Row versus column storage

Goetz Graefe

For a comparison of row- versus column-storage, consider the following simple, standard storage structures for a relational database.

- Table: a traditional table stored row after row, accessible by row identifier (a device-page-slot address or a b-tree key, often the table's primary key) – supporting all single-table predicates as well as "fetching" entire rows to "cover" any query.
- Indexes: a single column at a time (or very few columns at a time), ordered by column values (plus row identifier) – supporting "=", "in", "<", "between", and other predicates – multiple index scans support intersecting lists of row identifiers as well as joining two or more indexes on row identifiers to "cover" a query in non-traditional "index-only retrieval."
- Columnar storage: one column at a time (or very few columns at a time), ordered by row identifier – supporting all kinds of predicates as well as "materializing" one column at a time in order to "cover" a query.

Each one of these storage structures can be compressed, partitioned (within or across nodes in a cluster), and organized as write-optimized storage, e.g., as log-structured merge-forests, but all that is ignored for now.

1 First example: a very selective query

The example focuses on a single large table and a query with multiple predicates as well as multiple columns in the output. Let's assume no correlations or functional dependencies among columns in a table with 1 B = 1,000,000,000 rows. Columns A and B are of type integer, column C is of type string. The query is: Select A, B, C, D, E, F from T where A = 8 and B = between 13 and 26 and C like "%oops%". Each single-column predicate is satisfied by 1% of the rows. The query retrieves 1,000 rows.

1.1 Querying column storage

In column storage, the three predicates can be evaluated by three scans of the columns A, B, and C. Predicates propagate from column to column and might reduce the scan effort for any remaining predicates. In the example, after predicate evaluation for columns A and B, the predicate for column C is evaluated 100,000 times, not 1 B times. If column C is compressed using a dictionary, the predicate can be evaluated only once for each distinct value in column C, after first scanning the dictionary to determine codes for string values satisfying the predicate clause. Depending on the size of column values and of pages, the I/O pattern for column C may or may not be a full scan, e.g., if each I/O fetches less than 10,000 values of column C.

Once row identifiers are known of rows satisfying all three predicates, three read accesses for each row fetch the remaining three columns. In the uncommon case that each such read access fetches more than 1,000,000 column values, one access may fetch multiple useful values for columns D, E, or F.

Ignoring this uncommon case, the query requires three full columns scans of 4-8 GB each plus 3,000 random read accesses. (With a typical traditional disk drive, a random access takes time roughly equal to 1 MB of scanning. The total time is very roughly equal to 20 GB of scanning.)

1.2 Querying a traditional table

A traditional table without sort order or indexes is not worthy of further consideration. If each record occupies 200 bytes, a scan of 200 GB is required for any query including the example.

If the table's storage is sorted on columns A or B (e.g., using a primary index to store the table), range scans can reduce the scan volume to 1/100th of the table. A sort order on columns A and B (as major and minor sort keys) reduces the scan volume to 1/10,000th of the table or 20 MB, which is very competitive.

1.3 Querying traditional secondary indexes

Traditional row stores usually employ multiple indexes per table. Here, we assume simple single-column indexes on columns A, B, and C. Of the index on column A, a scan retrieves 1% of the index – a single list of row identifiers attached to a single key value in the index (80-160 MB without compression).

In the index on column B, where 14 distinct key values satisfy the predicate, a 14-way merge retrieves altogether 1% of the index. Merging these sets of row identifiers (within the index on column B, and from the index on column A) requires no sorting, only merging. A multi-key index on columns A and B retrieves only 1% of 1% of the index (1-2 MB).

The index on column C requires a full scan plus sorting 10 M row identifiers (~80 MB) to merge with the row identifiers from indexes on columns A and B. The index scan evaluates the query predicate only once per distinct value in column C. If there are few distinct key values in column C, e.g., 100, merging can replace sorting, very similar to the strategy for column B.

Bit vector filtering can propagate the predicates on columns A and B to the search within the index on column C, with some loss of precision due to hash collisions. For a table of 1 B rows, a bit vector filter on row identifiers requires at least 2 B bits or ¼ GB; or better 2 to 4 times as much. If such a bit vector filter is feasible, it reduces the sort effort to about 1,000 row identifiers (~8 KB).

Fetching the remaining columns from the table (by row identifiers) requires 1,000 random read accesses.

1.4 Comparison summary for a selective query

For the example query selecting 1,000 of 1,000,000,000 rows, traditional storage requires less I/O for the predicates and less I/O for fetching remaining columns but a little extra effort for merging lists of row identifiers.

For columns A and B, traditional indexes are superior by a factor depending on the number of distinct values in the column and within the query predicate. For column C, predicate propagation is somewhat simpler and somewhat more precise in column storage, and as efficient as predicate evaluation in column stores optimized for duplicate key values.

For fetching the remaining columns, a row store is superior for small row counts (e.g., 1 row per million), whereas a column store is superior for large row counts (e.g., 5% of all rows). A row store is superior for large column counts (e.g., 10 remaining columns) and inferior for small column counts (e.g., 1 or 2 remaining columns). Thresholds depend on column, row, and page sizes as well as the fraction of rows to fetch, using the calculation min (#rows to fetch, #rows in the table × #bytes per row ÷ #bytes per page) as approximation for the I/O to fetch rows.

2 Second example: a less selective query

Let's modify the prior example such that each predicate clause selects 10% of the table (not 1% as in Section 1). The entire query selects 1‰ or 1,000,000 rows among the table's 1,000,000,000 rows.

Querying column storage requires three scans for predicate evaluation. Propagating the selection on columns A and B to the scan of column C reduces string evaluations by 100 times. Assuming 1,000 or more column values per page, fetching columns D, E, and F requires full column scans. If each database page holds 1,000 column values, 6 column scans require 6,000,000 sequential page reads.

A full scan of the base table may be competitive in this example. Querying traditional indexes and row storage scans 10% of the indexes on columns A and B but the entire index on column C. Bit vector filtering may or may not be effective, depending on the size of the bit vector filter. Unless database pages hold more than 1,000 rows, fetching columns D, E, and F requires 1,000,000 random page reads. If indexes exist on columns D, E, and F, scanning and joining those indexes with the index scans on columns A, B, and C might be more efficient and in fact comparable to querying column storage.

Compression in column storage might permit 5,000 to 10,000 column values per database page. In that case, column storage is vastly superior with 600,000 to 1,200,000 sequential page reads versus 1,000,000 random page reads in the traditional database format. Equivalent compression of indexes on columns D, E, and F makes row storage with secondary indexes competitive with column storage and its column scans, assuming query optimization and query execution can join secondary indexes in order to avoid fetching rows from a table's primary storage structure.

It seems that, with respect to query performance, 1‰ query selectivity is a typical break-even point between row or column storage; details and the exact fraction depend on page, row, and column sizes, column count in the query, query predicates, sort order, skew, compression, effectiveness of bit vector filtering, and of course the set of available secondary indexes and of supported query execution plans in a traditional row-oriented database. A query selecting 1% or even 10% of a table's rows but only a subset of columns typically requires much less I/O in column storage than in row storage. Plentiful secondary indexes and non-traditional query execution plans can compete with column storage, in particular joining two or more secondary indexes on their common row identifier information in order to "cover" a query with "index-only retrieval."

3 Third example: a query for much of a table

Let’s modify the prior example some more by reducing it to a single predicate with 10% selectivity, i.e., the query retrieves 100,000,000 rows of the table’s 1,000,000,000 rows.

Performing the selection within an index is 5-10× faster than in an unsorted column store.

In both row and column formats, fetching remaining columns requires reading entire storage structures, e.g., the table’s primary index or the relevant columns. Inasmuch as only a subset of the columns are required, a column store can scan and fetch faster than a traditional primary index. As the fetched data is much larger than the data volume in the initial selection, the performance difference for selection (applying the query predicate) hardly matters. On the other hand, predicate evaluation is faster in the secondary index because each distinct column value exists only once in an index sorted on the column.

4 Variants and variations

4.1 Compression

As column stores rely heavily on scans, compression is essential for best query performance. In the example, naïve storage of column A requires 1 B integers, whereas compressed storage using a dictionary requires 1 B values of 1 byte each plus a dictionary with 100 entries. The total storage requirement for such a column is just above 1 GB.

Naïve storage of a secondary index on column A requires 100 distinct key values plus 1 B row identifiers. Compressed storage replaces almost all of the 1 B row identifiers with numeric deltas, i.e., differences between row identifiers. Each such delta fits in a single byte in most cases. Therefore, the total storage requirement for a secondary index is just above 1 GB.

With equal care and development effort for compression, the storage requirements of row and column storage are about similar. It is difficult to construct a case in which they differ by a factor 2 or even more.

4.2 Covering indexes and index-only retrieval

A traditional secondary index that supports the selection predicate by its sort order and its structure and that renders fetch operations obsolete by including all required columns in the index entries is often the best alternative to both row and column stores. Due to the selection predicate, the query evaluation plan scans only a fraction of the table’s rows; and due to index entries with fewer than all columns, the query evaluation plan scans only a fraction of the table’s columns. In this way, a traditional secondary index that “covers” a query is often faster than traditional indexing in a row store and faster than a pure column store. This technique is also known as “index-only retrieval.”

4.3 Indexes with non-key columns and without key columns

A secondary index may include further columns in addition to key columns and row identifiers. Due to the syntax used in some database systems, these are also called “stored” or “included” columns. They do not contribute to the sort order of the index because the row identifiers are unique, even if the user-defined index keys are not. Indexes with included columns are intended to cover many queries within the workload. Typical use cases include primary and foreign keys in many or all secondary indexes.

A secondary index may have only row identifiers and included columns, i.e., no user-defined leading key columns. With row identifiers as the only sort key, such an index is very similar to a column in a column store. This is particularly true if there is only one included column, if row identifiers and included columns are stored separately within each database page, and if compression is applied to both. For example, run-length encoding can compress a continuous sequence of row identifiers to a single value, a count, plus perhaps in increment.

4.4 Bitmap indexes

If a single key value occurs in more than 10% of all rows, a bitmap index offers even better compression, i.e., representing a set of row identifiers (their integer values) by neither their values nor their differences but as a bitmap.

Compression of sparsely populated bitmaps identifies runs and uses run-length encoding; note that these run lengths are very similar to numeric differences in sorted lists of row identifiers.

Bitmaps may replace merge operations with set operations, e.g., intersection and union operations by bit-and and bit-or instructions. In the example, the 14-way merge is a union, the subsequent merge with a list of row identifiers from the index on column A is an intersection.

4.5 Disjunctions – "or" queries and "in" lists

Like the 14-way merge in the example query (Section 1.3), "or" predicates map to union operations on lists of row identifiers. For example, a query predicate "where A = 8 or B = 47" requires the union of two lists of row identifiers, easily computed using merge logic or bitmap representations.

Query predicates with "in" lists, e.g., "where A in (8, 18, 28, 38)", are a syntactic shortcut for common disjunctions. In other words, this list predicate is equivalent to an "or" predicate and thus requires a union operation on lists of row identifiers. This is quite similar to the "between" predicate in the example query (Section 1.3).

4.6 Multi-column indexes

A compound index on columns (A, B) could, with a single 14-way merge, retrieve just 1% of 1% of the row identifiers (for the first example query). These about 100,000 row identifiers would be sorted and possibly compressed (to perhaps 2 bytes per row). The example query "where A = 8 and B between 13 and 26" requires merging 14 lists, each with about 7,000 row identifiers.

4.7 Zigzag merge join

Whereas a traditional merge join scans its two inputs from start to end, a zigzag merge join attempts to skip forward in one input based on the next key value in the other input. For example, when intersecting two lists of row identifiers, one with 100,000 entries and one with 10,000,000 entries, only one in 100 entries of the second list can have a match. Thus, it may be more efficient to search for the next match (using some variant of binary search or exponential search) than to scan (which is a variant of sequential search). In general, a zigzag merge join can skip forward in either input depending on input sizes and key value distributions. Special cases include two join inputs with vastly different key ranges in the join column and merge join with an empty input.

4.8 Bit vector filters

Finally, bit vector filters are another technique of speeding up joins and intersection operations, in particular in parallel query execution and in operations that spill within a memory hierarchy, e.g., from DRAM to disk but also CPU cache to DRAM.

In the example query, a bit vector filter can reduce the number of string operations. A bit vector filter may represent row identifiers of rows satisfying the predicates on columns A and B; this bit vector filter can be probed prior to pattern matching in column C. With 100,000 rows remaining after the predicates on columns A and B and 20% false positives, this query execution strategy requires pattern matching on only 125,000 values of column C rather than 1,000,000,000 values, an 8,000-fold improvement.

Both zigzag merge join and bit vector filters improve query execution performance for both column and index storage.

4.9 Lightweight indexing

Lightweight indexing, also known as zone maps in some products, divides table storage into blocks of moderate size, e.g., 1-4 MB, and summarizes the contents of each block. Blocks typically hold rows of only one type. The summary for a block might be a pair of rows (of the same type as the block's contents) populated with minimum and maximum values for the respective columns. Another possible summary uses a bit vector filter per column, or minimum and maximum as well as bitmap, or one or none of these based on some policy.

Minimum and maximum work particularly well for columns highly correlated with the sort order. A bit vector filter works particularly well for columns with only equality predicates in queries, few distinct values, and a non-random distribution of values to blocks.

One advantage of lightweight indexing (compared to traditional indexing, e.g., using b-trees) is efficient creation and maintenance while appending to the base data. In that usage, it works well for columns correlated with the order of insertion, i.e., with time.

B-tree indexes can accommodate lightweight indexing. For example, if some columns in a table are correlated, an index on one column can include the other columns; and while the first column rules the branch nodes, minimum and maximum of the correlated columns may augment each parent-to-child pointer within the index. This index can then also support reasonably efficient search on these columns.

In the case of date or time columns, which often are correlated, a time interval may provide tighter bounds and more effective selections than minimum and maximum of date values. For example, an index on 'ship date' might add 'receipt date' or 'receipt delay.' Receipt dates vary just as much as ship dates, but receipt delays (transit durations) are quite consistent and their minimum and maximum can be captured very compactly, e.g., in a b-tree's root node.

This technique also applies to partitions in a write-optimized b-tree or forest.

4.10 Write-optimized storage structures

The C-store design and the Vertica product popularized the idea of write-optimized storage for databases, e.g., an unsorted in-memory row store to absorb insertions faster than a sorted column store can, at the expense of occasional reorganization of the storage structure. For row stores, the traditional write-optimized index formats are log-structured merge-trees and stepped-merge forests. The specific storage format can be a forest (many individual trees) or a partitioned b-tree. Continuous merging is a recent variant using an in-memory transactional multi-version index, with additional benefits from a linear partitioned b-tree for each merge level.

These index designs augment traditional b-tree indexes. Insertion by bulk operations and by small transactions are captured (and new rows sorted) in memory and written to storage devices using large and efficient I/O operations. Among many alternative designs, continuous merging is most efficient as it combines design elements of data streaming and external merge sort. A transactional multi-version in-memory index enables transactional insertions, continuous reorganization, and queries in snapshot isolation. For modifications of existing information, logical updates map to insertions of replacement records and deletions map to insertions of tombstone records. Read-only locks within read-write transactions, even phantom protection, are possible but add complexity.

4.11 Domain indexes

Row storage or single-table indexing are not the opposite of column storage; they are the midpoint between column storage and domain-organized storage, also known as master-detail clustering or merged indexes. For example, the domain "customer identifier" occurs in many tables and their indexes, e.g., tables for customers, orders, invoices, etc. Domain-organized storage or domain indexing merges all these indexes into a single storage structure, e.g., a b-tree ordered on customer identifier, with subsequent sort keys identifying record types as well as additional keys as appropriate, e.g., order number or invoice number. – Domain indexes can turn joins into scans and thus speed up query execution as well as guide query optimization. They excel in assembly of individual complex objects and in index-to-index navigation, notably in hierarchies with multiple levels of memory and storage. They do not excel in business intelligence and analytics, where column stores and secondary indexes perform particularly well.

Ordered domain indexes can benefit from write-optimized storage structures and from internal lightweight indexing, among other optimizations.

5 Required query processing techniques

In order to reduce sorting efforts, metadata must describe any sort order of storage structures, e.g., sort order on row identifiers for each index key, query optimization must exploit sort order (as well as partitioning, compression, etc.), and query execution must provide efficient operations for sort-based set operations, i.e., with variants of merge join or generalized join. For example, internal and external merge sort ought to exploit tree-of-losers priority queues and offset-value coding; set operations for row identifiers ought to work on lists compressed using differences instead of values; and database updates ought to employ deferred, incremental, on-demand updates for secondary indexes, materialized and indexed views, and b-tree structures.

Index maintenance ought to exploit techniques from log-structured merge-forests, stepped-merge trees, write-optimized storage, external merge sort, and adaptive merging. Suitable techniques for concurrency control, logging, and recovery are assumed, e.g., snapshot isolation transaction exploiting multi-version storage, a fine granularity of locking, short lock durations, etc.