

# A Database Architecture and Joint Query Language to Support Efficient After Action Review

*Sanjeeb Nanda*

*Matt Archer*

SDS International Inc., Advanced Technologies Division  
3403 Technological Avenue, Suite 7  
Orlando, FL 32817  
407-282-4432

[snanda@sdslink.com](mailto:snanda@sdslink.com), [marcher@sdslink.com](mailto:marcher@sdslink.com)

*Dr. Joseph Weeks*

Air Force Research Laboratory  
Human Effectiveness Directorate  
Warfighter Readiness Research Division  
6030 South Kent Street  
Mesa, AZ 85212-6061  
480-988-6561 ext. 249

[joseph.weeks@mesa.afmc.af.mil](mailto:joseph.weeks@mesa.afmc.af.mil)

Keywords:

Hashing, events, records, database, metadata, MD2, MD4, MD5 and SHA.

**ABSTRACT:** *Contemporary tools for After-Action-Review (AAR) of large, recorded exercises do not offer efficient interfaces for locating and replaying events of interest during post-mission briefing. They typically provide methods for creating timelines that surround a set of events or simply provide bookmarks at periodic intervals of time, leaving it to the user to locate events within a timeline using sequential playback. Such mechanisms, however, require the timeline associated with an event that may be searched for during an AAR to be memorized or manually recorded, and the event located visually. Such mechanisms are inefficient when applied to massive recordings of distributed exercises that are rich in content. This paper proposes a solution to this problem using a novel architecture for a multi-attribute indexed database supporting a query language with standard logical operators. This JQL (Joint Query Language) provides an indexing architecture that is simulation-protocol-independent, scalable, has low space consumption and facilitates direct access to any specified event at a random location in the database in  $O(1)$  time-complexity. Using the query language, a user may extract from a recorded exercise any segment needed for AAR delimited by two events.*

## 1. Introduction

After-action-reviews are critical elements of effective applications of computer-driven, training simulations. Delivery of feedback very soon after the completion of an exercise with special emphasis on tasks designated as training objectives are cited by users as the key to performance improvement [2]. Simulation exercise loggers are capable of recording vast amounts of information. The means to quick access and retrieval of simulation exercise segments relevant to training objectives remains a highly challenging technology obstacle. There is a compelling need for recording simulation exercises to databases that facilitate fast direct

access to any randomly specified event. A significant factor governing the speed of such access is the time taken to locate the data corresponding to the desired events in secondary storage and for it to be swapped into system memory [1]. Note that, modern file systems such as NTFS map those parts of a file that are required by the subscribing applications into system memory as and when needed [3]. This ensures that a few egregiously large files are not swapped in from secondary storage in their entirety to hog system memory. Furthermore, the user has little control over when and how much of a file is swapped in or out, to or from system memory by the file system. Though it is a deterministic process, the embodying file system driver does not furnish any

interface for its control to the user. Hence, to reduce the frequency of file swapping, the size of the data structures used for constructing the database for the recorded exercise and its associated metadata should be space-efficient.

Once the format for representing the values of the various attributes comprising the records in an exercise have been defined and set, the amount of data necessary to record the exercise is invariant. Then, the challenge is to design the data structures for metadata that minimizes space consumption while fulfilling the following requirements.

- Req. (i) A record may contain a multiple number of keys.
- Req. (ii) The disk address of a record should be obtainable with minimum latency.

One of the data structures that may be considered as a candidate for organizing the metadata is the B tree. However, the use of B trees ordered by the values of a specific key, with embedded addresses of on-disk records is not beneficial when the number of attributes comprising the key is large. The following argument illustrates the reason. Consider a simulation involving  $T$  entities with a dead reckoning frequency of  $f$  Hz. that is being recorded to a database using  $k$  attributes in each record as a key. For instance the attributes of *entity id* and *time* may be used to form a key for each record. This simulation will generate  $N = fT$  records per second corresponding to the entities. Assuming that on-disk addresses are stored using 64 bits (8 bytes),  $8fT$  bytes of space will be used for storing on-disk addresses. Note that, this does not include the additional space needed to store the trees themselves. Therefore, assuming that the  $(N - 1)$  node pointers in each tree with  $N$  nodes are 4 bytes each, and each key is at least  $4k$  bytes in length (assuming that each attribute is at least 4 bytes), the amount of space used by the tree with  $N$  nodes equals  $4(kN + N - 1)$  bytes. Thus, the total amount of space used by the search tree for storing their data structures and the on-disk addresses of records in the exercise per second exceeds  $8fT + 4(kN + N - 1) = fT(4k + 12) - 4$  bytes.

For example, a simulation involving 100 entities, with the attributes of *entity id* and *time* in each record constituting a key and a dead reckoning frequency of 60 Hz, generates 6000 records per second and consumes over  $119.9 \times 10^3$  bytes for storing the corresponding on-disk addresses and the data structures for the tree. This is not a very large value. However, the problem is that, the choice of *entity id* and *time* as the attributes for a key is less than ideal, for the reason that it does not facilitate searches using any other useful attribute in a record. For instance, such a key will not facilitate the search for records that match given values for the *entity id* and the location of that entity. It is then very clear that a key should be comprised of all

attributes that are likely to be specified in a query. For instance, if we represent the position of an entity by the triplet of *latitude*, *longitude* and *altitude* and include those attributes to the existing ones to define a key, then the value of  $k$  would increase from 2 to 5. Then the amount of space in the preceding example used for storing on-disk addresses and the data structures for the tree would increase from  $119.9 \times 10^3$  bytes to  $191.9 \times 10^3$  bytes. In practice, such an approach will require values of  $k$  that are quite large. For instance, one may query a record where the state of the entity has a specific value, say where it is burning. It is quite evident from the aforesaid arguments that the use of B trees can result in ungainly space consumption. Furthermore, fulfilling Req. (ii) may be challenging as well for the following reason. The time taken to obtain a desired record in such trees requires less than  $\log_{M/2} N$  accesses, where  $N$  is the number of nodes in the B tree, and  $M$  is the maximum order of a node [5]. Although this is not a large value, it is nevertheless dependent on the value of  $N$ . An increase in the number of entities participating in the exercise, or the duration of the exercise, or the average number of attributes per record will increase this value. But, the most significant drawback that is sometimes ignored is the sizeable processing overhead that is incurred at each node, where key comparisons are made. Again, the larger the key, the greater the number of attributes

In the preceding discussion we showed that conventional techniques for indexing metadata and storing the associated data structures in memory is inefficient from the perspectives of space utilization and speed. Our scheme overcomes both these drawbacks by indexing metadata for exercises such that, no record contains a redundant on-disk address, and furthermore, a record corresponding to any desired attribute can be accessed in a constant number of steps. Before we describe our scheme, we first introduce some basic ideas.

## 2. Hashing

Hashing is used in a wide variety of applications. For instance, hash functions are frequently used for transforming keys into disk or memory addresses where the records corresponding to those keys are stored. However, a significant challenge in designing a desirable hash function is to reduce the collisions it produces for unique keys, i.e., where unique keys are transformed to the same address. The hash functions MD2, MD4 and MD5 proposed by Ronald Rivest and SHA-1 proposed by the NSA accomplish this objective with a very high degree of reliability. While MD2, MD4 and MD5 generate 128-bit signatures, SHA-1 generates a 160-bit signature for inputs of practically any arbitrary length. Fig. 2-1 illustrates the pseudo-code for the SHA-1

algorithm defined by the Secure Hash Standard [7]. Each aforementioned function theoretically guarantees the probability of a collision between the hash values of two randomly selected keys to be  $1/2^L$ , where  $L$  is the number of bits in their respective signatures. Since  $L \geq 128$  for each of the aforementioned functions it is easy to see that the probability of such collisions are virtually impossible.

This property can be applied to generate a signature for an entire record that is, for all practical purposes, unique to that record. Thus the signature produced by hashing a complete record can be used as the key for that complete record. Using this premise we formulate the architecture of the data structure necessary to expedite access to metadata.

```

// Initialize the variables h0 through h4
h0 ← 0x67452301; h1 ← 0xEFCDAB89; h2 ← 0x98BADCFE; h3 ← 0x10325476; h4 ← 0xC3D2E1F0;

// Pre-processing the input message
Append "1" bit to message; Append "0" bits until message length ≡ 448 (mod 512);
Append length of message as 64-bit big-endian integer to message;

// Process the message in successive 512-bit chunks
Break message into 512-bit chunks;

for each chunk do
{
    Break chunk into sixteen 32-bit big-endian words w(i), 0 ≤ i ≤ 15

    // Extend the sixteen 32-bit words into eighty 32-bit words
    for i ← 16 to 79 do { w(i) ← (w(i-3) ^ w(i-8) ^ w(i-14) ^ w(i-16)) leftrotate 1; }

    // Initialize hash value for this chunk
    a ← h0;    b ← h1;    c ← h2;    d ← h3;    e ← h4;

    for i ← 0 to 79 do
    {
        if (i ≥ 0) and (i ≤ 19) then
        { f ← (b & c) | ((~b) & d); k ← 0x5A827999; }
        else
        if (i ≥ 20) and (i ≤ 39) then
        { f ← b ^ c ^ d; k ← 0x6ED9EBA1; }
        else
        if (i ≥ 40) and (i ≤ 59) then
        { f ← (b & c) | (b & d) | (c & d); k ← 0x8F1BBCDC; }
        else
        if (i ≥ 60) and (i ≤ 79) then
        { f ← b ^ c ^ d; k ← 0xCA62C1D6; }

        temp ← (a leftrotate 5) + f + e + k + w(i);
        e ← d; d ← c; c ← b leftrotate 30; b ← a; a ← temp;
    }

    // Add this chunk's hash to result so far
    h0 ← h0 + a;    h1 ← h1 + b;    h2 ← h2 + c;    h3 ← h3 + d;    h4 ← h4 + e;
}

hash = h0 append h1 append h2 append h3 append h4;

```

Fig. 2-1. The pseudo-code for SHA-1 that generates a 160-bit signature, with the probability of a collision between the signatures of two arbitrary inputs produced by the algorithm being  $1/2^{160}$ .

### 3. In-Memory Architecture of the Solution

We represent each event i.e., the composite values for a given set of attributes, with a unique hashed value. A number of hash functions exist that can be used for this purpose. As previously discussed, some of the options include MD2, MD4, MD5 and SHA. Using any of these functions we can represent each event concisely and uniquely as a  $b$ -byte hash value, where  $b \leq 20$  bytes. This hash value will be used to derive the index into the hash table where the node comprised of that hash value and the 8-byte on-disk address of the corresponding event is stored. However, by the pigeon principle [4] collisions in a hash table of finite length are inevitable when the number of nodes exceeds the number of indices in the table. Hence nodes mapped to the same index in a hash table may be linked together using a unidirectional linked

list. Each such link to a node is a 4-byte pointer in system memory. Thus the amount of space consumed in a hash table for creating a node and its link is at most  $(20 + 8 + 4) = 32$  bytes.

Using the aforementioned scheme, an exercise conducted with  $T$  entities and a dead reckoning frequency of  $f$  Hz using hash-based event representations consumes at most  $(20 + 8 + 4)fT = 32fT$  bytes per second. For example, a simulation with 100 entities consumes  $192 \times 10^3$  bytes of space per second. We observe that, the space consumed per second for storing metadata using our hashing scheme i.e.,  $32fT$  bytes (which is independent of  $r$ ) is significantly better than that consumed by standard tree-based indexing i.e.,  $4r(4fT - 1)$  bytes for large value of  $r$ .

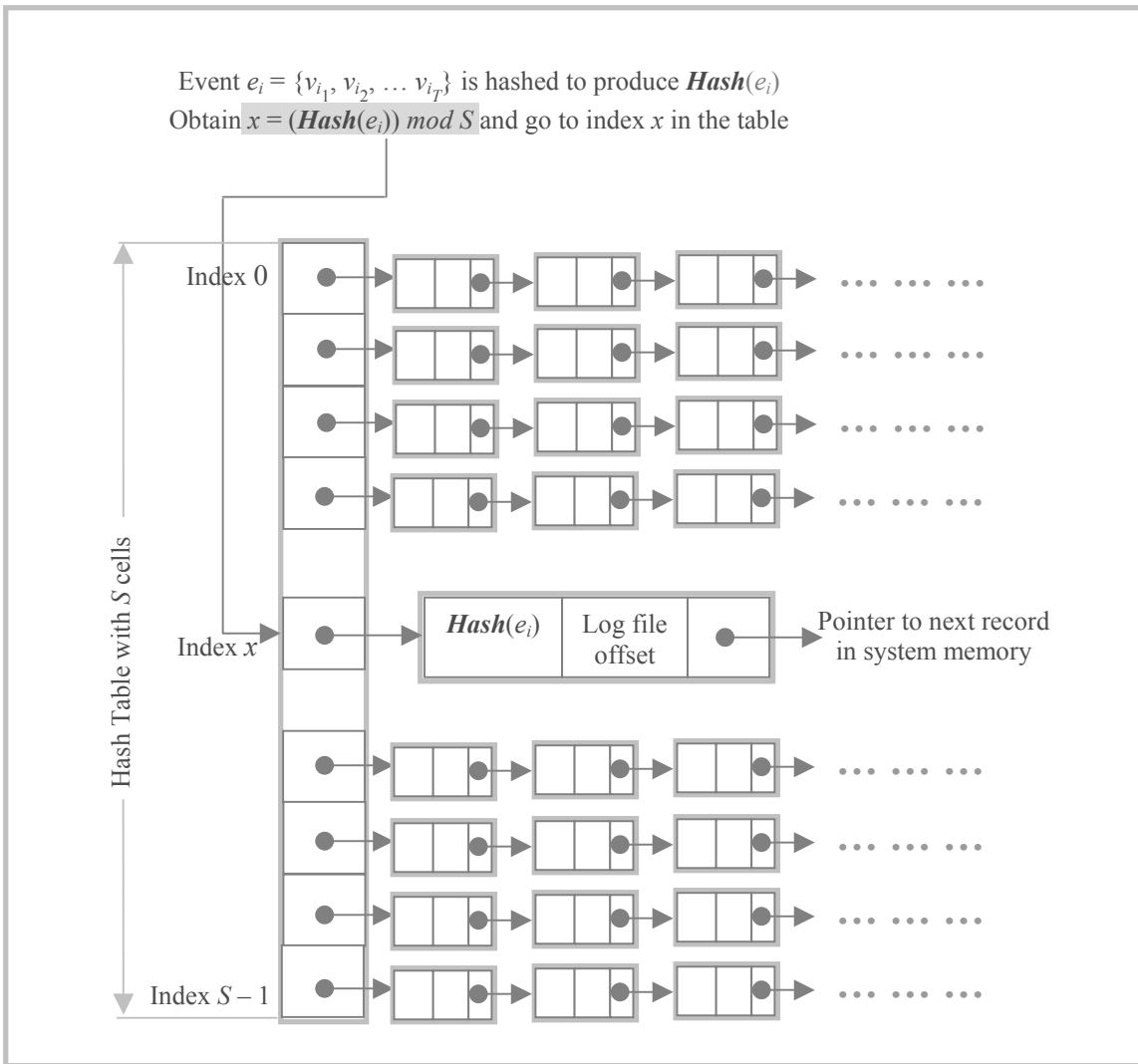


Fig. 3-2. Design of the hash table and the records representing events in an exercise.

Fig 3-2 displays a hash table with each cell containing a linked list of nodes, each representing an event, as previously described. Each node is comprised of three components – the hash value corresponding to an event in the exercise, the offset in the log file where the event should one exist (else it is null). As described in the figure, the linked list into which a record is placed is determined by taking the hash value for a record and obtaining the remainder  $r = (\text{hash}) \bmod S$ , where  $S$  is the size of the hash table. This remainder  $r$  equals the index of the table at which the desired linked list is anchored. An immediate implication of the design illustrated in Fig. 3-2 is that, the larger the size of the hash table i.e., the value  $S$ , the lesser the number of collisions and therefore the shorter the average length of the linked list at an index of the table. Ideally, the average length of a linked list should be small such that, the time spent in sequentially searching for a desired record is minimized. In any case,  $S$  should be proportionate to the number of records generated in an exercise such that, the average number of nodes at an index in the hash table is independent of the number of records generated by the exercise. In that case, the time spent in sequentially searching for a desired record in the hash table is  $O(1)$ . For instance,  $S$  may be chosen to be the total number of records generated during the duration of an exercise. Since the number of entities participating in the exercise and the dead-reckoning frequency are known, we can easily determine this value. Let us now state the definitions necessary to formally describe our solution and the manner in which hash signatures are derived for events.

#### 4. Calculating Hash Signatures and Accessing Offsets for Desired Events

We define an *attribute* as a logically unique type with a corresponding *domain* of values that can be assigned to that type. For instance, we may define *vehicle type* as an *attribute* with the corresponding *domain* being the set of values {M1A1, Ah64, F16, F15, T72, Mi24, Mig29, Su27}. Similarly, we may define *entity id* as an *attribute* with the domain for it being the set of all unsigned integers in the range 0 – 0xFFFFFFFF. Let  $A = \{a_1, a_2, \dots, a_M\}$  be a set of attributes and let  $D = \{\text{domain}(a_1), \text{domain}(a_2), \dots, \text{domain}(a_M)\}$  be the set of domains for  $a_1, a_2, \dots, a_M$  respectively. We may then represent an exercise as a sequence of events  $e_1, e_2, \dots, e_N$ , where  $e_i$  is an instance of a subset of  $D$ ,  $1 \leq i \leq N$ . That is,  $e_i = \{v_{i_1}, v_{i_2}, \dots, v_{i_R}\}$  is a set of  $R$  values where  $1 \leq i_1, i_2, \dots, i_R \leq M$ , and each value  $v_{i_k}$  is in  $\text{domain}(a_{i_k})$ ,  $1 \leq i_k \leq M$ . Now, we represent each event by a signature that is derived by hashing the result of concatenating all the *indexed attributes* of that event.

An interesting property of hash functions is that, they are unidirectional transformations. That is, given a hash function  $h$  and an input  $u$  we cannot determine the input  $u$  from the hash value  $h(u)$ . Let  $u = u_1, u_2, \dots, u_x$  and  $v = v_1, v_2, \dots, v_y$  be two sequences of attributes that define a pair of criteria. Then, if  $h(u) = \text{Hash}(u_1 u_2 \dots u_x)$  and  $h(v) = \text{Hash}(v_1 v_2 \dots v_y)$ , we can say that,  $h(u) = h(v)$  if and only if  $u_1 u_2 \dots u_x = v_1 v_2 \dots v_y$ , where  $u_1 u_2 \dots u_x$  and  $v_1 v_2 \dots v_y$  are obtained by concatenating the corresponding sequences of attributes  $u_1, u_2, \dots, u_x$  and  $v_1, v_2, \dots, v_y$  respectively. This property of hashing functions plays a large role in determining the manner in which an *Hash* value is derived and stored for an event  $e_i = \{v_{i_1}, v_{i_2}, \dots, v_{i_T}\}$ , for it affects the feasibility and complexity of the operations required to query events that meet a given criteria. We can break up our arguments into three cases. For the purpose of consistency, we may assume that each recorded event is comprised of the same number of attributes.

- (a) The set of attributes in the search criteria equals the set of attributes used to derive the hash value for an event.
- (b) The set of attributes in the search criteria is a subset of the attributes used to derive the hash value for an event.
- (c) The set of attributes in the search criteria is a superset of the attributes used to derive the hash value for an event.

##### Case (a)

If the number of attributes  $x$  in a search criteria  $u = u_1, u_2, \dots, u_x$  is the same as the number of attributes  $y$  in the events being searched, then it is easy to see that an event satisfying the search criteria must have a hash value that equals  $h(u) = \text{Hash}(u_1 u_2 \dots u_x)$ . Then, the occurrence of  $u$  in the exercise is determined by comparing  $h(u)$  against the hash values corresponding to the various events of the exercise. If any event has a hash value of  $h(u)$ , then it must be the event  $u$ .

##### Case (b)

In this case, we want to determine if an event with given values for the attributes  $v_1, v_2, \dots, v_x$  occurs in an exercise where the hash values corresponding to the events are obtained by concatenating the values of  $y$  attributes, with  $y > x$ . A query for this purpose must therefore append all possible permutations of values for the attributes  $v_{x+1}, v_{x+2}, \dots, v_y$  to  $v_1, v_2, \dots, v_x$  and compare the value of  $\text{Hash}(v_1 v_2 \dots v_y)$  with the hash values stored for that exercise, until a suitable match is

found. Of course,  $v_{x+1}, v_{x+2}, \dots, v_y$  must be contextually meaningful with respect to  $v_1, v_2, \dots, v_x$ .

For example, consider a query to determine the timeline of the event where a tank is burning. In this case the query has two defined attributes, the *entity type* and the *entity state*. Thus,  $x = 2$ . Then the event can be defined by the set of attributes  $\{v_1, v_2\}$ , where  $v_1$  is the attribute *entity type* having the value **tank**, and  $v_2$  is the attribute *entity state* having the value **firing**. Also, let us suppose that the keys for the events in the exercise are derived by hashing sets of attribute values of the form  $\{\textit{entity type}, \textit{entity id}, \textit{entity state}\}$ . Then, we should create hash values with all potential ids of entities as the value of the second attribute with **tank** and **firing** as the values for the first and third attributes respectively. Such comparisons can generate an overwhelming computational load unless the ids of entities are restricted to a specific and well-known set of values. Fortunately, the number of entities in an exercise is not very large – typically well under a 1000 – and their ids are well known. In reality, the problem is far more tractable since we need not compare using the ids of all entities. We can restrict the number of ids to those that correspond to the given value of an *entity type* attribute. For instance, in our example the ids of all the tanks are known and therefore, the number of candidate sets of attribute values of the form  $\{\textit{entity type}, \textit{entity id}, \textit{entity state}\}$  that have to be investigated is restricted to the permutations of the ids of tanks only.

From the aforesaid arguments, the larger the value of  $(y - x)$ , the greater the number of permutations that have to be investigated to find a prospective match for a given input. However, in practice, the specificity of queries can help to significantly reduce the computational burden. For example, a query that specifies the *entity id* as opposed to the *entity type* in addition to the *entity state* is extremely easy to process. The reason is, the value of *entity type* is known from a given value of *entity id*, and therefore it can be easily used to derive the unique set of attribute values of the form  $\{\textit{entity type}, \textit{entity id}, \textit{entity state}\}$ .

#### Case (c)

In this case, we want to determine if an event with given values for the attributes  $v_1, v_2, \dots, v_x$  occurs in an exercise where the hash values corresponding to the events are obtained by concatenating the values of  $y$  attributes, with  $y < x$ . A query for this purpose must then investigate all permutations of values for the attributes  $v_1, v_2, \dots, v_x$  with  $y$  attributes and determine if there exist  $c$  combinations,  $(x - y) + 1 \leq c \leq \binom{x}{y}$  such that, the hash values of those  $c$  combinations match  $c$

hash values in the exercise and the number of distinct attributes in the union of those  $c$  combinations is greater than or equal to  $x$ . Note that,  $\binom{x}{y} = x!/(y!(x - y)!)$ . It is clear that the complexity of such an operation is undesirably large if  $(x - y)$  and  $y$  are both large. This determines that multiple records, each matching a subset of the search criteria agree on their timelines.

As we shall show later, the time stamp associated with an event should not be kept as part of the record containing the *Hash* value that is resident in system memory. Hence, when a specific event is matched with a corresponding hashed value in memory, its time stamp is obtained from secondary storage. Therefore, when we search for  $c$  combinations, we must make sure that those combinations have time stamps that are nearly identical, i.e., they occur within a span of time that is small enough to be considered instantaneous. This fact exacerbates the computational complexity of searching for an event that is a subset of the search criteria.

Thus, to minimize the computational complexity of searches, the hashing scheme must be constructed adhering to the following general rules.

- The hash values for events in an exercise are derived from a small number of attributes – the number of attributes being a uniform value of  $y$ .
- The number of attributes admissible in a query  $x$ , is chosen such that  $|y - x|$  is minimized, where  $y$  is the number of attributes used for hash values in the in-memory data structure.

From case (b) and case (c) we observe that hashing on too many or too few attributes to create the search keys for the recorded database can increase the complexity of the searches. Hence, one of challenges in this scheme is deciding on what attributes to use in deriving hash signatures. To do this, we take a closer look at motivation of queries and the nature of their specification.

Queries are generally specified to extract timelines of an event based on a number of criteria, excluding time itself. Hence, the attribute of time itself should not be used for creating hash-based signatures. As noted earlier, the *entity id* attribute should be used for hashing, since it is easily restricted when an entity type is specified. Furthermore, the entity id is often explicitly specified too.

The location of any entity generally serves as a useful point of reference for specifying queries. However, queries by humans usually offer a coarse approximation for the location of an entity when requesting the timeline of an event. Furthermore, to ensure that the number of

admissible values for the location of an entity is not very large, we should ideally not hash the attribute of position itself, where the values of latitude and longitude are precisely defined in degree, minutes and seconds. Rather hashing should be performed on a coarser definition of the position. For example if the expanse of the database spans only one geo-cell, then hashing may be performed on the position truncated to degrees and minutes only. Note that, this would limit the number of positions of an entity to 3600 admissible values. Furthermore, any additional information provided by the user to narrow down the geographical location of an event sharply reduces the number of values to be investigated. For instance, if a user can specify the search to be restricted to a quadrant of a database spanning one geo-cell, then the number of admissible values for the positions of an entity is reduced from 3600 to 900. While this is still a substantially high value the nature of queries in practice serve to alleviate this problem substantially. In practice, queries are typically specified to determine the timelines defined by “interesting” events that are not common. Now, the records relating to an entity’s position within an area of 1 minute x 1 minute would all have the same hashed signature if all other attributes of those records were equal, and would therefore be all rooted on the same index in the hash table. However, an interesting event that is being queried will have one of those attributes different from the majority of the others in the same area (1 minute x 1 minute), and will therefore have a signature that is common with at most a few others. It is then a trivial task to go through the on-disk addresses of the relatively few nodes having that signature and recovering the appropriate records.

In general, attributes with small domain sizes should be included for hashing while those with large domain sizes should be considered only upon representing the values of the corresponding attributes with coarser granularity. For example, if the attribute *entity state* has four admissible values, then it should be used for deriving the hashed-based keys for the corresponding records. Then, if a query defines a value for that attribute, that value can be used to derive the unique hash signature matching the hash key for the corresponding event. On the other hand, if the query does not define a value for that attribute, the number of hash signatures that have to be generated for determining a prospective match with the hash key of the desired event is limited to four.

## 5. Conclusions

The proposed scheme should provide rapid access to timelines within recorded exercises for a large number of queries that are typically constructed for after-action-review. Furthermore, the use of unique hash values for

keys to represent events obviates the need for type-specific comparisons between the attributes of a key and the search criteria. In summary, a sizeable number of queries can be processed far quicker than when using conventional search trees. The proposed approach would expedite delivery of after-action-reviews, support the effectiveness of computer-driven training simulations, and improvements in human performance related to training objectives.

## 6. References

- [1] A. Baker: “Windows NT Device Driver Book: A Guide for Programmers”, Prentice Hall, 1996.
- [2] C. Campbell, Quinkert, K., and W. Burnside: “Training for Performance: A Structured Training Approach”, Special Report 45, August 2000, US Army Research Institute for the Behavioral and Social Sciences.
- [3] R. Nagar: “Windows NT File System Internals: A Developer’s Guide”, O’Reilly, 1997.
- [4] F. Roberts and B. Tesman: “Applied Combinatorics”, Prentice Hall, 2003.
- [5] R. Sedgewick: “Algorithms in C++”, Addison-Wesley, 1992.
- [6] X. Wang, D. Feng, X. Lai, H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD", CRYPTO 2004, <http://eprint.iacr.org/2004/199/>.
- [7] Secure Hash Standard, Federal Information Processing Standards Publication 180-1, 1995.

## 7. Author Biographies

**Sanjeeb Nanda** is a research and development engineer with the Advanced Technologies Division of SDS International. His interests and experience span the areas of simulation, biometrics, parallel computing, and fault-tolerance in storage. He is currently pursuing a doctorate in Computer Science at the University of Central Florida.

**Matthew Archer** is the Engineering Director at SDS International's Advanced Technologies Division. He has been actively involved in the simulation community for over 12 years. His current research interests are advanced visualization techniques, simulation, networked systems, and systems integration.

**Joseph Weeks** serves as a research and engineering psychologist with the Air Force Research Laboratory. His research interests include acquisition of expertise in decision-making and effectiveness of simulation training. He holds a Ph.D. in Educational Psychology from the University of Texas.