Indexing Multiversion Databases

Khaled Jouini
khaled.jouini@dauphine.fr
Geneviève Jomier
genevieve.jomier@dauphine.fr
Université Paris Dauphine
Place du Maréchal de Lattre de Tassigny
Paris, France

ABSTRACT

An efficient management of multiversion data with branched evolution is crucial for many applications. It requires database designers aware of tradeoffs among index structures and policies. This paper defines a framework and an analysis method for understanding the behavior of different indexing policies. Given data and query characteristics the analysis allows determining the most suitable index structure. The analysis is validated by an experimental study.

Categories and Subject Descriptors: H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing-indexing methods

General Terms: Theory, Performance.

Keywords: Index structures, branched and multiversion data, performance analysis, comparison.

1. INTRODUCTION

Many applications such as engineering design require the support of time evolving data with multiple lines of time evolution [6]. Temporal and multiversion databases dedicated to such applications, maintain several states of the modeled universe. Such a persistent state is called DataBase Version or DBV [2, 6].

A query to a multiversion database can be a version slice query, an historical query or a combination of both [6]. A version slice query searches for data within a DBV. An historical query follows the evolution of entities (objects or tuples) through a set of DBVs. Although many index structures have been proposed to optimize queries to temporal data, little work has been carried out on indexing data with branched evolution [6]. This paper compares different indexing techniques for such data. In particular, an analysis method based on the concept of data agility [7] and called the Steady State Analysis is proposed. The main contributions are: (i) the extension of temporal index structures to handle data with branched evolution; (ii) a comparative study of main trends in indexing multiversion data; (iii) a steady state analysis, which brings to light fundamental properties of different index structures and accurately predicts their behavior; (iv) an experimental validation of the analysis.

The paper is organized as follows. Section 2 introduces the main concepts. Section 3 describes different indexing structures. Section 4 briefly presents the steady state analysis. Section 5 validates the analysis through simulation. Section 6 concludes the paper.

2. CONCEPTS AND DEFINITIONS

Entity version and database version: an entity e is represented in each DBV v by a single version (e,v, val), where val is the value of the entity e in the DBV v. A DBV v contains one version of each entity represented in the database.

Except for the initial DBV v1, a new DBV is always derived from an existing DBV. Thus, the set of DBVs is structured in a genealogical tree, called the DBV tree [2] (figure 1.c). Since a DBV usually differs only partially from its parent, an entity e can keep the same value through a set of DBVs. To store unchanged entity version values only once the implicit sharing [2] is used: "if the value of (e,v) is not stored, its value is the same as that of (e,v'), where v' is the closest ancestor of v in the DBV tree, identifying a stored version of e". Figure 1 summarizes the former concepts.

DBV range: a DBV range R is a connected subset of the DBV tree (in the linear time case, R is a time interval) [6]. R is represented by a pair (vstart, vstop), where vstart is the DBV root of R and {vstop} a set of DBVs. A DBV vs appearing in {vstop} does not belong to R, but its parent does. Thus, vs explicitly terminates a branch starting at vstart. A particular value of R is (v,0), meaning that R is the subtree of the DBV tree, rooted at v.

3. MULTIVERSION DATABASE INDEXES

Three families of index structures are reviewed. The first family clusters entity versions by entity. It is represented by the B+V-tree proposed in this paper. The second family
clusters entity versions by DBV and is represented by a slightly modified Overlapping B+Trees (OB+tree) [1, 8]. The third family is a bi-dimensional index structure, represented by the Branched and Temporal Tree (BT-tree) [3].

3.1 B+V-tree

The use of the B+tree to index entity versions with linear evolution has been proposed in [5]. In our work, we extend the B+tree to handle data with branched evolution and call the resulting structure the B+V-tree. The B+V-tree clusters entity versions by entity and then by DBV. When a leaf $D_k$ overflows, it is split into $D_k$ and $D_l$, according to entity identifiers. This is shown in figure 2.c, where the update of $d$ in $v_3$ causes $D_2$ overflow: $c$ versions remain in $D_2$ and $d$ versions are moved to $D_l$.

If $D_k$ only contains versions of a single entity $c$, $D_k$ is split according to a DBV identifier $v_h$. Let $R$ be the DBV range corresponding to the versions of $e$ stored in $D_k$ before the split. $v_h$ is chosen, so that, $R$ is divided in two DBV ranges $R_1$ and $R_2$, where $R$ corresponds to the versions of $e$ remaining in $D_k$ and $R'$ to the versions of $e$ moved to $D_l$.

3.2 OB+tree

The OB+tree [1, 8] builds a B+-tree, denoted $B+(v)$, for each DBV $v$ and allows consecutive B+-trees to share unchanged nodes. First, the initial B+-tree, $B+(v_1)$, is created (figure 3.a). When a new DBV $v_2$ is derived from a DBV $v_1$, the root of $B+(v_2)$ is created. Its content is a copy of the root of $B+(v_1)$, so $B+(v_2)$ and $B+(v_1)$ share all their nodes except their root. When a shared leaf $D_k$ is affected by an entity modification, its sharing is broken: $D_k$ is duplicated in a new node $D_l$ where the modification is performed. After the duplication, $I_p$, the parent node of $D_k$, is modified to point to $D_l$: the sharing of $I_p$ between $B+(v_1)$ and $B+(v_2)$ is also broken. Node duplications continue up to the update of $B+(v_l)$ root (figure 3.b).

The OB+tree was proposed for data with linear evolution. We extend it to data with branched evolution by storing a

Figure 2: (a) B+V-tree at $v_1$. (b) Derivation of $v_2$, then update of $c$ in $v_2$. (c) Derivation of $v_3$, then update of $d$ in $v_3$.

Figure 3: (a) Initial B+tree $B+(v_1)$. (b) Derivation of $B+(v_2)$, update of $c$ in $v_2$. (c) Derivation of $B+(v_3)$, update of $d$ in $v_3$.

Figure 4: (a) $D_1$ contains entity versions of $v_1$. $D_1$ is entity split. (b) Update of $c$ in $v_2$. (c) The update of $d$ in $v_3$ causes $D_2$ version split.

Figure 5: Qualitative comparison

DBV range $R$ in the header of each node $D$. $R$ indicates the DBV where $D$ was created and the DBVs where it is duplicated, so it is easy: (i) to detect if $D$ is shared (ii) to find out the DBV range of the entity versions stored in $D$; this is useful for historical queries.

3.3 BT-tree

The BT-tree [3, 6] is a bi-dimensional index clustering entity versions on entity and DBV identifiers. An entry of an internal node is a triple $(e, v, I)$, where $e$ is an interval of entity identifiers, and $I$ the identifier of a child node. The leaves of the subtree rooted at the node $I$ contain the $(e, v)$, such that $e \in [e_{\text{min}}, e_{\text{max}}]$ and $v \in (v_{\text{start}}, v_{\text{end}})$. $I$ is said to be alive for each DBV $v \in (v_{\text{start}}, v_{\text{end}})$.

The BT-tree uses version split, entity split and combinations of both. An entity split is similar to a split in a B+-tree. It occurs when the overflowing leaf contains entity versions belonging to the same DBV (figure 4.a). A version split of a leaf $D_k$ separates its entity versions according to the DBV $v$ which triggers the split: (1) a new leaf $D_l$ is added to the $(v_{\text{end}})$ of $D_k$; (2) a new leaf $D_l$ with DBV range $(v, \emptyset)$ is allocated; (3) entity versions whose DBV range intersects both the DBV range of $D_k$ and $D_l$ are copied in $D_l$ (figure 4.b). After a version split, if the number of entity versions copied in $D_l$ exceeds a threshold $\theta$, the version split is followed by an entity split.

Whereas the B+V-tree allows entities to be modified in any DBV, the BT-tree and the OB+tree allow modifications only in the DBVs leaves of the DBV tree. Figure 5 synthesizes the main features of the different index structures.

4. STEADY STATE ANALYSIS

The steady state analysis aims at providing performance evaluation for the different index structures. Given data and application characteristics, the analysis provides guidelines for the selection of the most appropriate index structure.

A multiversion database at steady state is modeled by a set of $E$ entities and $V$ DBVs. Each entity $e$ is subject
to updates; each update occurring in a DBV $v$, generates a new version $(e,v)$, whose value is stored in the database. To be able to compare the different index structures, entity updates are only allowed in the DBVs leaves of the DBV tree. The proportion $a$ of entity versions updated in each DBV, called data agility in [7], is assumed to be constant. The total number of distinct entity versions is $E = E(1 + a(V-1))$. The $E$ entity versions are assumed to be indexed by a B+V-tree, an OB+tree and a BT-tree. The goal is to compare storage and query costs.

The storage cost is estimated by $N$, the number of leaves in the final index, and the redundancy factor. For the OB+tree and the BT-tree, $N$ is estimated by $n + m(V-1)$, where $n$ is the number of leaves in the initial index and $m$ the average number of new leaves created when $aE$ entities are updated in a DBV. $r$ is estimated by $\frac{\xi_E E}{E} = \frac{\xi_E}{2} - 1$, where $\xi_E$ is the number of stored entity versions.

The cost of a query $q$ is estimated by $P$, the number of visited leaves. Perfect match queries are distinguished from range queries. A perfect match query is assumed to involve a set of $q_v$ entries, and, for each entity, $q_v$ consecutive DBVs. A range query involves $q_r$ consecutive DBVs and $q_r$ entities with consecutive identifiers.

### 4.1 B+V-tree

#### 4.1.1 Estimation of $N$, average number of leaves

Let $b$ be the average number of entity versions stored in a leaf. $N$ is estimated by

$$N = \frac{E}{b} = E(1 + a(V-1))$$

#### 4.1.2 Estimation of $r$, redundancy factor

Since the B+V-tree has no version redundancy, $r = 0$.

#### 4.1.3 Estimation of $P$, number of visited leaves

Perfect match queries: due to the lack of space, here we only consider the common case where all the versions of an entity fit in a single leaf. Given a search for version(s) of an entity, each leaf has a probability $\frac{1}{b}$ to be visited and a probability $1 - \frac{1}{b}$ not to be visited. When versions of $q_v$ entities are searched, as all searches are independent, each leaf has a probability $\left(1 - \frac{1}{b}\right)^{q_v}$ not to be visited and a probability $1 - \left(1 - \frac{1}{b}\right)^{q_v}$ to be visited. Consequently,

$$P = N \left(1 - \left(1 - \frac{1}{b}\right)^{q_v} \right) = \frac{E}{b} \left(1 - \left(1 - \frac{b}{E}\right)^{q_v} \right)$$

Range queries: the average number of versions generated by $q_r$ entities is $q_r(1 + a(V - 1))$. To retrieve $q_r(1 + a(V - 1))$ entity versions, the average number of visited leaves is

$$P = \frac{q_r}{b} \left(1 + a(V - 1)\right)$$

### 4.2 OB+tree

#### 4.2.1 Estimation of $N$, average number of leaves

Estimation of $n$: as $B + (v_1)$ indexes $E$ entity versions, $n = \frac{E}{b}$.

Estimation of $m$: let $B + (v_1)$ and $B + (v_j)$ be two consecutive B+tree. $m$ is the average number of new leaves created in $B + (v_j)$. $m$ is estimated by $n \times A$, where $A$ is the probability for a leaf belonging to $B + (v_1)$ to be duplicated in $B + (v_j)$. $A$ is the leaf agility.

As the model only consider entity updates, all the $B+$trees have the same number of leaves as $B + (v_1)$. Thus, when an entity is updated in $v_j$, each leaf of $B + (v_j)$ has a probability $1 - \frac{1}{b}$ not to be affected. In $v_j$, $aE$ entity updates are thus considered. Thus, the probability for a leaf not to be affected by these updates is $(1 - \frac{1}{b})^{aE}$. Conversely, the probability to be affected by at least one update is $A = 1 - (1 - \frac{1}{b})^{aE}$. Accordingly,

$$N = \frac{E}{b} \left(1 + \left(1 - \left(1 - \frac{b}{E}\right)^{aE}\right)(V - 1)\right)$$

#### 4.2.2 Estimation of $r = \frac{\xi_E}{2} - 1$, redundancy factor

As each leaf stores $b$ entity versions on average, $\xi_E = b \times N = B = E(1 + a(V - 1)) \Rightarrow r = \frac{1 + A(V - 1)}{2 + a(V - 1)} - 1$.

#### 4.2.3 Estimation of $P$, number of visited leaves

$P$ is estimated by $p(1 + A(q_v - 1))$, where $p$ is the average number of visited leaves in a $B+$tree to retrieve $q_v$ entities. Perfect match queries: reasoning as in subsection 4.1.3 (changing $N$ to $n$ and $P$ to $p$), $p$ can be estimated by

$$P = \frac{q_v}{b} \left(1 + \left(1 - \left(1 - \frac{b}{E}\right)^{aE}\right)(q_v - 1)\right)$$

### 4.3 BT-tree

#### 4.3.1 Estimation of $N$, average number of leaves

Estimation of $n$: let $D$ be a leaf of the BT-tree alive at DBV $v$. Let $b'$ be the average number of entity versions stored in $D$ and alive at $v$. At steady state, $b'$ has been estimated by $\approx \theta B$ in 2 [3], where $B$ is the leaf node capacity. Initially the BT-tree indexes $E$ entity versions, thus $n = \frac{E}{b'}$.

Estimation of $m$: the average number of leaves alive at a DBV $v$ equals $\frac{E}{b'}$. Thus, when $aE$ entities are updated in $v$, each leaf alive at $v$ receives $\frac{aE}{b'}$ entity versions on average. When a leaf $D$ is created due to a version split at a DBV $v_{start}$, it initially contains $b'$ entity versions. Thus, another version split occurs to $D$ after the insertion of $B - b'$ entity versions. As $D$ receives on average $ab'$ entity versions from each DBV descendant of $v_{start}$, $D$ is version split after the generation of $\frac{ab'}{B-a'v}$ DBVs (starting from $v_{start}$). Thus, the generation of $\frac{ab'}{B-a'v}$ DBVs, on average, $\frac{E}{b'}$ new leaves are created in the BT-tree. Finally, the average number of leaves created when entities are updated in a DBV is

$$m = \frac{E}{b'} \Rightarrow m = \frac{aE}{B-b'}$$

As $N = n + m(V - 1)$,

$$N = \frac{E}{b'} \left(1 + \frac{ab'}{B-b'}(V - 1)\right)$$

#### 4.3.2 Estimation of $r = \frac{\xi_E}{2} - 1$, redundancy factor

When a leaf node is split, $b'$ entity versions are copied on average. The total number of leaf splits is $m(V - 1)$. Consequently,

$$\xi_E = E + \frac{m}{b}m(V - 1) \Rightarrow r = \frac{\frac{ab'}{B-a'v}(V - 1)}{\frac{ab'}{B-a'v}(V - 1)}$$

#### 4.3.3 Estimation of $P$, number of visited leaves

The estimation of $P$ for the BT-tree follows the same principle as for the OB+tree: $P = p(1 + A(q_v - 1))$, where $p$ is...
the average number of leaves visited to retrieve \( q_e \) entity versions of a DBV \( v_i \) and \( A \) the probability for a leaf alive at \( v_i \) to be version split at a DBV \( v_j \), child of \( v_i \). \( A \) is estimated by
\[
\frac{m}{n} = A = \frac{ab'}{B-v}.
\]

**Perfect match queries:** reasoning as in subsection 4.1.3 (changing \( N \) to \( \frac{1}{E} \) and \( P \) to \( p \) ), \( p \) can be estimated by
\[
P = \frac{E}{v} \left(1 - \left(1 - \frac{1}{E} \right)^{q_e} \right) \left(1 + \frac{ab'}{B-v} (q_e - 1) \right)
\]

**Range queries:** \( p = \frac{q_e}{v} \) \( \Rightarrow \)
\[
P = \frac{q_e}{v} \left(1 + \frac{ab'}{B-v} (q_e - 1) \right)
\]

5. SIMULATION

A large number of experiments have been performed to show the accuracy of the steady state analysis (see [4]). Due to the lack of space, only few results are presented herein. For the presented simulation, data are generated as follows.

- \( E \) and \( V \) are fixed respectively to 200K entities and 200 DBVs.
- \( E \) entities are inserted in the first DBV. Then, in each newly derived DBV, \( aE \) entities randomly selected, are updated. Each DBV is derived from a randomly selected DBV, with the following restrictions: (i) entity updates are allowed only in the DBVs leaves of the DBV tree; (ii) the distance between the DBV root and any DBV leaf of the final DBV tree is \( \geq 40 \) DBVs. The data page capacity is set to \( B' = 101 \) entity versions for the \( OB+tree \) and \( B = 92 \) for the \( B+tree \) and the BT-tree. As shown by figures 6.a, 6.b and 7, our model accurately estimates the query and the storage costs of the different index structures.

**Query cost:** the model shows that, in general, the \( B+V \)-tree is best suited for perfect match queries (figure 6.a).

Thus, if entity identifiers are system generated (without semantics), the \( B+V \)-tree appears as the best choice (because range queries are of little use). If the entity identifier is an attribute and range queries are frequent, the BT-tree is the best choice (figure 6.b). The OB+tree can be chosen only when version slice queries are frequent (figures 6.a and 6.b).

6. CONCLUSION

This paper studies the problem of efficiently indexing data with branched evolution. Its main contributions are: (i) the extension of temporal index structures to data with branched evolution; (ii) a comparative analysis of main trends in indexing multiversion data; (iii) a steady state analysis which accurately estimates the performance of the different index structures and provides guidelines for the selection of the most appropriate one; (iv) an experimental validation of these results.

7. ACKNOWLEDGMENTS

Warm thanks to Claudia Bauzer Medeiros for the helpful discussions and reviews that improved this paper.

8. REFERENCES