

(19) World Intellectual Property Organization
International Bureau



(43) International Publication Date
14 September 2006 (14.09.2006)

PCT

(10) International Publication Number
WO 2006/094365 A1

- (51) International Patent Classification:
G06F 12/12 (2006.01)
- (21) International Application Number:
PCT/AU2006/000326
- (22) International Filing Date: 10 March 2006 (10.03.2006)
- (25) Filing Language: English
- (26) Publication Language: English
- (30) Priority Data:
2005901175 11 March 2005 (11.03.2005) AU
60/661,273 11 March 2005 (11.03.2005) US
- (71) Applicant (for all designated States except US): **ROCK-SOFT LIMITED** [AU/AU]; Level 7, Shell House, 170 North Terrace, Adelaide, S.A. 5000 (AU).
- (72) Inventor; and
- (75) Inventor/Applicant (for US only): **WILLIAMS, Ross, Neil** [AU/AU]; Level 7, Shell House, 170 North Terrace, Adelaide, S.A. 5000 (AU).
- (74) Agent: **MADDERNS**; Level 1, 64 Hindmarsh Square, Adelaide, S.A. 5000 (AU).

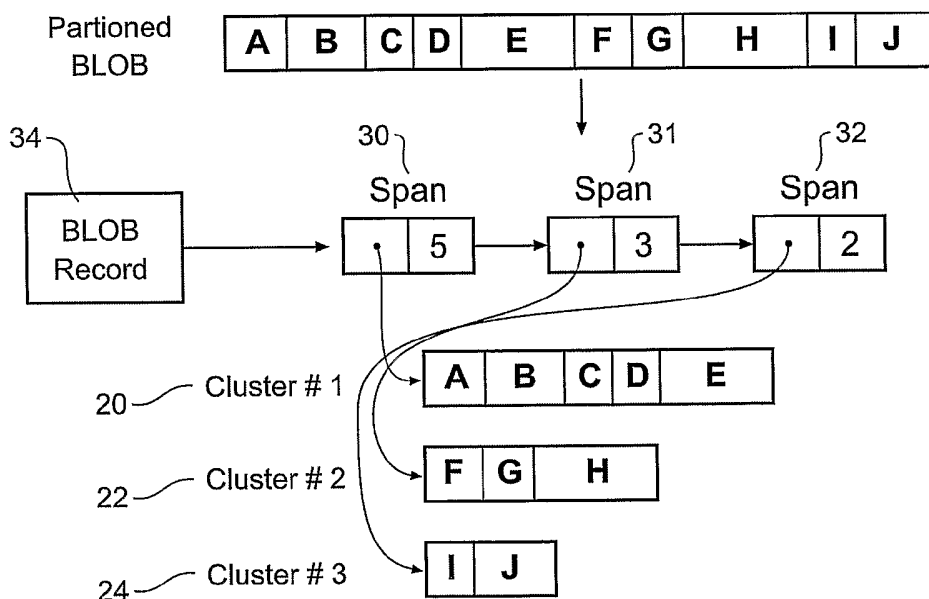
(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— with international search report

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHOD FOR STORING DATA WITH REDUCED REDUNDANCY USING DATA CLUSTERS



(57) Abstract: This specification describes a method and apparatus for storing data in a reduced redundancy form. Binary Large Objects (BLOBs) are partitioned into subblocks according to a partitioning method, and the subblocks are stored in subblock clusters. Each BLOB is represented as a list of spans of subblocks which identifies a contiguous sequence of subblocks within a cluster. Storage redundancy can be reduced because the spans of two different BLOBs can refer to the same subblocks. An index may be used to map subblock hashes to subblock cluster numbers.

WO 2006/094365 A1

Method for Storing Data with Reduced Redundancy Using Data Clusters

1 Field

- 5 The field of the invention relates to a method and apparatus for storing data in computer systems in a form that uses less storage space.

2 Background

- 10 Conventional computer storage systems typically store sequences of bytes as named files in file systems. Despite the fact that many files may be very similar to each other, and have large portions of data in common 130, 132 (**Figure 13**), these systems may not eliminate this redundancy. Instead, they may store each file separately 140, 142 keeping a number of copies 130, 132 of
15 the same data (**Figure 14**).

- Some conventional file systems incorporate conventional non-lossy text compression algorithms (such as GZip) to compress individual files, but this can be viewed as a "keyhole" redundancy elimination technique because it
20 analyses the redundancy of a single file at a time rather than the file system as a whole. These conventional text compression algorithms may be incapable of spotting similarities between widely separated data 150, 152 such as two similar files 130, 132 in different parts of a file system (**Figure 15**).

- 25 What is desired is a method and apparatus for representing data in a form that makes it possible to identify some of their repeated sequences of data and to reduce the number of copies of this repeated data that is stored.

3 Summary

In order to represent several different Binary Large Objects (BLOBs) 10, 12 in a way that causes the number of copies of their repeated sequences of data to be reduced, a representation may be used that allows each repeated sequence
5 to be referred to by more than one BLOB's representation. **Figure 16** depicts one way in which this might be achieved. In this embodiment, each BLOB 160, 162 is divided up into pieces called *subblocks* A, B, C, D, E, F, G and duplicate subblocks 164, 166 are identified and stored just once. Within this
10 framework, the following issues are addressed: the manner in which the BLOBs should be subdivided, the way in way in which the resulting subblocks should be stored, and the method for identifying duplicate subblocks.

15 In an aspect of the invention, each BLOB 10, 12 of data to be stored is divided into subblocks A-J using a partitioning method (**Figure 1**). A variety of partitioning method can be used, but in particular, a fixed-length partitioning method could be used that divides the data into fixed-length subblocks 60-65
(**Figure 6**), or a variable-length partitioning method could be used (**Figure 10**)
20 that divides the data into variable-length subblocks E, F, G, A, B, C, D at positions determined by the data itself (**Figure 1**). An example of this latter method is disclosed in US Patent #5,990,810 to Williams, the same inventor as this invention, which is incorporated into this specification by reference and depicted pictorially in **Figure 37**.

25

The subblocks become the unit of redundancy elimination and, in some embodiments, the system stores each unique subblock at most once. In other embodiments, the number of copies of each unique subblock is reduced, but may be greater than one.

In an exemplary embodiment, the subblocks of BLOBs are stored in groups called *subblock clusters* 20, 22, 24 (**Figure 2**). Each BLOB can be represented by an ordered list (or tree) of records ("*span records*") 30, 31, 32, each of which
5 identifies a contiguous sequence of subblocks within a single cluster 20, 22, 24 (**Figure 3** and **Figure 4**). The BLOB 10 may be represented 34 as the concatenation of the sequences identified by the list of spans 30, 31, 32 (**Figure 3** and **Figure 4**) and can be retrieved from the store by running down the BLOB's list of spans retrieving the subblock content in the subblocks
10 referred to by each span.

In an exemplary embodiment, a cluster 20, 22, 24 may contain subblocks from more than one BLOB X and Y (**Figure 4**), and a BLOB's subblocks may reside in more than one cluster (**Figure 3**). In an exemplary embodiment, a BLOB's
15 subblocks may be stored sequentially within one or more clusters (**Figure 2**). This improves the efficiency of BLOB retrieval because an entire sequence of subblocks within a BLOB can be read from the disk in a single sequential read operation. This is far more efficient than performing a random-access disk seek for each subblock.

20

In an exemplary embodiment, different spans in the same or different BLOBs to include the same subblocks (**Figure 4**). This allows redundancy reduction because BLOBs that contain the same subblocks may be represented by spans that point to the same subblocks (in clusters).

25

In a further aspect of the invention, each cluster is compressed using a data compression method so as to reduce the amount of space used by the clusters. The simplest way to do this is to compress the entire cluster. In some embodiments (particularly those that employ large clusters), it may be

desirable to compress each part of the cluster (e.g. individual subblocks or runs of subblocks) separately so as to allow subblocks within the cluster to be accessed without having to decompress the entire cluster (or at least the part of the cluster before the subblock to be read).

5

In a further aspect of the invention, a directory 70 of the subblocks within each cluster is created for each cluster and stored either within the cluster (typically at the start) (Figure 7) or separately 80, 82 (Figure 8). The directory could also be distributed throughout the cluster (Figure 9), for example by
10 storing each subblock's metadata before the subblock. The directory can contain a variety of metadata for each subblock such as its hash, its length, a subblock identifier, and its position within the cluster.

In a further aspect of the invention, subblocks that are shared by more than
15 one BLOB are identified. In an exemplary embodiment, a subblock index 50 is maintained that maps (or otherwise associates) subblock content or subblock hashes (the hash of a subblock's content), to clusters 52, 54, 56 (Figure 5). During store operations, each subblock to be stored is looked up in the subblock index. If present, the subblock is not stored again. If the subblock
20 is absent, it is stored in a cluster and an entry for it added to the subblock index. In either case, the new subblock is referenced by a span 58.

In an aspect of the invention, when the index indicates that a particular
subblock is already present in the store, the matching subblock's cluster is
25 accessed and the subblocks following the matching subblock in the cluster are compared with the subblocks following the matching subblock in the BLOB to be stored (Figure 10). This comparison can be performed without accessing the index, and in fact can be performed without accessing the actual subblock

content data, so long as the cluster containing the subblock has a subblock directory that contains subblock hashes.

4 Terminology

5

Absent Subblock: A subblock that is not present in the store.

BLOB (Binary Large Object): This is a finite sequence of zero or more bytes (or bits) of data. Despite its name, a BLOB is not necessarily large; a BLOB
10 could be as small as a few bits or bytes or as large as gigabytes.

BLOB Record: A record maintained in a store that records information about a particular BLOB. The BLOB record may also contain, or refer to, a list (or tree) of spans that define the BLOB content.

15

BLOB Table: A data structure that associates BLOB identifiers (for example, without limitation, BLOB hashes) to BLOB records.

Cluster: Short for "Subblock Cluster". A group of associated subblocks. A
20 cluster may have an associated *subblock directory* that provides information about the subblocks in the cluster.

Cluster Subblock Directory: A collection of metadata that provides information about subblocks in a cluster. A subblock's metadata can include
25 (but is not limited to) a subblock's length, hash, identifier, and reference count.

Contiguous: Two things, within an ordered group of things, are contiguous if they are adjacent. N things, within an ordered group of things, are contiguous

if the N things contain exactly N-1 adjacent pairs of things (i.e. if the N things appear as a single continuous run).

Contiguous Subblocks: Two subblocks are contiguous, in some context (e.g. a BLOB or cluster), if they are adjacent. N subblocks are contiguous, in some context, if the N subblocks contain exactly N-1 pairs of subblocks that are adjacent (i.e. the subblocks appear as a single continuous run).

Directory: See *Cluster Subblock Directory*.

10

Disk: A random access storage medium used by computers. Typically the term refers to spinning platters of metal holding magnetised data (hard disks). In the context of this document, the term may more broadly be taken to mean a random access storage medium that is significantly slower than

15 *Memory*.

Fixed-Length Partitioning Method: A method for partitioning data that divides the data into fixed-length subblocks. For example, a fixed-length partitioning method might divide a BLOB into 512-byte subblocks.

20

Hash: A fixed-length sequence of bytes (or bits) generated by a hash algorithm. Hashes of subblocks may be used as representatives of the subblocks to index and compare the subblocks.

Hash Algorithm: An algorithm that accepts a finite sequence of bytes (or bits) and generates a finite sequence of bytes (or bits) that is highly dependent on the input sequence. Typically a hash algorithm generates output of a particular fixed length. Hash algorithms can be used to test to see if two sequences of data might be identical without having to compare the

sequences directly. Cryptographic hashes practically allow one to conclude that two subblocks *are* identical if their hashes are identical. Hash algorithms can be used in exemplary embodiments (without limitation) to generate BLOB identifiers, compare subblocks, and generate hash table keys.

5

Hash of Subblock: See *Subblock Hash*.

Index: See *Subblock Index*.

10 **Index Bucket:** In embodiments that implement the subblock index using a hash table, the hash table may be organised as an array of buckets each of which contains a fixed number of entry slots each of which may either be empty or contain an entry. One purpose of index buckets is to organise a hash table into pieces that can be read from disk and written to disk as a group so
15 as to reduce the number of random access disk operations.

Index Entry: A record in the subblock index. In some embodiments an index record contains an index key and an index value. In some embodiments an index record contains part of an index key and an index value. In some
20 embodiments an index record contains just an index value. In some embodiments an index record contains no value and some or all of a key.

Index Key: The information about a subblock provided to the subblock index in order to retrieve information about the subblock. In some embodiments,
25 the information is retrieved by locating and reading an index entry.

Index Value: The information yielded about a subblock by the index when the subblock (or a derivative of the subblock, an example of which is its hash) is looked up in the index. In some embodiments, the value consists of the

location of the subblock on disk. In other embodiments there may be no value if the sole purpose of the index is to record the presence or absence of a key. In some embodiments, the value consists simply of a cluster number.

5 **Length of Subblock:** The number of bytes (or bits) in the subblock's content.

Linear Search: A way of searching for an object within a collection of objects by inspecting the objects in the collection one by one and where the choice of the next object to inspect is not influenced by the result of earlier inspections.

10

List Of Spans: An ordered list of spans. Such a list can be used to represent the content of a BLOB.

15 **Matching Run:** A sequence of subblocks (in a cluster) that matches another sequence of subblocks (which may be, for example, in a BLOB being stored). In some embodiments the sequence of subblocks is contiguous.

20 **Memory:** A random access storage medium used by computers, typically referring to Random Access Memory (RAM). In the context of this document, the term may more broadly be taken to mean a random access storage medium that is significantly faster than *Disk*.

25 **Partitioning Method:** A method for dividing a BLOB into one or more subblocks such that every byte (or bit) in the BLOB falls within exactly one subblock.

Present Subblock: A subblock that is present within the store.

Reduced Redundancy: Refers to the reduction, in any kind of data representation, of the number of copies of identical sequences of bytes (or bits).

- 5 **Reduced-Redundancy Store:** A storage system that eliminates, in its representation of data, some of the duplicated data within the set of data that it stores.

Reference to a Subblock: A piece of data that identifies a subblock. For
10 example, and without limitation, a reference may identify a subblock by content or by storage location.

Reference Counting: A method for determining when an entity is no longer
required. The method involves maintaining a counter that records the number
15 of references that exist to the entity. When the reference count drops to zero, the entity may be deleted. In some embodiments, BLOBs and/or subblocks have reference counts.

Span: A sequence of subblocks within a cluster. In some embodiments the
20 sequence is contiguous.

Span Record: A record that identifies a span within a cluster. In some
embodiments, a span record contains a cluster number field, a starting
subblock identifier field and a span length (in subblocks or bytes) field.

25

Store: See *Reduced Redundancy Store*.

Subblock: A sequence of bytes (or bits) that has been identified as a unit for the purpose of indexing, comparison and/or redundancy elimination. A BLOB may be partitioned into subblocks.

5 **Subblock Cluster:** A group of one or more subblocks that are stored together. "Cluster" for short.

Subblock Content: The actual data of a subblock, as distinct from the subblock's metadata.

10

Subblock Directory: See *Cluster Subblock Directory*.

Subblock Expiry Date: A piece of metadata associated with a subblock that defines the earliest date when the subblock is guaranteed not to be required
15 by the user.

Subblock Hash: The result of applying a hash algorithm to a subblock. Hashes of subblocks may be used, for example, as representatives of the subblocks to index and/or compare the subblocks.

20

Subblock Identifier: A piece of metadata associated with a subblock. An identifier is unique to the subblock within the cluster, and can therefore be used to unambiguously identify the subblock within its cluster. In some embodiments, subblocks in different clusters may have the same identifier.

25

Subblock Index: A data structure that maps (or otherwise associates) a subblock's hash (or the subblock itself) to the location of the subblock (e.g., without limitation, a cluster number (and possibly also a subblock identifier)).

Subblock Metadata: Information about a subblock. A subblock's metadata can include (without limitation) the subblock's length, the subblock's hash, the subblock's identifier, the subblock's expiry date, and the subblock's reference count.

5

Subblock Record: A record in a cluster subblock directory that contains metadata for a single subblock.

Subblock Reference Count: A piece of subblock metadata that records the current number of references to the subblock. In some embodiments, this will be the number of span records that define a span that includes the subblock.

Subblock Serial Number: A form of subblock identifier. For example, in an embodiment that uses a serial number system, subblocks arriving in a particular cluster are allocated a serial number, starting with 1 for the first subblock and working upwards. In some embodiments, serial numbers are not re-used if subblocks are deleted. In these embodiments, serial numbers provide a way to uniquely identify a subblock within a cluster.

User: A piece of software that is storing and retrieving BLOBs in the store.

Variable-Length Partitioning Method: A partitioning method that divides BLOBs into variable-length subblocks. In a preferred embodiment, a variable-length partitioning method will divide the data at boundaries determined by the content of the data. For example, without limitation, a partitioning method might define a subblock boundary at each position in a BLOB where the previous several bytes hash to a particular predetermined constant value.

25

Virtual Block Device: A device consisting of an array of fixed-length storage blocks provided by an operating system. The virtual device may correspond directly to a physical device, or may be constructed from one or more physical devices (eg. using RAID).

5

Whole Key: A key that is used as a source for smaller derived keys. As a data structure grows and larger derived keys are required, an increasing part of the whole key may be used to form the derived key.

10 Throughout this specification and the claims that follow, unless the context requires otherwise, the words 'comprise' and 'include' and variations such as 'comprising' and 'including' will be understood to be terms of inclusion and not exclusion. For example, when such terms are used to refer to a stated integer or group of integers, such terms do not imply the exclusion of any
15 other integer or group of integers.

The claims that follow in this specification are broad statements of the invention/s disclosed herein and are incorporated into the body of the specification by reference.

20

The reference to any prior art in this specification is not, and should not be taken as, an acknowledgement or any form of suggestion that such prior art forms part of the common general knowledge.

25 5 Brief Description of Figures

Figure 1 depicts the partitioning of a BLOB into subblocks.

Figure 2 depicts the storage of a BLOB's subblocks in clusters.

Figure 3 shows how a BLOB can be represented as an ordered list of spans that identify runs of subblocks within clusters.

Figure 4 shows how two different BLOBs that contain common sequences of data (subblocks A-C and G-J) can be represented in a way that does not
5 require each repeated subblock to be stored more than once.

Figure 5 depicts an index that maps each subblock's hash to the number of the cluster containing the subblock.

10

Figure 6 depicts a partitioning method that divides a BLOB into fixed-length subblocks.

Figure 7 depicts a cluster of subblocks that contains a subblock directory at
15 the start of the cluster.

Figure 8 shows how the directories of clusters may be stored separately from the clusters themselves.

Figure 9 shows how a cluster subblock directory's entries may be distributed
20 throughout the cluster.

Figure 10 depicts an aspect of storing a BLOB where, following the discovery that subblock A (of the BLOB being stored) is already present in cluster #1,
25 the subsequent subblocks in the BLOB (B, C and D) can be compared to the subblocks that follow A in its cluster (here again B, C and D), thereby avoiding having to lookup B, C and D in the subblock index.

Figure 11 depicts a BLOB table that maps BLOB hashes to BLOB records each of which contains (or refers to) an ordered list of spans that identify the subblocks in the BLOB.

5 **Figure 12** depicts a subblock index hash table and shows an entry of the table.

Figure 13 (prior art) depicts two files that contain two instances of the same sub-sequences of data. In addition, File A has identical data within itself.

10 **Figure 14** (prior art) shows how conventional storage systems store files without attempting to identify their common data.

Figure 15 (prior art) shows how conventional data compression will reduce the size of each BLOB but will not identify the common sequences of data
15 between BLOBs.

Figure 16 shows how the representation of two BLOBs that contain the same sequences of data can refer to those sequences of data so that the sequences only need to be stored once.

20

Figure 17 shows how the subblocks at either end of a matching run can be compared directly to see if there are any partial matches.

Figure 18 shows how span records could be augmented with two additional
25 fields "Start Skip" and "End Skip" (each of which holds a byte count) to represent a run of subblocks that includes partial subblocks at the ends of the run.

Figure 19 shows how, when a BLOB is stored, an isolated matching subblock (C) can cause fragmentation in the representation of the BLOB.

Figure 20 shows how fragmentation can be avoided by choosing to store an isolated subblock (C) in the store twice.

Figure 21 depicts a hash table collision in which two keys hash to the same position in the table.

Figure 22 depicts a hash table with an external overflow list.

Figure 23 depicts in-table overflow where overflowing entries are stored in the next empty slot.

Figure 24 depicts a hash table organised as an array of buckets, each of which contains a fixed number of entry slots.

Figure 25 shows how a hash table can be doubled in size by using an extra bit of the whole key.

20

Figure 26 depicts a tree of spans with a furcation of three. Organising spans into a tree makes random access within a BLOB fast. The numbers in the diagram are the lengths of the blocks represented by respective child nodes.

Figure 27 shows the purposeful skipping of subblocks serial numbers within a cluster so as to identify runs of subblocks that appear contiguously in the original BLOBs.

25

Figure 28 shows how a cryptographic hash function H can be used to compare two subblocks A and B without having to compare A and B directly. Instead, their hashes $H(A)$ and $H(B)$ are compared.

- 5 **Figure 29** depicts a subblock index that indexes subblocks A , B , C and D and whose keys are the hashes of subblocks (using hash function H) rather than the subblocks themselves.

Figure 30 shows how a cryptographic hash function H can be used to check
10 that a BLOB has retained its integrity despite having been divided into subblocks and stored in a reduced-redundancy store. The original BLOB's hash is stored with the stored BLOB and is compared with the hash of the retrieved BLOB.

- 15 **Figure 31** depicts an embodiment in which a reduced-redundancy storage system is implemented using ("on top of") an existing file system.

Figure 32 depicts an embodiment in which a reduced redundancy storage system is implemented using ("on top of") a virtual block device provided by
20 an existing operating system.

Figure 33 shows how clusters of varying lengths could be stored inside a single block device or a single file in a file system. A cluster index could be used to find a cluster quickly by its number.

25

Figure 34 shows how a collection of clusters could be stored in a corresponding collection of files in an existing file system. In this example, a directory tree forms a decimal digital search tree on the cluster numbers.

Figure 35 depicts an embodiment in which the structures and metadata required to store a BLOB have been created, but the data itself is not stored.

Figure 36 shows a span (second in the list of spans) that has been augmented with an alternative span that points to the same data as the original span (subblocks FGH), but located in a different part of the store (in this case a different cluster).

Figure 37 shows the partitioning of a block *b* into subblocks using a constraint *F*, and the calculation of the hashes of the subblocks using hash function *H*.

Figure 38 shows how a reduced redundancy storage system might be deployed on typical computer hardware. All the data structures reside on disk. The index is also held in memory along with some caches that store working copies of some BLOB records and clusters.

Specific embodiments of the invention will now be described in some further detail with reference to and as illustrated in the accompanying figures. These embodiments are illustrative, and are not meant to be restrictive of the scope of the invention. Suggestions and descriptions of other embodiments may be included within the scope of the invention but they may not be illustrated in the accompanying figures or alternatively features of the invention may be shown in the figures but not described in the specification.

6 Detailed Description

Figure 5 provides an overview of elements of a typical embodiment of the invention. This embodiment contains BLOB records 51, 53, span lists 58, clusters 52, 54, 56 and a subblock index 50. **Figure 38** shows how these elements might be deployed on typical computer hardware. All the data

structures reside on disk 380. The index 381 is also held in memory along with some caches that store working copies of some BLOB 382 records and clusters 383.

6.1 An Overview Of Hash Functions

- 5 Although hash functions are not used in all embodiments, hash functions provide advantages in many embodiments. The following is an overview of exemplary hash functions that may be used in connection with various embodiments of the present invention.
- 10 A hash function accepts a variable-length input block of bits and generates an output block of bits that is based on the input block. Most hash functions guarantee that the output block will be of a particular length (e.g. 16 bits) and aspire to provide a random, but deterministic, mapping between the infinite set of input blocks and the finite set of output blocks. The property of
- 15 randomness enables these outputs, called "hashes", to act as easily manipulated representatives of the input block.

Hash functions come in at least four classes of strength.

- 20 **Narrow hash functions:** Narrow hash functions are the weakest class of hash functions and generate output values that are so narrow (e.g. 16 bits) that the entire space of output values can be searched in a reasonable amount of time. For example, an 8-bit hash function would map any data block to a hash in the range 0 to 255. A 16-bit hash function would map to a
- 25 hash in the range 0 to 65535. Given a particular hash value, it would be possible to find a corresponding block simply by generating random blocks and feeding them into the narrow hash function until the searched-for value appeared. Narrow hash functions are usually used to arbitrarily (but deterministically) classify a set of data values into a small number of

groups. As such, they are useful for constructing hash table data structures, and for detecting errors in data transmitted over noisy communication channels. Examples of this class: CRC-16, CRC-32, Fletcher checksum, the IP checksum.

5

Wide hash functions: Wide hash functions are similar to narrow hash functions except that their output values are significantly wider. At a certain point this quantitative difference implies a qualitative difference. In a wide hash function, the output value is so wide (e.g. 128 bits) that the probability of any two randomly chosen blocks having the same hashed value is negligible (e.g. about one in 10^{38}). This property enables these wide hashes to be used as "identities" of the blocks of data from which they are calculated. For example, if entity E1 has a block of data and sends the wide hash of the block to an entity E2, then if entity E2 has a block that has the same hash, then the a-priori probability of the blocks actually being different is negligible. The only catch is that wide hash functions are not designed to be non-invertible. Thus, while the space of (say) 2^{128} values is too large to search in the manner described for narrow hash functions, it may be easy to analyse the hash function and calculate a block corresponding to a particular hash. Accordingly, E1 could fool E2 into thinking E1 had one block when it really had a different block. Examples of this class: any 128-bit CRC algorithm.

10

15

20

Weak one-way hash functions: Weak one-way hash functions are not only wide enough to provide "identity", but they also provide cryptographic assurance that it will be extremely difficult, given a particular hash value, to find a block corresponding to that hash value. Examples of this class: a 64-bit DES hash.

25

Strong one-way hash functions: Strong one-way hash functions are the same as weak one-way hash functions except that they have the additional property of providing cryptographic assurance that it is difficult to find *any* two different blocks that have the same hash value, where the hash value is
5 unspecified. Examples of this class: MD5, and SHA-1.

These four classes of hash provide a range of hashing strengths from which to choose. As might be expected, the speed of a hash function decreases with strength, providing a trade-off, and different strengths are appropriate in
10 different applications. However, the difference is small enough to admit the use of strong one-way hash functions in all but the most time-critical applications.

The term *cryptographic hash* is often used to refer to hashes that provide
15 cryptographic strength, encompassing both the class of weak one-way hash functions and the class of strong one-way hash functions.

Exemplary embodiments of the present invention may employ hash functions in at least two roles:

20

1 To determine subblock boundaries.

2 To generate subblock identities.

25 Depending on the application, hash functions from any of the four classes above could be employed in either role. However, as the determination of subblock boundaries does not require identity or cryptographic strength, it would be inefficient to use hash functions from any but the weakest class. Similarly, the need for identity, the ever-present threat of subversion, and the

minor performance penalty for strong one-way hash functions (compared to weak ones) suggests that nothing less than strong one-way hash functions should be used to calculate subblock identities.

5 The security dangers inherent in employing anything less than a strong one-way hash function to generate identities can be illustrated by considering a storage system that incorporates the invention using any such weaker hash function. In such a system, an intruder could modify a subblock (to be manipulated by a target system) in such a way that the modified subblock has
10 the same hash as another subblock known by the intruder to be already present in the target system. This could result in the target system retaining its existing subblock rather than replacing it by a new one. Such a weakness could be used (for example) to prevent a target system from properly applying a security patch retrieved over a network.

15

Thus, while wide hash functions could be safely used to calculate subblocks in systems not exposed to hostile humans, even weak one-way hash functions are likely to be insecure in those systems that are.

20 We now turn to the ways in which hashes of blocks or subblocks can actually be used.

6.2 The Use of Cryptographic Hashes

The theoretical properties of cryptographic hashes (and here is meant strong one-way hash functions) yield particularly interesting practical properties.

25 Because such hashes are significantly wide, the probability of two randomly-chosen subblocks having the same hash is practically zero (for a 128-bit hash, it is about one in 10^{38}), and because it is computationally infeasible to find two subblocks having the same hash, it is practically guaranteed that no intelligent agent will be able to do so. The implication of these properties is that from a

practical perspective, the finite set of hash values for a particular cryptographic hash algorithm is one-to-one with the infinite set of finite variable length subblocks. This theoretically impossible property manifests itself in practice because of the practical infeasibility of finding two subblocks
5 that hash to the same value.

This property means that, for the purposes of comparison (for identity), cryptographic hashes may safely be used in place of the subblocks from which they were calculated. As most cryptographic hashes are only about 128
10 bits long, hashes provide an extremely efficient way to compare subblocks without requiring the direct comparison of the content of the subblocks themselves.

Some of the ways in which cryptographic hashes are used in exemplary
15 embodiments of this invention are:

Comparing subblocks: Cryptographic hashes H can be used to compare
280 two subblocks A, B without having to compare, or require access to, the content of the subblocks (**Figure 28**).

20

Indexing subblocks: To index a collection of subblocks A, B, C, D, an index
290 can be constructed whose keys are the hashes of the subblocks 292, 294, 296, 298 (**Figure 29**).

25 **BLOB check:** Cryptographic hashes can be used to ensure that the partitioning of a BLOB 300 into subblocks 302 and the subsequent reassembly of the subblocks into a reconstructed BLOB 304 is error-free. This can be done by comparing 309 the hash 306 of the original BLOB with the hash 308 of the reconstructed BLOB (**Figure 30**).

6.3 Use of Hashes as a Safety Net

Embodiments of the present invention may add extra complexity to the storage systems into which they are incorporated. This increased complexity carries the potential to increase the chance of undetected failures.

5

The main mechanism of complexity is the partitioning of BLOBs into subblocks, and the subsequent re-assembly of such subblocks. By partitioning a BLOB into subblocks, a storage system creates the potential for subblocks to be erroneously added, deleted, rearranged, substituted, duplicated, or in some other way exposed to a greater risk of accidental error.

10

This risk can be reduced or eliminated by calculating the hash (preferably a cryptographic hash) of the BLOB before it is partitioned into subblocks, storing the hash with an entity associated with the BLOB as a whole, and then later comparing the stored hash with a computed hash of the reconstructed block. Such a check would provide a very strong safety net that would virtually eliminate the risk of undetected errors arising from the use of this invention (**Figure 30**).

15

Another way to perform a check on a BLOB is to hash the concatenation of the hashes of its subblocks and check that value when retrieving the BLOB from the store. This method has the advantage that less data must be hashed overall and this could make such an embodiment more efficient.

20

6.4 Storage of Subblocks within Clusters

There are a number of ways in which subblocks can be stored within clusters. The term "subblock content" refers to the sequence of bytes that forms the actual subblock. In an exemplary embodiment, subblocks 72 in a cluster 74 are stored back-to-back with no intervening metadata (**Figure 7**). In embodiments

25

where the cluster does not have its own directory, back-to-back subblock content may be all that the cluster need contain.

An advantage of storing subblocks back-to-back is that contiguous runs of
5 subblocks can be read from a cluster as a single sequential operation and the subblocks then held in memory and written out as a single sequential operation, without having to remove metadata first.

A number of methods can be used to determine how subblocks should be
10 split into clusters. One method is to write subblocks to a cluster until it has at least S subblocks, where S is a predetermined constant. Another method is to write subblocks to a cluster until it contains at least M megabytes, where M is a predetermined constant.

6.5 Cluster Subblock Directories

15 A cluster can have a subblock directory that provides information about the subblocks within the cluster and allows subblocks within the cluster to be located quickly.

If the cluster has a directory 70, the directory could be placed at the start of
20 the cluster (**Figure 7**) or end of the cluster. Another alternative is to interleave the directory 90 entries with the subblock content 92 (**Figure 9**). Finally, the directory 80, 82 can be stored separately (**Figure 8**).

One simple option is to place an upper limit L on the number of subblocks in
25 a cluster and represent directories as a count plus an array of L directory entries, regardless of the number of subblocks in the cluster. This yields a fixed-length directory 80, 82, allowing the directories of the clusters to be stored in a single array separately from the remaining cluster content 84, 86 (i.e. subblock content) (**Figure 8**).

6.6 Subblock Metadata in Cluster Subblock Directories

A cluster's subblock directory could store the length of each subblock.

Typically this would be measured in bytes. If the length of each subblock is stored, the cluster's subblock content can be separated into subblocks without
5 having to invoke the partitioning method to determine where the boundaries are between subblocks.

A cluster's directory could store the hash of each subblock. For example, a directory could store the 128-bit MD5 or 160-bit SHA-1 hash of each subblock
10 in the cluster. Storing the hash of each subblock X is useful because, during storage, it allows the system to confirm that a newly arrived subblock Y has been found in a cluster without having to compare the contents of subblock X with the contents of subblock Y. Instead, the system calculates the hash of subblock Y and compares it to the hash of subblock X (which can be found in
15 its cluster's directory). Thus, subblocks in BLOBs being stored can be tested for presence in the store using just the index and the cluster directories, with no need to read the content of subblocks in the store.

A cluster's directory could also store a subblock identifier for each subblock.
20 The subblock's identifier is unique within the set of subblocks within the cluster. One simple way of implementing subblock identifiers is to choose a fixed width (e.g. 16 bits), allocate a serial number counter within each cluster, and start from zero and allocate the next integer to each subblock as its serial number identifier. When the counter reaches its maximum value, the cluster
25 can simply be closed to new data. Alternatively, if subblocks have been deleted from the cluster, unused identifiers may be reallocated. This is one of many ways to implement a subblock identifier.

If serial numbers are used as subblock identifiers, their contiguity can be used to indicate the start and end of runs of subblocks 276-278 in a cluster that were stored from a single run of subblocks in a BLOB. In one embodiment, this is achieved by skipping (wasting) a serial number at the end of each stored run 272, 274 (**Figure 27**). If serial numbers are not used, a boolean value can be added to each subblocks's metadata to indicate the end of (with respect to the subblock run within the originating BLOB) subblock runs within the cluster.

6.7 Compression of Clusters

There are a number of ways in which compression (e.g., without limitation, GZip) could be incorporated into the system. One simple method is to apply compression as a single sequential operation on each cluster before it is written to disk. Another method is to compress each subblock individually. Another method is to compress each run of subblocks with contiguous serial numbers.

Clusters could be stored on disk in a compressed form. They could also be stored in memory in a compressed form.

6.8 Span Subblock-Run Identification

Each span identifies a run of subblocks within a particular cluster. In exemplary embodiments, the span contains information that identifies the cluster containing the run of subblocks. There is a greater range of possibilities for identifying the run of subblocks. For this, either the first and last subblock in the run can be identified, or the first (or last) subblock identified and a length provided. The length could be measured in bytes or subblocks.

To identify a subblock in an exemplary embodiment, a span may use the subblock's hash (in which case the cluster would have to be searched for the subblock (using the subblock's directory (if it has one)), the subblock's position within the cluster (e.g. "the third subblock") or the subblock
5 identifier.

Hashes are relatively wide. If there were (say) 1000 subblocks within a cluster, the subblock identifier should only need to be about 10 bits wide, yet a typical hash is 128 bits wide. Use of the position (measured in subblocks) of a
10 subblock within its cluster is more space efficient, but breaks down if subblocks are deleted from the cluster (as might happen if a BLOB containing the subblocks is deleted from the store). To avoid this, in exemplary embodiments, a unique identifier can be allocated to each subblock in the cluster (unique within the cluster). This identifier can be stored with each
15 subblock's metadata in the cluster's directory. Such an identifier can be narrow enough (in bits) but still distinctly identify a subblock, even if the subblocks are shifted within the cluster.

Another approach is to refer to subblocks by their hash, but to store the
20 smallest number of hash bytes that are required to distinguish the subblock from all the other subblocks in the same cluster. A small fixed-length field in the span record could be used to record how many bytes of hash are recorded. This method eliminates the need for subblock identifiers, yet does not burden the span records with lengthy hashes. The method causes span
25 records to have variable length. One potential problem with this method is that subblocks that are added to a cluster could cause existing references to become ambiguous. This problem can be overcome by noting such ambiguous references will always refer to the *first* subblock that satisfies the ambiguous reference.

Another method is to use subblock serial numbers, but to allocate them only to subblocks that are directly referred to by a span. As, in practice, very few subblocks are the first subblock of a span, a far smaller number of serial
5 numbers would need to be stored.

6.9 Partial Subblock Matching

During the storage of a BLOB 170, when a run of one or more matching subblocks B, C (the "matching run") 172 is found within a cluster 174, it is likely that some part of the non-matching subblocks on either side of the run
10 of matching subblocks will match the corresponding parts of the corresponding subblocks in the BLOB being stored. **Figure 17** shows a BLOB 170 that is being stored and a cluster 174 against which it is being compared. Using indexing, a matching run of subblocks BC has been found. The subblocks on either side do not match. A does not match E, and D does not
15 match F. So the matching run is just two subblocks long. However, having discovered the BC match, the surrounding subblocks can be compared at a finer grained level.

Comparing the end of subblock A with the end of subblock E reveals that
20 they share the same (say) 123-byte suffix. Similarly, comparing the beginning of subblock D with the beginning of subblock F reveals that they share the same (say) 1045-byte prefix. These are called partial subblock matches.

Once a partial subblock match has been found, there are a number of ways in
25 which it can be exploited. **Figure 18** shows how the span record structure could be augmented to include two extra fields "Start Skip" 180 and "End Skip" 182 that record the number of bytes that should be ignored at the beginning of the first subblock in the span and the end of the last subblock in the span. An alternative is to use two fields "Start Extend" and "End Extend"

that record the number of bytes to extend either end of the subblocks. An embodiment may choose to use either or both of each of the above fields.

Another way to refer to a range of bytes within a run of subblocks is to
5 replace the End Skip field with a length being the total number of bytes in the span.

6.10 Reducing Fragmentation

If the BLOB being stored contains many subblocks that are already in the store, but are scattered throughout many different clusters, the BLOB will end
10 up being represented by a list of spans that point all over the disk. It will, in short, be highly fragmented.

One particularly unfortunate form of fragmentation occurs when a single subblock matches within a long run of non-matching subblocks. **Figure 19**
15 depicts an example of this where BLOB1 190 has already been stored in the store and BLOB2 192 is being stored and where a single matching subblock C appears within an otherwise non-matching run of subblocks F-M in BLOB2. The result is that a single span record 194 for the matching subblock is created in the span list 196. This kind of fragmentation is likely to increase BLOB2's
20 retrieval time because a random disk access will have to be performed to access the first cluster 198 as well as the second 199.

Some embodiments can avoid this kind of single-matching-subblock fragmentation by treating isolated matching subblocks as not matching, and
25 to store them a second time. **Figure 20** shows how ignoring the isolated match of subblock C causes it to be stored twice, using extra space, but reducing fragmentation for BLOB2 202. This method can be generalized by ignoring all matching runs of less than a predefined threshold T of matching subblocks. In

some embodiments, any value of T greater than one is likely to reduce fragmentation; even a value of two would be helpful.

6.11 BLOB Table

A storage system that stores BLOBs will need to provide some way to allow
5 its user to refer to BLOBs so that they can be retrieved.

One method is to use the hash 110 of a BLOB as its identifier (**Figure 11**). Thus, a user would submit a BLOB to the storage system and make a note of the hash of the BLOB (e.g. the MD5 hash). To retrieve the BLOB, the user
10 would present the hash to the storage system, and the system would return the BLOB.

Another method is to assign arbitrary names to each BLOB. Conventional file systems do this.
15

Whatever naming scheme is adopted must be implemented. Such an implementation will consist essentially of a mapping from the BLOB 112 namespace to the BLOB records 114 themselves (which contain (or refer to) lists of spans 116) (**Figure 11**). This mapping can be achieved using all kinds
20 of conventional data structures such as digital search trees, B trees and hash tables.

6.12 Lists and Trees of Spans

Each BLOB 114 record referenced by the BLOB table 112 will contain any metadata of the BLOB and will either contain, or point to an ordered sequence
25 of span records 116 (**Figure 11**), each of which identifies a [contiguous] run of subblocks within a cluster.

Keeping spans in an ordered list of spans makes it efficient to retrieve an entire BLOB sequentially, but requires a linear search in order to perform a random access read on the stored BLOB (or a binary search if the span records can be accessed randomly). To speed up random access reads, a BLOB's spans
5 can be organised into a tree structure. **Figure 26** shows an example of a tree with a furcation of three (though any furcation could be used). Each non-leaf node represents a finite block of bytes which is the concatenation of the blocks represented by its child nodes. Each node contains three lengths which are the lengths of the blocks its child nodes represent. Each leaf node consists of a
10 span 260 which identifies a sequence of one or more subblocks within a cluster. A random access read of bytes J through K of the stored BLOB represented by such a tree can be performed by moving down the tree to find the spans that contain bytes J through K and then retrieving the subblock content bytes from the clusters.

15 **6.13 Subblock Index**

A subblock index (**Figure 5**) makes it possible to determine whether a particular subblock is already present in the store without performing a linear search of all the clusters in the store. The index can also provide information that assists in locating the matching subblock.

20

The index 50 can be viewed as an organised collection of *entries*, each of which binds an index key to an index value. Entries could be stored in the index explicitly as entry records (each consisting of a key field and an value field) or implicitly (if, for example, the index is organised as a binary digital search
25 tree on the keys with the values in the leaf nodes).

The index keys could be the subblock's content, the hash of the subblock's content or just part of the hash of the subblock's content. Storing just part of the hash of the subblock's content (e.g. the first eight bytes of an MD5 hash

rather than the entire sixteen bytes) can reduce the size of the index at the expense of the occasional collision. If more than one subblock has the same partial-hash, then the index must be capable of storing and retrieving both entries.

5

The index values should consist of a piece of information that assists in locating the subblock within the store. In one embodiment extreme, the value could provide a precise reference, consisting of a cluster number and information that identifies a particular subblock within the cluster (e.g. an identifier, subblock serial number or subblock hash). At the other embodiment extreme, the index value could consist of just a cluster number. Once the cluster number of a subblock is known, the cluster directory can be searched to find the subblock in the cluster, if it is there. To save even more space in the index, the index value could consist of only part of the cluster number (e.g. all but the bottom two bits of the cluster number), which would require more than one cluster to be searched.

10
15

A good combination of choices is to make the index keys the top eight bytes of the subblock hash and the index value the number of the cluster containing the subblock. So long as there is a directory for each cluster, these choices keep the index size down while still providing fast access to any subblock in the store.

20

The index can be implemented by a variety of data structures including a digital search tree, binary tree, and hash table.

25

6.14 Storing the Index

The index can be stored in memory or on disk. Reducing the size of the index is important if the index is held in memory. Experiments show that, in some embodiments, the system runs much faster if the index is held in memory.

Not having to store information identifying the position of the target subblock within a cluster reduces the size of the index significantly. Therefore, typical embodiments store only the cluster number in the index.

6.15 Use of a Hash Table for the Subblock Index

5 As the subblock index is so critical in determining the speed of a reduced redundancy storage system, it is important that this data structure be designed to provide the fastest possible access. A hash table provides a very good data structure for a subblock index as it provides access in $O(1)$ time. However, this hash speed access comes at a price. The next few sections
10 address the challenges that a subblock index poses.

6.16 Hash Table Collisions

This section contains a discussion of hash table collisions, and applies only if the index is implemented using a hash table.

15 A collision occurs in a hash table when two keys 210, 212 hash 214 to the same position (*slot*) 216 (**Figure 21**). One way to address this situation is simply to throw away the second entry. This can be an appropriate choice in some contexts. However, if the hash table is not allowed to be lossy, this option cannot be used, and one of a wide variety of techniques can be employed to
20 deal with this "overflow" situation.

One classic technique for dealing with a collision is to have a separate storage area called an overflow area 220. Each hash table slot contains an overflow field 222. If a collision occurs in the slot, the overflowing entry 224 is stored in
25 the overflow area and a pointer to the entry is placed in the slot 222 (**Figure 22**). The overflow area allows entries to point to each other too 226, allowing each overflowing slot to point to a list of entries (**Figure 22**). This technique works well if a separate overflow area is available (as it might be in

the form of a memory heap if the hash table were in memory). However, if the hash table is on disk, placing overflowing entries in an overflow area will usually involve performing at least one random access seek, which is very slow.

5

A cleaner approach to collisions is to store the colliding entry in the hash table itself. In a classic approach, when a collision occurs, the second item's key is hashed using a second hash function and the resultant slot examined. If it is empty, the entry can be stored there. If it is not, a third hash function can be invoked and so on until an empty slot is found. If the entire table is full, then the table will have to be split before the new entry can be added. In general, a hash function $H(K,X)$ can be defined where K is the key to be hashed and X is a positive integer which can be increased to find successive candidate locations in the hash table for a colliding entry. To search for a key K , slots $H(K,X)$ are examined for $X=1,2,\dots$ until a slot containing the key is found, or an empty slot is encountered (which indicates the end of the hash overflow chain within the table).

The problem with this approach is that, if the hash table is large and on disk, following a collision chain requires performing a series of random access seeks on the disk, which is extremely time consuming. This can be avoided by defining $H(K,X) = H(K,X-1)+1$; in other words, overflowing to the next adjacent slot (Figure 23) (and wrapping around at the ends of the table). This technique keeps the accesses local. If, when reading the first slot accessed, the next S slots are read as well, for small S the disk operation will take no extra time (e.g. reading 1K instead of 12 bytes) and will provide the overflow slots as well. Once the new entry is added, the slots can also be written back to disk as a group. The value S can be adjusted (possibly

25

dynamically) so as to ensure that it is rare for a collision chain to span more than S slots (and thereby require an additional disk access).

6.17 Hash Table Buckets

If the index is stored on disk, random access reads and writes to the index can be time consuming. So if there is a chance of an overflow from one slot into another, it makes sense to read and write more than one slot at a time. One way to do this is to divide the table into buckets 240 (Figure 24) and read and write buckets instead of entries. For example, one could replace a table of 1024 slots with a table of 64 buckets each of which contains 16 slots. To search for an entry, a bucket can be read and a linear search performed within the bucket (or possibly a binary search if the keys in the bucket are sorted). Only occasionally will a bucket fill, in which case the overflow can move to the next bucket. So long as the table is not allowed to grow too full, overflow chains should not become very long.

6.18 Hash Table Growth

One problem with using a hash table is that when it fills up, there is no obvious way to expand it.

One approach to this problem is simply to never allow the table to become full. This can be done by initially creating a hash table so large that it never becomes full in the particular application. However, in some applications, it may not be possible to predict the load on the hash table in advance, so other solutions must be found.

One approach is to abandon the hash table by creating a new larger hash table and transferring all the entries in the old table to the new table. This is a perfectly feasible approach so long as there is enough memory to hold both tables during the transfer.

Another approach is to double the size of the hash table whenever it becomes full, and transfer (about) half of the entries in the first (old) 250 half to the second (new) 251 half. **Figure 25** shows how this can be done. If the initial
5 hash table has 2^K entries, then the bottom K bits of the whole key can be used to index the table. If the table becomes full, it can be doubled. The new table will use the $K+1$ lowest bits of the whole key 254 as a key. The extra bit of the key that is now used (bit K) distinguishes between the old and new halves of the doubled table. The leftmost rest of the whole key remains unused. All that
10 remains to be done is to move the entries in the old half of the doubled table whose bit K is 1 to the corresponding position in the new half. In fact, overflow makes it a bit more complex than this. First, overflow may mean that an entry is not in its "natural" position in the old half of the table, so simply moving all the entries with bit K set would move some entries to
15 incorrect positions. This means that they need to be rehashed. Second, the removal of entries in the old half might cut some overflow chains, rendering some entries inaccessible. So when an entry is moved, the overflow chain of that entry has to shuffle back to fill the gap.

6.19 Subblock Index Partial Key Storage

20 One way to reduce the size of the index is not to store a copy of the index's key in each index entry. For example, if the index keys are 128-bit MD5 hashes (of the subblocks), then one way to reduce the size of the index is to record only part of the key in the entries of the index.

25 For example, if the index were implemented as a hash table 120, each hash table entry 122 would typically contain a cluster number 124 and a copy of the subblock hash 126 (**Figure 12**). This would ensure that if two subblocks hashed to the same position in the index's hash table, the two entries would be distinguishable. However, if the hashes were 128 bits wide and only 64 bits

of each hash were to be stored, then the entries would still remain distinguishable yet would consume half the space.

In the extreme case, the hash table would not contain any part of any key.

- 5 Instead, each subblock hash would hash to a position in the hash table and all the clusters found at that position would have to be searched. This is still far better than a linear search of all the clusters in the store.

The best approach is to store some part of the hash, but not all of the hash.

- 10 This means that, on rare occasions, there may be more than one matching entry in the hash table and all of the clusters referred to by the set of matching entries will have to be searched. Storing only part of the hash in the entries provides enough differentiation to avoid having to check several clusters but still uses significantly less space than a complete hash.

15 **6.20 BLOB Deletion**

- In some applications, there will be a need to delete BLOBs as well as store them. Deleting BLOBs can become involved because the obvious approach of simply deleting all the subblocks referenced in the BLOB's spans (and then deleting the BLOB's spans and BLOB record) fails because such an action
- 20 could delete subblocks that are also part of other (non-deleted) BLOBs. A more sophisticated approach is desirable.

- One approach to BLOB deletion is to add an extra piece of metadata to each subblock in the store: a reference count. A subblock's reference count stores
- 25 the number of spans (in all BLOBs) that include the subblock. Under a reference counting approach, a subblock's reference count is incremented when a new span is created that includes the subblock (i.e. during BLOB storage) and is decremented when such a span is deleted (i.e. during BLOB deletion). A subblock can be deleted when its reference count falls to zero.

The reference count approach allows the storage system to provide BLOB deletion functionality. However, the user might not need this functionality. An alternative to reference counting is an expiry system. In this system, each BLOB and each subblock has an expiry date. When a BLOB is stored, the user provides an expiry date and the BLOB is added, and a new list of spans created for the BLOB. As part of the addition process, the subblocks referred to by the span list have their expiry dates set to the maximum of their previous expiry date and the date of the BLOB that is newly referencing them. Once BLOBs and subblocks are labelled with expiry dates, a background process can delete expired BLOBs and subblocks at will.

6.21 Embodiments Using an Existing File System

Embodiments of the present invention could be implemented on top of an existing file system. **Figure 31** shows how this could be organized.

In such an embodiment, each cluster could be stored in a single cluster file 340. If clusters are numbered, the name of each cluster file could include the cluster number. The cluster files could be stored in a single directory 342, or a tree of directories 344 (**Figure 34**). A cluster could be modified directly by performing random access read and write operations upon its file, or could be modified by reading the cluster file completely into memory, modifying it, and writing the entire file back to disk using a sequential IO operations.

Another embodiment could employ an existing file system, but use only a single file. The clusters could be stored within the single file contiguously 330 and located using a cluster index 332 held in memory (**Figure 33**).

If fixed-length cluster directories are employed, the entire set of cluster directories could be stored in a single file that stores the directories as an

array, allowing random access to a particular directory using a random access to the file.

Each BLOB could be stored in a file whose name is the hash of the BLOB. The
5 BLOB files could be stored in a BLOB directory, or a directory (perhaps a digital search tree organized by successive bytes of the BLOB hash). Each BLOB file could contain the list of spans that represents the BLOB. To avoid incurring the file system's per-file space overhead, multiple BLOB's could be stored within a single "BLOB" file.

10 **6.22 Embodiments Using A Virtual Block Device**

Embodiments of the present invention could be implemented using a virtual block device 320 provided by an existing operating system 322 (**Figure 32**). The clusters could be stored within the virtual block device contiguously and located using a cluster index held in memory.

15 **6.23 Embodiments That Do Not Store The Data**

An embodiment could be created that is identical to any of the embodiments previously discussed, but which does not actually store any BLOB data (**Figure 35**). In such embodiments, all the storage structures and metadata could be constructed, but the BLOB/subblock content not stored. An
20 embodiment such as this could be useful in applications where a BLOB2 must be analysed in relation to a previously encountered BLOB1, but in which neither BLOB must actually be stored.

For example, in a security environment, it may be advantageous not to store
25 the BLOB content itself, but use BLOB metadata to analyse BLOBs in relation to previously encountered BLOBs. By using the storage structures and metadata representative of existing BLOBs, a store can analyse a document with respect to a body of previously encountered BLOBs without requiring

access to the previously encountered BLOBs. This could be applied in, for example, a secure gateway.

6.24 A Note on Scope

It will be appreciated by those skilled in the art that the invention is not
5 restricted in its use to the particular application described. Neither is the
present invention restricted in its preferred embodiment with regard to the
particular elements and/or features described or depicted herein. It will be
appreciated that various modifications can be made without departing from
the principles of the invention. Therefore, the invention should be understood
10 to include all such modifications within its scope.

THE CLAIMS DEFINING THE INVENTION ARE AS FOLLOWS:

1. A method for storing a BLOB comprising:
dividing the BLOB into a plurality of subblocks;
5 storing the subblocks in a plurality of clusters; and
creating a representation of the BLOB as a plurality of spans, where each
span identifies a sequence of subblocks within a cluster and where at least
one subblock is referred to by more than one span.
- 10 2. A method according to claim 1 wherein each span identifies a sequence of
one or more contiguous subblocks within a cluster.
3. A method according to claim 1 wherein the plurality of spans is an ordered
list.
- 15 4. A method according to claim 1 wherein the plurality of spans is a tree of
spans.
5. The method of claim 1 wherein two or more subblocks are stored as a
20 contiguous sequence of bytes in a cluster with no intervening metadata.
6. The method of claim 1 wherein the subblocks are interleaved with some
subblock metadata.
- 25 7. The method of claim 1 wherein each span identifies a sequence of
contiguous subblocks within a cluster using at least one of the following: a
cluster identifier, a cluster address, a subblock identifier, a subblock location
within a cluster, a length.

8. The method of claim 7 wherein the length is a number of subblocks.
9. The method of claim 7 wherein the length is a number of bytes.
- 5 10. The method of claim 1 wherein an upper bound is placed on the number of subblocks in each cluster.
11. The method of claim 1 wherein an upper bound is placed on the number of bytes in each cluster.
- 10 12. The method of claim 1 wherein the data is divided by partitioning the set of data b into a plurality of subblocks at least one position $k+1$ within b for which $b[k-A+1..k+B]$ satisfies a predetermined constraint and wherein A and B are natural numbers.
- 15 13. The method of claim 1 wherein the data structures to store the data are created, but the data itself is not stored.
14. A method for storing a set of data comprising:
- 20 dividing the data into a plurality of subblocks;
 storing the subblocks in a plurality of clusters; and
 creating a representation of the set of data as a plurality of spans, where each span identifies a sequence of subblocks within a cluster and where at least one subblock is referred to by more than one span.
- 25 15. The method of claim 14 wherein the data is a data file.
16. The method of claim 14, comprising the further step of reconstructing the set of data from the subblocks referenced by the group of spans.

17. The method of claim 14 wherein the data is divided by partitioning the set of data b into a plurality of subblocks at least one position $k | k+1$ within b for which $b[k-A+1..k+B]$ satisfies a predetermined constraint and wherein A and
5 B are natural numbers.
18. The method of claim 1 wherein each cluster has a directory of subblocks and the directory contains at least one of: the length of each subblock, the hash of each subblock, the position of each subblock within the cluster, an
10 identifier for each subblock.
19. The method of claim 18 wherein the cluster directory is stored within the cluster.
- 15 20. The method of claim 18 wherein the cluster directory is stored separately from the cluster.
21. The method of claim 18 wherein the cluster directory has a fixed length regardless of the number of subblocks that the cluster contains.
20
22. The method of claim 18 wherein the cluster directories are of fixed length and are stored separately from the clusters in a fixed length array of cluster directories.
- 25 23. The method of claim 18 wherein the cluster records the boundaries between contiguous runs of subblocks in the cluster.

24. The method of claim 18 wherein a boundary between subblocks in a cluster is identified by employing ordered identifiers and by allocating a non-contiguous identifier to a subblock at a boundary.
- 5 25. The method of claim 1 including the further step of:
compressing at least one cluster using a compression algorithm.
26. The method of claim 1 including the further step of:
compressing at least one subblock using a compression algorithm.
- 10 27. The method of claim 1 wherein at least two adjacent subblocks are compressed using a compression algorithm.
28. The method of claim 1 including the further step of:
15 maintaining an index that maps at least one subblock to the cluster containing the subblock.
29. The method of claim 1 including the further step of:
maintaining an index that maps the hash of at least one subblock to the
20 cluster containing the subblock.
30. The method of claims 28 or 29 wherein the index includes the position of each subblock within the cluster containing the subblock.
- 25 31. The method of claims 28 or 29 wherein the index is implemented as a digital search tree whose keys are subblock hashes.
32. The method of claim 28 wherein the index is implemented as a Btree.

33. The method of claim 28 wherein only every T'th subblock in each BLOB is indexed where T is a predetermined positive integer.
34. The method of claim 29 wherein only every T'th subblock in each BLOB is indexed where T is a predetermined positive integer.
35. The method of claim 28 wherein the index is implemented as one or more hash tables.
36. The method of claim 35 wherein a hash table entry for a subblock contains all or part of the hash of the subblock.
37. The method of claim 35 wherein the hash table contains buckets.
38. The method of claim 1 wherein each span refers to a finite sequence of one or more bytes within the cluster.
39. The method of claim 1 wherein each span contains at least one skip value x that indicates that the extent of the span is to be reduced by x bytes.
40. The method of claim 1 wherein each span contains at least one extension value x that indicates that the extent of the span is to be increased by x bytes.
41. The method of claim 28 including the step of:
checking for duplicate subblocks by checking the index before adding a subblock to a cluster.
42. The method of claim 29 including the step of:

checking for duplicate subblocks by comparing the hashes of subblocks to be stored with the hashes of at least one of the subblocks in a cluster where an index indicates a subblock is stored.

5 43. The method of claim 1 wherein spans identify a subblock using part or all of the hash of the subblock.

44. The method of claim 1 wherein at least one contiguous run of less than T present subblocks is duplicated in the store of subblocks, where T is a
10 predefined threshold of subblocks.

45. The method of claim 44 wherein T is two.

46. The method of claim 1 wherein at least one contiguous run of one or more
15 subblocks is duplicated in the store of subblocks.

47. The method of claim 1 wherein at least one span X is augmented with an alternative span that refers to a copy of the data referred to by span X.

20 48. The method of claim 28 wherein once the index has been used to find the location of a subblock X within a cluster, the cluster is searched forwards from subblock X to find the longest matching run of subblocks with the subblocks being stored.

25 49. The method of claim 29 wherein once the index has been used to find the location of a subblock X within a cluster, the cluster is searched forwards from subblock X to find the longest matching run of subblocks with the subblocks being stored.

50. A data processing apparatus for storing a BLOB of data comprising:
data processing means for dividing the BLOB into two or more subblocks;
data storage means for storing the subblocks in one or more clusters; and
for representing the BLOB as an ordered list of spans or a tree of spans,
5 where each span identifies a sequence of one or more contiguous subblocks
within a cluster and where at least one subblock is referred to by more than
one span.
51. A data processing apparatus according to claim 43 wherein the processing
10 means maintains an index that maps each subblock to the cluster containing
the subblock.
52. A data processing apparatus according to claim 44 wherein the processing
means checks for duplicate subblocks by checking the index before adding a
15 subblock to a cluster.
53. A computer readable memory, encoded with data representing a
computer program, that can be used to direct a programmable device for
storing a BLOB of data, comprising
20 processing means for operating the computer readable memory to divide
the BLOB into two or more subblocks;
data storage means usable by the computer readable memory for storing
the subblocks in one or more clusters; and for representing the BLOB as an
ordered list of spans or a tree of spans, where each span identifies a
25 sequence of one or more contiguous subblocks within a cluster and where
at least one subblock is referred to by more than one span.
54. A computer readable memory according to claim 53 wherein:

the processing means maintains an index that maps each subblock to the cluster containing the subblock.

55. A computer readable memory according to claim 54 wherein:

- 5 the processing means checks for duplicate subblocks by checking the index before adding a subblock to a cluster.

56. A computer program element comprising a computer program code means for storing a BLOB of data to make a programmable device execute:

- 10 a first function of dividing the BLOB into a plurality of subblocks;
a second function of storing the subblocks in a plurality of clusters; and
a third function of representing the BLOB as a group of associated spans, where each span identifies a sequence of one or more subblocks within a cluster and where at least one subblock is referred to by more than one
15 span.

57. A computer program element according to claim 56 wherein:

- a fourth function maintains an index that maps each subblock to the cluster containing the subblock.

20

58. A computer program element according to claim 57 wherein:

- a fifth function checks for duplicate subblocks by checking the index before adding a subblock to a cluster.

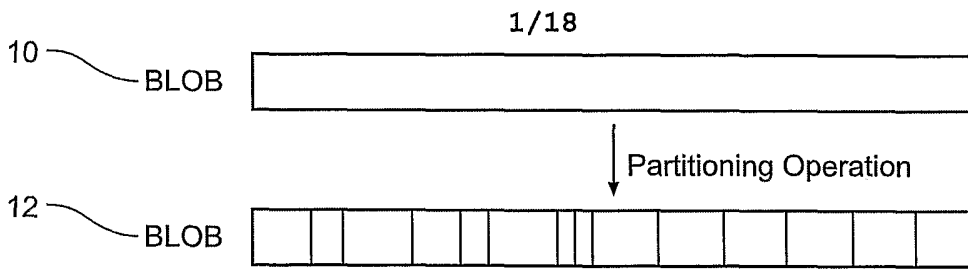


Figure 1

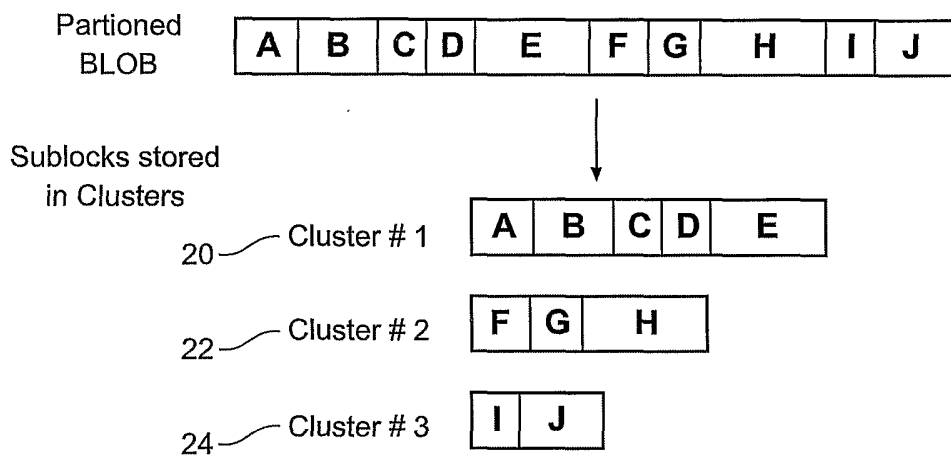


Figure 2

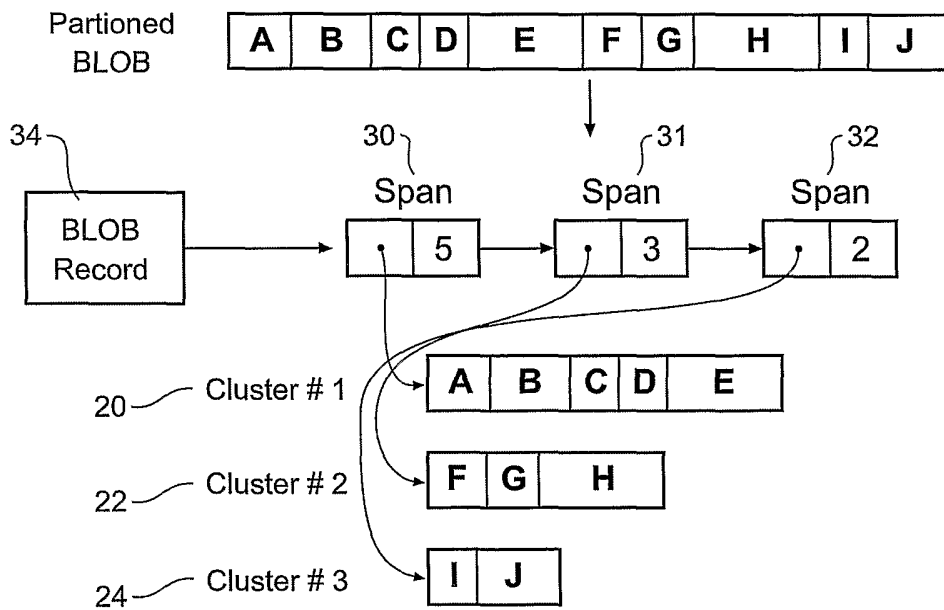


Figure 3

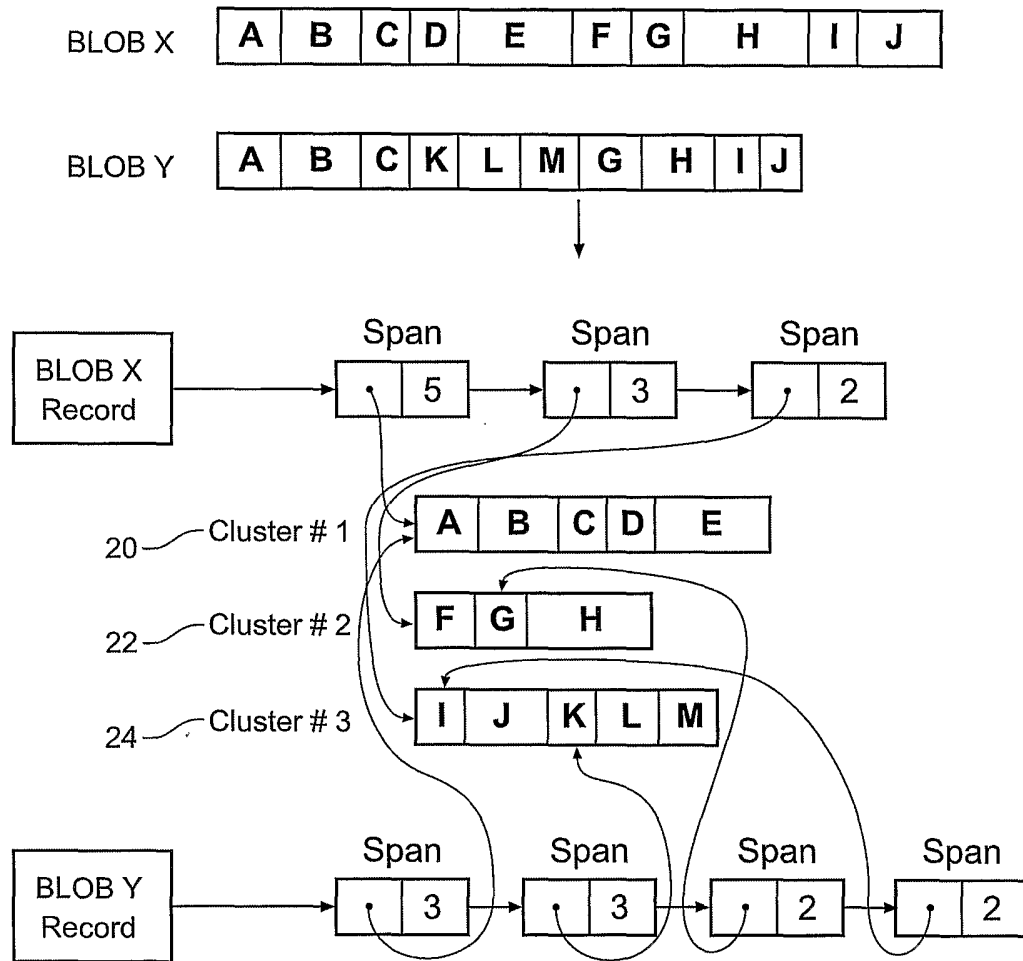


Figure 4

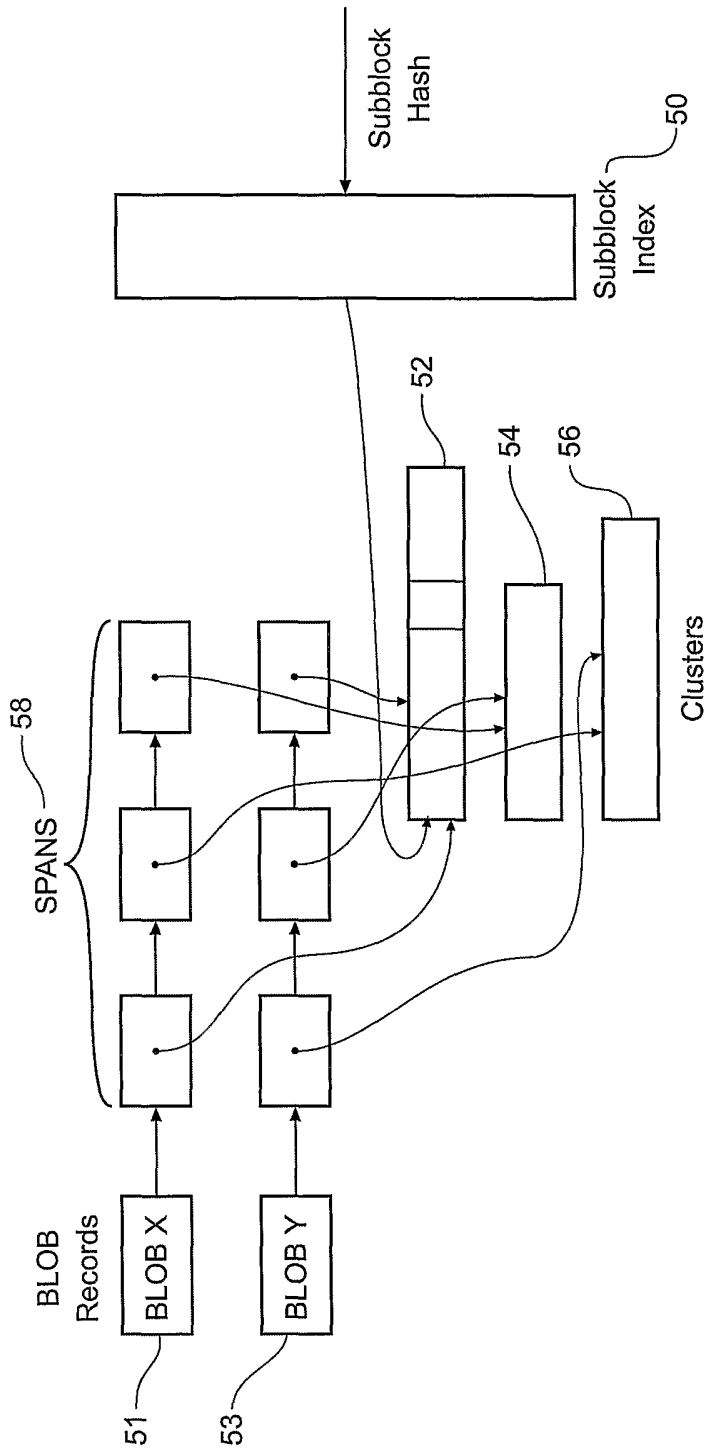


Figure 5

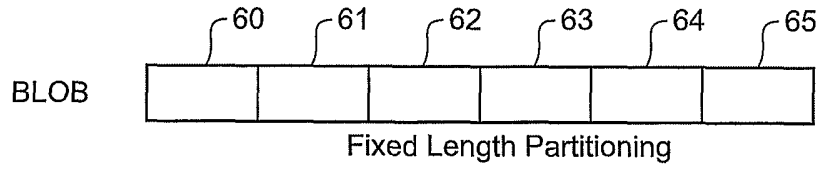


Figure 6

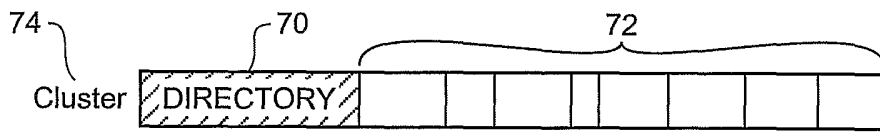


Figure 7

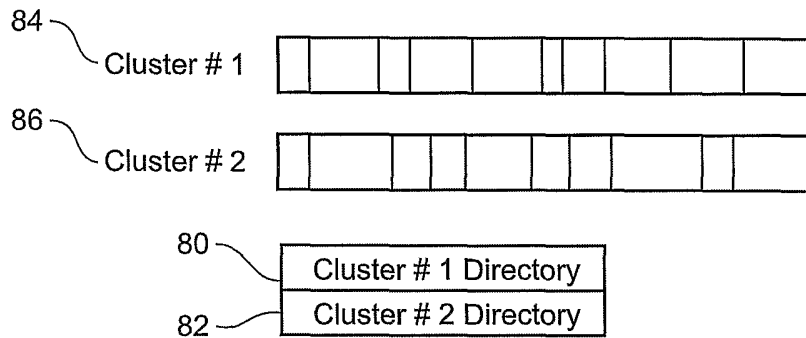


Figure 8

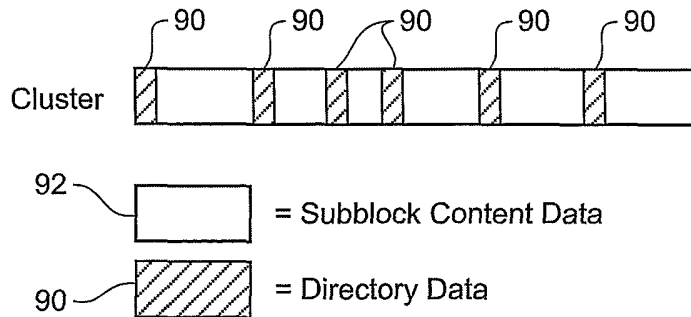
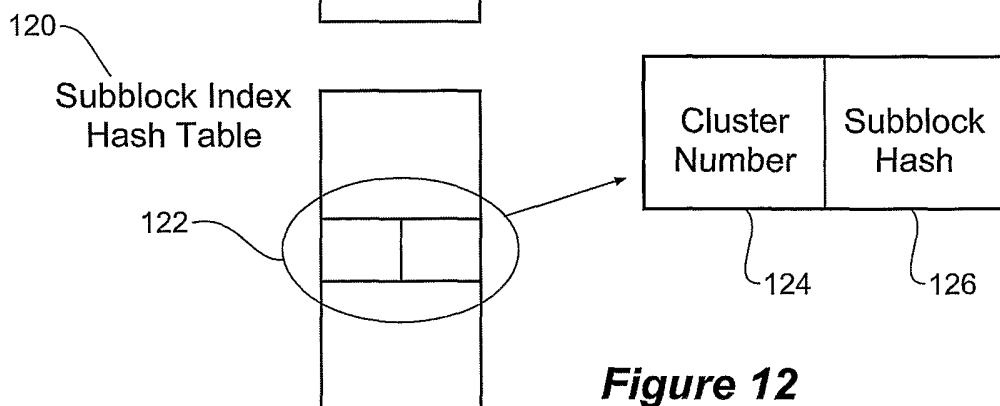
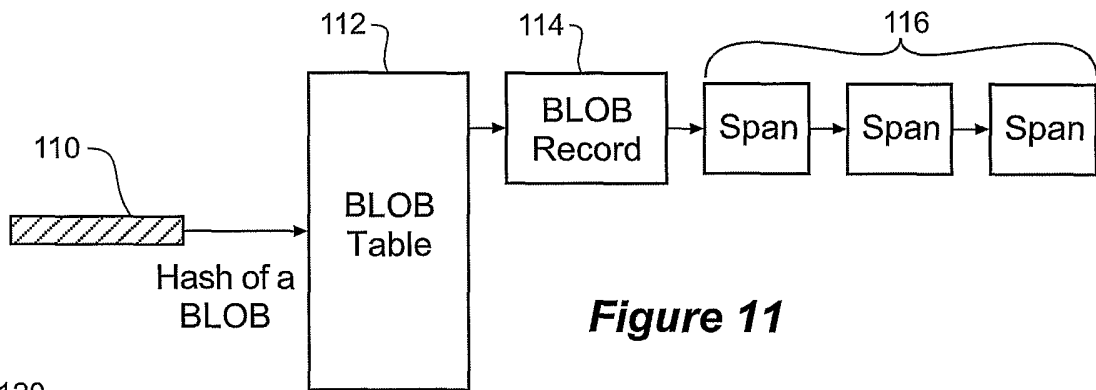
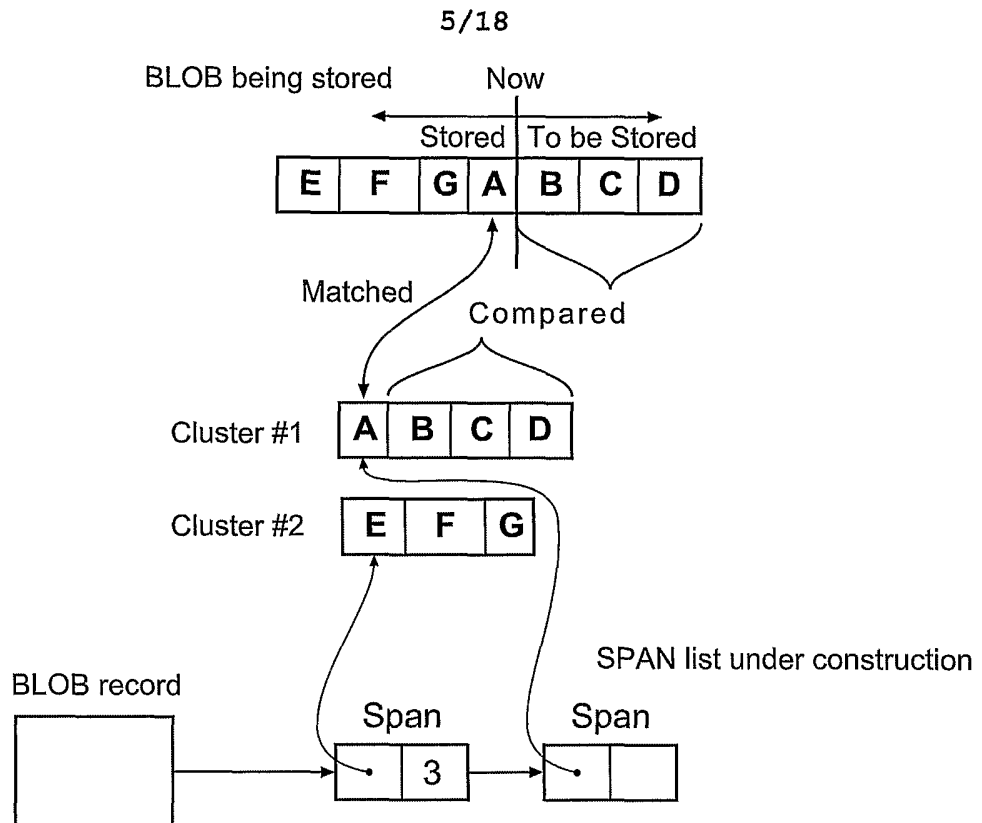


Figure 9



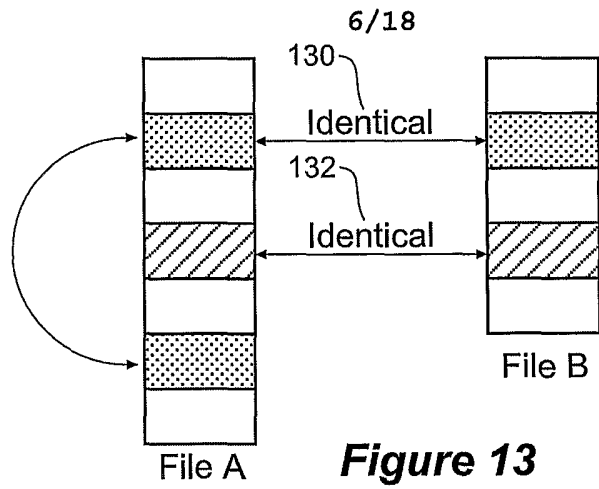


Figure 13

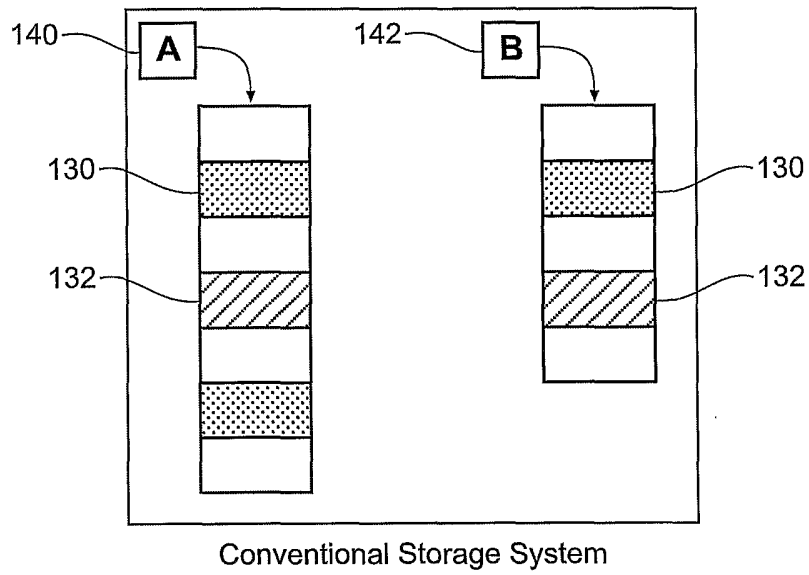


Figure 14

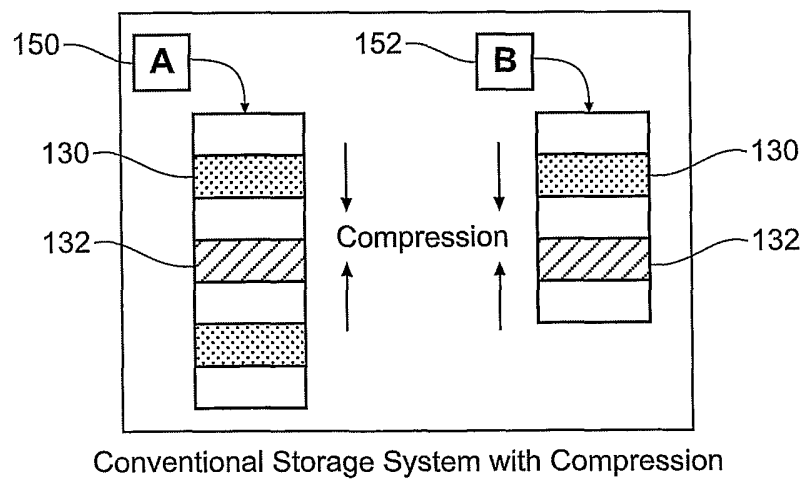


Figure 15

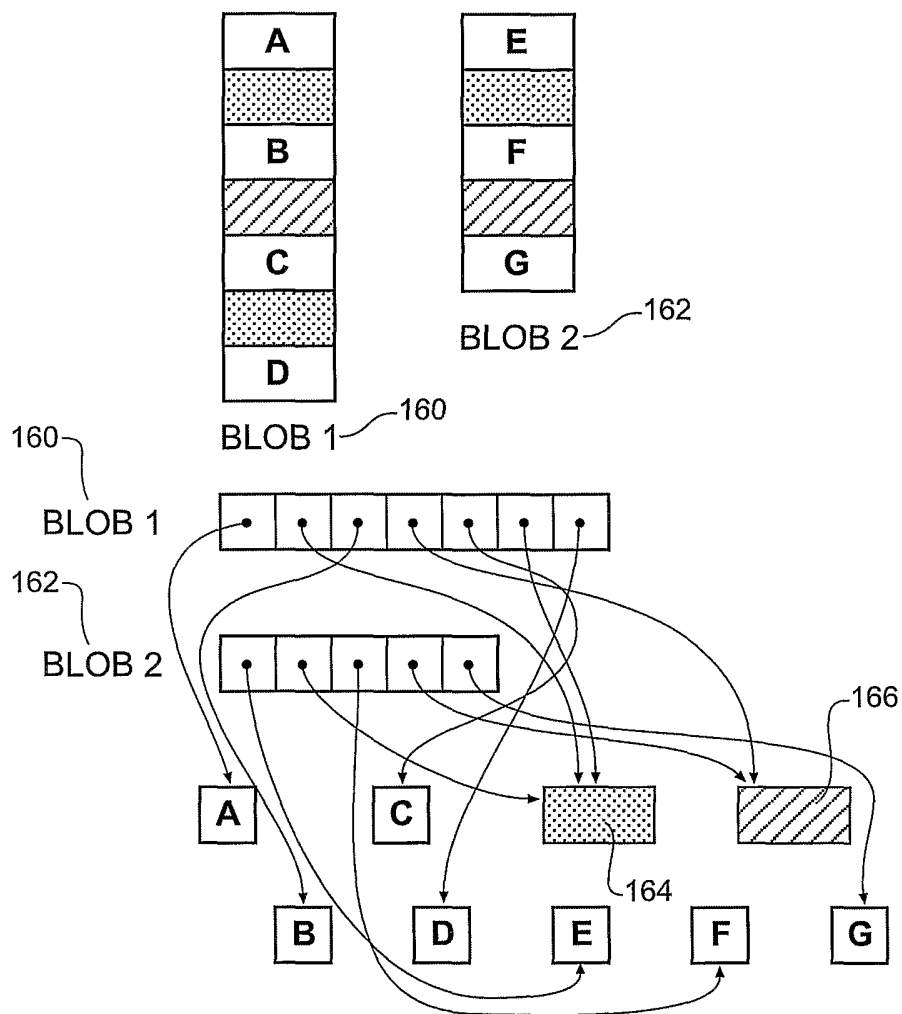


Figure 16

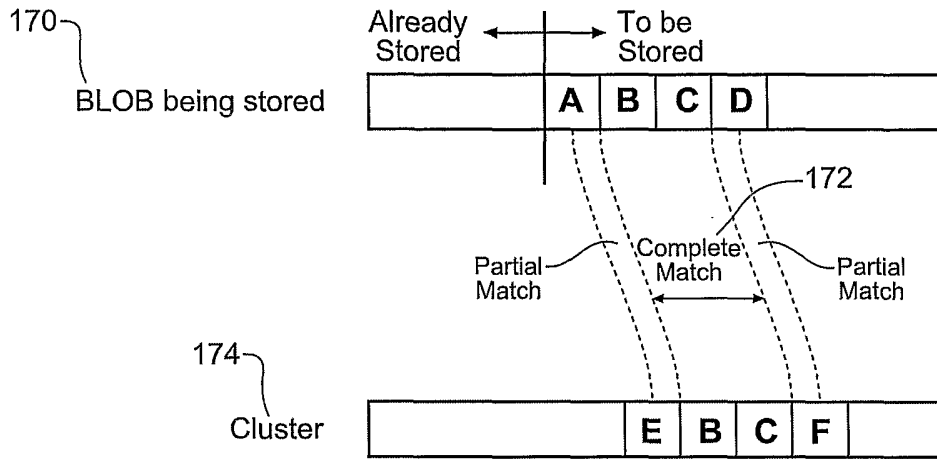


Figure 17

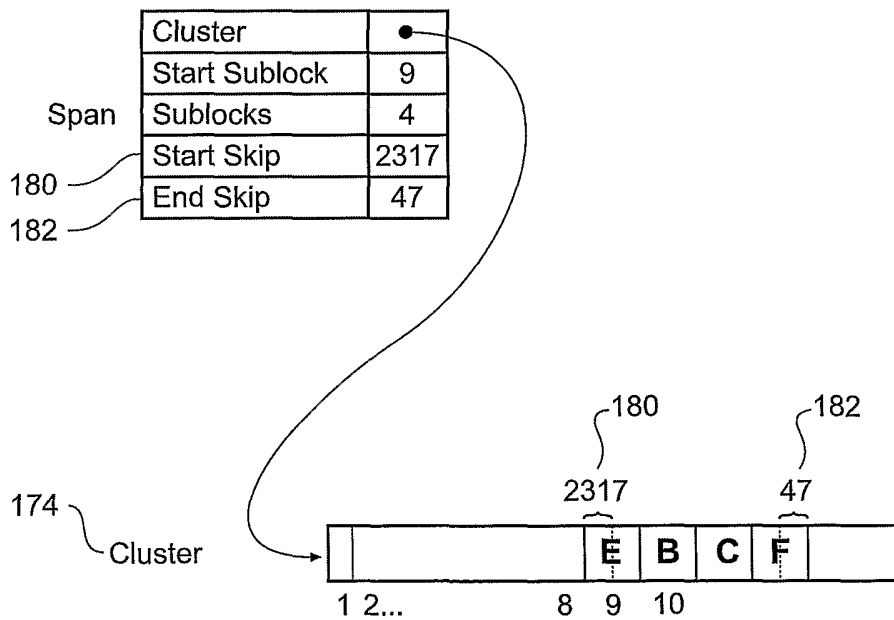


Figure 18

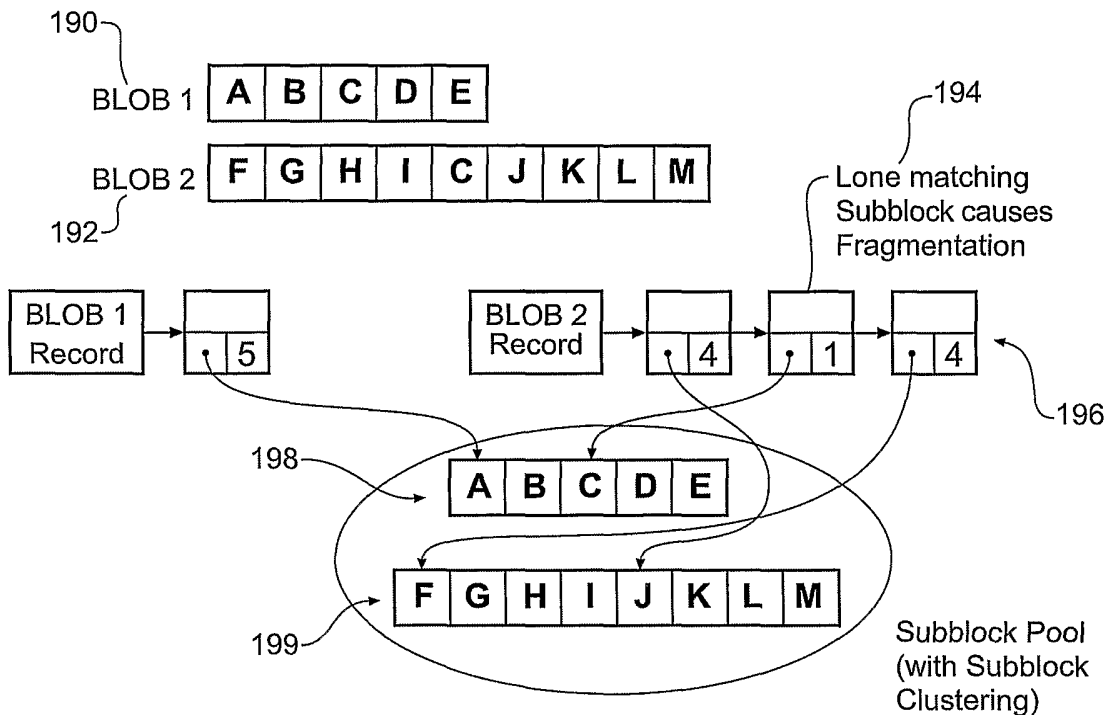


Figure 19

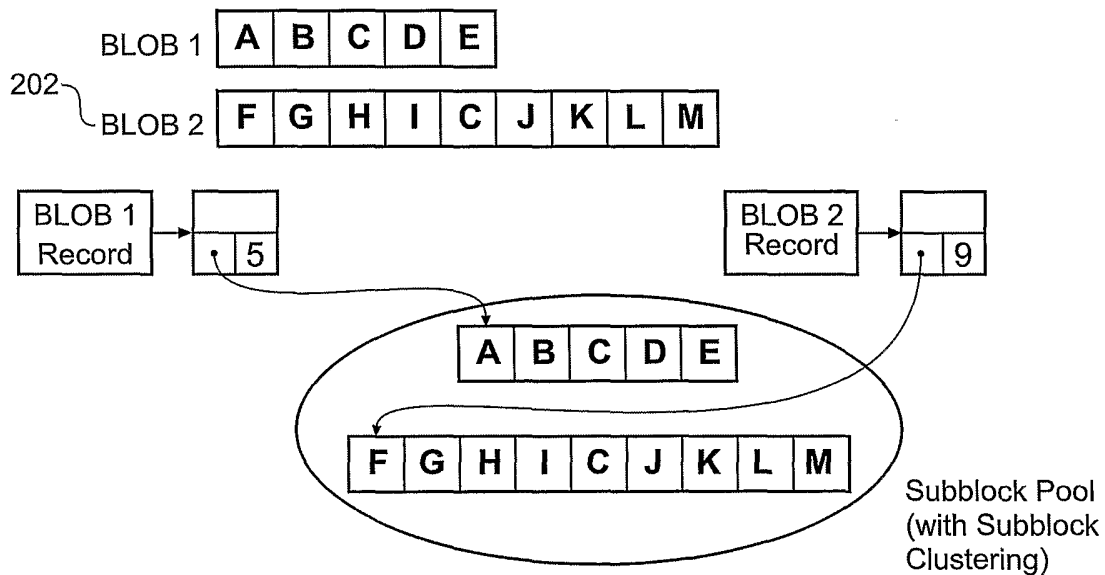
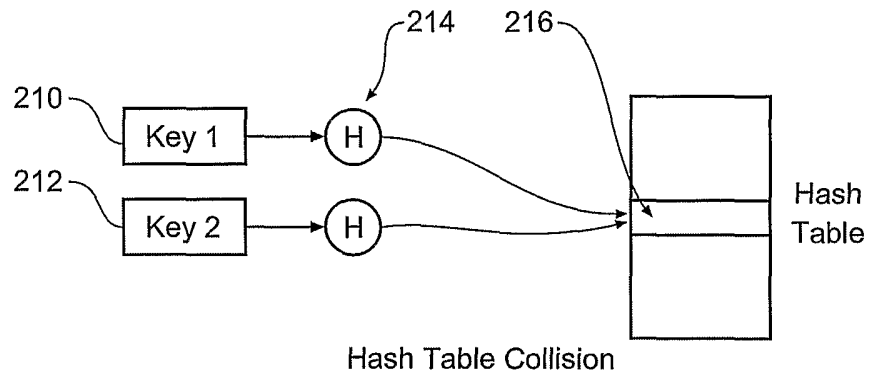


Figure 20



Hash Table Collision

Figure 21

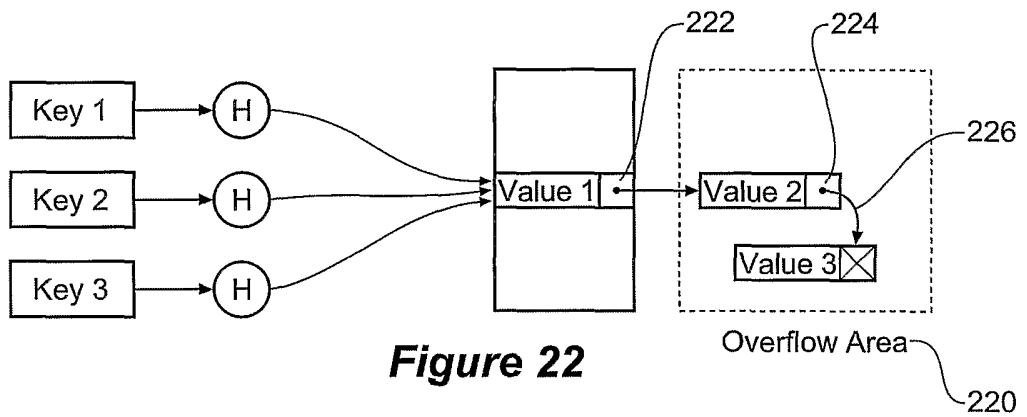


Figure 22

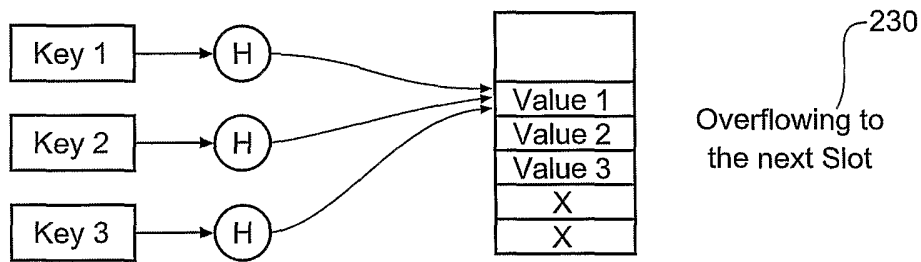


Figure 23

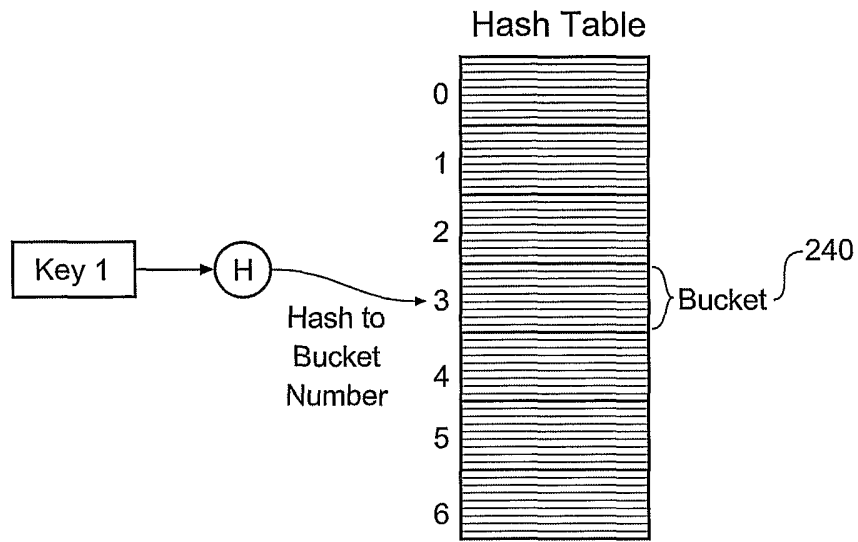


Figure 24

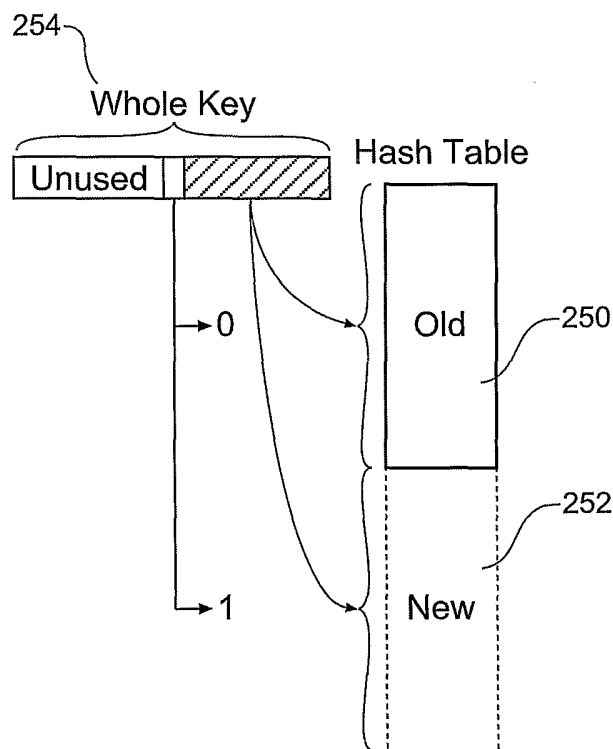


Figure 25

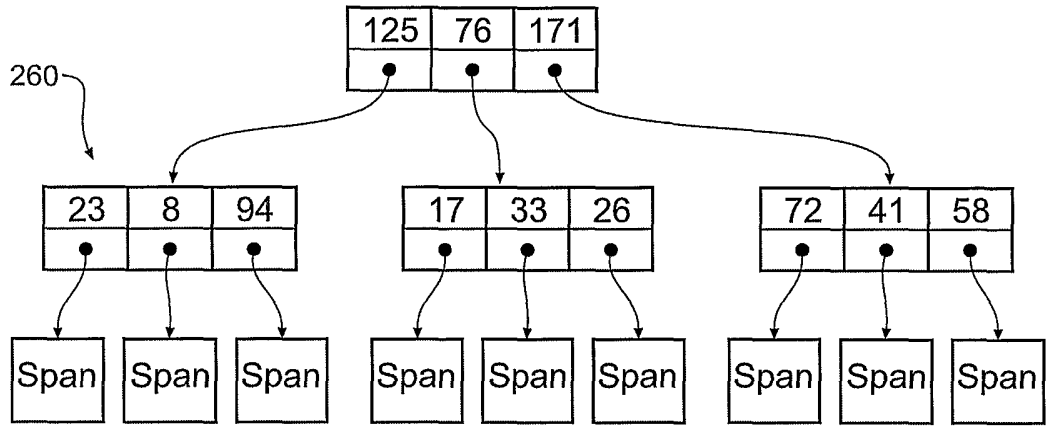


Figure 26

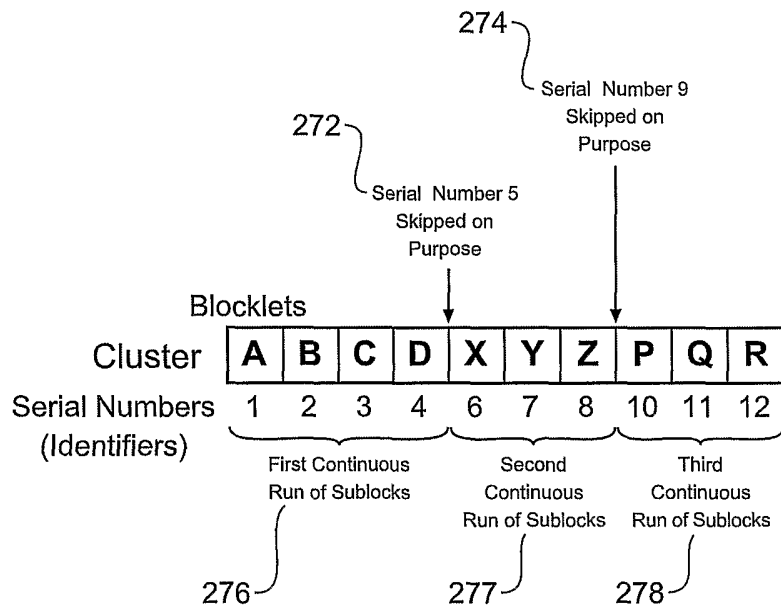


Figure 27

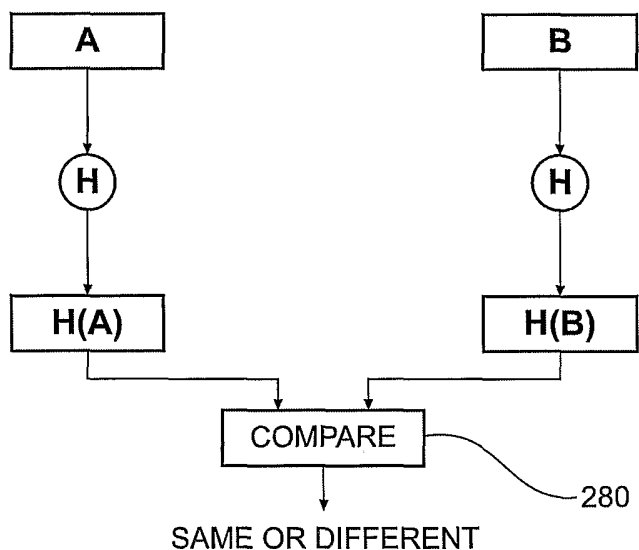


Figure 28

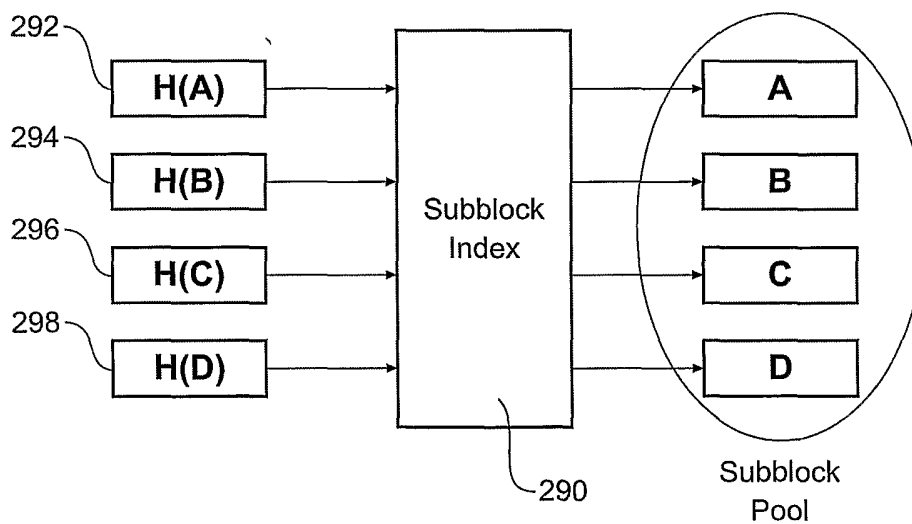


Figure 29

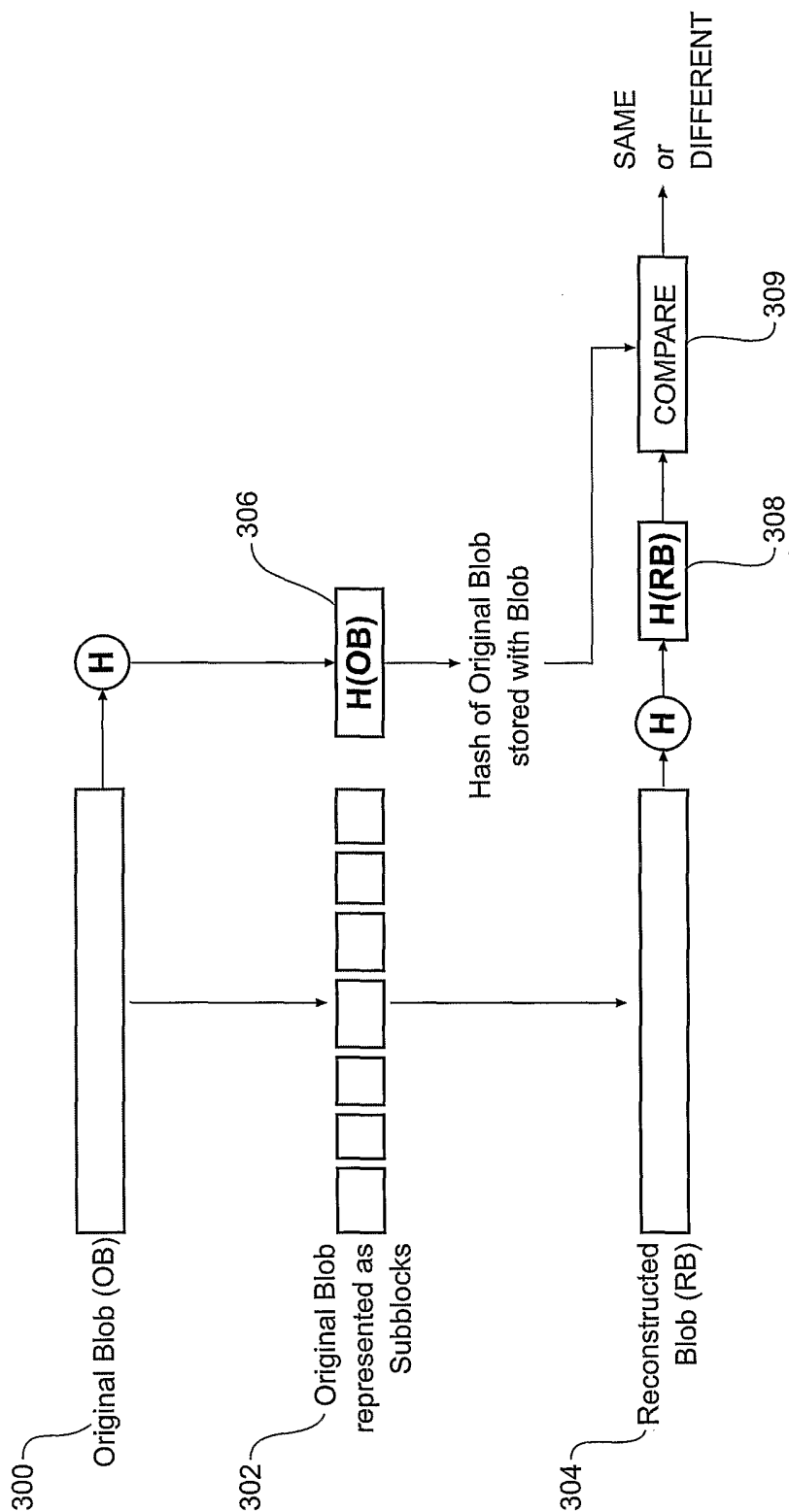


Figure 30

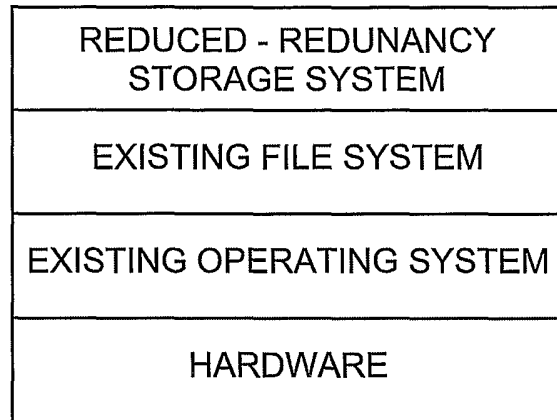


Figure 31

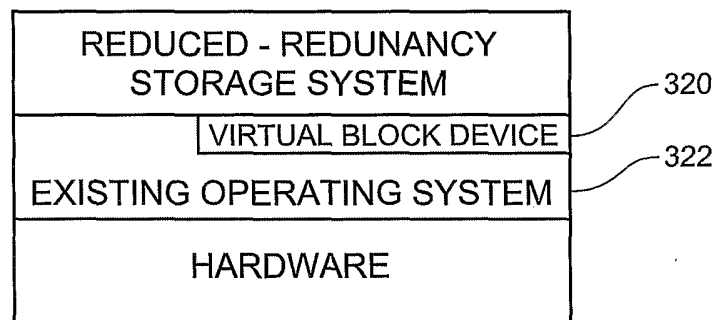


Figure 32

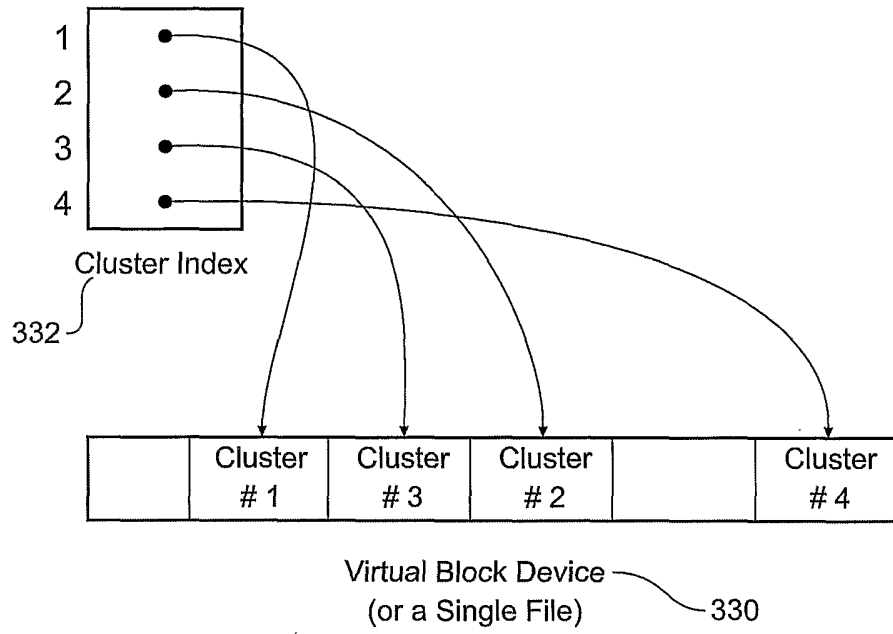


Figure 33

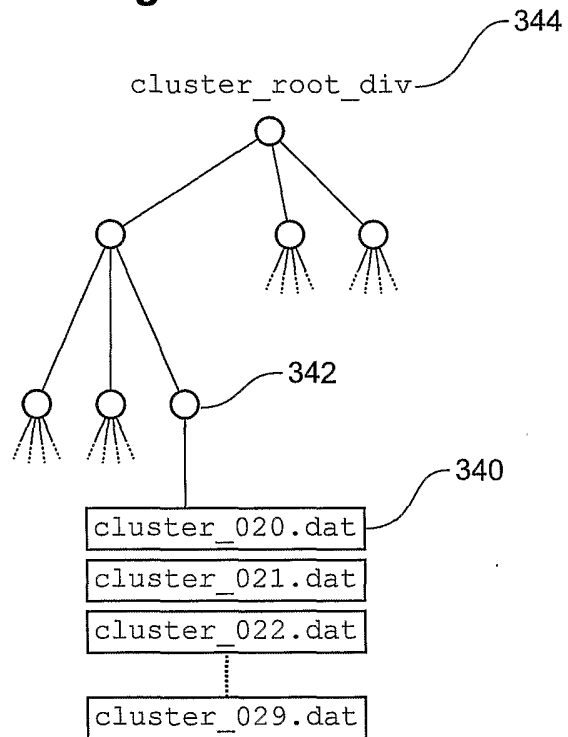


Figure 34

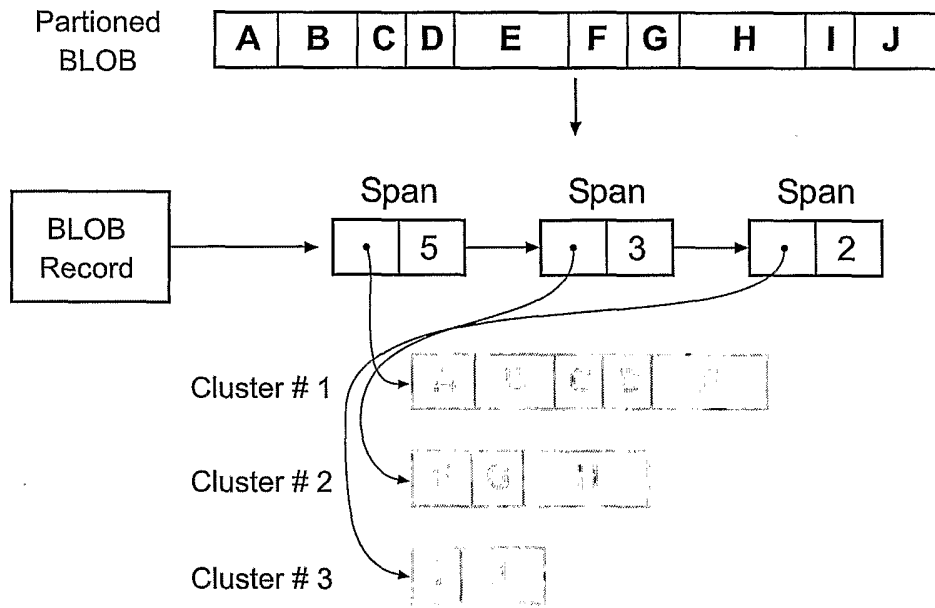


Figure 35

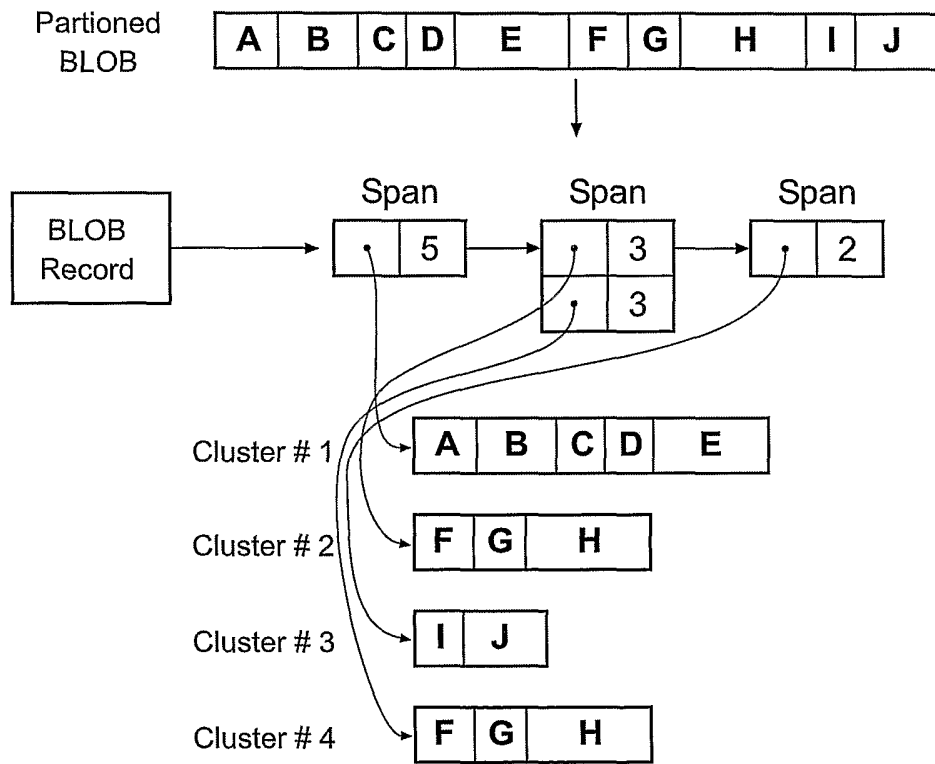


Figure 36

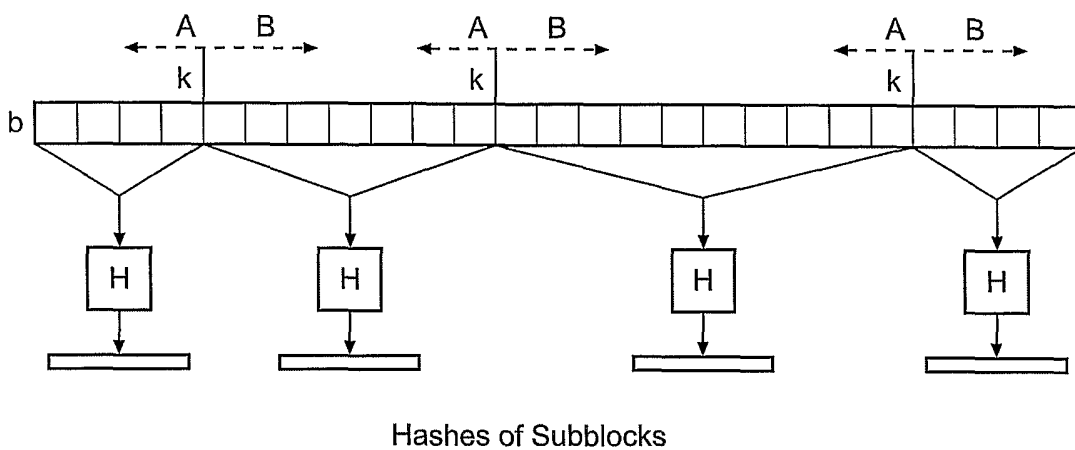


Figure 37

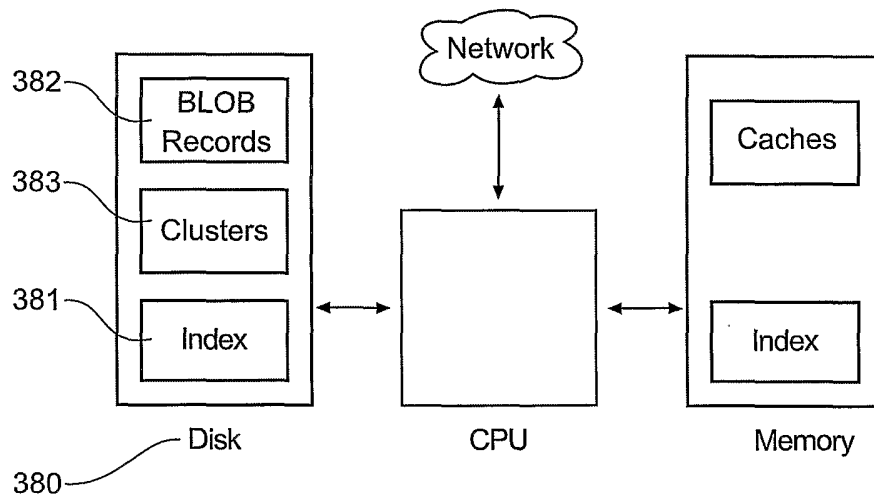
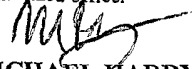


Figure 38

INTERNATIONAL SEARCH REPORT

International application No.

PCT/AU2006/000326

A. CLASSIFICATION OF SUBJECT MATTER		
Int. Cl.		
G06F 12/12 (2006.01)		
According to International Patent Classification (IPC) or to both national classification and IPC		
B. FIELDS SEARCHED		
Minimum documentation searched (classification system followed by classification symbols)		
Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched		
Electronic data base consulted during the international search (name of data base and, where practicable, search terms used) DWPI, IEEE, USPTO: G06F-012/ic, US class 711/, blob, block subblock, object, partition, subdivision, redundant, duplicate, repeat, hash, index, key, tree, directory, bitfilter, mask, flag, present, absent		
C. DOCUMENTS CONSIDERED TO BE RELEVANT		
Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US-6754799-B2 (FRANK) 22 June 2004 See whole document	1-58
A	US-6704730-B2 (MOULTON et al.) 9 March 2004 See whole document	1-58
A	US-6594665-B1 (SOWA et al.) 15 July 2003 See whole document	1-58
A	US-5990810-A (WILLIAMS) 23 November 1999 See column 22 line 15 to column 23 line 21 in context of the whole document	1-58
<input type="checkbox"/> Further documents are listed in the continuation of Box C <input checked="" type="checkbox"/> See patent family annex		
* Special categories of cited documents: "A" document defining the general state of the art which is not considered to be of particular relevance "E" earlier application or patent but published on or after the international filing date "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified) "O" document referring to an oral disclosure, use, exhibition or other means "P" document published prior to the international filing date but later than the priority date claimed "T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention "X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone "Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art "&" document member of the same patent family		
Date of the actual completion of the international search 01 June 2006		Date of mailing of the international search report 6 JUN 2006
Name and mailing address of the ISA/AU AUSTRALIAN PATENT OFFICE PO BOX 200, WODEN ACT 2606, AUSTRALIA E-mail address: pct@ipaustalia.gov.au Facsimile No. (02) 6285 3929		Authorized officer  MICHAEL HARDY Telephone No : (02) 6283 2547

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/AU2006/000326

This Annex lists the known "A" publication level patent family members relating to the patent documents cited in the above-mentioned international search report. The Australian Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

Patent Document Cited in Search Report		Patent Family Member					
US	6754799	US	2002178341				
US	6704730	AU	38189/01	AU	38267/01	AU	38269/01
		AU	41488/01	AU	43154/01	AU	49987/01
		AU	96665/01	CA	2399236	CA	2399522
		CA	2399529	CA	2399531	CA	2399555
		CA	2426577	EP	1266290	EP	1269316
		EP	1269325	EP	1269332	EP	1269350
		EP	1344321	US	6810398	US	6826711
		US	7000143	US	2001034795	US	2001037323
		US	2001042221	US	2001044879	US	2002010797
		US	2002048284	US	2002152218	US	2004148306
		US	2004225655	US	2005022052	US	2005120137
		WO	0161491	WO	0161494	WO	0161495
		WO	0161507	WO	0161518	WO	0161563
		WO	0237689				
US	6594665						
US	5990810	AU	46593/96	WO	9625801		
Due to data integration issues this family listing may not include 10 digit Australian applications filed since May 2001.							
END OF ANNEX							