Pancake Columnar Design

Martin Loncaric

January 20, 2022

Abstract

In order to create a system with high-throughput columnar bulk reads and low-latency incremental appends, PancakeDB has created two new columnar data formats: Pancake Flush Column (PFC) and Pancake Compressed Column (PCC). The novel motivations for these formats, different from Parquet and Orc, are to support incremental appends to PFC files and and to leverage database-managed metadata and grouping for both PFC and PCC files. We show experimentally that PFC files reduce incremental write costs by 79-97% and read costs by 93%-99%, and that PCC files reduce storage, disk read, and network costs over smart Parquet servers by 12-25%.

1 Introduction

PancakeDB is a novel database that ingests event data and streams for bulk reads. Ingestion is the task of taking a real-time data, often one row of data at a time over the course of years, and building a table. Though common in industry, this type of ingestion system typically required a custom solution [1] [2] (prior to PancakeDB).

An example bulk read ingestion system may support the following two operations:

- `write(table_name, row)`: appends a single row, or list of key-value pairs, to the table.
- `read(table_name, columns)`: returns a list of all historical values for each column requested.

So for instance, these operations would be valid:

- `write("purchases", {"user_id": "a", "cents": 123})`
- `write("purchases", {"user_id": "b", "cents": 234})`
- `read("purchases", {"cents"})`, responding with `{"cents": [123, 234]}`

In practice, each table may receive many writes per second over the course of years, resulting in billions of rows. To alleviate the challenge of bulk reading billions of rows, typically over an intra-datacenter network connection, software developers use columnar data formats to encode the response. Execution engines like Apache Spark process that response, decode the rows, and perform arbitrary SQL queries.

PancakeDB introduces two new columnar data formats:

- Pancake Flush Column (PFC) for storing a single uncompressed column in a format that supports appending a small number of rows at a time.
- Pancake Compressed Column (PCC) for storing a single compressed column.

In conjunction with PancakeDB which manages metadata and executes compactions, these formats enable incremental writes, differentiating them from existing industry favorites like Parquet and Orc. Per our results (Section 4), using PancakeDB also reduces storage, disk read, and network costs by 12-25% for bulk reads.

2 Prior Work

One early columnar data format is Row Column File (RCFile). RCFiles introduced the feature of *row groups* - subdivisions of the file, each contains a portion of the table's rows. Within each row group the RCFile layout is columnar. The main advantage of row groups is memory management: a deserialized row group is likely to fit into memory, so processes can write or read very large RCFiles one row group at a time [3].

RCFiles place information about the row count and byte length of each row group column in a row group metadata header. This enables readers to skip unnecessary columns, and it enables a quick row count by summing the row count metadata for each row group. A reader can decode just the metadata sections by scanning through the file for row group markers [3].

RCFiles also support column-wise data compression with the intended motivation that some columns might require heavier compression than others [3].

Orc and Parquet are newer columnar formats that also offer the above features: row groups, row group metadata, and column-wise compression. They each offer an improvement over RCFiles by including file-level footers with byte positions for each row group, so that readers no longer need to scan for markers. This enables metadata scans and reader parallelization over row groups [4] [5].

These columnar formats are typically stored in one of two places: an unstructured object store such as AWS S3 or Google Cloud Storage [6], or systems like the Hadoop ecosystem that can process the data locally before sending a response [7]. We will refer to the latter as "smart servers".

Object stores support upload and download operations for any file object, including downloading a specific range of a file, but lack the ability to append to a file. Therefore, common approaches of incrementally building columnar datasets in object storage rely on uploading entirely new files. Also, object stores have no internal understanding of columnar data formats, so in order to read exactly one row group column from a Parquet or Orc file, one must

1. download the last few kB of the file to obtain the footer and parse it to find the offset for the row group,

2. download the bytes following that offset to obtain the row group metadata header and parse it to find find the row group column offsets, and

3. download the bytes specified by those offsets to obtain the row group column.

Each read from an object store has high latency, so to avoid 3 round trips, execution engines like Spark resort to downloading the whole file and then parsing, now matter how small the query.

In contrast, smart servers support domain-specific operations like counting the number of rows or returning the data for a specific column. By processing the data locally before responding, they can avoid reading the entire columnar file from disk or sending it over the network. All examples the authors are aware of use Hadoop Distributed File System (HDFS).

3 Method

3.1 PancakeDB Data Layout

PancakeDB automatically groups each table partition into *segments*, by default of roughly 1 million rows each. At any given time, each table partition has an *active* segment that receives writes, and all previous segments have no additional data appended to them. When the active segment exceeds 1 million rows, it becomes inactive, and a new active segment is created. A segment is similar to a row group in that it contains metadata about the rows and keeps each chunk down to a manageable size. However, in PancakeDB, each segment column has a separate file. PancakeDB manages these files by keeping most partition- and segment-level metadata cached in memory, allowing readers to obtain the equivalent of row group metadata in a single round trip of much lower latency.

When new rows are written to a PancakeDB segment, they are briefly held in a row-format *staged rows* file. This allows the database to respond to the query in a consistent manner without slowly appending to many separate column files. In a frequent background loop ($\sim$ 10s), staged rows are flushed into the PFC columns for the segment, and the staged rows file is truncated. On reads, staged rows are treated as a virtual part of the PFC file, dynamically converted into columnar format. In an infrequent background loop, segments containing PFC columns are compacted into PCC columns. The new PCC columns constitute a new segment *version*, and the old segment version is deleted soon after. In this way, there is always at most one PFC file and one PCC file per segment column, and inactive segments are always entirely compacted as PCC files.

The active segment is also compacted when half full, so under normal operating conditions ($>$10 million rows written over $>$ 1 day), over 95% of rows are in PCC files, less than 5% are in PFC files, and less than 0.02% are in staged rows.

This data layout also allows PancakeDB to minimize data sent for queries requesting a subset of columns. Whereas Parquet and Orc store file offsets that point to the start or end of each row group column, PancakeDB simply stores a file path to each segment column. This allows PancakeDB to easily avoid loading irrelevant columns from disk or sending them over the network.

3.2 PancakeDB API

To leverage its data layout, PancakeDB supports the following API calls, simplified for explanatory purposes:

- `write_to_partition(table_name, partition, rows)`: appends the rows to the table partition's active segment staged rows file.
- `list_segments(table_name, partition)`: responds with a list of segments, including their ID's and metadata like row count.
- `read_segment_column(table_name, partition, segment_id, column)`: responds with bytes for the virtual PFC file, bytes for the PCC file, and a codec to use on the PCC file.

3.3 Pancake Flush Column Design

PFC files are built to be columnar, support incremental appends, and enable seeking to a specific row index. Seeking is used during state recovery in the event that some columns were successfully flushed and others were not. To enable fast seeking, PancakeDB periodically places the byte `0xfe` into the byte stream followed by 4 bytes for the row count up to that point. All other usages of `0xfe` are escaped with the escape byte `0xff` followed by two's complement, `0x01`. During state recovery, PancakeDB can quickly read the final few kB of data in a PFC file, find the last `0xfe` and its corresponding row count, and know how many rows are present.

A null value is appended to a PFC file by appending the byte `0xfd`. Non-null values are appended to a PFC file by encoding their contents, escaping any `0xfd`, `0xfe`, or `0xff` bytes, and appending to the file. For example, fixed-size values, such as a 64-bit integer or a single byte of a string, are encoded by their canonical byte representation. Nested values are encoded by a 2-byte count for the number of subvalues, followed by the encoding for each subvalue.

3.4 Pancake Compressed Column Design

PCC files are entirely optimized for bulk reads, eschewing search indices. Each PCC file consists of two sections: compressed repetition levels and compressed fixed-size values.

Repetition levels are a familiar concept from Parquet [4]. PancakeDB uses them in almost the same sense, but saves some complexity and file size by disallowing nested nulls. For instance, in Parquet a column with type `List<Float>` may contain entries `null` and `[1.23, null]`. In PancakeDB for the same schema, `null` would be valid, but `[1.23, null]` would not. As a result, PCC files use repetition levels to mean:

- 0: null
- 1: next top-level list
- 2: next 2nd-level list
- ...
- $n+1$: next fixed-size value

So for instance, the 3 strings "hi", "bye", and null could be encoded by repetition levels $2, 2, 1, 2, 2, 2, 1, 0$ and fixed-size characters "h", "i", "b", "y", "e".

The repetition levels are compressed with Quantile Compression [8].

The fixed-size values are compressed with a codec depending on data type. PancakeDB defaults to Zstandard [9] compression for string-like data types and Quantile Compression for numeric data types, including timestamps. Its design is flexible, and more codecs may be added in the future.

4 Results

To compare PancakeDB vs Parquet files, we generated a 10 million row dataset with 10 columns: 5 string columns whose entries are all randomly chosen English words and 5 integer columns whose values follow a power law distribution. We used ZStandard compression level 5 for Parquet, the same level PancakeDB uses on string columns. Since Parquet does not yet support Quantile Compression, we also used ZStandard on the Parquet file's integer columns. We grouped the rows together in an identical way for each data format: 10 row groups of 1 million rows each for Parquet, and 10 segments of 1 million rows each for PancakeDB.

Using this dataset, we compared PancakeDB's performance versus Parquet. For PFC files, we compared the disk space and number of distinct files required to store an incrementally growing dataset (Table 1). For PCC files, we compared the total storage size versus well-compacted Parquet storage (Table 2) as well as the disk read and network requirements (Table 3).

PFC files can be appended to without increasing the number of files, so unsurprisingly they also require an order of magnitude less data than incrementally added Parquet files.

PCC string columns take 12% less data than Parquet's, largely due to eliminating pagination, metadata, and definition levels that are not useful for bulk reads. PCC integer columns take 25% less data, taking advantage of the new Quantile Compression codec.

5 Conclusion

PancakeDB manages PFC and PCC files in a way that enables efficient incremental writes while still reducing storage, disk read, and network costs over Parquet. PancakeDB is able to do this because of design decisions within PFC and PCC, relying on the fact that PancakeDB manages their metadata intelligently. We expect this new methodology to adjust the data engineering landscape, lowering compute costs and simplifying internal systems for ingestion for bulk reads.

References

[1] Netflix Technology Blog, "Evolution of the netflix data pipeline." https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905, 2016.

[2] Uber Data Engineering, "Dbevents: A standardized framework for efficiently ingesting data into ubers apache hadoop data lake." https://eng.uber.com/dbevents-ingestion-framework/, 2019.

[3] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, (USA), p. 11991208, IEEE Computer Society, 2011.

[4] "Apache parquet documentation." https://parquet.apache.org/documentation/latest/.

[5] L. Leverenz, "Orc language manual." https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC.

[6] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Witakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia, "Delta lake: High-performance acid table storage over cloud object stores," *Proc. VLDB Endow.*, vol. 13, p. 34113424, aug 2020.

[7] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. Kuff, D. K. A. Leblang, N. Robinson, D. R. S. Rus, J. Wanderman, and M. M. Yoder, "Impala: A modern, open-source sql engine for hadoop," in *Proceedings of the 7th biennial conference on innovative data systems research*, 2015.

[8] M. Loncaric, "Quantile compression." https://graphallthethings.com/posts/quantile-compression.

[9] "Zstandard." https://facebook.github.io/zstd/.

	PFC	Parquet
1 incremental row / bytes	**97**	3,935
10 incremental rows / bytes	**967**	4,627
disk read time for 10k rows in 1-batches / ms	**1**	120
disk read time for 10k rows in 10-batches / ms	**1**	15

Table 1: PFC files can be appended to, unlike Parquet files, which keeps disk usage an order of magnitude smaller and relatively contiguous. Disk read time was measured on an AWS throughput-optimized HDD.

	PCC + metadata files	Parquet
storage size / MB	**338**	402

Table 2: PCC files offer lower storage costs, primarily through a lighter-weight format.

	PancakeDB	Parquet + smart server	Parquet + object store
`count(*)` disk read	**0***	**0***	402
`count(*)` network	**0**	**0**	402
`select(string_col)` disk read	**51**	58	402
`select(string_col)` network	**51**	58	402
`select(int_col)` disk read	**17**	23	402
`select(int_col` network	**17**	23	402

Table 3: All sizes reported as rounded to the nearest MB.
*exactly 0 bytes, since PancakeDB and smart Parquet servers cache this information in memory.