Embedding Indices and Bloom Filters in Parquet files for Fast Apache Arrow Retrievals

by

Arun Balajiee Lekshmi Narayanan

Submitted to the Graduate Faculty of

Department of Computer, School of Computing and Information

in partial fulfillment

of the requirements for the degree of

Master of Sciences

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE, SCHOOL OF COMPUTING AND INFORMATION

This thesis was presented

by

Arun Balajiee Lekshmi Narayanan

It was defended on

July 30th 2020

and approved by

Panos K. Chrysanthis, Department of Computer Science

Alexandros Labrinidis, Department of Computer Science

Daniel Mosse, Department of Computer Science

Thesis Advisors: Panos K. Chrysanthis, Department of Computer Science

Konstantinos Kosta, Department of Computer Science

Embedding Indices and Bloom Filters in Parquet files for Fast Apache Arrow Retrievals

Arun Balajiee Lekshmi Narayanan, M.S.

University of Pittsburgh, 2020

Apache Parquet is a column major table file format developed for the Hadoop ecosystem, with support for data compression. Hadoop SQL engines process queries like relational databases but read the parquet file to retrieve data. The caveat is that reading takes time and needs to be optimized. Irrelevant to a query I/O must be avoided for faster reads. The file is organized in rows segmented serially per column, which are segmented serially into DataPages. Two indices were proposed, namely, *ColumnIndex* (storing DataPage minimum and maximum values) and *OffsetIndex* (storing DataPage offsets), which support reading only the required DataPages in retrieving a row, skipping irrelevant DataPages.

In this thesis, we investigate methods to accelerate row retrieval in parquet files within Apache Arrow, which is an in-memory big data analytics library that supports fast data processing applications on modern hardware. Towards this, we first implement the proposed ColumnIndex and OffsetIndex. We then propose and integrate the indices with *Split Block Bloom Filters* (SBBF). Our hypothesis is that a combination of the indices and SBBF should enhance the overall performance by avoiding unnecessary I/O in queries with predicate values not present in the parquet file.

We validate our hypothesis through extensive experimentation. Our experiments show that using either indices or SBBF reduces average reading time by 20x. Their combination reduces the average reading time by an additional 10%. Adding indices does not significantly increase the parquet file size, but adding SBBF approximately increases the parquet file size by 2x. We contribute our code to Apache Arrow open source project along with a conceptual design for DataPage level SBBF for further read optimization.

Table of Contents

List of Tables

List of Figures

Preface

I want to acknowledge the valuable inputs provided by my advisors, Prof. Panos K. Chrysanthis and Dr. Constantinos Costa, throughout my work on this thesis. Without their countless reviews and revisions, this document would not look the same. I want to thank Deepak Majeti and Anatoli Shein, my mentors during the internship at Vertica, for assigning me the task of embedding Arrow parquet with ColumnIndex-OffsetIndex, which has now become a part of my work in this thesis. I want to thank my committee members, Prof. Alexandros Labrinidis and Prof. Daniel Mosse for taking time out of their schedule to be there at the defense, at short notice. I want to thank the PhD students at the ADMT lab for helping me out with the experimental setup and reviewing my thesis document. I want to thank Vineet K. Raghu, PhD, for introducing me to Prof. Panos, during his work during his PhD. I want to thank Nathan Ong for being a support during tough times. I want to thank Prof. Erin Walker for offering me the role as a programmer at her lab, which financially supported me during the Summer of 2020, when I was working on this thesis. I want to thank my friends from the FACETLab, for spending their valuable time in reviewing my defense presentation and this thesis document. I want to thank several of my friends from the CS department for being there with me during my tough days.

Finally, I dedicate my Thesis to my parents, sister and friends who have made me what I am today!

1.0 Introduction

MapReduce [1] was proposed to support fast keywords search, execute large dataset queries and search result retrievals. Its huge success pushed several competitors in the area to attempt at building frameworks that could abstract MapReduce, for other applications to implement. A well-known success in these terms is Apache Hadoop[1]. Hadoop File System (HDFS) [2] has been a popular platform for applications to leverage the MapReduce paradigm, in a distributed environment. Over the course of the last decade, since the inception of HDFS, several file formats have been designed to leverage the various features offered by HDFS [3]. All the file formats on HDFS developed over the years have been trying to achieve efficiency in being compatible with MapReduce [3]. Among these file formats, one recent success is Parquet[2].

Apache Parquet is efficient in leveraging the MapReduce framework and in comparison with few other popular file formats on HDFS, offers better support for skipping unnecessary data reads [4], compression and file reading times on HDFS [3]. Hadoop SQL engines execute queries on Hadoop file system (HDFS) by reading parquet files for row retrievals. Currently, parquet is used as the default file format to store databases by Hadoop SQL engines like Apache Spark[3], Impala [5], Hive[4] and Vertica [6], as parquet supports a few compression schemes to adapt the file sizes to the sizes of the Hadoop DataNodes (the data blocks in HDFS). Parquet uses the Record Shredding and Assembly algorithm[5] that allows for localized compression on a per column level for small files storing large amounts of data and new encoding schemes for flexibility in storage of different data types as they are invented. Parquet is a *write-once, read-many (WORM)* times format. If the data in a parquet file has to be updated, the whole file has to re-created.

Currently, parquet is transitioning out of the Hadoop ecosystem, towards being used by

[1] https://hadoop.apache.org/
[2] https://parquet.apache.org/documentation/latest/
[3] http://spark.apache.org
[4] http://hive.apache.org
[5] https://github.com/julienledem/redelm/wiki/The-striping-and-assembly-algorithms-from-the-Dremel-paper

cloud-based data warehouse services, such as AWS Redshift[6] offered by Amazon Web Services [7], which run on general linux-based file systems. The scope for use is being explored further in Extract, Transfer, Load (ETL) on IoT devices [8]. Therefore, adding improvements to the parquet file format has large impact for enterprises and businesses.

Apache Arrow is an in-memory big data analytics library that supports programming language independent, fast data processing applications on modern hardware. A Hadoop SQL engine can write columnar data in-memory, using Arrow and Arrow creates the parquet files with different configurations that is unique to the Hadoop SQL engine (as shown in Figure 1). In effect, Arrow is a library to copy and convert columnar storage data. Arrow provides Application Programming Interfaces (APIs) in multiple programming languages, including C++, for reading and writing files in parquet. For example, Vertica is a column store database, written in C++, that uses Apache Arrow, to store and read tables in parquet file format. The support in C++ for reading and writing parquet files is solely maintained by an open source community that also maintains Apache Arrow[7]. Spark uses Arrow to offer columnar data transfer benefits in Python programming language.

For storage efficiency, the parquet's stores data per column separately, allowing better compression on the datatype of a column. However, Arrow API retains the row representation of a table during data retrieval from columnar storage. This basically corresponds with the basic SQL query for row retrieval:

$$\textit{select * from file x where a = <search-value>;}$$

In Arrow, the above SQL query translates into an Arrow function call

$$\begin{aligned} &parquet :: ParquetFileReader :: Make(:: arrow :: MemoryPool \ * pool,\\ &std :: unique\_ptr < ParquetFileReader > reader,\\ &std :: unique\_ptr < FileReader > *out) \end{aligned}$$

Arrow offers low-level parquet file APIs for applications to use, improving the performance of parquet in Arrow has a larger impact in the parquet community. This observation has motivated the work in this thesis, the need to optimize the reading of parquet files.

[6]https://aws.amazon.com/about-aws/whats-new/2019/12/announcing-amazon-redshift-data-lake-export/

[7]https://github.com/apache/arrow/tree/master/cpp

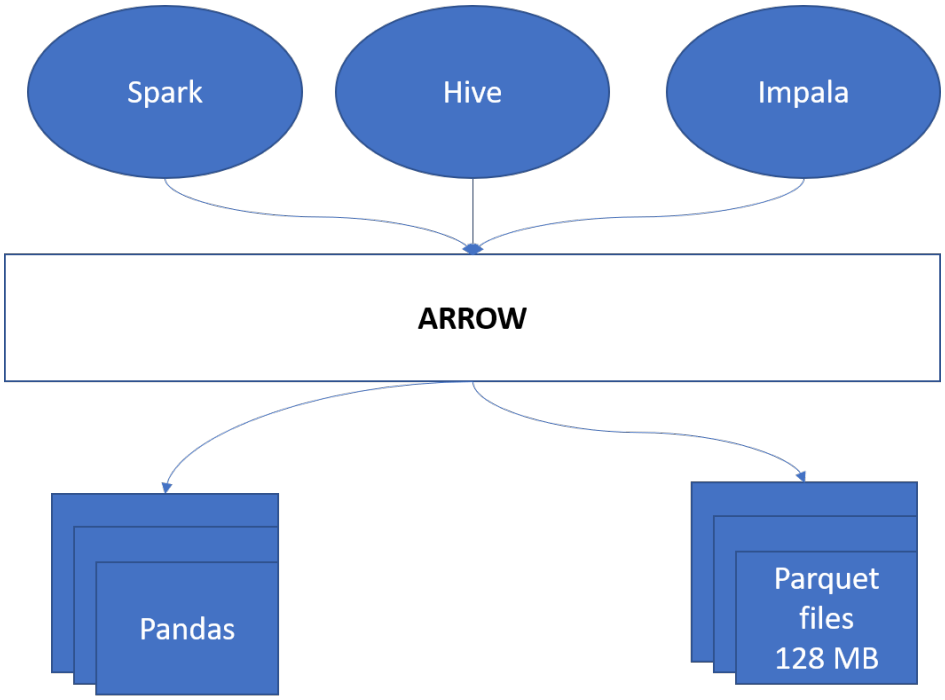


Figure 1: A simplified view different applications that use Arrow to perform in-memory computations and convert data into secondary data storage formats

1.1 Parquet State of the Art

Parquet file format was proposed in 2013 and since then, has gone through several structural changes and extensions. In Figure 2, we visually represent these changes to the parquet file format. The rows of the table are segmented into RowGroups[8]. A RowGroup is a collection of rows under all columns of the table. Each RowGroup is segmented further per column into ColumnChunks. Each ColumnChunk has the same number of corresponding rows as other ColumChunks. ColumnChunks of various data types are supported. The size of a ColumnChunk varies with the compression and encoding scheme for the data type in parquet. The ColumnChunk is further segmented into DataPages. A DataPage stores a subset of the collection of rows in a ColumnChunk. The DataPages are written in batches into a parquet file, with no support to update operations.

Each DataPage is the basic unit of data storage. As a result, MapReduce [1] can split independent, parallel threads of mappers and reducers to read separate DataPages per ColumnChunk per RowGroup for fast independent computation on portions of data. Hadoop SQL engines leverage the MapReduce, while executing queries to retrieve data from HDFS. This makes parquet a suitable file format to store databases on HDFS for Hadoop SQL engines.

For fast query processing by Hadoop SQL engines, reading data efficiently from parquet files is crucial. Since these files are practically large, the parquet files are configured using techniques and best practices [9] to support fast query execution. The most relevant technique to our work is indexing, since our work in this thesis builds on this idea. Each DataPage is appended with an index at the beginning that stores the details on minimum (min) and maximum(max) values of all the rows of that column in that DataPage, called Statistics, as shown in Figure 2. Hadoop SQL engines can use the Statistics to skip the DataPages, if the values in the DataPage does not have the range of values to be matched with the predicate. DataPage skipping reduces the reading time of a parquet file. However, skipping DataPages using Statistics still requires reading DataPages before discarding them.

[8]We use capitalization to match the terms in the parquet specification document. https://github.com/apache/parquet-format/blob/master/src/main/thrift/parquet.thrift

[9]https://parquet.apache.org/documentation/latest/

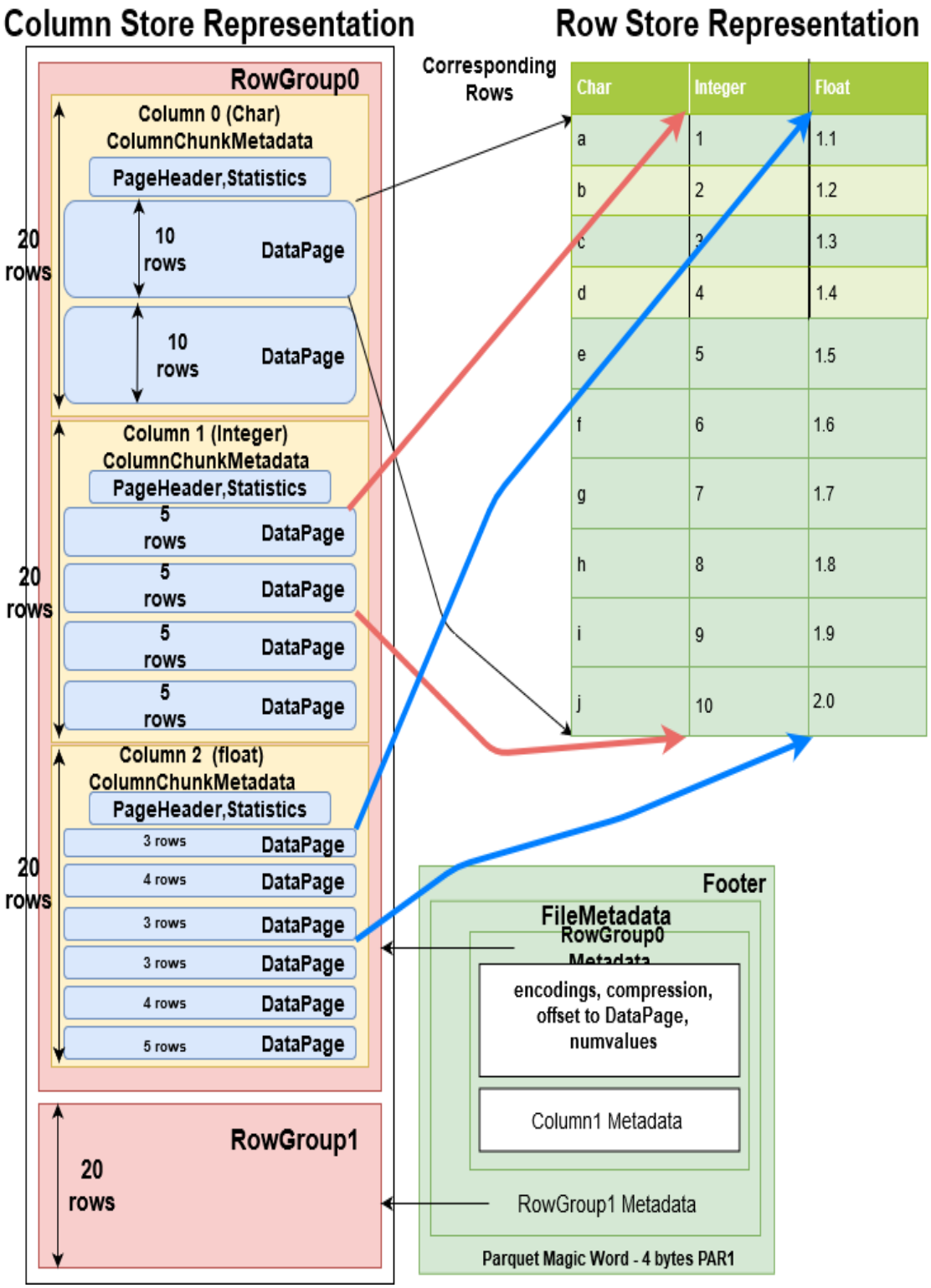


Figure 2: A simplified view of a RowGroup, its ColumnChunks and the DataPages arranged hierarchically and serially

That is, the entire file is still read before selecting the exact DataPage. The parquet community proposed a modification to optimize the use of Statistics in skipping DataPages. The parquet format introduced the *ColumnIndex and OffsetIndex* [10] to improve the file reading times for reading one value under a column from the entire file. Both, ColumnIndex and OffsetIndex, are embedded at the footer of the file. Loading ColumnIndex and OffsetIndex from a specific location in the footer of the file is faster. In fact, parquet file reads begin from the metadata at the footer.

For each DataPage in a ColumnChunk, the ColumnIndex stores an entry of the minimum and maximum values of the rows in the DataPage, originally stored in Statistics. This way, reading the Statistics at the beginning of every DataPage can be avoided. Statistics, as in Figure 2, also become unncessary and can be removed from the beginning of the DataPage to reduce the file size. To achieve complete optimization of file reading time, the community proposed the OffsetIndex to store information about the cardinal number (the number of the row in the row store representation of the table, in Figure 2) of the first row in the DataPage and the offset of the DataPage in the ColumnChunk. This way, the OffsetIndex and ColumnIndex store the information for all ColumnChunks in a RowGroup, separately. Using the ColumnIndex in combination with OffsetIndex could help in reading only the exact DataPage relevant for data retrieval from a ColumnChunk. Applications that use parquet for storage and retrieval, have to address this proposal with an implementation suitable to them. Arrow is yet to implement this in C++ for parquet.

1.2 Thesis Problem Statement

Impala, a Hadoop SQL engine, implemented the Parquet community proposal to use ColumnIndex and OffsetIndex in C++, ahead of all other applications that use parquet files. The Impala implementation presents an experimental evidence of improvements in parquet file read times and can be used as a benchmark.

[10] Lars Volcker and Marcel Kornacker, ColumnIndex Layout to Support Page Skipping, https://bit.ly/32u4aba

The ColumnIndex and OffsetIndex has not been implemented in the version of Arrow in C++ for parquet.

- *Aim 1: The first objective of this thesis is to utilize ColumnIndex and OffsetIndex to improve the parquet file reading time in Arrow C++.*

When using an index to scan data quickly, sometimes, we may not be aware of the fact that the data is not present in the parquet file. In such a case, using an index alone may not help in the reduction of reading time of the file. If a value is not present in the file, it is cost effective to avoid reading any DataPages. Bloom filters are designed to achieve this purpose. A Bloom filter may not be able to verify if a value that is present in the file is present, but can verify if a value that is not present in the file, is not present with high probability.

We hypothesize that using Bloom filters in combination with ColumnIndex & OffsetIndex should further improve parquet file read times. At the time of writing this thesis, parquet files with Bloom filters is not implemented either in Arrow C++ or in Impala. Combining Bloom filters with ColumnIndex and OffsetIndex is also yet to be explored by the community.

- *Aim 2: The second objective of this thesis is to utilize Bloom filters in combination with ColumnIndex and OffsetIndex in Arrow C++ to further improve the parquet file reading time in Arrow C++.*

1.3 Approach

In our implementation and experiments, we consider parquet files that are unencoded and uncompressed. Compression is an optimization for reducing file size, which does not have a direct impact on our goal of improving the reading time. However, our implementation uses the standard API offered by Arrow to read files with or without compression.

We first implement the serialization of ColumnIndex and OffsetIndex in Arrow C++ when writing parquet files. We setup the ColumnIndex and OffsetIndex for use while reading parquet files by implementing ColumnIndex and OffsetIndex deserialization on a per column-level in Arrow C++. We use the ColumnIndex to scan the minimum and maximum values

of DataPages and pick the right DataPage that has the matching range for the predicate. We use the corresponding OffsetIndex to get the cardinal number (the number of the row in table in row store format) of the first row of the relevant DataPage and use the SkipRows function in Arrow C++, to skip to the DataPage. We scan the values in the DataPage against the predicate to find the matching row. We measure the time taken for this file read operation to verify if using the ColumnIndex and OffsetIndex can improve file read times. Further, we measure the change in file size on adding ColumnIndex and OffsetIndex to the parquet file.

We consider implementing the Bloom filters at the level of ColumnChunks, in between RowGroups and at the level of DataPages, at the end of each ColumnChunk. We implement the serialization and deserialization of column-level Split Block Bloom Filters (SBBF) [11] per ColumnChunk in a RowGroup and propose a design for the structural implementation of DataPage-level SBBF, for further read optimization, in Chapter 5. The column-level SBBF is implemented per column and the predicate is tested with the specific SBBF of the column to be accessed to retrieve a value. We deserialize the SBBF on a per column-level to be used during file reads to access a value. We check if the value exists in that column before proceeding to scan DataPages. We measure the time taken for this file read operation to verify if using the SBBF can improve file read times and the change in file size on adding SBBF to the parquet file. Further, we profile the experiments for complete analysis.

After implementing the methods to serialization and deserialization of ColumnIndex, OffsetIndex and SBBF in Arrow C++, we proceed with utilizing them in combination with each other. We implement flags that choose to use only the SBBF, the ColumnIndex and OffsetIndex or both in combination. We measure the differences in file reading times of parquet files in these different scenarios. Further, we profile the experiments for complete analysis. From our results, we show a successful implementation of ColumnIndex and OffsetIndex as well as a successful implementation of combining Bloom filters with ColumnIndex and OffsetIndex in Arrow C++ to propose to the parquet community.

[11] https://github.com/apache/parquet-format/blob/master/BloomFilter.md

1.4 Contributions

Motivated by the importance of the parquet files in efficient processing of big relational data, in this thesis, we design and implement the *PittCS Arrow parquet file*, which optimizes the reading of parquet files in Apache Arrow C++. Specifically, we make the following four contributions:

- Implement serialization and deserialization of ColumnIndex and OffsetIndex to support fast DataPage skipping and utilizing the ColumnIndex-OffsetIndex to reduce parquet file reading time in Arrow C++.
- Implement serialization and deserialization of column-level Split Block Bloom Filters (SBBF) and proposing the novel idea of utilizing Bloom filters in combination with the ColumnIndex and OffsetIndex to further reduce parquet file reading time in Arrow C++.
- Evaluate experimentally the reading of PittCS Arrow and found that:
  (i) using ColumnIndex-OffsetIndex reduces average parquet file reading time by 20x;
  (ii) using SBBF reduces the file reading time by 20x; and
  (iii) using SBBF in combination with ColumnIndex-OffsetIndex by an additional 10% improvement in comparison with using plain ColumnIndex-OffsetIndex or plain SBBF.
- Propose the DataPage-level SBBF as a two-level Bloom filter to use in combination with ColumnIndex-OffsetIndex and column-level Bloom filter for further read optimization.

We contribute our code and our performance evaluation results to the Arrow C++ for parquet community via github[12].

1.5 Roadmap

In the next chapter, we provide the necessary background on parquet files and put our work in the context with respect to existing file formats for storing big data. Specifically, we complete explaining parquet, ColumnIndex-OffsetIndex and Bloom filters in detail.

[12] https://github.com/a2un/arrow/

In Chapter 3, we present the main contribution of this thesis, the design and implementation of PittCS Arrow, which optimizes Arrow C++ parquet files. We first introduce Arrow and discuss the parquet reader and writer in Arrow C++ in detail. Subsequently, in reference to Arrow, we explain our implementation, PittCS Arrow, for reading parquet files with Bloom filters, ColumnIndex-OffsetIndex and combination of Bloom filters with ColumnIndex-OffsetIndex in detail. In Chapter 4, we discuss the instrumentation and performance evaluation of PittCS Arrow. Further, we present and analyze the results in detail, which support our two aims in this thesis in optimizing parquet files in Arrow C++. In Chapter 5, we conclude our thesis with a summary of our contributions and future work. In particular, we discuss our design to implement DataPage-level Bloom filters as a final contribution in this thesis. Lastly, we proceed to discuss the larger implications of PittCS Arrow in open source parquet community.

2.0 Background & Related Work

In this chapter, we provide the necessary background on parquet files and put our work in the context with respect to existing file formats for storing big data. Specifically, we complete explaining parquet in Section 2.1, ColumnIndex-OffsetIndex in Section 2.2, Bloom filters in Section 2.3 in detail. We proceed to discuss related work in the area of columnar storage, implementation of columnar storage in file formats and the recent applications of Bloom filters in databases in Section 2.4.

2.1 Parquet File Format & Layout

2.1.1 Parquet file components

As mentioned in the introduction, the following are the components of a parquet file, as in Figure 3:

- *RowGroup*: Stores a collection of rows.
- *ColumnChunk*: Stores the rows, of a RowGroup, in a column.
- *DataPage*: Stores a portion of the rows.
- *Dictionary Page*: Stores the information regarding the number of times a value in a row repeats in a column. It is written before all DataPages, at the beginning of the ColumnChunk.
- *ColumnChunkMetadata*: Stores the offset to the first data page in a ColumnChunk, compression codec, encoding scheme
- *RowGroupMetadata*: Stores the ColumnChunkMetadata.
- *FileMetadata*: Stores the file version, table schema.
- *Footer*: Stores the FileMetadata and RowGroupMetadata.

Parquet only allows and supports *int*32 (32 bit integers), *int*64 (64 bit integers), *int*96 (a mixed type of unsigned integers of 32 bits to support reading and writing datetime data),

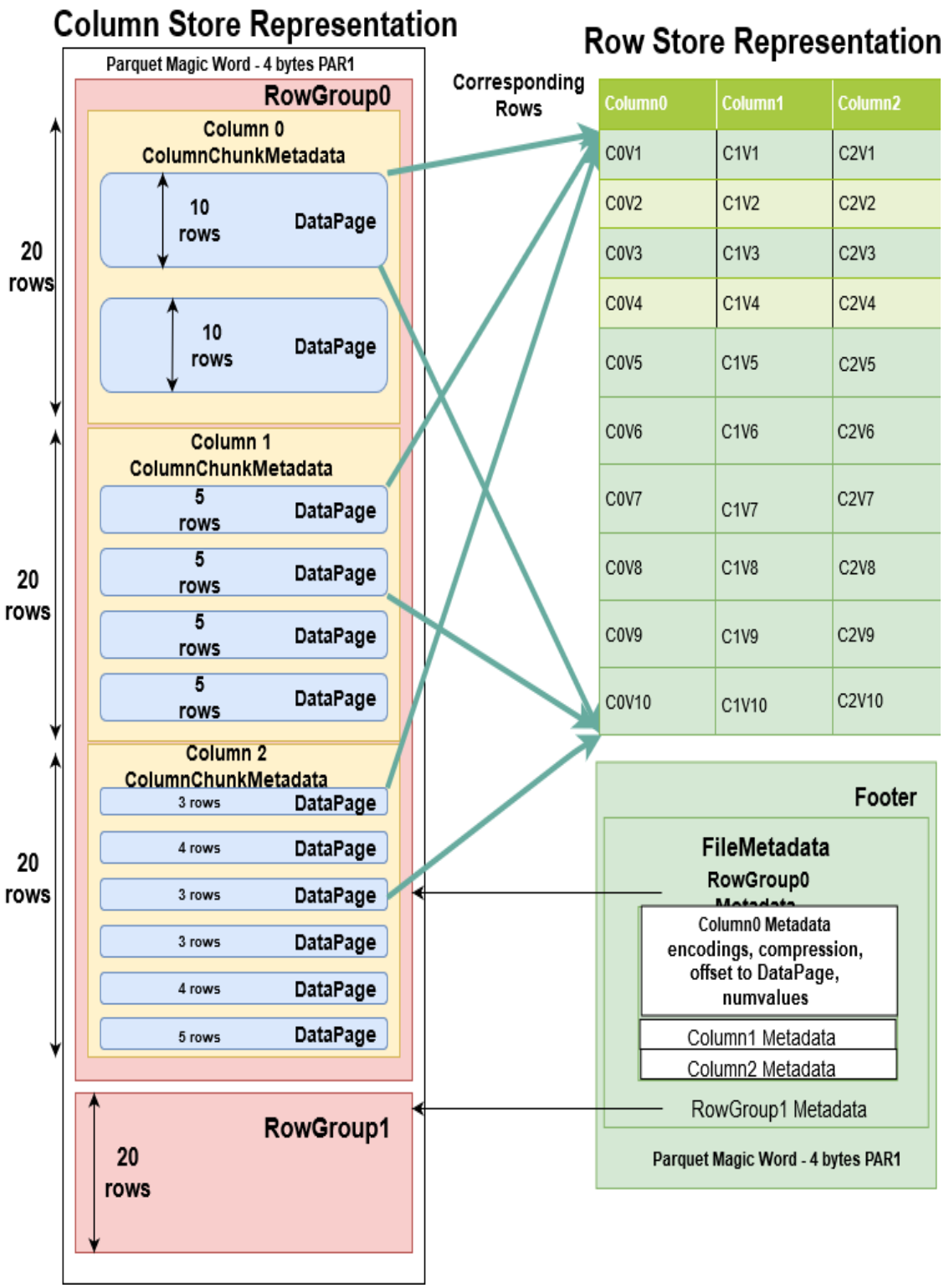


Figure 3: Parquet row alignment

float, *double*, *ByteArray* (strings) and *FixedLengthByteArray* (strings with fixed length) for columns in any schema.

The format is explicitly designed to separate the FileMetadata from the data. This way, one instance of the metadata can reference to multiple portions of the table stored in multiple parquet files.

We observed earlier, in Section 1.1, that the parquet file has an hierarchical structure. In the Figure 3, on the right, we have a row store representation of the data in a table and on the left, we have a column store representation of the same data in the form of a parquet file. The data under each column is serially written, in the order that the column appears from left in the table. The RowGroup0 has the corresponding twenty rows of the table, each ColumnChunk has the corresponding twenty rows under the corresponding column of the table. ColumnChunk 0 has the rows under Column 0, ColumnChunk 1 has the rows under Column 1 and ColumnChunk 2 has the rows under Column 2. Each ColumnChunk has DataPages, which are the basic unit of data storage. In ColumnChunk 0, two DataPages contain ten rows each, in ColumnChunk 1, four DataPages store five rows each and in ColumnChunk 2, each DataPage has variable number of rows, adding up to twenty rows.

The parquet file hierarchy allows a column-level compression of data. In Figure 3, we consider that the data type of Column 0 allows lesser compression than Column 1 and Column 1 allows lesser compression than Column 2. Specifically, the same number of rows under Column 0 can only be stored in two DataPages, Column 1 has four DataPages per ColumnChunk and Column 2 has six DataPages stored per ColumnChunk. This way, the number of DataPages in a ColumnChunk varies with the type of compression applied.

Even if the number of DataPages in each ColumnChunk is different, the corresponding rows in each DataPage coincide in the three ColumnChunks in a RowGroup.

2.1.2 Parquet thrift template

Apache Thrift [1] is a data serialization format that allows defining data types in a template file. Components of a parquet file are defined in a template in a language independent

[1] https://thrift.apache.org/

interface[2]. Arrow C++ uses the TCompactProtocol[3], written by parquet community, to deserialize the template file.

2.1.3 Parquet data encoding schemes

Encoding schemes in parquet files set the definition and repetition levels, which are defined below:

- *Definition level*: Stores information on the count of null values per column to avoid repetition.
- *Repetition level*: Stores the information on the level (DataPage in a ColumnChunk) the value repeats.

Parquet files currently support the plain (no definition or repetition levels), plain dictionary (DictionaryPage encoding to store information of the rows in the DataPage) and Run Length Encoding (RLE) encoding schemes. In plain encoded parquet files, the repeating values occur as is. In files with encoding, the values are encoded by the scheme and the repeating values are only stored once in a DataPage.

Early versions of Arrow C++ and Impala only support RLE.

2.1.4 Parquet data error prevention & recovery

Data corruption in a parquet file is when the data is unreadable because of a corruption in the metadata. This means that if the FileMetadata is corrupt the parquet file cannot be recovered. Similarly, if the RowGroupMetadata is corrupt then that RowGroup cannot be recovered. Again, if a ColumnChunkMetadata is corrupt the ColumnChunk of that RowGroup cannot be recovered. Finally, if a DataPage header is corrupt then that DataPage and the other following DataPages in that ColumnChunk also become corrupt.

When writing large amounts of data into a file, smaller RowGroups distribute the data across the file. Using RowGroupMetadata, individual RowGroups of a file can be recov-

[2]https://github.com/apache/parquet-format/blob/master/src/main/thrift/parquet.thrift
[3]https://github.com/apache/thrift/blob/master/lib/cpp/src/thrift/protocol/TCompactProtocol.h

Table 1: Configurable parameters in a parquet file

Configurable Parameter	**Value**
Number of a RowGroups	>= 1
Size of a RowGroup (HDFS)	<=256 MB
Size of a DataPage (HDFS)	<=1 MB

ered. Hence, parquet files with small sized RowGroups have lesser chance of losing data on corruption.

2.1.5 Parquet file configurations

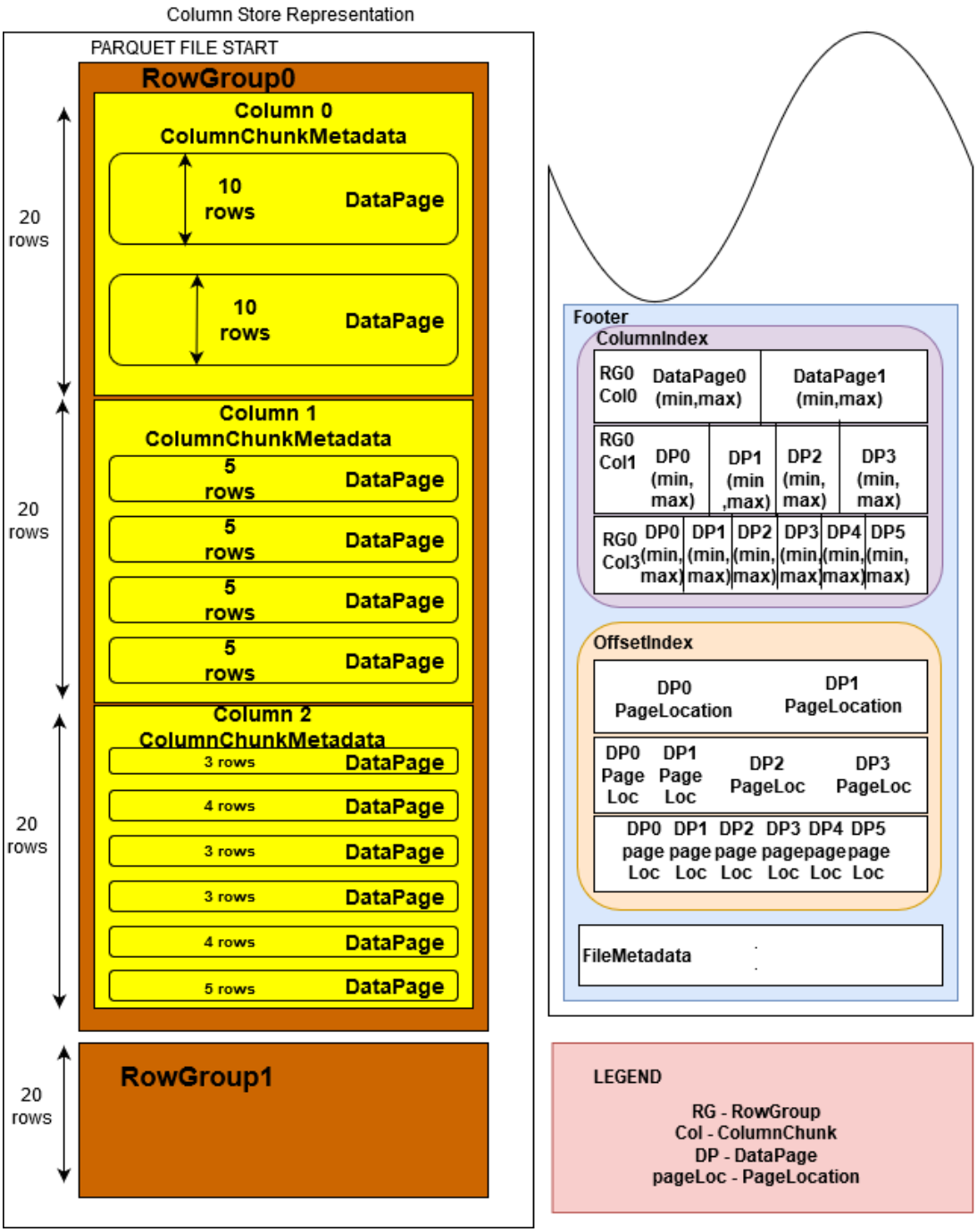
Parquet files can be configured with RowGroup and DataPage sizes, based on the file system used to store parquet files.

In a Hadoop File System (HDFS), as seen in Table 1, each file block is of size 256 MB. In this file system, the parquet files generally are configured with 1 RowGroup of size 256 MB and DataPages are typically of size 64 kB, with a maximum size of 1 MB.

In a linux file system, the file block size is 4 kB. The parquet file size on this file system has no upper limit and can have more than one RowGroup with DataPages of any size. Based on these values, other factors such number of DataPages in a ColumnChunk also get pre-configured at the time of writing a parquet file. Similarly, the number of ColumnChunks varies with the number of columns in the table written into a parquet file.

2.2 ColumnIndex and OffsetIndex

ColumnIndex and OffsetIndex are placed at the beginning of the file footer. As shown in Figure 4, the ColumnIndex and OffsetIndex store the following information:



ColumnIndex and OffsetIndex Layout

Figure 4: ColumnIndex-OffsetIndex store entries corresponding to DataPages

- *ColumnIndex*: Stores minimum and maximum values per DataPage per ColumnChunk in one RowGroup.
- *OffsetIndex*: Stores cardinal number (the number of the row in a table in row store format) of the first row in a DataPage and page offset per DataPage per ColumnChunk in one RowGroup.

2.3 Bloom filters

The parquet community proposed a specialized form of Bloom Filters, called the *Split Block Bloom Filters (SBBF)* [4], inspired from the article on network applications of Bloom filters [9] by Broder and Mitzenmacher. Broder and Mitzenmacher talk about splitting Bloom filters to achieve efficient hashing and low false positive probability. To build a Bloom filter for big data, with space and hashing efficiency, the parquet community takes inspiration from Putze et al. [10], to build a block Bloom filter.

SBBF are a special type of Bloom filters, that contain n blocks of 256 bits each. Each block has serially aligned 8 words of 32 bits each set to zero. Figure 5 gives a simplified view of an initialized Split Block Bloom Filter. Additionally, the SBBF is initialized with a preset false positive probability. SBBF has a differing number of blocks, n, based on the false positive probability, as a parameter. To hash a value in the SBBF, the value is converted into its 64-bit equivalent. The most significant 32-bits are used in selecting the block to perform the hash. The least significant 32-bits are used to hash into the block. To verify if a value is present in the Bloom filter, the value is converted into its 64-bit equivalent. The most significant 32-bits are used to select the block to verify the hash. The least significant 32-bits are used to hash, which is verified with the block.

To hash a value in a block, 8 different hashes are performed on the value. Each hash flips one bit in each word of the block. Hence, one bit is flipped in all the 8 words in a block, as shown in Figure 6.

[4]https://github.com/apache/parquet-format/blob/master/BloomFilter.md

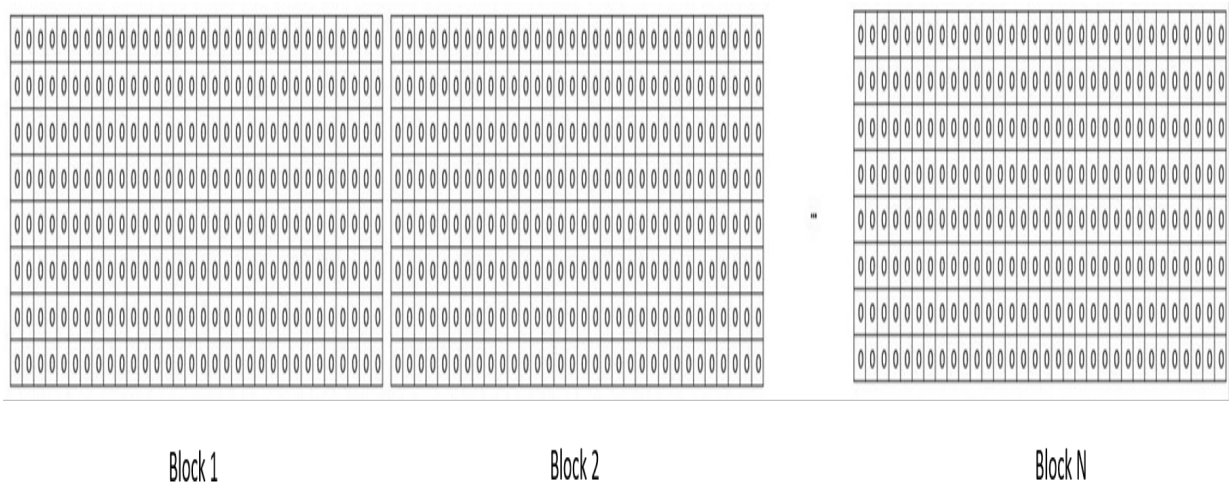


Figure 5: A simplified view of Split Block Bloom Filters

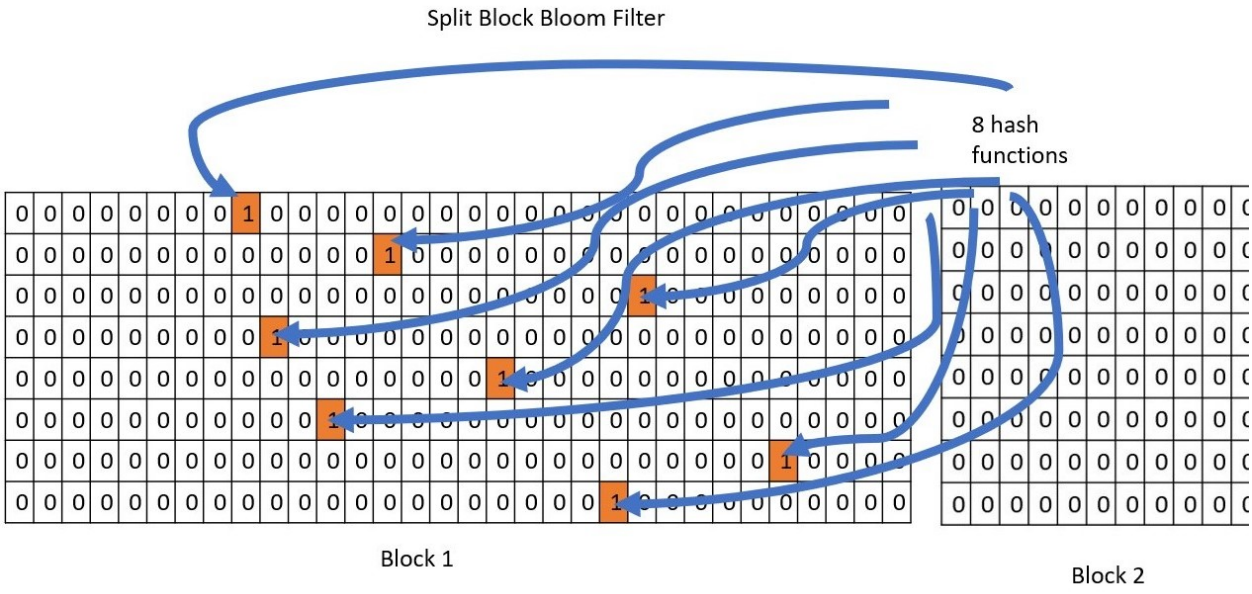
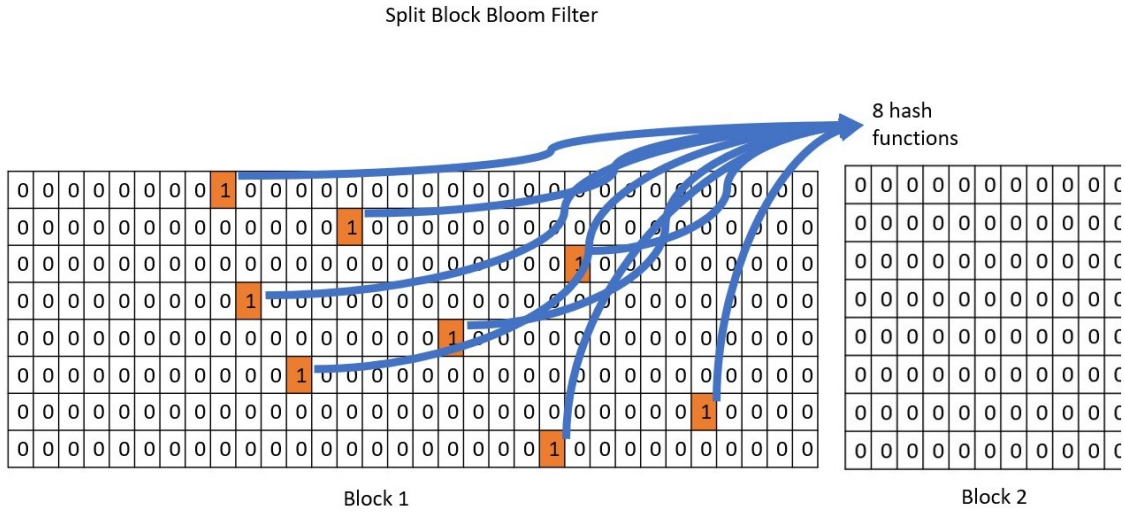
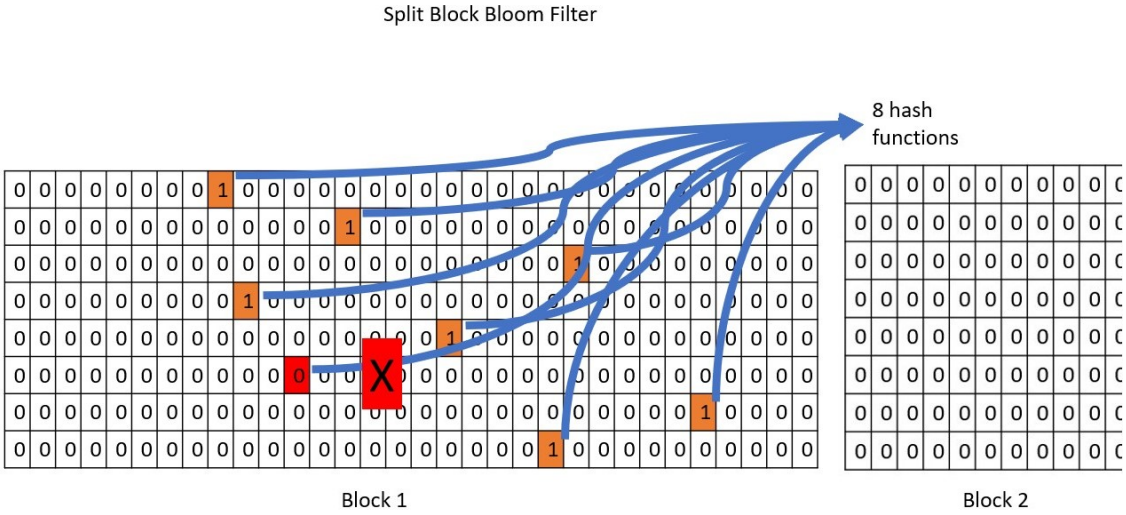


Figure 6: Block insert. One bit is flipped in each of the 8 words in a block.



If all bits match, the value probably exists in the data.

Figure 7: Block check - matching hash. The value is hashed and verified with the bit in each word.



If at least one bit mismatches, the value is not present in the data with high probability.

Figure 8: Block check - mismatch in hash. The value is hashed and verified with the bit in each word.

To verify if a value is present in the set, the value is hashed 8 times, to verify if it matches every word in the block. Only a perfect match considers the value to be a "probable" member of the set. The value can be verified as not present in the Bloom filter with high probability, when there is a mismatch of bits in at least one word. Figures 7 and 8 depict this visually.

2.4 Related Work

The prevailing notion in database storage formats are of two types—row–oriented and column–oriented (or columnar or C–store) databases.

The traditional storage model for relational databases is the Row or N-ary Storage Model (NSM) (a.k.a slotted Pages) [11], which stores relational databases in disk blocks, in a sequence of N-tuples for the rows of the table. NSM does not offer cache-efficiency in query processing, that is reading the data. In order to mitigate this inefficiency, the concept of indexing was proposed for fast access to store data blocks. The most commonly used index in relational database is B+-tree.

In query processing applications such as analytics, cache utilization and performance is becoming increasingly important. Ailamaki, et al. [12, 13] present that cache-efficiency can improve query performance by grouping data in pages. They propose an extension to NSM called PAX that only changes the layout inside the pages to support caching and not the I/O costs of the query. This way, the memory bandwidth reduces by 75% and the queries run faster by 11%. PAX stores the values under each column into mini–pages within the disk pages, contiguously in a column–major format. In effect, PAX is a quasi-columnar format similar to parquet. While this format is suitable for relational databases, they are not adopted by new forms of databases.

Columnar databases are the more recent formats proposed to support OLAP (OnLine Analytical Processing). MonetDB [14], H–Store [15] and C–Store [16] are some examples of columnar databases. Data stored in columnar format is convenient to read, com-

press/decompress, and process specific values required by a specific transaction on specific column projections of the table, such as in Vertica [6]. Vertica is a big data analytics platform that uses various columnar file formats on Hadoop file system (HDFS) clusters.

An innovation in columnar big data file format is Dremel, by Melnik et al. [17], an implementation that helps in fast web search query retrievals using MapReduce. The records are shredded into different portions of data and stored in a dictionary-like key-value pair format. The paper discusses the idea of definition and repetition levels in data for fast access to keywords in the data. These keywords help access the information at different portions of data in one or two read operations, reducing the reading time of the file. Caching the accessed information further improves the data retrieval times. Parquet takes inspiration from the explanation in the Dremel implementation to create definition and repetition levels for data encoding.

Optimzed Row Columnar (Orc)[5] is the columnar file format to store data by Hive SQL engine on Hadoop. Orc supports Atomic, Consistent, Isolated, Durable (ACID) transactions, data stream processing with smaller file sizes. The Orc writer chooses the correct encoding for the file data at runtime in the form of columnar "stripes" and builds an internal index of the data. Stripes are a directory of data stream locations. When a Hive SQL query reads data from an Orc file, Hive uses predicate pushdown to compare the predicate using the indexes stored in the Orc file to filter stripes in a file relevant to the query and narrows the search to a set of rows. This reduces the query run time by significant amounts. Parquet takes inspiration from Orc for its own implementation of ColumnIndex and OffsetIndex to support predicate pushdown.

While using ColumnIndex and OffsetIndex has been the prevailing solution to improve query run time and reduce file reading times, using Bloom filters for this purpose is rarely explored. Bloom filters [18], by Burton H. Bloom, is the idea of the trade–offs between space and hashing efficiency. Several instances of use of Bloom filters can be quoted, but we focus on the implementation in a big data application, namely, Bigtable [19]. Bigtable uses the Google *SSTables* file format to store large amounts of data. SSTables contain persistent key-values pairs that are arbitrary byte strings. SSTables contain a sequence of blocks, each

[5]https://orc.apache.org/docs/

of typical size 64 kB. Optionally, SSTables can be mapped to the memory directly, instead of disk storage. SSTables add several refinements in the implementation, including Bloom filters. Bloom filters in Bigtable act as gatekeepers to restrict disk read accesses during row retrievals from SSTables. Bloom filters in Bigtable reduce disk accesses for reads and the I/O costs. Apache Kudu[6] [20] is a relational database system, with a structurally similar data storage system to that of Bigtable. In effect, Kudu is similar to parquet file format in terms of storing data in columnar format, but retrieving results in row store representation. It should be noted that Kudu and PittCS Arrow implementations for Bloom filters were concurrently developed–Kudu's code was released in June 2020. Impala uses Bloom filters in Kudu as a part of its query engine and does not store them in parquet files as in PittCS Arrow. That is, to reduce the number of rows retrieved during table join operations in query executions, Impala uses Bloom filters to achieve nearly 20x improvements in query executions on Kudu[7] in the Hadoop file system. Along the lines of these implementation, our implementation of a Bloom filter is to be the first–level check before using ColumnIndex and OffsetIndex, in order to reduce parquet file reading time with lesser I/O operations.

To summarize the current-state-of-art, Table 2 shows the different columnar file formats and their supported features to improve file reading times. It is clear that no Parquet implementation supports the features of PittCS Arrow parquet proposed in this thesis.

2.5 Chapter Summary

In this chapter, we completed explaining parquet, ColumnIndex-OffsetIndex and Bloom filters in detail. Further, we discussed the background and related work in the area of columnar file formats similar to parquet. In the next chapter, we present the main contribution of this thesis, the design and implementation of PittCS Arrow, which optimizes Arrow C++ parquet files. We first introduce Arrow and discuss the parquet reader and writer in Arrow C++. Subsequently, in reference to Arrow, we explain our implementation, PittCS Arrow,

[6]https://kudu.apache.org/overview.html
[7]https://issues.apache.org/jira/browse/IMPALA-3741

Table 2: Taxonomy of the state-of-art-implementations and our contribution PittCS Arrow

Application, library, database	**Supported file format**	**Programming Language**	**Statistics**	**ColumnIndex & OffsetIndex**	**Bloom filters**
MonetDB	Plain file	C	N	N	N
H–Store	Plain file	C++/Java	N	N	N
C–Store	Plain File	C/C++	N	N	N
Bigtable	SSTable	Java	N	N	Y
Kudu	Kudu	C++	N	N	Y
Vertica	Orc	C++	N	Y	N
Hive	Orc	Java	N	Y	Y
Arrow	Parquet	C++	Y	N	N
Vertica	Parquet	C++	Y	N	N
Hive	Parquet	Java	Y	N	N
Spark	Parquet	Scala	Y	Y	N
Impala	Parquet	C++	Y	Y	N
PittCS Arrow	Parquet	C++	Y	Y	Y

for reading parquet files with Bloom filters, ColumnIndex-OffsetIndex and combination of Bloom filters with ColumnIndex-OffsetIndex in detail.

3.0 Optimizing Arrow C++ parquet files

In this chapter, we present the main contribution of this thesis, the design and implementation of PittCS Arrow, which optimizes Arrow C++ parquet files. We first introduce Arrow and discuss the parquet reader and writer in Arrow C++ in detail in Section 3.1. Subsequently, in reference to Arrow, we explain our implementation for reading parquet files with Bloom filters, ColumnIndex-OffsetIndex and combination of Bloom filters with ColumnIndex-OffsetIndex in detail in Section 3.2.

3.1 An Arrow Primer

Arrow is a derivative of the Apache Parquet project. Arrow is highly efficient in performing vector in-memory computations [1]. Arrow takes inspiration from the parquet file format to create its own version of "Arrow" files that are vectorized memory-mapped abstraction of parquet files. Arrow project is the primary maintainer for the parquet file format and for the use of parquet in C++. Arrow offers low-level API for in-memory operations on columnar data that is used by Spark and Vertica. A recent considerable interest has been generated in using the efficient low-level API offered by Arrow, for reading and writing parquet files on Field Programmable Gate Arrays (FPGA) and is an actively researched topic [21].

As parquet files are WORM files, Arrow C++ implements two operations on parquet files, *Create(write) and Read*

$$arrow :: io :: FileOutputStream :: Open(\text{"}test.parquet\text{"});$$
$$parquet :: WriterProperties :: Builderbuilder;$$
$$parquet :: ParquetFileWriter :: Open(outfile, schema, builder.build());$$

[1] https://arrow.apache.org/faq/

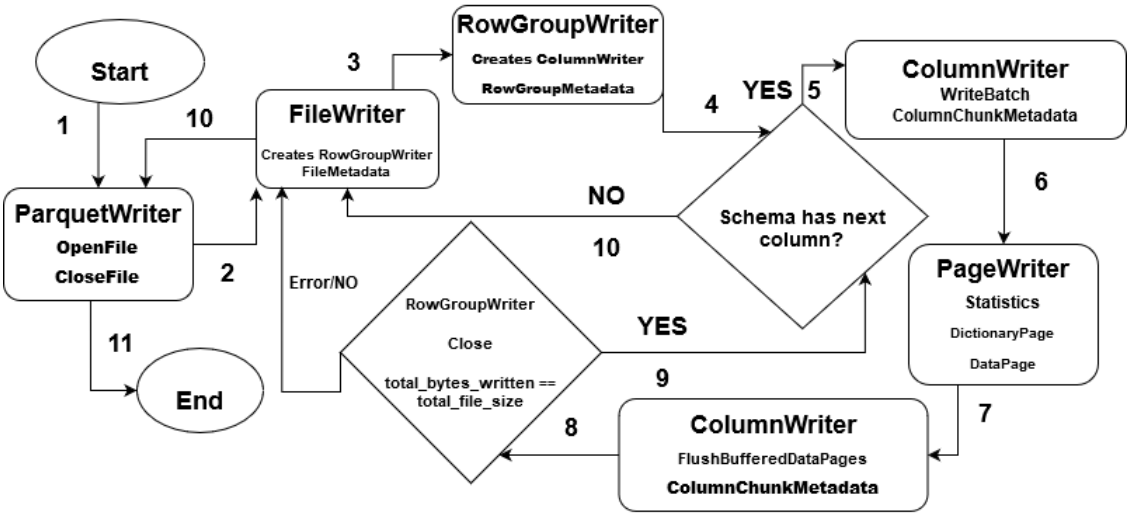


Figure 9: A simplified flow diagram with numbers to indicate the direction of flow of creating a Parquet file in Arrow C++.

to create parquet files, and

$$parquet :: ParquetFileReader :: Make(:: arrow :: MemoryPool \ *pool,$$
$$std :: unique\_ptr < ParquetFileReader > reader,$$
$$std :: unique\_ptr < FileReader > *out)$$

to read parquet files.

3.1.1 Arrow C++ Flow of Control for Creating parquet files

To start the process of writing data in a parquet file, Arrow creates an instance of ParquetWriter. Using a default setup or based on the configurations provided by the user, an instance of ParquetWriter generates the parquet file schema. Schema contains the information on the columns of the table, the data type, repetition and definition levels of each column.

Specifically, to explain the flow of control in Arrow C++ to write data into a parquet file, we follow the steps in Figure 9.

Step 1: The schema is generated for the parquet file is generated.

Step 2: The ParquetWriter creates a FileWriter instance, using the generated schema. FileWriter writes the magic word "PAR1" at the beginning of the file to start the process of writing.

Step 3: FileWriter creates instances of RowGroupWriters, in sequence, for all RowGroups to be written in a file.

Step 4: Each RowGroupWriter creates instances of ColumnWriters for each column in the schema. The RowGroupWriter stores the offsets to the ColumnChunkMetadata for each column in the RowGroupMetadata.

Step 5: Each ColumnWriter writes the ColumnChunkMetadata with the statistical information and offsets to DataPages.

Step 6: Each ColumnWriter calls instances of PageWriter, which finally writes rows into the DataPages in batches. The PageWriter also considers the compression and encoding schemes while writing the DataPages, to create DictionaryPages whenever necessary.

Step 7: Once the PageWriter completes writing all rows of the column, it returns the number of bytes of data written to ColumnWriter.

Step 8: The ColumnWriter returns the number of bytes written by PageWriter to RowGroupWriter. The RowGroupWriter verifies, using the schema information, that the number of bytes written is of the same size as the size of data. On verification, the RowGroupWriter proceeds to the next step.

Step 9: The RowGroupWriter proceeds to writing data for the next ColumnChunk. This process repeats for all ColumnChunks in a RowGroup, serially.

Step 10: Once a RowGroup is successfully written, the RowGroupWriter returns the RowGroupMetadata to the FileWriter. The FileWriter checks if more RowGroups have to be written and the repeats the process. Once all RowGroups in the file are written into the file, the FileWriter writes the FileMetdata and RowGroupMetadata, for all RowGroups, in the footer. The FileWriter completes by writing the byte size of the FileMetadata at the end of the file and a string "PAR1" of size four bytes, to identify the parquet file. The offset to the footer of the file is the byte size of the FileMetadata. During a parquet file read, the parquet reader makes an initial seek to the end of the file to read the size of the

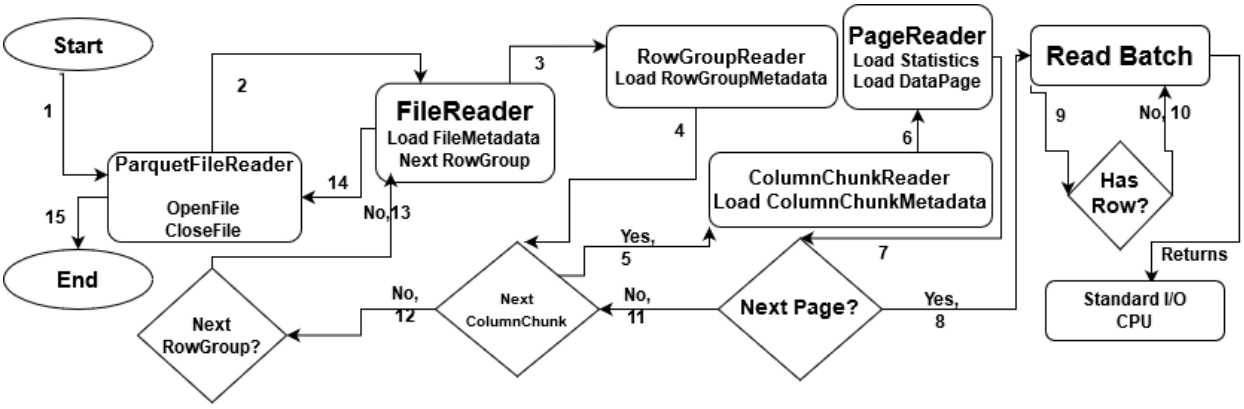


Figure 10: A simplified flow diagram with numbers to indicate the direction of flow of reading a Parquet file in Arrow C++

FileMetadata. The reader seeks backwards by this size to the beginning of the footer.

Step 11: The ParquetWriter then closes the file descriptor to complete the write operation of the parquet file. This marks the end of the control flow.

3.1.2 Arrow C++ Flow of Control for Reading parquet files

Since the parquet file is segmented, different sections are processed to read the contents of the parquet file. To start reading, Arrow creates an instance of ParquetReader with the file name and schema.

We explain the control flow for reading a parquet file in Arrow, using the steps shown in the Figure 10.

Step 1: The ParquetReader creates a file descriptor to load the file.

Step 2: The ParquetReader instance invokes a FileReader instance. The FileReader creates a file descriptor and begins by seeking to the end of the file, to read the size of the FileMetadata. The FileReader seeks backwards by that size to the beginning of the footer, which is also the beginning of the FileMetadata.

Step 3: The FileReader deserializes the FileMetadata to invoke RowGroupReader instances. Each RowGroupReader instance loads the RowGroupMetadata the respective

RowGroup.

Step 4: RowGroupReader invokes separate instances of ColumnChunkReader for each ColumnChunk in the RowGroup.

Step 5: ColumnChunkReader serially invokes PageReader for each DataPage.

Step 6 & 7: The PageReader calls ReadBatch to read DataPages serially.

Step 8 & 9: ReadBatch reads the values the rows in the DataPages and returns the values to standard I/O on reaching the end of the DataPage. This marks the end of the reading control flow. However, internally Arrow follows a few steps to close the file which is not part of the reading control flow.

Step 10: PageReader exhausts reading all the DataPages in a ColumnChunk,
the ColumnChunkReader instance is also closed.

Step 11 & 12: This process is repeated by the FileReader for other RowGroups.

Step 13: FileReader returns control to ParquetFileReader and the file is closed.

3.2 PittCS Arrow Parquet files

We modify the parquet creating (writing) control flow in Arrow C++ (Figure 9) to generate parquet files that store Bloom filters in between RowGroups and ColumnIndex-OffsetIndex at the footer. We modify the parquet reading control flow in Arrow C++ (Figure 10) to read parquet files either with ColumnIndex and OffsetIndex or with Bloom filters or with the combination of Bloom filters and ColumnIndex-OffsetIndex.

3.2.1 PittCS Arrow Parquet Writer

Writing ColumnIndex

We use Thrift deserialization methods in C++ to write ColumnIndex and OffsetIndex at the footer of the parquet file. Figure 11 represents our modifications to the control flow to introduce OffsetIndex and ColumnIndex.

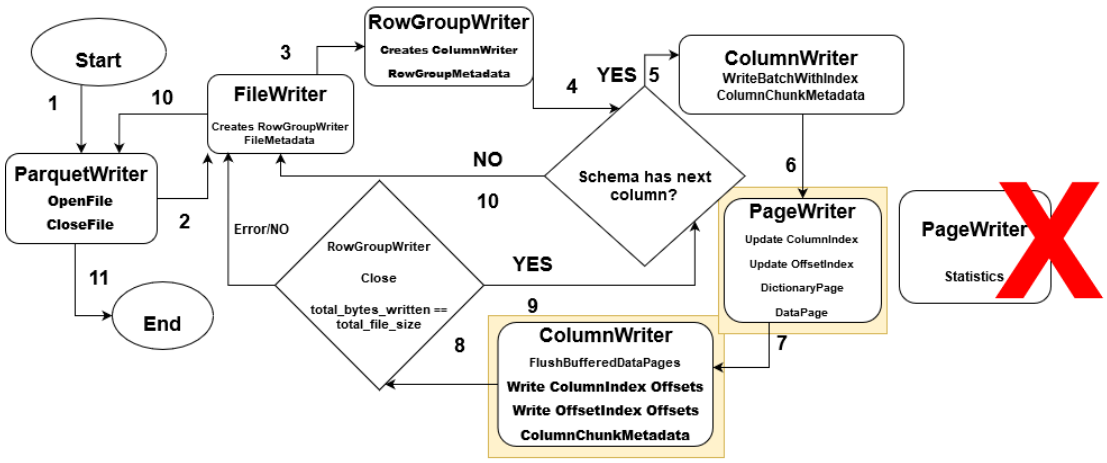


Figure 11: A simplified control flow diagram with yellow highlighted boxes indicating modifications for writing ColumnIndex-OffsetIndex into parquet file in PittCS Arrow.

To achieve this we modify the steps four, five and six of the original control flow for writing a parquet file in Arrow C++. We start by creating an instance of ParquetWriter.

Step 1: The schema for the parquet file is generated.

Step 2: The ParquetWriter creates an instance of FileWriter.

Step 3: The FileWriter creates of RowGroupWriter to write a RowGroup into the file.

Step 4: Each instance of RowGroupWriter checks the schema for the next ColumnChunk to be written into the file.

Step 5 & 6: An instance of ColumnWriter invokes instances to PageWriter and creates an instance of the ColumnIndex. As the PageWriter includes the statistics to all the DataPages in a ColumnChunk, we update the statistics in the ColumnIndex. We write the offsets to the DataPages in the OffsetIndex.

Step 6 & 7: We use the instance of ColumnWriter to write the offsets to the ColumnIndex and OffsetIndex (in the footer) into the ColumnChunkMetadata.

Step 7: The RowGroupWriter verifies the number of bytes written into the file.

Step 8: The RowGroupWriter checks if anymore ColumnChunks are left to be written into the file.

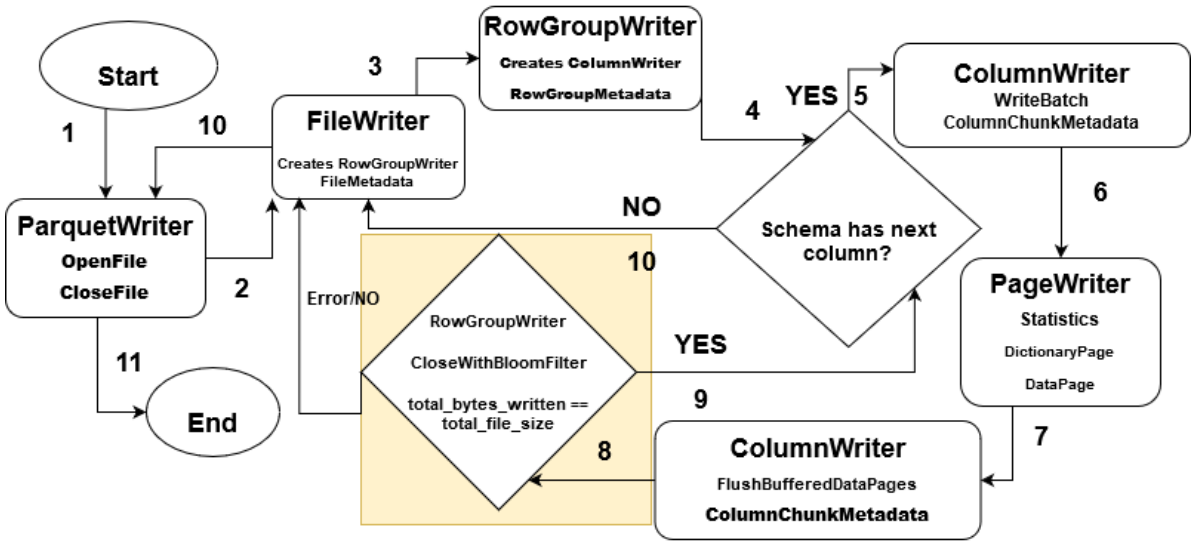


Figure 12: A simplified control flow diagram with yellow box highlighting the modifications to the control flow for writing Bloom filter into a parquet file in PittCS Arrow.

Step 9: On successful completion of writing all ColumnChunks, the RowGroupWriter returns control to FileWriter. The FileWriter repeats the process of creating instances of RowGroupWriter until all the RowGroups are written into the file. It returns the control to ParquetWriter.

Step 10: The FileWriter completes creating the parquet file by writing the footer in the parquet file, along with the ColumnIndex-OffsetIndex.

Step 11: The ParquetWriter closes the file and this marks the end of the control flow.

Writing SBBF

To write SBBF in the parquet files, we initialize the SBBF with its false positive probability. We hash the values, to be written into the file by a PageWriter, into the Bloom filter. Figure 12 represents our modifications to the control flow, to introduce Split Block Bloom Filters in parquet files. Figure 13 depicts the layout of the parquet file after writing the Bloom filters in between RowGroups.

To achieve this we modify the steps four, five and six of the original control flow for writing a parquet file in Arrow C++. We start by creating an instance of ParquetWriter.

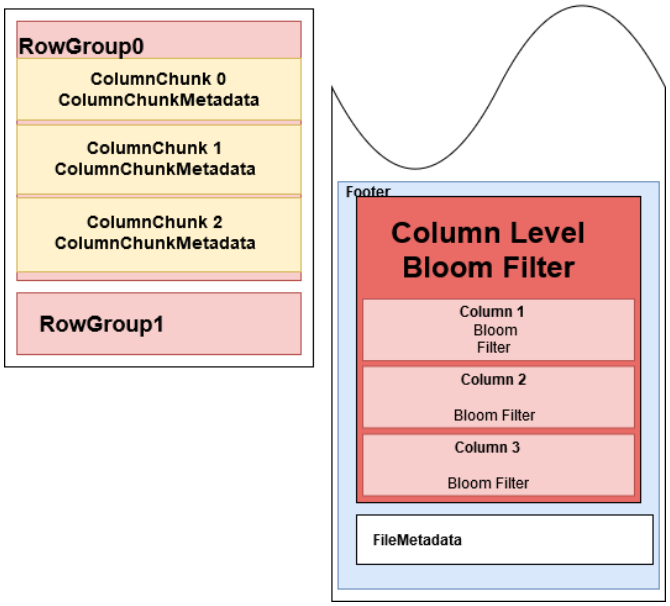


Figure 13: File layout of column-level Bloom filters in PittCS Arrow parquet files

Step 1: The schema for the parquet file is generated.

Step 2: The ParquetWriter creates an instance of FileWriter.

Step 3: The FileWriter creates of RowGroupWriter to write a RowGroup into the file.

Step 4: Each instance of RowGroupWriter checks the schema for the next ColumnChunk to be written into the file. We also create a list of SBBFs to store the SBBF in memory until written into the file by the RowGroupWriter.

Step 5: We intialize a column-level SBBF with the instance of ColumnWriter.

Step 6: An instance of ColumnWriter invokes instances to PageWriter. As the PageWriter writes values into DataPages, we also hash the values into the SBBF. Once the PageWriter completes writing the values into DataPages, it returns the number of bytes written into the file to ColumnWriter. The ColumnWriter adds its SBBF to the list of SBBFs stored in memory.

Step 7: ColumnWriter returns the control to RowGroupWriter with the number of bytes written into the file.

Step 8: The RowGroupWriter verifies the number of bytes written into the file. On

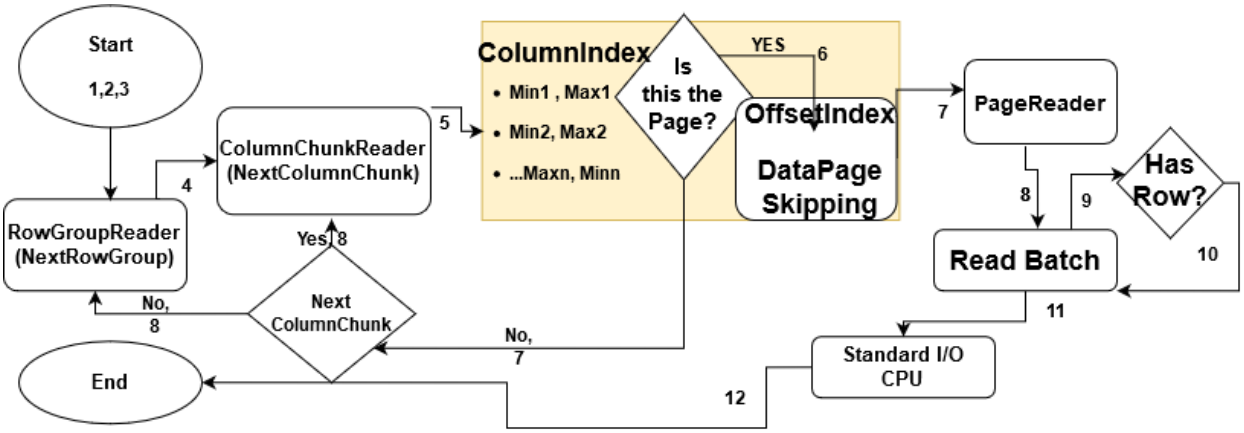


Figure 14: A simplified flow diagram with numbers to indicate the direction of control flow and yellow highlighted boxes to indicate the modifications for reading with ColumnIndex on Parquet file in PittCS Arrow.

successfully verification, the RowGroupWriter writes all the SBBFs in the order as stored in the list of SBBFs in memory. It also notes the offsets to the column-level SBBFs and invokes individual ColumnChunkWriters to update the ColumnChunkMetadata with the bloom filter offset.

Step 9: The RowGroupWriter checks if anymore ColumnChunks are left to be written into the file.

Step 10: On successful completion of writing all ColumnChunks, the RowGroupWriter returns control to FileWriter. The FileWriter repeats the process of creating instances of RowGroupWriter until all the RowGroups are written into the file. It returns the control to ParquetWriter.

Step 11: The ParquetWriter closes the file and this marks the end of the control flow.

3.2.2 PittCS Arrow Parquet Reader

In our implementation, we modify the ColumnChunkReader to retrieve the offsets to ColumnIndex and OffsetIndex in ColumnChunkMetadata. We deserialize the ColumnIndex

and OffsetIndex after loading them from the ColumnIndex offset and OffsetIndex offset in the file.

We start from the RowGroupReader. The RowGroupReader serially invokes instances of ColumnChunkReader.

At Steps 5 and 6, we show the process of using ColumnIndex and OffsetIndex in Figure 14 to select candidate DataPages and to determine the offsets to the candidate DataPages. Arrow offers a functional implementation to skip a certain number of rows before starting to read the parquet file. We implement row skipping to the first row of candidate DataPage(s).

We use ReadBatch to match the rows with the predicate. For each data type supported by parquet, we implement a distinct search paradigm to match the predicate, using ColumnIndex. For integers, we implement plain range search. For float and double data types, we implement range search with 10^{-16} to 10^{-11} standard error. For ByteArrays (strings), we implement range search with the longest prefix match.

In the case of reading Bloom filters with ColumnIndex-OffsetIndex or performing row scans on DataPages, at Steps 5 and 6, we modify the control flow of Arrow C++, as shown in Figures 15 and 16. We implement flags to choose Bloom filter and/or ColumnIndex-OffsetIndex during parquet file reading for row retrieval.

At the Step 7, once the DataPages are selected, the PageReader invokes ReadBatch to read the rows from the DataPages. This marks the end of the sub-control flow.

Reading with ColumnIndex and OffsetIndex

At Steps 5 and 6 of the Arrow control flow in the Figure 14, we load ColumnIndex-OffsetIndex onto the memory using the thrift deserialization protocol for C++ and prepare the ColumnIndex and OffsetIndex for use. Let us consider a case of six columns in a relational table, a RowGroup in the parquet file has six ColumnChunks. Thus, the ColumnIndex has six lists of DataPage minimum and maximum values. The OffsetIndex stores six lists of DataPage first row indices and offsets. All min and max values of DataPages in one list that corresponds to the ColumnChunk, is used to retrieve the DataPage whose range matches for the predicate. We retrieve the cardinal number (the number of the row in a table in row store format) of the first row of the DataPage from the corresponding list in OffsetIndex, for row skipping to the DataPage.

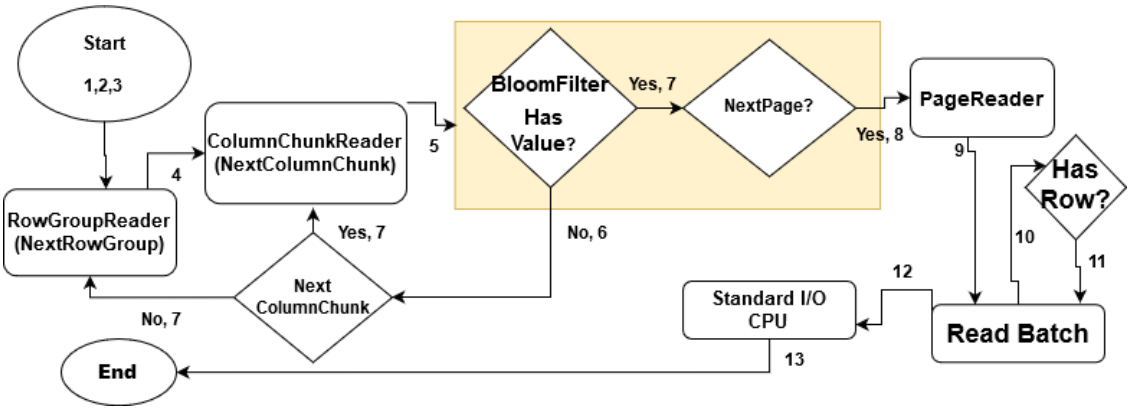


Figure 15: A simplified flow diagram with numbers to indicate the direction of control flow and yellow highlighted boxes to indicate the modifications for reading with Bloom filter in a parquet file in PittCS Arrow.

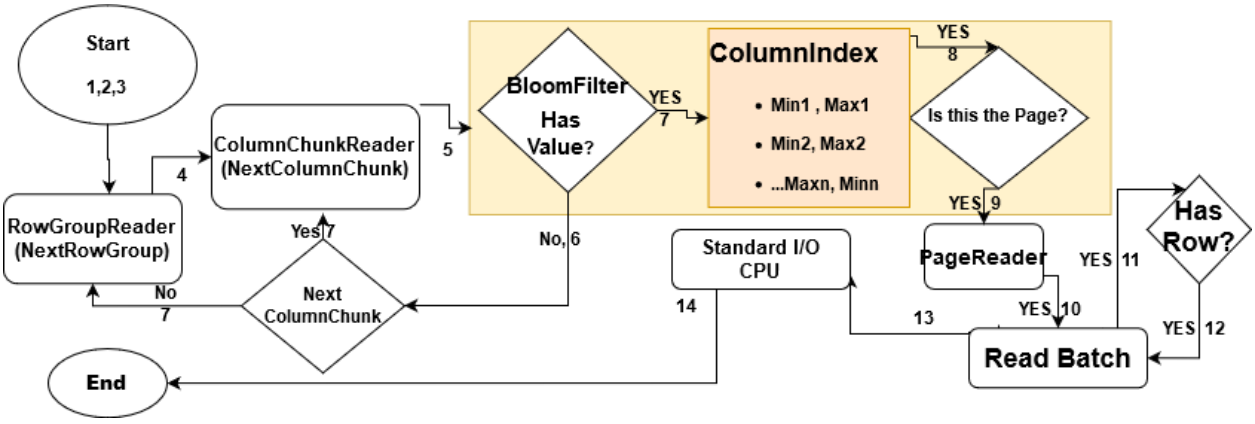


Figure 16: A simplified flow diagram with numbers to indicate the direction of control flow and yellow highlighted boxes to indicate the modifications for reading the parquet file with the combination of Bloom filter, ColumnIndex and OffsetIndex in PittCS Arrow.

Reading with SBBF

At Steps 5 and 6 of the Arrow control flow in the Figure 15 we load the SBBF onto the memory using the thrift deserialization protocol for C++ and prepare the SBBF for use. For each ColumnChunk, we load the offset to the SBBF in the footer. This offset is used in reading the Bloom filter from the file onto the memory. The Bloom filter is used to detect if the value does not exist in the ColumnChunk.

Reading with the combination of SBBF, ColumnIndex and OffsetIndex

At Steps 5 and 6 of the Arrow parquet reading control flow as shown in Figure 16 we load the SBBF and ColumnIndex onto memory using the Apache Parquet thrift protocol for C++ and prepare them for reading. After deserializing the Bloom filter, the value is detected for presence in the ColumnChunk. Only in the case that the value may be present in the ColumnChunk, the ColumnIndex and OffsetIndex are used to extract the exact DataPage to search for the row retrieval.

3.3 Complexity Analysis of Embedded ColumnIndex-OffsetIndex and Bloom filters in PittCS Arrow

In this section, we discuss the cost of the proposed two embedding structures, namely ColumnIndex-OffsetIndex and SBBF, from theoretical point of view in order to better understand our experimental evaluation presented in the next chapter.

3.3.1 Complexity Analysis of Adding ColumnIndex-OffsetIndex in parquet file

In theory, writing ColumnIndex-OffsetIndex in a parquet file is a constant time operation. Since the minimum and maximum values are already being computed to be added to the Statistics in a DataPage, the same computation will be performed for updating those entries in ColumnIndex. Computing the offset to a DataPage is also a constant time operation because the offset is simply the location at which the OffsetIndex will be written, which is known at the time of writing while immediately updating the OffsetIndex.

Our focus, however, is to understand the computational complexity of scanning a value from the parquet file with ColumnIndex-OffsetIndex. The time to read the parquet file with ColumnIndex-OffsetIndex (t_r) in parquet file is the sum of the time to load ColumnIndex-OffsetIndex (t_l), time to retrieve the candidate DataPage (t_p) and time to scan the value in the candidate DataPage (t_s), that is,

$$t_r = t_l + t_p + t_s \tag{3.1}$$

Given that time to load the ColumnIndex-OffsetIndex happens once when the file is open for reading, we can consider an amortized constant time for the purposes of theoretical assumption. The time taken to detect the candidate page can be discussed in further detail. We use the minimum and maximum values in ColumnIndex to detect the candidate DataPage. In the best case scenario where the data is sorted in a ColumnChunk, the minimum and maximum values are also sorted for the DataPages. In such a case, the candidate DataPage can be detected in an average of $O(n/2)$ time with short-circuiting, where n is the number of minimum and maximum entries in the ColumnIndex for the ColumnChunk. In the worst case scenario where the data is not sorted or the predicate value can potentially does not exist or only exists in the last DataPage of sorted data, the candidate DataPage can only be detected in an average of $O(n)$ time. Using binary search for detecting candidate DataPage of sorted data should take $O(logn)$ time. Once the potential DataPage is detected, retrieving the offset should be a $O(1)$ operation, since OffsetIndex is essentially a map to all the offsets of the DataPages. Further, to scan and retrieve the value in a DataPage for the result, can also involve short-circuiting in the case of sorted data to perform the operation in $O(k/2)$ time or linear scan in the case of unsorted data to perform the operation in $O(k)$ time, where k is the number of rows in the DataPage. Using binary search to scan the values in a sorted DataPage will take $O(logk)$ time. This process has to be repeated for all the candidate DataPages that could potentially hold the value to be returned as result from the file.

In essence, scanning the value from the parquet file with ColumnIndex-OffsetIndex should take $O(m(logn + logk))$ to $O(m(n + k))$, where m is the number of candidate DataPages, that is,

$$m(logn + logk) <= t_r <= m(n + k) \tag{3.2}$$

3.3.2 Complexity Analysis of Adding Bloom filters

In theory, writing Bloom filters into a parquet file is a constant operation to write the SBBF into a parquet file and $r * 8 * O(1)$ time to hash all values into the Bloom filter, where r is the number of rows to be hashed into the bloom filters.

Our focus is to understand the computational complexity of reading a parquet file with Bloom filters. Since the Bloom filter has to be loaded on to the memory, the time to read a parquet file along with Bloom filter (t_r) is the sum of the time to load the Bloom filter from the file (t_{lbf}). Since we load the Bloom filter only once, we do not consider that in the overall file reading time. Hence, the time to detect if the value is present or not in the file (t_p), that is,

$$t_r = t_p \tag{3.3}$$

The Bloom filter can be loaded from the parquet file in $O(1)$ time the from the footer. Detecting if a value is present in the DataPage or ColumnChunk is also a $O(8)$ operation in a Bloom filter. Hence, the overall time in reading a parquet file in Bloom filter should also be a constant c time operation, that is,

$$t_r = O(c) \tag{3.4}$$

Although reading the parquet file with a Bloom filter is a constant time operation, this constant varies with the DataType of the values hashed into the Bloom filter and the size of the Bloom filters, which is determined by the false positive probability (fpp). The larger the size of the Bloom filter, the more the time taken to load the Bloom filter from the parquet file and the larger is the value of the constant. But, once loaded for a column, the Bloom filter should contribute the same amount of time to the overall parquet file reading time, for each value verification.

In essence, scanning the value from the parquet file with ColumnIndex-OffsetIndex should take $O(m(logn + logk))$ to $O(m(n + k))$, where m is the number of candidate DataPages, that is,

$$m(logn + logk) <= t_r <= m(n + k) \tag{3.5}$$

3.4 Chapter Summary

In this chapter, we discussed the parquet reader and writer in Arrow and our implementation, PittCS Arrow, in detail. The actual code for PittCS Arrow reader and writer are available in github[2].

To implement PittCS Arrow writer for ColumnIndex-OffsetIndex, we modify the Arrow writing control flow in the function call to WriteBatch in ColumnWriter to instantiate the ColumnIndex and OffsetIndex entries for that ColumnChunk, modify the PageWriter to add the minimum and maximum values of DataPages into the ColumnIndex. Further, we modify the PageWriter to store the offsets to the DataPages, to be added as entries in the OffsetIndex. The ColumnWriter sets the offsets to ColumnIndex and OffsetIndex in the ColumnChunkMetadata. To implement PittCS Arrow writer for Split Block Bloom Filters, we modify the ColumnWriter, to instantiate a Bloom filter per ColumnChunk. For each ColumnChunk, the PageWriter hashes the values being written into the DataPage also into the Bloom filter. Once the Bloom filter has all the values in the ColumnChunk hashed, the RowGroupWriter writes the bloom filter instance into the file at the footer along with the offsets to the Bloom filters in the ColumnChunkMetadata.

To implement PittCS Arrow reader for ColumnIndex-OffsetIndex, we modify the Arrow reading control flow. We deserialize the ColumnIndex-OffsetIndex in ColumnChunkReader using the offset stored in the ColumnChunkMetadata. Similarly, we deserialize the Bloom filter in the ColumnChunkReader using the offset stored in ColumnChunkMetadata. Once the Bloom filter or ColumnIndex-OffsetIndex is deserialized on to the memory, they are used to either detect if the value is present in the column (Bloom filter) or where the DataPage

[2]https://github.com/a2un/arrow/

in which the value may be present in the column (ColumnIndex-OffsetIndex). Once the location of the value and its presence in the column is determined, we modify the PageReader to perform a localized scan to detect and retrieve the value from the file.

In the next chapter, we discuss the instrumentation and performance evaluation of PittCS Arrow. Further, we present and analyze the results in detail, which support our two aims in this thesis in optimizing parquet files in Arrow C++.

4.0 Experimental Evaluation

In this chapter, we proceed further with describing the instrumentation and evaluation for measuring performance improvements in PittCS Arrow parquet file reading times. We describe the methodology for our experiments and present the experimental platform in Section 4.1. We present our observations of the file reading times for searching a predicate in the parquet file and the change in size of the file with the introduction of Bloom filters, ColumnIndex and OffsetIndex in Section 4.2. Finally, we analyze and discuss our results in detail in Section 4.3.

4.1 Experimental Methodology

4.1.1 Evaluation Metrics & Statistics

Average Response Time

We measure the average file reading time using the high-precision time *chrono* library in C++ to measure the difference in response times in milliseconds.

We measure average response times in two forms, namely

(i) **Amortized Average time**: This is the average response time per query when including the time taken to load the ColumnIndex-OffsetIndex and/or Bloom filter for the query processing.

(ii) **Average time**. This is the average response time per query without including the time taken to load the ColumnIndex-OffsetIndex and/or Bloom filter at the first invocation.

File Size

We measure the change in file size on adding ColumnIndex-OffsetIndex and the change in the file size on adding Bloom filters. We also measure the change in size of the metadata after adding Bloom filters and ColumnIndex-OffsetIndex at the footer of the file.

Experimental Profiling

For the experiments with Bloom filters, we profile the false positives on member and non–member queries, for each column. We, only measure the truth of the Bloom filters and not the time to verify if the value is present in the set.

4.1.2 Dataset

We use a table schema for five columns: $int32$ (32-bit integer), $int64$ (64-bit integer), $float$, $double$ and $ByteArray$ (string of 124 characters in length). We use the generated schema to create the metadata information for the parquet file. By considering all the supported data types as columns of the table schema, our dataset covers all the different types used by the database tables that can be stored in the parquet file.

We generate two parquet files, with sorted and unsorted data, with 10 Million rows in 1 RowGroup of size 1 *GB*. To generate a file with sorted data, we create values in sequence of row numbers, for 10 Million rows. This way the data is always guaranteed to be sorted. To generate a file with unsorted data, we create a random value for every row, for 10 Million rows. We generate parquet files with plain encoding and no compression. Since the two parquet files use a schema of 5 columns, each RowGroup has 5 ColumnChunks. The size of each DataPage in each ColumnChunk is upto 1 *MB*. The files are written in plain encoding for each datatype and are uncompressed. All the parameters in our experiment are listed in Table 3. The generated parquet files are embedded with Bloom filters and ColumnIndex-OffsetIndex.

4.1.3 Experimental Setup

Response Time Comparison

We implement a testbed where the queries are executed using the parquet access methods utilizing the ColumnIndex, OffsetIndex and Bloom filters. In each run, we read the parquet file based on one particular value of a column. To measure the response times for our approach, we define two types of file accesses:

Table 3: Control and Dataset Parameters

Parameter	**Value**
Number of parquet files	7
Size of parquet file	128MB, 1 GB, 2.2 GB
Data Arrangement	sorted and unsorted
Number of rows per file	10M rows
Number of RowGroups per Parquet file	1
Number of columns	5
Column data types	int32, int64, float, double, strings
With or without file caching	With file caching
String character length	124 bytes
Number of ColumnChunks per RowGroup	5
Number of DataPages per ColumnChunk	>= 5
Size of DataPage	<= 1 MB
Size of RowGroup	128MB, 1 GB, 2.2 GB
Compression	Uncompressed
Encoding	Plain
Bloom filter false positive probability	0.001, 0.01, 0.1, 0.2, 0.5
Number of member queries per run	1000
Number of runs	5 & 10 runs
Member query predicate range	value between zero & total number of rows
Non–Member query predicate range	value not present in the file
Compression Codec	GZip

i) **Member Queries**: The file operations performed for retrieving a row that is present in the file.

ii) **Non-Member Queries**: The file operations performed for retrieving a row that is not present in the file.

In both cases, we are reading a value in one particular column. As discussed in Chapter 1, Section 1, the parquet format in Arrow retains the row representation of a table during data retrieval from columnar storage. That is, the experiments reflect the following SQL select statement with a single attribute (column) predicate:

*select * from file x where a = <search-value>;*

We repeat this test for all columns in the file. In general, the parquet files' metadata could potentially be used by query engines to support more complex selection conditions such as range queries.

We run the parquet file queries with and without file caching (*O_DIRECT flag*[1]). However, we did not notice any significant difference in the file reading time measurements with or without caching. Hence, we include only the file reading times with caching, in our experiments. For workload simulation, we run the member and non–member queries on both, sorted and unsorted, parquet files. We execute 1000 member and 1000 non–member file accesses, separately, in 5 runs. We generate predicates randomly for each run, to be read from the file.

Bloom filter sensitivity

We perform sensitivity analysis on the Bloom filters for the false positive probability values of 0.001, 0.01, 0.1, 0.2 and 0.5 on parquet files with Bloom filters for 10M rows, with not particular data arrangement (with or without sorting) in five columns for ten runs with thousand non–member queries per run.

Indirect Comparison of performance with Impala

We run an instance of PittCS Arrow parquet reader on a 128 MB parquet file to match the implementation of Impala parquet reader, since Impala's implementation limits the maximum size of parquet files to the maximum size of a Hadoop DataNode (data block in the

[1]https://man7.org/linux/man-pages/man2/open.2.html

Table 4: Computational infrastructure

Infrastructure	Value
CPU	12 cores, 3.2 Ghz
Memory	32 GB
OS	Ubuntu 18.04
Linux kernel	5.3
openAFS	version 1.8.5
CMake	version 3
Arrow	version 0.14
GCC	version 7
Thrift	version 0.9
C++	version 17
Chrono	version 17 (C++ STL)

Hadoop File System), 128 MB. We measure the average file reading time using the high-precision time library in C++, *chrono* in milliseconds.

Bloom filter compression

We run an instance of the PittCS Arrow parquet writer to create parquet files with uncompressed bloom filters at false positive probability value of 0.001 for 10M rows. We apply GZip compression codec provided by Arrow, to measure the factor of compression in parquet files and bring the benefits of compression in PittCS Arrow parquet files.

4.1.4 Computational Infrastructure

All performance evaluations are executed on a system with twelve core 3.2 GHz processor and 32 GB memory. The system runs Ubuntu 18.04 on linux kernel version 5.3 and openAFS file system version 1.8.5. The software to run the performance evaluations is built with Arrow 0.14 on CMake version 3, with C++ version 17, GCC version 7 and Thrift version 0.9. We use the *chrono* header files provided by the C++ Standard Template Library (STL), to measure the file reading times. We summarize the computational infrastructure in Table 4.

4.2 PittCS Arrow Experimental Results

In this section, we measure the performance of our implementation, PittCS Arrow, using the experimental setup discussed above.

4.2.1 Binary search with ColumnIndex-OffsetIndex on sorted data

Inspired by our complexity analysis of ColumnIndex-OffsetIndex in this chapter, Section 3.3.1, we included binary search in selecting the candidate DataPages while selecting the matching range of minimum and maximum values of DataPages with the predicate. However, in parquet files that are small with small number of DataPages, we must note that adding binary search with the ColumnIndex-OffsetIndex on sorted data in selecting the DataPages should not add significant benefits to the file reading time. reading small parquet files with small number of DataPages does not add significant benefits, that is we achieve the same 20x improvement in file reading times of using plain ColumnIndex-OffsetIndex without binary search. We present our results in the Figure 17.

4.2.2 Comparisons of Average Response Time

In the first query in every run on every column, we include the time to load the ColumnIndex-OffsetIndex or Bloom filter once onto memory along with the time taken to read the file. In the subsequent queries, we measure *only* the file reading time. We measure the average parquet file reading time of 50000 file accesses separately on *sorted* and *unsorted* parquet files.

We use notations to describe the comparisons of average response times, as presented in Table 5. We compare the average file reading time in four cases namely, *without ColumnIndex & OffsetIndex or Bloom filters (WOIBF)*, *with Bloom filters (WBF)*, *Bloom filters with ColumnIndex-OffsetIndex (WIBF)*, *Bloom filters with ColumnIndex & OffsetIndex (WIBF)*, as shown in Figures the 18, 19, 20 and 21 for sorted and unsorted data in parquet files.

WOIBF vs WI (Figures 18, 19, 20, 21 and Table 6)

We see an *average* (the average of the values under member and non–member queries,

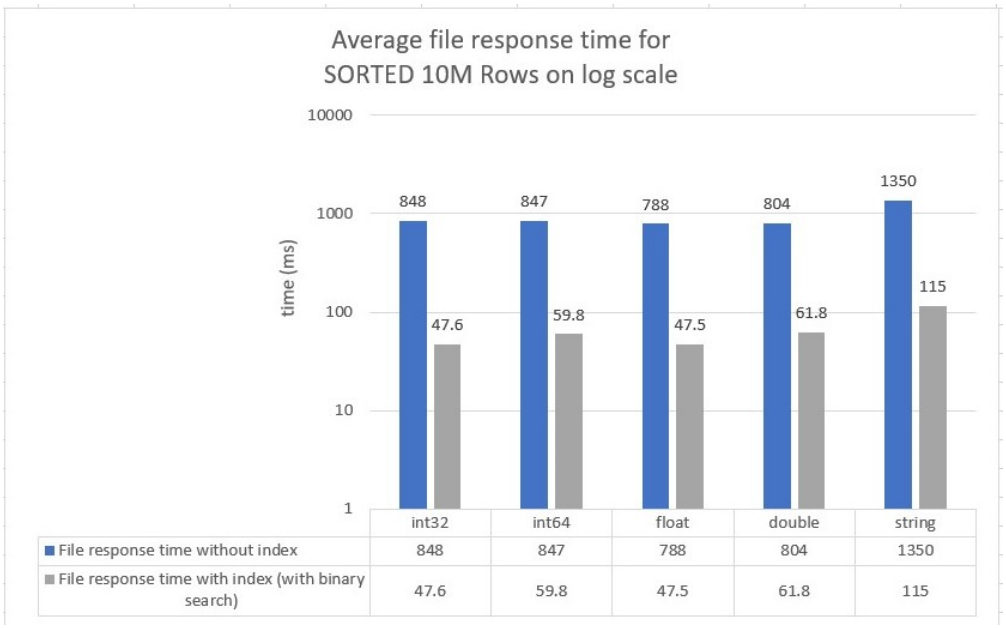


Figure 17: Average file reading times for sorted parquet files with 10M rows for member queries shows a 20x reduction using ColumnIndex-OffsetIndex and binary search.

respectively) decrease in file reading time by 9.27 when using ColumnIndex-OffsetIndex for *member* queries and by 25.14 when using ColumnIndex-OffsetIndex for *non–member* queries, as shown in Table 6. We observe approximately 20x improvement on log scale of the average parquet file reading time in milliseconds, in all cases, in the Figures 18 and 20.

WOIBF vs WBF (Figures 18, 19, 20, 21 and Table 7)

In the profile of the experiment, we see 0% false positives in the case of the member queries on integer and string columns. 10%-12% of the member queries to only double and float columns were falsely categorized as not present. 98%-99% of non–member queries to all columns were correctly categorized as not present.

As expected, we observe a significant decrease in the *average* file reading time by 23.6 in the case of non–member queries, as shown in Table 7. We observe approximately 20x improvement of the average parquet file reading time in milliseconds, in all cases, in the Figures 18 and 20.

Table 5: Notation used for average file reading time comparisons

File access method	Abbreviation
without ColumnIndex-OffsetIndex or Bloom filters	WOIBF
with ColumnIndex & OffsetIndex	WI
with Bloom filters	WBF
Bloom filters with ColumnIndex-OffsetIndex	WIBF

WOIBF vs WIBF (Figures 18, 19, 20, 21 and Table 8)

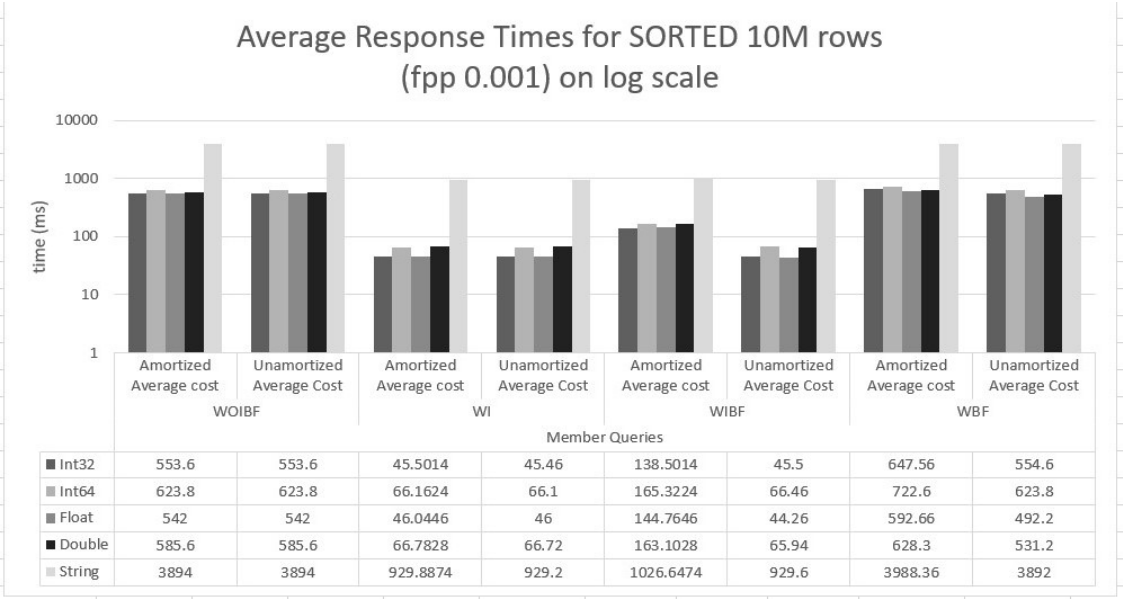
In the profile of the experiment, we see 0% false positives in the case of the member queries on integer and string columns. 10%-12% of the member queries on double and float columns were falsely categorized as not present. 98%-99% of non–member queries for all columns, were correctly categorized as non–members. We see that the *average* file reading time reduces by 9.27 for *member* queries and by 28.63 for *non–member* queries, as shown in Table 8. The real gain in file reading time is in the case of non–member queries, which the Bloom filters are designed to handle. We observe approximately 20x improvement in the average parquet file reading time in milliseconds, in all cases, in the Figures 18 and 20.

WI vs WIBF (Figures 18, 19, 20, 21 and Table 9)

We see that the *average* (the average of the values under member and non–member queries, respectively) file reading time is nearly the same when using or without using the Bloom filter. In case of non–member queries there is a 10% reduction in the file reading time, as shown in Table 9. We observe approximately 10% improvement of the average response time, in all cases, in the Figures 18 and 20.

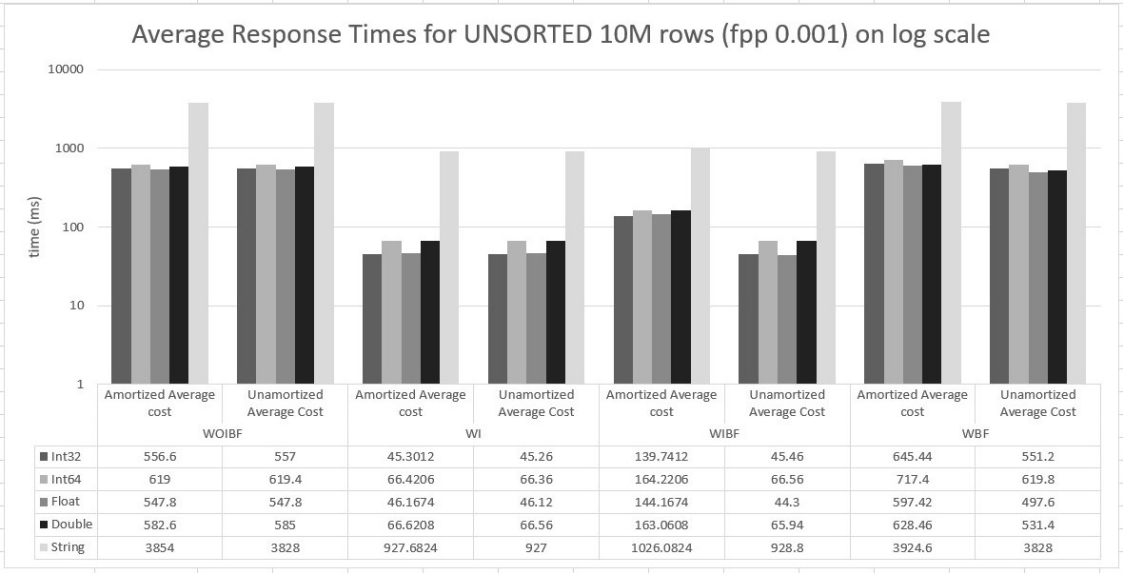
In the profile of the experiment, we see 0% false positives in the case of the member queries on integer and string columns. 10%-12% of the member queries to only double and float columns were falsely categorized as not present. 98%-99% of non–member queries to all columns were correctly categorized as not present.

ColumnIndex-OffsetIndex load time vs Bloom filter load time (Figures 22 and 23)



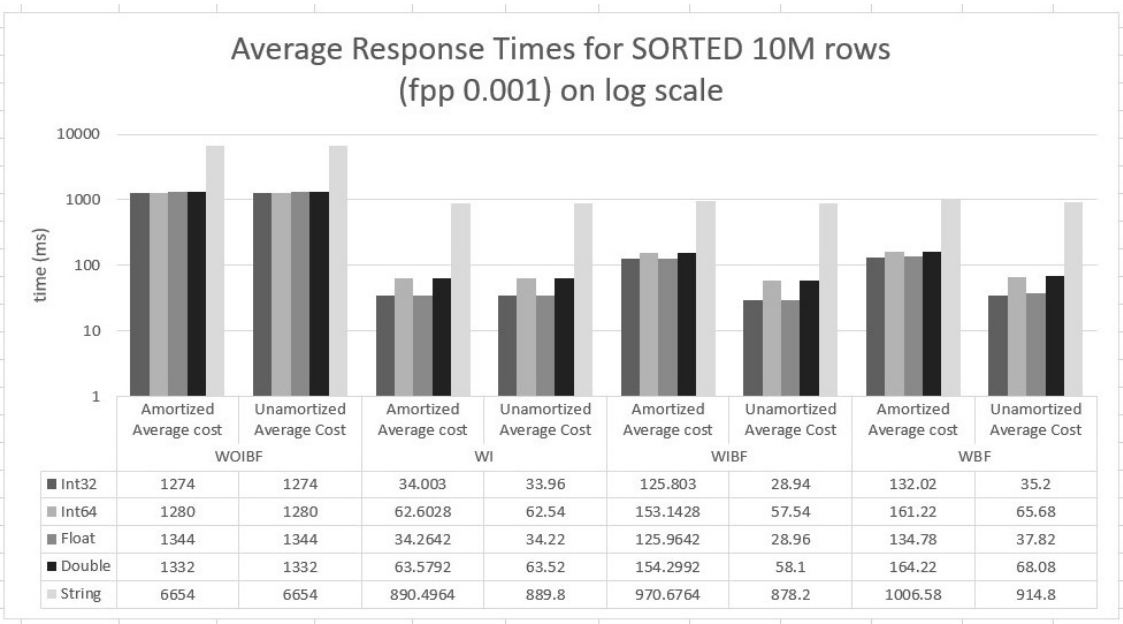
	WOIBF Amortized Average cost	WOIBF Unamortized Average Cost	WI Amortized Average cost	WI Unamortized Average Cost	WIBF Amortized Average cost	WIBF Unamortized Average Cost	WBF Amortized Average cost	WBF Unamortized Average Cost
Int32	553.6	553.6	45.5014	45.46	138.5014	45.5	647.56	554.6
Int64	623.8	623.8	66.1624	66.1	165.3224	66.46	722.6	623.8
Float	542	542	46.0446	46	144.7646	44.26	592.66	492.2
Double	585.6	585.6	66.7828	66.72	163.1028	65.94	628.3	531.2
String	3894	3894	929.8874	929.2	1026.6474	929.6	3988.36	3892

Figure 18: Average file reading times for sorted parquet files with 10M rows for member queries has 20x reduction using ColumnIndex-OffsetIndex and ColumnIndex-OffsetIndex with Bloom Filters.



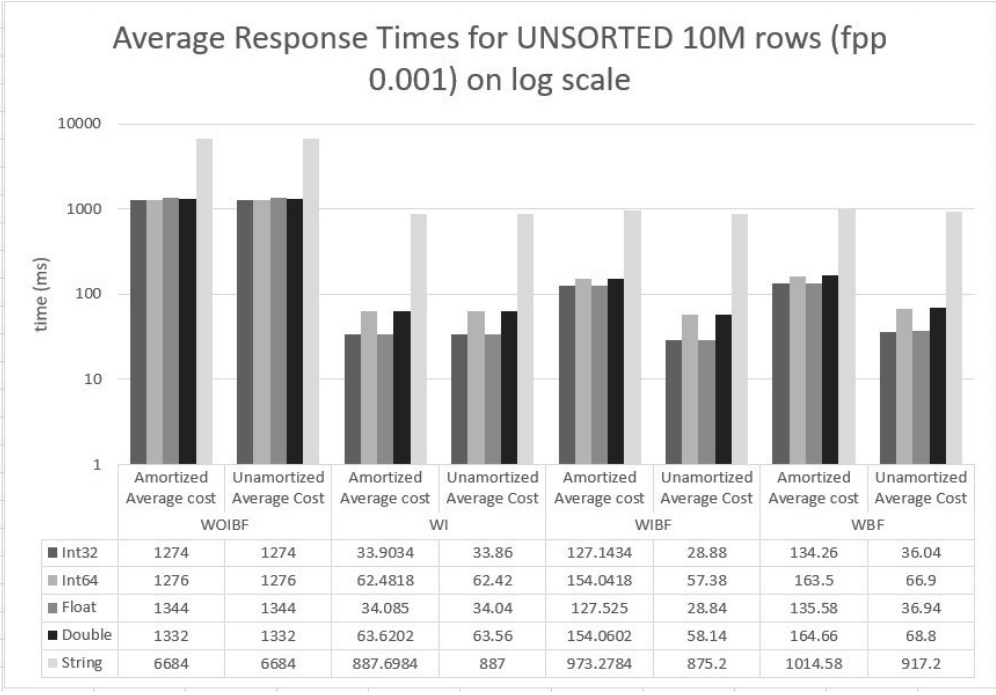
	WOIBF Amortized Average cost	WOIBF Unamortized Average Cost	WI Amortized Average cost	WI Unamortized Average Cost	WIBF Amortized Average cost	WIBF Unamortized Average Cost	WBF Amortized Average cost	WBF Unamortized Average Cost
Int32	556.6	557	45.3012	45.26	139.7412	45.46	645.44	551.2
Int64	619	619.4	66.4206	66.36	164.2206	66.56	717.4	619.8
Float	547.8	547.8	46.1674	46.12	144.1674	44.3	597.42	497.6
Double	582.6	585	66.6208	66.56	163.0608	65.94	628.46	531.4
String	3854	3828	927.6824	927	1026.0824	928.8	3924.6	3828

Figure 19: Average file reading times for unsorted parquet files with 10M rows for member queries has 20x reduction on using either Bloom filter or ColumnIndex-OffsetIndex.



	WOIBF Amortized Average cost	WOIBF Unamortized Average Cost	WI Amortized Average cost	WI Unamortized Average Cost	WIBF Amortized Average cost	WIBF Unamortized Average Cost	WBF Amortized Average cost	WBF Unamortized Average Cost
Int32	1274	1274	34.003	33.96	125.803	28.94	132.02	35.2
Int64	1280	1280	62.6028	62.54	153.1428	57.54	161.22	65.68
Float	1344	1344	34.2642	34.22	125.9642	28.96	134.78	37.82
Double	1332	1332	63.5792	63.52	154.2992	58.1	164.22	68.08
String	6654	6654	890.4964	889.8	970.6764	878.2	1006.58	914.8

Figure 20: Average file reading times for sorted parquet files with 10M rows for non–member queries has 20x reduction using ColumnIndex-OffsetIndex or Bloom filters.



	WOIBF Amortized Average cost	WOIBF Unamortized Average Cost	WI Amortized Average cost	WI Unamortized Average Cost	WIBF Amortized Average cost	WIBF Unamortized Average Cost	WBF Amortized Average cost	WBF Unamortized Average Cost
Int32	1274	1274	33.9034	33.86	127.1434	28.88	134.26	36.04
Int64	1276	1276	62.4818	62.42	154.0418	57.38	163.5	66.9
Float	1344	1344	34.085	34.04	127.525	28.84	135.58	36.94
Double	1332	1332	63.6202	63.56	154.0602	58.14	164.66	68.8
String	6684	6684	887.6984	887	973.2784	875.2	1014.58	917.2

Figure 21: Average file reading times for unsorted parquet files with 10M rows for non–member queries has 20x reduction utilizing either Bloom filter or ColumnIndex-OffsetIndex.

Table 6: Amortized average reading times (in milliseconds) WOIBF vs WI with sorted data

Query Type	**Data Type**	**WOIBF**	**WI**	**Ratio WOIBF by WI**	**Percentage Reduction**
member	int32	553.6	45.46	12.17	91%
	int64	623.8	66.1	9.43	85%
	float	542	46	11.78	91%
	double	585.6	66.72	8.77	88%
	string	3894	929.2	4.19	76%
non–member	int32	1274	33.96	37.51	97%
	int64	1280	62.54	20.46	95%
	float	1344	34.22	39.27	97%
	double	1332	63.52	20.96	94%
	string	6654	889.8	7.47	86%

Table 7: Amortized average reading times (in milliseconds) WOIBF vs WBF with sorted data

Query Type	**Data Type**	**WOIBF**	**WBF**	**Ratio WOIBF by WBF**	**Percentage Reduction**
member	int32	553.6	554.6	0.99	0%
	int64	623.8	623.8	1	0%
	float	542	492.2	1.10	0%
	double	585.6	531.2	1.10	0%
	string	3894	3892	1.00	97%
non–member	int32	1274	35.2	36.19	97%
	int64	1280	65.68	19.4	94%
	float	1344	37.82	35.5	97%
	double	1332	68.08	19.5	94.8 %
	string	6654	914.8	7.27	86%

Table 8: Amortized average reading times (in milliseconds) of WOIBF vs WIBF with sorted data

Query Type	**Data Type**	**WOIBF**	**WIBF**	**Ratio WOIBF by WIBF**	**Percentage Reduction**
member	int32	553.6	45.5	12.17	91%
	int64	623.8	66.46	9.43	85%
	float	542	44.26	11.78	91%
	double	585.6	65.94	8.77	88%
	string	3894	929.6	4.19	76%
non–member	int32	1274	28.94	44.02	97%
	int64	1280	57.54	22.24	95%
	float	1344	28.96	46.40	97.8%
	double	1332	58.1	22.92	95%
	string	6654	878.2	7.57	86.8%

Table 9: Amortized average reading times (in milliseconds) of WI vs WIBF with sorted data

Query Type	**Data Type**	**WI**	**WIBF**	**Ratio WI by WIBF**
member	int32	45.46	45.5	0.99
	int64	66.1	66.46	0.99
	float	46	44.26	1.03
	double	66.72	65.94	1.01
	string	929.72	929.6	0.99
non–member	int32	33.96	28.94	1.17
	int64	62.54	57.54	1.08
	float	34.22	28.96	1.18
	double	63.52	58.1	1.09
	string	889.8	878.2	1.01

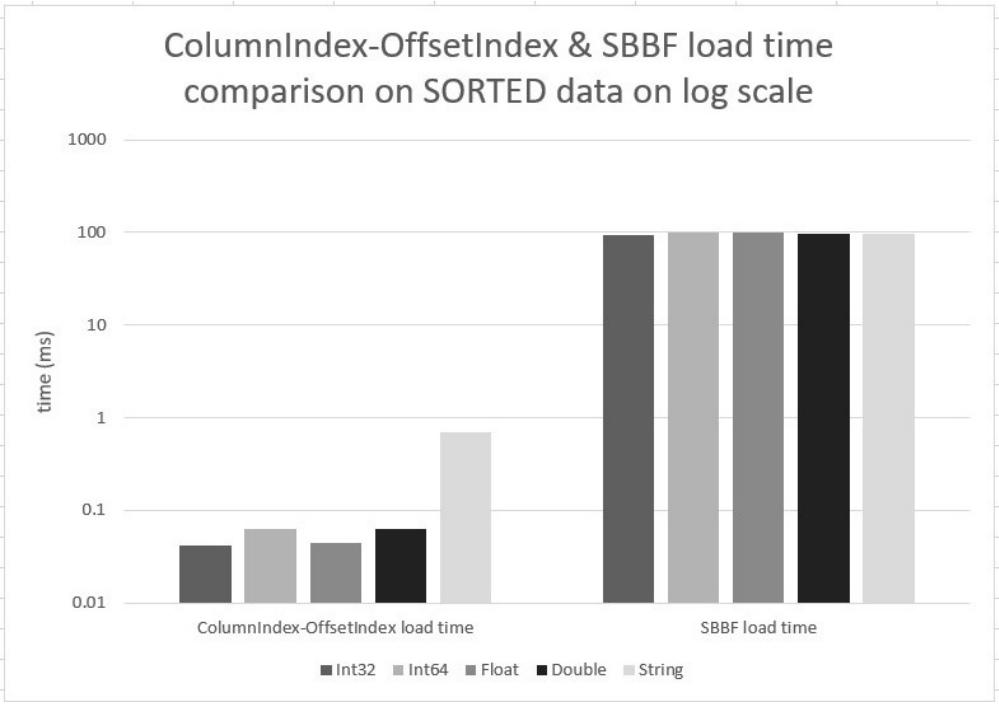


Figure 22: Average ColumnIndex-OffsetIndex and SBBF loading times for sorted parquet files with 10M rows.

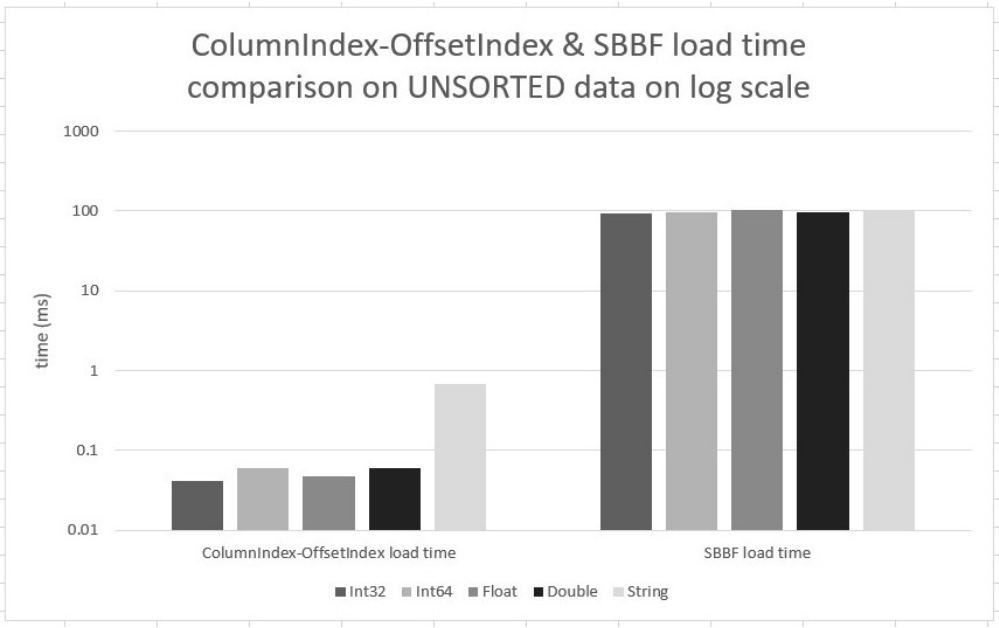


Figure 23: Average ColumnIndex-OffsetIndex and SBBF loading times for unsorted parquet files with 10M rows.

Table 10: Amortized average reading times (in milliseconds) on sorted vs unsorted data in parquet files

Query Type	**Metric**	**Read time improvement in sorted data**	**Read time improvement in unsorted data**
member	WBF	1.04	1.04
	WI	9.27	9.27
	WIBF	9.27	9.27
non–member	WBF	23.61	23.81
	WI	25.14	25.53
	WIBF	28.63	28.7

We notice that the time taken to load ColumnIndex-OffsetIndex onto memory is less than the time taken to load the Bloom filter. In general, the time taken to load Bloom filters or ColumnIndex-OffsetIndex onto memory is less than the time taken to search a value from the parquet file, as expected. We summarize our findings in the Figures 22 and 23.

4.2.3 Comparisons of File Size

We use notations to describe our case, as presented in Table 11. We compare the change in file sizes, namely in *without ColumnIndex-OffsetIndex or Bloom filters (WOIBF)* and with *Bloom filters (WBF)*.

WOIBF vs WI (Figures 24 and 25)

The parquet file without Bloom filters and ColumnIndex-OffsetIndex is of size 1 *GB*. The parquet file with ColumnIndex-OffsetIndex is of size 1.0003 *GB*. We observe that the increase in number of bytes per data type for the file is close to 0.03%. We summarize the change in file size with ColumnIndex-OffsetIndex in Figures 24 and 25.

WOIBF vs WBF (Figure 25)

The size of a parquet file is doubled when embedded with Bloom filter for a false positive

Table 11: Notation used for comparisons of change in file size

Condition	**Abbreviation**
without ColumnIndex-OffsetIndex or Bloom filters	WOIBF
with ColumnIndex & OffsetIndex	WI
with Bloom filters	WBF

probability of 0.001. This value also varies with the false positive probability. In our case, the false positive probability was fixed at 0.001 for all the experiments. We measure the increase in file size to be up to twice the original size of the file, as shown in Figure 25. As verified from the change in file size experiment, the change in size of the file with a Bloom filter with false positive probability of 0.001 for 10M rows shows an increase by 500 MB for five columns. Proportionately this reduces to 200 *MB* increase in file size when the false positive probability is 0.5 that is, 40 *MB* sized Bloom filter per column in the table.

In comparisons with the increase in footer size, we notice that the increase in file size is mainly contributed proportionately by the increase in footer size. We summarize our results in Figure 26. We see the increase of footer up to ten times the size of the original footer size on adding ColumnIndex-OffsetIndex on log scale and an increase up to 10 times the increase of footer size up to ten times the size of the footer with Bloom filters.

4.2.4 Indirect Comparison with Impala

We compare the file reading times in PittCS Arrow parquet reader with 128 MB parquet files, *without ColumnIndex & OffsetIndex or Bloom filters (WOIBF)*, *with Bloom filters (WBF)*, *Bloom filters with ColumnIndex-OffsetIndex (WIBF)*. We summarize our results in Figures 27 and 28 to observe the 20x improvement in PittCS Arrow parquet reading times.

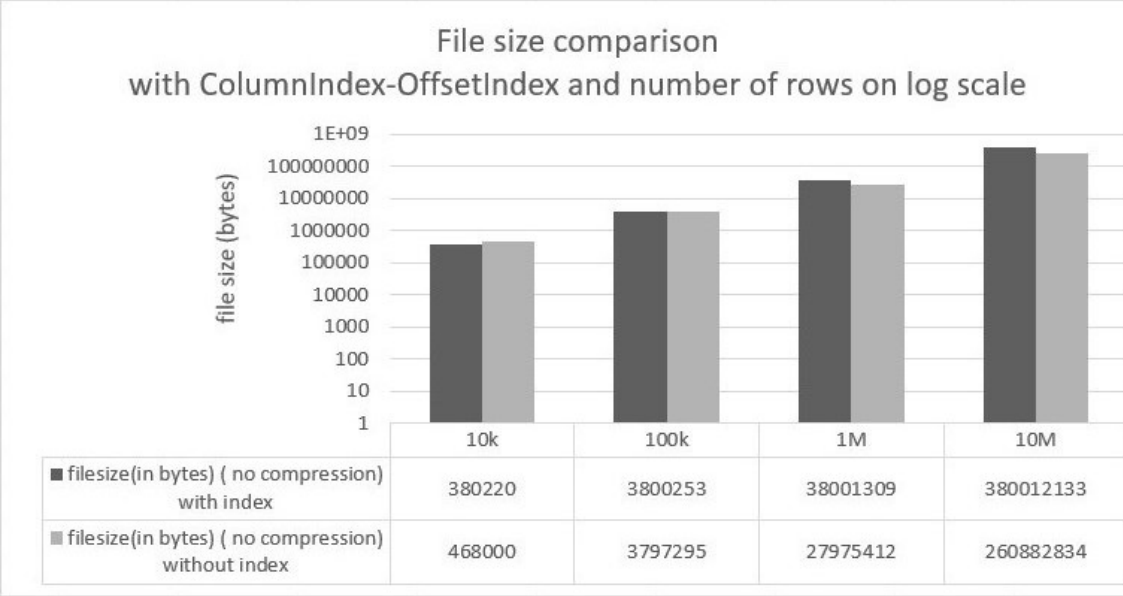


Figure 24: Measurements in log scale to compare change in file size with ColumnIndex-OffsetIndex

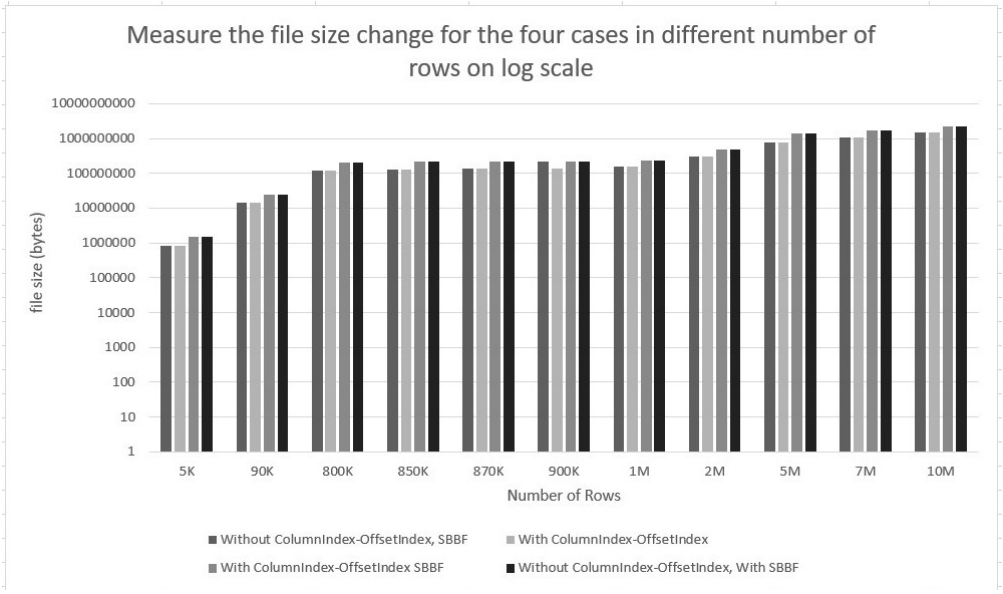


Figure 25: Measurements in log scale to compare change in file size with ColumnIndex-OffsetIndex and Bloom filters

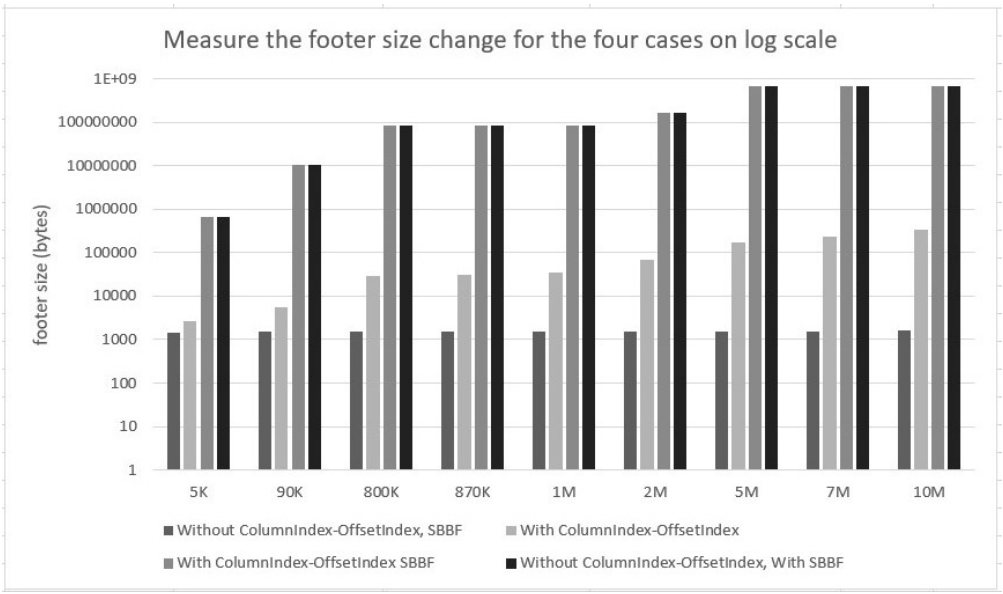


Figure 26: Measurements in log scale to compare change in footer size with ColumnIndex-OffsetIndex and Bloom filters

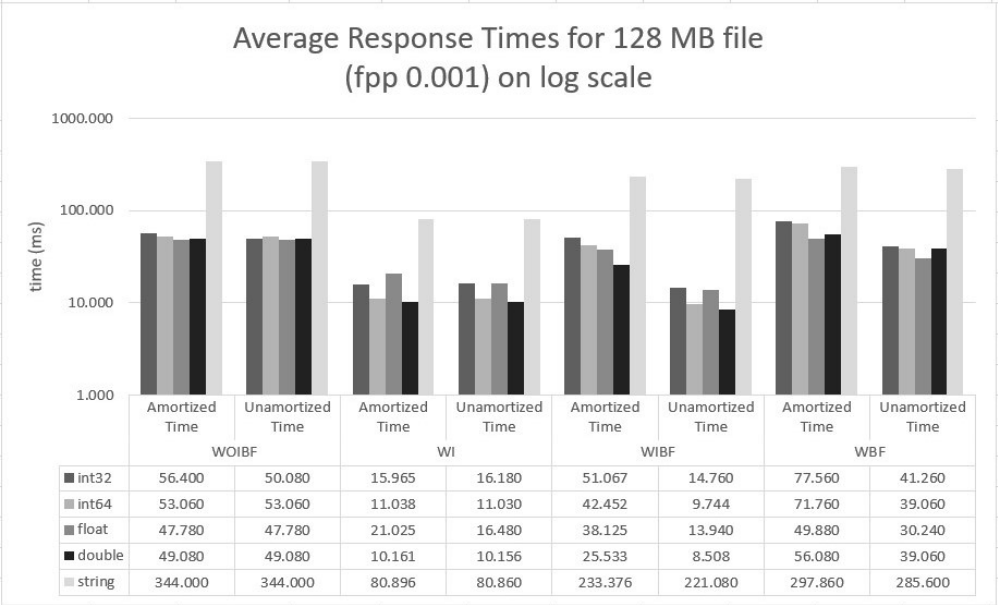
4.2.5 Bloom filter compression

Given the about 2x increase in the size of parquet file due to Bloom filters, and the integral principle of compression in parquet files, we measure the impact of compression on Bloom filters. We notice that using GZip compression codec, the size of Bloom filters reduce from 500 *MB* for five columns to 300 *MB*, that is, we achieve 40% compression. When we apply compression to the parquet file with Bloom filters, an uncompressed parquet file without Bloom filters of size 1.5 *GB* reduces to the size of 756 *MB*, as presented in Table 12.

4.2.6 Bloom filter sensitivity

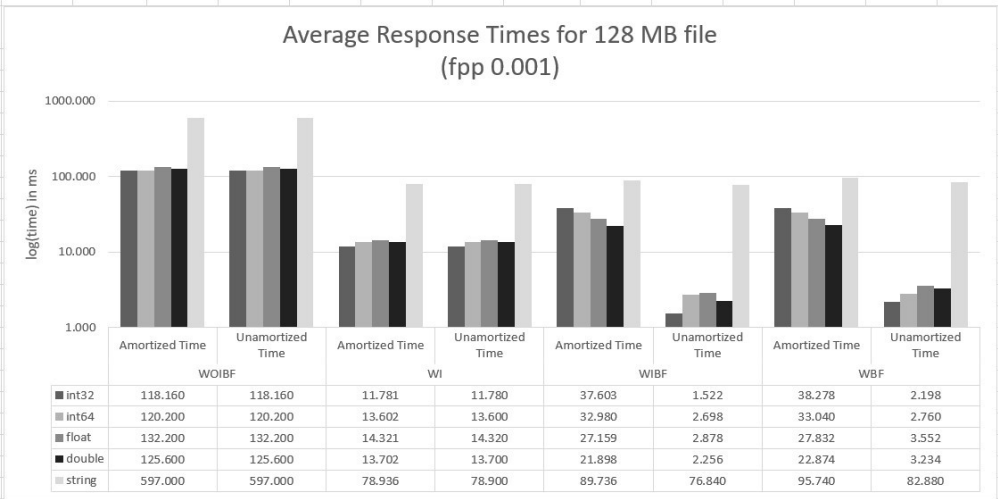
Again motivated by the trade-off between decrease in response time and increase in file size due to adding Bloom filters in parquet files, we run a sensitivity analysis.

In our sensitivity analysis, we do not notice significant changes in the false positive percentage of non–member across the different false positive probabilities settings for Bloom



	WOIBF		WI		WIBF		WBF	
	Amortized Time	Unamortized Time	Amortized Time	Unamortized Time	Amortized Time	Unamortized Time	Amortized Time	Unamortized Time
int32	56.400	50.080	15.965	16.180	51.067	14.760	77.560	41.260
int64	53.060	53.060	11.038	11.030	42.452	9.744	71.760	39.060
float	47.780	47.780	21.025	16.480	38.125	13.940	49.880	30.240
double	49.080	49.080	10.161	10.156	25.533	8.508	56.080	39.060
string	344.000	344.000	80.896	80.860	233.376	221.080	297.860	285.600

Figure 27: Average file reading times for sorted parquet files with 128 MB file for member queries has 10% reduction on log scale using ColumnIndex-OffsetIndex and ColumnIndex-OffsetIndex with Bloom Filters.



	WOIBF		WI		WIBF		WBF	
	Amortized Time	Unamortized Time	Amortized Time	Unamortized Time	Amortized Time	Unamortized Time	Amortized Time	Unamortized Time
int32	118.160	118.160	11.781	11.780	37.603	1.522	38.278	2.198
int64	120.200	120.200	13.602	13.600	32.980	2.698	33.040	2.760
float	132.200	132.200	14.321	14.320	27.159	2.878	27.832	3.552
double	125.600	125.600	13.702	13.700	21.898	2.256	22.874	3.234
string	597.000	597.000	78.936	78.900	89.736	76.840	95.740	82.880

Figure 28: Average file reading times for sorted parquet files with 128 MB file for non–member queries shows has 10% reduction on log scale using ColumnIndex-OffsetIndex and ColumnIndex-OffsetIndex with Bloom Filters.

Table 12: Comparisons of change in file size with compression

Size of original file without SBBF	**Number of Rows**	**Original file Compression without Bloom filter**	**Size of file with SBBF after compression**
1.5 *GB*	10M	Uncompressed	1.8 *GB* (compressing the Bloom filter)
166MB	10M	Compressed (GZip)	756MB

filters with 0.001, 0.01, 0.1, 0.2 and 0.5, for 10M rows with no particular data arrangement. This is an intriguing outcome of the experiment, we analyze further in Section 4.3.

4.3 Analysis

We observe that irrespective of the data arrangement (data with sorting or without sorting), the general trend in the results show a significant decrease in file reading times when using ColumnIndex-OffsetIndex, Bloom filters or a combination of Bloom filter and ColumnIndex-OffsetIndex. This implies that our implementation is does not depend on the dataset and will show the same relative improvement in file reading times.

We see that the difference in time taken to scan files for non–member queries is uniformly higher than the case of member queries for all data arrangements and data types. This can be attributed to the notion that ColumnIndex-OffsetIndex can selectively pick a DataPage to scan, as opposed to full scan without ColumnIndex-OffsetIndex, even for non–member queries. Similarly, Bloom filters can assist in stopping the reading of the file early, upon detection of a value not present in the file.

As expected, we observe that Bloom filters perform better in reducing the file reading times in non–member queries by 20x without ColumnIndex-OffsetIndex and an additional 10% with ColumnIndex-OffsetIndex.

We observe many false positives reported for double and float data type column Bloom filters in member queries. We attribute this occurrence to the approximation we make while hashing the double and float values using 64-bit integers into those Bloom filters. We can conclude this by observing the profiles of the Bloom filter experiments. In the cases where double and float values show high number of false positives, the file reading times are also low. This means that the Bloom filter is functioning as it should, only that approximation of data causes a high occurrence of false positives. In all other cases, the Bloom filter report no false positives and function correctly for non–member queries. We think that there could be a balance between adjusting the false positive probability and the level of approximation in matching the double and float values.

We verify that using ColumnIndex-OffsetIndex reduces the file reading time. This can be attributed to the fact that after reading the ColumnIndex-OffsetIndex, the program running the query can selectively read the exact DataPage that has the potential candidate value.

We verify that Bloom filters perform on par with ColumnIndex-OffsetIndex in all types of queries. In all cases, combining Bloom filters with ColumnIndex-OffsetIndex always offers reduction in file reading times in comparison with not using either of them. Bloom filters and ColumnIndex-OffsetIndex can quickly verify if a value is not present in the file with fewer scans and if the value is present in the file quickly retrieve the value with selective reading of DataPages.

We notice that even if the time taken using ColumnIndex-OffsetIndex in combination with Bloom filter is less, in comparison with plain ColumnIndex-OffsetIndex, the decrease in the file reading time is not as significant as other results. ColumnIndex-OffsetIndex selectively reads a DataPage. Since we use a Bloom filter at the column-level, the value can be detected to be present at the column-level but not at the DataPage level. This implies that the DataPage has to still be scanned to verify if the value is not present in the DataPage. We address this limitation in our implementation with an extension, as future work, in Chapter 5.

The time taken to load ColumnIndex-OffsetIndex is less than the time taken to load Bloom filter because ColumnIndex-OffsetIndex stores a limited number of values as opposed to Bloom filter, whose size varies with the false positive probability. The smaller the false

probability with which the Bloom filter is initialized, the more the number of blocks per SBBF, with blocks that are unused for hashing. This also implies that an application looking to leverage the benefit of using the ColumnIndex-OffsetIndex and Bloom filter can take the maximum advantage only if they load the ColumnIndex-OffsetIndex and Bloom filter onto the memory once before the query operation and continue to use them for further queries. Loading the ColumnIndex-OffsetIndex and Bloom filter separately, for each and every query only adds an additional cost to the file reading time that can be avoided.

With low false positive probabilities of 0.001, 0.01 and 0.1, in most cases the Bloom filters can stop the operation of reading the file early, with less chances of reporting false positives. We notice that compressing the entire file with Bloom filter reduces the file size to approximately half of the original uncompressed parquet file without Bloom filter and could potentially reduce the space taken in storing the parquet file with Bloom filter. To further reduce the Bloom filter load time, we can compress the Bloom filters using GZip to achieve 40% compression and reduce the overall file reading time.

When we measure the Bloom filter sensitivity, we notice that even with high false positive probability such as 0.5, the Bloom filter presents good accuracy in detecting non–member values from the rows of the column. This is because for a false positive probability of 0.5, the structure of the SBBF is such that *all the 10M values in the column are hashed into different blocks and there are unused blocks with no hashed values to cause conflicts. Hence, there is higher accuracy achieved by the Bloom filter with less data (number of rows) and comparatively large sized Bloom filters generated with high false positive probability of 0.5.*

In the indirect comparison of PittCS Arrow with Impala, as observed in Chapter 2 (Table 2) and other Hadoop SQL engines, we note that improvement in file read time in Impala considers the primary goal to improve query execution speeds, with parquet files of limited sizes ($<= 1\ GB$), specific RowGroup parameters (1 RowGroup, $<=1\ GB$) and file system. By considering file read optimization in the two aims of this thesis, PittCS Arrow works for parquet files of size 1 *GB* and 1 RowGroup as well as parquet files of any size with any number of RowGroups, on a linux file system.

4.4 Chapter Summary

In this chapter, we described the instrumentation and performance evaluation of PittCS Arrow. We presented and analyzed the results in detail, which support our two aims in this thesis in optimizing parquet files in Arrow C++. In the next chapter, we conclude our thesis with a summary of our contributions and future work. Further, we discuss our design to implement DataPage-level Bloom filters as a final contribution in this thesis. Lastly, we proceed to discuss the larger implications of PittCS Arrow in open source parquet community.

5.0 Conclusion & Future Work

In conclusion, with our experimental results presented in the previous chapter, we supported our claim that implementing the combination of Bloom filters with ColumnIndex-OffsetIndex can improve parquet file reading times. In this chapter, we summarize our contributions and future work. Specifically, in Section 5.1, we summarize our contributions and findings. Further, we present our design of page-level Bloom filters, as the final contribution in this thesis in Section 5.2. Lastly, we discuss the larger implications of our implementation in the open source parquet community in Section 5.3.

5.1 Summary of Contributions

Motivated by the importance of the parquet files in efficient processing of big relational data, in this thesis, we designed and implemented the PittCS Arrow parquet file, which optimized the reading of parquet files in Apache Arrow C++. Specifically, we made the following four contributions:

- Implemented serialization and deserialization of ColumnIndex and OffsetIndex to support fast DataPage skipping and utilizing the ColumnIndex-OffsetIndex to reduce parquet file reading time in Arrow C++.
- Implemented serialization and deserialization of column-level Split Block Bloom Filters (SBBF) and proposing the novel idea of utilizing Bloom filters in combination with the ColumnIndex-OffsetIndex to further reduce parquet file reading time in Arrow C++.
- Evaluated experimentally the reading of PittCS Arrow parquet and found that:
  (i) using ColumnIndex-OffsetIndex reduces average parquet file reading time by 20x;
  (ii) using SBBF reduces the file reading time by 20x; and
  (iii) using SBBF in combination with ColumnIndex-OffsetIndex by an additional 10% improvement in comparison with using plain ColumnIndex-OffsetIndex or plain SBBF.

- Proposed the DataPage-level SBBF as a two-level Bloom filter to use in combination with ColumnIndex-OffsetIndex and column-level Bloom filter for further read optimization.

5.2 Future Work

PittCS Arrow implements the column-level Bloom filter in Arrow C++ with clear performance gains. However, the information regarding a value being present in DataPages or not is still unavailable. Since, scans that use ColumnIndex-OffsetIndex happen only at selective DataPages, there is scope to improve the DataPage-level file reading time. In our future work, we propose to implement and evaluate page-level Bloom filters which can be used with and without ColumnIndex-OffsetIndex, with and without column-level Bloom filters. As a part of this thesis, we did the first step by proposing a design for page-level Bloom filters, presented in Section 5.2.1, below.

Adding concurrency to parquet reader and writer will additionally bring the benefits of parallelism. We think that adding concurrency to our implementation in PittCS Arrow is a scope for future work because PittCS Arrow is built on top of Arrow and the current Arrow supports concurrency for in-memory operations but not in its implementation of parquet reader. Supporting concurrency in PittCS Arrow will make highly competent framework to read and write files in parquet, on par or better than the existing distributed system solutions to read and write files in parquet, such as Impala on Hadoop file system. We discuss the idea of introducing concurrency in PittCS Arrow in Section 5.2.2

5.2.1 Proposal for page-level Bloom filters

In this section, we propose the structural layout of DataPage-level Bloom filters. As shown in Figure 29, we propose that the DataPage-level Bloom filters can be placed at the end of each ColumnChunk after the last DataPage. Each DataPage has a separate instance of its DataPage-level Bloom filter. Each DataPage-level Bloom filter is an SBBF, which is initialized along with the beginning of writing a DataPage. This is an implementation of a

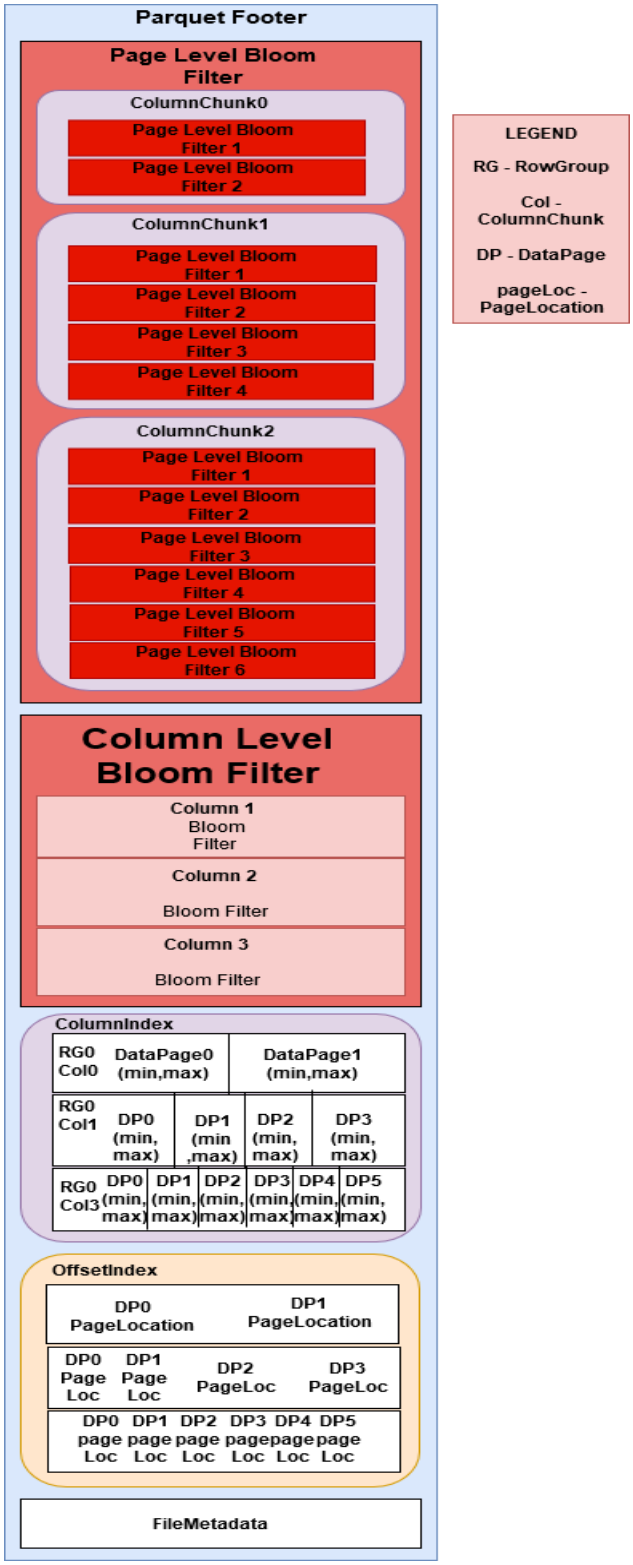


Figure 29: Two-level implementation of Bloom filters shows the parquet file layout after the DataPage-level Bloom filter implementation

two-level Bloom filter which will be used in combination with ColumnIndex-OffsetIndex and column-level Bloom filters.

The rows written into a DataPage are hashed into the DataPage-level Bloom filter. The offsets to the DataPage-level Bloom filters are stored in a list of offsets stored in OffsetIndex, when written with ColumnIndex-OffsetIndex and in a list of offsets stored in ColumnChunkMetadata when written without ColumnIndex-OffsetIndex.

Each DataPage-level Bloom filter is loaded by accessing the offset in the list of offsets at the time of reading the parquet file. By the structure of its layout, a DataPage-level Bloom filter is only accessible when the ColumnChunkReader invokes PageReaders to read the DataPages. In this way, the PageReader call can be revoked by the ColumnChunkReader if the DataPage-level Bloom filter verifies that the value is not present in a DataPage and reading the DataPage can be avoided.

Proposal for Writing page-level Bloom filters

To write page-level Bloom filters, we follow the method of writing column-level Bloom filters as discussed in Section 3.2.1, but with modifications in ColumnWriter.

At steps four, five and six in Figures 15 and 16 of the PittCS Arrow flow control, the ColumnWriter initializes a new instance of Bloom filter for every WriteBatch.

Inside WriteBatch, we decide if a new DataPage is generated. Until a new DataPage is generated, we hash the value, that is being written into the DataPage, into the Bloom filter of the current DataPage.

We repeat this process until we hit a new DataPage. ColumnWriter repeats this process for all DataPages. Finally, the Bloom filters for values in each DataPage are written separately at the end of the ColumnChunk.

When the Bloom filters are written into the file, the ColumnWriter also adds the Bloom filter offset to the list of all DataPage-level Bloom filter offsets in ColumnChunkMetadata.

Proposal for Reading page-level Bloom filters

To read page-level Bloom filters, we follow the method of reading column-level Bloom filters as discussed in Section 3.2.2, but with small modifications.

At steps five and six in Figures 15 and 16 of the PittCS Arrow flow control, we deserialize the DataPage-level Bloom filter, using the list of offsets to the DataPage-level Bloom filters.

The predicate is checked for its presence in any of the selected DataPages.

When using DataPage-level Bloom filters in combination with column-level Bloom filter, the value is first checked for presence by column-level Bloom filter and then the value is checked for presence in any DataPage.

When using DataPage-level Bloom filters in combination with ColumnIndex-OffsetIndex, the value is first checked for presence in any DataPage and then the ColumnIndex-OffsetIndex is used to select the DataPage.

When using a combination of DataPage-level Bloom filter, ColumnIndex-OffsetIndex and column-level Bloom filter, the column-level Bloom filter first checks for the presence of the value in the file, then the DataPage-level Bloom filter checks for the presence of the value in any DataPage. The ColumnIndex-OffsetIndex is used to selectively read only the DataPages that could "probably" contain the values and not the DataPages that do not contain the value with high probability.

5.2.2 Proposal for Reading and Writing with Concurrency

We pose that adding concurrency in reading and writing parquet files in PittCS Arrow will additionally bring the benefits observed in query execution engines such as Impala. To truly run in a distributed system, PittCS Arrow can leverage the structure of the parquet file format to spawn separate instances for different components of the parquet file to incorporate parallelism. For instance, separate instances of RowGroupWriters and RowGroupReaders can be spawned for different separate RowGroups in the parquet file. Each RowGroup is first written into separate files. The FileWriter could act as the master thread in consolidating all the RowGroups into one file while writing the FileMetadata at the footer of the final parquet file.

Table 13: Comparison of PittCS Arrow with state-of-art-implementations for reading and writing parquet files

Application, library, database	**Supported file format**	**Programming Language**	**Statistics**	**ColumnIndex & OffsetIndex**	**Bloom filters**
Hive	Parquet	Java	Y	N	N
Arrow	Parquet	C++	Y	N	N
Vertica	Parquet	C++	Y	N	N
Spark	Parquet	Scala	Y	Y	N
Impala	Parquet	C++	Y	Y	N
PittCS Arrow	Parquet	C++	Y	Y	Y

5.3 Discussion

From our experimental results in the previous chapter, we support our two aims of this thesis to optimize parquet file reading times in Arrow C++. We recommend that adding and utilizing ColumnIndex-OffsetIndex improves the parquet file reading times. We also recommend that using Bloom filters in combination with ColumnIndex and OffsetIndex can further improve the parquet file reading time.

We notice a trade-off associated with improving the file read times and the size of the file. The file size increases by 0.03% on adding ColumnIndex-OffsetIndex and the file size doubles on adding Bloom filters. While adding ColumnIndex-OffsetIndex does not affect the size of the file significantly, the size of the file changes significantly on adding Bloom filters. Our recommendation to achieve optimal outcome is to create parquet files with Bloom filters that have false positive probability greater than 0.001. With small false positive probability, the Split Block Bloom Filter (SBBF) allocates more blocks to hash values, with potentially many unused blocks in the SBBF. This increases the size of the file with the Bloom filter, without improving the file reading time any further than 20x.

We conclude our final contribution in this thesis by summarizing our design proposal to the parquet community for page-level Bloom filters. DataPage-level Bloom filters are initialized and hashed separately with the rows of individual DataPages in a ColumnChunk. Each DataPage has its designated DataPage-level Bloom filter. This DataPage-level Bloom filter can be used at the time of reading the parquet file to avoid reading DataPages irrelevant to the query I/O and reduce the file reading times. With this design proposal for DataPage-level Bloom filters, we present an implementation that goes beyond the state-of-art-implementations in parquet and complements and enhances our best performing PittCS Arrow parquet file in this thesis.

In this thesis, we present that our implementation, PittCS Arrow, can read and write parquet files using ColumnIndex-OffsetIndex and Bloom filters on a general linux file system. As summarized in Table 13, none of the current applications support all these features in parquet readers and writers. PittCS Arrow with the incorporation of column-level Bloom filters is a novel contribution to the parquet open source community and a benchmark for parquet file read optimizations.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[3] Álvaro Alonso Isla. HDFS File Formats: Study and Performance Comparison. Master's thesis, University of Valladolid, 2018.

[4] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *The VLDB Journal*, 7(12):1295–1306, 2014.

[5] Marcel Kornacker, Alexander Behm, and Victor Bittorf et al. Impala: A modern, open-source SQL Engine for Hadoop. In *Proceedings of Seventh Biennial Conference on Innovative Data Systems Research*, volume 15, pages 270–280, 2015.

[6] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The Vertica Analytic database: C-store 7 years later. *arXiv:1208.4173*, 5:1790–1801, 2012.

[7] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Amazon Web Services*. Alpha Press, 2010.

[8] Maziar Kaveh. ETL and Analysis of IoT data using OpenTSDB, Kafka, and Spark. Master's thesis, University of Stavanger, Norway, 2015.

[9] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[10] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, Hash-and Space-efficient Bloom Filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121, 2007.

[11] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* WCB/McGraw-Hill, 2nd edition, 2000.

[12] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. Data Page Layouts for Relational databases on Deep Memory Hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[13] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving Relations for Cache Performance. *The VLDB Journal*, 1:169–180, 2001.

[14] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining Query Execution. In *Proceedings of Second Biennenial Conference on Innovative Data Systems Research*, volume 5, pages 225–237, 2005.

[15] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[16] Michael Stonebraker, D Abadi, A Batkin, X Chen, and M Cherniack. C-store: a column-oriented DBMS. In *The VLDB Journal*, pages 553–564, 2005.

[17] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *The VLDB Journal*, pages 330–339, 2010.

[18] Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *ACM Communications*, 13(7):422–426, July 1970.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), June 2008.

[20] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: Storage for fast analytics on fast data. *Cloudera*, pages 1–13, 2015.

[21] Lars van Leeuwen. High-Throughput Big Data Analytics Through Accelerated Parquet to Arrow Conversion. Master's thesis, Delft University of Technology, 2019.