1. What is the purpose of the split matrix in Natix? Give an example of its use in Natix.

   Proposed answer.
   In Natix, with split matrix we can then determining the insertion location to improvement storage & retrieval performance.

   Note that with the split matrix the user can enforce splitting a document into records such that those parts of the document that are likely to be modified concurrently reside in different records. As we will see, concurrent updates on different records are possible.

   Hence, a high level of customized concurrency is possible while avoiding an excessive amount of locks.

2. In 2006, eXist changed to a new indexing scheme and says it can support dynamic insertion without renumbering the nodes. Describe the new scheme and why the dynamic insertion works.

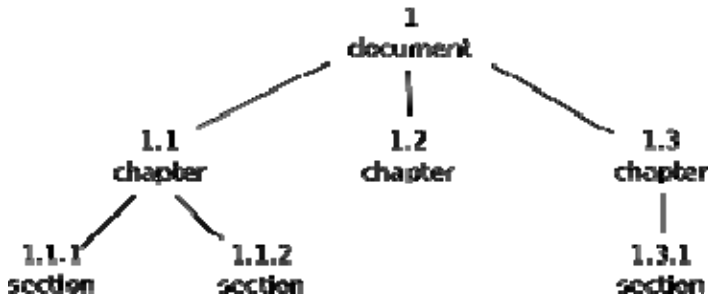
   Proposed answer.

   Their new scheme called dynamic level numbering introduced level info into the node label so that insertion at a certain level does not affect other nodes' index. The description of the new scheme is below.

   In early 2006, we finally started a major redesign of the whole indexing core of eXist. As an alternative to the old level-order numbering, we chose to implement "dynamic level numbering" (DLN) as proposed by Böhme and Rahm (Böhme, T.; Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), 2004). This numbering scheme is based on variable-length ids and thus avoids to put a conceptual limit on the size of the document to be indexed. It also has a number of positive side-effects, including fast node updates without reindexing.

   Dynamic level numbers (DLN) are hierarchical ids, inspired by Dewey's decimal classification. Dewey ids are a sequence of numeric values, separated by some separator. The root node has id 1. All nodes below it consist of the id of their parent node used as prefix and a level value. Some examples for simple node ids are: 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3. In this case, 1 represents the root node, 1.1 is the first node on the second level, 1.2 the second, and so on.

Figure: Document with DLN ids



Using this scheme, to determine the relation between any two given ids is a trivial operation and works for the ancestor-descendant as well as sibling axes. The main challenge, however, is to find an efficient encoding which 1) restricts the storage space needed for a single id, and 2) guarantees a correct binary comparison of the ids with respect to document order. Depending on the nesting depth of elements within the document, identifiers can become very long (15 levels or more should not be uncommon).

The original proposal describes a number of different approaches for encoding DLN ids. We decided to implement a variable bit encoding that is very efficient for streamed data, where the database has no knowledge of the document structure to expect. The stream encoding uses fixed width units (currently set to 4 bits) for the level ids and adds further units as the number grows. A level id starts with one unit, using the lower 3 bits for the number while the highest bit serves as a flag. If the numeric range of the 3 bits (1..7) is exceeded, a second unit is added, and the next highest bit set to 1. The leading 1-bits thus indicate the number of units used for a single id. The following table shows the id range that can be encoded by a given bit pattern:

ID ranges

No. of units	Bit pattern	Id range
1	0XXX	1..7
2	10XX XXXX	8..71
3	110X XXXX XXXX	72..583
4	1110 XXXX XXXX XXXX	584..4679

The range of available ids increases exponentially with the number of units used. Based on this algorithm, an id like 1.33.56.2.98.1.27 can be encoded with 48 bits. This is quite efficient compared to the 64 bit integer ids we used before.

Besides removing the document size limit, one of the distinguishing features of the new indexing scheme is that it is insert friendly! To avoid a renumbering of the node tree after every insertion, removal or update of a node, we use the idea of sublevel ids also proposed by Boehme and Rahm. Between two nodes 1.1 and 1.2, a new node can be inserted as 1.1/1, where the / is the sublevel separator. The / does not start a new level. 1.1 and 1.1/1 are thus on the same level of the tree. In binary encoding, the level separator '.' is represented by a 0-bit while '/' is written as a 1-bit. For example, the id 1.1/7 is encoded as follows:

0001 0 0001 1 1000

Using sub-level ids, we can theoretically insert an arbitrary number of new nodes at any position in the tree without renumbering the nodes.

The reference for the above is at http://exist-db.org/exist/xmlprague06.xml?q=.%2F%2Fsection%5Bft%3Aquery(.%2F%2Ftitle%2C%20'indexing')%5D%20or%20.%2F%2Fsection%5Bft%3Aquery(.%2C%20'indexing')%5D&id=2.4#2.4