

The JS Storage Engine

Alon Itai* and Moshe Shadmon**

*CS Department
Technion
Haifa, Israel
alon@cs.technion.ac.il

**Ori Software Development Ltd.
San Jose, CA
moshe@orisoftware.com

Abstract

Database systems view data in different ways. Some regard the data as nodes in a network, others as tuples in a table and others as abstract objects. There are a number of traditional techniques for implementing and indexing each of these database systems. For example, the B-tree is a commonly used index structure. The JS storage engine is a novel way of viewing and representing data. This technology deals with all keys as character strings, which are stored in a layered index (a trie-like data structure). With this technology we can:

- Search for an item by any of its keys, within a small number of disk accesses.
- Quickly access logically related items.
- Ensure referential integrity.
- Efficiently implement complex operations such as joins.
- Efficiently process the data sequentially according to any of its keys.
- Simultaneously view the same physical implementation by several database models (relational, object-oriented, etc).

The JS therefore allows a great deal of flexibility when managing data, including the ability to view the same data in many different ways, while retaining high performance.

1. Introduction

A database model is way to view and interpret data. For example, the relational model organizes data in tables (called relations), the network model views data as a set of nodes, connected by links, and the object oriented model considers data as a set of objects of a finite set of classes. In addition, the models intrinsically support languages for queries: declarative, abstract methods to access the data. The relational model, for instance, defines a relational algebra, and programming languages such as SQL implement it. The database model and query language are the way the data is presented to the user by the data management system.

In addition to a data model, a database system must have a storage engine: a method of storing data on disk and accessing it. In order to support fast searches, the storage engine usually contains a set of indexes in addition to the data. In most current systems the index is implemented as B-trees (or sometimes a hash tables). The database management system (DBMS) maps the database model to the storage engine, providing a means to access and perform operations on the data. A DBMS over the relational model should, therefore, provide a compiler (or interpreter) for the query language that maps queries of the query language to operations on the storage engine. The most popular example is the mapping of SQL queries to physical query plans, which may include B-tree operations.

The JS storage engine replaces both the index and other search paths with a new structure. This allows us to store and quickly search for an entire record and simultaneously all of the properties of a data item.

Moreover, the new technology allows us to search for any key in a uniform manner, regardless of the type of record to which it belongs. Also, the structure reflects the order between keys, thus enabling efficient sequential processing.

The storage engine provides the link between data items that are logically related, enabling the system to easily find additional attributes and properties, once an item is located. In this respect, the system is reminiscent of the network model. However, it does not suffer from the problems that plagued network based systems: non-uniform treatment of data, and long search paths. The layered index allows the database system to traverse any search path in logarithmic time. Thus, while in the network model, one might have to follow a long path to reach the desired item; in the JS the item is usually reached within a single access.

An additional feature is that the view of the data need not be fixed in advance. In relational database terminology, one can view the same data as several distinct tables simultaneously. The JS engine storage provides any number of views of the same data. With the JS, the views can be determined dynamically by the application traversing the index in an appropriate way.

The uniform method of storing the data and the index automatically ensures the referential integrity of data, since the access to a data node is physically linked to the data node of the referred field. When inserting an item, links to referred items are automatically introduced. If a referred item is missing, the insertion process fails, thus guaranteeing referential integrity for existence. Similarly, on updates or deletes, referential integrity constraints based on data existence can be immediately checked since relationships between objects are kept explicitly.

More complex (and costly) operations such as aggregation and joins can also be implemented more efficiently than with B-trees. For example, finding the average balance of the accounts of a bank branch B does not require us to scan the entire database to perform the selection. Rather, the node representing B acts as a starting point to find all the accounts of the branch, and only those accounts are accessed.

The uniform mechanism for accessing the JS, regardless of the underlying structure of the data, allows us to treat the data using any appropriate model. For example, the index can deal with data that is object oriented. When a new relation is created as a result of a query (say a join), we can easily inherit from its parent relations the methods needed to access the attributes of the new relation. (This is because the index keeps relationships between classes explicitly, allowing us to examine the properties and methods of the ancestor classes while we are evaluating the query.) Therefore, the methods inherited by the new relation are provided automatically. Additional class specific methods can also be defined for the new relation. Moreover, the JS allows objects themselves to be complex structures such as tables. These features allow the index to function seamlessly as an object oriented data manager. The same physical implementation can provide for more than one data model, i.e., the same physical structure supports the OO model, the relational model and the hybrid object-relational model.

To summarize, the storage engine is a novel structure that has the following advantages:

- Any data item can be reached within one to two disk accesses.
- Data is treated uniformly, regardless of the relation it belongs to.
- The space required by the index is considerably less than that required by traditional engines (B-tree).
- There are links between logically related data items, thus reducing the time needed for complex queries.
- Referential integrity is automatically supported at the engine level.
- The Object-Relational model is fully supported.
- The same physical implementation can simultaneously be viewed as several data models.

1.1. Outline

Our data structure, the layered index, is essential to understanding how data is represented and is a key to the efficiency of the model. Its main features are described in Section 2. Section 3 describes how data is represented, while Section 4 analyzes the efficiency of the model and describes how complex operations are performed. Section 5 explains how to maintain referential integrity and support multiple keys. Section 6 shows how the same physical structure can simultaneously support several data models. Section 7 concludes with a summary of the highlights of the JS method.

2. The layered index data structure

The JS engine employs a sophisticated data structure, the layered index, which is described in detail in [2]. We present a short description of the layered index here. The basic structure of the layered index is a Patricia trie (PT) (see Knuth [3]). The PT is a condensed form of a trie that has the following properties:

1. A PT is a data structure whose keys are character strings that support searches as well as sequential processing. The data structure is dynamic in the sense that it allows insertions and deletions without a complete rebuild.
2. Searching a PT T for a key $k \in T$ leads to a record with key k . If $k \notin T$ then the search either fails, or leads to a record r with where $\text{Key}(r) \neq k$. Thus when the search arrives at a record r , one must check whether $\text{Key}(r) = k$.
3. Each PT non-leaf node has at least two children. To provide an index to n keys one would need n links and no more than $n-1$ fixed sized nodes. In many practical situations, the PT has far fewer nodes. Thus the storage requirement of a PT is independent of the length of its keys.

4. A PT is not necessarily balanced; a PT with n nodes might contain a path of length n . Thus searching a PT might require time $\theta(n)$.

Figure 1 depicts a PT of a library that has 7 books. The numbers inside the nodes indicate the depth of the node in a full trie: i.e., the position of the examined character in the string. Because in a PT not all positions are examined, a search for a key *not* in the PT might lead to a different record. For example, searching for “Gun-smoke” leads to the record “Gone with the wind”. So on reaching a record we must check to verify that we have reached the right one.

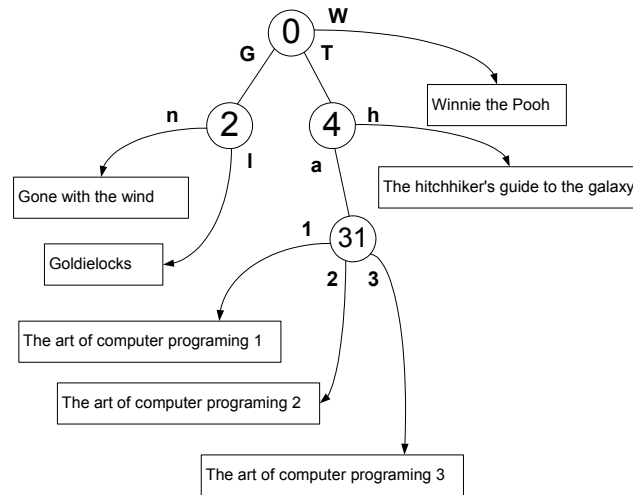


Figure 1. A Patricia trie for a library.

Large databases contain a large number of records whose keys do not fit in main memory and must be stored on the disk. Using a PT to provide access to the keys might be inefficient since a PT search path of the length n could require n disk accesses (in the worst case). Our goal is to provide balanced access to the PT. We refer to the basic PT as the *vertical trie*, and add additional *horizontal layers* to provide this balance. We partition the vertical PT into blocks such that each block contains a subtree. This is the first “layer” of the index.

The second layer is a PT built over the common prefixes of all the keys reachable from each block of the first layer. Thus, the second layer acts as a horizontal complement to the vertical PT in the first layer. We can construct another horizontal layer to index the blocks of the second layer. In fact, we can create as many layers as necessary, until the n^{th} horizontal layer contains just a single block. The number of layers of an index to a file of N records is $h = \log_B N$, where B is the average number of PT-nodes in a disk page.

For example, if an 8KB block has about 1000 outgoing pointers, it follows that a PT with 10^9 records is indexed by a 3-layer structure. Although layer one (containing the original vertical trie) is large, the other

two layers would only require about 8MB, and could be accommodated in main memory. Thus, one can access a record with practically one (index) disk access.

Some techniques used by B-tree implementers to reduce the size of the index and the number of disk accesses can be used for layered indexes too:

- Increasing the block size reduces the size of the index and the number of disk accesses required for a search.
- The index of the primary key can be implemented as a sparse index, which requires only one pointer per block, rather than one pointer per record, thus drastically reducing the size of the index.

To summarize, the layered index provides a very compact data structure that allows one to search and update long character string keys within one to two disk accesses [2].

The remainder of the paper describes how to efficiently represent data using the layered index. To simplify the exposition, we will describe the representation as if it used a PT. One should bear in mind, however, that the real implementation uses a layered index, thus eliminating long searches.

3. Representation

In classical data models the basic data item is a record, which consists of several fields each having a name, type, and content. Some fields are designated as *keys*, and can be used to retrieve the record. For instance, consider the record (**John Doe, 123-45-6789, 101 Main Street**). The record consists of three fields: name, social security and address. The name and the address fields are of type character string, and the social security is an integer. In this example, we assume that the first two fields are *keys*, i.e., the record can be retrieved either by the name or by the social security.

The JS engine treats all data uniformly: it regards each key as a character string. In particular, we store the character string representations of the integers and floating-point numbers. The string representation should preserve the “less than” relation of integers and floating point numbers. To that end we pad the integers with zeros, normalize floating-point numbers, and represent the exponent before the mantissa. We can *store* data in its native format, and map it to strings on the fly for indexing. In order to treat all key types in a uniform manner, we associate with each key type a designator: a unique character (or character string) that identifies the key. The *designator* is prefixed to the key, to form a *universal* key.

For example, assume that the designator of “name” is *N* and the designator of “social-security” is *S*. Thus, either “**NJohn Doe**” or “**S123-45-6789**” will reach the previous record.

All universal keys are inserted into a single JS data structure, as shown in Figure 2. This diverges from the B-tree implementation of relational databases, where each key type requires a separate B-tree.

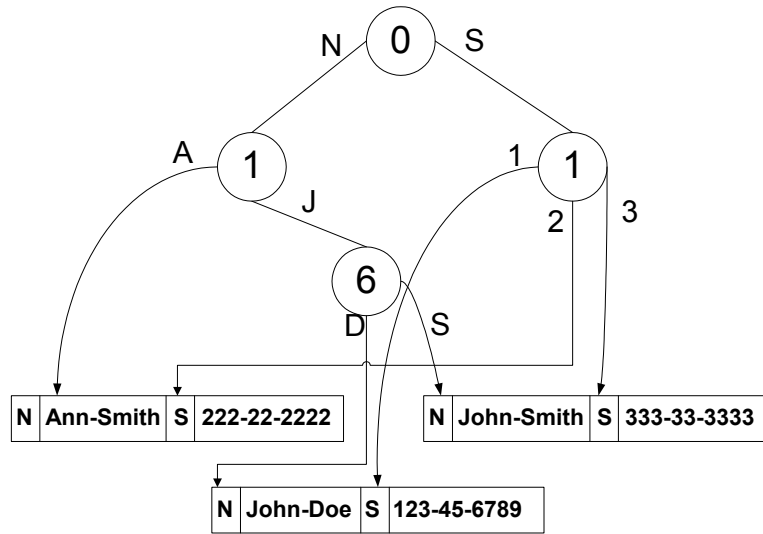


Figure 2. A PT addressing three records via their universal keys.

3.1. Composite keys

A single key does not always suffice to retrieve a record, such as in the case that the database contains several people named “John Doe”. In these cases, we can distinguish between the synonyms, perhaps by the social security number. Thus we could reach the person with social security number 123-45-6789 by searching the key “**NJohn DoeS123-45-6789.**” In practice, most object-relational implementations build a globally unique OID (Object Identifier) to refer to the distinction. The JS can do the same if there is no ancillary information available.

For retrieval, we use *composite keys*: the concatenation of several keys, which are searched by lexicographic order. In those cases where the primary key does not uniquely identify the key, the search would utilize the remainder. If, for example, there were 10 people named “John Doe” and only one “Adam Smith”, then after searching “**NJohn Doe,**” we need to continue to the social security field to find a particular “John Doe.” However, because the PT is insensitive to the length of keys, adding the social security field does not impact the size or efficiency of the index. For example, if there is only one “Adam Smith” record, we do not need the social security field. However, we can store it anyway without degrading the index performance. Then, if a new “Adam Smith” with a different social security number is inserted, we do not have to go back and “fix up” the original record. The additional information is maintained without cost to the index, and searched only when needed. In conventional search engines, such as B-trees, the entire composite key needs to be maintained for search, even when it is superfluous. This can enlarge the size of the B-tree and degrade the performance of searches.

3.2. Subordinate keys

Sometimes there is a semantic connection between several fields. Consider a public library, which has books, titles, borrowers etc. These entities are related. For instance, there is a one-to-many relation between a title of a book and its copies. Let A and B be entities, such that there is a one-to-many relation between the instances of A to those of B . In this case, we say that B is subordinate to A ; i.e., each of the physical copies of “Gone with the wind” is subordinated to the title. We wish to enter the data into the database in a way that will reflect the subordination relation.

First we will enter the titles of the books into the database (with designator, say T). Thus the data structure would contain the keys “ T Gone with the wind”, “ T The hitchhiker’s guide to the galaxy”, etc. The leaves corresponding to these keys will have pointers to the data records, which contain additional information.

Suppose that a 4-digit catalogue number identifies each copy. There are several ways to enter the copies in the data structure. The classical solution is to enter the catalogue number prefixed with a designator, thus we could have the keys “ C 1113” to “ C 1122” and “ C 2221” to “ C 2226.” Each new leaf will point to a record of the copy. However, this solution does not reflect the subordination relation between the title “Gone with the wind” and the book with catalog number 1113.

The JS offers another solution: Add the copies of each title as a sub-trie rooted at the node representing the title. This is shown in Figure 3. (In the figure, the \wedge symbol indicates the NULL character.) Thus the node “ T Gone with the wind” will be the root of a sub-trie containing the keys “ C 1113” to “ C 1122”. In the same manner we can insert the author of a book as additional record subordinated to the title. This is shown in the figure as an A edge from node 19. In fact, we can add as many subordination relationships as necessary, simply by adding a new subtrie for each type of item. Recall that this structure is balanced by horizontal layers; thus, even though the addition of subtrees makes the PT larger and more complicated, access is still efficient.

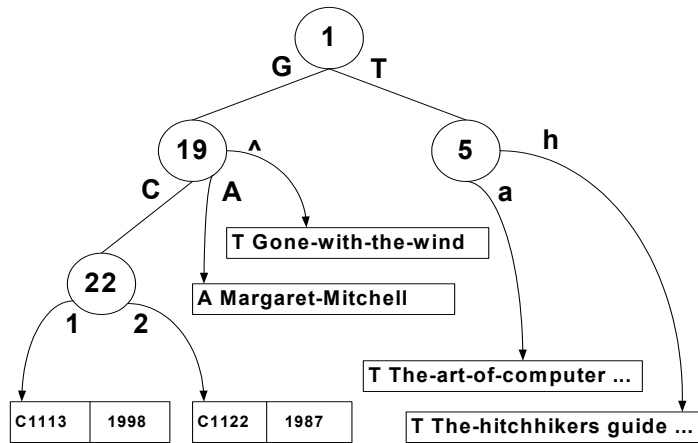


Figure 3. “Gone with the wind” and its subordinated copies.

Now, consider the effect of searching the composite key “**TGone with the windC1113**.” The search first reaches the node “**TGone with the wind**” and then continues to the record representing the copy. Thus the subordination relation between the copy and the title is fully reflected in the data structure. It is now easy to list all the copies of “Gone with the wind” by simply traversing the sub-trie rooted at the node labeled “**TGone with the windC**” (node 22).

Note that the primary key title and the composite key (title, catalogue number) coexist in the same data structure, and both can be accessed efficiently. Thus, using the key “**TGone with the windC1122**” we would reach the appropriate record. To achieve a similar effect in a B-tree, we would need two distinct trees: one for title, and one for catalogue numbers. To find all the copies of “Gone with the wind” we would need to traverse the entire “copy” B-tree and perform a selection by a foreign key. Another possibility is that we use the B-tree to index a precomputed “join index.” Such a join index can accelerate any query that contains a specific join. However, a join index is not general enough to be used for all queries, and must compete for resources (e.g. memory and disk space) with other more general indexes. Moreover, the join index must be updated whenever the data is updated. With the JS, there is no need for a separate index for each type of join, and thus resource contention is reduced. Moreover, once we have updated the index, there is no need to perform additional updates on other indexes.

If we want to retrieve books by catalogue number then we can implement both the primary key catalogue-number and the composite key (title, catalogue-number) in the same data structure. The search for “**C1113**” and “**TGone with the windC1113**” will lead to the same data record. Since the JS data structure is very compact, there is very little additional space overhead. Only when a new copy is acquired or lost will we have to update both keys.

The relation between a key and its subordinated keys need not be permanent; it can change dynamically. Consider the borrowing relation between a borrower and the books he borrows. The copies of the books borrowed by Adam Smith will form a sub-trie rooted at the “**NAdam SmithC**” node. Thus the path “**NAdam SmithC1120**” indicates that Adam Smith holds a book whose catalogue number is 1120. Each time a book is lent or returned this sub-trie will have to be updated.

Consider Figure 4. Three edges labeled *C*, *T*, and *N* leave the root of the trie. The edge labeled *C* leads to a sub-trie where all copies are listed (by catalogue number). The leaves of this trie lead to the records of the copies. (The figure depicts only the links to the 10 copies of “Gone with the wind”.) The *T* labeled edge leads to a trie where the books are listed by title; thus, “Gone with the wind” follows the edge labeled “*G*”. The leaves of the trie point to the title record that would probably list the author and other information. A leaf associated with a particular title is also root of a trie containing all the copies of the book. The leaves of this trie point to the records of the copies.

The N edge leaving the root is the root of a trie listing all borrowers by the name. The leaves of this trie point to the books borrowed by that person. Thus from the leaf pointing to Adam Smith emanates an edge labeled C that is the root of a trie listing all books borrowed by him. Since he borrowed only one book, copy #8 of "Gone with the wind," the trie contains only a root that point to the record of the appropriate copy.

If the borrowers had other information that we would wish to search for, such as payments, then the payment information would have a separate designator (say P) and an edge labeled P would lead from the leaf of Adam Smith to a trie containing the payment information.

This construct is similar to that of composite keys. In both cases we construct a composite key, and search it as if it were primary key. The difference however, is that the node of the first part of a composite key does not represent any entity, and thus does not point to any database record.

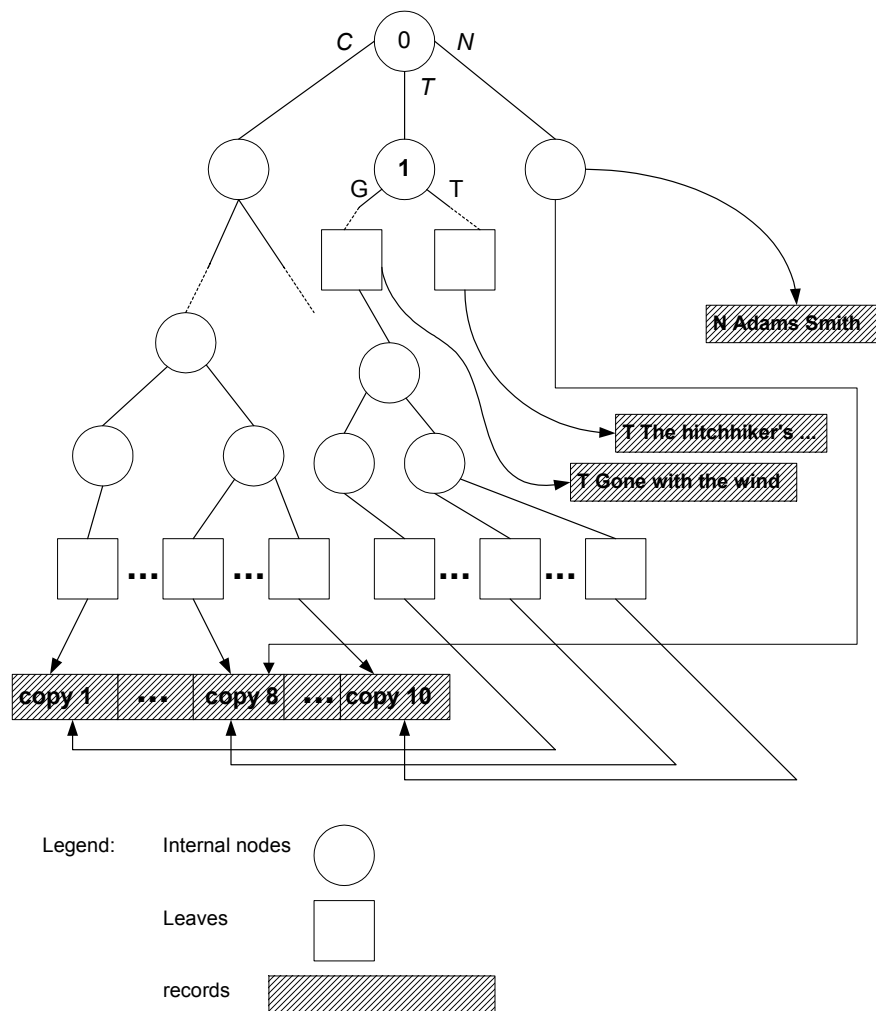


Figure 4. Copy, title and borrower keys, subordination between title and copy, and subordination between a borrow and a copy.

3.3. Schema and meta-data

To allow the engine to perform searches, the data structure should contain *meta-data*: a description of the schema, its various fields and keys, their designators and the subordination relations between them. This is the only external information needed to access the data structure. Therefore, regardless of the actual layout of the data on disk, or the internal structure of the JS, it is possible to use the meta-data to form meaningful queries that can then be translated into traversals over the index structure. The universality of this approach allows us to perform queries in the client-server model. According to that model, the client sends SQL-queries to the server, which has sole access to the database. This model provides a clean interface between different machines, with possibly different software, to the database. The JS fits neatly within this approach. Each client machine passes the SQL-query to the server. Using the meta-data, the server is able to process any SQL-query.

4. Performance

4.1. Index size

The JS data model allows us to represent many indexes compactly. As with B-trees, most of the space of a layered index is needed for the leaf layer: the layer that points to the records. In this layer, we need a pointer to each record, and some additional information that can be compacted to 2 bytes per record. Thus the layered index automatically compresses long keys. If a disk-address can be represented in 4 bytes, then the leaf layer of a databases of N records will requires $6N$ bytes. Thus the layered index for 10^6 records could easily fit in main memory. If there are more than 10^6 records, one layer must be stored on disk, but this layer can be efficiently accessed using the horizontal structure.

The JS can accommodate an index for several keys within the same structure i.e., with very little additional extra space. Moreover, the index can explicitly represent relationships between items (using the subordination mechanism described in Section 3.2) without significantly increasing the index size.

We can also implement the primary key index as a sparse index to achieve additional space savings. This would require only one pointer per block, rather than one pointer per record, thus drastically reducing the size of the index. A database with N keys, and an average of B records per block, would require only $2N/B$ blocks for the primary index.

4.2. Search time

For a database with about 10^9 keys, the layered index allows access to any record within two disk accesses (one for the index, and one for the data). If the access is to a record B subordinated to A then the path to B includes the address of A , and at most one more disk access is needed to access record A . Consider navigating in Figure 4 from Adam Smith to a copy of a book he had borrowed. In the JS we would search

for “**NAdam Smith**” (using the horizontal layer structure) to find the node at the root of the sub-trie “**NAdam SmithC.**” Traversing this subtrie provides the answer to our query. Therefore, we can answer the query “What books were borrowed by Adam Smith?” without having to perform a potentially expensive join.

4.3. Sequential processing

The tree-like structure of the tries allows us to process the database sequentially by any one of the keys. We can follow a depth-first traversal of the PT to find all of the keys of the data in sorted order. As with B-trees, the records of the primary key can be accessed in a block-wise manner: in a single access one can access all the records of a block, thus the number of disk accesses is minimal (N/B , where N is the number of records and B is the average number of records in a block). For secondary keys savings this dramatic are not possible. However, in practice, only a single disk-access is required for each record when using the layered index.

4.4. Joins

The data representation of the JS allows us to perform joins very efficiently. Consider, for example, a library that has members (identified by name). Some members have issued requests, while some members are delinquent. Thus we need two tables: r (requests) and d (delinquent members, and the list of books not returned on time). Suppose we want to form a list of all requests issued to delinquent members, together with the reason for listing them as delinquent. In other word, we wish to perform a join between r and d .

One possibility for a classical implementation (using B-trees and/or hash-tables) is that “name” is a key of both r and d . In this case, to find the join one can perform a “Merge-join”, an algorithm that requires traversing all the blocks of r and d .

The performance of the JS depends on the way we implemented the relations. We may follow the classical implementation and create two separate PT’s: one for requests and one for delinquent members. The JS will be somewhat more efficient, since the underlying data structure is more compact, it has fewer layers and thus a larger part of it may reside in main memory.

However, there is a better choice: maintain two tables under a single PT, with “name” serving as the key. Thus the request for Adam Smith will be accessed as “**AAdam SmithR**” and the delinquent Adam Smith can be accessed as “**AAdam SmithD.**” See Figure 5.

To perform the join, we would need to traverse only the part of the PT that contains the names. From each node that represents a name we check whether an edge marked by the designators **R** and **D** leaves that node. If both edges exist, we add a tuple (or tuples) to the join. Since the name part of the PT contains only the names, it is fairly small and might even fit in main memory. The remainder of the search depends on the

size of the output, and is independent of the size of the original relations. Thus, this operation is very efficient.

This implementation is as efficient to search as implementing pre-aggregated joins, where one would maintain an additional table containing the result of the join [4]. The price of pre-aggregated joins is, of course, that every update to either of the two original relations requires us to update the join table too. In the JS we do not need to perform an additional update; it is performed implicitly at no extra cost!

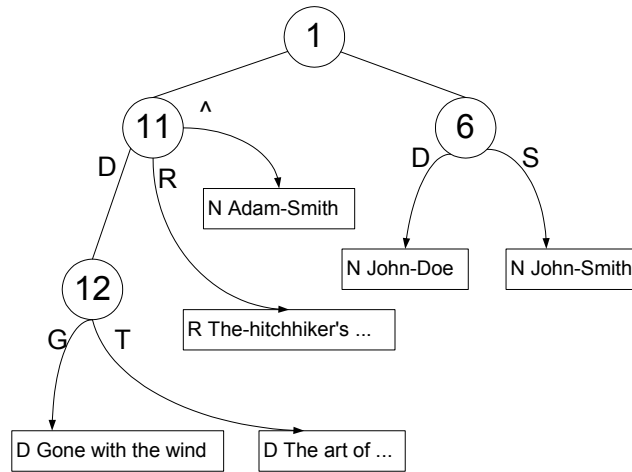


Figure 5. Performing joins with the JS.

5. Properties of the model

5.1. Referential integrity

Often we need to ensure that a value that appears in one relation appears in a second relation [5]. For example, suppose “Adam Smith” borrowed “The art of computer programming: volume 4”. To add this transaction to the database, we have to make sure that “Adam Smith” is a library member, and the library has a copy of the book. A relational database would need to check this before writing the lending transaction. These checks would be performed in multiple B-trees, and the trees may be locked until the transaction terminates.

In the JS some of these checks are supported automatically at the engine level. The insertion process of a subordinate record passes through the record to which it is subordinated. Thus one does not need to explicitly check whether this record exists, and such a referential integrity test is handled without additional accesses.

Figure 6 shows an attempt to link “The art of computer programming: vol 4” to the borrow transaction. We first add the record “NAdam Smith7The-art...vol.4.” We successfully establish the link from the name to the subordinated field “title” (i.e., a link from the node labeled 12), but when we try to establish a

reference from the title (node labeled 41), we discover that the book does not exist. Thus the second insert fails, implying a violation of the referential integrity.

Thus after the insertions we do not need to perform any checks, while a classical implementation would need to perform an insertion (into the borrow table), update two index files (name, title) and (title, name), and finally perform two integrity checks (to the title table and the borrower table).

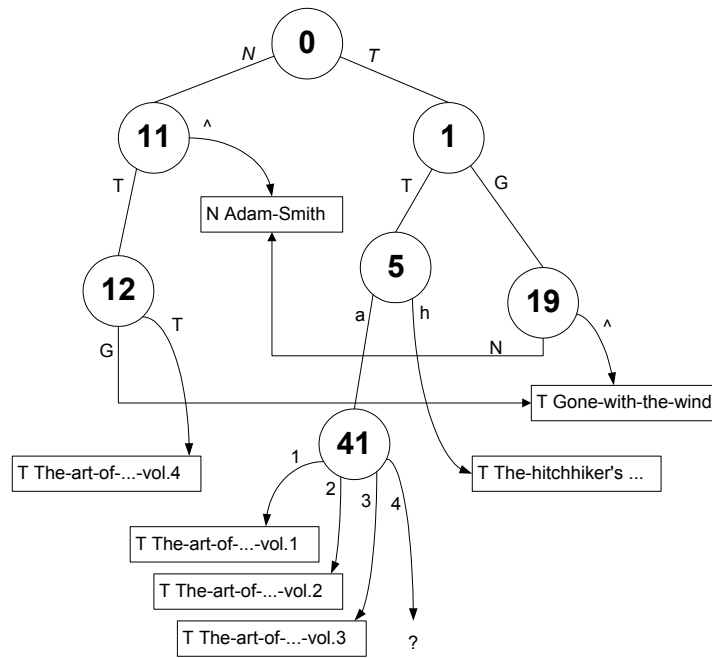


Figure 6. Performing integrity checks.

5.2. Multi-dimensional views

Sometimes two different composite keys lead to the same node. Suppose that each book can be uniquely defined by (ISBN) or the composite key (author, title). The JS allows us to simultaneously maintain both paths to the same physical record. We can therefore reach the record by either search path.

Once a record is accessed its price field can be updated, regardless of the path by which it was reached. Suppose the ISBN of “Gone with the wind” is 8334, and we wish to change its price from \$6.50 to \$7.34. Then updating the tuple (8334,\$6.50) to (8334,\$7.34) has the effect of updating the tuple (Margaret Mitchell, Gone with..., \$6.50) to (Margaret Mitchell, Gone with..., \$7.34).

Suppose, moreover, that the library holds several copies of each book. An implementation that relies on B-trees for indexing would first have to maintain two separate tables: $T_1 = (\text{ISBN}, \text{copy})$ and $T_2 = (\text{author}, \text{title}, \text{copy})$. Since in the JS the copies are represented by a trie rooted at the book record (see Figure 7), adding a copy is effected by adding a node to this trie. Hence, adding a new copy via the ISBN path has the

effect of adding the copy also to the (author, title) path. A classical implementation would require two distinct updates, to tables T_1 and T_2 .

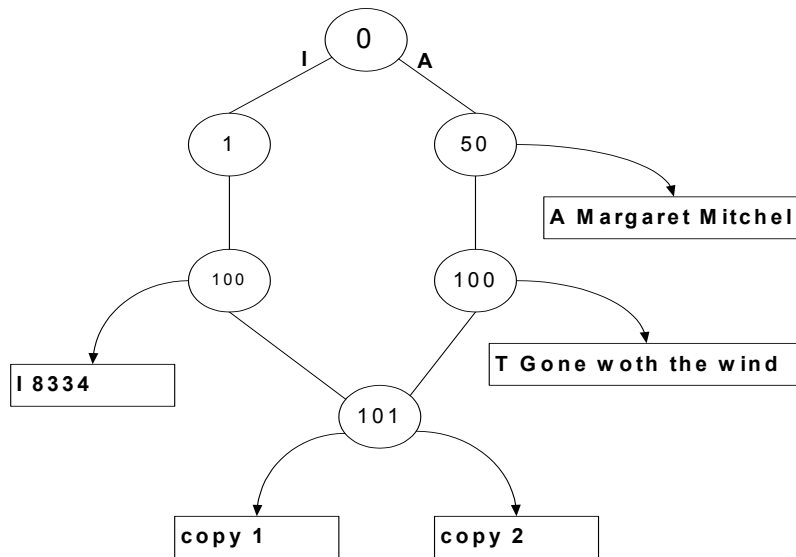


Figure 7. Multiple views.

6. Multiple models

A unique feature of the JS is that one can accommodate all the major database models within the same data structure. Moreover, the same data is represented only once and can be simultaneously accessed by different users, each assuming a different model.

6.1. The relational model

This is the most popular database model. Data is organized as tables, and all connections between records are through the values of the keys. For example, suppose a library database has tables $T_1 = (\text{title}, \text{copy})$ and $T_2 = (\text{borrower}, \text{copy})$. Then in a classical model, to find the titles of the books borrowed by “Adam Smith” we select T_2 along “borrower = Adam Smith,” extract the copy number of the retrieved tuples, and finally search for the copy in T_1 . In contrast, in the JS searching “**NAdam Smith**” would bring us to a root of a trie containing all the copies borrowed by Adam Smith. The leaves of this trie point to the records of the book, which contain all the information on the copy, including the title.

A major advantage of the relational model is the clean user interface provided by the relational algebra and its implementations such as SQL. Implementations of the relational database contain an SQL compiler that translates these queries to B-tree operations. Clearly, these queries can equally well be translated to JS

operations where they take advantage of the connections between related items. (One could, of course, ignore these connections and imitate the traditional implementations of relational tables.)

6.2. Network and hierarchical models

The JS is similar to the network model in that related items are connected. However, while in the network model, the items are connected via pointers, in the JS the connection is via sub-tries inherent to data structure. Although long search paths might occur in the network model, the JS data structure (see Section 2) guarantees that the lengths of the paths actually examined are logarithmic (the base of the logarithm is B : the average number of records per block). Hence in practice the search requires usually one or at most two disk access.

Note that one could also implement network and hierarchical queries in the JS. Thus the same data structure simultaneously supports both the network and hierarchical models, as well as the relational model.

6.3. The object-oriented model

The object-oriented approach considers all data as objects. Every object belongs to a class, which determines its structure and the methods (functions) that can be applied to it. The classes are organized in a hierarchy, from which the data fields and methods may be inherited. The object-oriented approach was developed in ephemeral setups: an object exists only while the program that created it is active. In database applications data needs to be supported for a longer periods, i.e., they should be persistent. Thus, to apply this approach to databases we need to retain objects on the disk. In the object-oriented approach to databases, objects defined as persistent are stored on the disk and are available to other (authorized) programs. Note that all programs have to agree on the internal representation of the persistent objects.

The JS can easily support persistent objects. Since their structure is uniformly encoded with the aid of designators, later incarnations of the programs, as well as other programs, can access such persistent objects. Note that at the same time a persistent object can also be part of a relational table. There is no need to duplicate data. An important component of object-oriented query languages is the ability to follow paths, e.g. a series of nested objects or class hierarchies. Each path through the database can be encoded as a long string that is indexed by the JS. Long or complex paths can be evaluated with efficient lookups in the index, providing significant optimization for queries.

6.4. The object-relational model

In contrast to the object-oriented approach, the relational approach considers all data as tables. The output of an SQL query is a table whose structure depends on the query. Object relational systems allow us to treat essentially relational data in an object oriented way, e.g. with complex nesting of objects, user

defined data types, and methods defined over the data. Since all data in object-oriented languages should be objects, we need to convert the results of queries into objects. Each query might give rise to a different object, which should be manipulated by special methods. The object-relational approach provides an interface to convert tables to objects. The interface requires the user to specify the relationship between the objects and the table attributes. If some attributes themselves are tables, we need to allow relational algebra operations on these tables too. The application program, preventing the database engine from optimizing the queries, performs these conversions.

The JS treats all data in a uniform manner, thus providing the interface between the object-oriented application program and the data structures. The application program's queries are formulated in terms of universal keys, so the database can optimize the query strategy. The database returns universal keys, which the object-oriented application program can readily process by the object-oriented methodology.

The sequence of designators of the search path to the object determines its class, and the designators to various fields allow the object-oriented program to resolve polymorphism of the method calls. See 0 for a detailed description of using the JS engine to implement object-oriented and object-relational databases.

6.5. Blobs and user defined functions

The object-oriented approach allows users to add user-defined typed (UDT) and user-defined functions (UDF). For example, one could add the video of "Gone with the wind" or the audio recordings of the radio show "The hitchhiker's guide to the universe". The new UDT can be based on or be related (by subordination) to any other data type. Now, with the universal key, the application can navigate to the new UDT from the defined classes from which the new UDT can inherit methods and other properties. In the example, when navigating in the index, one could navigate from the book to the video and then extract the theme song.

7. Conclusions

The JS data engine is a new method for indexing and searching data. It uses the layered index data structure, in which one can perform searches over long paths in logarithmic time. We simplified the exposition by assuming that the underlying data structure was a Patricia trie, but one should bear in mind that the real implementation is based on the layered index.

Many features of current models carry over to our method:

- Locking and recovery algorithms can be transferred from B-tree.
- The physical location of records is transparent to the user and is not hardwired into the system.

- One may continue to use traditional query languages, such as SQL, so there is no need to reprogram application programs.

The method has substantial performance advantages over the implementation of relational databases indexed via B-trees. One does not need to maintain a separate tree for each search path. Also, by using the subordination mechanism, we can readily realize logical connections between data items, thus speeding up complex operations such as joins and aggregates.

Finally, the uniform approach enables us to efficiently view the same database via several different models such as the relational model, the object-oriented model, the object-relational model, the hierarchical model and so on.

References

- [1] A. Itai and M. Shadmon. Supporting Object and Object-Relational Models
- [2] A. Itai and M. Shadmon. The JS data structure. Technical report
- [3] D. E. Knuth. The Art of Computer Programming, volume 3. Addison-Wesley, 1973.
- [4] O. Shmueli and A. Itai. Complexity of views: Tree and cyclic schemes. *SIAM J. Computing*, 16:17-37, 1987.
- [5] A. Silberschatz, H. F. Korth, and S. Sudarshan. Database system concepts. McGraw-Hill, 3 edition, 1997.