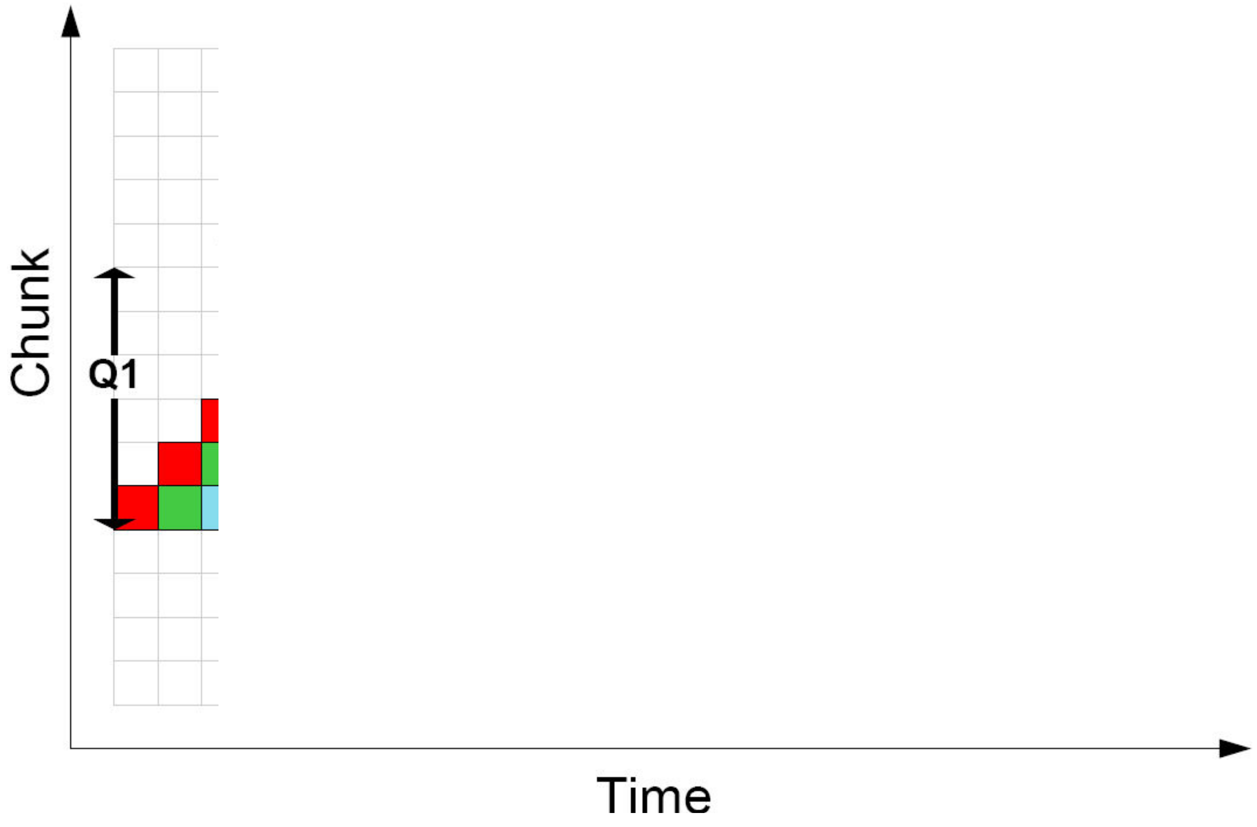
Shared scans

- Observation: queries often do not need data in a sequential order
- Idea: make queries "share" the scanning process
- Two existing types:
  - Attach
  - Elevator



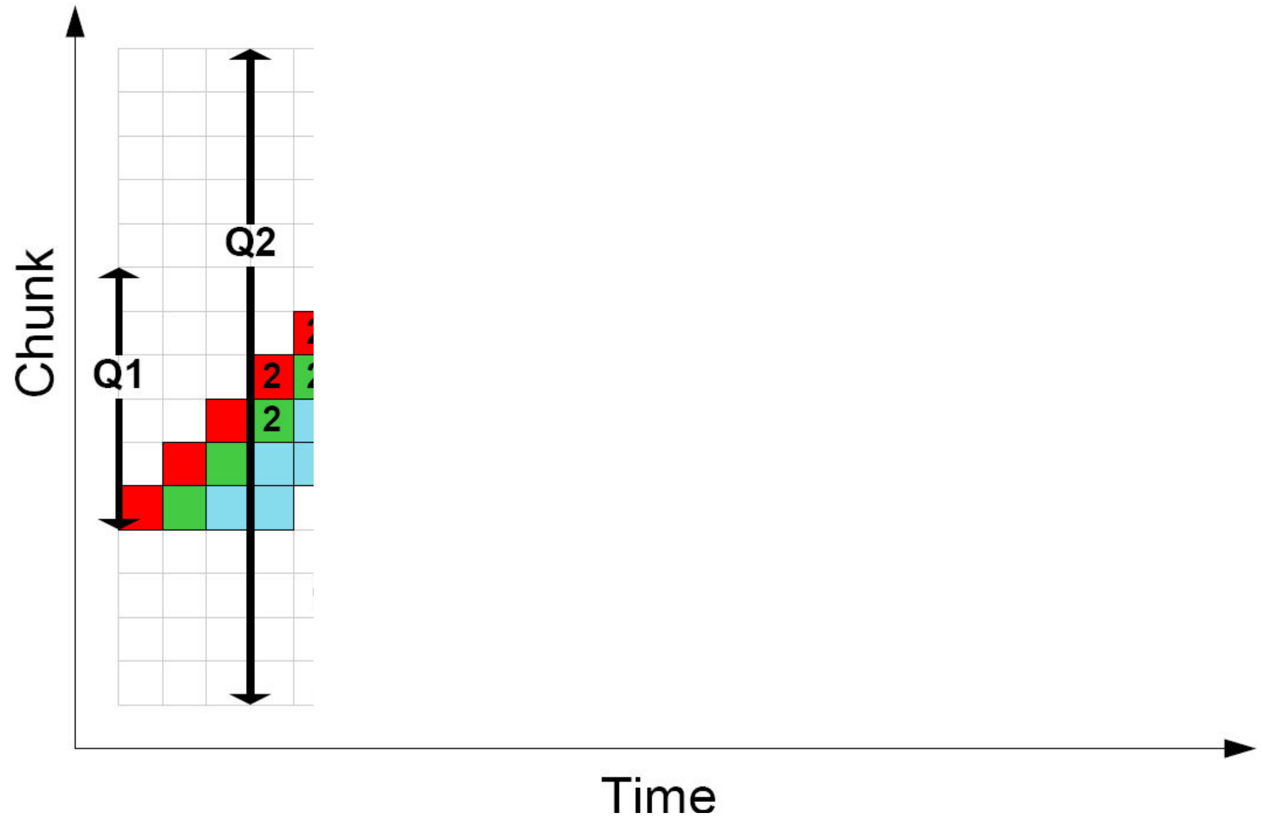
"Attach" policy

- Attach to a running query with data overlap



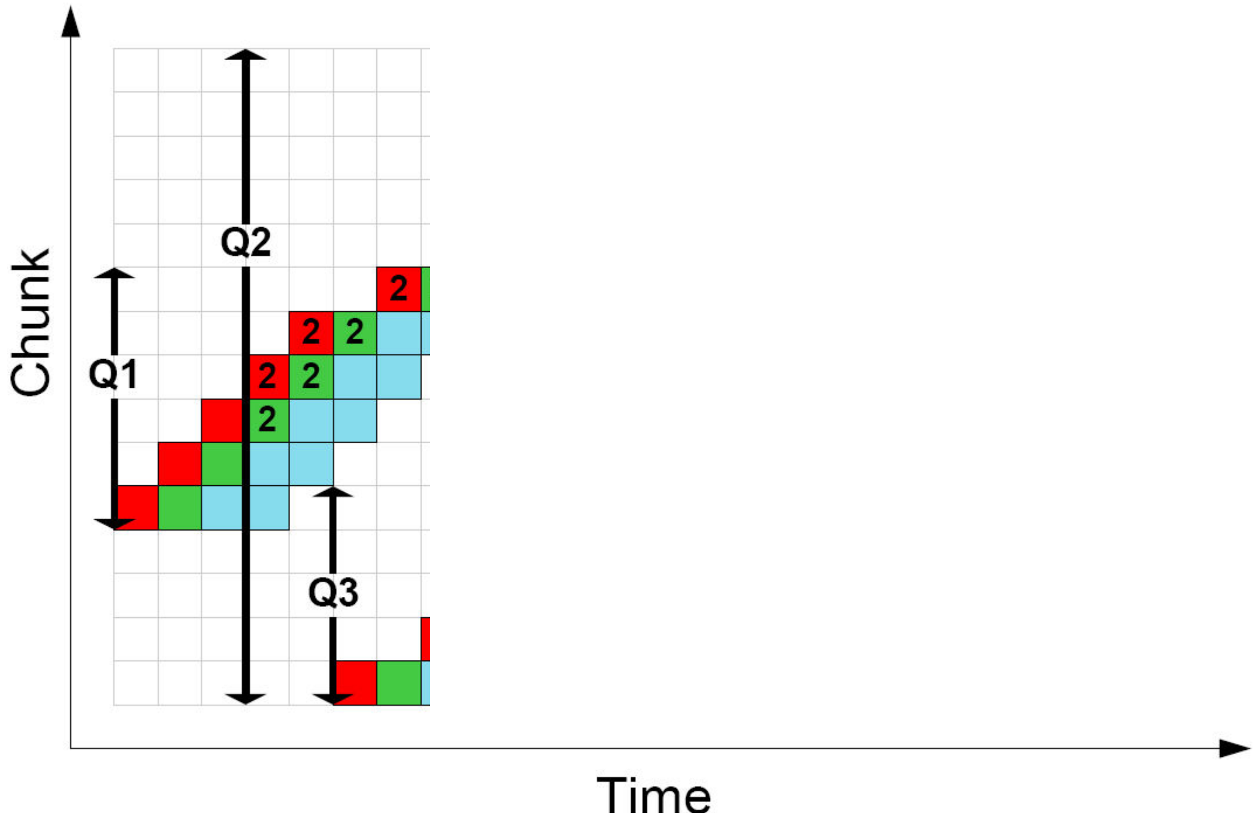
"Attach" policy

- Attach to a running query with data overlap



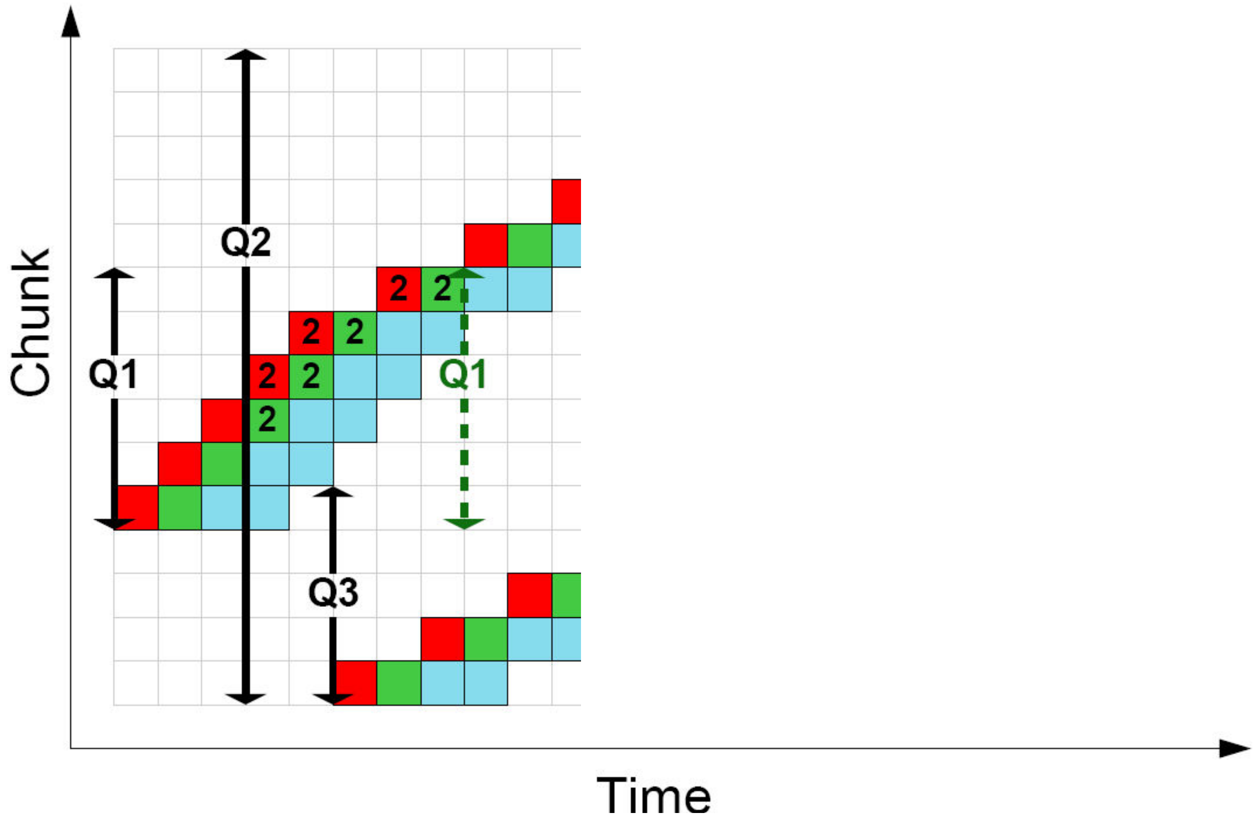
"Attach" policy

- Attach to a running query with data overlap



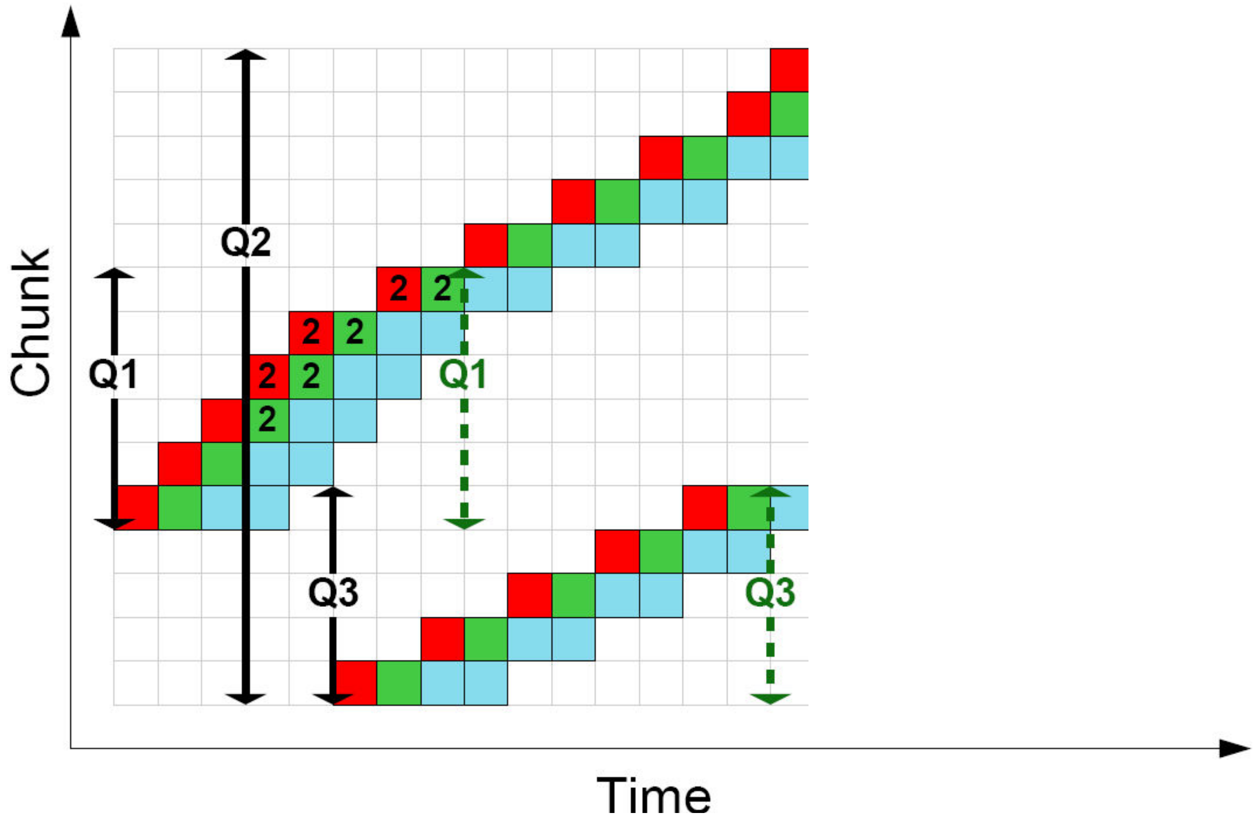
"Attach" policy

- Attach to a running query with data overlap



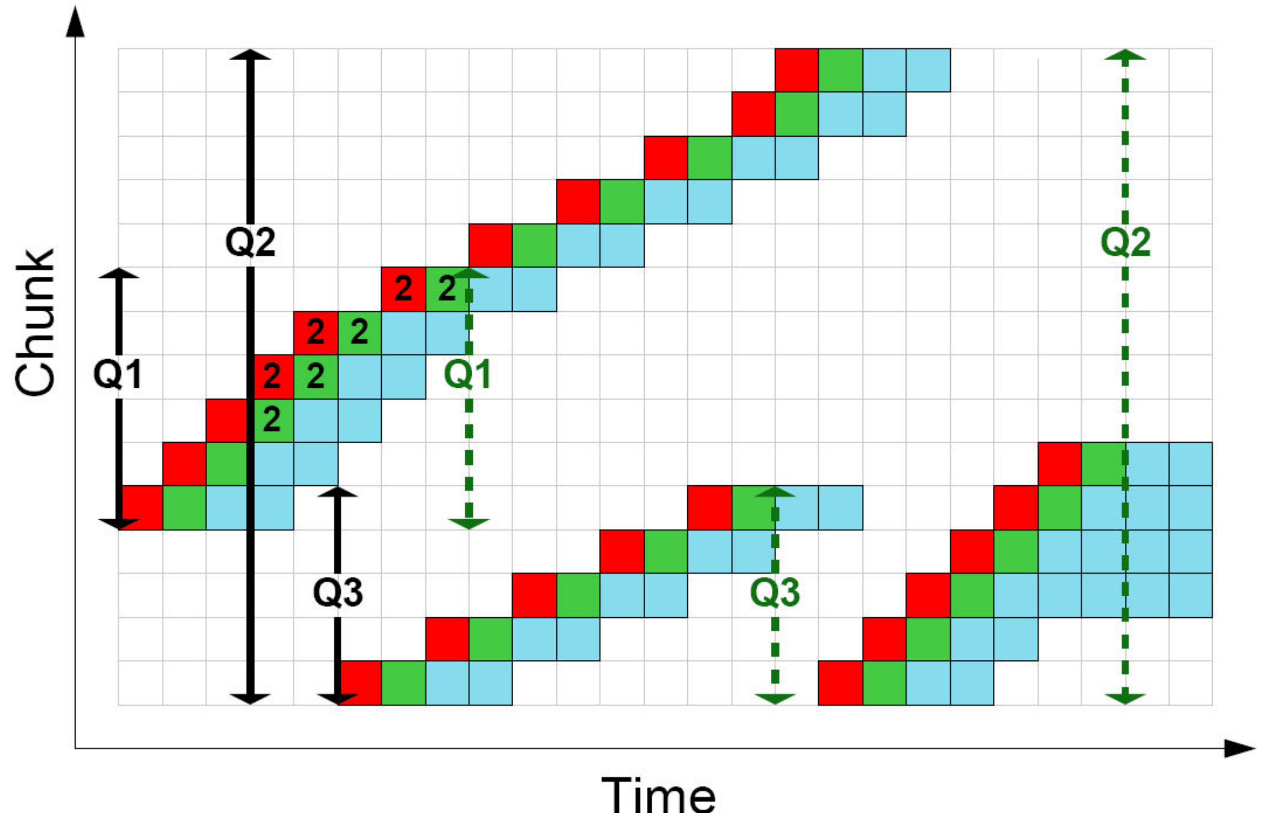
"Attach" policy

- Attach to a running query with data overlap



“Attach” policy

- Attach to a running query with data overlap

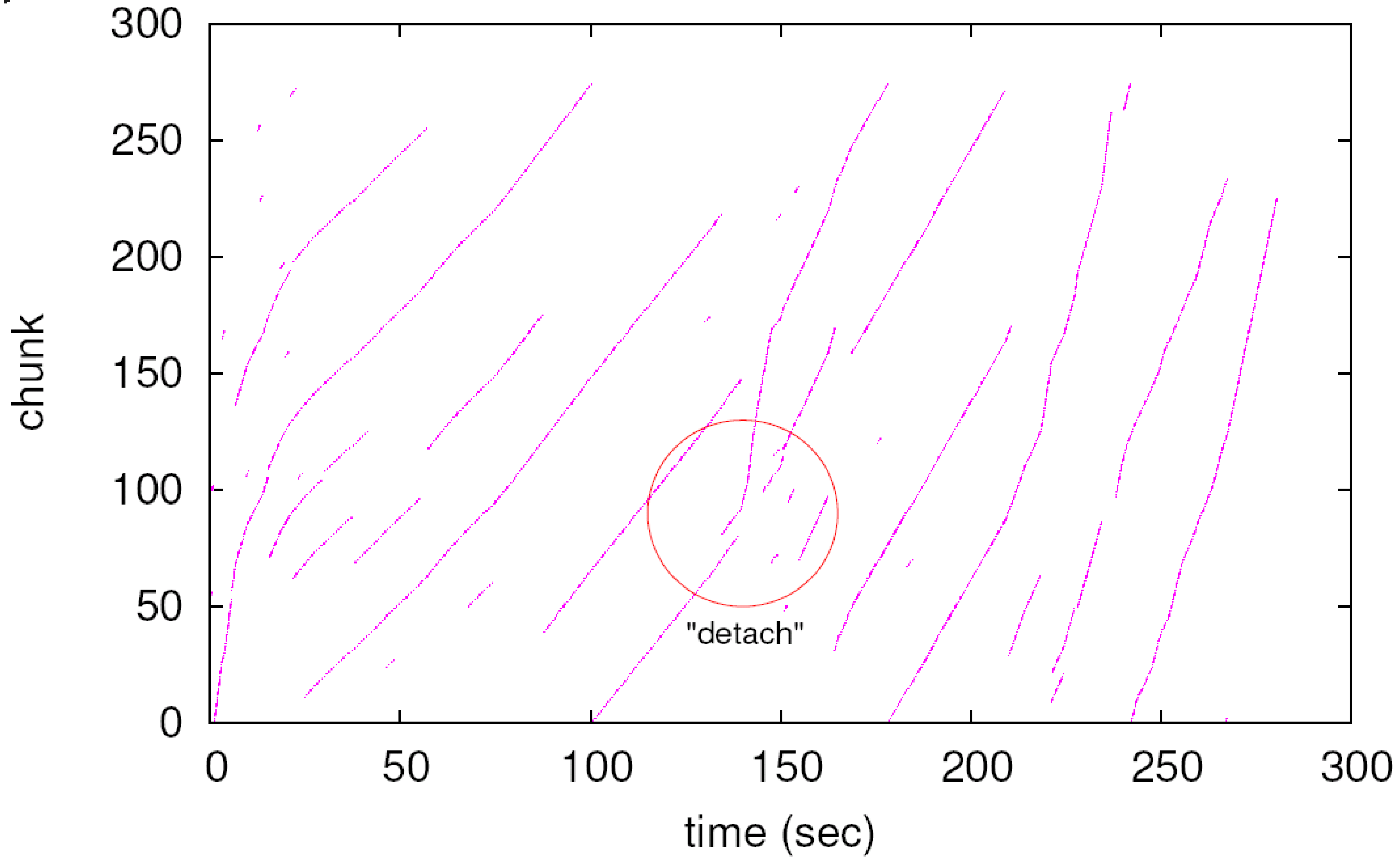


"Attach" performance

- Better than Normal
- Only one overlapping range is used
- Queries with different speeds can "detach"

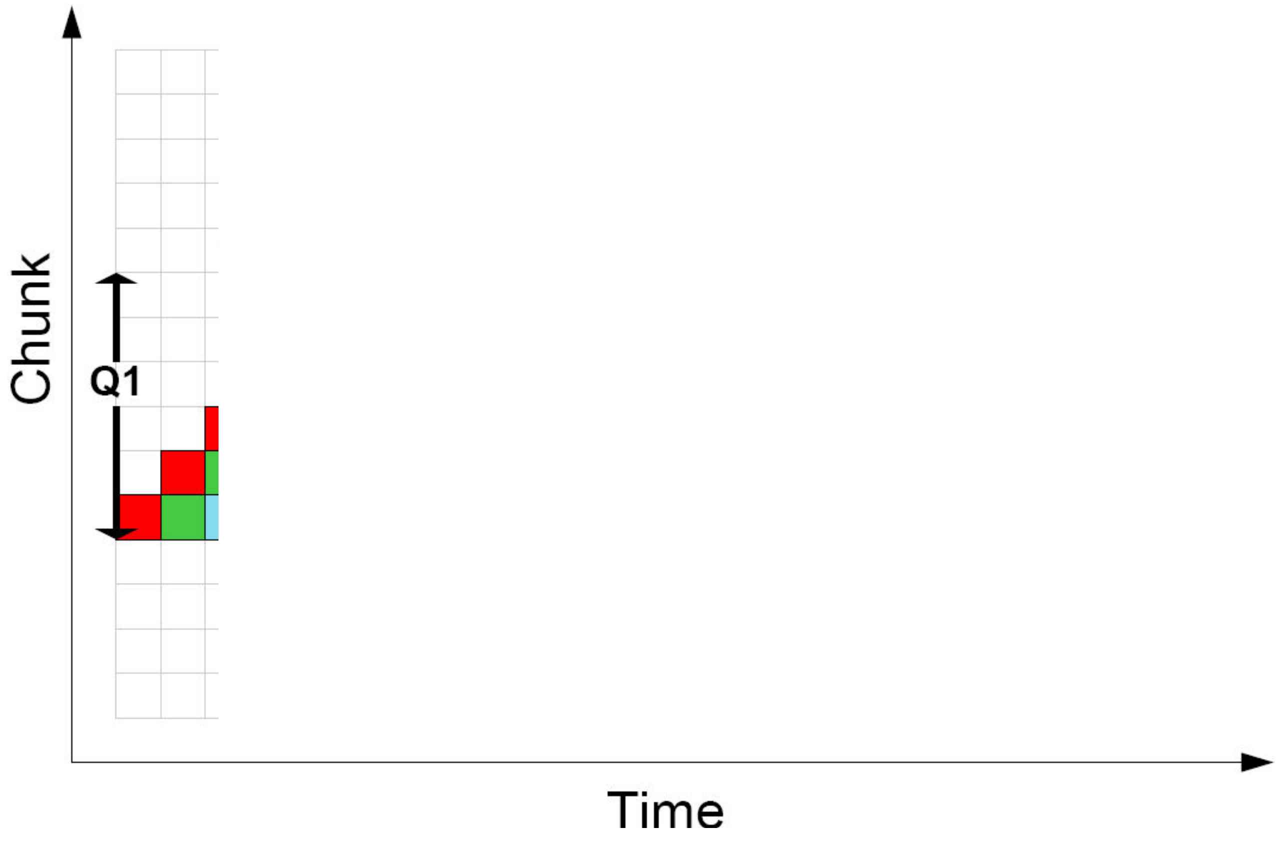
- Slightly improved by Lang et al, ICDE 2007

“Attach” in real life



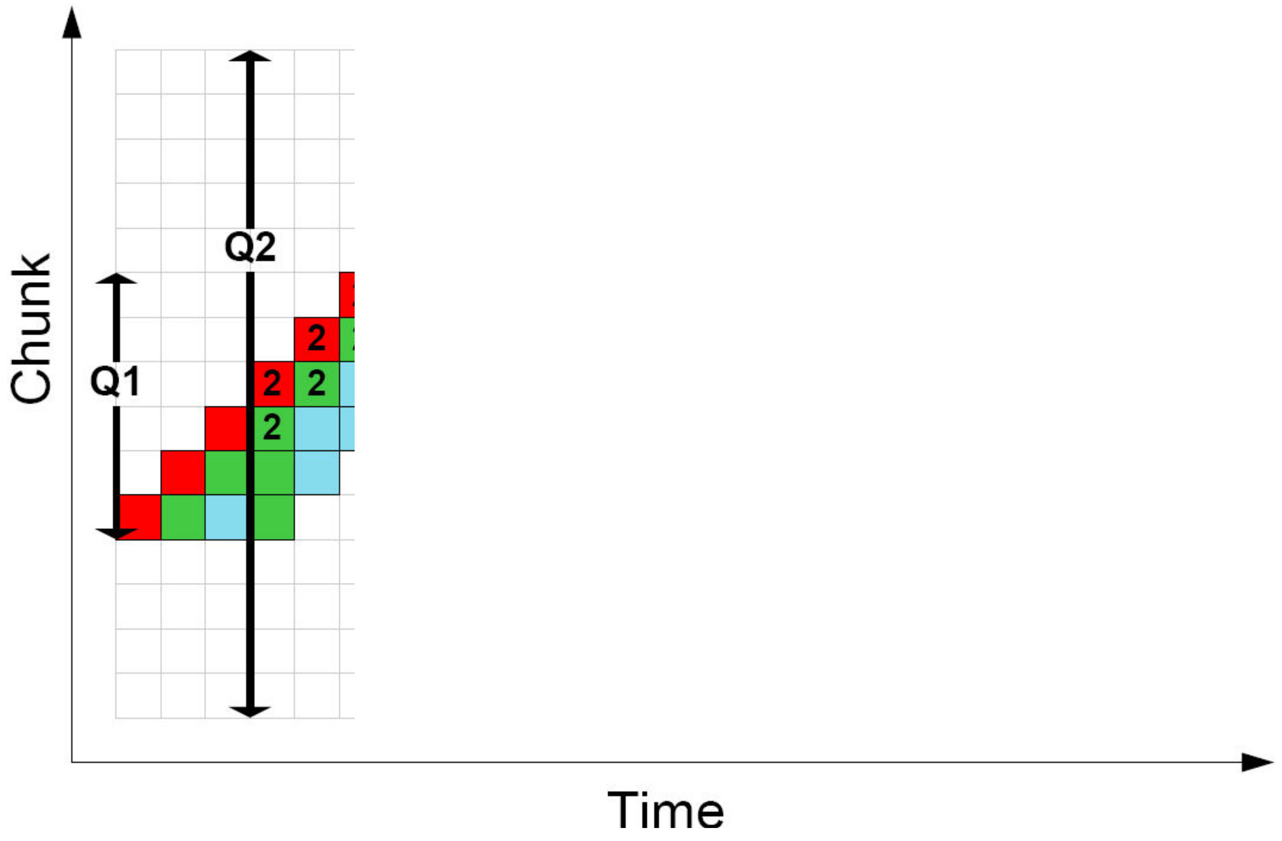
“Elevator” policy

- A single sliding window over a table



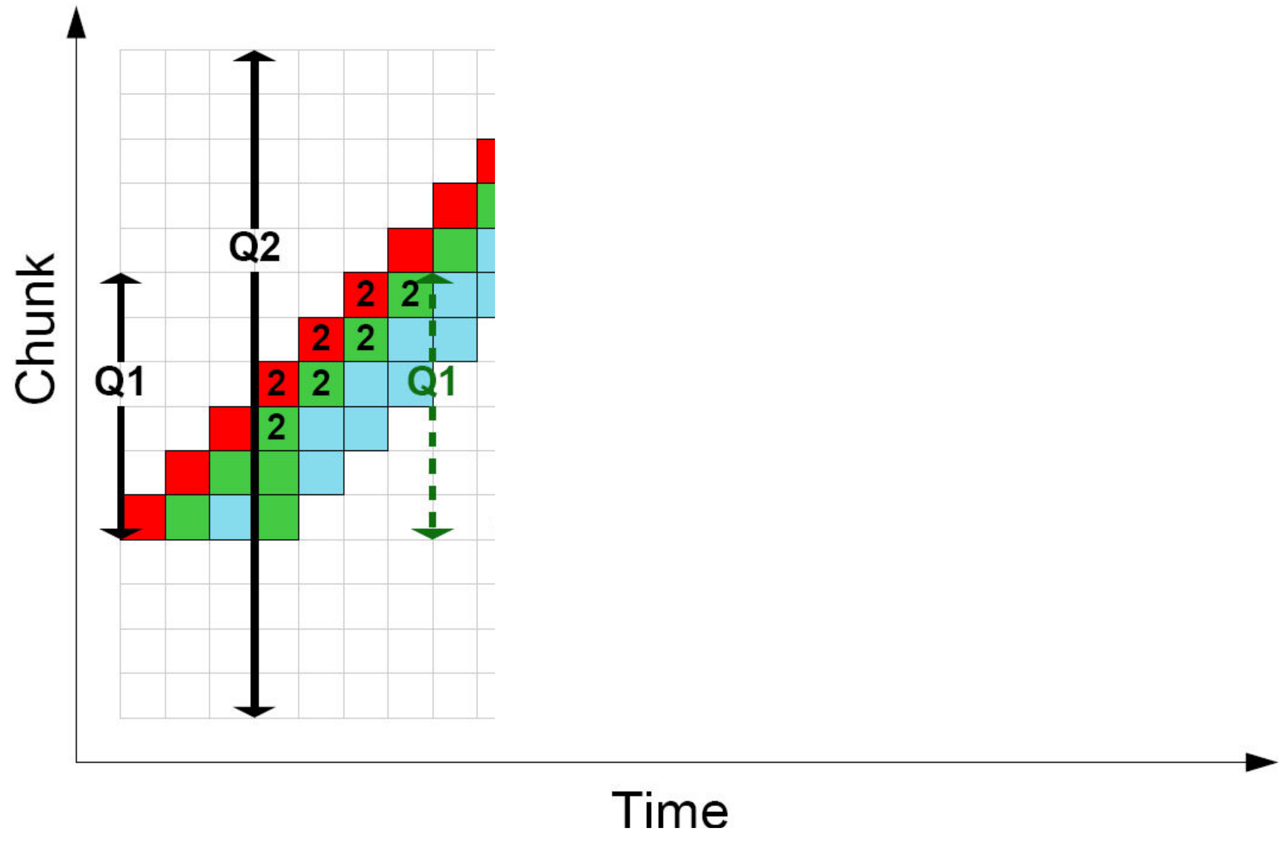
"Elevator" policy

- A single sliding window over a table



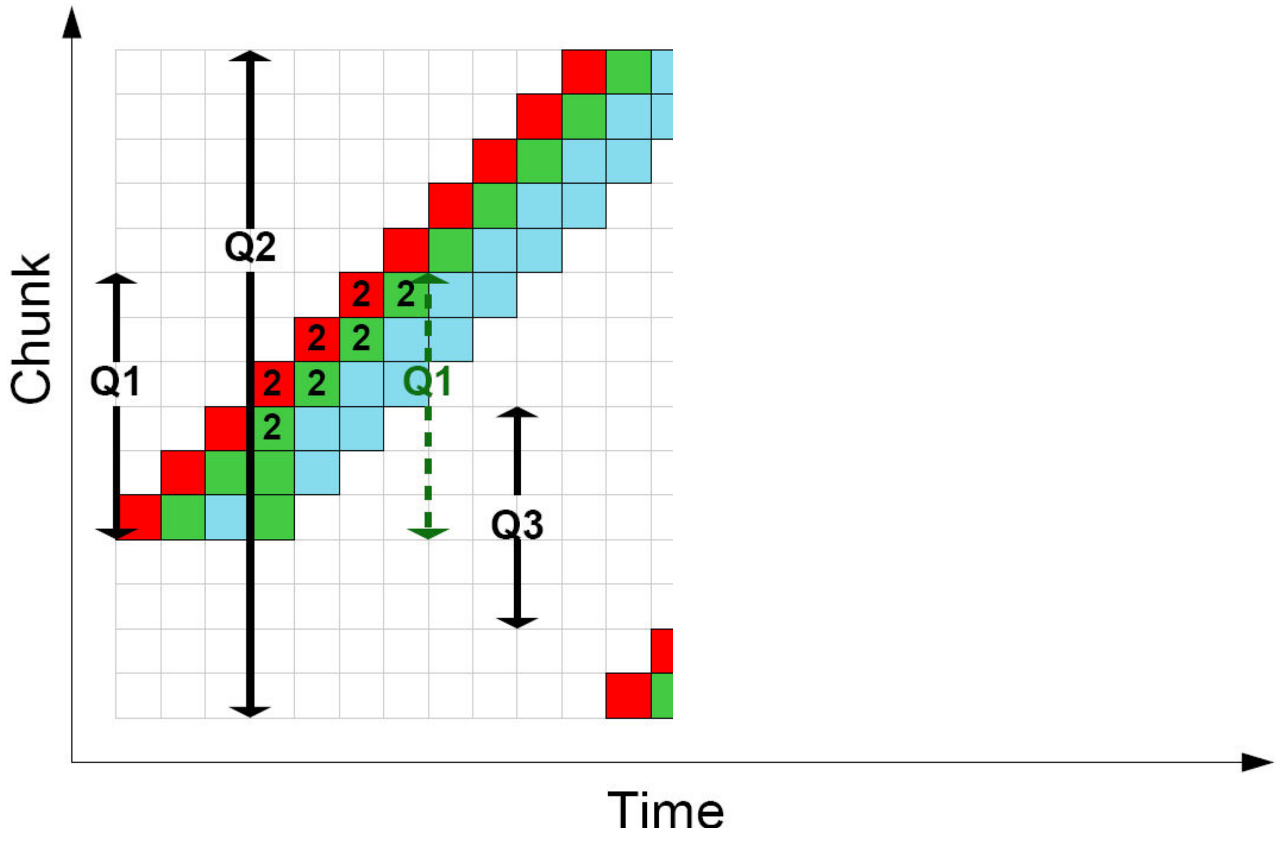
"Elevator" policy

- A single sliding window over a table



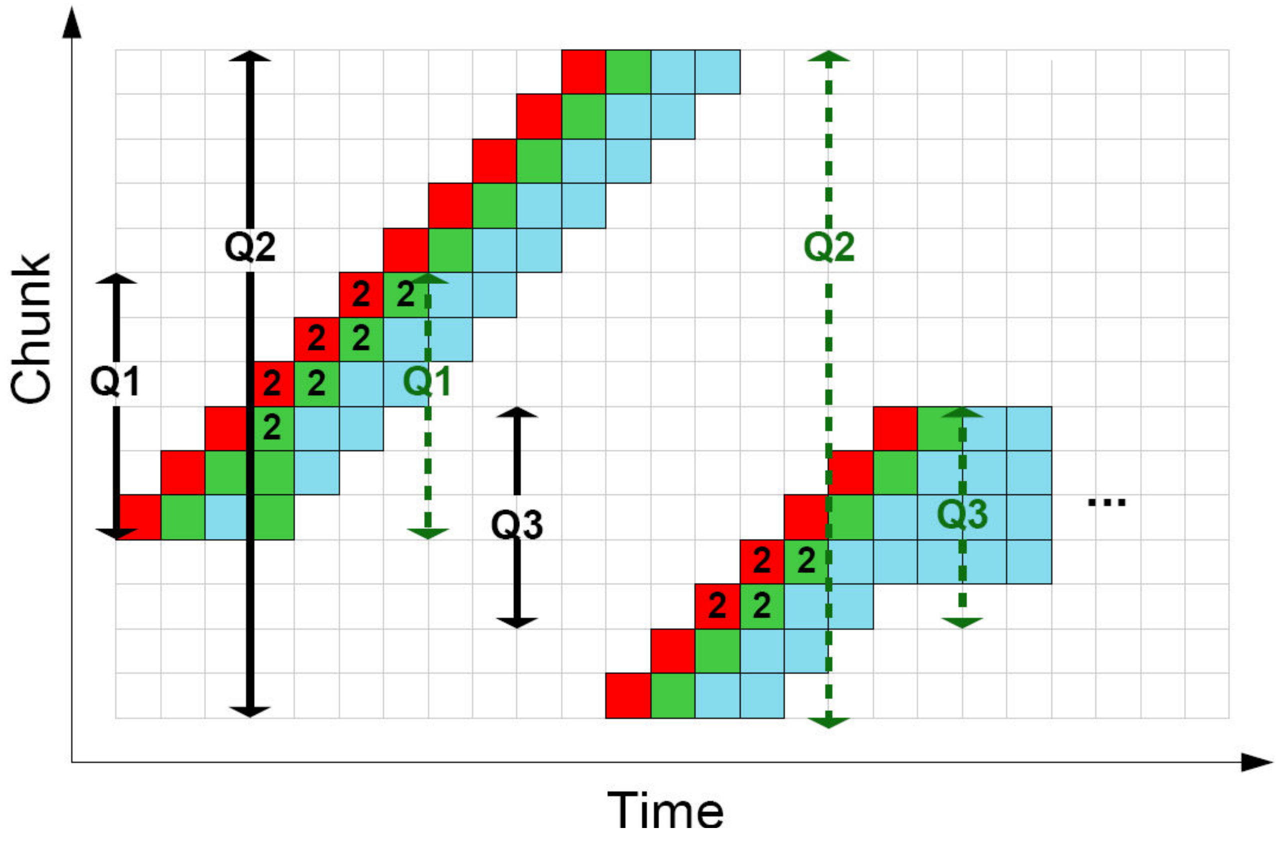
"Elevator" policy

- A single sliding window over a table



“Elevator” policy

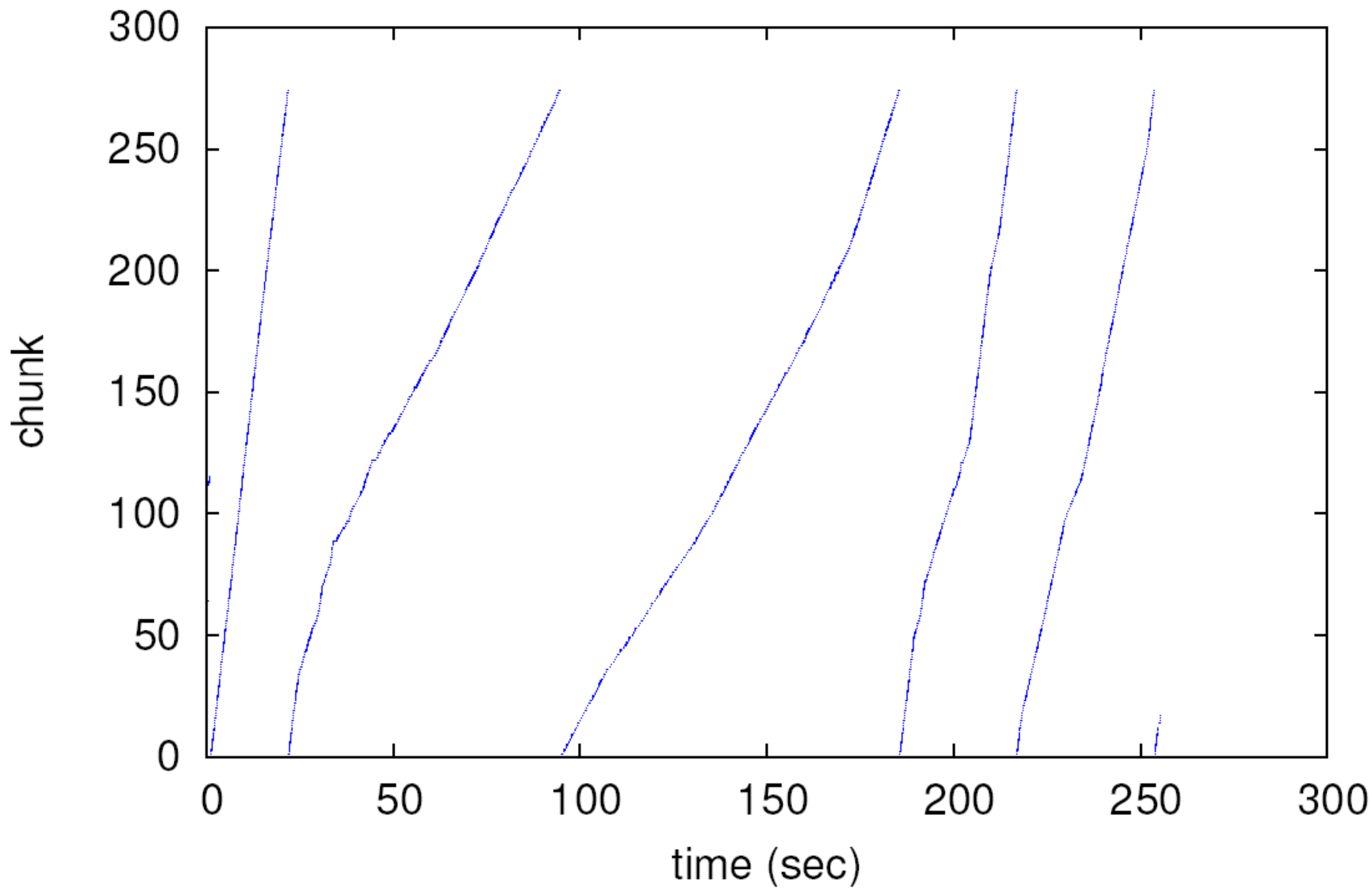
- A single sliding window over a table



“Elevator” performance

- Maximizes sharing, minimizes I/Os
- Good for long queries and uniform loads
- Short queries wait for the window
- Fast queries wait for the slow ones

“Elevator” in real life



Shared scans – main problem

- Query read sequence in shared scans:
  - Broken into 2 parts
  - Then fully static
  - Misses opportunities in a dynamic environment

Cooperative scans

- Core ideas
  - Dynamically adapt to the current situation
  - Allow fully arbitrary chunk order
- Goals:
  - Maximize data sharing
  - Optimize latency and throughput
  - Work for different types of queries

Active Buffer Manager

- ABM knows the status of all the queries
- Queries investigate ABM content:
  - If some chunks buffered, choose one to use
  - If not, wait for ABM to provide a new chunk
- ABM in a loop:
  - Chooses a query to serve
  - Chooses a chunk to load
  - If out of space, chooses what to keep
  - Loads a chunk and notifies the queries

"Relevance" functions

- ABM knows the status of all the queries
- Queries investigate ABM content:
  - If some chunks buffered, cl
  - If not, wait for ABM to provide a new chunk

useRelevance()

- ABM in a loop:
  - Chooses a query to serve
  - Chooses a chunk to load
  - If out of space, chooses wh
  - Loads a chunk and notifies the queries

queryRelevance()

loadRelevance()

keepRelevance()

useRelevance()

- Choose a chunk with a minimal number of queries interested
  - Allows early chunk eviction

queryRelevance()

- Choose “starved” queries only
  - Queries having data are doing fine
- Promote short queries
  - Better query-stream throughput
  - Avoid round-robin request scheduling
- Promote long-waiting queries
  - Don’t let short queries starve the long ones

loadRelevance()

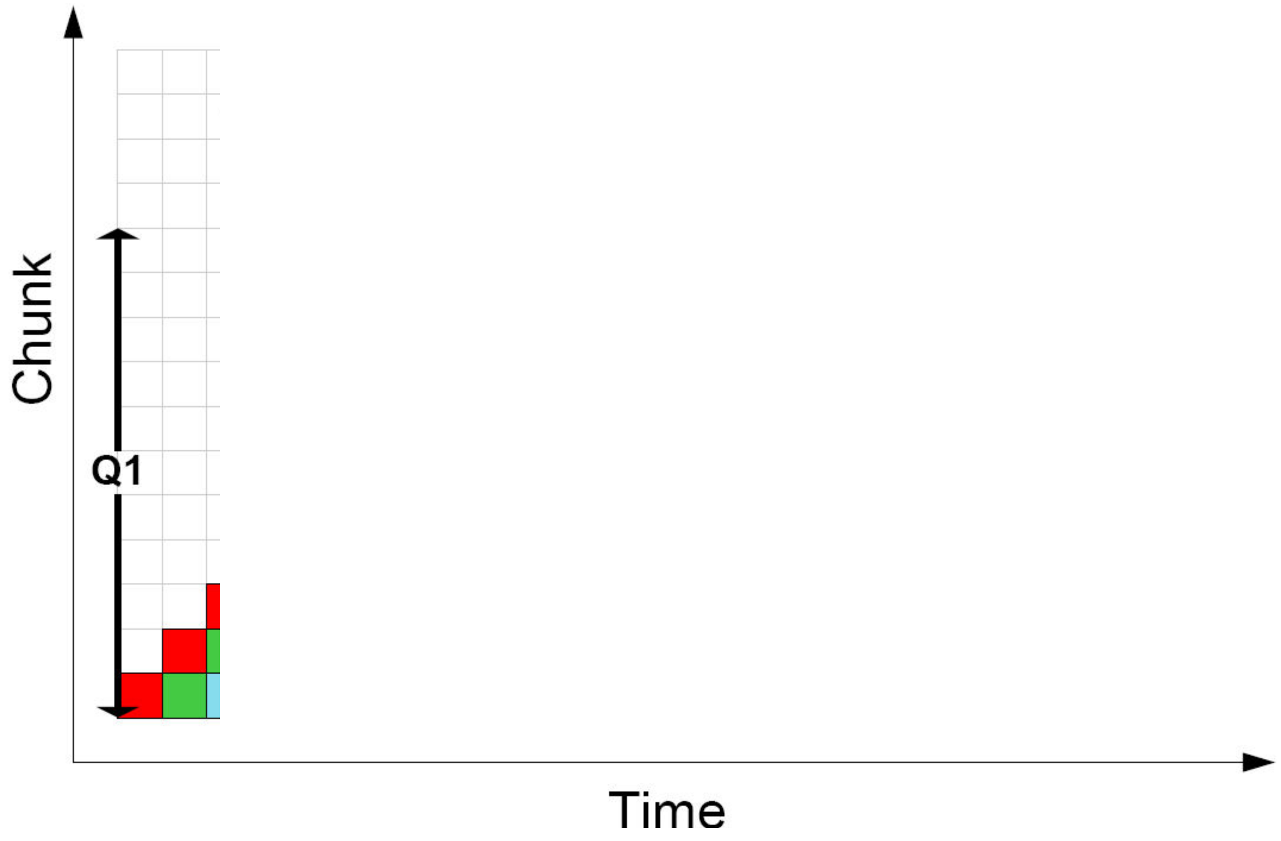
- Load chunks interesting for the maximum number of starved queries
  - Keep many queries busy
  - Amortize loading cost among many queries

keepRelevance()

- Keep chunks interesting for the maximum number of (almost) starved queries
  - Avoid blocking queries

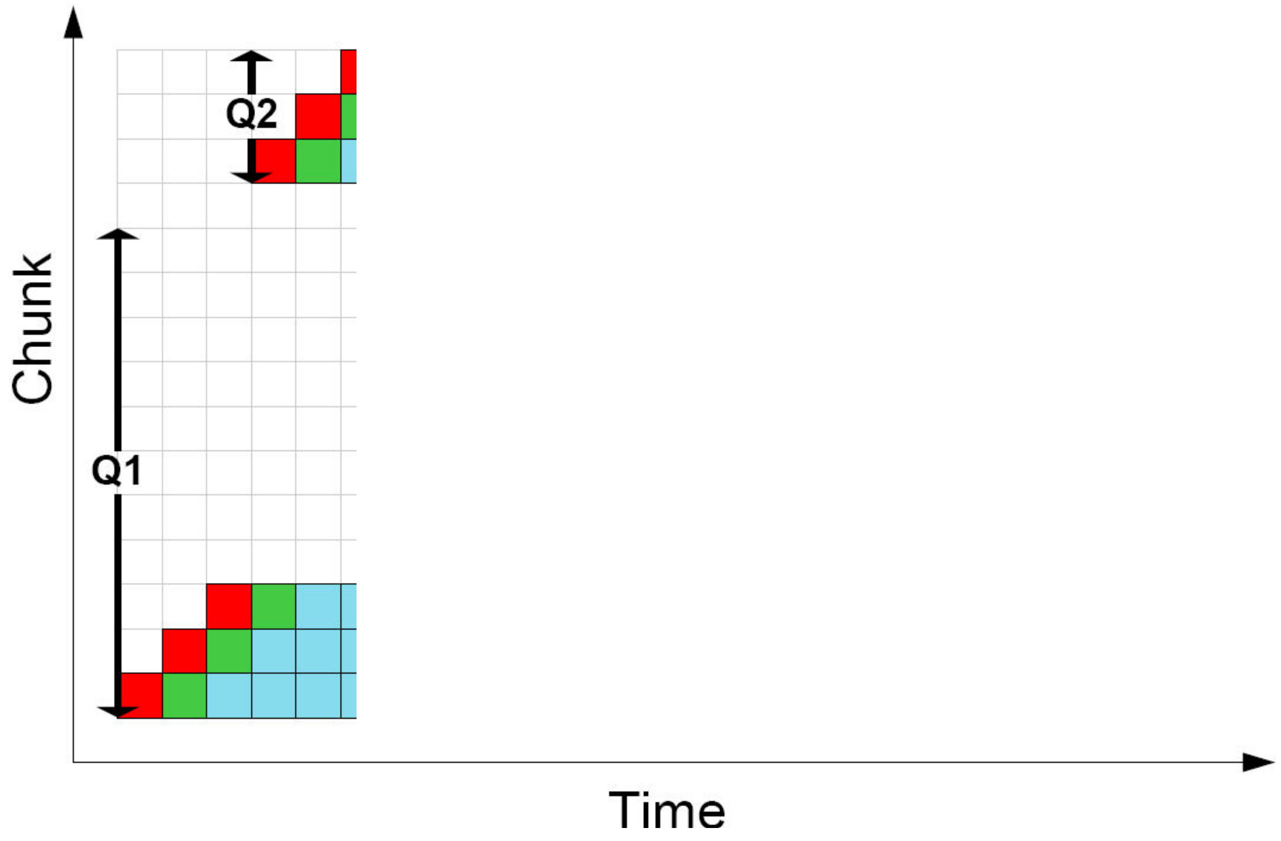
"Relevance policy"

- Follow the relevance functions



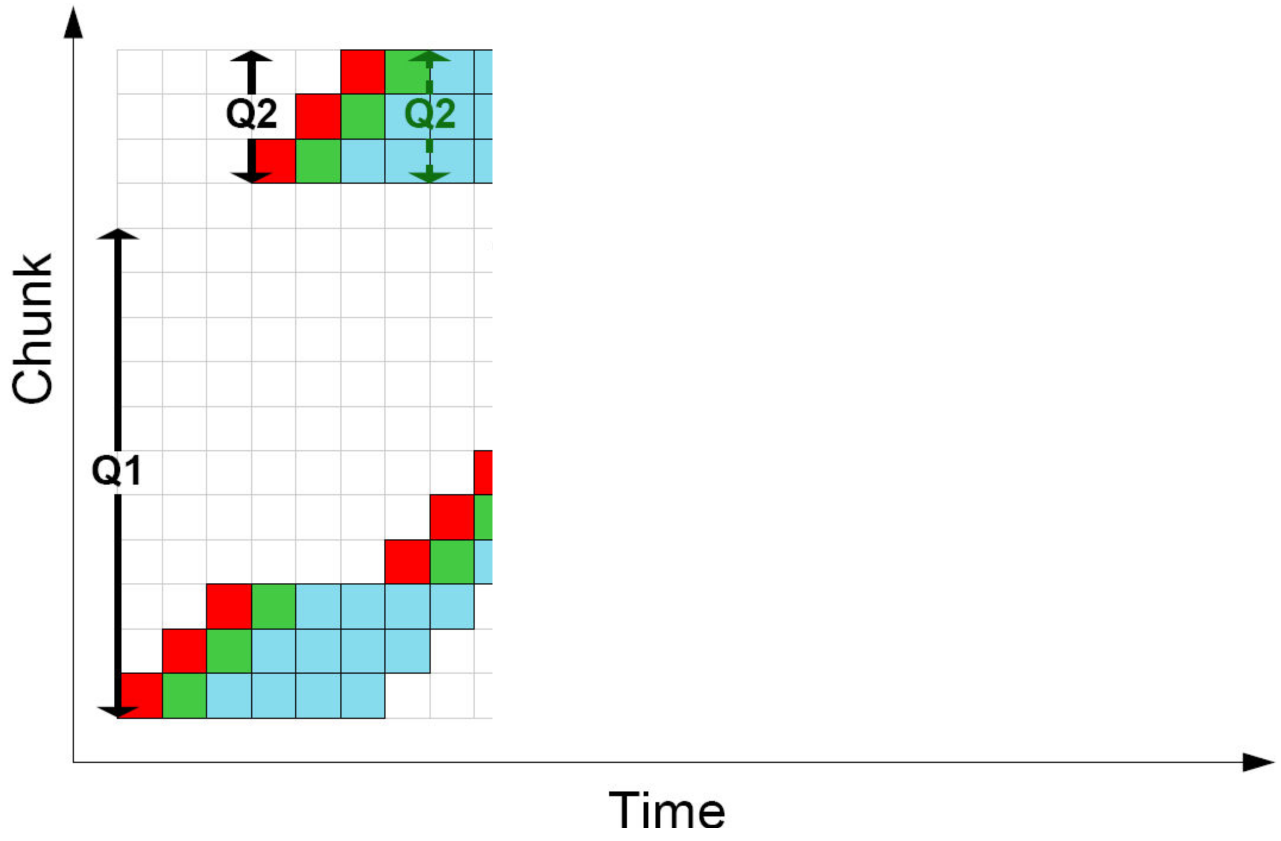
"Relevance policy"

- Follow the relevance functions



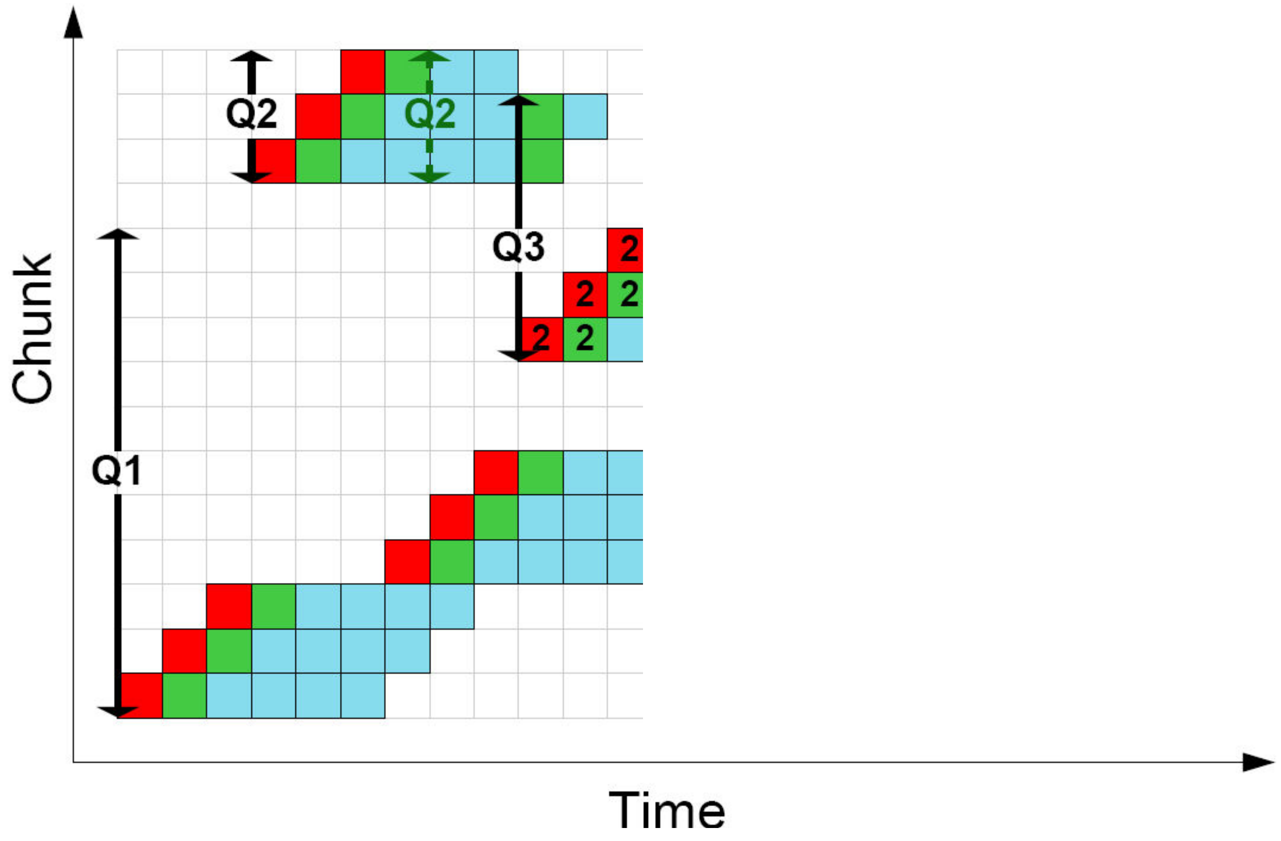
“Relevance policy”

- Follow the relevance functions



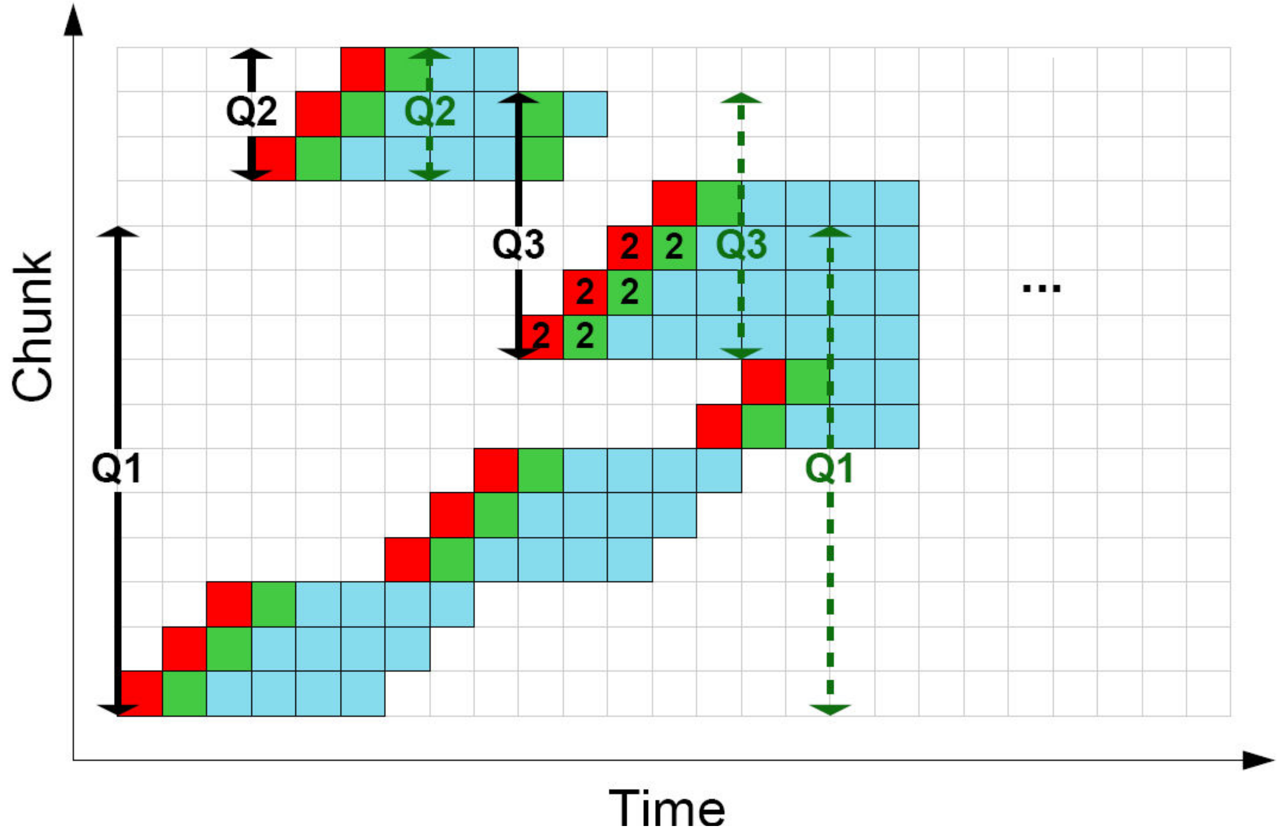
"Relevance policy"

- Follow the relevance functions

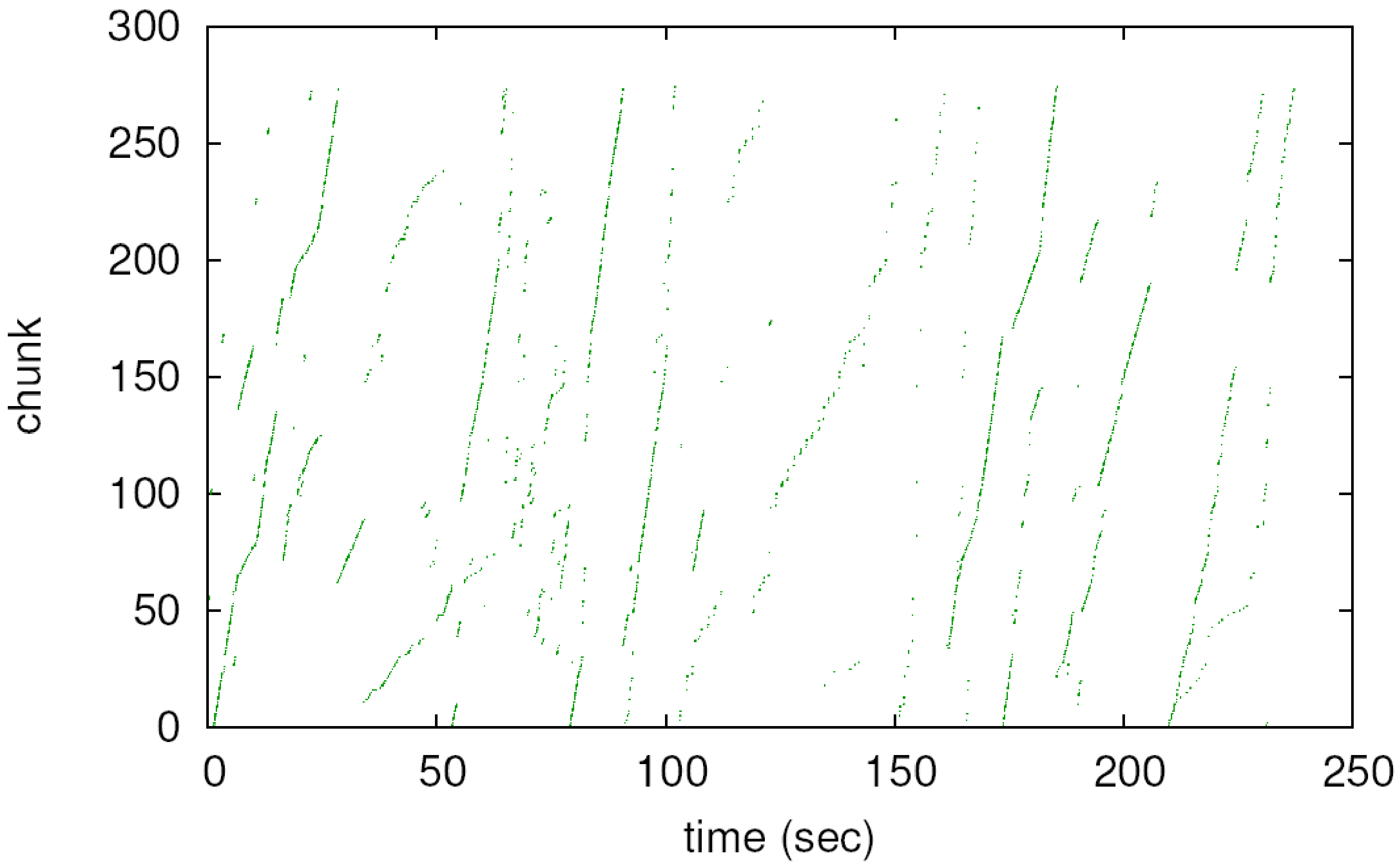


“Relevance policy”

- Follow the relevance functions



“Relevance” in real life



Simple benchmark

- TPC-H SF-10 dataset
- MonetDB/X100, PAX storage
- Two query speeds: Fast (Q6), Slow (Q1)
- Varying scan ranges: 1%, 10%, 50%, 100%, at random positions
- 16 concurrent streams, 3 seconds delay
- 4 random queries in each stream

Slow queries

	Standalone	Normal	Attach	Elevator	Relevance
S-1%	*0.38*	1.67	1.19	**15.01**	**0.55**
S-10%	*3.55*	**21.58**	15.12	20.29	**11.30**
S-50%	*17.73*	**78.23**	46.98	**37.39**	37.77
S-100%	*35.27*	**179.35**	105.51	**79.39**	98.71

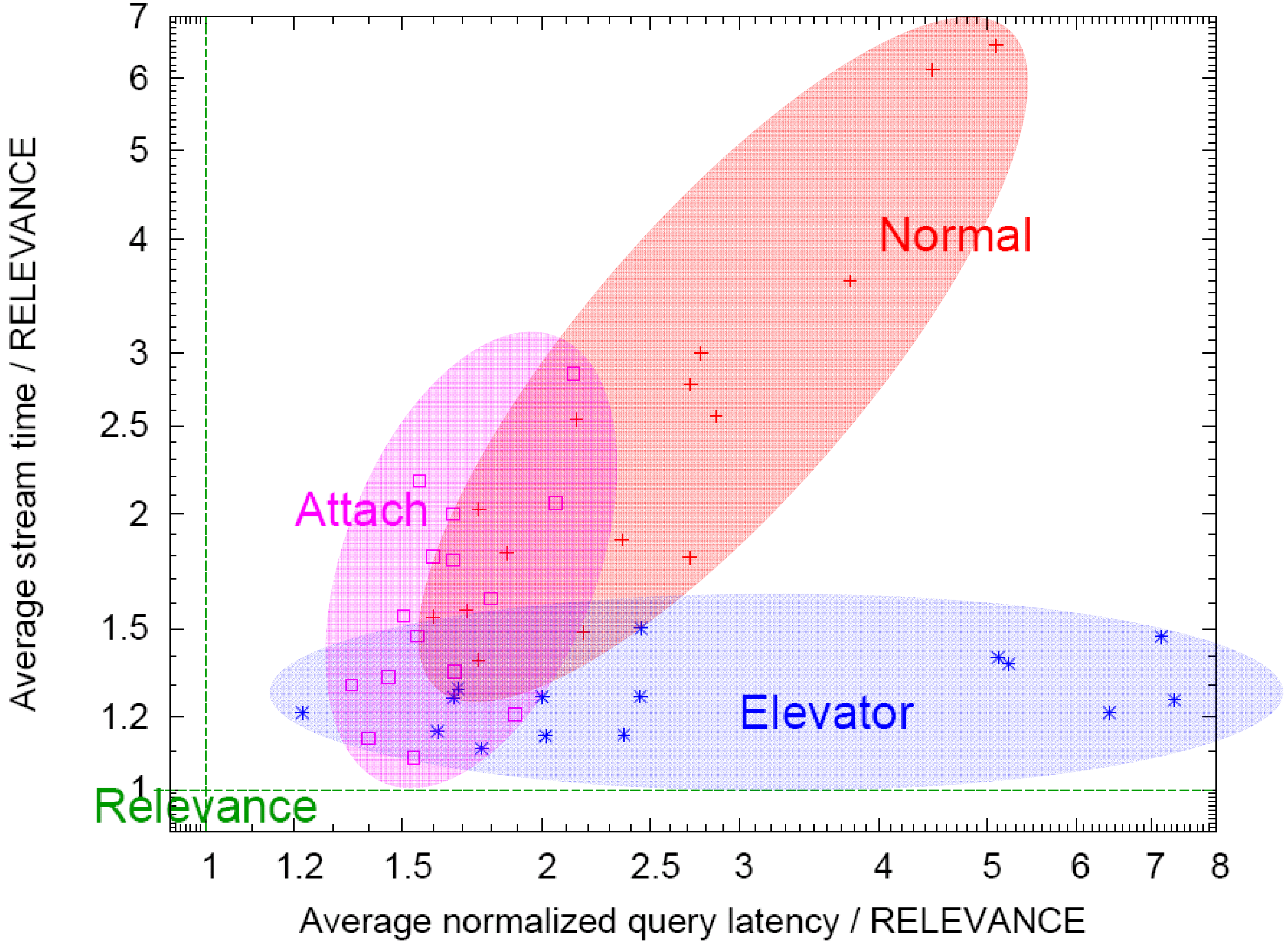
Fast queries

	Standalone	Normal	Attach	Elevator	Relevance
F-1%	*0.26*	1.71	1.02	**5.31**	**0.52**
F-10%	*2.06*	13.97	6.23	**15.17**	**3.30**
F-50%	*10.72*	**103.59**	58.77	44.87	**18.21**
F-100%	*20.37*	**192.82**	96.98	59.60	**29.01**

Global results

	Normal	Attach	Elevator	Relevance
Avg. stream time (sec)	**283**	160	138	**99**
Avg. normalized query latency	6.42	3.72	**13.52**	**1.96**
Total time (sec)	**453**	281	244	**238**
CPU usage (%)	**53.20**	81.31	90.20	**93.94**
Number of I/Os	**4186**	2325	**1404**	1842

Different query mixes



Other experiments (paper!)

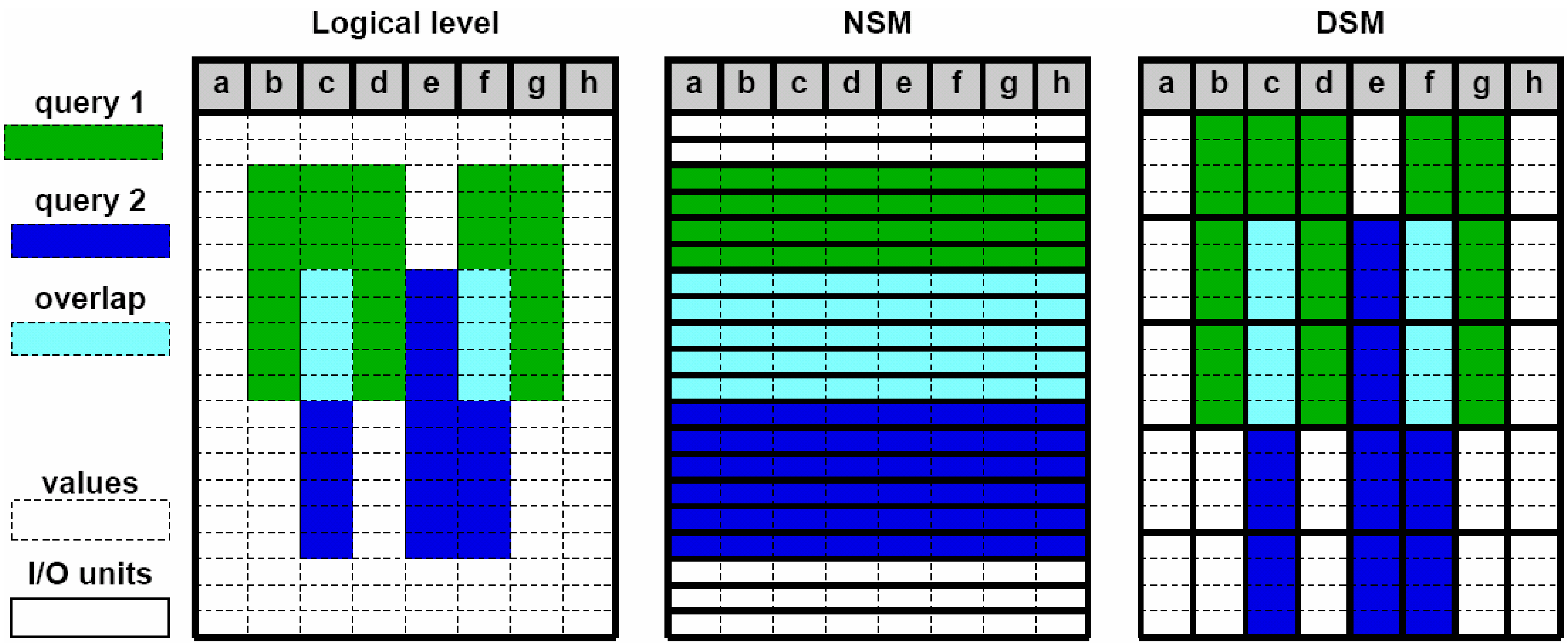
- Relevance still best with:
  - Different query lengths and number
  - Different buffer capacity
- Scheduling costs:
  - Below 1% in the worst tested case
  - Can grow for really large tables
  - Area for optimizations

Cooperative Scans with DSM

- Reduced sharing opportunities
- Physical-logical data mismatch
- More complex ABM implementation and relevance functions

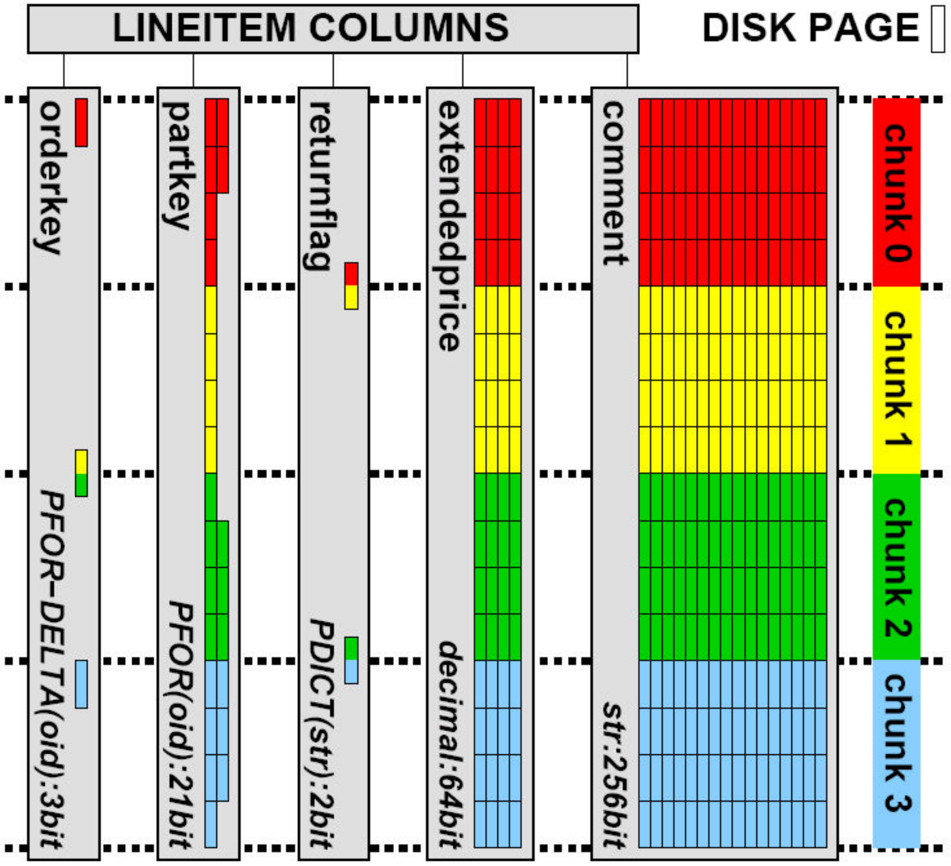
Sharing opportunities in DSM

- Both vertical and horizontal overlap needed



Chunks in DSM

- Larger I/O requirements
- Columns with various physical sizes
- Logical chunks overlap physically
- Chunks as logical concepts



ABM in DSM

- More complex policies
  - 2-dimensional decisions: chunk + column
  - Columns with different queries interested
  - Even column loading order matters!
- Still, it works
  - Results depend on the overlap (paper)

DSM – global results

	Normal	Attach	Elevator	Relevance
Avg. stream time (sec)	**536**	338	352	**264**
Avg. normalized query latency	7.05	4.77	**15.11**	**2.96**
Total time (sec)	**805**	621	562	**515**
CPU usage (%)	**61**	77	82	**92**
Number of I/Os	**6490**	4413	**2297**	3639

Conclusions

- New, dynamic scan processing strategy
- Consistently improves query latency and system throughput
- Works for both NSM and DSM
- Future work:
  - Investigating relevance functions
  - Adapting order-aware operators

Thank you!

Questions?

BACKUP

SLIDES

What is it all about?

- Scan-oriented database scenarios
- Multiple scans on the same table
- Exploit inter-queries relationships
- Dynamic scan scheduling
- Significantly improved performance!

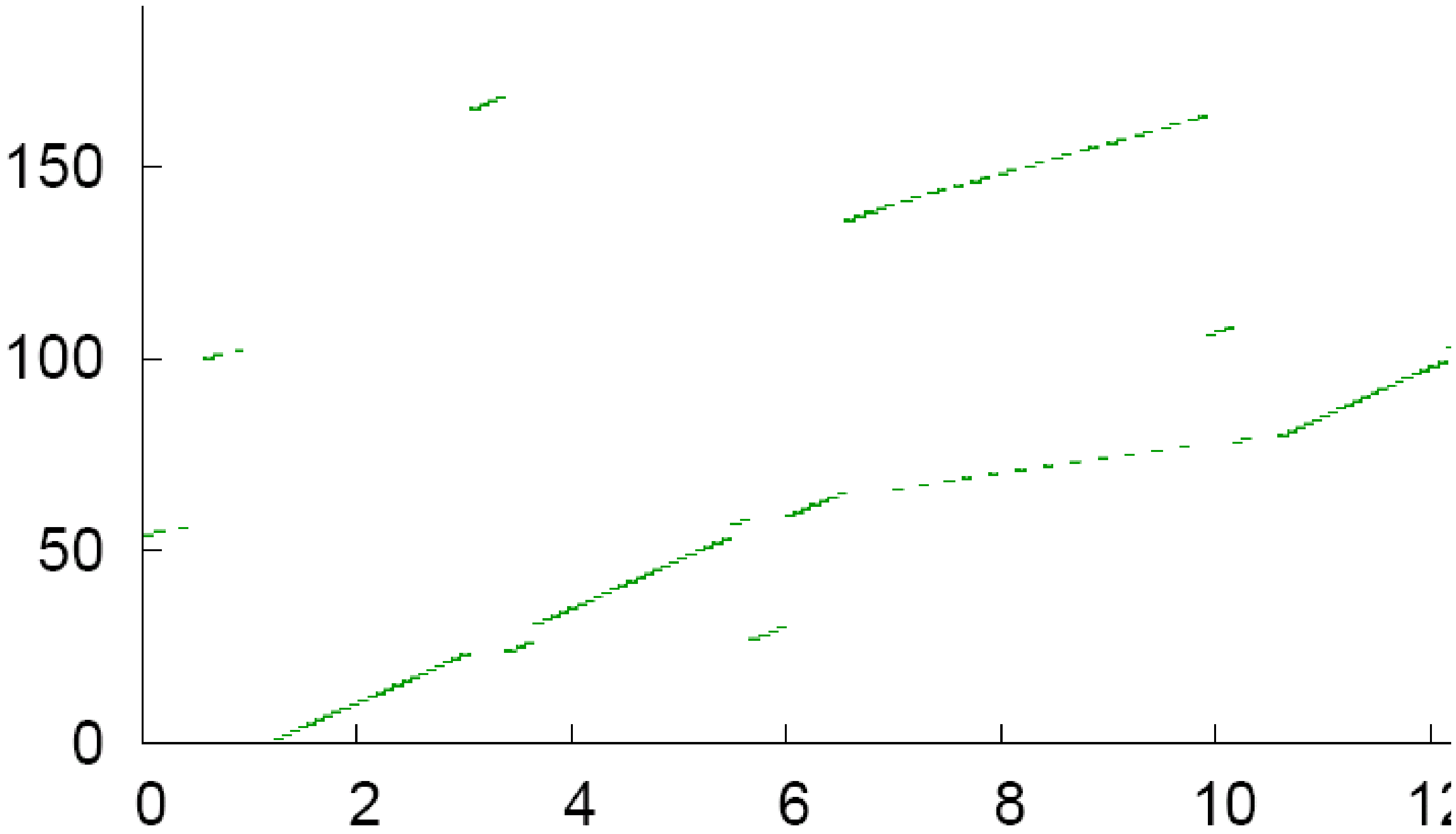
“Normal” policy

- Fully sequential read order for each query
- Limited sharing opportunities
- Query gets only a fraction of disk bandwidth
- FCFS / round-robin scheduling bad for latency

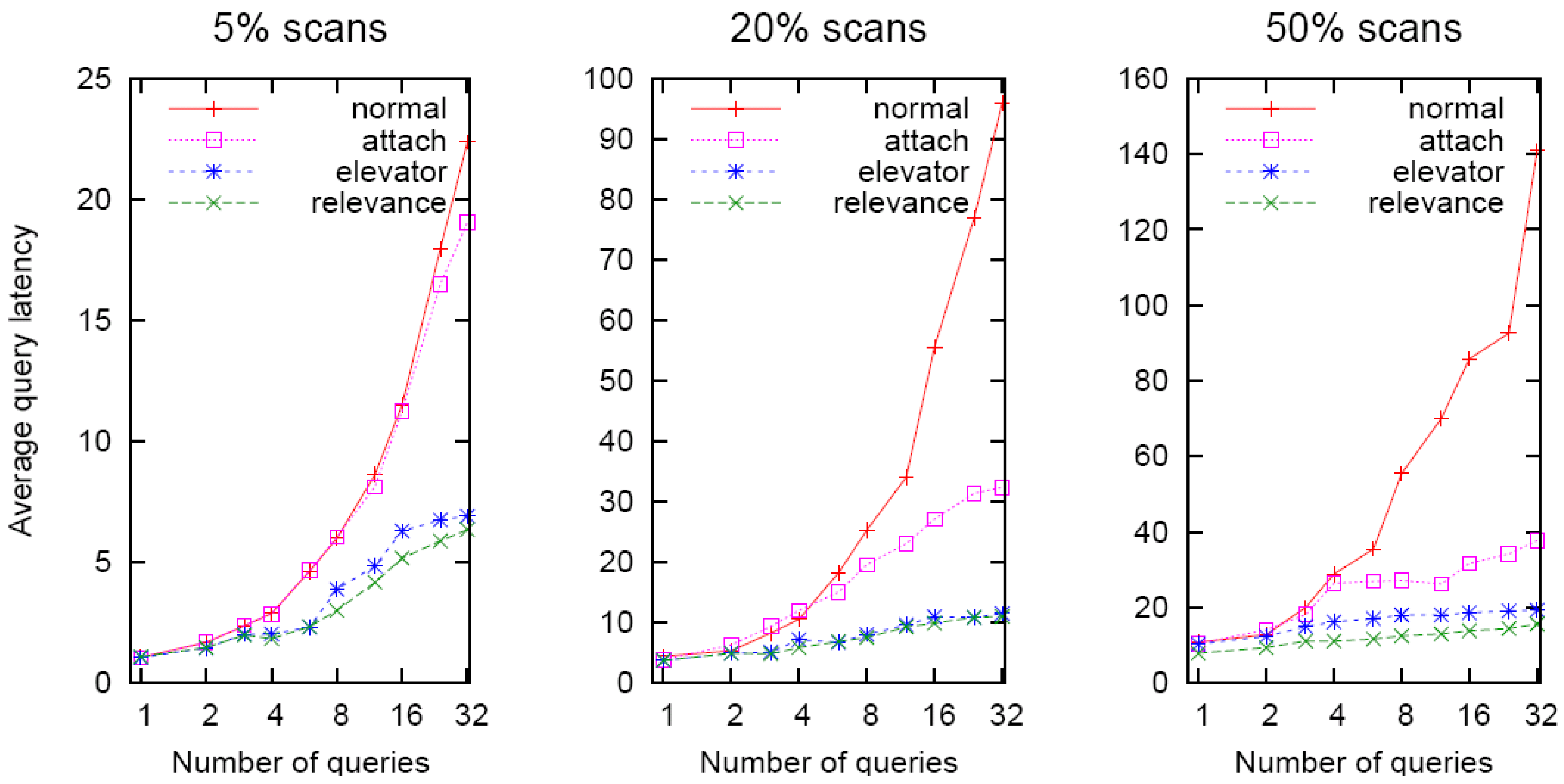
“Attach”

- Idea:
  - New query finds a running, overlapping one
  - It starts processing at that query position
  - When at the end, starts from the beginning
  - Queries share I/O and buffer space (unless…)
- Problems:
  - Queries with different speeds can “detach”
  - Only one overlapping range is used

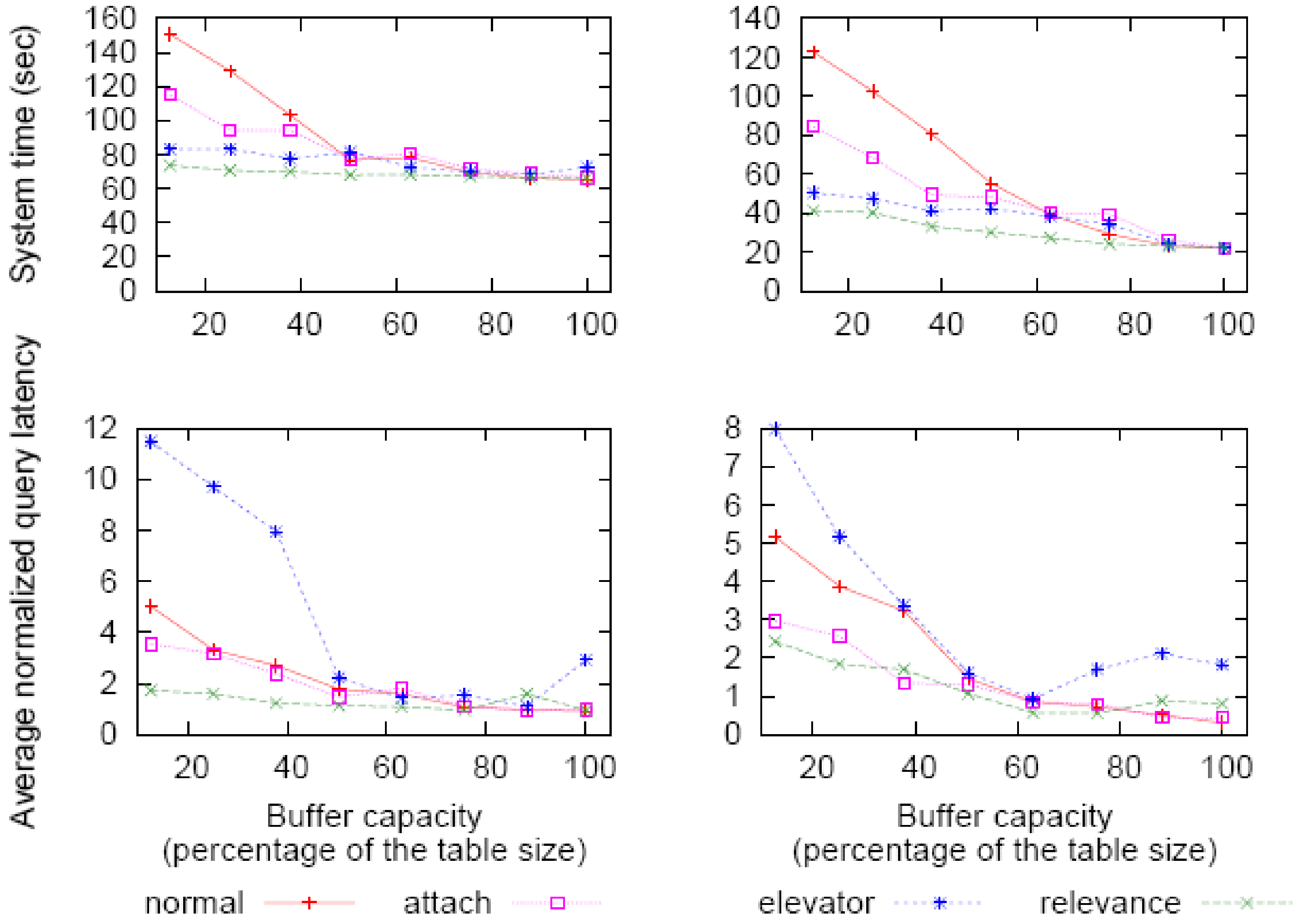
Relevance – microscope view



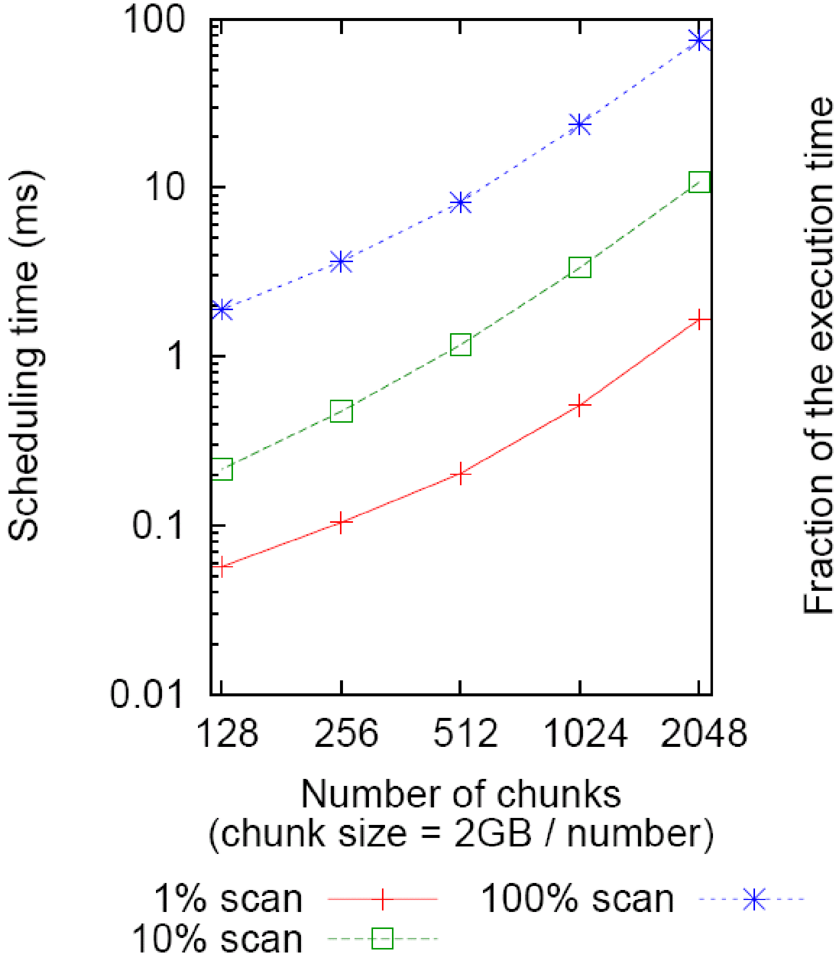
Impact of query sizes

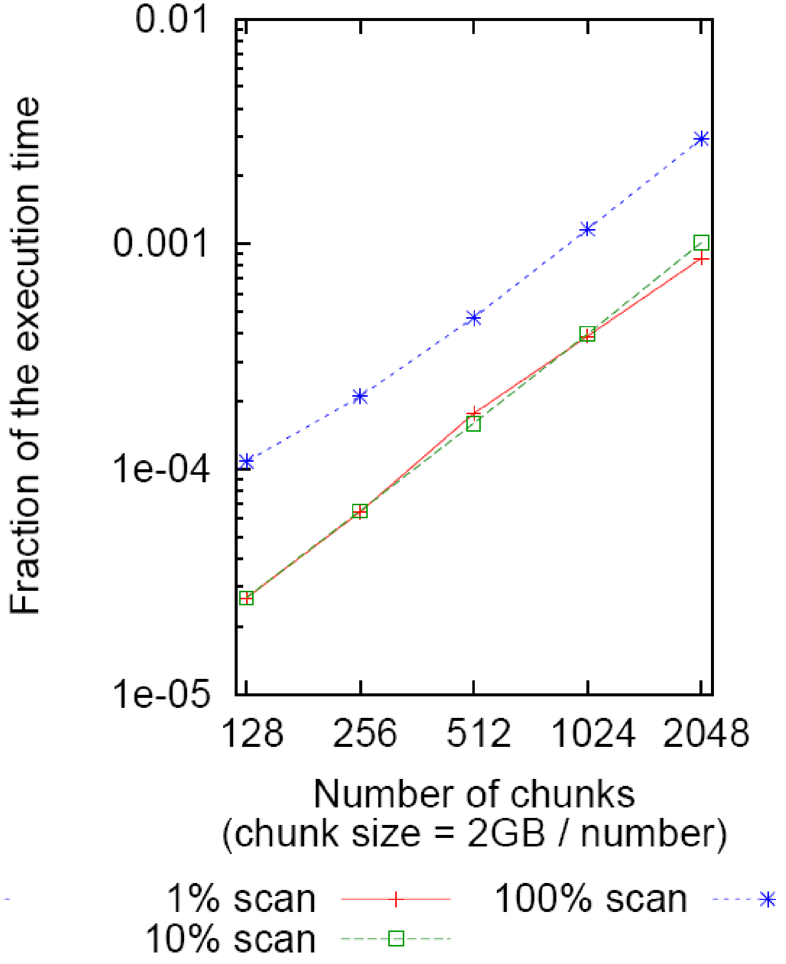


Impact of buffer-pool size



Scheduling cost





Order-aware operators

- CScans: “random” chunk order
  - Breaks order-aware operators
  - Still, chunks internally ordered
- Adapting order-aware operators
  - Ordered aggregation: easy
  - Merge-join:
    - Easy if the other table in memory
    - Hard otherwise

END

OF

BACKUP

SLIDES