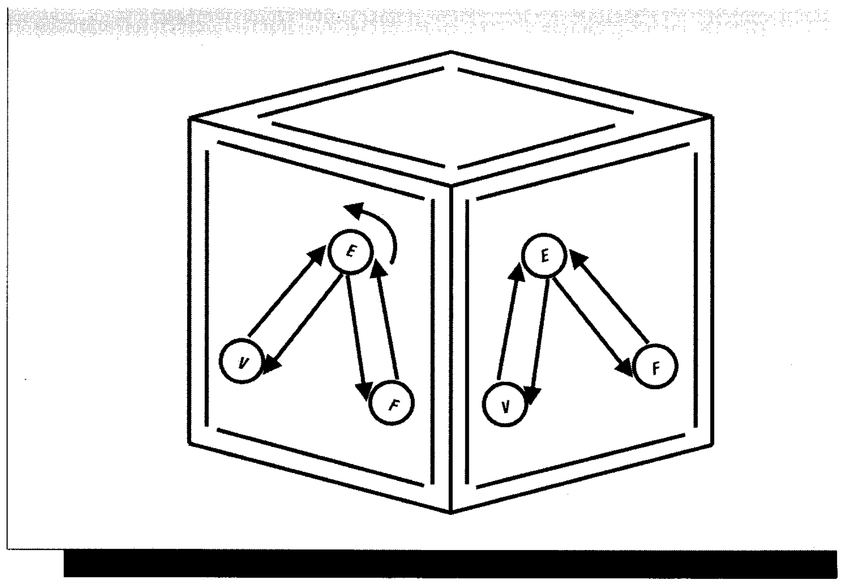


Performance Anomalies in Boundary Data Structures

Seshagiri Rao Ala
City University, London

Previous boundary data structures assumed that data resides in main memory. This proposed Δ-shaped data structure proves more compact and efficient under certain conditions.

We rely on CAD data in a number of computer-integrated manufacturing applications: machine control, automatic inspection, packaging, vehicle guidance, and so forth. Whether the CAD data is represented by some conventional, winged edge (WE), or symmetric data structure (SDS), engineers and managers need to have fast access to it. But alternative data structures such as WE and SDS trade off storage efficiency for access time, and they're designed for specific applications. For example, the winged triangle data structure[1] explicitly assumes the absence of edge-based queries, which is true only of Boolean operations in solid modeling. In computer vision applications—a key component of the CIM environment—edge-based queries are common. We need an integrated approach for structuring the data so that we can use it in a range of applications. The integrated approach ensures the integrity of data and avoids the redundancy of multiple databases tailored to individual applications.

Working toward such a fast and integrated approach, I have sought to estimate different data representations' performances accessing CAD data within a CIM environment. Other researchers implicitly assumed that CAD data resides in main memory; they estimated data access based simply on RAM access times. However, for access to very large databases (such as for a nuclear power plant), main memory probably can't hold all the data. Rather than assuming that data resides in main memory, I assume that data will be accessed using a paging mechanism in a virtual memory environment.

The method I describe has excellent storage efficiency ($6E$, with E being the total number of edges). It also permits improved access time in a virtual memory environment. I present the results of my research here, as well as results for similar performance measures associated with an alternative representation.[2] But first, let's look at the work others have done.

0272-17-16/92/0300-0049$03.00 ©1992 IEEE

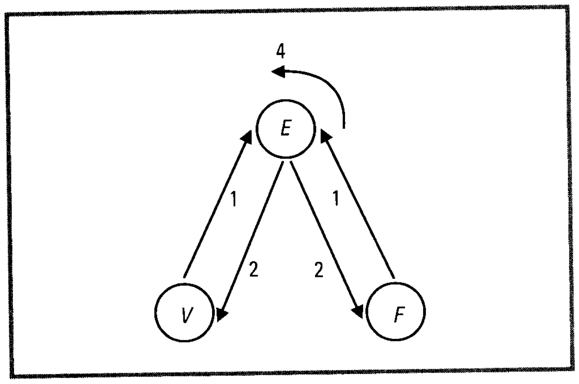


Figure 1. Winged-edge data structure.

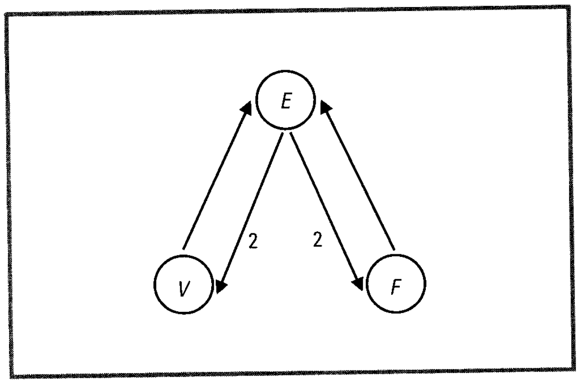


Figure 2. Symmetric data structure.

Previous work

A classic discovery was the winged-edge (WE) data structure[3] illustrated in Figure 1. Woo discovered the symmetric data structure, shown in Figure 2. Several others modified and implemented the SDS (for example, De Floriani and Falcidieno[4]). Several derivatives of the WE data structure followed. The proliferation of edge-based data structures, ostensibly to improve access efficiency, enriched the three basic entities (face, vertex, and edge) with three new entities: loop, cavity, and segment. Loop and cavity entities found applications for multiply connected faces (for example, Weiler[5]) and multiple shell objects, respectively.

Note that in the figures the numbers on the arcs ($V \rightarrow^{1} E$ in Figure 1) imply that for each vertex, one topologically adjacent edge is explicitly stored. The arcs without a number, like $F \rightarrow E$ in Figure 2, imply a variable number of edges adjacent to a face.

In the WE data structure, a traversal (such as extracting the edges of a face) requires an additional check for the direction of traversal of each edge, since each edge occurs in the traversal of two faces (see Figure 3, where edge e occurs in both faces f and f_2). To save on this time-consuming check, researchers split the single-edge entity into two records (for example, e into s_1 and s_2) in vertex edge (VE) and face edge (FE)[5,6] data structures and three records (two segment records and one edge record connecting the two segments, such as s_1, s_2, and e) in hybrid edge[7] and half edge[8] data structures (see Figure 4). Each segment participates in describing one face only (such as s_1 in f). Also, a segment has only one vertex associated with it (for example, in the VE scheme, $s_1 \rightarrow V = v$), unlike the edge record of the WE scheme, which has a two-vertex array field. Each segment in VE and FE data structures references its mate segment and two segments around its vertex and face, respectively. (For example, in the VE scheme, $s_1 \rightarrow S = s_{12}, s_{22}$, while in the face edge scheme, $s_1 \rightarrow S = s_{11}, s_{41}$.)

Like segments in face edge, each segment in half edge and hybrid edge references two segments around its face. However, a segment does not reference its mate segment. Instead, an edge record binds it with its mate. The hybrid edge and half edge

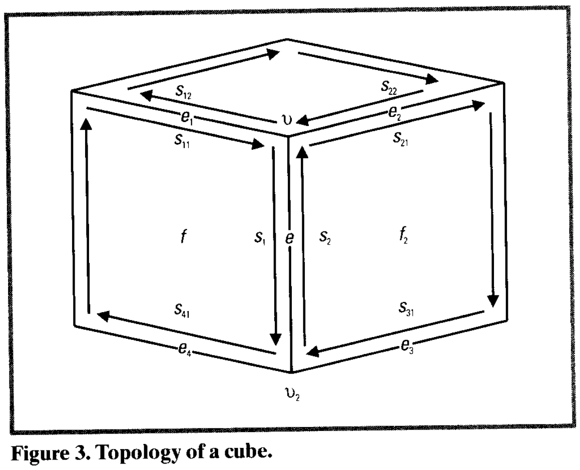


Figure 3. Topology of a cube.

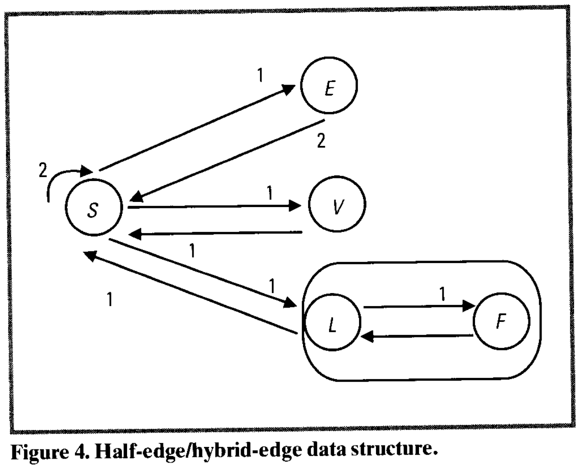


Figure 4. Half-edge/hybrid-edge data structure.

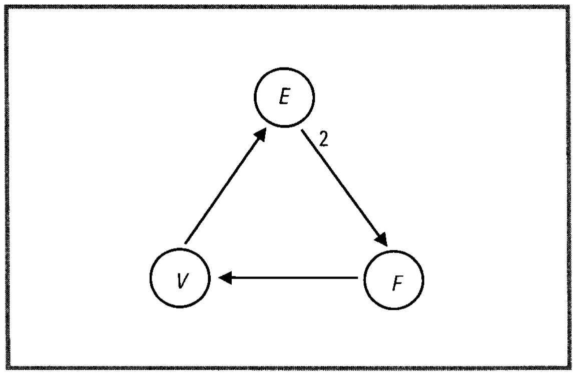


Figure 5. Δ data structure.

differ only in their support structure (unlike half edge, hybrid edge has no $v \rightarrow^{1} E$ but a cavity entity) and linked list implementation (half edge uses a doubly linked list, while hybrid edge uses a singly linked one). These differences hold no interest for comparative studies, so I consider the two identical in this article and refer to them as HE.

Woo[9] and Weiler[5] studied data structure efficiency. Both, however, lacked the perspective of a virtual memory environment. Wilson[6] analyzed several data structures for communicability. I extend the analysis for virtual memory environments. Incidentally, the Δ data structure[2] shown in Figure 5 has the least storage ($6E$, versus $9E$ and $8E$ for WE and SDS, respectively; see Table 1) among the constant-time data structures. Thus we can consider it the right choice for data exchange between different CAD systems.

Virtual memory and databases

Virtual memory makes it possible to run programs much larger than main memory. For the virtual memory reference mechanism, we divide both main memory and the program instruction and data space into pages of fixed size (for example, 8 Kbytes in Sparc workstations). For a program to access any item, the page to which the item belongs must reside in main memory. The system places the program pages into the physical frames (main memory pages) to make them available for programs requesting data. Requests not satisfied by pages residing in main memory cause page faults.

Each of the entities has a record encoding its fields (see Figure 6). Note that the term "record" signifies a collection of related information, all of which tends to lie in the same physical storage unit (a page) and which the system must retrieve before a program can access the fields. Once a record is available in main memory, the field comparison cost is negligible compared to the disk access cost.[5]

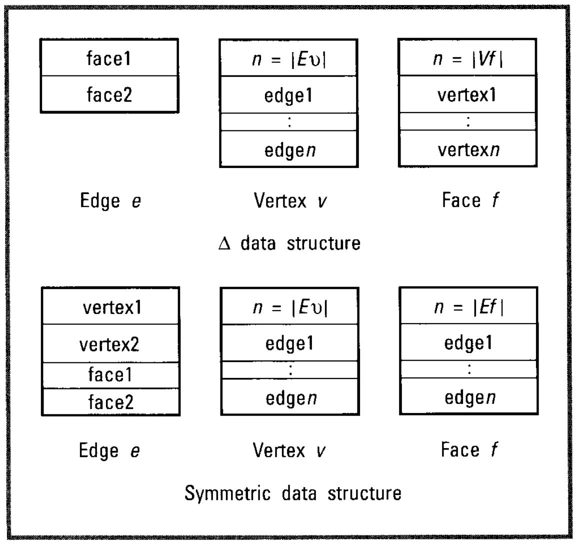


Figure 6. Record structure.

To access any database record, the disk block to which the record belongs must reside in main memory. The database management system (DBMS) maintains an area in main memory for the database buffer pool. Slots in this buffer are physical frames into which the system places disk blocks to make them available for programs requesting data. Data requests not satisfied by blocks residing in the buffer cause database faults.

These database faults are like page faults in virtual memory. Similarly, the DBMS must implement block fetch and replacement policies. Stretching the analogy further, virtual memory maps virtual memory pages to physical frames, while the DBMS maps the block address space (each block identified by a number) to the buffer pool slots.

Empirical results

Previously reported experiments (see the references in Kearns and DeFazio[10]) in a virtual memory environment demonstrated that data references induce more paging than instruction references because of their greater randomness. Also, page turning frequency decreases with the proportion of data and instruction space resident in main memory.

Table 1. Storage values for different relations.[9]

Rel.	$F\rightarrow V$	$V\rightarrow E$	$E\rightarrow F$	$F\rightarrow E$	$E\rightarrow V$	$V\rightarrow F$	$V\rightarrow V$	$F\rightarrow F$	$E\rightarrow E$
i	1	2	3	4	5	6	7	8	9
m_i	$2E$	$2E$	$2E$	$2E$	$2E$	$2E$	$2E$	$2E$	$4E$

Table 2. Record accesses for different data structures.

	$\upsilon \to E$	$e \to F$	$f \to V$	$\upsilon \to F$	$e \to V$	$f \to E$	$\upsilon \to V$	$e \to E$	$f \to F$
Δ	1	1	1	$N_{E\upsilon} + 1$	$N_{Fe} + 1$	$N_{Vf} + 1$	$N_{F\upsilon} + 1 + N_{E\upsilon}$	$N_{Fe} + 1 + N_{Ve}$	$N_{Vf} + 1 + N_{Ef}$
SDS	1	1	$N_{Ef} + 1$	$N_{E\upsilon} + 1$	1	1	$N_{E\upsilon} + 1$	$N_{Fe} + 1$	$N_{Ef} + 1$
WE	$N_{E\upsilon} + 1$	1	$N_{Vf} + 1$	$N_{F\upsilon} + 1$	1	$N_{Ef} + 1$	$N_{V\upsilon} + 1$	1	$N_{Ff} + 1$
HE	$1 + 3N_\upsilon$	$1 + 2$	$1 + N_f$	$1 + 3N_\upsilon$	$1 + 2$	$1 + N_f$	$1 + 3N_\upsilon$	7	$1 + 3N_f$
VE	$1 + N_\upsilon$	2	$1 + 2N_f$	$1 + N_\upsilon$	2	$1 + 2N_f$	$1 + N_\upsilon$	2	$1 + 2N_f$
FE	$1 + 2N_\upsilon$	2	$1 + N_f$	$1 + 2N_\upsilon$	2	$1 + N_f$	$1 + 2N_\upsilon$	2	$1 + N_f$

Some database studies reported strong sequentiality (sequences of increasing database block addresses in a reference string) and weak locality in contrast to virtual memory, which exhibits strong locality (references to a subset of the block space). Locality in databases arises when the data used by one transaction are also used by another. Unlike virtual memory, very little rereferencing occurs within a given transaction. Blocking several records increases locality. This is analogous to the increase in locality (and the consequent reduction in page faults) of virtual memories with dense packing of the active storage area.

In the case of virtual memory, the working set size exhibits a rapid decline in slope beyond a window size. But the above database studies reported that working set size continues to be a linear function of window size, thus indicating the absence of locality. If the buffer management policy uses a least recently used (LRU) policy, which is biased towards locality, then the access time for sequential data might approach the secondary storage access time.

We can exploit sequentiality behavior in database references by prefetching (in a single I/O operation) the next consecutive P blocks in anticipation that they will be referenced in the near future. This policy consumes P buffer slots and incurs extra time to transfer them into main memory. However, it is advantageous in high-latency storage media.

Table 3. Sum of record accesses for different data structures.

		Δ	WE	SDS	HE	VE	FE
N	3	33	27	23	61	39	39
	6	51	45	35	103	66	66

Lagging I/O speeds

While CPU processing times have diminished to 0.05 of their previous levels, disk seek times are half what they were and transfer times have reduced by a factor of 4 (for example, see Kearns and DeFazio[10]). Thus, seek time has become more critical than CPU time or disk transfer rate. Field comparison depends solely on CPU time and main memory access, while record access time depends primarily on seek time. Thus, over the years record accesses have gained more importance.

Determining record access costs

First, to clarify the notation used here: An uppercase letter like E (denoting the entity) refers either to the set of all edges or its cardinality, depending on the context. To refer to a particular instance of an entity, I use a lowercase letter.

The notation for a relation or mapping is an arrow between two entities. For example, $f \to V$ is the set of vertices adjacent to a given face f, where N_{Vf} refers to the cardinality of the set.

Note that

$$N_{Ev} = N_{Fv} = N_{Vv} = N_v \quad (1)$$

$$N_{Ef} = N_{Ff} = N_{Vf} = N_f \quad (2)$$

$$N_{Ee} = 4, \quad N_{Fe} = N_{Ve} = 2 \quad (3)$$

N_v and N_f refer to the number of adjacent neighbors that can be edges, faces, or vertices for the given vertex and face, respectively. We obviously have $N_v \geq 3$, $N_f \geq 3$, and $N_e \geq 2$. N without a subscript refers to N_v and N_f. Woo and Wolter[11] proved that the average value of N_v or N_f is at most 6 for any solid. Hence, we have $3 \leq N \leq 6$.

Note that the above equations assume the objects are 2-manifolds. To simplify my exposition, I defer the treatment of the extra entities loop and cavity. Thus, the estimates assume only three basic entities V, E, and F and, in the case of HE, an additional *segment* entity (collapsing the loop and face entities into one, as shown in Figure 4).

Table 2 lists the number of record accesses for different schemes. The following example clarifies the method of computation: $e \to V$. In WE, $e \to V$ requires accessing the record corresponding to a single edge, while in HE, we first need to access $e \to S$ to get the two segments s_1 and s_2 of the edge e, then access two more records $s_1 \to V$ and $s_2 \to V$. Thus, we incur two extra record accesses by adopting HE instead of WE for the relation $e \to V$.

Table 3 lists the ranges of record accesses, simplifying the figures in Table 2. It uses the equalities in Equations 1, 2, and 3 and sums record accesses for all nine queries. For comparison, also assume $N_v = N_f = N$.

Weiler's estimates[5] for VE, FE, and WE are per adjacency element (and hence need multiplication with the appropriate N and increment of unity for initial conditions) except for $e \to F$, $e \to V$, and $e \to E$. The minimum number of record accesses occurs for SDS ranging from 23 to 35. The worst case occurs for HE, which is nearly three times the SDS record cost.

In Table 4 we find the the minimum record accesses, for $3 \leq N \leq 6$, in each storage class starting from $4E$ up to $20E$. Table 4 also gives example data schemes for each class. For example, the $6E$ class requires a minimum of 51 record accesses and stores the relations listed in Table 1, with i values of 1, 2, and 3—that is, the relations $F \to V$, $V \to E$, and $E \to F$. With a storage limit of $4E$, we can store at most two relations (such as $F \to V$ and $V \to E$). Since the minimum number of relations to connect three entities is 3, queries other than the two explicitly stored relations (such as $E \to F$) require file inversion—that is, a sequential scan of the entire file, which has $O(E)$ entries. You can see this from the E in Table 4.

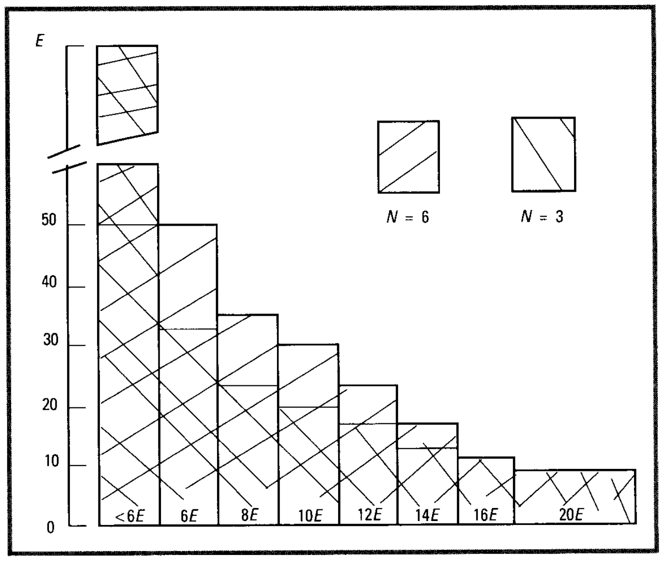


Figure 7. Reduction in number of record accesses.

Δ gives best return on storage

Figure 7 reveals that Δ (storage $6E$) has the maximum reduction in number of record accesses per unit storage. Thus, it is the elbow of the storage-time curve. The next best change occurs for the SDS ($8E$). Table 4 exhibits the law of diminishing returns from $6E$ onwards. Notice the $20E$ class—the maximum possible storage—termed the generalized data structure (GDS). GDS has the minimum number of record accesses.

Constant-time data schemes

Comparing with the GDS scheme, which stores all possible relations (hence requiring the maximum storage of $20E$) but possesses the least record accesses, we see that Δ is more efficient in time also. This occurs because Δ requires only 30 percent of storage, can hold a very large fraction of its total data space in main memory, and has fewer page faults. We will look at this in greater detail later.

Note that the Sparc workstation I used for experiments is not representative of all computers. Nonetheless, you can reach the same broad conclusions on any computer operating in a virtual memory environment.

Criteria for optimality

Time for a query's execution depends on the number of field comparisons, number of disk accesses, main memory access time, CPU processing speed, and disk access (mainly seek) time. The number of disk accesses depends on the number of records referenced.

Let p equal the ratio of the number of faults (termed page faults in the virtual memory environment and database faults in the database environment) and the total number of records referenced. In other words, p is the proportion of records requiring disk accesses.

Let f_n and r_n equal the number of field comparisons and record accesses, respectively.

Note that some records needing examination might already be present in main memory. The number of page faults is only $p \times r_n$ (each fault induces one disk access).

Let f_t and r_t equal the time for each field comparison (main memory access and CPU processing) and fault service time

Table 4. Storage class versus number of record accesses.

Storage		4E	6E	8E	10E	12E	14E	16E	20E
i		1 — 2	1 — 3	2 — 5	1 — 5	1 — 6	1 — 7	1 — 8	1 — 9
N	3	E	33	23	20	17	14	11	9
	6	E	51	35	29	23	17	11	9

Table 5. Experimental data (data space in Mbytes, access time in seconds for 20,000 iterations).

Mb	4.77	7.63	11.44	15.26	19.1	22.89	24.8
Time	2	1,417	2,234	2,584	2,757	2,860	2,913
p	0	0.33	0.53	0.63	0.69	0.72	0.74
x	1	0.62	0.42	0.31	0.25	0.21	0.19

(mainly disk seek time—see "Lagging I/O speeds" above), respectively.

A general formula is

$$T = f_n \times f_t + p \times r_n \times r_t \tag{4}$$

T equals the time for execution of the nine possible queries (see Table 2), one time for each.

We assume the cost equals the sum of all nine individual queries. Because this involves each query, it provides a better indicator of the overall performance of a typical CIM environment, where different users query a common database.

Note that if we assume all the data resides in main memory, then $p = 0$ and we have $T = f_n \times f_t$. This implicit assumption underlies most earlier attempts (such as Kalay's[7]) to enhance WE's access efficiency.

Record versus field access costs

Given a low fault rate, the greatest cost results from the field comparison, which Weiler[5] considered a measure of the speed. We want to find the range of p for which record access cost dominates field comparison cost. Among the constant-time data structures (listed in Table 3), Δ requires the maximum number of field comparisons. For Δ, I estimated the number of field comparisons to be 250 and the number of record accesses to be 51 (see Table 3). Hence, the worst case requires 250 main memory accesses and CPU comparisons. Taking $f_n = 250$ and $r_n = 50$ and conducting the experiments on a Sparc workstation, I estimated r_t and f_t to be 30 milliseconds and 3 microseconds, respectively. (Note that, although r_t ranged from 30 ms to 45 ms, to be on the safe side I took the lower value.) Substituting these into Equation 4 yields

$$T = 250 \times 3\ \mu\text{s} + p \times 50 \times 30\ \text{ms} \tag{5}$$

Thus, the ratio of field comparison cost and disk access cost is $0.0005/p$. We can conclude that if the page fault percentage is greater than 1 percent, the cost of field comparisons is less than 5 percent of disk access cost. If so, we can ignore it. The experimental data below substantiates this. As discussed above, the page fault rate depends on the amount of main memory and the size of the data.

The Sparc workstation used had a main memory of 8 megabytes (some of it occupied by the Unix system and hence unavailable to programs). I varied the data size from 4.77 Mbytes to 24.8 Mbytes, as shown in Table 5. Thus x, the ratio of the number of physical frames available (that is, the main memory available for the program) and the total number of virtual memory pages occupied by the data (that is, the data size) varied from 1 to 0.19. I measured the corresponding page faults and T and tabulated them in Table 5.

When the data structure occupied 4.77 Mbytes, all of the data resided in main memory. Hence, there was a negligible number of disk accesses ($p \cong 0$). Thus, the T value shown against $x = 1$ results solely from field comparisons.

However, when the data size increased to 7.63 Mbytes, main memory could hold only 62 percent of the data. This led to many disk accesses. Although only 33 percent page faults occurred, the access time rose by a factor of 700, as shown in Table 5. The difference in T for $x = 1$ and 0.62 results from the extra disk accesses. So, the determining factor is the number of record accesses for $p > 1$ percent. The experiments thus corroborate the statement following Equation 5.

Comparing Δ to GDS

The number of page faults decreases with the fraction of total space (mainly data space, since instruction space is negligible in our case) held in main memory. Other factors (such as page size or page replacement policy) influencing the page-fault rate are beyond the scope of this article. Thus, the number of disk accesses decreases with x, the proportion of data held in main memory. Since GDS and Δ require $20E$ and $6E$ storage respectively (see Table 4), the proportion for GDS, x_g, is $6E/20E = 0.3$ times that of x_Δ of Δ.

$$x_g = 0.3 x_\Delta \tag{6}$$

From Table 5 observe that as x decreases by a factor of 3 (for example, from 1 to 0.31), access time increases by a factor of 1,292. Hence, it is likely that GDS will have substantially higher faults than Δ (for the same main memory), resulting in higher virtual memory overhead. Let T_Δ and T_g refer to T in Δ and GDS and p_Δ and p_g refer to p in Δ and GDS.

As noted in the previous section, I estimated f_n for Δ to be 250 and accordingly inserted 250 in Equation 5. To show that the following discussion applies even when the actual value of f_n is 10 times the estimate, insert 10×250 in Equation 5, yielding

$$T_\Delta = 2{,}500 \times 3\ \mu\text{s} + p_\Delta \times 50 \times 30\ \text{ms}$$

Since GDS stores all information explicitly, it requires no field comparisons. From Table 4 it accesses 9 records (for simplicity, assume $r_n = 10$ in Equation 5). Thus

$$T_g = p_g \times 10 \times 30\ \text{ms}$$

Thus the condition for Δ to be more efficient than GDS is that

$$p_g > 5 \times p_\Delta + 0.025 \quad \textbf{(7)}$$

If $p(x)$ describes the fault-rate curve, then from Equation 6 we have $p_g = p(x_g) = p(0.3x_\Delta)$. We find the solution of the $p(x)$ curve (x refers to x_Δ) that satisfies $p_g = 5 \times p_\Delta + 0.025$ or, equivalently,

$$p(0.3x) = 5 \times p(x) + 0.025$$

to be

$$p(x) = \frac{0.025(x^{\ln 5/\ln 0.3} - 1)}{4} \quad \text{for } 0 < x < 1 \quad \textbf{(8)}$$

This curve represents the lower bound for the region, in which Δ is more efficient than GDS. As shown in Figure 8, the experimental curve lies above this curve, and GDS is inefficient compared to Δ. However, for certain intervals (such as $x_g > 1$) GDS is more efficient than Δ. (In the next section, we find the interval for x for which this is true.)

Note that if we assume f_n is 250, we have

$$p(x) = \frac{0.0025(x^{\ln 5/\ln 0.3} - 1)}{4} \quad \text{for } 0 < x < 1$$

which also leads to the same conclusion.

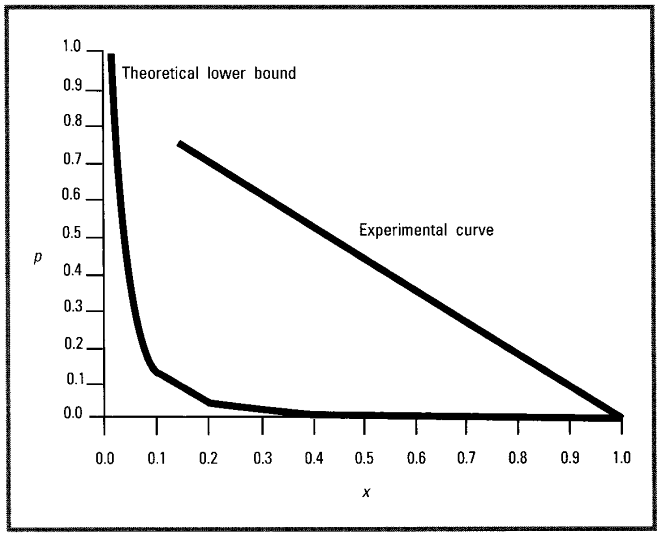


Figure 8. Memory versus page faults: x (ratio of main memory and data size) versus p (proportion of records requiring disk access).

Condition for memory size

Now let's explore the condition for memory size for which Δ is more efficient than GDS.

It is interesting to note that though the number of relations stored varies from 3 (Δ) to 9 (GDS), the total number of records stored is

$$V + E + F = 2E + 2 \cong 2E$$

which holds for SDS, WE, Δ, and GDS (but not for the VE, FE, and HE data structures). The number of records of Δ and GDS that main memory of size M can hold is $(M/6E) \times 2E = M/3$ for Δ and $(M/20E) \times 2E = M/10$ for GDS.

Assume that neither locality nor sequentiality is present.

$$p_\Delta = \frac{2E - \frac{M}{3}}{2E} = \begin{cases} 1 - \frac{M}{6E} & \text{if } 0 \le \frac{M}{6E} < 1 \\ 0 & \text{if } \frac{M}{6E} \ge 1 \end{cases}$$

$$p_g = \frac{2E - \frac{M}{10}}{2E} = \begin{cases} 1 - \frac{M}{20E} & \text{if } 0 \le \frac{M}{20E} < 1 \\ 0 & \text{if } \frac{M}{20E} \le 1 \end{cases}$$

Since $x_g = M/20E$, we have (let x_g be x)

$$p_g - 5 \times p_\Delta = \begin{cases} \frac{47x}{3} - 4 & \text{if } 0 < x < 0.3 \\ 1 - x & \text{else if } 0.3 \le x < 1 \\ 0 & \text{else if } x \ge 1 \end{cases}$$

The maximum $(p_g - 5 \times p_\Delta) = 0.7$ occurs at $x = 0.3$. From Equation 7,

$$p_g > (5 \times p_\Delta + 0.025) \text{ for } 0 < x < 0.3 \Leftrightarrow x > 0.257$$
$$p_g > (5 \times p_\Delta + 0.025) \text{ for } 0.3 \le x < 1 \Leftrightarrow 1 - x > 0.025 \Leftrightarrow x < 0.975$$

The condition for Δ to be more efficient than GDS is

$$0.257 < x < 0.975$$

which applies for the case with no locality or sequentiality. However, in the presence of locality or sequentiality, we expect p_Δ and p_g to be lower.

Comparing Δ to SDS

Proceeding as above, we find that Δ is more efficient than SDS for

$$0.474 < x_{SDS} < 0.999$$

Multiple entities

Next we explore why including extra topological and geometric entities is inefficient. We have three basic topological entities E, F, and V. As mentioned before, several additional entities resulted from attempts to extend and enhance the efficiency of WE. We also need to examine the effect of multiple geometries, since each of the three basic entities has an associated geometry. For polyhedra, we do not need to store the geometry for all three—we can derive the geometry of any two entities from the geometry of the third.

Multiple topology

We can extend data structures to faces with holes without resorting to the extra loop entity. Let's look at that process and also consider the half-edge record's effect in increasing record accesses.

Half-edge entity

We want to compare HE (created by exploding the edge entity) to WE with respect to storage and time. Excluding the loop entity, HE differs from WE in only two additional arcs: $S \rightarrow E$ and $E \rightarrow S$.

What effect does HE have on storage? The extra storage required because of bifurcation of the edge entity into two half edges is given by

$$E \rightarrow S + S \rightarrow E = 2E + 2E = 4E$$

Note that $|S| = 2E$ because each edge has two half edges.

What effect does HE have on access time? From Table 3, we see that HE has 34 extra record accesses over WE (58 when $N = 6$). We can therefore conclude that extra entities decrease main memory accesses and comparisons done by the CPU, but increase storage and record accesses substantially. Proceeding as we did in the section "Comparing Δ to GDS," a comparison of WE and HE reveals that WE is more efficient for $0 < x_{\mathrm{HE}} < 0.9999$. (This assumes storages of $13E$ and $9E$, f_n of 10 and 100, and r_n of 61 and 27 for HE and WE.)

Multiply connected faces

Now we need to discuss two important earlier attempts to extend data schemes to represent holes. We will also consider an alternative scheme, which avoids using extra entities like the loop, yet exhibits storage and time efficiency.

1. Bridge-edge representation (BE). Holes are implicitly represented by fictitious edges (termed bridge edges) connecting loops of multiply connected faces. Because the fictitious edges occur twice in visiting a face, determining the next edge when given only an edge results in ambiguity. The same holds true if we use vertices to represent holes. Hence, the bridge edge[12] uses an edge and the next vertex alternately. The scheme involves extra storage because of the need to represent both edge and vertex in a face list. The extra storage for L holes is

$$2E + 2L \times 2 = 2E + 4L \qquad \textbf{(9)}$$

Add to this the storage due to the fictitious edges (one per loop), which is $8L$ in the case of WE.

2. Loops. We can explicitly represent the holes (see, for example, Weiler[5]) by an additional entity called a loop. A face is a list of loops. Let L_i refer to the number of loops in the ith face and L to the total number of loops for the whole object. Note that L excludes the outer loop. In the case of SDS, the extra storage results because of $L \rightarrow F$ and $F \rightarrow L$ and is given by

$$\sum_{F} 2(L_i + 1) = 2L + 2F \qquad \textbf{(10)}$$

3. Alternative scheme. The alternative scheme to represent faces with holes represents holes by a string of vertices, the sense of the string being opposite to the outer contour. Each hole is connected to the outer boundary by two vertices, one of which has a negative sign to symbolize that it is a fictitious edge. To get edges, we take two consecutive vertices. If the first vertex is positive, then it constitutes a valid edge; otherwise, it is an artificial edge. Also, the members (denoted by the vertex IDs) of the vertex list are unique because, although each number in the vertex string can appear twice, they appear with opposite signs. This alternative scheme combines the best features of both the BE and loop schemes in having unique face representation and requiring the least storage.

Now let's compare the schemes with respect to storage. The extra storage incurred in Scheme 3 is

$$\sum_{F}(L_i + 1) = \begin{cases} L + F & \text{if } L > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \textbf{(11)}$$

Comparing Equation 11 with Equation 9, we see there is no penalty if $L = 0$. But in the bridge-edge and the loop schemes, even in the absence of holes, the extra storage cost is $2E$ and $2F$, respectively.

Comparing the schemes with respect to time, note that the number of record accesses does not increase in Scheme 3. But the CPU must perform more comparisons because of the need to test a vertex's sign in the list of vertices for faces. However, this extra test for the sign is required only for processing $F \rightarrow E$, not for others like $E \rightarrow V$.

What about extra record accesses in the BE and loop schemes? The BE extension involves no extra record accesses. Let $F_L = 1 +$ number of holes in face F. The SDS with loop scheme involves an extra cost of F_L each for $F \rightarrow V$, $V \rightarrow F$, and $F \rightarrow E$, 2 for $E \rightarrow F$, and $2F_L$ for $F \rightarrow F$, for a total of $5F_L + 2$. Thus, even when a face has no holes, the extra record accesses are $5 \times 1 + 2 = 7$.

We can therefore conclude that the alternative scheme is more efficient for extension to holes. Note we cannot use this scheme in the case of SDS.

Multiple geometry

Assume that we have polyhedral models only. The criteria is the access time of the geometry of all three entities. Let's investigate two schemes: (1) storing only the vertex coordinates and (2) storing all (for example, the WE implementation in Baumgart[3]) the geometry—the coordinates for the vertex and the line and plane equations for the edge and face.

Storage estimation Schemes 1 and 2 require $3V$ and $3V + 4E + 3F \cong 7E$, respectively. Thus, storage of all geometry requires approximately four times the storage required for the vertex coordinates only.

Now consider time estimation. In Scheme 1, computing a line equation requires retrieving one geometric record for each of the end points. Computing a face plane requires access to three geometric records to retrieve coordinates of three vertices. For simplicity, we do not consider the topological record accesses. Substituting 6 (1 for the vertex geometry) for r_n and 100 for f_n in Equation 4,

$$T_1 = 100 \times 3\,\mu\text{s} + p_1 \times 6 \times 30\,\text{ms}$$

where

$$p_1 = \frac{V - \frac{M}{3}}{V} \cong 1 - \frac{M}{2E}$$

In Scheme 2, f_n is zero, since no computation is involved, and $r_n = 3$ for accessing three geometric entities. Substituting these, we have

$$T_2 = p_2 \times 3 \times 30\,\text{ms}$$

where

$$p_2 = \frac{2E - \frac{M}{3.5}}{2E} = 1 - \frac{M}{7E} = 1 - x_2$$

Thus for $0.167 < x_2 < 0.997$, storing only the vertex coordinates is also more efficient in time. Note that considering topological record accesses (in the case of WE, one for an edge to get its end points and seven for a face to get its vertex identifiers; see Table 2) slightly tilts the scale in favor of Scheme 2. Taking 1,000 instead of 100 for f_n, the range is rather narrow—$0.172 < x_2 < 0.97$.

Implementation methods

The other anomaly relates to theoretical estimates of storage. (For example, SDS requires more storage than WE in practice.) The estimates are misleading because you cannot implement some data structures with arrays. To determine their effect on storage, I surveyed several different methods of implementing the data structures.

Linked lists and arrays

Although SDS takes $8E$, a practical implementation requires $10E$. Because of the variable nature of N_F and N_V, an array implementation is not possible for SDS. However, we can implement WE with arrays because $N_{VE} = N_{FE} = 2$, $N_{EE} = 4$, and $V \rightarrow E$ and $F \rightarrow E$ store only 1 edge each. Array implementation of WE requires $9E$. But the SDS requires linked (we assume singly linked) list implementation, which increases the theoretical storage by $4E$, calculated as follows:

$$\text{Storage for SDS} = F \rightarrow E + E \rightarrow F + E \rightarrow V + V \rightarrow E$$
$$= 2E + 2E + 2E + 2E + 2E + 2E = 12E$$

$$\text{Storage for } \Delta = F \rightarrow V + E \rightarrow F + V \rightarrow E$$
$$= 2E + 2E + 2E + 2E + 2E = 10E$$

The actual storage for SDS tallies with De Floriani's and Falcidieno's implementation.[4] Unlike the array implementation, where the data will be contiguous, linked lists might cause the data to be noncontiguous.

Winged triangle[1] is a recently published data structure permitting array implementation. However, as discussed elsewhere,[2] the winged-triangle representation does not belong to the class of constant-time data structures. Hence, I do not use it for comparative studies here.

Dynamic allocation of memory

As the data is read, we can dynamically (for example, using the C language) increase the storage for variable relations like $F \rightarrow V$ (see Figure 6). The increase is the storage for the pointers that point to pointers to each face and vertex. The storage increase for the pointers is $F + V \cong E$. Assuming that a pointer occupies the same memory as an integer, the increase is E, for a total of $7E$.

We must also store the number of vertices in a face as the first element in the face array. Similarly, we must store the number of edges adjacent to a vertex as the first element in the vertex array. These changes will make storage of Δ $8E$ and SDS $10E$.

Relational implementation

Kalay[13] argued for an auxiliary relational data structure for nonmanipulative operations. (For manipulative operations, you modify—create, delete, and so forth—CAD entities using a variety of techniques. With nonmanipulative operations, you mainly retrieve data for display or interrogration purposes.) I limit my discussion to space and time estimation for pure relational implementation of the three basic entities. Since a relational database requires tuples of constant length (called "degree of the relation" in relational parlance), we need two relations or tables to store each of the variable relations, such as $V \rightarrow E$.

If the ordering of the edges around a vertex is not important, one table suffices. Also, if a system-defined ordering (clockwise order of the edges) of the tuples can be maintained with a suitable operator for retrieving the clockwise neighbor from a given edge for a given vertex, then one table with two attributes suffices. This table (Scheme 1 below) will have two attributes, vertex_id and edge_id, the key being the concatenation of the two attributes.

The two schemes for $V \rightarrow E$ are

```
Scheme 1
  Relation Vertex_edge
    (vertex_id: integer,
    edge_id: integer)

Scheme 2
  Relation Vertex_1st_edge
    (vertex_id : integer,
    1st_edge_id : integer)
  Relation Vertex_next_edge
    (vertex_id : integer,
    edge_id : integer,
    next_edge_id : integer)
```

Scheme 1 requires $4E$ storage, while Scheme 2 requires $6E + 2V$. Similarly, $F \rightarrow V$ requires $4E$ or $6E + 2F$, depending on whether we employ Scheme 1 or 2. For the constant relations $E \rightarrow V$ and $E \rightarrow F$, one table suffices (each requires $2E$ storage). Thus, relational implementation of Δ requires $10E$ under Scheme 1 and $14E + 2F + 2V \cong 16E$ (since $F + V = E + 2$) under Scheme 2. The corresponding figures for SDS are $12E$ and $16E + 2F + 2V \cong 18E$. Thus, relational implementation involves a significant increase in storage. Queries on relational databases require frequent join operations, which require a great deal of data retrieval from the permanent store and a consequent increase in page faults and time.

Conclusions

We have considered several data schemes that exhibit constant access time. They reveal space and time anomalies in a virtual memory environment. A study of the data structure that stores all possible relations reveals a startling phenomenon: more storage does not necessarily mean less time because of the virtual memory overhead. On the other hand, the new data structure I proposed has minimum storage among all constant-time data structures, yet requires less access time.

This situation calls for a rethinking of the data structure space versus time phenomenon. An immediate consequence is that many popular data structures that are modifications of the winged-edge data structure incur extra storage but do not actually meet their avowed objective of reducing access time. This argues for compact data structures, not only for less storage but also for lower access time. We can conclude that compactness has a beneficial side-effect—it reduces interrogation time by having fewer page faults and by clustering the related data. In a nutshell, "Small is beautiful."

Directions for further research

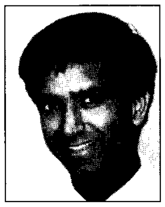
Reported empirical studies (for example, Kearns and DeFazio[10]) involve non-CAD databases. Even then, there seems to be no convergence of results as to the presence of locality or sequentiality in database reference behavior. All that can be said is that clearly discernible patterns are absent, in contrast to the strong locality exhibited by program memory references. Further research is required to determine the precise behavior of CAD database references. Also, we need to ascertain whether the nature of the nine fundamental queries and the data structure reveal the presence of sequentiality or locality and devise strategies to exploit them. ❑

Acknowledgments

I acknowledge the support of the U.K. Science and Engineering Research Council. I would like to thank Denis Chamberlain and Tim Ellis for support. Thanks also to John Snell and the referees for many valuable suggestions.

References

1. A. Paoluzzi, M. Ramella, and A. Santarelli, "Boolean Algebra over Linear Polyhedra," *Computer-Aided Design*, Vol. 21, No. 8, Oct. 1989, pp. 474-484.
2. S.R. Ala, "Design Methodology of Boundary Data Structures," *Int'l J. Computational Geometry*, Special Issue on Solid Modeling, Vol. 1, No. 3, Sept. 1991, pp. 207-226. Also in *Proc. ACM Symposium on Solid Modeling and CAD/CAM Applications*, June 1991, pp. 13-23.
3. B.G. Baumgart, "A Polyhedron Representation for Computer Vision," *Proc. AFIPS Nat'l Computer Conf.*, 1975, pp. 589-596.
4. L. De Floriani and B. Falcidieno, "A Hierarchical Boundary Model for Solid Object Representation," *ACM Trans. Graphics*, Vol. 7, No. 1, Jan. 1988, pp. 42-60.
5. K.J. Weiler, "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments," *IEEE CG&A*, Vol. 5, No. 1, Jan. 1985, pp. 21-40.
6. P.R. Wilson, "Data Transfer and Solid Modeling," in *Geometric Modeling for CAD Applications*, M.J. Wozny, H.W. McLaughlin, and J.L. Encarnacao, eds., Elsevier Science, 1988, pp. 217-249.
7. Y.E. Kalay, "The Hybrid Edge: A Topological Data Structure for Vertically Integrated Geometric Modeling," *Computer-Aided Design*, Vol. 21, No. 3, April 1989, pp. 130-140.
8. M. Mantyla, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Md., 1988.
9. T.C. Woo, "A Combinatorial Analysis of Boundary Data Structure Schemata," *IEEE CG&A*, Vol. 5, No. 3, Mar. 1985, pp. 19-27.
10. J.P. Kearns and S. DeFazio, "Diversity in Database Reference Behavior," *Performance Evaluation Review*, Vol. 15, No. 1, May 1989, pp. 11-19.
11. T.C. Woo and J.D. Wolter, "A Constant Expected Time, Linear Storage Data Structure for Representing Three-Dimensional Objects," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 14, No. 3, May/June 1984, pp. 510-515.
12. F. Yamaguchi and T. Tokieda, "Bridge Edge and Triangulation Approach in Solid Modeling," in *Frontiers in Computer Graphics*, T.L. Kunii, ed., Springer Verlag, Berlin, 1985.
13. Y.E. Kalay, "A Relational Database for Nonmanipulative Representation of Solid Objects," *Computer-Aided Design*, Vol. 15, No. 5, Sept. 1983, pp. 271-276.



Seshagiri Rao Ala is a researcher in the Construction Robotics Unit of City University, London. From 1985 to 1989 he served the Indian Railways (Rail Coach Factory) as works manager for computerization and automation, including robotics. His current research interests are graphics, geometric modeling, computer vision, and construction robotics.

Ala received his MTech in systems and management from the Indian Institute of Technology, Delhi.

Readers can reach Ala at Construction Robotics Unit, Machine Vision Group, School of Engineering, City University, London EC1V OHB, U.K.