OPSI: The indexing system of PyTables 2 Professional Edition

Francesc Altet i Abad
Ivan Vilata i Balaguer

Cárabos Cooperativa Valenciana

July 11, 2007

In a world that is continuously providing enormous amounts of data (be it from satellite teledetection, industrial/biological sensors or historical data for financial use, among many other sources), the need to quickly search through it and discover its interesting trends is becoming more and more important. Indexing is the process that allows to extract the interesting bits of information out of these oceans of data in a very short time, thus making possible the analysis of huge amounts of data.

PyTables is a hierarchically-organized database that is focused on dealing with such large data scenarios. In PyTables 1.x series, indexing was implemented using a simple approach, so-called PSI, that worked fine for medium sized tables, but that was not very efficient for dealing with large ones. With the advent of PyTables 2.0 Pro, we are introducing OPSI, a completely new implementation of the indexing engine that significantly improves look-up times for all sizes of tables, and most specially for very large ones (> 100 millions of rows). In addition, the new implementation provides an innovative feature that lets the user create indexes having a certain level of quality, allowing her to select the one that is best suited to her particular needs.

This article describes the implementation of this new indexing system, while showing experimental figures about its performance.

1 Introduction: PSI (Partially Sorted Indexes)

PyTables [1] is a Python [2] package that provides a series of capabilities for dealing with extremely large amounts of both homogeneous and heterogeneous data in a simple and effective way. It leverages the powerful and easy-to-learn Python language and the features that are available in the HDF5 library [3] and the NumPy package [6] in order to achieve this goal.

A technique that has proved to be very effective for searching through those large amounts of data is *indexing*. When properly used, it allows to reduce lookup times in huge tables from the several hours that would take a regular sequential search to just some tens of milliseconds. This technique is commonly used in relational database engines like Postgres [4], but there are also implementations over the HDF5 library like FastQuery [5], which uses a radically different indexing approach than the one presented here.

When indexing capabilities were first introduced in PyTables 1.x series, the main goals at that time were:

- To index very large datasets by using reasonable amounts of time and resources.
- To compress indexes in an efficient way.

After considering several alternatives, the decision was to implement a very straightforward algorithm: break the dataset in *slices* and sort each slice completely, saving the sorted values (together with the original indices) in separate datasets in the same file. The slices were made of *chunks* of the same size that could be compressed efficiently by the underlying HDF5 library. The layout of this implementation can be seen in figure 1.

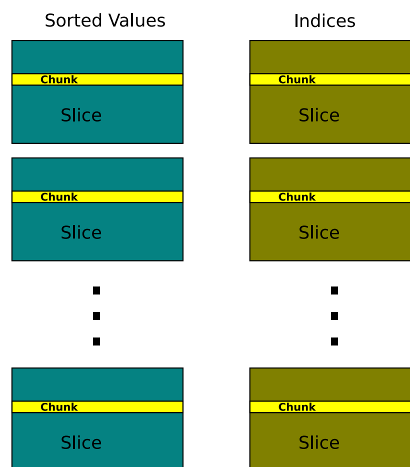

Figure 1: The layout of index data in 1.x series.

The algorithm to do searches using this system was also straightforward: do a binary search for the limits on each slice, extract the selected values, and repeat this procedure for all the slices.

This algorithm doesn't have to completely sort the dataset so as to complete the index; this is why it is called a *Partially Sorted Index* (PSI). It is easy to see that this approach takes a time to create the index that grows in a completely linear way with the length of the dataset (for large enough ones), and, as said before (*use a reasonable amount of time*), this was a key factor for choosing it. Unfortunately, this also has an immediate corollary: the time to search also grows

linearly, i.e. the advantage of binary searches in slices is lost for large enough datasets. To worsen this, the original implementation didn't index the trailing values in the original dataset that were not enough to fill a slice, and they had to be searched using a traditional sequential data traversal algorithm. So, in cases were the trailing number of values were near (but not equal) to the slice size, the total time for the look-up could potentially be dominated by the sequential search times of traiiling values rather than the sum of the binary search times for the rest of complete slices.

Finally, PyTables 1.x was not able to use indexes when doing complex queries (i.e. those with conditions on several columns or including mathematical operations). Although this was more a limitation of the *implementation* than an *intrinsic* limitation of a PSI system, the fact was that the only queries that had an opportunity to use a PSI index were the simpler ones (i.e. of the form `'low < column <= high'`).

To summarize, here are the pros and the cons of PSI indexes:

Pros

- Very fast indexing.
- Near linear dependency of index creation times with dataset sizes.
- Possibility to compress indexes.
- Very simple, and hence robust, indexing algorithm.

Cons

- Near linear dependency of query times with dataset sizes. This could lead to unacceptable response times when querying large datasets.
- The fact that trailing values of the dataset don't get actually indexed can potentially increase the total look-up time quite a little.
- Not possible to use indexes when doing complex queries.

2 Introducing OPSI (Optimized Partially Sorted Indexes)

In order to overcome most of the limitations of PSI, the PyTables 1.x indexing system, a new, much improved engine has been developed and introduced for the first time in PyTables 2.0 Pro [7]. The new system, so-called *OPSI* (Optimized PSI), is also based on a PSI schema (i.e. the indexes are partially sorted as well), although with some critical enhancements:

1. Added a couple of cache tables in order to faster locate the slices with interesting information.
2. The trailing values of datasets are indexed now (i.e. a binary search can be performed on them).

3. A series of *LRU (Least Recently Used) caches* have been implemented at several levels of each index, allowing much faster response times for queries hitting similar places of an index.

4. An algorithm has been implemented in order to reduce the level of "unsorted" values, i.e. effectively reducing the "entropy" of indexes. In other words, the algorithm improves the *quality* of indexes. In addition, such a quality level is user-selectable.

5. And last, but not least, a new query interpreter has been added supporting complex queries, that is, queries that include common mathematical expressions or conditions over several columns.

Enhancements 1, 2 and 3 do dramatically improve the performance over its 1.x series ancestor. Enhancement 4 allows the user to choose the most appropriate index quality to her needs; this represents a serious advantage over the traditional "completely sorted index" approach that most relational databases in the market implement, as we will prove later on. And finally, enhancement 5 allows a query flexibility comparable to implementations found in relational databases.

In the next sections, we are going to describe each of these new enhancements in order to better understand its implementation. However, and prior to explaining them, it would be useful to get an idea of the layout used for keeping the index information in OPSI.

First of all, and as was already mentioned, OPSI uses a partially sorted indexes (PSI) approach, that is, a series of *slices* that are completely sorted and that are composed, on their turn, by *chunks* of elements. Moreover, a couple of logical structures have been added for OPSI: *blocks* and *superblocks*. The later are just collections of slices, and we will see which role they are playing in the context of reducing the entropy of the index in section 2.4. The new layout is depicted in figure 2.

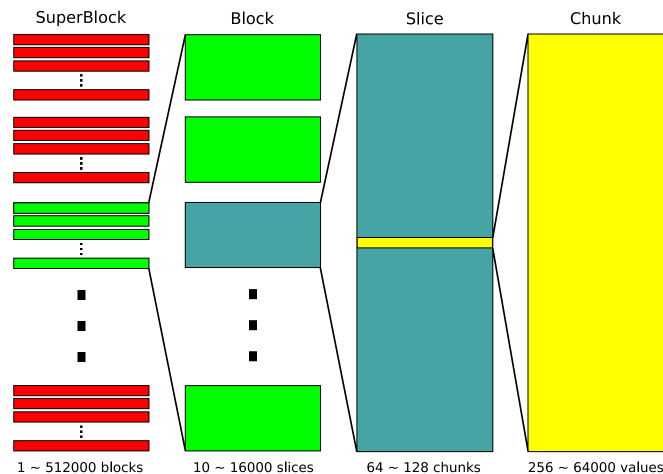

Figure 2: The layout of an OPSI index (only the set of sorted values is represented). A continuous rectangle means that values inside it are guaranteed to be completely sorted.

Having introduced the OPSI layout, we will proceed to describe the other components of this indexing system.

2.1 Addition of persistent cache tables

As explained in section 1, the general procedure to do searches in a PSI index is to perform a binary search for the limits in a single slice, extract the selected values, and repeat this procedure for all the slices. So, we can see how looking at the values in slices is a very common operation. Because of this, and in order to reduce the number of I/O operations, a couple of cache tables have been implemented for OPSI:

- **Level 1 cache** (or *slice* cache): keeps the begin and the end values for each *slice*.
- **Level 2 cache** (or *chunk* cache): keeps the begin values for each *chunk*.

The layout of both cache tables can be seen in figure 3.

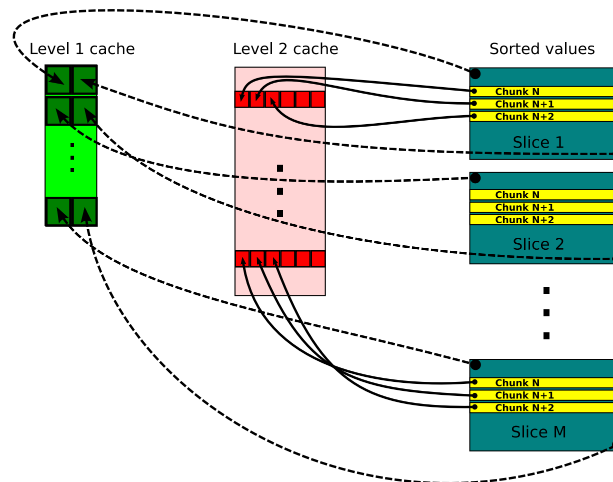

Figure 3: Layout of 1st and 2nd level caches.

The level 1 cache avoids doing a binary search on a complete slice if the interesting set of values fall out of the begin/end range of the slice. On its hand, the level 2 cache avoids complete chunk retrievals in case the interesting values fall out of the begin/end range for the chunk. So, when working together, these caches do effectively avoid much I/O effort. In addition, if the quality of the index is high (i.e. its entropy is low), their function is critical in that they avoid a lot of uninteresting visits to most slices and chunks, hence reducing query times to a bare minimum.

2.2 Indexed trailing values

When using and indexing system that is based in sorted slices that are of the same size (as the one being described here), there is an intrinsic problem with the trailing values that don't fit on a complete slice. In plain PSI, we have already seen that these trailing values were not indexed at all, and that a pure sequential search was needed so as to process them. In the new OPSI indexing system, a separate dataset for indexing such values has been implemented and more logic has been added in order to make possible a binary search on the trailing values as well. This allows to completely remove the overhead of the aforementioned sequential searches for trailing values.

With this, the time for executing queries is practically equal for datasets that are an exact multiple of the slice size than those that are not. This effectively allows a smoother response in query times for all ranges of dataset sizes.

2.3 Intelligent LRU caches

In order to obtain really fast query times for queries that exhibit some kind of repetition pattern, be it spatial or temporal (and if lots of queries are performed, it would be rare that such a repetition pattern does not appear), there is no replacement for using an in-memory cache. Such a cache would keep the results for previous queries and their information quickly reused in case they are needed for computing the current query.

However, caches are very tricky objects, and one has to be careful of not keeping every piece of data passing through them if one doesn't want to exceed memory requirements. There are many ways of choosing the piece of data in cache that has to be disposed of in order to make room for new-coming data when the cache has reached its full capacity. In this case, a LRU (Least Recently Used) algorithm has been implemented in order to take advantage of the most frequently visited data.

Also, chances are that a cache doesn't get any hit (or very few of them) during a series of queries. In this case, the cache is wasting a precious time collecting data that ultimately won't be useful at all. In order to avoid this, OPSI index caches feature the capability of *auto-sensing* their efficiency, so that if it doesn't reach a certain threshold, the cache is disabled automatically. In case a cache got disabled, it will be re-enabled sometime afterward so as to probe whether it can be useful again for newcoming queries or not. This smart mechanism allows to automatically keep enabled only those caches that are working efficiently, lending to a much rational use of CPU and memory bandwidth resources.

Several of these caches have been set up in order to be able to catch repetition patterns in most of the hot spots in the I/O chain. Moreover, they have been implemented in a hierarchy that allows very quick response times when the high level cache has the desired data, while still allowing pretty good time improvements when one has to descend the hierarchy so as to find the interesting bit of information.

The complete set of implemented caches for OPSI is listed at appendix A.

2.4 Selectable quality of the OPSI indexes

Perhaps one of the most distinctive features of OPSI indexes is the fact that the user can select the degree of *entropy* of an index (see appendix B for a precise description on how *entropy* is defined here), or put in another way, the *quality* of the index.

In order to achieve this, OPSI implements a basic algorithm that allows to reduce the degree of entropy of an index by exchanging data inside its *blocks* and *superblocks* (see figure 2), so that data values that are similar get closer on each iteration. The algorithm comes in a couple of flavors, one that reorders slices lying at the same superblock, and another that reorders data chunks that lie at the same block. Figures 4 and 5 depict both processes.

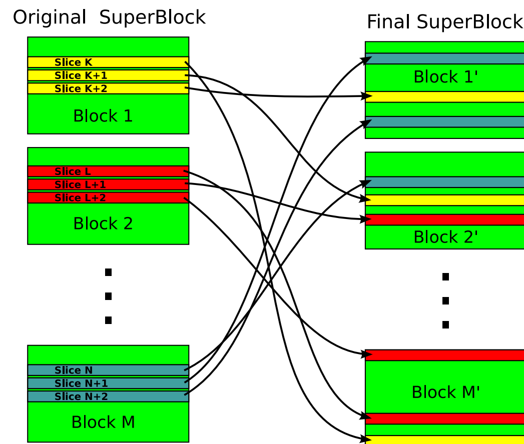

Figure 4: Entropy-reduction process for slices in an index.

As already said, the rearrangement of values works by placing the elements showing greater *affinity* (i.e. having similar values) closer and closer. In order to do this, several methods have been implemented, namely, using the *begin*, the *end* or the *median* value of each slice/chunk. For example, if the *begin* value method is chosen for a slice reorganization (see figure 4), the order between slices in the final superblock is determined by sorting the begin values for each slice in the original superblock and copying the slices to their destination following this new order. Something similar happens during the chunk reorganization with the difference that the final block is built by re-sorting the slices of the intermediate block (see figure 5)[1].

Perhaps an analogy can make the entropy reduction process clearer. Picture a world-wide Physics Olympics contest with a couple of *important* rules:

[1]Remember that, in a PSI schema, individual slices always have to be completely sorted. Incidentally, this later re-sorting of slices also contributes to a further reduction of the entropy.

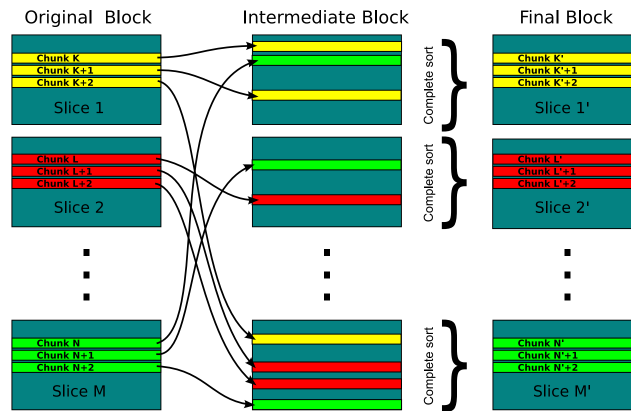

Figure 5: Entropy-reduction process for chunks in an index.

1. The outcome of the contest must assign an score for *each* participant.
2. For budget restrictions, only one event scoring can be done in one single room at a time.

Initially, the different participant *schools* (slices) can have several *teams* (chunks) whose composition can change on each event scoring. In addition, the different schools are grouped by *country* (block) and *continent* (superblock). Now, in order to decide a fair score for most of the individuals the judges have decided the next strategy:

They start celebrating an event on each participant school and they get a first classification on teams (the best one in first place, the following best in second place, and so on), and inside each team, they decide a classification as well. After this has been done for each school, they will celebrate a national event in every country in order to obtain a classification for the individuals that are representative of every team (usually the best ones) in the country, so that they can fit in a single classroom. Then, they will use this new national classification so as to send the representative's complete team to the different schools in order. After that, the judges will take individuals that are representative for each country of a continent and will do an international event on each continent in order to come up with a first international classification on each continent. Then, for each continent, they reorganize complete schools based on the international classification of school representatives.

By now, they already have a classification that fulfills the two rules specified above, altough it can be far from perfect (it has a potentially high entropy). At this point, if the judges want a better classification, they may want to continue doing as many national/international event iterations as they want until one of these events would happen:

- The classification doesn't move significantly on each iteration (this means that the classification is good enough, or put in another words, its internal entropy is low enough).
- The time or the budget for the contest exceeds its limits.

As it can be noticed, by using this algorithm, one can achieve good enough score (reduction of entropy) for the budget (CPU and memory) that the organization (the user) is willing to invest in the process.

Of course, and returning to the OPSI scenario, a reorganization of slices taken as entire entities, despite the fact that it effectively reduces the entropy of the index in absolute terms, is not of any help for the searching algorithms because an internal re-sorting of eash slice is not happening at all. This is why the slice reorganization is only performed as a pre-conditioner of the chunk reorganization: it facilitates the relocation of similar (in the sense of having similar *values*) chunks that fall in different blocks. By issuing several iterations of successive slice and chunk reordering, one can substantially reduce the entropy of the index and, in many cases, eventually achieve a null entropy (i.e. a completely sorted index), specially for high optimization levels and datasets up to 10^{10} rows (this is an experimentally tested figure, but it can very likely be higher than that), as we will see in section 3.

The reduction of entropy as a function of the optimization level of the OPSI algorithms can be graphically devised in figure 6. With an optimization level of 0, the only operation done is basically a sort of every *slice* in the index. Although this creates an index that can already be used for binary searching purposes, its values do still show a rather high ratio of dispersion and hence many slices have to be searched (queries take more time). When using optimization level 3 we see that values still show some degree of dispersion, but much less than with level 0. Finally, optimization level 6 achieves a completely sorted index (i.e. entropy 0) for this particular dataset, that allows performing the quickest searches, as will experimentally be shown in section 3.

Finally, it is worth noting that chunks lying in different superblocks will never have the opportunity to be put in the same slice (in the analogy above, you don't end with a world-wide winner, but with 5 continent-wide winners instead). So, the implemented algorithms are useful for effectively reducing the entropy at the level of one single superblock. However, this is not a limitation in practice, as a superblock is able to keep, by default, up to 2^{43} (around 8×10^{12}) elements[2], which is a reasonably high figure for nowadays systems.

As a consequence of this, and for indexes of datasets larger than a superblock, a separate search has to be carried out on each superblock. Indeed, superblocks are the ideal candidates to implement a parallel version of OPSI, for both creating and searching the index (see section 5).

2.5 Fine tuning of the size of building blocks

As we have seen in figure 2, the index values are broken down in relatively small *slices*. On its turn, each slice is made of small *chunks* of data. As it can be guessed, the sizes for chunks and slices are critical for achieving good search performance, so many experiments have been carried out in order to determine the best values for the slice and chunk sizes for the different ranges of dataset sizes.

Choosing the optimal *slice size* is a key factor for achieving a good balance between using a reasonable amount of memory to index (the size of *each* slice should be small enough in order to fit well in main memory), and times to query (slices should be as large as possible in order to

[2] Even this limit can be surmounted by changing some OPSI parameters, at the expense of consuming more memory.

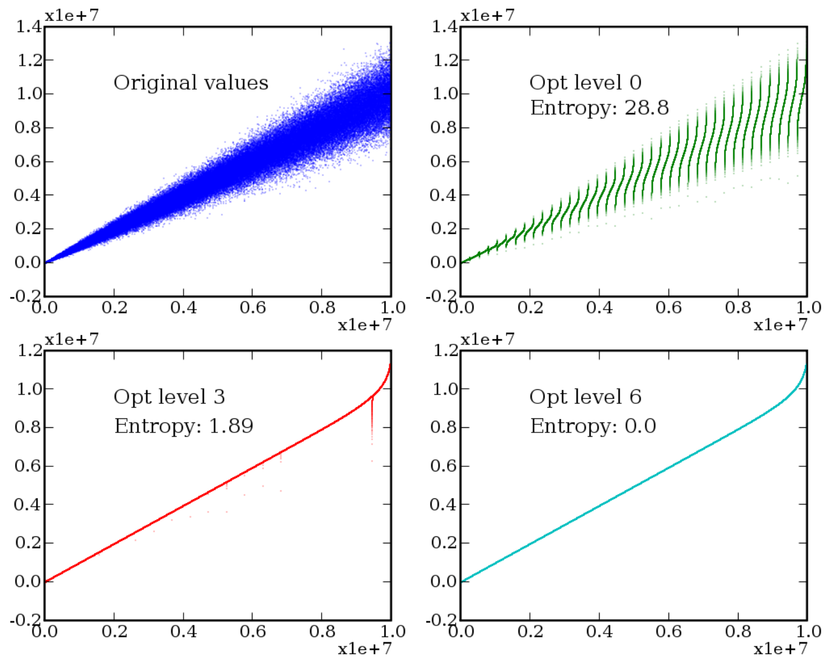

Figure 6: Typical entropy reduction process for a table with 10 millions of rows.

leverage the advantages of binary search as much as possible).

The *chunk size* is another very important parameter in that it is, by construction, the minimum amount of data that is read from disk on each I/O operation, so it has to be carefully tuned in order to reduce to a bare minimum the number of I/O operations in an index look-up while keeping the reading and decompressing times small enough for each chunk.

In the current implementation, these sizes are dynamically determined depending on the expected size of the dataset and also the optimization level. As will be shown in section 3, the resulting parametrization has proven to work pretty optimally for creating indexes for a wide range of the datasets.

Finally, it is worth mentioning that the sizes of *blocks* and *superblocks* do not directly affect query times, but rather indirectly through the reduction of the entropy during the process of index creation. Regarding the maximum size for *blocks* (those entities composed by slices), it is evaluated so that the amount of memory needed for swapping chunks inside one of them is contained enough to allow regular computer memory configurations to deal with it. Finally, the computation of the maximum size for *superblocks* is made in similar terms than blocks (i.e. that its reorganization isn't too costly in terms of memory usage).

2.6 New query interpreter allowing complex expressions

Although complex selections have little to do with a PSI paradigm, the inclusion of this feature in OPSI is important enough to include a quick discussion here. As you know, in the PyTables 1.x implementation of PSI, one was able to use only simple range selections of the form "`low <= column < high`" in in-kernel or indexed queries, but when there was a need to evaluate a more complex expression, it was necessary to do it by using regular Python loops and conditions. Of course, this was an important limiting factor for achieving good performance in in-kernel complex queries and, in addition, prevented the use of indexes in them. Starting with PyTables 2.0, a new query interpreter, based on the **Numexpr** package [9], has been integrated. With this, users can evaluate expressions like:

```
cond = "(1.3<=col1) & (col1<=43.23)" # simple query
cond += "& (sqrt(col1+3.1*col2+col3*col4)>3)" # complex query
```

and use them for doing indexed (or in-kernel, in case `col1` is not indexed) queries, like in:

```
result = [ row['col1'] for row in table.where(cond) ]
```

It is important to emphasize that Numexpr is oriented to achieve first-class performance when evaluating complex expressions like the one above. It achieves this by being able to compile the expression into its own op-codes, then passing the compiled code to its computing kernel, which has been highly optimized for executing array operations (note that index values can be seen as homogeneous arrays). On its hand, the Numexpr kernel splits the array operands in slices that fit well in the CPU cache, and then applies the op-code operations over them. The result is that it can get the most of the CPU computing capabilities for array-wise computations.

Just to give you an idea of its performance, Numexpr is able to evaluate complex expressions generally faster than a highly optimized package like NumPy: common speed-ups are usually between 0.95x (for very simple expressions, like `'col1 > 1'`) and 5x (for relatively complex ones, like the '`cond`' expression in the example above), but up to 10x speed-ups can be seen in some cases, specially when evaluating very complex expressions. Also, as the highly optimized computing kernel of Numexpr is the main reason behind the great performance that exhibits PyTables 2.0 in in-kernel searches (you can check [10] for more detailed benchmarks).

Despite the fact that, for indexed searches, the adoption of Numexpr does not necessarily speed-up queries too much (its main goal with this kind of searches is to be used as a *query interpreter* for complex conditions), the time improvement can still be important when the *residual expression*, (that is, the remaining conditions were the query optimizer decides that indexes cannot be used) has to be evaluated over a potentially large set of results.

3 Experiments

The best way to check whether a new algorithm is effective or not is to design a series of benchmarks and then compare it against some reference. In this section, we will be offer a detailed exposition of the experiments that have been carried out in order to check the performance of the current OPSI implementation for PyTables 2.0.

3.1 Experimental setup

All the benchmarks presented below have been carried out with the next server configuration:

- AMD Opteron @ 2 GHz processors
- SATA disk @ 7200 RPM (350 GB of hard disk available)
- 8 GB of main memory
- SuSE Linux 10.0
- GCC 4.0
- Python 2.5
- PyTables Pro 2.0 (final)

For comparison purposes, the Postgres 8 (version 8.0.8) relational database engine in combination with the psycopg2 (version 2.0.5.1) Python interface has been chosen[3].

The range of table sizes that has been chosen is limited basically by the amount of available space on disk but also by time availability (as it will be shown, the indexing time for very large tables can be really huge). The fact that the PyTables benchmarks do reach significantly larger table sizes than the Postgres ones is a direct consequence of the reduced use of disk and much faster index speed on the PyTables side.

Regarding the dataset chosen for the benchmarks, it is made up of synthetic data built using a normal random distribution[4] with limits varying for each row (0 for the inferior limit and the superior limit being the current row number). This election guarantees that, for very large datasets, the values contain enough entropy to allow the extrapolation of these results to many sets of real data. The resulting distribution can be seen in figure 7.

Finally, and despite the fact that the OPSI implementation allows any data type supported by PyTables (whenever it is *scalar*, of course), only *Float64* values have been considered for conducting the experiments stated here. This decision has been taken because extending this study to other data types would have exceeded our time constraints and besides, other benchmarks have already been conducted for other data types (see [10], where *Int32* data types were chosen). In any case, the perception of the authors is that most data types should expose similar behavior, so the results presented here can be extrapolated to other types without losing too much generality.

In the next sections, we are going to visit the different results for OPSI performance. We will mainly concentrate on how the important parameters of OPSI like the entropy, query speed, index time or index sizes, among others, depend on the size of the container tables, the optimization level and the use, or not, of data compression.

[3] We haven't tried out other databases for this study mainly because of time limitations. In any case, the goal of the study is to show that OPSI performs well enough, not that it performs better than any other engine (despite the fact that this could be the case for some scenarios).

[4] Obtained using the Mersene Twister random generator that comes with NumPy.

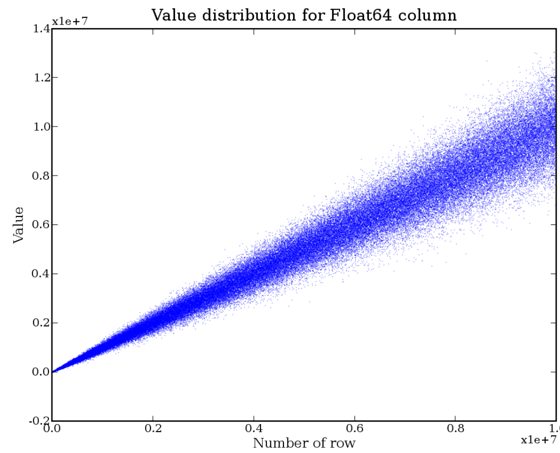

Figure 7: Distribution of values for a column with 10 millions of rows.

3.2 Entropy reduction

As previously stated, OPSI has the ability of reducing the entropy of an existing PSI index so that the query process can be completed faster. See the appendix B for a precise description on how the term "entropy" is used here.

In figure 8 it is shown how the OPSI algorithm can cope with reducing the entropy for a large range of table sizes. As you may notice, we have represented the entropy *plus* 1 in the y axis, just to provide a convenient representation in a double-logarithmic scale plot.

For small optimization levels (0 and 3) and for small and medium table sizes, the achieved entropy is relatively low. However, for larger tables, the entropy grows almost linearly. The optimization level 6 (the default) can reduce the entropy to 0 (i.e. achieving a completely sorted index) in tables up to $2x10^9$ rows and at this point it starts to grow pretty quickly. Finally, optimization level 9 is able to achieve a completely sorted index for all the range of table sizes that has been checked (up to 10^{10} rows), and entropy 0 could still be probably achieved for significantly larger sizes.

In the next section we will see how the index entropy actually affects query times.

3.3 Simple queries

Measuring query times can be a tricky thing because of the significant amount of parameters that affect them. In this section, many experiments have been carried out in order to properly assess the influence of these different parameters.

All of the measurements in this section have been done for a simple query, expressed in PyTables by the next condition:

```
res = [r[col2] for r in table.where("(low<=col4) & (col4<=high)")]
```

and its equivalent SQL query for Postgres:

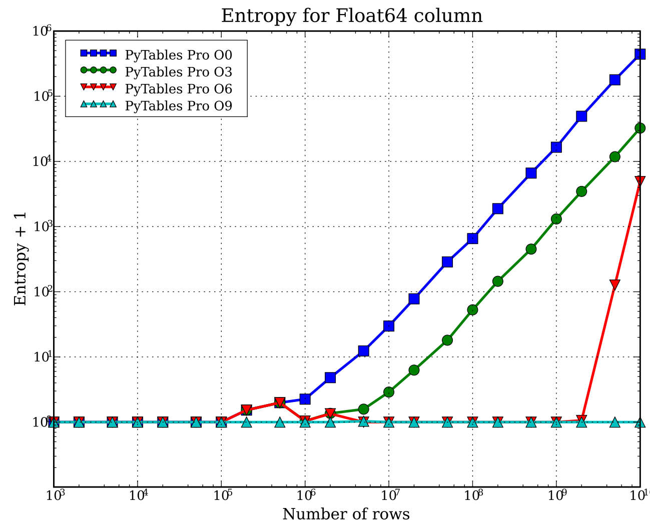

Figure 8: Dependency of the index entropy as a function of the table size and the optimization level.

```
select col2 from table where low <= col4 and col4 <= high
```

which means that all the values in field *col2* fulfilling that the value of the field *col4* is in the range $[low, high]$ are selected. However, for all the measurements in this section, the $[low, high]$ interval is very small and the number of results retrieved from the table is actually *none*. Such a number of results for a query is commonly known as "number of hits of the query" or *nhits* for short.

We have concentrated on simple queries with *nhits* equal to 0 because this is a good way to measure the performance of the indexing engine when executing pure lookups (in this way, we should not worry about times for wrapping the resulting data into Python containers, which aren't negligible, specially for relatively high values of nhits). The more general case for complex queries with nhits different from zero will be discussed in section 3.4.

The parameters that greatly affect query times like the state of the cache (cold or warm), the compression, the optimization level or the query repetition pattern will be discussed next.

3.3.1 Response times with cold cache

As already said, the OPSI does implement several levels of caches that, when warmed, can accelerate queries quite a lot. In addition, other levels of cache exist, like the HDF5 caches or the filesystem cache of the underlying operating system.

In this section, cache effects are avoided by performing only 10 queries using very different random limits with a freshly opened database and taking the average response time. This average time is what is represented in figure 9.

It can be noted that the query times for an index with optimization level 0 (worst) are pretty

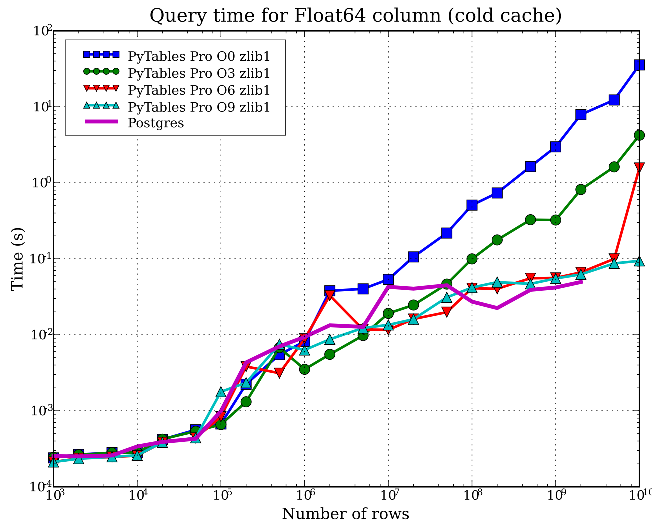

Figure 9: Dependency of the query time when the caches are cold as a function of the table size and the optimization level.

good for small and medium tables, although for tables larger than 10^9 rows the times are in the order of 10 s. For optimization level 3, the times for larger tables can be up to 10x better. For optimization level 6, the times are very good except for tables larger than $5x10^9$rows, where the entropy of the index grows considerably (we have already seen this in figure 8). For the optimization level 9 (best), the times are very good (always under one tenth a of second) for all the ranges of table sizes, and similar to Postgres times (sometimes better, sometimes worse).

3.3.2 Response times with warm cache

When the same query (with different limits each time) is done over the same indexed columns over and over, the different caches start to warm and hence, the response times start to improve. In figure 10, one can be observe the average of the latest 500 queries of a total of 2500 (this is a significant figure that ensures that all the hierarchy of caches has been warmed, even for the biggest tables).

It can be seen that, for optimization level 0 (O0), OPSI can perform queries in tables up to 10^8 rows in less than one tenth a of second, which is quite good; however, for larger sizes, the query time increases considerably. With level O3, queries can be done in just 10 milliseconds, or less, for tables smaller than 10^8 rows, and in less than one second for larger ones. Finally, levels O6 and O9 are able to execute queries in around 10 milliseconds for tables with up to 10^9 rows and in less than a tenth of a second for larger ones (O6 does perform a little badly for the 10^{10} size, because of the increased entropy). In particular, level O9 does perform better than Postgres for all the range of table sizes.

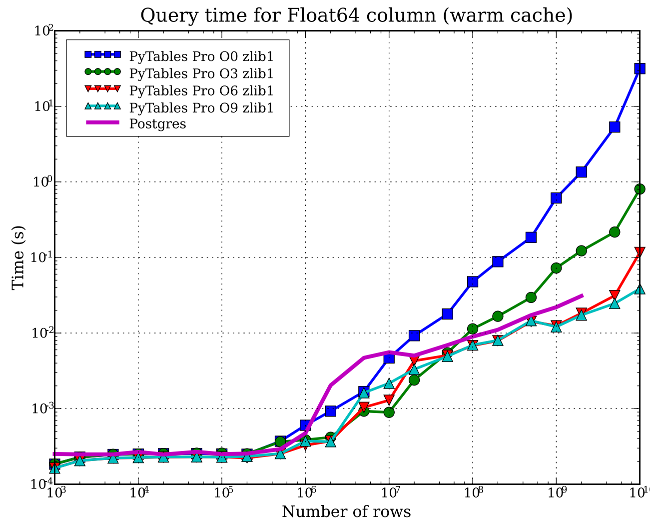

Figure 10: Dependency of the query time when the caches are warm as a function of the table size and the optimization level.

3.3.3 How compression affects query times

It has been repeatedly claimed that OPSI is able to compress its indexes, but until now, nothing has been said about how this could affect query performance. In figures 11 and 12 one can observe the impact of compression in query times for both the cold and warm cache cases.

In figure 11 (cold caches) it can be noticed that there is not a lot of difference between using compressors or not using them; it can be seen, though, a slightly better behavior of the compressed indexes versus the non-compressed ones. On its hand, figure 12 (warm caches) reveals the same pattern more or less, i.e. that there is not much difference in performance when using compression or not. There are, though, a couple of special cases for the warm caches scenario that require a bit of discussion. The first one is for optimization level 0 (O0), where using compression is self-defeating for an index that fits in the filesystem cache, but the situation is the inverse when the index does not fit in this cache (> 10^9 rows). The second one is for the maximum optimization level (O9), where using compression has a very little impact on performance for sizes that fit in the filesystem cache, but it has a clear advantage over non-compressed indexes when the index does not fit in cache.

On the other hand, in the graphs can also be noticed that the LZO compressor does perform slightly better than Zlib, but not by a great extent (the difference is much more apparent for compression than decompression, as will be seen in section 3.5).

Given these facts, and provided that compression reduces the size of indexes by a factor of 3x typically (see section 3.6), it has been decided to adopt Zlib with compression level 1 as the default for creating indexes in PyTables 2 Pro (the user can change this, of course). Despite of being faster, we have not chosen LZO because its use is not as spread as Zlib, and in addition, it is not required by normal HDF5 installations, while Zlib is. Moreover, whenever compression is

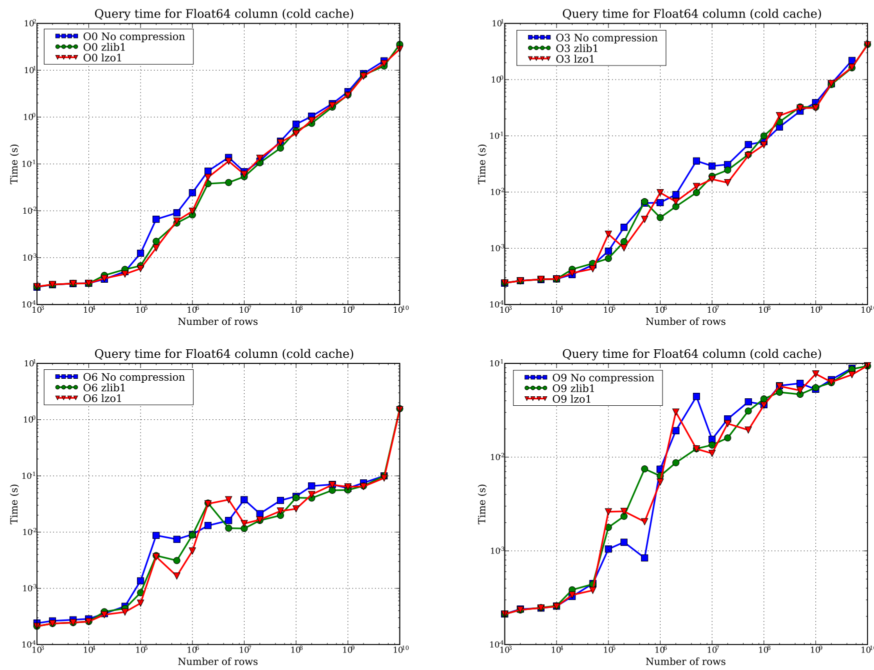

Figure 11: Dependency of the query time with compression and optimization level when the caches are cold.

active, the *shuffle* filter is always used as a preconditioner to achieve better compression ratios. This has very little impact on CPU usage, and the ratios are noticeably higher, as can be seen in the "Optimization tips" chapter of the PyTables reference [8].

This is why, whenever we refer to indexes in OPSI, and it is not explicitly specified whether they are compressed or not, the reader should understand that they are compressed with Zlib (compression level 1) and using *shuffle* as a preconditioner.

3.3.4 Times for queries in the first-level OPSI cache

In order to assess which is the response time of the first-level cache for the OPSI, a measurement of the response times for repeated queries has been carried out. You can see the result in figure 13. In it, it can be seen that the OPSI cache is doing a quite good job in retrieving the repeated query very fast (about 120 μs). In comparison, the Postgres cache does take more than 2x more (about 250 μs). It is also curious to see how both the PyTables Pro and Postgres times are more or less constant for all table sizes, but the Postgres response time for an index size of 2x10^9 elements grows considerably (the reason for this is unknown, but the fact is experimentally reproducible).

This measurement has been made just to offer an idea on the degree of performance that the internal LRU caches for OPSI can reach. Such a high performance is responsible for part of the

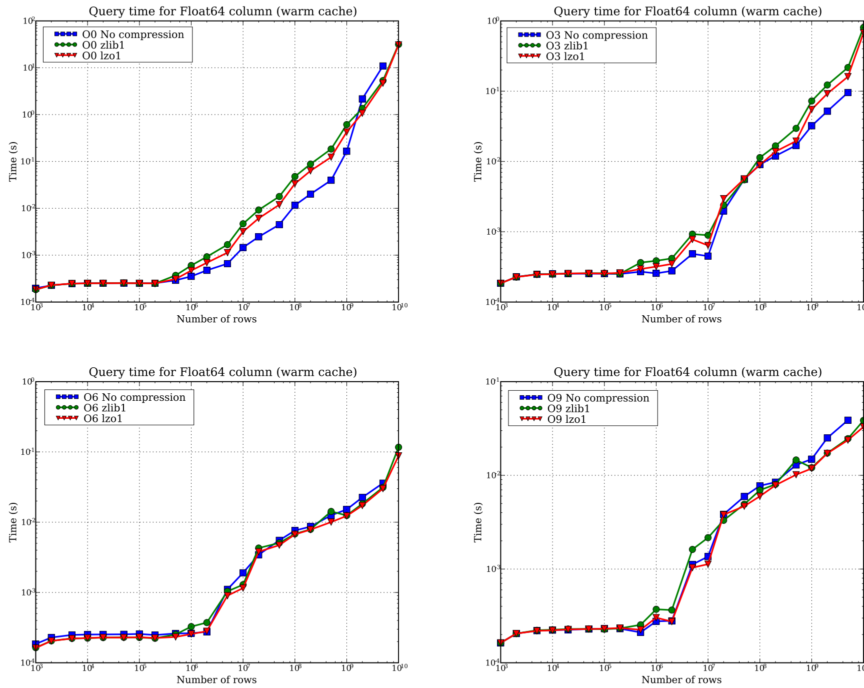

Figure 12: Dependency of the query time with compression and optimization level when the caches are warm.

good response times in warm caches scenarios.

3.4 Complex queries

In previous sections we have seen how OPSI indexes perform on simple queries with no results ($nhits = 0$). However, in real life, queries are complex and do have results as an outcome. So, in order to give an idea on how OPSI performs in this new scenario, let's take the next query:

```
res = [ row[col4] for row in table.where(
        "(low<=col4) & (col4<=high) & (sqrt(col1+3.1*col2+col3*col4) > 3)") ]
```

where the same simple condition than in previous experiments has been chosen, *plus* another complex condition that checks the values of other columns of the same table (more specifically, *col4* is an indexed Float64 column and *col1*, *col2* and *col3* are unindexed ones of types Int32, Float64 and Int32 respectively). Also, different values for the range $[low, high]$ will be selected in order to allow for a variable number of hits as the outcome of the query.

In figure 14 it can be seen the mean time for the first 5 distinct queries (but having approximately the same number of hits) along a table with 10^9 rows (a gigarow) for different query ranges (and hence, having different number of hits). With speed-ups between 2x-50x over the

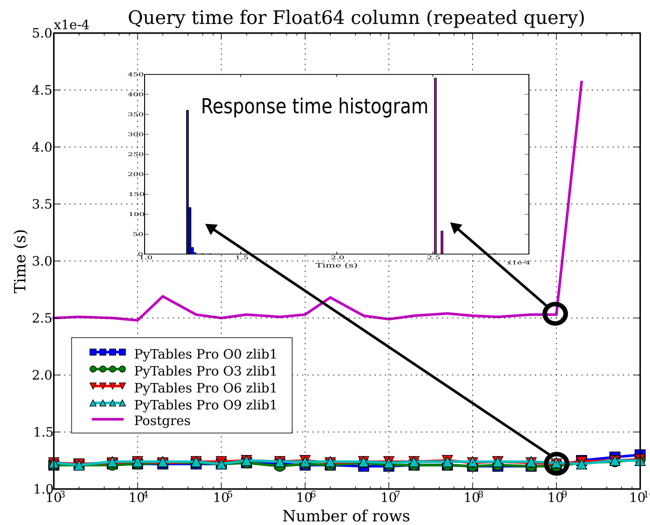

Figure 13: Dependency of the query time when the query is repeated as a function of the table size and the optimization level.

Postgres engine, it is apparent that the OPSI indexes do perform extremely well in this more “real life” scenario.

One of the main reasons behind OPSI indexes being so fast in this scenario in comparison with Postgres is, most probably, that typical Python interfaces for relational databases have to wrap every single result within a Python container, while OPSI puts the retrieved results in arbitrarily large NumPy containers whose creation is very fast in comparison (this has been discussed more accurately in the NumPy mailing list [11]). So, the advantages of NumPy adoption by OPSI are not only in terms of using its fast computing capabilities, but also, and utterly important, of leveraging its flexible and extremely fast to create data containers[5].

3.5 Index creation times

One of the advantages of OPSI indexes is that they build pretty fast compared with traditional databases. In figure 15, one can see the index creation time for different optimization levels. As expected, indexes with optimization level 0 are created quite fast compared with higher optimization levels. It is also noticeable that the OPSI indexing times for very large tables are much shorter than Postgres (up to 50x for minimum optimization level and 5x for maximum optimization level).

Also, it is interesting to notice how the compression level affects the index creation time. This can be seen in figure 16. It turns out that using compression noticeably increases indexing times for small and medium table sizes (up to 5x), but for large ones ($> 10^8$ rows), the times

[5]Consequently, developers of database wrappers for Python should consider to adopt NumPy containers (or a similar solution) as well.

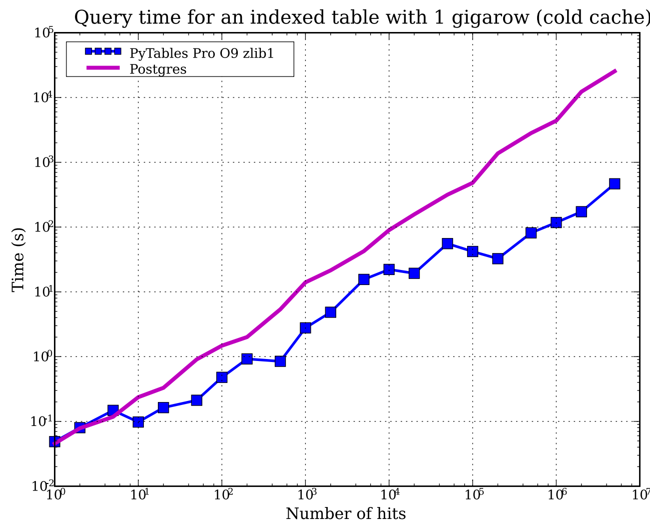

Figure 14: Times for doing a query with different number of hits on a indexed table with one gigarow.

when using compression or not are similar. This is because, when the dataset doesn't fit in the filesystem cache, the extra time that the compression takes is compensated by the corresponding reduction of time spent in writing index data on disk (the compressed data takes far less space, as can be seen in section 3.6).

3.6 Index sizes

As OPSI supports index compression (it is enabled by default, as was already said), it is interesting to see which are the real wins in terms of the compression level that can be achieved. The sizes for compressed indexes compared against uncompressed ones can be observed in figure 17.

The use of compression for this case makes the index datasets reduce 2.2x over the non-compressed ones. It is worth noting that uncompressed OPSI indexes already take 1.4x less space than Postgres indexes, and that compressed OPSI indexes take approximately 3x less space than Postgres ones. This is very important because, by using OPSI indexes, you can deal with 3x larger datasets (provided that the datasets are compressible to the same extent as indexes, which is plausible) than Postgres when using the same computer.

3.7 Memory usage in index creation

One of the reasons why OPSI indexes are created relatively fast is that they keep a complete slice of elements in main memory in order to proceed with the partial sort. This allows the CPU to perform each partial sort very quickly and touching the disk as little as possible. However,

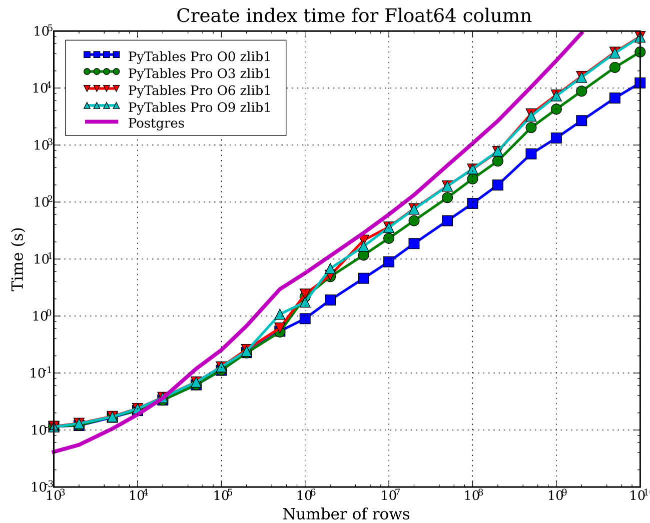

Figure 15: Times for creating an index as a function of the table size and index optimization level.

one of the drawbacks of this approach is that you need a considerable amount of memory for keeping the whole slice in memory.

In figure 18, it can be seen the approximate amount of memory that an OPSI index requires for being created. There, it can be noted that this amount depends heavily on the optimization level and shows an exponential dependency on the size of the column, that ceases when the slices cannot grow anymore (i.e. when the $8x10^{12}$ limit for the superblock size has been reached, see section 2.4).

For the matter of comparison, for creating an index with $2x10^9$ float64 elements, OPSI requires 78 MB, 126 MB, 126 MB and 222 MB for optimization levels 0, 3, 6 and 9 respectively. On its hand, Postgres requires 80 MB (69 MB for the Postgres daemon plus 11 MB for the Python interpreter) for doing the same task. So, OPSI does need, specially with higher optimization levels, significantly more memory to create indexes than traditional indexing engines. However, the maximum amount that can ever be consumed with the current parametrization of the OPSI implementation of PyTables Pro is, as can be seen in figure 18, 3 GB approximately (more for string fields with more than 8 characters), and this for the case of indexing really huge tables ($> 8x10^{12}$ elements). Provided that systems with 4 GB of RAM are easily available nowadays and, in addition, are much more cheaper than a RAID of disks being able to keep tables (with the corresponding indexes) with $8x10^{12}$ rows or more, this cannot be seen as an important limitation at all.

Also, in order to be able to further reduce the entropy for the largest tables, it might be desirable for the current parametrization to be tuned in the next few years to adapt to machines with more than 4 GB of memory.

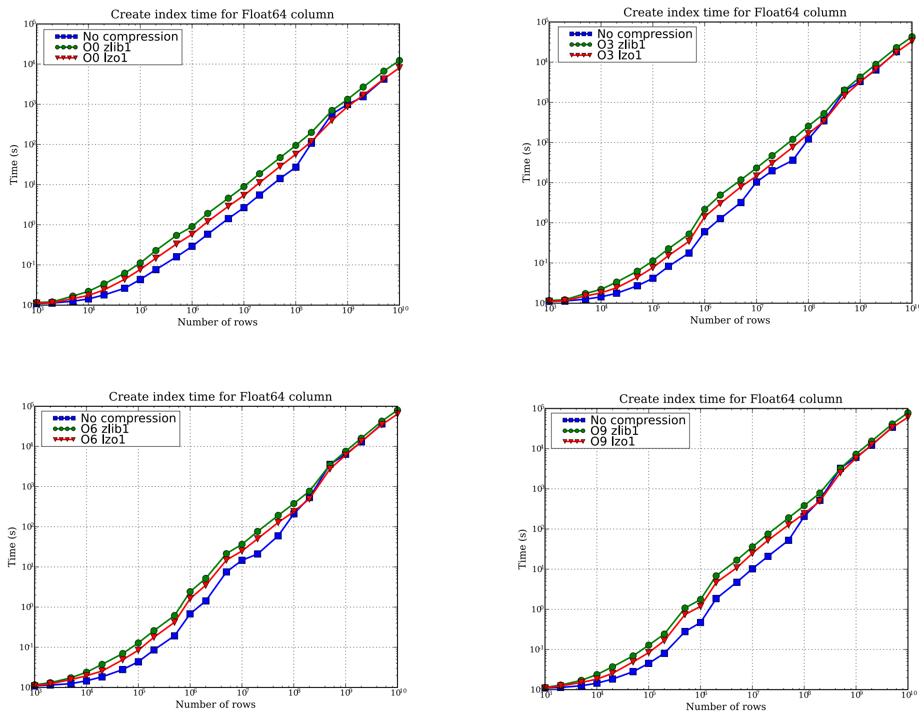

Figure 16: Dependency of the index creation time with compression and optimization level.

4 Conclusions

From what has been depicted in the description and experimental sections, several conclusions can be drawn about OPSI indexes:

- They allow to select the quality level of the index, which lets the user choose the most appropriate level for her needs.
- When doing simple queries, OPSI achieves speeds that are similar to traditional indexing engines. When the caches are cold, OPSI can do queries in quite less than 100 ms for tables with 10^9 rows. When the caches are warm, the typical query times for tables with 10^9 rows are around 11 ms (approximately the same time than a single disk *seek*).
- For complex queries, OPSI can be between 2x and 50x faster than Postgres. This is mainly because of the fact that OPSI results are returned in the highly efficient NumPy containers.
- OPSI indexes are compressed by default. This allows for up to a 3x reduction of disk space for keeping the indexes in comparison with relational databases.
- Creation of OPSI indexes is pretty fast and between 5x and 50x faster (depending on the optimization level) than Postgres. This fact can be critical when you have the need to index extremely large tables in a reasonable amount of time.

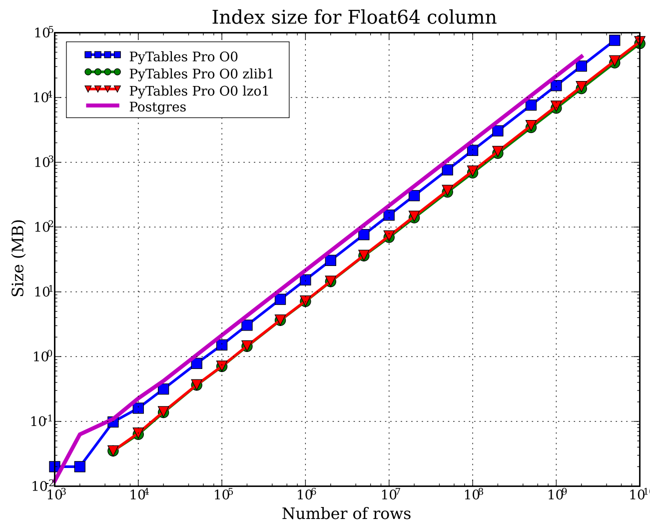

Figure 17: Index sizes as a function of the table size and index optimization level.

- As a small drawback, OPSI requires a significant amount of RAM in order to create an index (up to 3 GB of main memory for really huge tables). However, this is not a limitation in practice because modern computers can be able to cope with OPSI's demands of RAM.

All of these reasons make OPSI indexes a very good option for indexing very large amounts of data very fast, with selectable index quality and top-level query performance. Because of its advantages, authors foresee a wider adoption of the PSI indexing paradigm for other databases (relational or not) in the near future. Meanwhile, users can start enjoying these advantages right now in the OPSI implementation of PyTables Pro 2.x series.

5 Future work

5.1 A better query optimizer

The query optimizer that comes with the new query interpreter can only currently detect whether the indexed columns can be used if they appear at the leftmost side of the condition expression (this is why it is important to put the indexed columns at the very beginning of the condition in order to maximize the chances for using them). However, our intention is to provide this optimizer with more intelligence so that it is able to discover more opportunities to use indexes in complex queries.

5.2 Parallel OPSI

As we have seen in section 2.4, superblocks are the perfect candidates for dividing the effort of creating/searching OPSI indexes in several processors and/or computers. As the data between

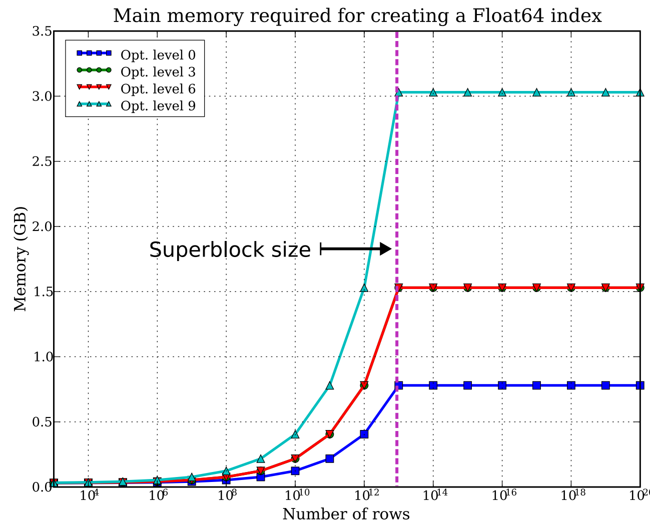

Figure 18: Memory taken by an index creation as a function of the table size and the index optimization level.

different superblocks is completely independent one of each other, both the creation and the searching algorithms admit an embarrassingly parallel approach that can be efficiently implemented not only in multi-core machines with a single (and possibly parallel) filesystem, but also in clusters of loosely coupled nodes (using MPI in combination with Beowulf or just plain COW clusters).

The resulting parallel indexing system (POPSI?) could be perfect to index immensely huge amounts of data (up to 2^{63}elements) in a scalable and relatively cheap way.

6 Availability

OPSI indexes are available in PyTables 2.0 Professional Edition [7] or higher from Cárabos Coop. V. This product that can be acquired at `http://www.carabos.com/buy`. There is currently a liberation process going on for releasing all the software from Cárabos as open source. Be sure to read about it at `http://www.carabos.com/liberation`.

7 Acknowledgments

We are specially grateful to Wladimiro Díaz, from the *Departament d'Informàtica* of the *Universitat de València* for allowing us to use its computing facilities for carrying out the experiments of this article.

Appendix A: The OPSI LRU caches

Below is a list of the caches implemented for OPSI in PyTables Pro. As you can see, there is a cache for covering almost any kind of repetition pattern in a series of queries.

LIMDATA It caches a table with the rows that result from a query on an indexed table. Only valid for the Table.readWhere() method. This is the highest level cache and the fastest if your bounds in queries get repeated very frequently.

LIMBOUNDS Caches the starting points and lengths of values in slices that satisfy the query. This cache can be applied to all the range of methods for doing queries. However, it only applies if the same query is repeated over time (exactly the same as the above).

TABLE Cache for rows coming from sparse reads in tables. Such sparse reads are typical in indexed look-ups, but can appear in other scenarios as well. It can accelerate searches returning a small subset of rows that have been already returned in previous queries (or other read methods).

RANGES This is the in-memory version of the *slice cache table*. It is not exactly an LRU cache (in fact, it is the only exception in this list) just because the amount of data that it contains is very small, and can be loaded completely in memory.

BOUNDS A cache for keeping the data of the *chunk cache table*. This is relatively low-level, and it is useful when the values satisfying the queries expose spatial locality, this is, when the values in current query falls near of other values fetched in previous queries.

SORTED Caches the values of the *sorted* dataset that have been selected in previous queries.

INDICES Caches the values of the *indices* dataset that have been selected in previous queries. Together with the SORTED cache, they are the least efficient caches of the series, but still, they can be helpful when the more efficient ones have failed.

Appendix B: Computing the entropy of a PSI

We have devised a quick way to express the degree of "unsortedness" of a *partially sorted index* (PSI). Along this article, this parameter is called *entropy*, and the exact way in which it is computed is explained here.

In figure 19, we have represented the values for 6 slices of a given PSI index. As it can be seen, the values inside each slice are completely ordered and hence, they do not overlap. However, the values of different slices do actually overlap between them (in the end, the values represent a *partially* ordered index). Given the slices S_i and S_j (where $j > i$), we define the overlap $o_{i,j}$ between them as:

$$o_{i,j} = \begin{cases} max(S_i) - min(S_j) & \text{if } max(S_i) > min(S_j) \\ 0 & \text{otherwise} \end{cases}$$

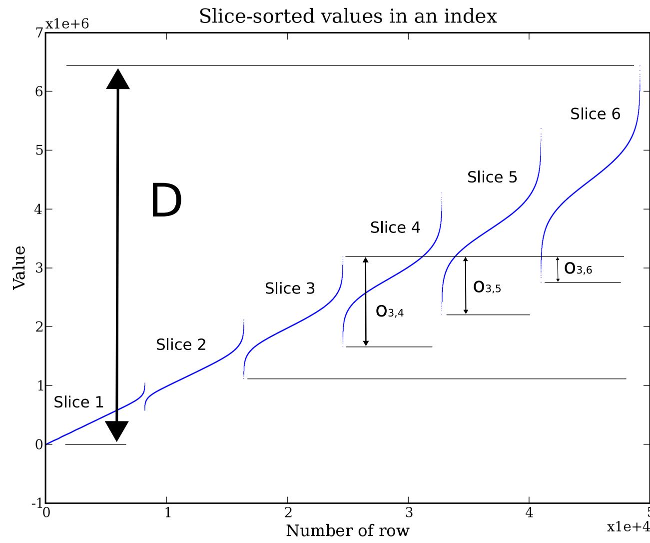

Figure 19: The parameters used in the computation of the entropy in a PSI index.

Moreover, we define D as the distance between the largest and the smallest value in PSI. These $o_{i,j}$ and D parameters have been graphically depicted in figure 19.

Given this, we can finally define the entropy $\mathcal{E}$ of a PSI index, as:

$$\mathcal{E} = \frac{\sum_{i=1}^{N}\sum_{j=i}^{N} o_{i,j}}{D}$$

This definition has the interesting property that, for the case where a PSI is completely sorted, it takes the 0 value. This is consistent with the general definition of entropy, where a system having no entropy means that its internal state is completely ordered.

Finally, note how our definition of entropy is divided by D. This is so in order to better assess the level of the "unsortedness" independently of the range that the values span.

References

[1] Francesc Altet, Ivan Vilata, Scott Prater, Vicent Mas, Tom Hedley, Antonio Valentino and Jeffrey Whitaker. PYTABLES. A hierarchical database. `http://www.pytables.org/`

[2] Guido van Rossum et al.. THE PYTHON LANGUAGE. A remarkably powerful dynamic programming language. `http://www.python.org/`

[3] The HDF Group. THE HDF5 LIBRARY. A general purpose library and file format for storing data. `http://www.hdfgroup.org/HDF5/`

[4] The Postgres development team. POSTGRESQL. A powerful, open source relational database system. http://www.postgresql.org/

[5] Luke Gosink, John Shalf, Kurt Stockinger, Kesheng Wu, Wes Bethel. HDF5-FASTQUERY: ACCELERATING COMPLEX QUERIES ON HDF DATASETS USING FAST BITMAP INDICES. http://citeseer.ist.psu.edu/739947.html

[6] Travis Oliphant et al.. NUMPY. Scientific Computing with Numerical Python. http://numpy.scipy.org/

[7] Cárabos Coop. V.. PYTABLES PROFESSIONAL EDITION. http://www.carabos.com/products/pytables-pro

[8] Francesc Altet, Ivan Vilata, Scott Prater, Vicent Mas, Tom Hedley, Antonio Valentino and Jeffrey Whitaker. PYTABLES USER'S GUIDE. http://www.pytables.org/docs/manual

[9] David Cooke and Timothy Hochberg. NUMEXPR. Fast evaluation of array expressions by using a vector-based virtual machine. http://www.scipy.org/SciPyPackages/NumExpr

[10] Francesc Altet and Ivan Vilata. FINDING NEEDLES IN A HUGE DATASTACK. http://www.pytables.org/docs/FindingNeedles.pdf

[11] Francesc Altet, Timothy Hochberg and others. DISCUSSION IN THE NUMPY LIST ABOUT THE COST OF RETRIEVING DATA OUT OF A DATABASE IN PYTHON. http://thread.gmane.org/gmane.comp.python.numeric.general/9704