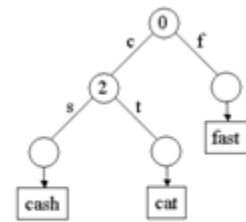


Technology

The Index

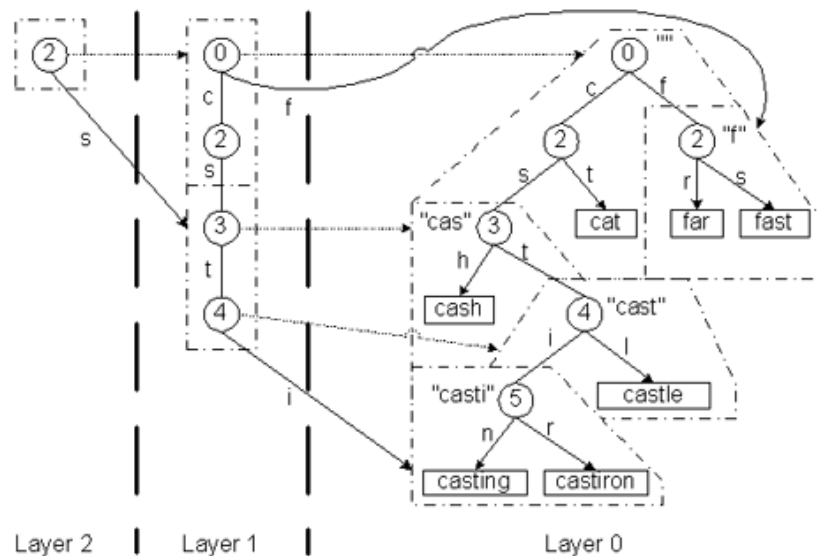
The ScaleDB technology is based on a compressed form of tries called Patricia tries. A Patricia trie differs from a standard trie in that nodes with one child are compressed into their parent node, so that all nodes have at least two children. Patricia tries achieve a high level of compression as the size of the Patricia trie does not depend on the length of inserted keys. Rather, each new key adds at most a single link and node to the index, regardless of the actual key length. Furthermore, unlike B-trees, Patricia tries grow slowly even as large numbers of strings are inserted because of the aggressive (lossy) compression inherent in the structure.

Although researchers have long known about Patricia tries, such structures have rarely been used to manage large amounts of data, especially disk-based data, because they are unbalanced and best suited for usage in main memory. What is needed is a structure that has the graceful scaling properties of Patricia tries, but is also balanced and optimized for disk-based access like B-trees. The ScaleDB is such a structure. It uses a novel layered trie approach to transform a Patricia trie into a disk-based index: extra layers of Patricia tries allow a search to proceed directly to a portion of the index that can answer a query. Every query accesses the same number of layers, providing balanced access to the index. We can think of these extra layers as a horizontal index complementing the vertical structure of the original Patricia trie (which resides in layer 0).



A Patricia tries

The search process examines one block per horizontal layer. In a disk-based index this means that the search could require one I/O per layer, unless the needed block is in the cache. The vast majority of space required by the index is for the original, vertical Patricia trie (layer 0), the other horizontal layers (layer 1,2,...n) are significantly smaller. In practice, this means that an index storing a large number of keys (e.g. a billion) requires three layers; layer 0 must be stored on disk but layers 1 and 2 can be easily stored in main memory. Key lookups require at most one I/O, for the leaf index layer (in addition to data I/O's).

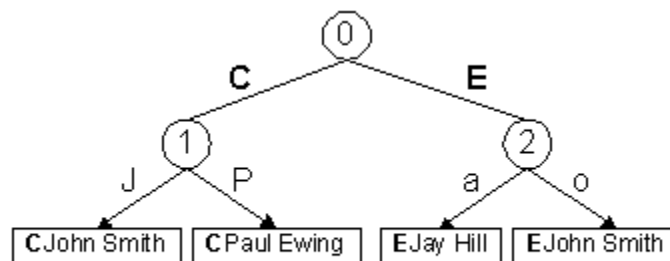


A Layered Patricia trie

Designators

ScaleDB introduces the concept of designators to solve the problem of intermixing of unrelated search paths. Designators are special symbols or strings of symbols that are placed within key strings for the express purpose of distinguishing between key strings belonging to different search paths, while at the same time grouping together related key strings. In other words, designators are simply parts of key strings that are generated based upon some rule associated with the corresponding key, and not based upon the particular key value for the corresponding object. For the purpose of constructing the trie structure, designators are like any other part of key strings and do not require a change in any of the algorithms that operate on the structure.

As an example of the use of designators, using an example of employee and customer data, the employee name and customer name keys might use the designators “E” and “C”, respectively. Thus, the key string for employee “John Smith” would be “EJohn Smith”, while the key string for customer “John Smith” would be “CJohn Smith”. The figure below shows a simple example with four key strings, two using the designator “E” and two using the designator “C”. When searching for employee “John Smith”, we would use the query string “EJohn Smith”, and therefore would only encounter the key string for employee “John Smith” during the search traversal and not the key string for customer “John Smith”.



Patricia trie for four key strings that use two designators, “C” and “E”

As demonstrated by the example, designators ensure that related key strings fall into the same subtrie. For example, both customer key strings, designated with “C”, fall into the left subtrie of to root node. At the same time, no employee key string can exist in the left subtrie, since such key strings are prepended by the designator “E”. Hence, we see that to be useful, designators must occur in the very beginning of key strings, and that this is a direct consequence of the fact that tries essentially index based on prefixes of strings. Actually, as shown below, it is often meaningful to use multiple designators in a single key string. In such cases, the key strings are composed of several components, each of which starts with a designator. We use the term designated key component to refer to components of keys whose values map into portions of key strings prepended with a designator. For example, for a composite “name-address” key for a “person” collection, we might designate the “name” key component with “N” and the “address” key component with “A”, in which case the key strings would have format “N<name>A<address>”.

Relationships Between Key Types and Designators

A key type essentially refers to a rule for constructing a key string for an object belonging to a particular collection, namely the designated key components and designators that are included in the key string, e.g., as for the above “name-address” key example. Thus, each key type is related to one or more designators. However, each designator may be shared by more than one key type, in cases where we want the same search path to potentially lead to objects in different collections, so the relationship is really “many-to-many”. For example, if we want a search on “name” to lead to both “employee” objects and “customer” objects, we might use the “N” designator for both the “employee.name” and “customer.name” keys. Perhaps the most important use of this feature would be for data integration, e.g., for allowing search access to two customer databases with very different schemas without actually merging the two.

Indexing Relationships

The ability to index long strings can be exploited to index relationships between objects, simply by concatenating the keys of the objects. For example, say we have a “customer” collection and an “invoice” collection, where each invoice is related to a single customer. This relationship can be captured by indexing the composite key “customer.ID-invoice.ID”, where the “ID” key in each collection is some unique identifier. This key would provide a search path through which we can, for example, find all invoices by a certain customer.

The example above is of a “one-to-many” nature, since one customer can relate to many invoices. “One-to-one” and “many-to-many” relationships can be represented in a similar manner. For example, a “product” collection and the “invoice” collection would have a “many-to-many” relationship, since each product can occur in multiple invoices and each invoice can contain multiple products. We might want to capture this relationship through both a “product.ID-invoice.ID” and a “invoice.ID-product.ID” composite keys, so that we can locate all invoices containing a particular product and all products in a particular invoice, respectively.

Relationships such as those mentioned above can be represented in various data representations, e.g., based on the relational data model. However, the most natural representation is one similar to the hierarchical or object-oriented data models, wherein all objects participating in a relationships can be located without further index lookups. Thus, in such a representation, when an object o1 is “subordinate” to an object o2 in a “one-to-one” or “one-to-many” relationship, o1 essentially incorporates a reference to o2 that allows accessing it. For example, in the case of “customer” and “invoice” example, each “invoice” object would contain a reference to the corresponding “customer” object.

For “many-to-many” relationships, such as the “product-invoice” relationship, it is not sufficient to store references in the objects themselves, since each participating object can be related to multiple objects. Instead, we must use some sort of “join” objects that contain references to all objects participating in a relationship. Thus, for example, for the “product-invoice” relationship, we would have “join” objects that represent a product occurring in a particular invoice. This “join” object can then be indexed according to both composite keys mentioned above, and would allow accessing the associated “product” and “invoice” objects.