ACM SIGMOD 2012 programming contest

Ioana Ileana (team Slowpoke)

1st year PhD student in the DBWeb group at Telecom ParisTech, advised by Bogdan Cautis.
PhD area: Web data extraction and archiving.

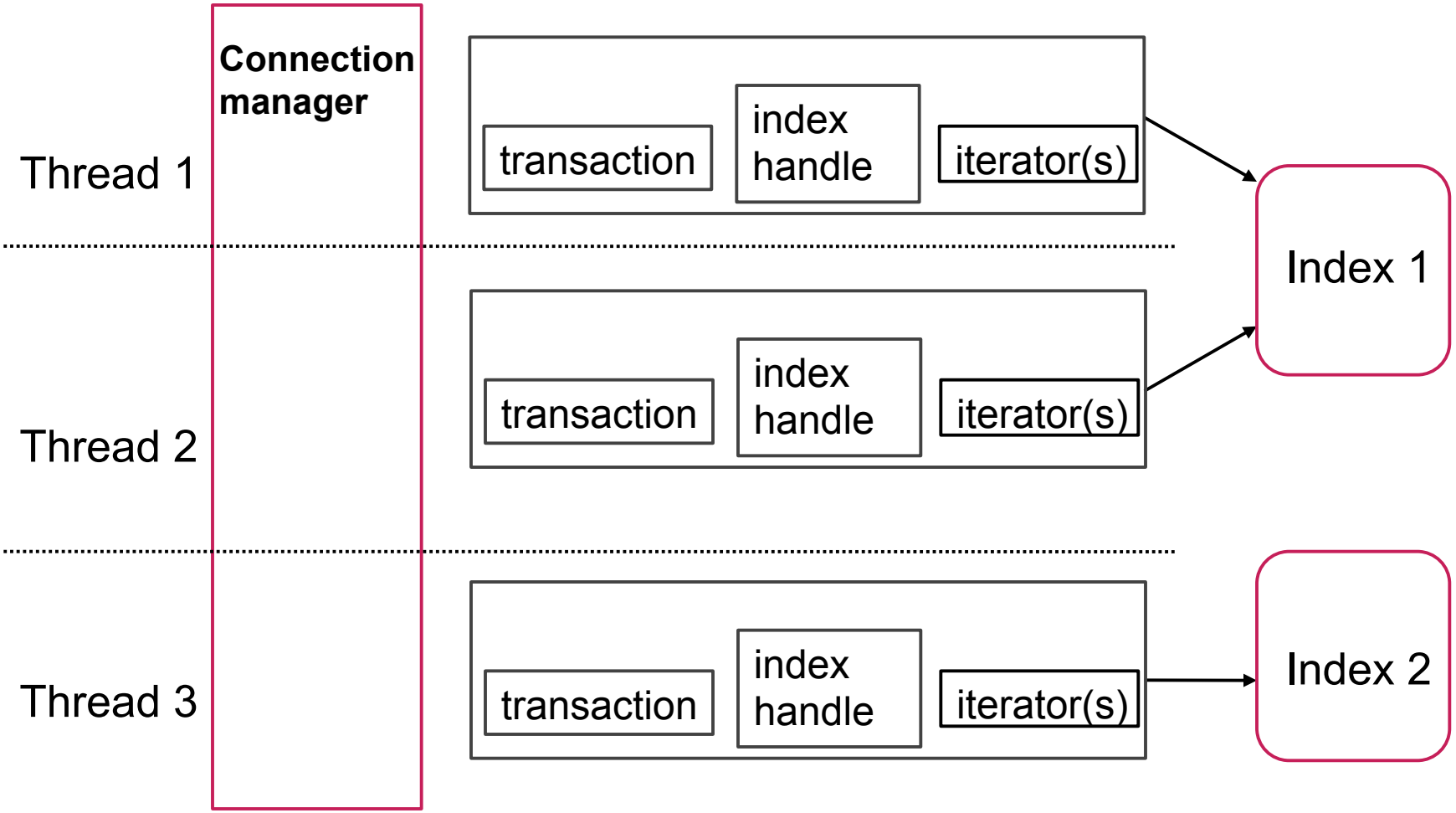
Target : simple, robust, scalable solution
Main challenges : parallelism, global order preserving

- Main bricks
- Search data structures
- Memory management

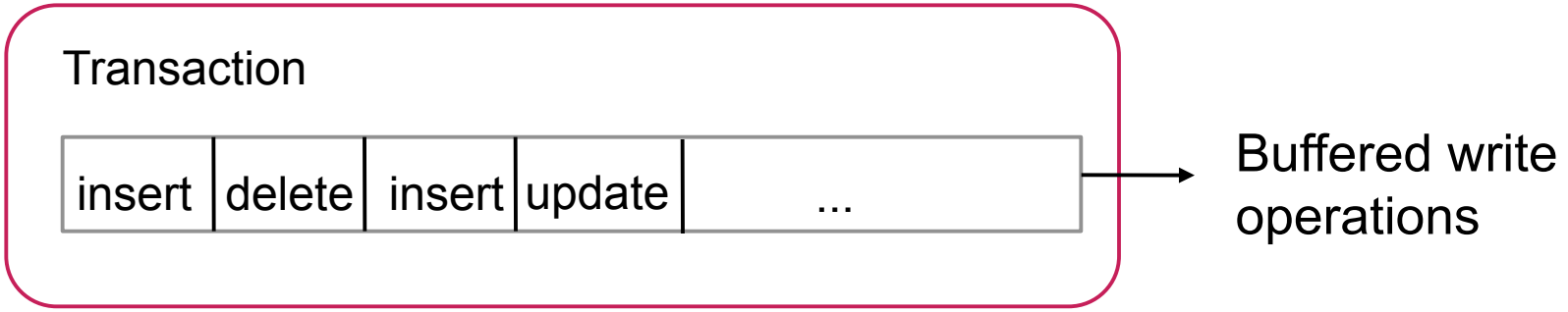
Target : simple, robust, scalable solution
Main challenges : parallelism, global order preserving

- Main bricks
- Search data structures
- Memory management

Main bricks: overview

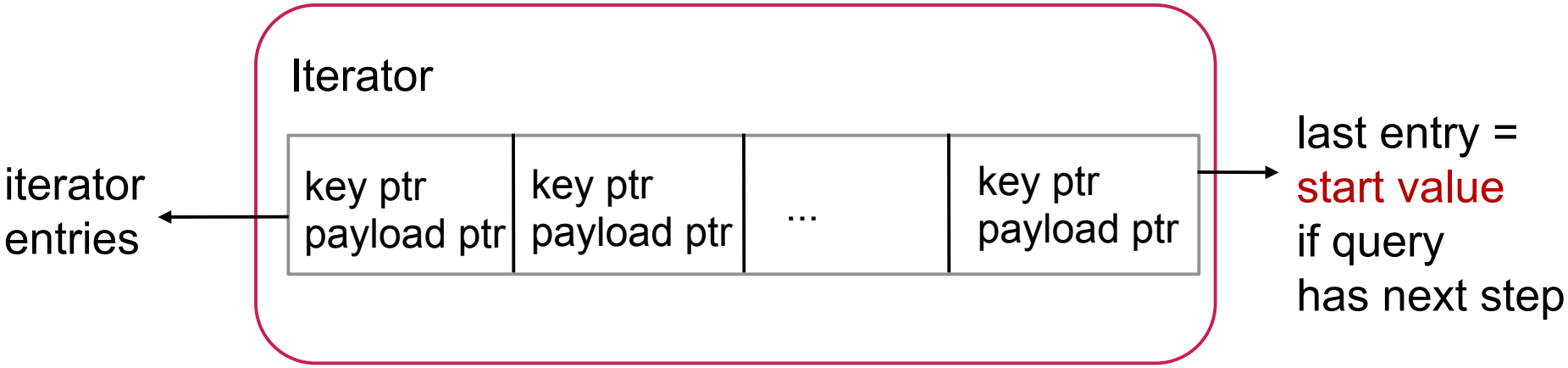


Main bricks: transactions



- Write operations are **buffered** until commit/abort
- One transaction's buffered operations are either
  - **atomically** executed upon commit
  - or discarded upon abort

Main bricks: iterators



- Maximal capacity, beyond which a new query is triggered
- First filled with index results, then adjusted with write op buffer
- Global order always preserved in the entries array

Main bricks: conclusions

Strengths

- No rollback management or index copy
- No memory exhaustion for queries returning a large amount of results
- Speed up: full iterator is a “stop search” criterion

Limitations

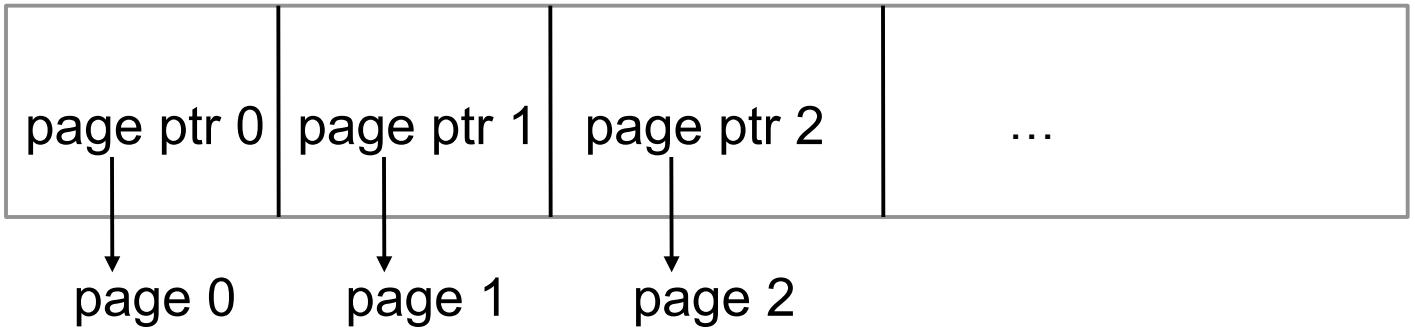
- Adapting iterator capacity
- Delaying write operation execution

Target : simple, robust, scalable solution
Main challenges : parallelism, global order preserving

- Main bricks
- Search data structures
- Memory management

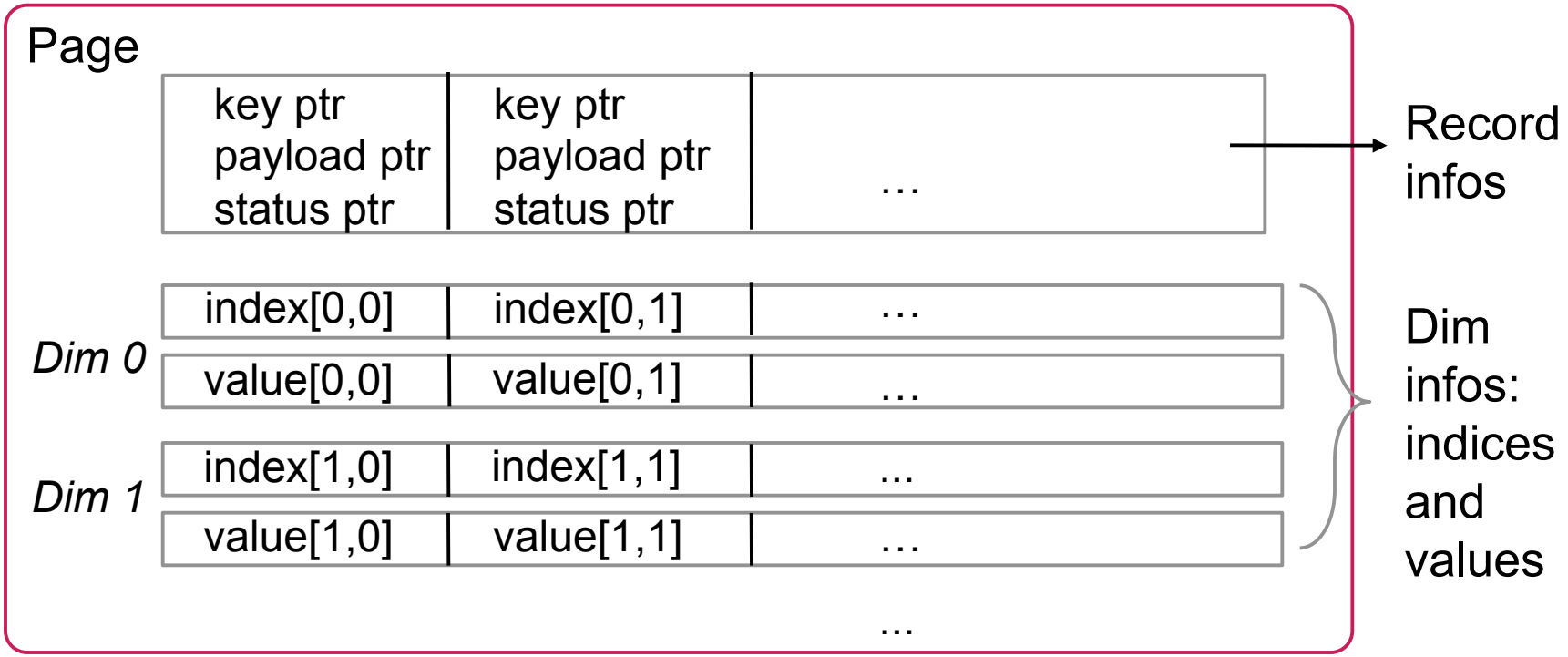
Search data structures: page array

- Constraint: keep the global order
- Idea: store the data in an ordered array, further divided in “pages”



- All data inside and among pages is ordered
- Two level search: on page array and inside pages
- Two level locking: on page array and inside pages

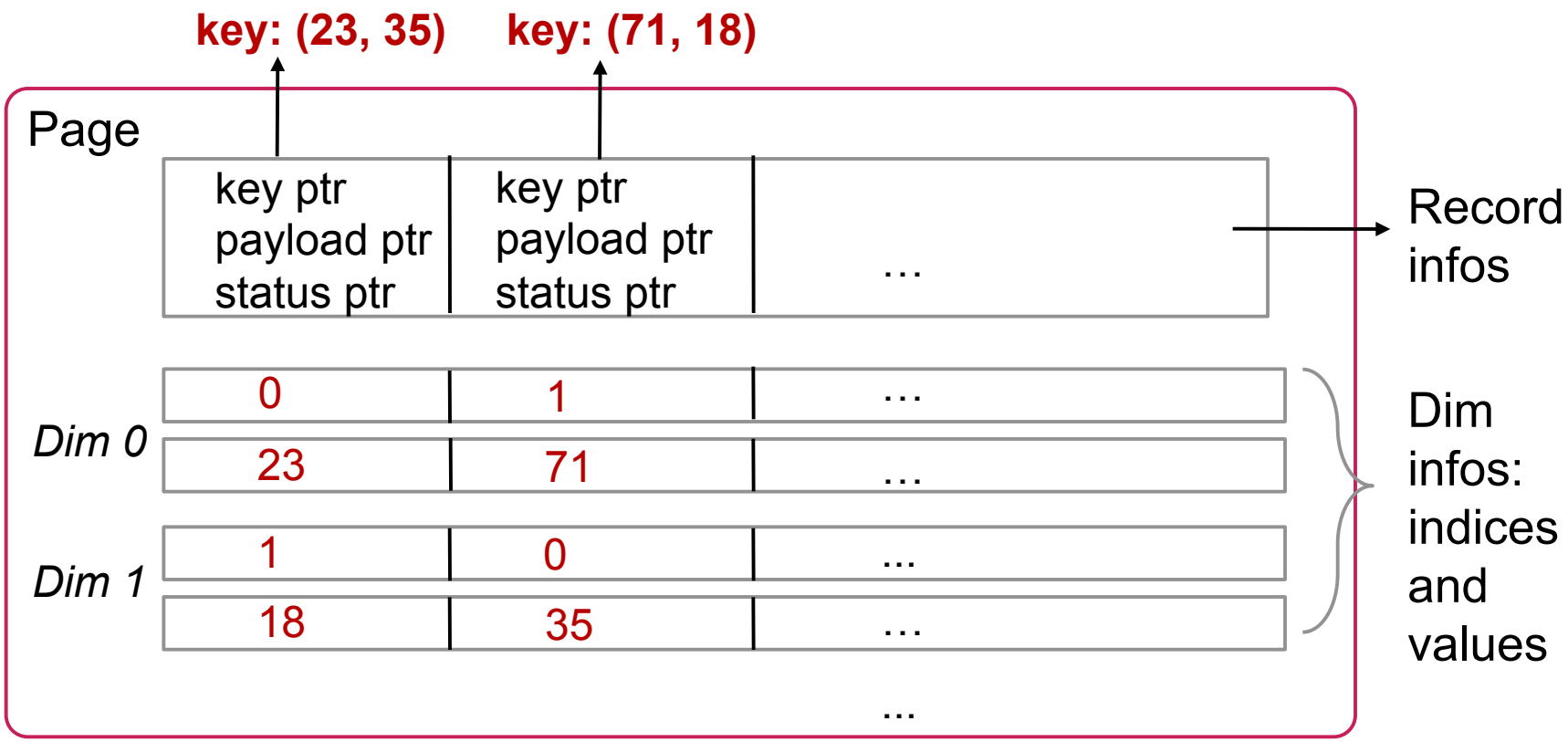
Search data structures: pages (1)



- RecordInfos: array sorted by global order holding pointers to keys and payloads
- Values: arrays sorted by the key values for each dimension
- Indices: arrays holding the RecordInfo indices corresponding to values

Search data structures: pages (2)

Example : 2 records, 2 dimensions



Sorted array insertion : find position, memmove, insert

Search data structures: pages (3)

Inserting in a page:

- Sorted array insertion in the RecordInfos array
- Indices update in the Indices array
- Sorted array insertion in the DimInfos arrays

Splitting a page:

- Upon reaching MAX_PAGE_SIZE → page split, new page creation
- RecordInfos and DimInfos are redistributed
- Sorted array insertion of the new page in the page array

Search data structures: queries

Handling inserts, deletes, updates, “no-wildcard” point queries:

- Binary search on the page array to find concerned page(s)
- Binary search on the RecordInfos array to find concerned record(s)

Handling range and wildcard point queries

- Use first dimension values and iterator's first value for the binary search on the page array
- Binary search on values array for each dimension
- Use indices array to fill a bitmask; results = set bits

Search data structures: conclusions

Strengths:

- Simple, fast binary search as the only search algorithm
- Results respect global ordering without further sorting on query time
- Reasonable insert cost due to pages
- Small, cache-adapted search structures

Limitations:

- Dimensions are treated individually
- Although filtering is fast on page level, we could benefit from a more efficient index structure on top of the page array

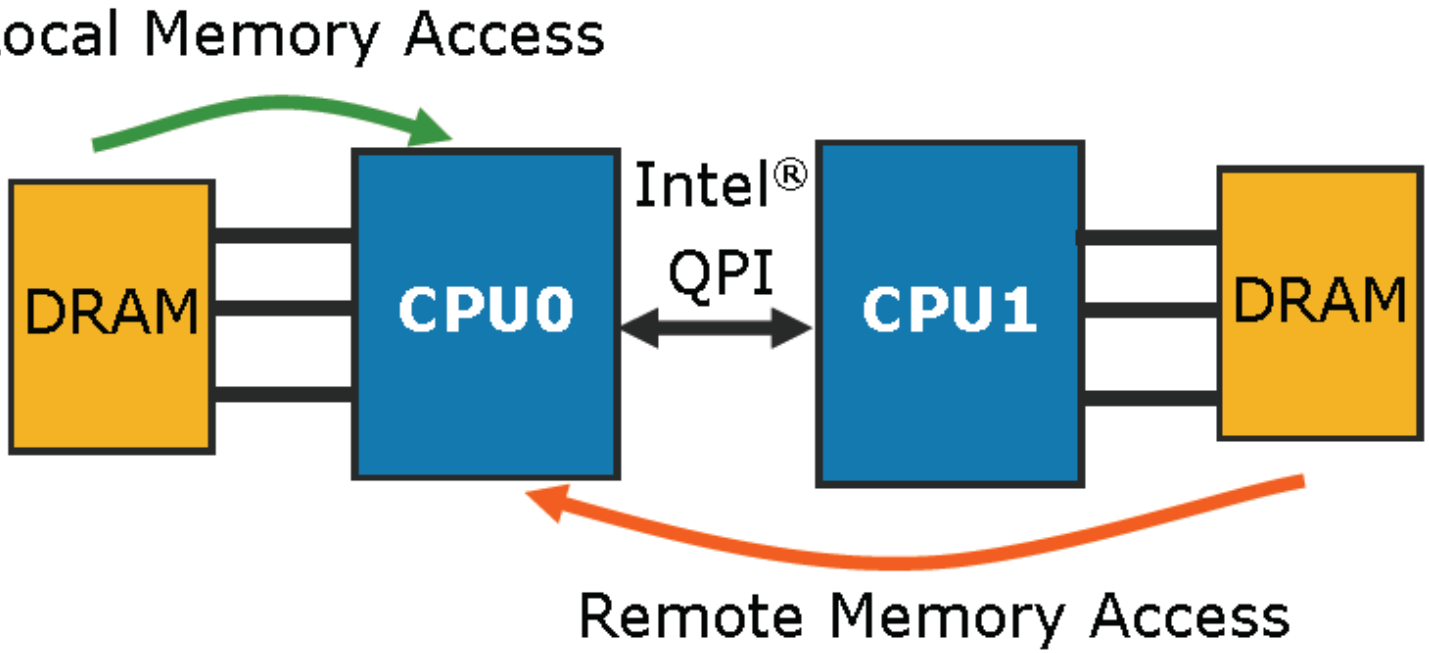
Target : simple, robust, scalable solution
Main challenges : parallelism, global order preserving

- Main bricks
- Search data structures
- Memory management

Memory management : issues

- Malloc → too costly (too complex); we need simpler allocation
- "Book" an amount of mapped memory; allocate by simply moving a cursor in constant time
- Use object pools: iterators, transactions, byte chunks for keys, VarChar data and payloads
- Cache optimization → indices and values in the DimInfos structure
- Tools: Valgrind/Cachegrind, VTune

Memory management : NUMA



Disadvantage of non uniform memory access -> should group threads on physical processors by their common memory accesses

1. Clearly separate allocated memory for each index
2. Dispatch even indexes (and their attending threads) on the first physical processor /NUMA node, odd indexes & their threads on the second

Memory management : conclusions

Platform specific optimizations are useful when dealing with large datasets

- Balancing memory access
- Data pre-fetching to avoid Last Level Cache misses
- Mitigate TLB collisions → same offset on different pages
- Avoid split-store → data spanning across several cache lines

Conclusions

Fast, simple, robust algorithms
+
Platform-specific optimizations
=
Good overall performance

Target reached : simple, robust, scalable solution
Still lots of potential for optimization!

Many many thanks to:

- **Damien Millescamps**, for his extensive technical advice and overall support
- **Bogdan Cautis**, for giving me time and encouragements to work on the contest
- **Lukas Maas** and **Thomas Kissinger**, for their great patience and reactivity