Extending HDF5 Datasets: Enhancements to the Chunk Indexing Methods

Vailin Choi*, Jerome Soumagne* Quincey Koziol*
*The HDF Group
Champaign, IL 61820

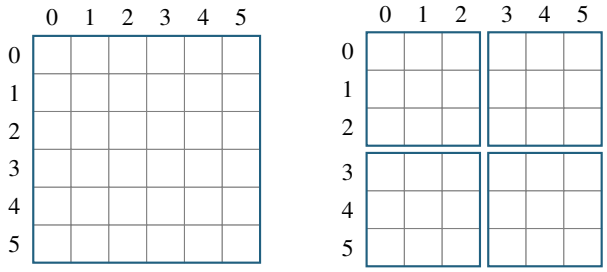
***Abstract*—The HDF5 library requires the use of chunking when defining extendable datasets. Chunking makes it possible to extend datasets efficiently without having to reorganize contiguous storage excessively. However, to retrieve elements in the datasets that are stored in chunked form, the chunk containing those elements must be located and accessed. Therefore, the library must define and use efficient indexing methods for the retrieval of those chunks. We present in this paper the indexing methods that are being used, as well as some of the enhancements made to improve performance of these methods.**

I. Introduction

The HDF5 library [1] [2] provides users with a high degree of flexibility for data management—users may define and organize data into different groups and datasets and build a hierarchical tree of data. Datasets may be contiguously mapped from the application memory to a file, or stored in more complex patterns to ease further access and analysis of the data. One commonly used option is to define and create a dataset as *extendable*, which allows new values to be appended to it. An HDF5 dataset is extendable when the maximum dimension sizes of its dataspace are greater than the current dimension sizes, with the maximum dimension size being specified as either *fixed* size or *unlimited* size.

When defining an extendable dataset, the HDF5 library requires the dataset to be partitioned into fixed-size multi-dimensional chunks—this operation, illustrated in figure 1, is called *chunking*. Chunking makes it possible to extend datasets efficiently without having to reorganize contiguous storage excessively. To retrieve elements in the datasets that are stored in chunked form, the chunk containing those elements must be located and accessed. Several data structures can be used to achieve that and the locations of chunks for a dataset are currently stored in a B-tree data structure. This structure maps the index of the chunk to the file offset where the chunk's elements are stored. However, depending on the use case and in particular for extendable datasets, it presents some drawbacks, which are addressed in this paper.

This paper is organized as follows. Section II presents the current indexing method used within the library for chunking as well as modifications that were made to improve performance when a dataset is being extended. Section III presents performance evaluation results. In Section IV we discuss related work before presenting conclusions and future research directions in Section V.



(a) Contiguous. **(b)** Chunked.

Fig. 1: HDF5 dataset storage.

II. Architecture

Chunking is a technique used in the HDF5 library in various cases: to optimize I/O to disk [3], to compress data [1], and to efficiently extend datasets. To retrieve elements in the datasets that are stored in chunked form, the chunk containing those elements must be located and accessed. The HDF5 library's default data structure for indexing chunks is based on a B-tree data structure, which maps the coordinate offset of the chunk to the file offset where the chunk's elements are stored.

A. B-Trees

B-trees are generally used for indexing data blocks and are one of the main components of many file systems [4] [5]. Insertion of a new entry and lookup is realized in $O(\log_b n)$, where n is the number of nodes and b the order of the tree. In the case of HDF5 datasets and in the original B-tree structure, referred to as *B-Tree version 1* (BT1), the record for locating a chunk stores the following information:

- The coordinates of the chunk in the dataset's dataspace. This is used as the key for locating chunks in the B-tree.
- The size of the chunk, in bytes.
- The address of the chunk in the file.
- Additional metadata.

As shown in figure 2, each leaf points to the address of a chunk. When a dataset is extended, a new entry is added, which may generate a node split if the node where the record needs to be inserted is full. It is worth noting that most chunked datasets are either fixed dimension or extend in only one dimension—very few users take advantage of the ability to extend multiple dimensions of a dataset.

Because HDF5 datasets only allow their dimensions to be increased or decreased at their upper bound, a B-tree used

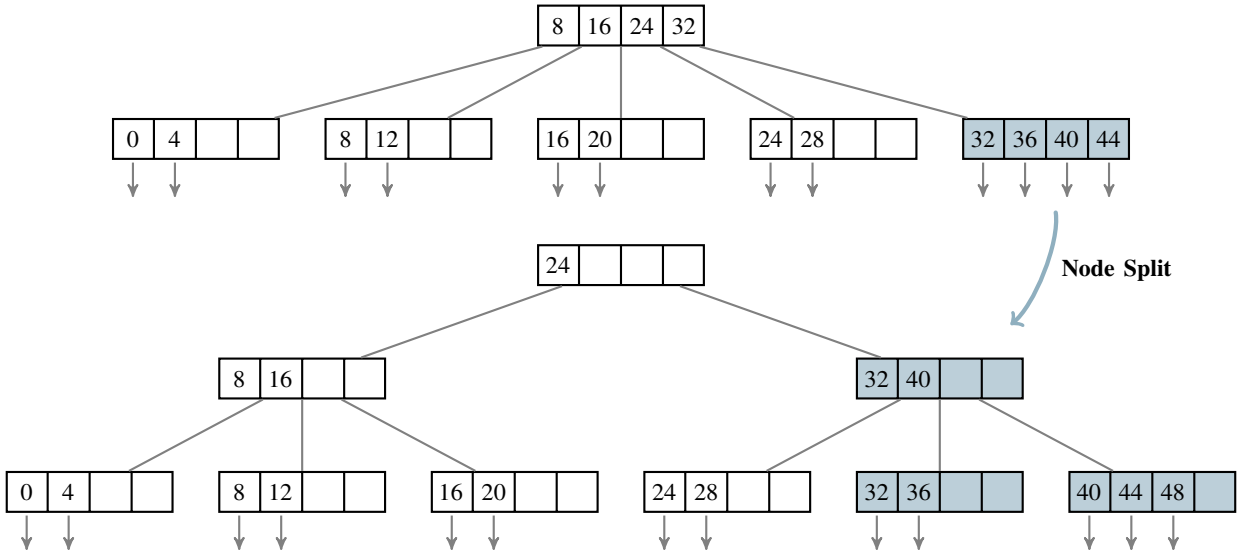


Fig. 2: Simplified version of original B-tree structure used for indexing chunks. In the case of extendable datasets, whenever a new chunk is added to the B-tree, its right-most node, if full, is split and a new half empty node is created.

to index chunks for a 1-D dataset will only ever insert or remove records from its right-most node. One of the problems this causes is that inserting/removing only from the right-most node lets most records half empty, as illustrated in figure 2. To fix that issue, one of the main improvements made with a new version of the B-tree structure used, referred to as *B-tree version 2* (BT2), is to re-balance the entries so that nodes that were half-empty in version 1 are now full. Some additional modifications have also been made to the BT2 to improve performance:

1) When appending to a dataset, the library will insert a new record to the B-tree. To determine that the record to be inserted is not in the tree, a naive implementation traverses the corresponding tree nodes and compares the records in each node before the insertion. To accelerate the search, because entries are frequently added at the upper bound, the maximum and minimum records in the tree are determined and stored in memory. The library can then quickly determine that the new record to be inserted is not in the tree if it is greater than the maximum record or less than the minimum record.
2) The default BT2 node size is 512, which leads to a greater tree depth than version 1 for the append test scenario. This contributes to a lower performance because the traversal time is increased. The node size for the tree is therefore modified to 2048, which will flatten the tree with a tree depth that is the same as the BT1.

B. Extensible Arrays

While the version 2 B-tree previously mentioned provides a better balancing of the records in the tree, using it in the 1-D case (or for datasets with only one unlimited dimension) still only ever inserts or removes records from the right-most node. This negates most of the advantage of using a complicated data structure like a B-tree to index the chunks. Additionally, for applications, which wish to rapidly append records to the dataset, having to traverse and/or update multiple B-tree nodes for each record insertion imposes an additional performance penalty. This is especially a drawback when splitting one or more B-tree nodes is required to accommodate new record for a chunk.

For this particular case, indexing chunks requires a more appropriate data structure, referred to as *extensible array* (EA), and shown in figure 3. This structure can be seen as a dynamic array structure, which can contain new elements, as needed—insertion, removal and lookup of elements are realized in constant time. Brodnik et al's paper [6] shows how to implement a similar structure with $O(1)$ append, shrink and lookup operations, along with optimal metadata overhead for indexing the array elements. In our case and for indexing chunks in HDF5 datasets, the array elements in the data blocks are pointers to the chunks on disk. To conceptually group the data blocks that the index block points to, the extensible array defines *super blocks*. Super blocks can be seen as *indirect blocks* in the data structure and, as mentioned in [6], help to show the pattern of changes to the data blocks: first the size of the data block doubles, then the number of data blocks of that size doubles, then the size doubles again, then the number of data blocks of that size doubles, etc.

A couple of things should be noted about this data structure (described in figure 3):

1) The super blocks for the first two data blocks are omitted and the pointers in the index block point directly to the data blocks.
2) Constant time (i.e., $O(1)$) lookup, append and shrink are still in effect. The number of I/O accesses to find any chunk address is always 2 or 3: index block $\rightarrow$ super

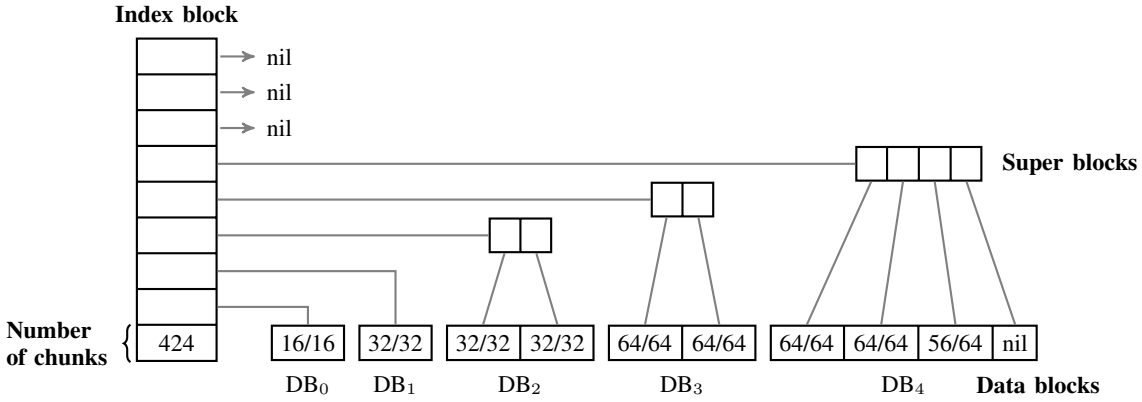


Fig. 3: Simplified version of extensible array structure used for indexing chunks.

block → data block. In actual operation, the index block will certainly be *hot* enough to stay in the HDF5 metadata cache, along with many/most of the super blocks, making the average number of I/O accesses to retrieve a chunk address between 1 and 2, in all likelihood.

3) Super blocks/data blocks, which are not yet needed, are not allocated and have a *nil* pointer in the index block/super block (respectively) that refers to them.
4) The height of the index block is constant and its height can be computed at creation so that resizing it is unnecessary. This can be done by computing the maximum number of chunks possible for the file's address range (typically 64-bits) and deriving the maximum number of data/index blocks needed to index that many blocks. For example, if there are 16 chunks in the initial data block, a chunk size of 2048 (with a single unlimited dimension), and a 4-byte (integer) data element size, the maximum height of the index block needed to store 1.84×10^{19} bytes of elements (which is the maximum offset possible in a 64-bit file) is 46:

$$\begin{aligned} 16 \times \sum_{i=0}^{n} 2^i &= \frac{2^{64}}{2048 \times 4} \\ 16 \times (2^{n+1} - 1) &= \frac{2^{64}}{2048 \times 4} \\ 2^{n+1} &= \frac{2^{64}}{2048 \times 4 \times 16} + 1 \\ 2^{n+1} &= 2^{47} + 1 \\ \lceil n \rceil &= 46 \end{aligned}$$

C. Hash Table

In the case where chunks are present in cache (i.e., in memory), chunks are retrieved from the cache hash table. When a dataset's dimensions are extended and the dataset's rank is greater than one, the HDF5 library scans and updates each entry in the chunk cache, due to the modified dataspace. In the original implementation, each entry is hashed based on the chunk index, which varies according to the dataset dimension sizes. When the dimension sizes are modified, the library recalculates each entry's chunk index and updates the cache accordingly. This is an expensive operation since the update time increases as the number of entries is increased in the cache.

To improve performance of chunk retrieval from the cache, two modifications are made to the chunk layer:

1) The fastest changing dimension in the chunk index calculation is saved after initial computation so that it can be later used when updating the chunk cache.
2) A different formula to hash chunk entries into the cache is used so that it no longer depends on the chunk index. The new formula is based on the $\log_2$ function, which reduces the number of chunk cache updates due to consecutive dataset extension operations. As shown in figure 4, using scaled coordinates (i.e., relative to the dimension sizes), the $\log_2$ function is used to compute a binary representation of the coordinates. The number of bits that are necessary to encode the coordinates only increases when the new dimension's size crosses a power of 2 boundary.

D. Summary

With these data structures in place, when the chunked dataset is on disk and has only one unlimited maximum dimension, the extensible array indexing method is used, allowing chunk access in $O(1)$ time. When it has more than one unlimited maximum dimension sizes, the B-tree version 2 indexing method is used, allowing chunk access in $O(\log_b n)$, with b the order of the tree and n the number of nodes. When it is in cache, the chunk is accessed through the cache hash table and the computed hash value.

III. Evaluation

To evaluate the performance of these new methods, we first consider the extension of datasets so that 1-byte chunks are appended, up to $2,500,000$ bytes. Note that the default storage for chunked datasets is still the default B-tree indexing (BT1). In order to retain as much forward compatibility as possible, all the improvements made are only used when the `H5F_LIBVER_LATEST` value is used as the *high bound* to

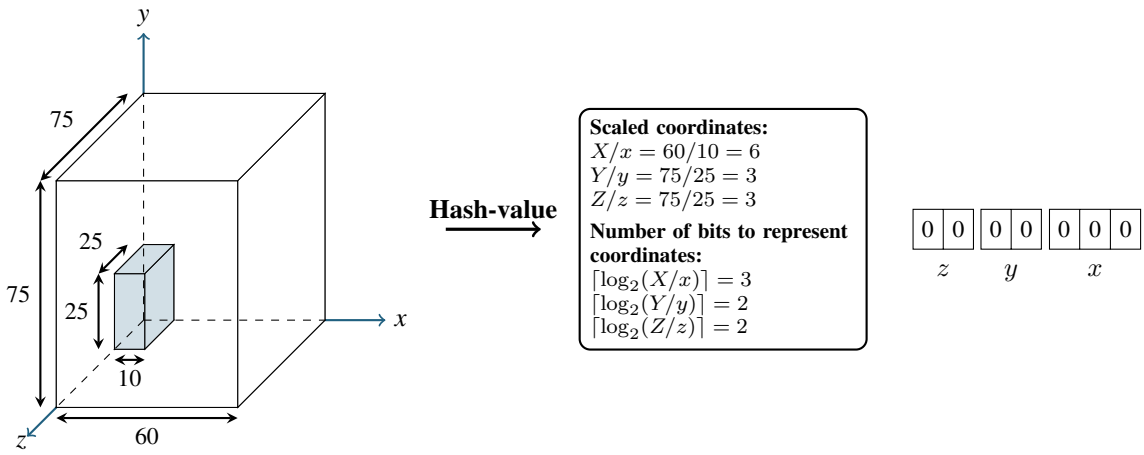


Fig. 4: Example of hash-value computation that is used to retrieve chunks using scaled coordinates in a 3-dimensional dataset.

the call to `H5Pset_libver_bounds`, which can be set by an application when the file is opened.

A. Test Scenarios

We consider multiple scenarios, which are defined in settings 1, 2 and 3.

a) Setting 1: creates a 1-dimensional chunked dataset with the following dataspace:

```
curr[0] = 0;
max[0] = H5S_UNLIMITED;
sid = H5Screate_simple(1, curr, max);
```

This compares the BT1 and EA indexing methods, when appending to the dataset created with the old and new library format respectively.

b) Setting 2: creates a 2-dimensional chunked dataset with the following dataspace, which has one unlimited dimension:

```
curr[0] = 0;
curr[1] = 1;
max[0] = H5S_UNLIMITED; /* or 1 if append
                         * in Y direction */
max[1] = 1; /* or H5S_UNLIMITED if append
             * in Y direction */
sid = H5Screate_simple(2, curr, max);
```

This compares the BT1 and EA indexing methods when appending to the dataset created with the old and the new library format respectively. Append operations are done along the X or Y directions.

c) Setting 3: creates 2-dimensional chunked datasets with the following dataspace, which has two unlimited dimensions:

```
curr[0] = 0;
curr[1] = 1;
max[0] = H5S_UNLIMITED;
max[1] = H5S_UNLIMITED;
sid = H5Screate_simple(2, curr, max);
```

This compares the BT1 and BT2 indexing methods when appending to the datasets created with the old and new library format respectively. Append operations are done along the X and/or Y directions.

TABLE I: Result for all three test settings before optimization.

Indexing Method	Time (s)	
Along the X-direction		
EA	149.82	$\simeq -4\%$
BT1	156.15	$\simeq +7\%$
BT2	166.56	
Along the Y-direction		
EA	150.08	$\simeq -8\%$
BT1	163.82	$\simeq +2\%$
BT2	167.91	
Along the XY-direction		
EA	–	
BT1	104.08	$\simeq +10\%$
BT2	114.39	

B. Preliminary Results

First, measures are realized on the library before optimization. The result listed in table I indicates that the EA indexing method performs better than BT1, while the BT2 indexing method performs worse than BT1. Note also that the library takes a fair amount of time for all three indexing methods.

C. Results

Measures are realized after optimization. The new results shown in table II indicate that after optimization BT2 performs better than BT1. General chunk index fixes have made a significant improvement for all three indexing methods—almost 80%.

As one can see in figure 5, for significant numbers of chunks, the BT2 and EA indexing methods perform better than BT1. EA shows good scalability and a much better performance than BT1/BT2 as the number of chunks increases. BT2 generally shows better performance than BT1 (it is also worth noting that even if this is not shown in this result, the amount of space taken by BT2 is reduced). Overall these results confirm the optimization made within the library and were what we expected given the complexity of the methods used.

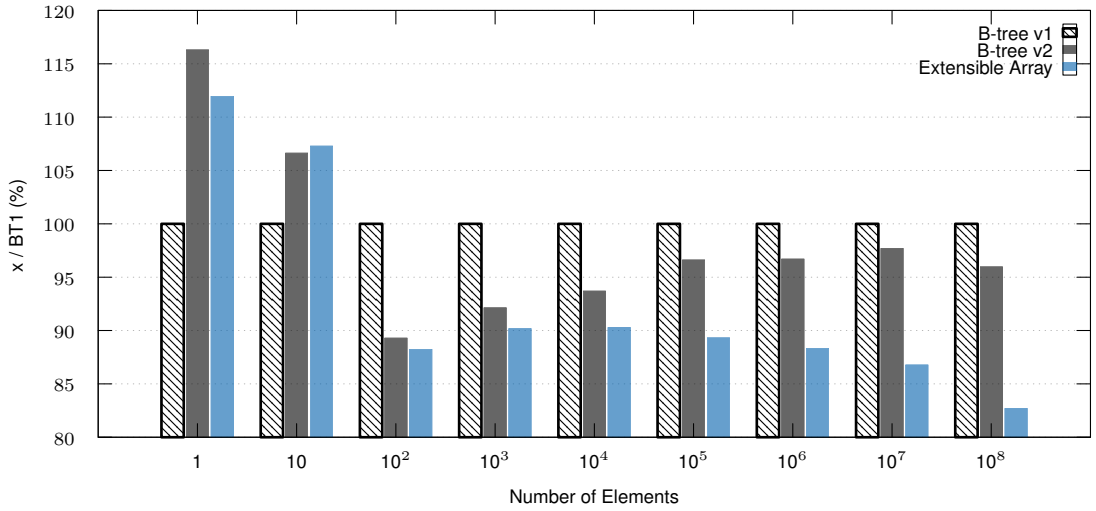


Fig. 5: Performance of BT2 and EA chunk indexing methods compared to BT1. The extensible array method shows good scaling performance, while the B-Tree version 2 shows improved results compared to BT1.

TABLE II: Result for all three test settings after optimization.

Indexing Method	Time (s)	
Along the X-direction		
EA	32.10	$\simeq -12.4\%$
BT1	36.63	
BT2	35.39	$\simeq -3.4\%$
Along the Y-direction		
EA	32.2	$\simeq -17.5\%$
BT1	39.01	
BT2	34.90	$\simeq -10.5\%$
Along the XY-direction		
EA	–	
BT1	36.11	
BT2	35.29	$\simeq -2.3\%$

IV. Related Work

Nimako et al. [7] consider another data structure for indexing chunks, the O_2-Tree, which performs query operations (searches) in $O(\log_2 n)$ time, where n is the number of internal nodes. They also mention that the conventional arrays and HDF5 incur the cost of reorganizing already allocated array elements. Based on these results and the new optimization made, it would be interesting to make a new comparison. The O_2-Tree may perform better than a B-Tree data structure but the re-sizable array data structure will still perform better in the single unlimited dimension case, as its complexity is in $O(1)$.

The universal B-tree [8], which is designed for multidimensional data, should also be another alternative to consider. The UB-tree is balanced and has all the guaranteed performance characteristics of B-trees, i.e. it requires linear space for storage and logarithmic time for insertion, lookup, removal of elements. In addition the UB-tree has the fundamental property that it preserves clustering of objects with respect to Cartesian distance. Therefore, the UB-tree shows its main strengths for multidimensional data. It also has very high potential for parallel processing, which makes it a very good candidate for further experimentation within the HDF5 library.

V. Conclusion and Future Work

The various enhancements made to the chunk indexing methods show good performance and whereas there is no gain in complexity with the *B-Tree version 2*, performance and space utilization have been significantly improved. For datasets that are extended in a single dimension, the extensible array offers a very good improvement compared to the standard B-tree, since elements can be accessed in constant time.

One component that is not taken into account in this work is the frequency of access of the elements. We should consider this in future work. An idea could be to make the *B-Tree version 2* dynamically adjustable by using methods such as the Huffman coding [9] already used for data compression, and which can represent the frequency of appearance of elements. Adding this component could benefit applications that frequently access specific regions of chunked data and further reduce the overhead introduced by the chunk indexing method.

References

[1] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and its Applications," in *Proceedings of the EDBT/ICDT Workshop on Array Databases*, ser. AD '11. New York, NY, USA: ACM, 2011, pp. 36–47.

[2] M. Folk, R. McGrath, and N. Yeager, "HDF: an update and future directions," in *International Geoscience And Remote Sensing Symposium (IGARSS)*, vol. 1, 1999, pp. 273–275 vol.1.

[3] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf, "Tuning HDF5 for Lustre File Systems," in *Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2010, LBNL-4803E.

[4] D. Comer, "Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[5] M. Folk and B. Zoellick, *File Structures*, 2nd ed. Boston, MA, USA: Addison-Wesley Publishing Company, Inc., 1992.

[6] A. Brodnik, S. Carlsson, E. Demaine, J. Ian Ian Munro, and R. Sedgewick, "Resizable Arrays in Optimal Time and Space," in *Algorithms and Data Structures*, ser. Lecture Notes in Computer Science, F. Dehne, J.-R. Sack, A. Gupta, and R. Tamassia, Eds. Springer Berlin Heidelberg, 1999, vol. 1663, pp. 37–48.

[7] G. Nimako, E. Otoo, and D. Ohene-Kwofie, "Chunked Extendible Dense Arrays for Scientific Data Storage," in *41st International Conference on Parallel Processing Workshops (ICPPW)*, Sept 2012, pp. 38–47.

[8] R. Bayer, "The Universal B-Tree for Multidimensional Indexing: General Concepts," in *Worldwide Computing and Its Applications*, ser. Lecture Notes in Computer Science, T. Masuda, Y. Masunaga, and M. Tsukamoto, Eds. Springer Berlin Heidelberg, 1997, vol. 1274, pp. 198–209.

[9] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sept 1952.