

An Overview of Future Improvements to OPSI

Francesc Altet

Cárabos Coop. V.



Urbana-Champaign 10/08/07

Features of PyTables Pro

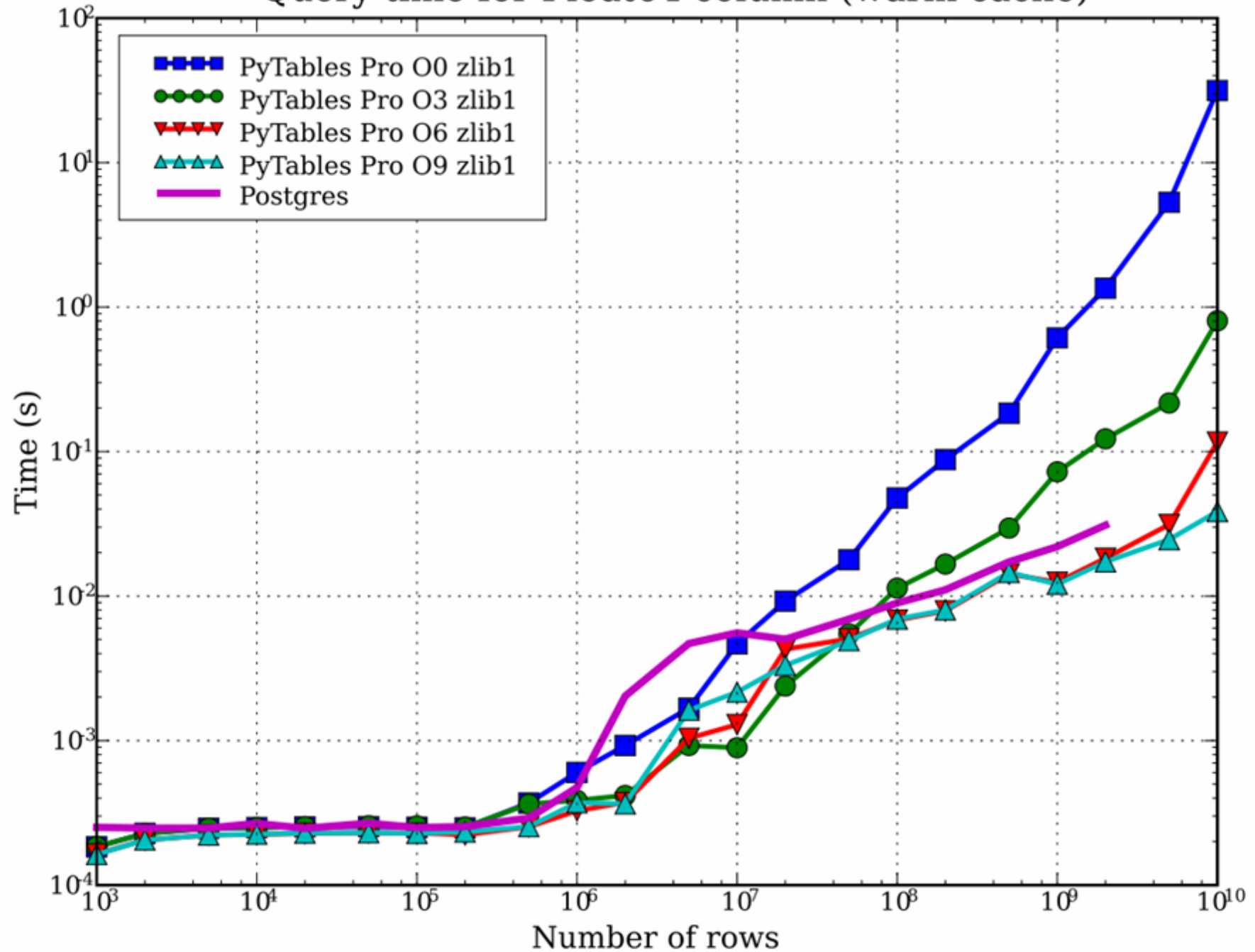
- OPSI (Optimized Partially Sorted Indexes)
 - Indexing engine optimized for HDF5 features (chunking, compression, data types)
- Improved LRU node cache performance (up to 20x faster than PyTables Standard)
- Focus on stability (meant for use in production environments)
- All-in-one installers for Windows and Mac OS X



OPSI Features

- Based on well-tested PSI engine (PyTables 1.x)
- Improvements over PSI
 - Better query times
 - Selectable index quality
 - Complex queries
- Current limitations
 - Only one index can be used in a complex expression
 - Only supports compound types, not atomic types

Query time for Float64 column (warm cache)



Plans for the Near Future

- Optimize the retrieval of results in queries with a large number of hits (low selectivity)
 - The current algorithm is quite efficient for medium or high selectivity, but less so for low selectivity
- Ability to use several indexes in complex queries
 - If col1 and col2 are indexed, then the expression $(col1 < 3.1) \ \& \ (col2 > 2.3)$ cannot be computed using both indexes (the first one will be used instead)

Low Selectivity Retrieval

- A table with 4 columns:

```
class Record(tables.IsDescription):
```

```
    col1 = tables.Int32Col()
```

```
    col2 = tables.Int32Col()
```

```
    col3 = tables.Float64Col()
```

```
    col4 = tables.Float64Col()
```

- 1 billion rows (1 Gigarow)

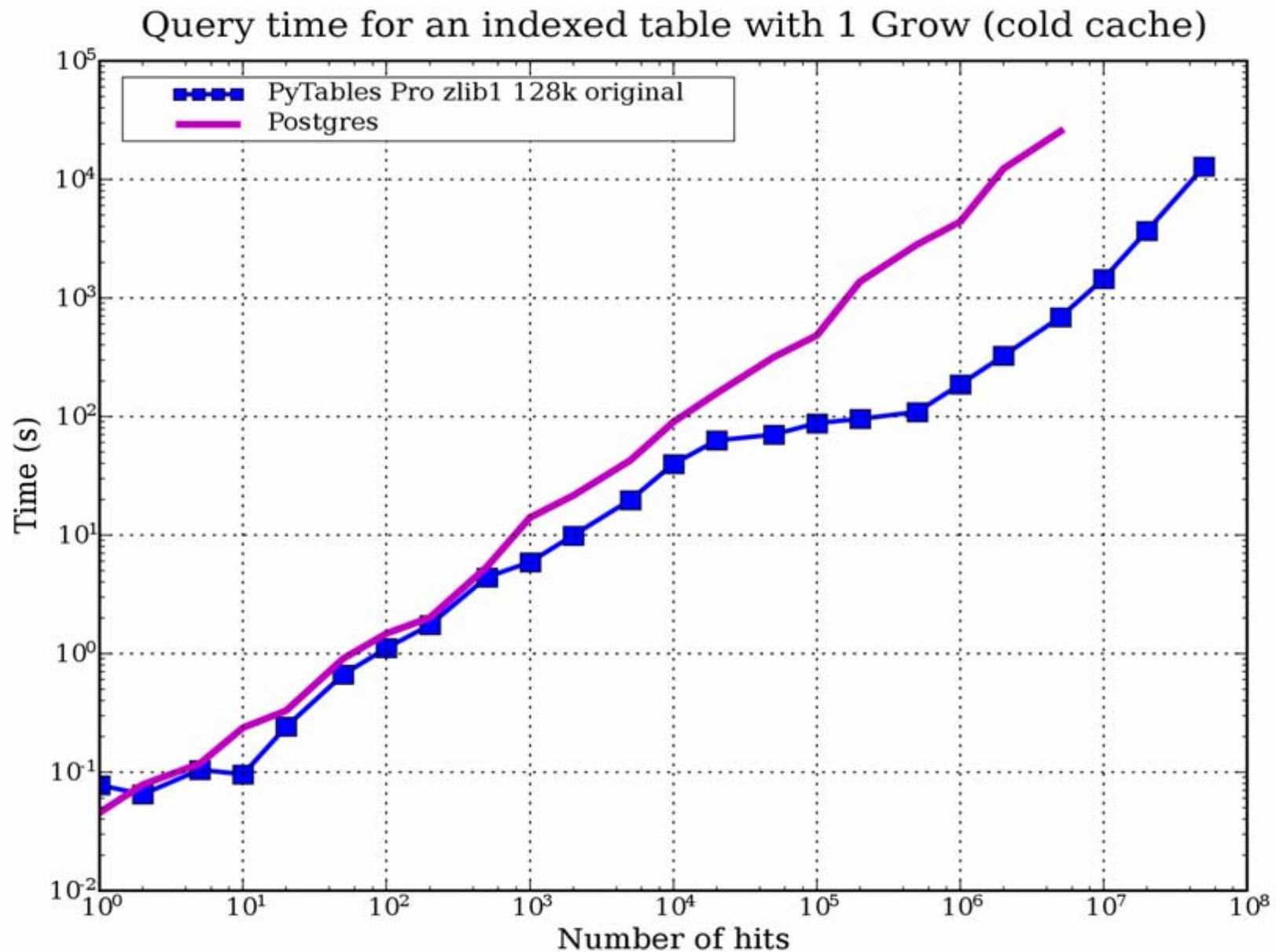
- AMD Opteron @ 2 GHz

- SATA disk @ 7200 rpm

- Query:

```
(lower<=col4) & (col4<=upper) &  
(sqrt(col1+3.1*col2+col3*col4) > 3) ↑
```

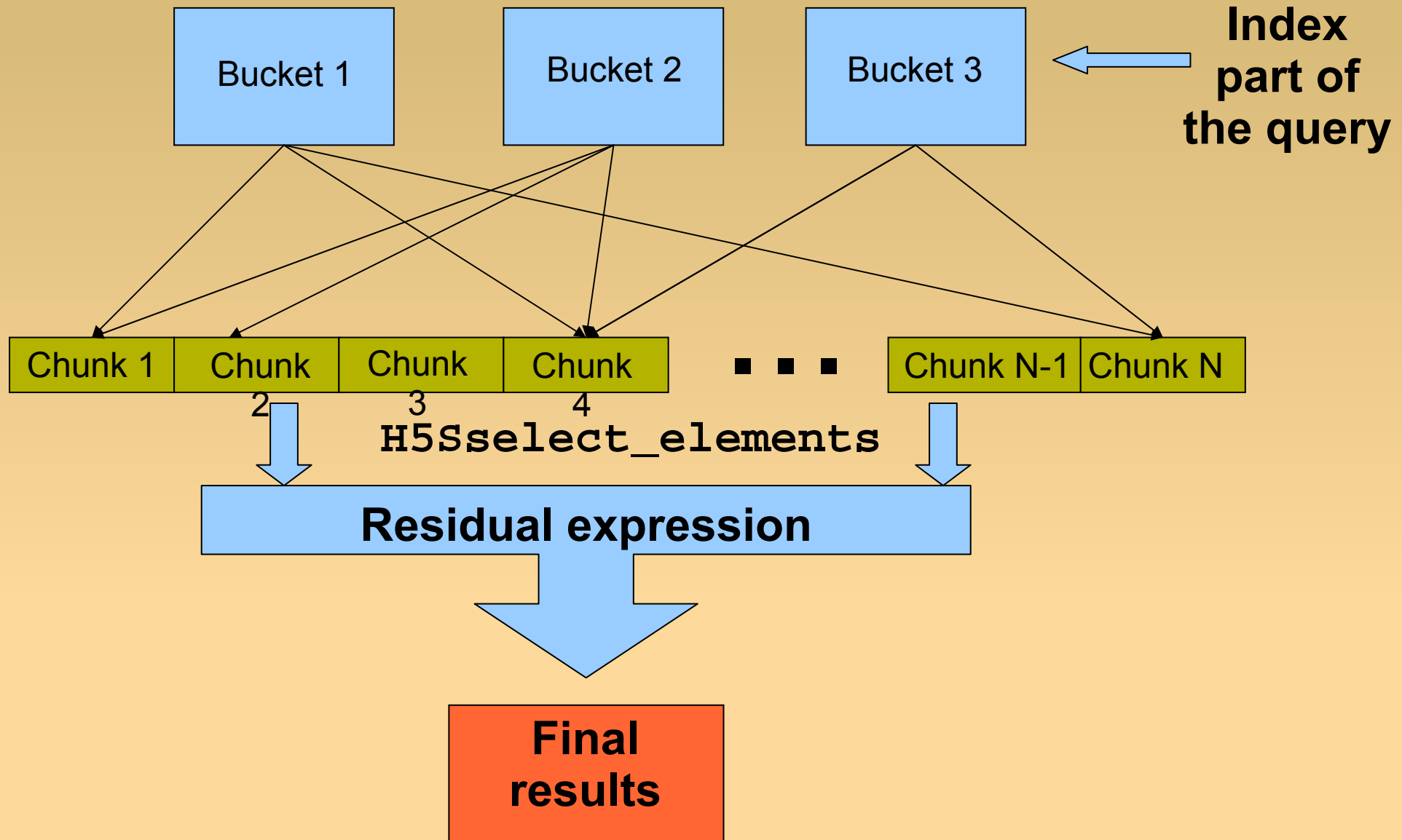
Low Selectivity Retrieval



Low Selectivity Retrieval

- Current approach:
 - Get the set of coordinates satisfying the indexed part of the query
 - Break the set into buckets and read a bucket at a time (using `H5Sselect_elements`)
 - Read the elements from disk and apply the residual query
 - Return the rows that satisfy the query condition

Current approach



Problems with the Current Approach

- Potential chunk revisiting (and very difficult to find the chunk in HDF5 cache because of capacity problems)
- Even if the chunk is found in HDF5 cache, it still has to be decompressed again
- Non-ordered access to chunks, resulting in longer disk access times

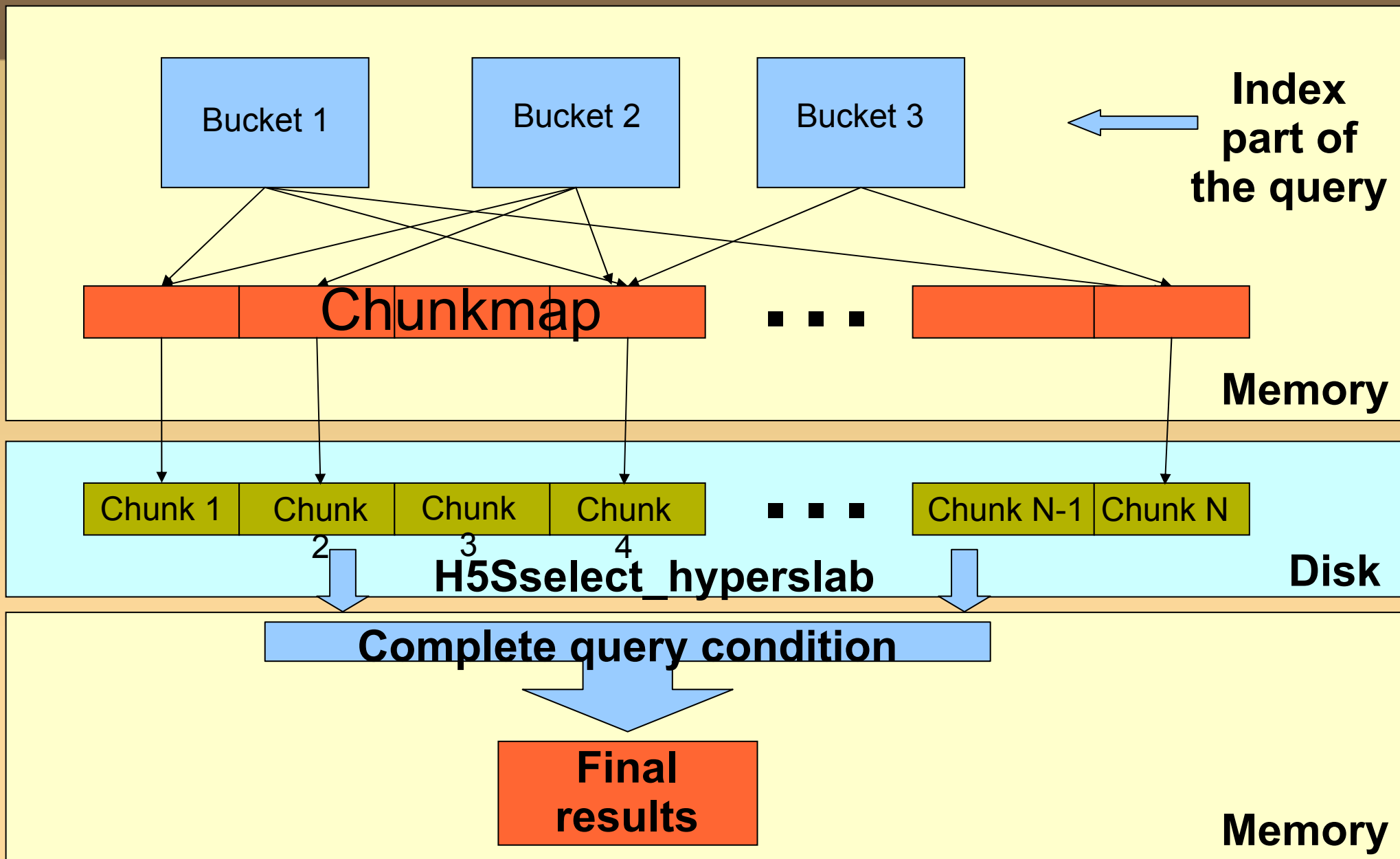
A Message from the Fifth Century, BC

“In general, commanding a large number is like commanding a few. It is a question of dividing up the numbers. Fighting with a large number is like fighting with a few. It is a question of configuration and designation.”

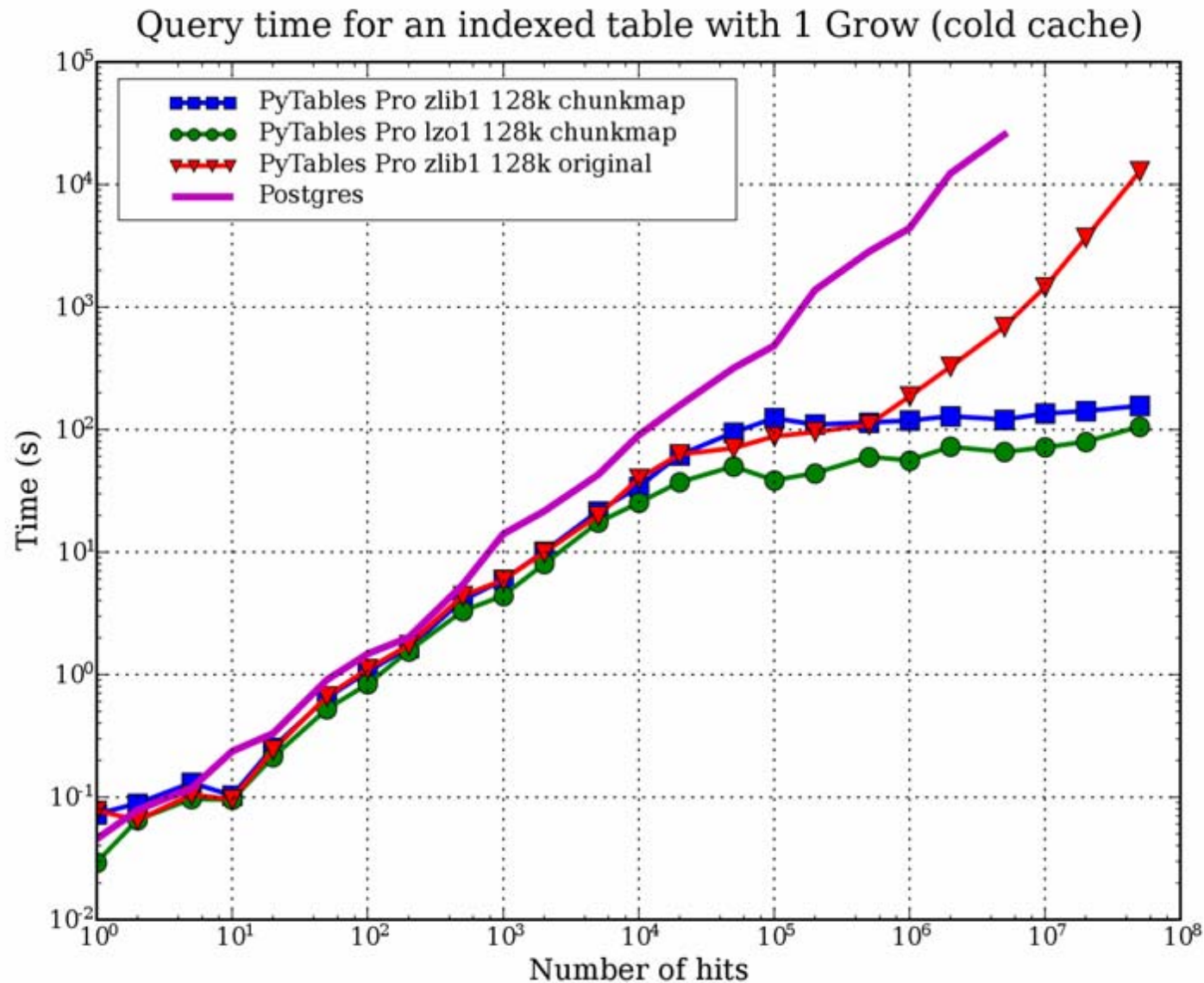
Sun Tzu – The Art of War

Section 5 (Strategic Military Power) verse 1

Solution: A Chunk Map



Chunkmap Performance



Chunkmap: Pros & Cons

■ Pros

- The interesting chunks are visited only once
- Chunks are accessed in a strict sequential order, minimizing the amount of trips of disk heads
- The chunkmap on disk has much lower entropy than the original indices: much better compression

■ Cons

- It requires memory: 1 byte per chunk. It can be up to 1 bit per chunk (packed chunkmap)
- It requires more CPU, as the incoming data from disk has to be filtered through the query condition

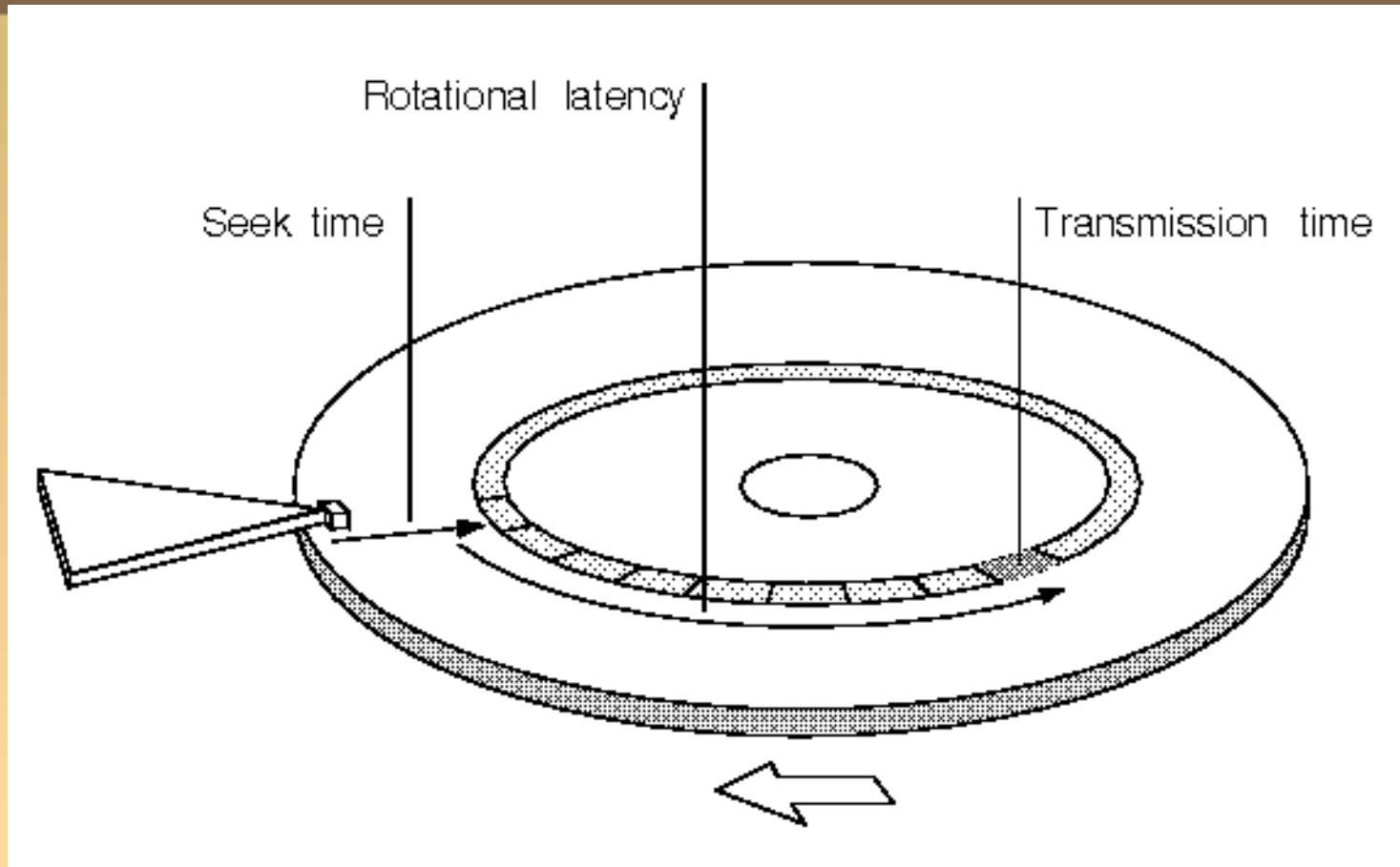
How Much Memory is Required by a Chunkmap?

- Let's imagine a table with 1 billion of rows and 1000 bytes/row. That's around 1 TB in size.
- Size of the chunkmap, depending on the chunksize:
 - 32 KB CS: 32 MB (4 MB packed)
 - 64 KB CS: 16 MB (2 MB packed)
 - 128 KB CS: 8 MB (1 MB packed)
 - 256 KB CS: 4 MB (0.5 MB packed)

Optimal Chunksize?

- What is the optimal chunksize for reducing the chunkmap to a minimum without penalizing retrieval times too much?
- We have to choose a size that takes a relatively short time to read compared with disk access times (the main bottleneck in sparse reads)
- What is the mean latency when doing sparse reads?

Typical Disk Access Times



Times for 7200 rpm drives



Average rotational latency: 4.1 ms
Seek times: from 2 ms to 18 ms

Typical Disk Access Times

- For general random sparse access data on disk, these figures usually give 12 ~ 15 ms
- However, for sequentially ordered sparse access of chunks that are close to each other, the typical times are bound by the rotational latency or less, i.e. ≤ 4.1 ms access times.

Optimal Chunksize (revisited)

- The optimal chunksize for reducing the amount of memory allocated to the chunkmap has to be chosen so that reads would constitute a relatively small fraction of the average rotational latency of a disk
- The most significant cost in time to process the chunk is the sum of:
 - The time to physically read it from disk
 - The time to uncompress it
 - The time to apply the query condition to it

Times to Process a Chunk

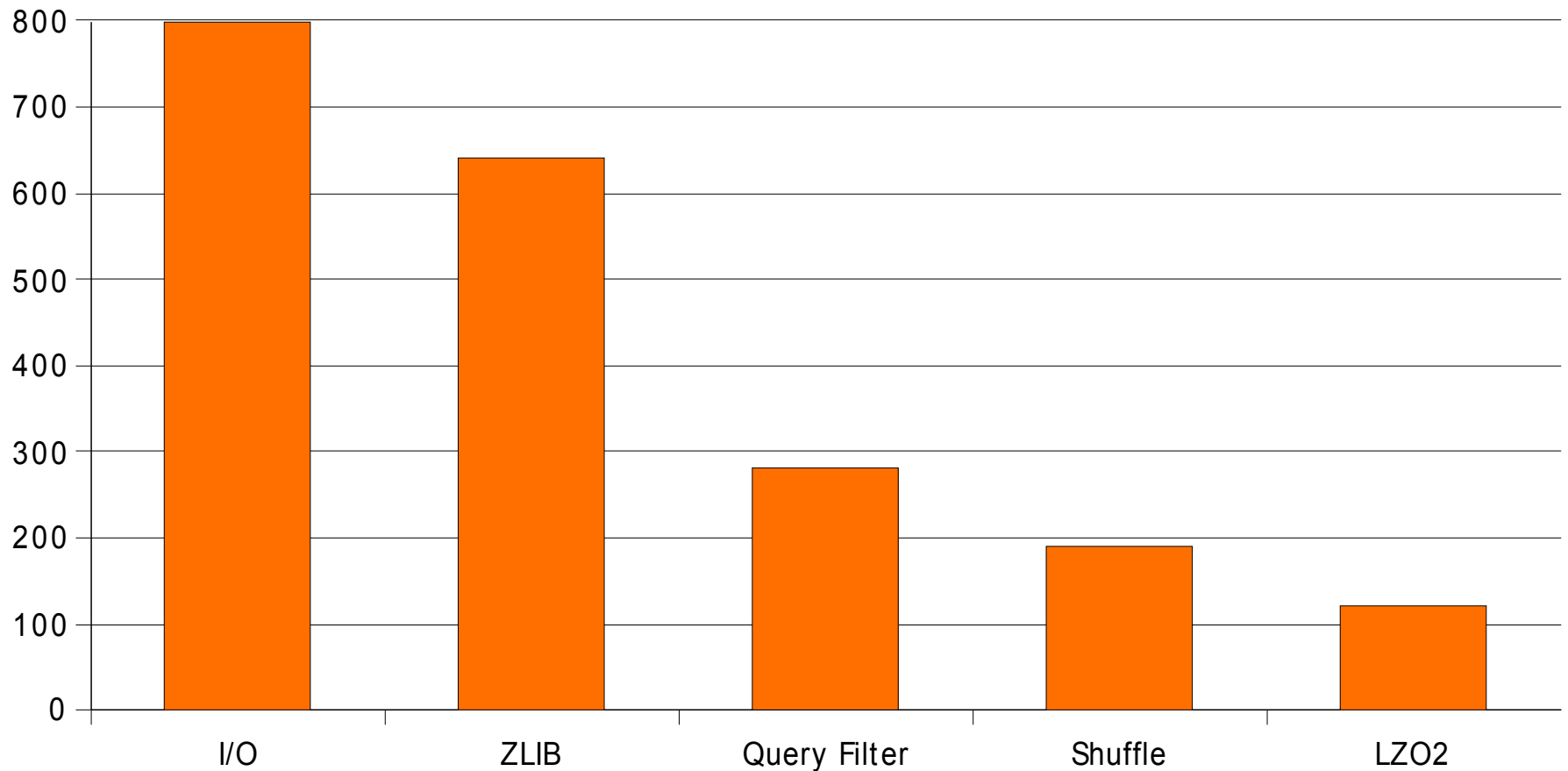
- This depends on many factors. For an example, we will choose:
 - Chunk size: 128 KB
 - Compression on (225% of reduction)
 - Modern 7200 rpm SATA disk drive
 - Modern CPU (Intel Core2 or AMD Opteron)
 - Query Filter:
 - $(\text{lower} \leq \text{col4}) \ \& \ (\text{col4} \leq \text{upper}) \ \& \ (\text{sqrt}(\text{col1} + 3.1 * \text{col2} + \text{col3} * \text{col4}) > 3)$

Times to Process a Chunk

Using ZLIB: 1.8 ms

Using LZO2: 1.3 ms

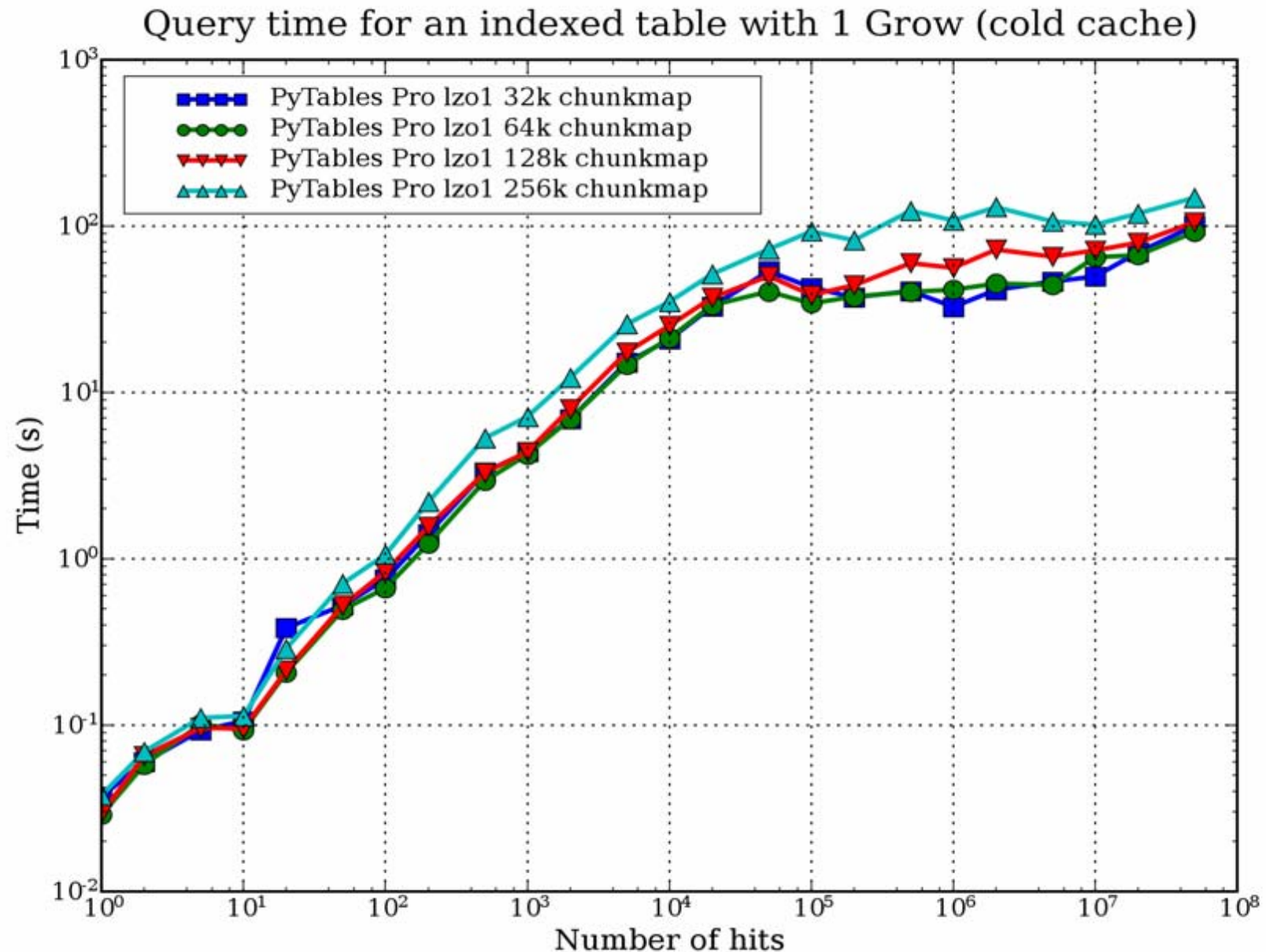
Times (μ s) for a 128 KB chunk (57 KB compressed)



Times for Different Chunksizes

- Times and overhead for low selectivity:
 - 32 KB: 0.45 ms, 11% overhead
 - 64 KB: 0.90 ms, 22% overhead
 - 128 KB: 1.8 ms, 44% overhead
 - 256 KB: 3.6 ms, 88% overhead
- 32 KB or 64 KB would be a good choice for increased low selectivity retrieval speed
- 128 KB would strike a good balance between overhead (44%) and the memory used by the chunkmap (8 MB, or 1 MB packed)

Times for Different Chunksizes





Some Considerations

- The query conditions are evaluated very efficiently thanks to the NumExpr computing kernel integrated into PyTables
- Compression reduces the total I/O time. Not new, but interesting anyway
- The use of LZO2 compressor can be very effective in this scenario (as compared to ZLIB)
- Shuffle takes longer than LZO2, but is worth the while: compression is much higher

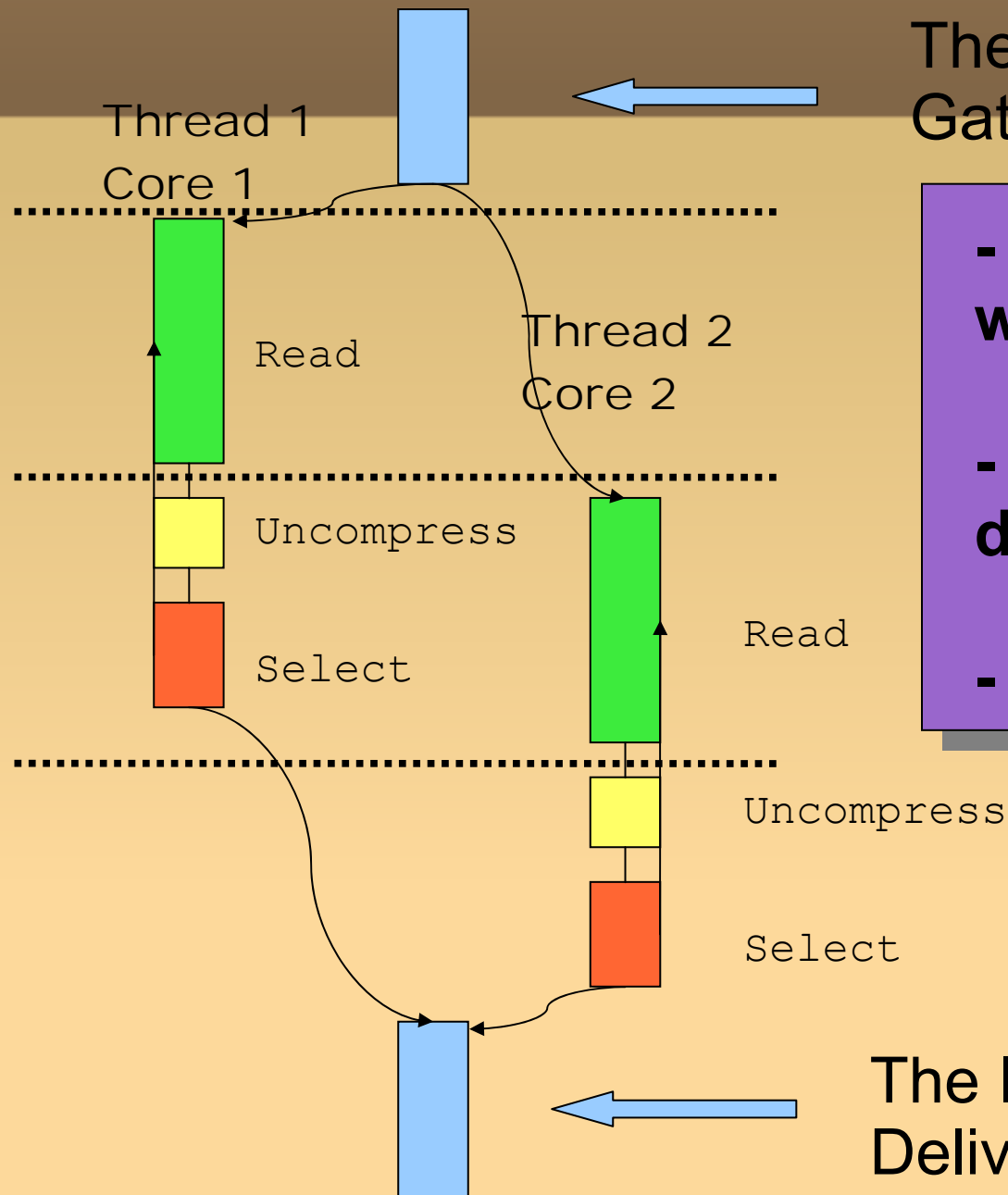
A Few Words About NumExpr

- Fast evaluation of array expressions element-wise by using a **vector-based** virtual machine
- It works by splitting up the operand arrays in chunks that fit into the cache of CPUs, allowing the CPU to attain very high-performance while performing the operations
- We have added support for boolean and string types, heterogeneous arrays (compound types), and optimized the amount of memory copies of unaligned arrays

Using MultiCore CPUs

- Nowadays, it is possible to use multicore CPUs and concurrent programming with threads to further accelerate the reading process in low selectivity environments

MultiCore & Threaded Disk Access



The I/O buffer is empty
Gather more data

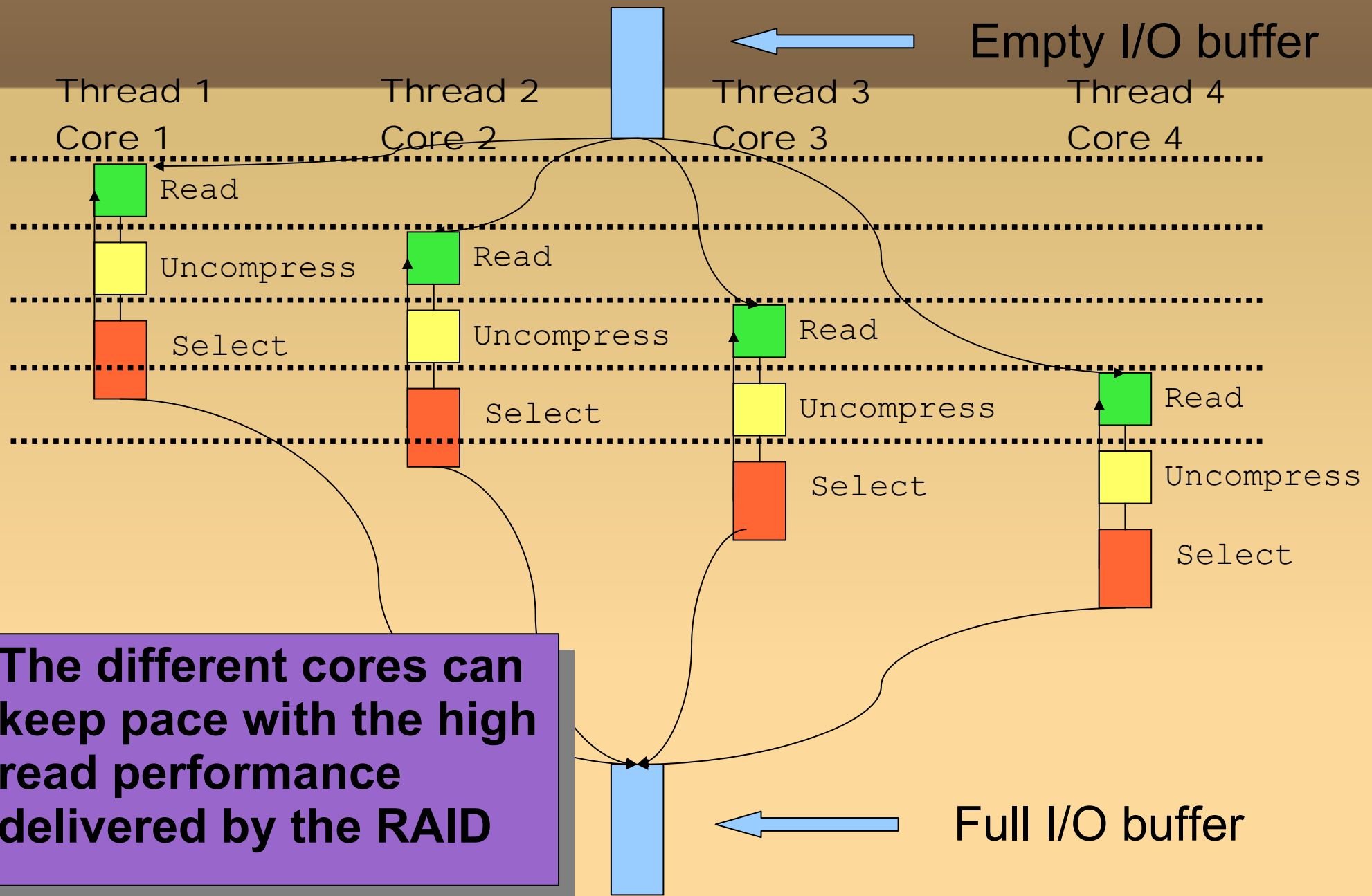
- The computations overlap with I/O
- The only bottleneck is disk speed
- Up to 1.3x speed-up

The I/O buffer is full
Deliver elements to Python space

Multicore & RAID

- With the advent of multicore CPUs, having a 2, 4 or 8-core system is not uncommon in current workstations
- In addition, drastic reductions in the cost of a medium-sized disk (500 GB costs about \$120), makes it possible to build cheap but fast RAID systems reaching multi-TB of capacity
- This system configuration should be considered the norm right now!

Multicore & RAID

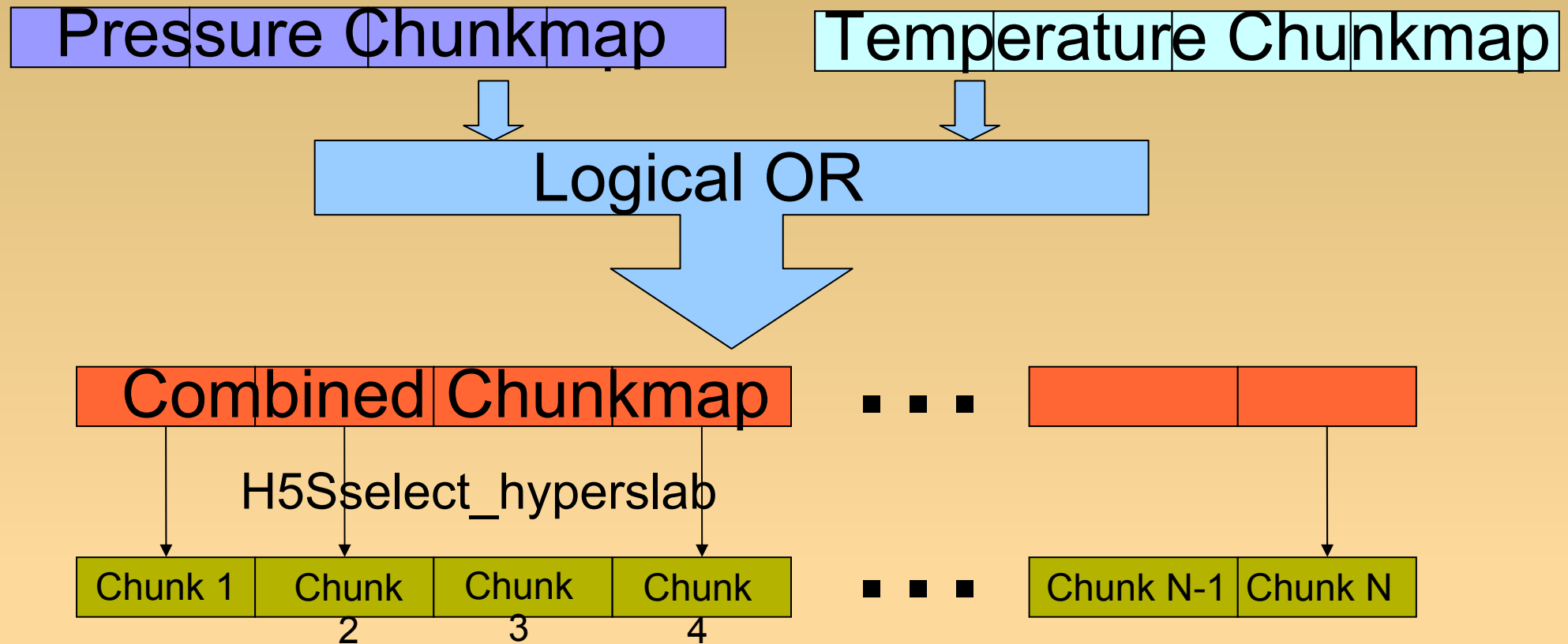


Using Several Indexes in Queries

- Perhaps the most appealing use of chunkmaps is that they can utilize several indexes on a single query
- Examples:
 - `'(pressure < 20) & (temperature > 50)'`
current OPSI is not able to use the indexes simultaneously
 - `'(pressure < 20) | (temperature > 50)'`
current OPSI can't use any index (because the conditions are 'ORed')

Using Several Indexes in Queries

■ '(pressure < 20) | (temperature > 50)'



Using Several Indexes in Queries

- NumExpr will be used to combine any amount of logical combinations among chunkmaps
- **Challenge:** From a potentially complex query expression such as:

```
((pressure < 20) & (temperature > 50) |  
  ((lati > 20) & (lati <=40) & (longi < 30))
```

find the maximum number of usable indexes
- This can represent a fair amount of work for very complex expressions!
- Start with the simplest ones and refine the query optimization as needed (not new)



Medium/Long Term Goals

- Try reducing the precision of values of the indexes
 - Faster convergence during index creation
 - Less entropy: better compression, less disk space
 - Inexact results in queries
- Column-wise tables
 - Current table datasets in PyTables are row-wise
 - They are perfect for dealing with tables with a small/medium number of fields
 - Column-wise may prove to be more efficient in scenarios where a large number of fields is required



Final Thoughts

- Chunkmaps seem like a good idea for OPSI
 - They perform much better when the selectivity is low, while retaining the same efficiency for high selectivity queries
 - They permit the use of several indexes in complex queries without too much effort (not taking into consideration the battle to optimize queries!)
- Precision reduction seems easy to implement
- Column-wise tables can be very interesting in some scenarios, but implementation could be difficult

Appendix: OPSI vs MRBM

- After reading the paper on Multi-Resolution Bitmaps, I decided to make a quick comparison between it and OPSI
- WARNING! These conclusions are **very** preliminary and I may be completely wrong!

Main Characteristics

■ OPSI

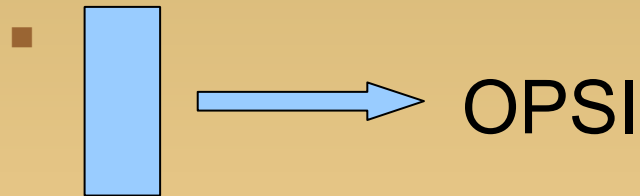
- Better suited to tables that have fewer fields ($< 10 \sim 15$) and more rows (no practical limit)
- I/O bounded (both in latency and throughput)
- Needs less space for each index
- It supports strings right out-of-the-box

■ MR-BitMap

- Better suited to tables with many fields (> 100) and fewer rows (< 1 billion?)
- CPU bounded
- Requires more space for indexes

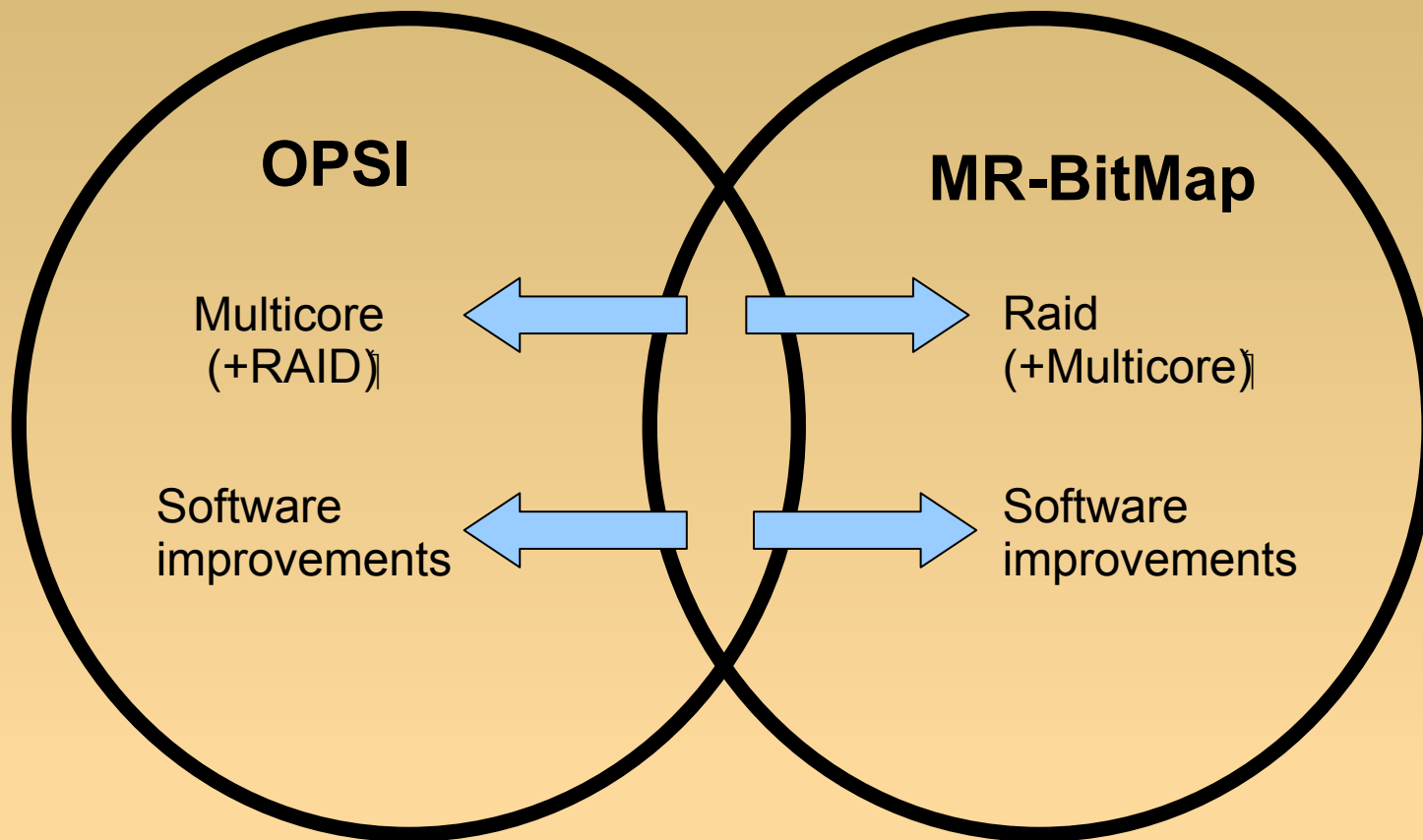
Which One Is Better?

- It depends on many factors



- Low cardinality: MR-BitMap, OPSI
- High cardinality: OPSI, MR-BitMap
- Many others!

Use Domains



Suggestions for a More Detailed Comparison of OPSI/MR-BitMap

- If we were to undertake an implementation of OPSI in C, perhaps it would be wise to do a detailed comparison of OPSI and MR-BitMap first
- Some ideas from one approach can be applied to the other!
- A repository of real-world data for testing purposes would be useful. It would provide a common base of data for determining the suitability of each approach in each context