A Framework for Access Methods for Versioned Data

Betty Salzberg*[1], Linan Jiang[2], David Lomet[3], Manuel Barrena[4]**, Jing Shan[1], and Evangelos Kanoulas[1]

[1] College of Computer & Information Science, Northeastern University, Boston, MA 02115
[2] Oracle Corporation, Oracle Parkway, Redwood Shores, CA 94404
[3] Microsoft Research, One Microsoft Way, Redmond, WA 98052
[4] Universidad de Extremadura, Cáceres, Spain

Abstract. This paper presents a framework for understanding and constructing access methods for versioned data. Records are associated with version ranges in a version tree. A minimal representation for the end set of a version range is given. We show how, within a page, a compact representation of a record can be made using start version of the version range only. Current-version splits, version-and-key splits and consolidations are explained. These operations preserve an invariant which allows visiting only one page at each level of the access method when doing exact-match search (no backtracking). Splits and consolidations also enable efficient stabbing queries by clustering data alive at a given version into a small number of data pages. Last, we survey the methods in the literature to show in what ways they conform or do not conform to our framework. These methods include temporal access methods, branched versioning access methods and spatio-temporal access methods. Our contribution is not to create a new access method but to bring to light fundamental properties of version-splitting access methods and to provide a blueprint for future versioned access methods. In addition, we have not made the unrealistic assumption that transactions creating a new version make only one update, and have shown how to treat multiple updates.

1 Introduction

Many applications such as medical records databases and banking require historical archives to be retained. Some applications such as software libraries additionally require the ability to reconstruct different historical versions, created along different versioning branches. For this reason, a number of access methods for versioned data, for example [11,4,1,10,7,13,8], have been proposed.

In this paper, we present a framework for constructing and understanding versioned access methods. The foundation of this framework is the study of *version splitting* of units of data storage (usually disk pages).

* This work was partially supported by NSF grant IRI-9610001 and IIS-0073063 and by a grant for hardware and software from Microsoft Corp.

** This work was partially supported by DGES grant PR95-426.

Version splitting takes place when a storage unit becomes full. However, unlike in B-tree page splitting, some data items are *copied* to a new storage unit. Thus the data items are in both the old storage unit and the new storage unit. The motivation for copying some of the data when a storage unit is too full to accept a new insertion is to make the **stabbing query** (sometimes called "version-slice query", "time-slice query" or "snapshot query") ("Find all data alive at this version") efficient. Version splitting (and version-and-key splitting and page consolidation, both of which include version-splitting) cluster data in storage units so that when a storage unit P is accessed, a large fraction of the data items in P will satisfy the stabbing query.Many access methods for versioned data [11,4,1,10,7,13,8,12,6,9] use version-splitting techniques.

Our contribution is to explain version-splitting access methods as a general framework which can be applied in a number of circumstances. We also consider a more general situation where one transaction that creates a version can have more than one operation.(Many existing papers assume that a new version is created after each update. This is an unrealistic assumption.) It is hoped that the clearer understanding of the principles behind this technique will simplify implementation in future versioned access methods. In particular, it should become obvious that several methods which have been described in very different ways in the literature share fundamental properties.

Outline of Paper

The paper is organized as follows. In the next section, we will describe what we mean by versioned data and how it can be represented. Version splitting, version-and-key splitting and page consolidation is presented in section 3. In section 4, we describe operations on upper levels of the access method. In section 5, we will show how the related work fits into our framework. Section 6 concludes the paper with a summary. **Boldface** is used when making definitions of terms and *italics* is used for emphasis.

2 Versions, Versioned Data, and Version Ranges

In this section, we discuss versions, versioned data and version ranges. To illustrate our concepts, we begin with a database with three data objects or *records*, which are updated over time. We will start our first example with only two versions, followed by the second one with three versions. After presenting these examples, we will give some formal definitions.

2.1 Two-Version Example

First we suppose we have only two versions of the database. The first version is labeled v_1 and the second version is labeled v_2. In this example we have three distinct record keys. Each record is represented by a triple: a version label, a version v_1

(v_1, k_1, d_1)
(v_1, k_2, d_2)
(v_1, k_3, d_3)

Fig. 1. Database starting with one version.

version v_1	version v_2
(v_1, k_1, d_1)	(v_2, k_1, d_1')
(v_1, k_2, d_2)	(v_2, k_2, d_2)
(v_1, k_3, d_3)	(v_2, k_3, d_3)

Fig. 2. Database with two versions.

(v_1, k_1, d_1) (v_2, k_1, d_1')
$(\{v_1, v_2\}, k_2, d_2)$
$(\{v_1, v_2\}, k_3, d_3)$

Fig. 3. Records are associated with a set of versions.

key, and the data. So with two versions, we get six records. **Keys** are version-invariant fields which do not change when a record is updated. For example, if records represent employees, the key might be the social security number of the employee. When the employee's salary is changed in a new version of the database, a new record is created with the new version label and the new data, but with the old social security number as key.

Figure 1 gives the records in version v_1. The k_i's are the version-invariant keys, which do not change from version to version. The d_i's are the data fields. These can change. Now let us suppose that in the second version of the database, v_2, only the first record changes. The other two records are not updated. We indicate this by using d_1' instead of d_1 to show that the data in the record with key k_1 has changed. We now list the records of both v_1 and v_2 in Figure 2 so they can be compared.

Note that there is redundancy here. The records with keys k_2 and k_3 have the same data in v_1 and v_2. The data has not changed.

What if instead of merely three records in the database there were a million records in the database and only one of them was updated in version v_2? This motivates the idea that the records should have a representation which indicates the set of versions for which they are unchanged. Then there are far fewer records. We could, for example, list the records in Figure 3.

Indicating the set of versions for which a record is unchanged is in fact what we shall do. However, in the case that there are a large number of versions for which a record does not change, we would like a shorter way to express this than listing all the versions where there is no change. For example, suppose the record with key k_2 is not modified for versions v_1 to v_{347} and then at version v_{348} an update to the record is made. We want some way to express this without writing down 347 version labels. One solution is to list the start and the end version labels, only. But there is another complication. There can be more than one end version since in some application areas, versions can branch [10,7,8].

2.2 Three-Version Example with Branching

Now we suppose we have three versions in the database. When we create version v_3, it can be created from version v_2 or from version v_1. In the example in Figure 4, we have v_3 created from v_1 by updating the record with key k_2. The record with key k_1 is unchanged in v_3. We illustrate the version derivation history for

version v_1	version v_2	version v_3
(v_1, k_1, d_1)	(v_2, k_1, d'_1)	(v_3, k_1, d_1)
(v_1, k_2, d_2)	(v_2, k_2, d_2)	(v_3, k_2, d'_2)
(v_1, k_3, d_3)	(v_2, k_3, d_3)	(v_3, k_3, d_3)

Fig. 4. Database with three versions.

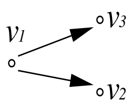

Fig. 5. Version tree for the three-version example.

$(\{v_1, v_3\}, k_1, d_1)$	(v_2, k_1, d'_1)
$(\{v_1, v_2\}, k_2, d_2)$	(v_3, k_2, d'_2)
$(\{v_1, v_2, v_3\}, k_3, d_3)$	

Fig. 6. Records are listed with a set of versions.

this example in Figure 5. Now we show the representation of the records in this example using a single version label with each record.

We list with each record the set of versions for which they are unchanged in Figure 6.

We see that we cannot express a unique end version for a set of versions when there is branching. There is a possible end version on each branch. So instead of a list of versions we might keep the start version and the end version *on each branch.*

However, we also want to be able to express "open-endedness". For example, suppose the record with key k_3 is never updated in a branch. Do we want to keep updating the database with a new version label as an "end version" for k_3 every time there is a new version of the database in that branch? And what if there are a million records which do not change in the new version? We would have to find them all and change the end version set for each record. We shall give a representation for end sets with the property that only when a new version updates a record need we indicate this in the set of end versions for the original record.

To explain these concepts more precisely, we now introduce some formal definitions.

2.3 Versions

We start with an initial version of the database, with additional versions being created over time. **Versions V** is a set of versions. Initially $V = \{v_1\}$, where v_1 is called the **initial version**. New versions are obtained by updating or inserting records in an old version of V or deleting records from an old version in V. (Records are never physically deleted. Instead, a kind of tombstone or null record is inserted in the database.)

The set of versions can be represented by a tree, called the **version tree**. The nodes in the version tree are the versions and they are indicated by version labels such as v_1 and v_2. There is an edge from v_j to v$_k$ if v_k is created by modifying (inserting, deleting or updating the data) some records of v_j. At the time a new version is created, the new version becomes a leaf on the version tree. There are many different ways to represent versions and version trees, e.g.[2]. We do not discuss these versioning algorithms here because our focus is an access method for versioned data, not how to represent versions. The version tree of our three-version example is illustrated in Figure 5.

Temporal databases are a special case of versioned databases where the versions are totally ordered (by timestamp). In this case, the version tree is a simple linked list.

We denote the partial order (resp. total order for a temporal database) on the nodes (versions) of the version tree with the “less than” symbol. We say that for $v \in V$, **anc(v)** $= \{a|a < v\}$ is the set of **ancestors** of v. The set **desc(v)** $= \{d|v < d\}$ is the set of **descendents** of v. A version v_k is **more recent** than v_j if $v_j < v_k$ (i.e. $v_k \in desc(v_j)$).This is standard terminology. For our three-version tree in Figure 5, desc$(v_1) = \{v_2, v_3\}$, anc$(v_3) =$ anc$(v_2) = \{v_1\}$ and v_2 and v_3 are more recent than v_1.

2.4 Version Ranges

As we have seen in the two-version and three-version example above, records correspond to sets of versions, over which they do not change. Such a set of versions (and the edges between them) forms a connected subset of the version tree. We call a connected subset of the version tree a **version range**. (In the special case of a temporal database a version range is a time interval.) We wish to represent records in the database with a triple which is a version range, a key and the record data. We show here how to represent version ranges for records in a correct and efficient way.

A connected subset of a tree is itself a tree which has a root. This root is the **start version** of a version range. Part of our representation for a version range is the start version. We have seen that listing all the versions in a version range is inefficient in space use. Thus, we wish to represent the version range using the start version and *end versions on each branch.*

The major concern in representing end versions along a branch is that we do not want to have to update the end versions for every new version for which the record does not change. We give an example to illustrate our concern.

Let us look at Figure 7(a). Here we see a version tree with four nodes. Suppose the version v_4 is derived from v_3 and the record R with key k_3 in our (three-record) database example is updated in v_4. So we might say that v_3 is an end version for the version range of R. However, the Figure 7(b) shows that a new version (version v_5) can be derived from version v_3. If v_5 does not modify R, v_3 is no longer an end version for R. This example motivates our choice of “end versions” for a version range to be the versions where the record has been modified. The end versions will be “stop signs” along a branch, saying “you can’t go beyond here.” End versions of a version range will not belong to the version range.

For our example with R in Figure 7(b), we say the version range has start version $= v_1$ and end version $= v_4$. The set of versions inside the version range where R is not modified is $S = \{v_1, v_2, v_3, v_5\}$. Later, any number of descendents of versions in S could be created. If these new descendents do not modify R, one need not change the end set for the version range of R, even though the

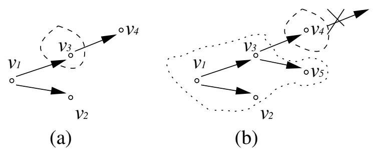

Fig. 7. Version range of R can not go further along the branch of v_4.

$((v_1, \{v_2\}), k_1, d_1)$	$((v_2, \{\ \}), k_1, d_1')$
$((v_1, \{v_3\}), k_2, d_2)$	$((v_3, \{\ \}), k_2, d_2')$
$((v_1, \{\ \}), k_3, d_3)$	

Fig. 8. The three-version example with version range[1].

version range of R has been expanded. No descendent of v_4, however, can join the version range of R. Now we give a formal definition for end versions of a version range.

Let vr be a version range (hence a connected subset of the version tree). Let $start(vr)$ be the start version for vr. Remember that "<" is a partial order, so saying $\neg(a <= b)$ does not imply that $a > b$. Given these preliminaries we state our definition as a minimality constraint on a set of versions.

The **set of end versions for** vr (denoted $end(vr)$) is the minimal set of versions ev with the property that $v \in vr$ if and only if $start(vr) <= v$ and $\forall ev \in end(vr), \neg(ev <= v)$. That is, the set of end versions is the smallest set of versions such that elements of vr other than start(vr) are descendents of start(vr) which are not end versions nor descendents of end versions.

Saying that the set of end versions with this property is minimal implies two interesting properties of end versions:

1. End versions must be descendents of the start version. Otherwise they could be on some other branch, neither a descendent nor an ancestor of the start version and hence redundant.
2. End versions cannot be ancestors or descendents of one another. Otherwise, the more recent one would be redundant.

Using the definitions in this section, we represent records with a three-tuple: (version range vr, key, data). The version range is in turn a pair $(start(vr), end(vr))$. The three-version example is thus represented in Figure 8.

3 Pagination

In this section, we show how to store records in storage units (usually disk pages) which partition the version-key space and produce good access properties. Let us call the storage units "pages". We will only look at data pages in this section. In the next section we will look at the index pages which direct search to data pages.

[1] In figures we use { } to represent the null set, whereas in the text we use $\emptyset$

3.1 Data Pages

Data pages correspond to one version range and one key range. A **key range** for a page P is of form $[LowKey(P), HighKey(P))$. (Key ranges are half-open.) (We consider only one-dimensional key spaces in this discussion.) Keys of records stored in a data page P always lie within the key range of P. Version ranges of a record stored in P always have a non-empty intersection with the version range of P.

A **key-version range** (kr, vr) is a combination of key range kr and version range vr. We denote $\mathbf{KR}(\mathbf{P})$ as the key range of page P, $\mathbf{VR}(\mathbf{P})$ as the version range of page P and $\mathbf{KVR}(\mathbf{P})$ as the key-version range of page P. Using this notation, a data page D with $KVR(D) = (kr, vr)$ stores all records (vr', k, d) such that $k \in kr$ and $vr \cap vr' \neq \emptyset$.

Two key-version ranges (kr_1, vr_1) and (kr_2, vr_2) **intersect** when $kr_1 \cap kr_2 \neq \emptyset$ and $vr_1 \cap vr_2 \neq \emptyset$. The set of data pages partitions the key-version space. This implies no two distinct data pages have intersecting key-version ranges and every point in key-version space is in exactly one data page.

3.2 Compact Record Representation in Pages

It is possible to omit the end versions of a version range when storing a record in a data page and still have correct search. When we do this we say that we have a **compact-record representation**. This not only saves space, it makes updates very easy. The record being updated does not need to be found or modified; one only inserts the new record with the new data and the new start version and the same key.

In the three-version example, if we use the usual representation of version ranges as a pair (start version, set of end versions) we have Figure 9(a).

In this example, the end version set for the first record, $R1$, with key k_1 is $\{v_2\}$, indicating that $R1$ was updated in version v_2 to create a new record. The start version of a new record (updating a previous record) is the same as an end version of the previous record with the same key. We use this redundancy to eliminate listing end versions of version ranges for records in data pages. Let vr be a version range and let (vr, k, d) be a record in a page P. We say $(start(vr), k, d)$ is a **compact record**. The representation of the three-version example using compact records is shown in Figure 9 (b). As we can see, the two different representations of version ranges can be constructed from one other. So in the rest of the paper, without lose of generality, we will adopt the compact record representation.

Search for a given key k and version v which has been directed to page P must look at all the records in P with key k and find the one whose start version sv is the most recent one such that $sv \leq v$.

If only the start versions, and not the end versions are stored, one must explicitly mark deletion events to indicate that along some branch, a record is no longer there. For this reason we define null records.

A **null record** is a triple $(vr, k, null)$ where for each $v \in vr$, the versioned record corresponding to key k has been deleted. A null record is really a marker indicating that there is no data associated with version range vr and key k. If $(vr', k, null)$ is a null record we say $(start(vr'), k, null)$ is a **null compact record**.

From now on, (v, k, d) means a compact record, and in the special case when $d = null$, $(v, k, null)$ is a null compact record. Here, v is the start version for the version range of the record.

3.3 Operation Properties for Efficiency

In the next few subsections, we discuss page splitting and page consolidation. The goal in these operations is to produce efficient stabbing queries without too much replication. We will show the operations do yield efficient queries. The replication factor has been measured experimentally in many papers (in particular, [11]) not to be “too bad”; at most an average of three times the size of the database with no replication and no empty space, a good trade-off for the query efficiency.

To be deemed “efficient for stabbing queries” the access method should have the property that whenever a data page is accessed in a stabbing query for version v, a substantial percentage of the records in the page are alive for v. (A record is **alive for** v if its version range contains v.) After describing current-version splitting, key splitting, version-and-key splitting and page consolidation, we shall show under what conditions efficiency guarantees for the stabbing query can be made.

3.4 Splitting by Current Version

A **current version** is a leaf of the version tree. When new updates, deletes or inserts are made by a version v which is a current version, they should be inserted into the data page P whose key range contains the key of the update and whose version range contains the parent of the new (current) version v in the version tree. However, if P is full, a new page P' must be allocated. The page P' will contain the new record. The records of P which were updated by v will be moved to page P' and some of the records in P will be copied to page P'.

The new version v will become an end version for VR(P). The version range for P' will be $(v,\emptyset)$. This is called **current-version splitting**. In this section, we always split by a current version, i.e., a leaf of the version tree. (In some papers we discuss in the related work section [11,8] splitting by non-current versions is suggested.)

Records Copied or Moved to the New Page. The records which are copied to the new page P' are those whose version range intersects both the version range of P' and the version range of the old page P. The records which are

$((v_1, \{v_2\}), k_1, d_1)$
$((v_1, \{v_3\}), k_2, d_2)$
$((v_1, \{\ \}), k_3, d_3)$
$((v_2, \{\ \}), k_1, d_1')$
$((v_3, \{\ \}), k_2, d_2')$

(a)

(v_1, k_1, d_1)
(v_1, k_2, d_2)
(v_1, k_3, d_3)
(v_2, k_1, d_1')
(v_3, k_2, d_2')

(b)

Fig. 9. Three version example with its compact record representation.

(v_1, k_1, d_1)
(v_1, k_2, d_2)
(v_1, k_3, d_3)
(v_2, k_1, d_1')

(a)

(v_1, k_1, d_1)	(v_1, k_1, d_1)
(v_1, k_2, d_2)	(v_3, k_2, d_2')
(v_1, k_3, d_3)	(v_1, k_3, d_3)
(v_2, k_1, d_1')	

(b)

Fig. 10. When (v_3, k_2, d_2') is inserted, page D is split by current version v_3.

moved are records in P whose start version is v, and which are not null records. Null records only mark the end of a version range for another record, so there is no need to copy them to the new page if they do not have that function there.

More precisely, Let D be a data page identified by a key version range (kr, vr). We define **contents(D)** $= \{(v, k, d)|(v, k, d)$ is a compact record in $D\}$. We now define the subset of contents(D) which will be moved or copied to a new page during a current-version split.

Let v_n be the new version which makes an update causing D to be current-version split. The set of compact records *moved* from D to the new page is:

$$\{(v', k, d)|(v', k, d) \in contents(D) \wedge ((v' = v_n) \wedge (d \neq null))\}$$

This is the set of records created by v_n. This happens when the new version updated several records in D and the first few fit in the page, but at some point the page D became full and further updates by v_n required a split. No null records are moved.

Let T be a logical (not physical) temporary page holding records created by v_n with key in KR(D). The set of compact records of page D to be copied to the new page is defined to be:

$$\{(v', k, d)|((v', k, d) \in contents(D)) \wedge (v' < v_n \wedge d \neq null) \wedge$$
$$(\forall (v'', k, d') \in contents(D \cup T) \text{ such that } (v'' \leq v_n \wedge v'' \neq v'), v'' < v')\}$$

When we copy records from D to the new page, we do not want to copy any with the same key as any record in T. The above definition for copied records has this property. In the case, where a key k is *not* a key of a record in T, the record in D with key k having start version as the most recent ancestor of v_n is copied. Null records are not copied.

Let us give an illustration using the two version example and the three-version example. Suppose we have in page D our two-version records, create by v_1 and v_2 and represented as compact records as in Figure 10 (a).

Suppose D can only hold 4 records. Now we update the record with key k_2 in v_3 as before. We then have the records in the new page, D' as shown in Figure 10 (b).

We have copied the two records which are not changed by v_3 and we have inserted the new updated record. The record created by version v_2 is not included in the new page because its start version is not an ancestor of v_3. All three records in D' are alive for v_3. The upper levels of the index will be directing search for v_1 and for v_2 to D and for v_3 to D'.

When we copy a compact record to a new page, we do not change its start version even if the start version is not in the version range of the new page. In the example in Figure 10(b), we retained the start version v_1 in the two moved records even though v_1 is not in $VR(D') = (v_3, \emptyset)$. There are several reasons for this:

1. If a version range (or time interval) query (rather than a stabbing query) is made, we will be able to recognize identical records obtained from different data pages. (This is a query to find all the records alive in a version range.)
2. Copying is easier. No changes are made to the copied records.
3. Search within a page is unchanged and still correct.
4. Finding the set of historical records with the same key may have less disk accesses. For example, given the most recent version number sv, to find all historical records of key k_1, we can search the index pages for key k_1 and version $v < sv$ to find the previous versions. Otherwise, search will be less efficient if a record of this version is copied over many pages.

3.5 Key Splits and Version-and-Key Splits

We will also be splitting data pages by key. For this we define subsets of contents of pages which fall within a given key range. Splitting pages by key is done exactly like in B-trees: a split key sk is chosen in KR(P). Then all records with key less than sk remain in P and all records with key greater or equal to sk are *moved* to the new page.

If the number of records copied or moved to a new data page during a current-version split is above a certain threshold value T_k, a version-and-key split is made. Here a current-version split is followed by a key split. Note that $T_c < \lfloor T_k/2 \rfloor$ where T_c is the threshold for consolidation and T_k is the threshold for version-and-key split.

A key split instead of version-and-key split will be used if the full page has version range $(v, \emptyset)$, where v is the current version. This can happen when a transaction makes multiple updates. Figure 11 is an example. Assume v_2 is the current version. VR(P_2) $= (v_2, \emptyset)$. Assume maximum page capacity is 4. When a record (v_2, k_7, d_7) is inserted into P_2, a version-and-key split will be triggered, as shown in figure 11(b). Actually the version split is not necessary since the version range of P_2 is only one version. In this situation, a pure key split, as shown in figure 11(c), should be used instead. After the split, P_3 will be posted to the same parent as P_2. It is the only parent of P_3. The pure-key-split problems mentioned later in this section and in section 4.1 will not happen in this situation because the version range here contains only the current version. Note that this is the only situation where a key split is not combined with a version split. We call this a **restricted key split**. It is restricted to the case when the (old) full page version range contains only one version.

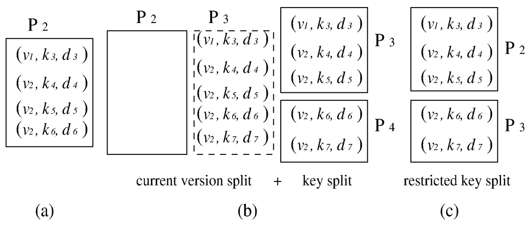

Fig. 11. When (v_2, k_7, d_7) is inserted, a restricted key split instead of a version and key split is used.

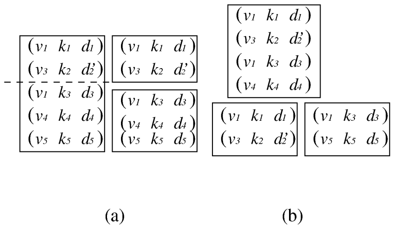

Fig. 12. After (v_4, k_4, d_4) and (v_5, k_5, d_5) are inserted, D' need to be split. (a) Pure key split with split key $= k_3$. (b) Version-and-key split: first split at version v_5 and then key split at k_3.

Our framework does not include pure key splits other than restricted key splits as in figure 11, only version-and-key splits and version splits. Here is an example to explain why we never do non-restricted key splits.

Look at D' in Figure 10(b). There are three records in D', all alive for v_3. Now suppose we insert into D' the record (v_4, k_4, d_4) using the version tree from Figure 7(b). At this point there are three records alive in D' for v_3 and four for v_4.

Now we wish to insert (v_5, k_5, d_5) in D' but D' is full. We shall use the version tree in Figure 7(b) for v_5 also, so we have v_4 and v_5 in $desc(v_3)$. Suppose we do a pure key split by split key $sk = k_3$, assuming $k_1 < k_2 < k_3 < k_4 < k_5$.

As shown in Figure 12(a), in the old page D' we have two records alive for v_3, v_4 and v_5. In D'', the new page with the higher key values, (v_1, k_3, d_3) is the only record alive for v_3, and (v_1, k_3, d_3) and (v_4, k_4, d_4) are alive for v_4 and (v_1, k_3, d_3) and (v_5, k_5, d_5) for v_5. The point is that in D'' we now have *only one record* alive for v_3. *Pure key splits cannot give good guarantees for numbers of records alive for a given version after the split unless the version range of the original page contains just one version (the restricted key split case).*

If we had split by v_5 first, and then done a key split by k_3, as we do in Figure 12(b), we would get two pages whose version ranges are both $(v_5, \emptyset)$ and both would have two records alive for v_5. The original D' would have 4 records, three alive for v_3 and four for v_4 as before.

3.6 Consolidation

In B-trees, pages are consolidated when their *contents* falls below a certain level. In versioned access methods, pages never lose contents from record deletions, which are logical, not physical. However, the number of records in the page satisfying the "stabbing" query ("Find all data alive for this version") may fall below an acceptable threshold T_c.

Let **pageSlice**(D, v) be the set of records in D whose version range contains v. This is the set of records **alive** in D at version v. After a record is deleted from

D, one checks to see if $|pageSlice(D, v)| < T_c$ where v is the version of the delete operation and T_c is the threshold. If so, we say D is **sparse** and we attempt to perform a page consolidation on D.

Consolidation is allowed when there is a suitable **sibling** with which to consolidate: another page with the same parent index page and with an adjacent key range. In this case, a current-version split is made first, both on the sparse page and on its sibling. The two new pages are then combined. If the combined page has too many records, a key split is made.

There are very few scenarios where a suitable sibling would not be available. This would happen when the whole database for a given version v fits in one data page and then only current-version splits are made (no version-and-key splits). This could happen near the creation time of the database until a sufficient number of insertions are made, or it could happen in a highly degenerate case when so many deletions were made that either one data page would hold all the records alive for some version v or there are too many null records to fit in one data page. (It is not possible that one data page becomes sparse when deleting at v and has no sibling while another data page (with a different parent) has records alive at v because upper levels would have consolidated before that happened.)

In the case when a transaction makes a large number of deletes, a special problem occurs. Let us look at an example in figure 13. Assume a transaction that creates the current version v_2 deletes all four records in page P_1 and inserts one record with key k_3. Assume the maximum page capacity is 5. After record $(v_2, k_1, null)$ is inserted in P_1 and an attempt is made to insert (v_2, k_3, d_3) in P_1, P_1 is version split as shown in figure 13(b). Now P_1 has v_2 as the end version of its version range. VR(P_2) is $(v_2, \emptyset)$. Some of the records in P_2 in figure 13(b) are "temporary records", which will be replaced by records of the current version with the same key. For example, (v_1, k_2, d_2) will be replaced by $(v_2, k_2, null)$ and (v_1, k_4, d_4) will be replaced by $(v_2, k_4, null)$. Note that this replacement only happens when the page's version range is $(v_{current}, \emptyset)$. After replacing these records, P_2 becomes sparse as shown in figure 13(c). Say that there is a sibling P_5, described in figure 13(d), with which P_2 can be consolidated. We do a version split on v_2 for P_5 and a version split on v_2 for P_2 (meaning here, we only copy live records) and obtain a new consolidated page P_3 with version range $(v_2, \emptyset)$. We now have two pages P_2 and P_3 with the same version range and overlapping key ranges. For this case, consolidating a sparse page whose version range is only one version, we call P_2, as in figure 13(d), a **ghost page**. A ghost page has a **ghost mark** in its parent indicating that it is NOT to be used in any search not **strictly including** its one version. (A range strictly includes a version v if v is in the range and is not the start version of the range.) This rules out using ghost pages in exact match search. The purpose of maintaining ghost pages is merely to facilitate version range searches in determining end versions of records. We anticipate few ghost pages in most applications since massive deletions are rare. Following our policy for moving records created by split versions, P_2 now contains only null records as in figure 13(d).

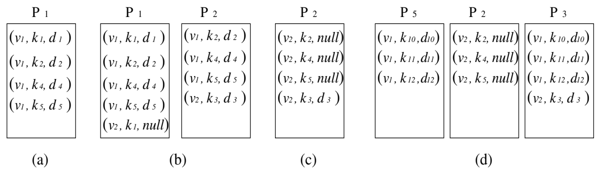

Fig. 13. After deletions and consolidation with P_5, all records in P_2 will be null records. P_2 is called ghost page.

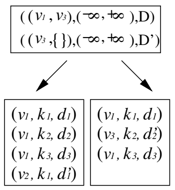

Fig. 14. Index page and data pages for the three-version example.

3.7 Stabbing Query Efficiency

The following assertions illustrate why copying some records as we do in version splitting, version-and-key splitting and consolidation helps stabbing queries to be efficient. In what follows, we assume that we start with one page D with the initial version v_1 having n alive records. The first assertion arises from the observation that if only inserts and updates are made, no version can have less than the number of records alive for v_1, the initial version. If, in addition, only version-splits are made, all the records alive for the split version are copied or moved into the new page.

Assertion 1 *If only version splits are made and there are only inserts and updates (no deletes), then for any data page D and any version $v \in VR(D)$, there will be at least n records in D satisfying the stabbing query for v.*

If we also do version-and-key splits, and assume T_k is the threshold for version-and-key splits, we get our second assertion. This is due to the observation that version-and-key splits only occur when the number of records alive for the splitting version to be copied or moved is greater than T_k, so the number in each of the two new pages is at least $T_k/2$.

Assertion 2 *If we do only updates and insertion and have only current-version or version-and-key or restricted-key splits, the stabbing query for $v \in VR(D)$ will obtain at least min(n, $T_k/2$) records in P.*

Now allow deletes and let T_k be the threshold for version-and-key split and let T_c be the threshold for consolidation. We get a third assertion.

Assertion 3 *If it is always possible to find a sibling for node consolidation when $|pageSlice(v)| < T_c$ then we can guarantee the stabbing query for $v \in VR(D)$ will obtain at least $min(T_c, n)$ records in D, allowing version splits, version-and-key splits, restricted-key splits and node consolidation. (Note that ghost page will be not used for consolidation or stabbing query.)*

This shows that the stabbing query for v will be efficient since search in upper levels of the access method, as we show in the next section, will only retrieve data pages D with $v \in VR(D)$. In each of these accessed data pages, we have shown that at least $min(T_c, n)$ records satisfying the query will be found (provided that consolidation siblings are always available when needed).

4 Upper Levels

In this section we consider index pages, which direct search, as well as data pages. Let P, C be two (index or data) pages. We say page C is a **child page** of page P if the disk address of page C and some description of the key-version range of C is stored in page P. We will use **children(P)** to denote the set of child pages of page P. If $C \in \mathit{children}(P)$, we say page P is a **parent page** of page C. We will use **parents(C)** to denote the set of parent pages of page C.

The set of index pages and data pages form a Directed Acyclic Graph, or DAG. If C is a child page of P, there is an edge from P to C. Data pages do not have any outgoing edges. They are all leaves of the DAG. Two pages which are the same distance from the set of data pages are said to be at the same **level**. All the pages at levels above the data pages are index pages.

Index pages also correspond to key-version ranges. The set of index pages at a given level partitions the key-version space. An index page P with $KVR(P) = (kr, vr)$ channels searches for the version, key pair (v, k) with $v \in vr$ and $k \in kr$.

The contents of an index page are references to its children and we will use a list of the children of an index page I as **contents(I)**. In Figure 14, we show the index page and two data pages for the three-version example when the data page has split at v_3. An entry in an index page referencing a child C is of the form (start($VR(C)$), end($VR(C)$), $KR(C)$, disk page address(C)). (In the related work section, we will discuss some alternative forms for child entries in index pages.)

Access methods that fit our framework satisfy the following:

Invariant 1 *If page C is one level below page P and $KVR(P)$ intersects $KVR(C)$, then page $C \in \mathit{children}(P)$.*

At each level, since Invariant 1 is true, it is possible to decide exactly which page to access on the next level. For exact match search (search on one version and one key) there is only one page to visit at each level.

4.1 Index Page Splits and Consolidations

Index page splits and consolidations are similar to those of data pages. A current version split *copies* entries whose version ranges intersect the version range of both the old page P and the new page N. Any child entry whose version range lies only in $VR(P)$ stays in P. Any child entry whose version range lies only in $VR(N)$ is *moved* to N.

Since, in index page version splits, children entries can be copied from P to N, this creates multiple parents for these children. This is why the access method is a DAG and not a tree.

Now for index pages, we need to take into account that children pages have a key *range*, unlike data records, which have only a single key value. In this case there is an additional reason why it is desirable to do no pure key split without a version split first.

It is unlikely that for a given index page I, there is a key value k such that for every child C of I, either $k >= HighKey(KR(C))$ or else $k <= LowKey(KR(C))$. Thus, if we do a pure key split, we will probably have to copy child entries whose key range intersects the key range of both the new and old index page. Consider for example a database which starts with one data page D and then does a version-and-key split with split key sk, creating new data pages D' and D''. If we use sk as a split key for the parent index page I, some records in D will have keys greater than sk and others will have keys larger than sk. Thus, D will be a child both of I and of the new index page I'.

If, on the other hand, we do a current-version split first, we can choose a split key which is a boundary between two of the children and all the other children also have key ranges strictly above or strictly below the split key. In this case, we need not have copies of the same children entries in both two pages resulting from the key split.

When version splits occur on root nodes, previous work has considered two strategies. One is to increase the height by creating a new root with the old root as its child [7,8,11]. The other strategy is to maintain multiple roots and create a forest with shared subtrees [1,4,10]. In this case, when a version split occurs at a root, the new page becomes an additional root. A directory is kept with the addresses and version ranges of each root. Different trees have different heights and cover disjoint version ranges. Single root methods have the property that pages on each level partition the version-key sparce. Multiple root methods have the property that pages of each level within a given tree (under one root) partition the version range of the tree and the key range.

Consolidation of an index page I is indicated when consolidation of some of children(I) at some current version v has resulted in too few children of I alive for v. That is,

$$|\{P|(P \in children(I)) \text{ and } (v \in VR(P))\}| < T_{ci},$$

where T_{ci} is a threshold for index page consolidation. We say that the **fan-out** of I at v is **sparse**. In this case, as with data page consolidation, we find a sibling and do a current-version split on both the sparse page and its sibling and combine the result into one or two new index pages.

Before children are unable to consolidate because there is no suitable sibling for a given version v, the parent must have sparse fan-out at v. Thus the parent will consolidate with another index page on the same level, gaining suitable siblings for its child. This is why not finding suitable siblings for consolidation is unusual and only occurs in the degenerate cases we discussed before.

The index page splitting and consolidation definitions above guarantee the following: if any index page P satisfies Invariant 1, then any resulting page R from splitting or consolidating page P satisfies Invariant 1 too.

4.2 Posting

In order to have correct search, when a split or a consolidation takes place, information about the new page(s) N and the new boundaries of the old page

P must be posted to the parents of P. If this information were posted to all the parents of P, it is clear that Invariant 1 would still hold. But in fact, if we do current-version splitting and no pure key splits (no key splits that are not version-and-key splits nor restricted-key splits) less is needed. Posting need take place to only one parent.

Let v be a current version. If N is a new page created from any split or consolidation, $\mathrm{VR}(N) = (v, \phi)$. (This is *not* true if we allow pure key splits or splitting at other than current versions.) Further, since there are no pure key splits on index pages, for all index pages I, if $P \in \mathrm{children}(I)$, $\mathrm{KR}(P) \subseteq \mathrm{KR}(I)$. So there is one index page I among the parents of P such that $\mathrm{KVR}(N) \subseteq \mathrm{KVR}(I)$. This is the only parent where posting takes place.

5 Related Work

In this section, we outline how the methods proposed in the literature fit or do not fit our framework. Note that most of these methods are called "trees" although they are DAGs. (When restricted to one version, each of these DAGs is a tree.) None of these methods consider the problems of versions with multiple updates as we have done.

In [4], a write-once optical disk is used and the storage units are sets of optical disk pages. Since an update of optical disk data at the time the paper was written required indelibly burning about 1Kbyte of data and 300 bytes of checksum, it was not possible to go back and insert endpoints to version ranges of records. So the compact representation of records is used. This is a linear version tree, or temporal access method. It is presented as a way to store a B-tree and update it even though old versions had to be kept (because they could not be erased). There is no page consolidation. The multiple root strategy is used. This is called the Write-Once B-tree, or WOBT.

Another paper, [1] does have page consolidation and it does not have compact record representation in data pages. This is also a temporal access method with multiple roots. It is called MVBT, or Multi-version B-tree.

The paper [13] is based on the observation that page consolidation is done on *sparse* pages which however are not necessarily *full* pages. There is empty space in these pages. This paper places two or more logical pages (with a key range and time interval) in one physical page. There are then multiple references to a physical child page in a parent page. This increases space utilization. This is a temporal method.

The Fully Persistent B-tree [10] has page consolidation. It does not use the compact record representation. It has extra "version blocks" in the index levels which make the height of the "tree" larger than need be. It uses multiple roots.

(Versioned access methods are called **fully persistent** [3] if any version can be updated creating a new version. This causes branching in the version tree. A **partially persistent** access method only allows update on a current version, creating a linear version tree. Temporal access methods are partially persistent.)

The BT-tree, or Branched and Temporal tree [7] is also a fully persistent (branched) access method. It does page consolidation and it uses the compact data record representation. In index pages, instead of using the child entries we have described, a small binary tree called a **split history** or **sh** tree is used. This directs search depending on the key values and version values in the internal sh-tree nodes. The leaves of the sh-tree are child page addresses. The BT-tree has a single root.

All of the above methods do only current-version splits and version-and-key splits and no pure key splits. The next two methods allow splitting at versions other than the current version. As in current-version splitting, records whose version range is in the version range of both pages are copied and records whose version range is only in the new page are moved to the new page. The difference is that the set of moved records is larger than just those created by the splitting version.

The TSB-tree [11] is a temporal method and uses compact representation of records. It has no page consolidation. It has a single root. To save space and make retrieval quicker, pure key splits and non-current version splitting are allowed. In order to make posting to only one parent possible, it is required to split index pages I at a version v with the property that for all *current* children C of I, $v \leq start(VR(C))$. (**Current pages in a temporal access method** have $end(vr) = \emptyset$.) This results in current pages (the only ones that are split in a temporal access method) having only one parent.

The other paper to consider non-current version splits is the BTR-tree [8]. This is done to reduce the number of copies of records made when there is a great deal of branching. To achieve single-parent posting, only certain versions can be used for splitting. The set of possible splitting versions is derived from information gathered during the search. The BTR-tree uses compact data record representation and it supports page consolidation. It uses an sh-tree in index pages. It has a single root.

Recently, there have been some methods proposed for spatial and moving objects data (spatial-temporal data) which use current-version splitting. For example, [12] [6] both do version-splitting on an R-tree. Since the R-tree has spatial overlapping, neither satisfies Invariant 1 (with the key range understood to be a spatial key range). Thus exact match search (for a key and version) requires backtracking. On the other hand [9] is based on [5] (the hB-Pi tree), which is a spatial method without overlapping, so Invariant 1 is satisfied. The paper [9] uses the compact data record representation.

6 Summary

In this paper, we have presented a framework for versioned access methods. Records are associated with version ranges, which are connected subsets of the version tree. A definition for end sets for version ranges using minimality was given. Compact record representation, using only the start version of the version range, was introduced with its benefits in algorithmic simplicity and space usage.

We have shown, for the first time, how to handle versions which contain multiple updates. Previous work made the unrealistic assumption that each update was in a different version, created by a different transaction.

Current-version splits, version-and-key splits and consolidations were discussed and their effects on stabbing query efficiency were presented. For upper levels of the index, an invariant was introduced which allows visiting only one page at each level of the access method when doing exact-match search (no backtracking). Splits and consolidations of index pages preserve this invariant.

References

1. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. On optimal multiversion access structures. In *Proc. Int. Symp. on Spatial Databases*, pages 123–141, Singapore, 1993.
2. Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, 1987.
3. James R. Driscoll, Neil Sarnak, and Daniel D. Sleator. Making data structure persistent. *Journal of Computer and System Sciences*, 38, February 1989.
4. M. C. Easton. Key-sequence data sets on indelible storage. *IBM J. Res. Development*, pages 230–241, 1986.
5. Georgios Evangelidis, David B. Lomet, and Betty Salzberg. The hB-Pi-Tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Jounal*, pages 1–25, January 1997.
6. Marios Hadjieleftheriou, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. Efficient indexing of spatiotemporal objects. In *EDBT 2002, LNCS 2287*, pages 251–268, 2002.
7. Linan Jiang, Betty Salzberg, David Lomet, and Manuel Barrena. The BT-Tree: A branched and temporal access method. In *International Conference on Very Large Data Bases*, pages 451–460, 2000.
8. Linian Jiang, Betty Salzberg, David Lomet, and Manuel Barrena. The BTR-Tree: Path-defined version-range splitting in a branched and temporal structure. In *Proceedings of the Eighth International Symposium on Spatial and Temporal Databases, SSTD 2003, Santorini Island, Greece, LNCS 2750.*
9. Evangelos Kanoulas and Georgios Evangelidis. Indexing of spatiotemporal data with the hB-Pi Tree. In *HDMS'02 1st Hellenic Data Management Symposium*, Athens, Hellas, July 2002.
10. Sitaram Lanka and Eric Mays. Fully persistent B^+-trees. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 426–435, 1991.
11. D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 354–363, 1990.
12. Yufei Tao and Dimitris Papadias. The MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases*, pages 431–440, Sep. 2001.
13. Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. In *IEEE Transaction on Knowledge and Data Engineering*, pages 391–409, May/June 1997.