No Bits Left Behind

Eugene Wu
MIT CSAIL
eugenewu@mit.edu

Carlo Curino
MIT CSAIL
curino@mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

One of the key tenets of database system design is making efficient use of storage and memory resources. However, existing database system implementations are actually extremely wasteful of such resources; for example, most systems leave a great deal of empty space in tuples, index pages, and data pages, and spend many CPU cycles reading cold records from disk that are never used. In this paper, we identify a number of such sources of waste, and present a series of techniques that limit this waste (e.g., forcing better memory locality for hot data and using empty space in index pages to cache popular tuples) without substantially complicating interfaces or system design. We show that these techniques effectively reduce memory requirements for real scenarios from the Wikipedia database (by up to $17.8\times$) while increasing query performance (by up to $8\times$).

1. INTRODUCTION

One of the core tenets of good database design is that resources, especially RAM and disk bandwidth, should be used efficiently. DBMS designers have used a number of techniques to optimize hardware utilization, such as keeping hot tuples in the buffer pool, and performing sequential access to the disk whenever possible. Every bit of memory or bandwidth used to fetch or store data that is not needed is a bit lost.

Despite this fact, many database implementations do a poor job of using these precious memory resources. For example, the typical index fill factors have been shown to be around 68% [10], and this figure can get much worse in the face of updates and deletes [6]. This 32% of space left unused is the product of a conscious design decision aimed at reducing expensive node split operations. Nonetheless, since index sizes often rival the size of the tables themselves, this is a very significant amount of memory dedicated to storing absolutely nothing.

Another example results from the use of fixed-size pages, which need to be read in their entirety just to fetch a single tuple. If the other tuples on a page are not needed, most of this I/O is completely wasted. Using a workload trace from Wikipedia, we found that 99% of accesses to Wikipedia's *revision* table, which stores metadata about article revisions, are focused on the 5% of tuples that represent the latest revision of the articles. The index clustering used in Wikipedia leads to heap pages that contain as little as 2% of frequently queried data.

A third example of waste has to do with inefficient encoding of data. This can arise due to poor choices in physical representation (e.g., a string representation for an integer, using bytes to store booleans or lack of compression) or semantic decisions (e.g., storing full timestamps when the application only requests years). We performed an analysis of tables on multiple databases and found between 16% to 83% of waste due to inefficient physical encoding.

In this paper, we describe these and several other examples of database system resource waste in more detail, and propose a number of solutions to these problems, including:

1. A technique to deal with the problem of wasted space in index pages, that "recycles" this space, using it as a cache for hot data. We show that the performance benefit we obtain on a substantial class of queries from the actual Wikipedia workload can be orders of magnitude higher, and that this technique can be implemented in a way that doesn't change index APIs or increase system complexity.
2. A technique to deal with the problem of cold tuples polluting pages containing hot data, that clusters commonly accessed tuples together, improving performance a factor of 6 while reducing total index sizes a factor of 19.
3. Analysis techniques that deal with the problem of encoding waste by identifying the most efficient type for a given column, and treating the programmer-supplied type merely as a declarative "hint" rather than an actual storage type. Additionally, we suggest ways to eliminate redundant data and to exploit ID fields by embedding semantic information in the values.

In summary, various artifacts of database system implementations, and poor user choices account for significant amounts of wasted storage in databases today. We believe there is a huge opportunity for research that i) optimizes DBMS structures and policies in a workload-specific manner, ii) empowers users with tools that automate waste detection and, iii) performs automatic space allocation and layout tuning. Work on self tuning databases [2] that reorganize column layouts [5], select optimal indexes [3], and calculate optimal database knobs [9] are high level optimizations in the right direction, however there is still ample opportunity at every level of the DBMS architecture, as we will illustrate in the rest of this paper.

This paper presents our *vision* to improve resource utilization in databases; as such, many of our experiments are simplified or incomplete—we plan to further extend this analysis in the future.

2. UNUSED SPACE

Unused space is defined as waste due to bytes that are allocated on disk or in RAM but *do not contain data*. Such waste is often allocated or reserved for expected data writes in the future, or as the result of poor data placement policies. We present one possible way to reuse this waste as it occurs in B+Tree pages.

2.1 Index Caches

The average fill factor of B+tree index pages is 68% [10]. In practice, real implementations are often substantially less – for example, in a frequently updated database for our CarTel (`http://cartel.csail.mit.edu`) research project, the fill factor is only 45%. The waste could be eliminated by compacting the pages with a fill factor of 100%, which would benefit read-only workloads, but lead to excessive node splits and result in low utilization of those pages in the presence of inserts. We want a way to use this space as a cache (that can be overwritten when the space is needed to store legitimate index contents); we describe

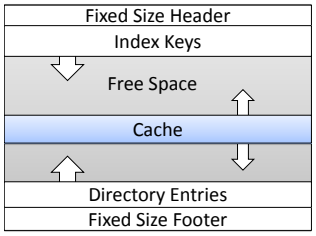


Figure 1: Anatomy of an index page

such a scheme in the rest of this section.

B+tree leaf nodes often store tuple pointers rather than physical tuple data, requiring an additional page access to retrieve the associated data page. In OLTP workloads, if the page access necessitates an additional disk I/O, the overhead can be substantial. By caching some tuples' data in their corresponding index leaf pages, the index can be used to directly answer queries, which makes use of unused space and avoids following pointers from leaf pages. We show experiments that suggest this can lead to substantial performance improvements on a trace of the Wikipedia workload without substantially complicating the index API or implementation.

As an alternative to a caching-based approach, one could imagine using covering indexes (i.e., adding all of the fields used in any query to the index key), which can also avoid accessing the heap to answer queries. However, covering indices still store cold data, waste space and bloat the index size, which wastes more total bytes, and increases pressure on RAM.

2.1.1 Cache Overview

The index cache stores field values of the most frequently accessed tuples in the index leaf pages, without introducing any disk I/O overhead. Most importantly, we preserve existing index maintenance algorithms and piggy-back off normal query processing to perform cache maintenance.

To simplify the discussion, we assume that the index keys and the tuples are fixed length. Metadata about the field lengths is stored on the index's metadata page in memory. The typical index page layout (Figure 1) consists of a fixed size page header and footer, a region of index key values, and a logically ordered directory of small fixed size pointers to the keys. The data and directory start from the high and low areas of the page and grow towards the center, and the page header stores pointers to the start and end of the free space, which is used as the cache store.

The cache space is split into slots where the beginning of each slot is aligned to the cache entry size (e.g., if the item size is 25 bytes, then the start of each slot is a multiple of 25). A slot either contains data or is zeroed out. Items begin with the tuple id to identify the data, followed by the field values.

When an index page is read during a lookup for a tuple with id t, we scan the cache slots for a cached version of t. Queries that project a subset of the index key and the cached fields can be answered without retrieving the data pages from the buffer pool, or worse, from disk (e.g., if the query projects A,B,C, the index key is A, B, and we cache C.) On a cache miss, we construct and insert the new cache item into the cache of the index leaf page that references the tuple, possibly evicting an existing item from that page's cache has not been accessed recently. In order to not introduce additional disk I/O, cache modifications do not dirty the page.

We avoid impacting existing index operations by allowing key inserts to freely overwrite the periphery of the cache space at any time. Thus it is important to keep hot items in the interior of the free space, where they will be overwritten last. It is possible to calculate the most stable location, S, as a function of (ignoring fixed size headers and footers) the index key size K, directory pointer size D, and page size P: $S = \frac{K}{K+D} \times P$

The cache is logically split into buckets of N slots each. When an item is first inserted, it is placed in a random free slot, or if no slots are free, evicts a random item in a peripheral bucket. On a lookup, we swap the item with a random entry in the adjacent bucket closer to S. This way, when the key and directory regions expand, the least accessed cache items are overwritten first.

2.1.2 Efficient Consistency Enforcement

So far, we have described how to read and populate the index cache. We now briefly outline an efficient index cache invalidation scheme to handle system crashes and modifications to a field whose cache page has been stolen. To support full index invalidation, we add a cache sequence number to each index page header (CSN_p), and maintain a global CSN for the index (CSN_{idx}). By preserving the invariants that 1) $CSN_p <= CSN_{idx}$ and 2) a page cache is only valid if $CSN_p = CSN_{idx}$, we can efficiently invalidate the entire cache by incrementing CSN_{idx}. Although this guarantees correctness, it is inefficient to invalidate the entire index on update queries. Instead, we create and store predicates that uniquely identify the updated tuples and append them to an in-memory log. When an index page is read during normal query execution, we zero the cache space if any predicates match keys in the page. If the list grows above a threshold, we can increment CSN_{idx} and clear the list. Finding a way to efficiently index and search these predicates is an area for future work.

2.1.3 Latching

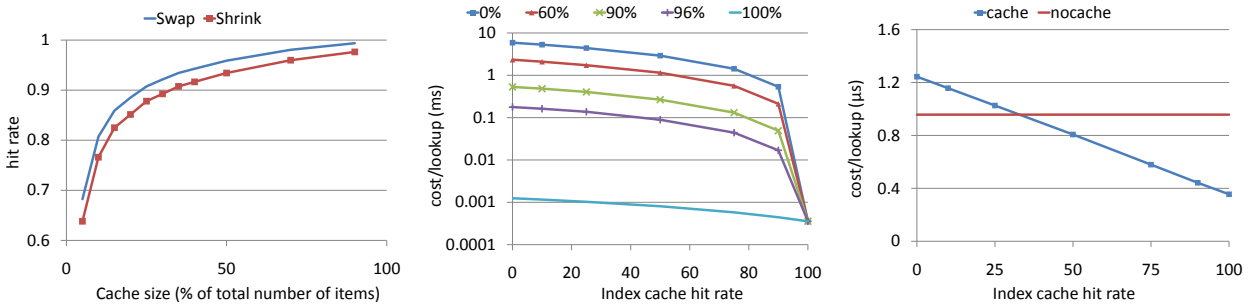
One point of concern is that we are turning every index leaf page read into a write, which may introduce higher lock/latch contention. Fortunately, normal index operations can freely overwrite the cache, so we only need to acquire short term latches for the duration of the cache writes. Additionally, we can give up a write operation if the latch is not immediately available.

2.1.4 Performance

In an analysis of Wikipedia, we found that the most popular class (40%) of queries accesses the *page* table using the *name_title* index, which uses a composite key of (namespace id, page title), and projects up to 4 additional fields. The index contains 360 MB of key data and, assuming that the index is 68% full and all 4 fields are cached (25 bytes/cache item), the index can store up to 7.9 million cache items – representing over 70% of the tuples in the *page* table and allowing us to answer nearly all of these queries through the index (due to skew in the page access).

We ran a simulation to study how the hit rate varies with the cache size using a zipfian distribution similar to Wikipedia ($\alpha = .5$) and found that the swapping based cache management algorithm exhibits high hit rates (Figure 2(a)). Each point is the average hit rate after 100k lookups and the x-axis is the percentage of the items that the cache can hold (which will vary depending on the tuple size and index fill factor). We compare $Swap$, which simulates a read-only workload that does not overwrite the index cache (constant cache size), and $Shrink$, which simulates a read/insert workload that overwrites half of the index cache at a constant rate over the duration of the experiment. $Swap$ exceeds 90% hit rate when the cache size is only 25% of the table. $Shrink$ only reduces the hit rate by 5%, showing that swapping effectively moves hot items towards the middle.

Figure 2(b) is a micro-benchmark that illustrates the performance benefits of index caching over a random lookup distribution. We assume that the index is fully in memory, and simulate the index and buffer pool using large in-memory arrays. An index cache miss must access a random page in the buffer pool, and a buffer pool miss must read a page from an on-disk file. We



(a) Hit rate as cache size varies, zipfian distribution ($\alpha = .5$)

(b) Query performance as index cache and buffer pool hit rates vary.

(c) Index cache performance with buffer pool hit rate = 100%

Figure 2: Index caching experiments

measure the performance improvement as the index cache (x-axis) and buffer pool (lines) hit rates vary from 0 to 100%. Different lines show what happens with increasing buffer pool hit rates. The hit rate depends on the available RAM for the buffer pool, as well as the sizes of the data pages and working set. Higher buffer pool hit rates, unsurprisingly, lead to better performance. The x-axis shows what happens as the hit rate of our index cache increases — higher cache hit rates substantially improve performance as we avoid both the memory access to the buffer pool as well as the additional disk I/O in the event that the data page is not in the buffer pool. Thus, even with when the dataset fully fits in memory (100% buffer pool hit rate), we still see a 2.7× improvement in performance by avoiding the additional memory accesses to pages in the buffer pool. We measured the index cache hit rate for our subset of the Wikipedia workload to be above 90%, suggesting that Wikipedia should see query performance gains of up to several orders of magnitude by employing this technique.

Figure 2(c) extends the previous experiment to illustrate the overhead of using index caching. We assume the buffer pool hit rate is 100% and see that index caching has 0.3us of overhead, which disappears when the cache hit rate exceeds 35%, and ultimately outperforms $nocache$ by 2.7×.

In our experiments, we hand picked the fields to cache in the index and discovered a number of useful heuristics for selecting these fields. First, the fields should be stable (i.e., rarely updated). Updates must access the updated field values in the heap tuple, so frequently updated fields do not benefit from index caching. Second, the cached fields should be chosen to fully answer a large class of queries. We found that over 40% of Wikipedia queries can be directly answered through an index cache on 4 attributes. These heuristics are at odds with each other, so the optimal choice of fields to cache is dependent on the workload, and is an interesting direction for future work.

2.2 Additional Directions

There are many other types of data that might be cached in index pages, for example: statistics, pre-computed query results, or other index pages are all options. In addition, these same concepts can be applied to data pages as well. For example, data pages can cache the results of foreign key joins, to avoid additional disk accesses for join queries. More ambitiously, if a data page is consistently read during the execution of a complex join query, caching the query results can offer substantial query performance improvements.

3. LOCALITY WASTE

We define locality waste as waste due to *ineffective placement* of data. A common example is when a full tuple must be read into memory even though the query only accesses a small subset of the fields. Equally wasteful is when cold tuples are read into memory only because they are co-located near a single hot tuple (e.g., on the same page). In this section we provide an example of access frequency based horizontal partitioning that improves query lookups in Wikipedia's *revision* table by over 8×, and sketch an improvement to index caching in update intensive scenarios using vertical partitioning.

3.1 Horizontal Partitioning

The first form of locality waste we discuss results from hot and cold tuples being co-located on the same data page. This occurs when the tuple placement strategy (e.g., append to table) does not match the access patterns. We argue for clustering and partitioning tables by access pattern rather than by range or hash partitioning. This is particularly beneficial for lookup based workloads that access a small set of hot tuples distributed throughout the table.

For example, Wikipedia's *revision* table stores a tuple for every new page revision; each tuple contains a unique revision ID, the page ID, a pointer to the text content, and several additional fields. 99.9% of page requests access the 5% of the tuples that represent the most recent revisions for each page; however, these hot tuples are scattered throughout the table, with as few as one hot tuple per data page (2% utilization). Hash and range partitioning are not possible because the access frequency is not related to any field values.

Figure 3 shows the possible performance benefits of access based clustering on Wikipedia's *revision* table. Our clustering algorithm relocates hot tuples by deleting then appending them to the end of the table. The 0%, 54%, and 100% curves vary the percentage of hot tuples that are clustered while the $Partition$ curve additionally creates a separate partition for the hot tuples. The workload is derived from 10% of 2 hours of Wikipedia's Apache logs. Lookup performance improved by 1.8× after clustering 54% of the hot tuples (2% of the table), and by 2.15× after clustering all hot tuples. Creating a partition for hot tuples reduces query costs by 8.4×. The reason partitioning has such a profound impact is that reducing the index size from 27.1 GB for the full table to 1.4 GB for the hot partition allows the entire index to fit in RAM.

The properties of the workload dictate how to identify hot tuples and move tuples between the hot and cold partitions. In Wikipedia, hot *revision* tuples are those that are pointed to from the *page* table, so newly inserted revision tuples can replace the previously hot tuple for the same page, which is then moved to the cold partition (note that this does require updating foreign key pointers and/or using forwarding tables to redirect queries using old ids to the new tuples). Other applications may have different policies, or require automated tools to keep track of access patterns.

3.2 Vertical Partitioning

Just as horizontal partitioning reduces locality waste, vertical partitioning can also be used to improve database performance [1, 7] and maximize memory utilization. For example, separating

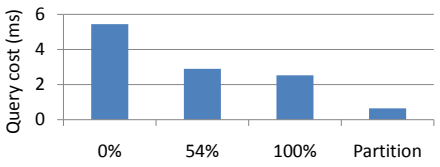


Figure 3: Cost per query

the cached fields from the uncached fields can complement index caching by minimizing the amount of redundant data read into memory when queries access fields not found in the index. Additionally, splitting the table based on the field update rate can increase the write density per page. Weighing the benefit of vertical partitioning against cost of merging the partitions together makes makes this problem non-trivial and interesting.

4. ENCODING WASTE

We define encoding waste as waste stemming from data that is *inefficiently represented*. This can be due to using inappropriate field types (strings for int values), poor compression, or more generally, storing data at a higher semantic granularity than what the application expects. For example, if the application stores timestamp values yet only expects years then the field contains encoding waste. In this section, we revisit the process of schema definitions and present techniques that exploit the semantic information in fields.

4.1 Automated Schema Optimization

Column values can be analyzed to understand the typical value range or the content properties (e.g., only numerical strings) and compare them against the declared types in the schema. Similarly, large fields that are either never accessed or only projected or accessed through equality predicates are good candidates for compression.

We analyzed several of the largest tables in the Cartel and Wikipedia databases and found that they can all reduce their physical encoding waste by 16% to 83% through simple techniques. For example, Wikipedia's *revision* table uses a 14 byte string to represent a timestamp that can easily be encoded into a 4 byte timestamp. Most commonly, we found a large number of int fields that store small value ranges which can easily be encoded in 8, or even 4 bits. Although individually small optimizations, the total amounted to over 23.5 GB (20%) of waste in the tables we inspected. Removing these unused bits increases the data density and consequently improves query execution performance.

We argue that schema type definitions should be treated as hints rather than hard constraints. Schemas are typically designed before the application is built when the workload is unclear. At this stage, it is safer to over-allocate space. However, as the database grows, the need for performance tuning, and the knowledge of field values and allocation requirements grows. At this point, automated tools can infer true field types and value distributions to modify internal field definitions and minimize encoding waste, or suggest these optimizations to the user.

4.2 Semantic IDs

The value of ID fields is often emantically meaningless to the application, other than as a unique identifier. In other words, the uniqueness, not the *value*, is important to the application. For example, Wikipedia and many applications that use object-relational-mapping (ORM) layers define an AUTO_INCREMENT primary key field for each table with such properties. Similarly, fields such as version number are used to induce order for a unique entity, whereas the actual value is inconsequential.

One response to this waste is reduction. Fields can be reduced if proxies exist whose values exhibit the same properties that the application expects. For example, ID fields representing uniqueness can be eliminated and the tuple's physical address can be used as a proxy. Column stores already infer the id using the tuple offset [8]. More generally, if there is a functional dependency $X \rightarrow Y$ and the semantic properties of Y can be directly inferred from X, then Y can be dropped.

The second response is to exploit the semantic opaqueness of primary ID fields and reassign the value to improve database performance. We propose embedding partition information directly in the ID field as a mechanism to implement the policy described in Section 3.1. If the data is clustered on the ID field, then simply updating the ID value is enough to physically move the tuple. Otherwise, the hot tuples can be shuffled to the end of the table by transactionally deleting and inserting the tuples.

This same idea can be applied to simplify query routing in distributed partitioned databases. Recent database partitioning work [4] attempts to find a partitioning that minimize the number of distributed transactions for a given workload. Unfortunately, this may require data placement at a per-tuple level, which necessitates a large routing table that maps tuple IDs to their physical location. Such tables can easily become a resource and performance bottleneck and limit the scalability of the routing infrastructure. Embedding a tuple's physical location in its ID alleviates this bottleneck and we believe it is an exciting area of future work.

5. CONCLUSIONS

In this paper, we highlighted three classes of waste in modern database systems—unused space, locality waste, and encoding waste. We suggested several techniques that have the potential to reduce or reuse this waste to dramatic effect. In particular, we proposed *access-based horizontal partitioning*, which groups hot tuples together, and *index caching*, which reuses unused space in B+Tree indexes, and showed that they can result in order-of-magnitude efficiency gains. In addition, we introduced the ideas of interpreting the schema as a layout hint, dropping implicit fields, and embedding information in fields based on application-level semantics. The encouraging results we obtained suggest that it may be time to revisit canonical designs (e.g., B+Trees with a 68% fill factor) in favor of more efficient ones, especially when (as we have shown) this efficiency can be obtained without impacting API or UI complexity.

6. REFERENCES

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In SIGMOD, 2004.
[2] S. Chaudhuri. Self-tuning database systems: A decade of progress. In in VLDB, 2007, pages 3–14.
[3] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In VLDB, 1997.
[4] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. PVLDB, 3(1), 2010.
[5] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In CIDR, 2007.
[6] T. Johnson and D. Shasha. Utilization of b-trees with inserts, deletes and modifies. In PODS. ACM, 1989.
[7] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In SSDBM, 2004.
[8] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In VLDB, 2005.
[9] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In VLDB, pages 20–31, 2002.
[10] A. C.-C. Yao. On random 2-3 trees. Acta Informatica, 9:159–170, 1978.