Formalization of Indexing in Data Management

Maarten M. Fokkinga
DB group, dept INF, fac EWI, University of Twente, Netherlands

Version 0.96, of May 3, 2004, 13:02

Abstract. Indexing is an important notion for efficient data management. It prescribes the use of a particular additional data structure for direct or improved access to the locations where requested values have been stored. Indexing itself makes no assumptions about the structure of the actual storage of values or indices (but a realization does). All this is elegantly formalized in the Z notation.

1 Introduction. Indexing is an important notion for efficient data management. We formalize the notion of indexing: the use of an additional data structure *Idx* for direct or improved access to the locations where requested values have been stored in a store *Store*. The notion of indexing itself makes no assumptions about the organization and structure of the storage of values or indices: *Store* and *Idx* are parameters standing for just sets.

To express our ideas, we use a stylized set and predicate notation, with a facility for scope control. The notation is known as the Z notation [2]; there is an international standard for it, it is widely used for software specification, and there are various tools to support usage of the notation, such as a type checker and theorem prover [1]. All our displayed formulas have been type-checked (but some universal quantifications near the beginning of a formula have not been printed).

In the next section the crux of indexing is formalized; thereafter follows a section with some actual implementations that satisfy the specifications we have given. Both sections suggest that we are dealing with 'databases' only, but in the final section we briefly show that indexing in other fields of data management are covered as well.

The crux of indexing

2 Overview. In this section we give four specifications of a database for storing $U \times V$ values: *DB*, *DBL*, *DBi*, and *DBI*. What we specify as a database is, in terms of the relational model, just one table with two anonymous attributes, having type U and V, respectively. Besides parameters U and V, the specifications make also use of parameters *Store*, *Loc*, and *Idx*; eventually each of these parameters needs be replaced by one set or another.

Specification *DB* is the interface given to an end-user of the database; it provides *put* and *get* operations for storing and retrieving values. For brevity and simplicity we abstract away a delete operation: so, once a value has been put into the store, it remains in the store forever.

Specification *DBL* is an auxiliary refinement of *DB*, formalizing the notion of *location* of a value in the store, without which the notion of index makes no sense.

Specifications *DBi* and *DBI* formalize the crux of *indexing*; without any assumptions about implementations of *DB*, except that locations are available as specified by *DBL*, it assumes the existence of an additional data structure *Idx*, which is just a set parameter, and prescribes how this is to be used, thus making the retrieval more efficient: values are retrieved by direct access (in case of *DBi*) or improved access (in case of *DBI*) to their location. Specification *DBI* is more general than *DBi*.

3 End-user interface. A database system provides an initial store *init*, and operations *put* and *get* to store and retrieve pairs (u, v) of values; the specification is called *DB*. Sets U, V, and *Store* are parameters of the specification; nothing is assumed about these (and need be known by the end-user) except that they are sets. In an implementation, some particular choice for these has to be made; U and V will be the data types that the user want to store, and, independently, *Store* could be a set or a table or a tree of $U \times V$ values:

DB[U, V, *Store*]

$$
\begin{array}{l}
init : Store \\
put : Store \longrightarrow (U \times V) \longrightarrow Store \\
get : Store \longrightarrow U \longrightarrow \mathbb{P}\ V \\
\hline
get\ init\ u = \varnothing \\
get\ (put\ s\ (u', v))\ u = get\ s\ u \cup (\textbf{if}\ u = u'\ \textbf{then}\ \{v\}\ \textbf{else}\ \varnothing)
\end{array}
$$

Thus *DB* merely proclaims the existence of an *init*, *put* and *get* with the mentioned types and properties, in terms of parameters $U, V, Store$. The properties say, roughly, that *get s u* yields the set of all v-values for which the pair (u, v) has been put into the store by *put* where the initial store is *init*. Since nothing is specified about the structure of *Store*, this specification characterizes not only relational databases, but also hierarchical and network databases, or whatever!

It is easy to make U into a key of the database, by requiring that there is at most one pair $(u, \ldots)$ in *Store* for each u. The notion of index, defined below, is independent of the choice to make U into a key or not.

4 Locations. The notion of index only makes sense when the notion of location exists: an index of a value is a *location* of the value in the store. So here we extend specification *DB* with locations, and call it *DBL*. The specification does not say how a location looks like. There can be various realizations; for instance, a location can be a number, so that *Store* is a kind of table, or a location can be a path in a tree, so that *Store* is a kind of tree. Function *newloc* yields a location not in use by its store:

DBL[U, V, *Loc*, *Store*]

$$
\begin{array}{l}
DB[U, V, Store] \\
newloc : Store \longrightarrow Loc \\
putl : Store \longrightarrow Loc \longrightarrow U \times V \longrightarrow Store \\
getl : Store \longrightarrow Loc \nrightarrow U \times V \\
\hline
put\ s\ uv = putl\ s\ (newloc\ s)\ uv \\
get\ s\ u = \{v : V \mid (u, v) \in \mathrm{ran}(getl\ s)\}
\end{array}
$$

Thus *DBL* merely proclaims the existence of *init*, *put*, *get*, *newloc*, *putl*, and *getl* with the types and properties as mentioned here implicitly (namely in *DB*) or explictly. Note that the evaluation of *get s u* suggested here is a scan over the entire "contents of store *s*", that is, over the range of *getl s*. The definitions here of *put* and *get* together with their properties from the imported *DB*, imply several additional properties of *putl*, *getl*, and *newloc*. For example:

$$\forall\, DBL[U, V, Loc, Store] \bullet \quad \mathrm{dom}(getl\ init) = \varnothing \ \wedge \ newloc\ s \notin \mathrm{dom}(getl\ s)$$

A realization of *DBL* should establish these derived properties — for otherwise it is not a realization of *DBL*. Given a realization of *DBL*, one might want to *hide* operations *newloc*, *putl*, and *getl*, since these are not to be used by and end-user; such hiding is done in paragraph 7.

5 Indexing. An *index* is a data structure that, for each value, stores the locations where that value has been stored; or, more general, it gives for each value a *part of the storage* where that value is located if it occurs at all. We formalize these variants of indexing in *DBi* and *DBI*, respectively. Since our previous specification of a database is so general, we can say that *an index itself is a database*, storing pairs (u, l) of a value u and a location l incase of *DBi*, and storing pairs (u, L) of a value u and a set L of locations in case of *DBI*.

The simple case DBi. As explained, an indexed database, *DBi*, actually consists of two databases; one to store the users $U \times V$ pairs (specified by $DBL[U, V, Loc, Store]$ with *Store* as store) and another to store the value-location pairs (specified by $DB[U, Loc, Idx]$ with *Idx* as store). We combine these two into one new database with the product $Store \times Idx$ as store: $DB[U, V, Store \times Idx]$. In order not to confuse the operations of the two constituent databases with each other and with the newly formed database, we decorate their names with a single and double quote, respectively. In terms of these auxiliary operations the end-user operations are specified:

$DBi[U, V, Loc, Store, Idx]$

$DBL'[U, V, Loc, Store]$
$DB''[U, Loc, Idx]$
$DB[U, V, Store \times Idx]$

$init = (init', init'')$
$put\ (s, idx)\ (u, v) = (put'\ s\ (u, v), put''\ idx\ (u, newloc'\ s))$
$get\ (s, idx)\ u = \{l : get''\ idx\ u \bullet second\ (getl'\ s\ l)\}$

Thus *DBi* merely proclaims the existence of *init'*, *put'*, *get'*, *newloc'*, *putl'*, *getl'*, *init''*, *put''*, *get''*, *init*, *put*, and *get* with the types and properties as mentioned implicitly and explicitly. In addition to what *DB* specifies for *init* and *put*, the first two explicit properties specify that *init* and *put* delegate the effects on the left component of the $Store \times Idx$ store to the auxiliary *init'* and *put'*, and that they delegate the effects on the right component of the $Store \times Idx$ store to the auxiliary *init''* and *put''*. The third property of *DBi* gives an equation for *get* (in addition to what *DB* specifies for *get*); the equation suggests an evaluation that no longer is a scan over the range of *getl' s* (as suggested in *DBL'*) but instead a *direct retrieval from the locations get'' idx u given by the index*, with no assumption at all how *Store* and *Idx* looks like. If *get''* can be evaluated efficiently, then so can *get*. This is the crux of indexing!

Given a realization of *DBi*, one might want to *hide* all primed and doubly primed operations, since these are not to be used by and end-user; such hiding is done in paragraph 9.

The general case DBI. Rather than storing a single location for each value, in the more general case an index gives a part of the storage where the value can be found, if it occurs at all. This allows for two extremes: the part of the storage is just one location (like in *DBi*), or the part of the storage is just the entire storage. The change to *DBi* in order to obtain *DBI* is clear: generalize, at appropriate places, the single location to a set of locations:

$$
\begin{array}{|l}
DBI[U, V, Loc, Store, Idx] \\
\hline
DBL'[U, V, Loc, Store] \\
DB''[U, \mathbb{P}\, Loc, Idx] \\
DB[U, V, Store \times Idx] \\
\hline
init = (init', init'') \\
\exists\, L : \mathbb{P}\, Loc \bullet put\,(s, idx)\,(u, v) = (put'\, s\,(u, v),\ put''\, idx\,(u, L \cup \{newloc'\, s\})) \\
get\,(s, idx)\, u = \{v : V; L : get''\, idx\, u \mid (u, v) \in (getl'\, s)\,(\!|L|\!) \bullet v\} \\
\hline
\end{array}
$$

Not only the above equation for *put* is a straightforward generalization of the equation for *put* in *DBi*, but also the equation for *get* looks very similar to an equation for *get* that is derivable in *DBi*:

$$
\begin{array}{l}
\forall\, DBi[U, V, Loc, Store, Idx] \bullet \\
get\,(s, idx)\, u = \{v : V; l : get''\, idx\, u \mid \{(u, v)\} = (getl'\, s)\,(\!|\{l\}|\!) \bullet v\}
\end{array}
$$

Again, if get'' can be evaluated efficiently, then so can *get*.

Realizations of the specifications

To show the realizability of the specifications given above, we give several realizations: with relations, tables, and trees. We begin with the "hard" part: realizations without indexing; once this has been done, *realizations of indexed databases come for free* (paragraph 9)!

6 Store as relation. The simplest realization of *DB* is to take relations $U \leftrightarrow V$ as *Store*:

$$
\begin{array}{|l}
DB0[U, V] \\
\hline
DB[U, V, U \leftrightarrow V] \\
\hline
init = \varnothing \\
put\ s\ uv = s \cup \{uv\} \\
get\ s\ u = \{v : V \mid (u, v) \in s\} \\
\hline
\end{array}
$$

Since *DB*0 imports *DB*, it trivially implies the properties of *DB*:

$$\forall\, DB0[U, V] \bullet DB[U, V, U \leftrightarrow V]$$

However, *DB*0 could be inconsistent, for example because its newly added properties conflict with those of the imported *DB*. We claim without proof that this is not the case:

$$\exists\, DB0[U, V] \bullet true$$

Similar claims can be made for the following realizations as well.

7 Store as table. We take stores to be tables (sequences, in our notation), and locations to be natural numbers. Thus we implement not only *DB* but even *DBL*:

$DB1[U, V]$

$DBL[U, V, \mathbb{N}, \mathrm{seq}(U \times V)]$

$$
\begin{aligned}
&\mathit{newloc}\ s = 1 + \#(\mathrm{dom}\ s)\\
&\mathit{init} = \langle\,\rangle\\
&\mathit{putl}\ s\ l\ uv = s \frown \langle uv \rangle\\
&\mathit{getl}\ s\ l = s\ l
\end{aligned}
$$

Notice that *putl* discards its l-argument; it is safe to do so since from *DBL* it follows that the actual l-argument is indeed the location where *putl* stores the uv-value. It also follows that the 'definition' in *DBL* of *get* may now be simplified somewhat:

$$\forall\, DB1[U, V] \bullet \mathit{get}\ s\ u = \{v : V \mid (u, v) \in \mathrm{ran}\ s\}$$

The evaluation of $\mathit{get}\ s\ u$ suggested here is still a scan over the entire sequence (table) s.

Aside. By hiding *newloc*, *putl*, and *getl* from $DB1$ we obtain a realization of *DB* that no longer satisfies *DBL* since it doesn't specify the required auxiliary operations:

$$DB1[U, V] \setminus (\mathit{putl}, \mathit{getl}, \mathit{newloc})$$

Formally, this hiding is defined by putting an appropriate existential quantification for *putl*, *getl*, and *newloc* in front of $DB1[U, V]$. We claim without proof that in this $DB1$-with-hiding no more properties for *init*, *put*, *get* are derivable than in *DB*. Thus, by giving this $DB1$-with-hiding to an end-user, the end-user knows just what is specified by *DB*, whereas an implementation that satisfies the full $DB1$ does have the auxiliary *putl*, *getl*, *newloc* and all the extra properties.

8 Store as tree. [This paragraph is yet another, somewhat complicated and not very practical, illustration; it may be skipped without loss of continuity!] We take the store to be a binary tree with pairs (u, v) at the leafs. A path in the tree functions as location; a path is a sequence of indications 0 (left) and 1 (right): $\mathrm{seq}\{0, 1\}$. Function *newloc* decides, given all paths of a tree, what path to extend with what label; actually, the *newloc* defined below keeps the tree balanced — it extends the shortest path with the appropriate label. We first recall the well-known lexicographic order on sequences:

$$
\begin{aligned}
(\_ \prec \_) == \{p, q : \mathrm{seq}\,\mathbb{Z} \mid (\exists\, n : 0\,..\,\mathit{min}\{\#p, \#q\} \bullet\\
(\forall\, i : 1\,..\,n \bullet p\ i = q\ i) \wedge (\#p = n < \#q \vee p\ (n+1) < q\ (n+1)))\}
\end{aligned}
$$

$DB2[U, V]$

$DBL[U, V, \mathrm{seq}\{0, 1\}, \mathrm{seq}\{0, 1\} \pfun U \times V]$

$$
\begin{aligned}
&\mathit{newloc}\ \varnothing = \langle 0 \rangle\\
&p \in \mathrm{dom}\ s \wedge (\forall\, p' : \mathrm{dom}\ s \mid p' \neq p \bullet p \prec p') \Rightarrow\\
&\qquad \mathit{newloc}\ s = \textbf{if}\ \mathit{front}\ p \frown \langle 1 \rangle \in \mathrm{dom}\ s\ \textbf{then}\ p \frown \langle 0 \rangle\ \textbf{else}\ \mathit{front}\ p \frown \langle 1 \rangle
\end{aligned}
$$

It follows that *init* and *put* establish and maintain the property that the lengths of the paths differ at most one, so that the stores are balanced trees:

$$Balanced[U, V] == \{s : \mathrm{seq}\,\mathbb{N} \pfun U \times V \mid (max(\#(\mathrm{dom}\, s)) - min(\#(\mathrm{dom}\, s))) \leq 1\}$$

$$\forall\, DB2[U, V] \bullet init \in Balanced \land put \in Balanced \longrightarrow (U \times V) \longrightarrow Balanced$$

Again, $DB2$ can now be restricted to a realization of DB only, by hiding the auxiliary entities.

9 Realizations with indexing. Since an indexed database is just a combination of two databases, we are readily done with indexed realizations: they come for free! We can take each realization of DBL (for instance, $DB1$ and $DB2$) as the "underlying database", and each realization of DB (for instance, $DB0$, $DB1$, and $DB2$) as the "index database". For example, to specify $DB1$ as the underlying database and $DB0$ as the index database, we take appropriate instances of $DB1'$ and $DB0''$ and glue them together with the properties described in DBI:

$$DB\_1\_0[U, V] \mathrel{\widehat=} DBI[U, V, \mathbb{N}, \mathrm{seq}(U \times V), U \rightarrowtail \mathbb{N}] \land DB1'[U, V] \land DB0''[U, \mathbb{N}]$$

Similarly for the other five combinations of an underlying and index database.

As in paragraph 7, we can hide all auxiliary entities from DB_1_0:

$$DB\_1\_0[U, V] \setminus (init', put', get', putl', getl', newloc', init'', put'', get'')$$

This is a specification that proclaims the existence of *init*, *put*, and *get* with some properties. We claim without proof that no more properties of $init, put, get$ can be derived from this specification than those that follow from DB — although DB_1_0 itself specifies the existence of the auxiliary entities together with their intended properties and suggested efficient evaluation for *get*.

Concluding remarks

10 Physical aspects. Our formalization of the crux of indexing is entirely at the so-called *logical* level; *physical* aspects like caching, the place on disk where values are stored, and the choice whether to keep the index in memory, are completely absent. These aspects, of course, do have an influence on the efficiency, and, after all, efficiency is the reason why indexing is introduced in the first place. Yet, the formalization shows that already at the logical level *the main* (*a main*?) efficiency gain is apparent: instead of a scan over the entire store, a retrieval can be done with direct access to the locations where the relevant values have been stored.

11 Beyond databases. Having formalized the notion of indexing in databases, the question arises what indexing is in information retrieval or in any other field of data management. Fortunately, our notion of 'database' as specified by DB in paragraph 3 is so general that these other fields are covered as well. For example, in information retrieval one stores *document addresses* and retrieves document addresses by giving *terms* that should occur in the documents — this description abstracts from ranking the documents. These systems satisfy DB when we take U to be the set of all terms in all documents, and V to be the set of document addresses. Inserting a document into an information retrieval system amounts to

putting pairs (t, a) into the store for all terms t that occur in the document at address a. A query with term t yields the document addresses whose documents contain t; according to *DB* this can be done by *get* since $get\, s\, t$ yields "all a for which (t, a) has been put into s" (see paragraph 3). In a realization without indexing there is no other option for an implementation of *get* than to inspect each pair in the store and see whether t is the first component of the pair; with indexing, the index gives for each term t the locations where a pair (t, a) is stored, so that the required document addresses are efficiently retrieved.

References

[1] Jonathan Bowen. The Z notation — WWW page. `http://www.comlab.ox.ac.uk/archive/z.html`.

[2] J.M. Spivey. *The Z notation: a reference manual (2nd edition)*. Prentice Hall International, UK, 1992.