Temporal databases

Rasmus Pagh

Reading: [Arge01, sec. 1+ “persistent B-trees”/sec. 2.1] Slides on persistent B-trees by Lars Arge.
Course evaluation

• It is my plan to do a substantial revision of DBT next year.
• Please use the course evaluation to give your input to this process:
  – What parts of the lectures/project worked well (should be kept)?
  – What parts worked less well?
  – What did you miss?
• Your opinion is appreciated!
  Deadline Friday April 17.
Today

• Temporal data, what and how?
• Indexing temporal data.

• Independent part (afternoon): Guest lecture by Philippe Bonnet on flash-based storage technology.
What is a temporal database?

• Database with a notion of “time”.
• Several possible notions:
  – Valid time
  – Transaction time
• Typically, “time” is used in a special way in queries:
  – Example: How many employees did we have on April 1, 2008?
• Today, we focus on transaction time.
  – Essentially want to be able to access all previous versions of the database.
Timestamping tuples

• **Simple idea:** Extend each relation schema with two attributes that encode a time interval:
  - Tst (start time/insertion time)
  - Tet (end time/deletion time). Tet of current tuples have special value \( \texttt{uc} \) (think \( \infty \)).

• A query “for time t” should include the extra condition \( \text{Tst} \leq t \text{ AND } t \leq \text{Tet} \) on each relation.

• Important that primary keys do not change – want to be able to relate entities over time.
TSQL2 temporal extensions

- Gives more convenient ways of expressing temporal conditions, e.g. join conditions as “the tuples existed at the same time”. (SQL alternative?)
- Gives operations on time intervals (union, intersection,...).
- Ways of “aggregating” time intervals, e.g., finding time intervals not covered by a set of intervals.
- Today, we do not go further into the language aspects of temporal DBs.
Maintaining time stamps

- New tuples are inserted with current time (transaction time) as Tst.
- Deletions are not performed – instead Tet is set to the current time.
- Changes to tuples are conceptually done by deleting the old version and inserting the new one.
  - Can be wasteful in terms of space. A possibility is to split each relation into many relations with one attribute each in addition to the primary key ("temporal normal form").
• Consider how B-tree indexes might be used to select tuples that satisfy $T_{st} \leq t$ AND $t \leq T_{et}$.

• Argue that in general, B-trees will not allow us to find the matching tuples efficiently.
Next: Persistent B-trees

• Multiversion B-trees (aka. partially persistent B-trees) is an efficient index for temporal data.

• Assumption: “Transaction time” is used, i.e., timestamps may only be set to the current time.

• Warm-up: Persistent linked lists. (Board.)
Persistent B-tree

• Easy way to make a B-tree persistent
  – Copy structure at each operation
  – Maintain “version-access” structure (B-tree)

• $O(\log_B N + T/B)$ I/O query, any version 😊
  – $O(N/B)$ I/O update time 😞
  – $O(N^2/B)$ space 😞
Persistent B-trees, better way

- **Next idea**: Instead of copying the whole tree for each update, copy just the nodes that are “affected”, and re-use the rest.

- **Affected nodes**:
  - Updated nodes.
  - Nodes on the path to an updated node (specifically, we get a new root at each time instance).

- **Now update time is** $O(\log_B N)$ 😊
- **Space is** $O(N \log_B N)$ blocks 😞
Persistent B-tree

- **Idea:** Elements (in internal and leaf nodes) are augmented with “existence interval” and stored in one structure

- Persistent B-tree with parameter $B$:
  - Directed acyclic graph
    * Nodes contain elements augmented with existence interval
    * At any time $t$, nodes with elements live at time $t$ form B-tree with leaf and branching parameter $B$ (i.e., each node/leaf has at least $B/4$ and at most $B$ children/keys in them)
  - B-tree with leaf and branching parameter $b$ on “root nodes”.

Query at any time $t$ in $O(\log_B N + T_B)$ I/Os
Persistent B-tree: Updates

• Updates performed essentially as in a B-tree

• To obtain linear space we maintain new-node invariant:
  – New node contains between $\frac{3}{8}B$ and $\frac{7}{8}B$ live elements and no dead elements
  – Intuition: Ensure that many update operations take place before the node is replaced.
Persistent B-tree Insert

• Search for relevant leaf $u$ and insert new element
• If $u$ contains $B+1$ elements: Block overflow
  – Version split:
    Mark $u$ dead and create new node $u'$ with $x$ live elements
  – If $x > \frac{7}{8}B$: Strong overflow
  – If $x < \frac{3}{8}B$: Strong underflow
  – If $\frac{3}{8}B \leq x \leq \frac{7}{8}B$ then recursively update $parent(u)$:
    Delete (persistently) reference to $u$ and insert reference to $u'$
**Persistent B-tree Insert**

- **Strong overflow** \((x > \frac{7}{8} B)\)
  - Split \(u\) into \(u'\) and \(u''\) with \(\frac{x}{2}\) elements each \((\frac{3}{8} B < \frac{x}{2} \leq \frac{1}{2} B)\)
  - Recursively update \(\text{parent}(u)\):
    - Delete reference to \(u\) and insert reference to \(v'\) and \(v''\)

- **Strong underflow** \((x < \frac{3}{8} B)\)
  - Merge \(x\) elements with \(y\) live elements obtained by version split on sibling \((\frac{1}{2} B \leq x + y \leq \frac{11}{8} B)\)
  - If \(x + y \geq \frac{7}{8} B\) then (strong overflow) perform split into nodes with \((x+y)/2\) elements each \((\frac{7}{16} B \leq (x + y) / 2 \geq \frac{11}{16} B)\)
  - Recursively update \(\text{parent}(u)\): Delete two insert one/two references
Persistent B-tree Delete

• Search for relevant leaf $u$ and mark element dead
• If $u$ contains $x < \frac{1}{4}B$ live elements: Block underflow
  – Version split:
    Mark $u$ dead and create new node $u'$ with $x$ live elements
  – Strong underflow ($x < \frac{3}{8}B$):
    Merge (version split) and possibly split (strong overflow)
  – Recursively update $\text{parent}(u)$:
    Delete two references insert one or two references
Persistent B-tree

Insert
  \[\text{Block overflow}\]
  \[\text{Version split}\]
  \[\text{Strong overflow}\]
  \[\text{Split}\]
  \[\text{done} -1,+2\]

Delete
  \[\text{Block underflow}\]
  \[\text{Version split}\]
  \[\text{Strong underflow}\]
  \[\text{Merge}\]
  \[\text{Strong overflow}\]
  \[\text{done} -2,+1\]

\[\text{-1,+1}\]
\[\text{-2,+2}\]
\[\text{0,0}\]
Persistent B-tree Analysis

• **Update:** $O(\log_B N)$
  – Search and “rebalance” on one root-leaf path

• **Space:** $O(N/B)$
  – At least $\frac{1}{8} B$ updates in leaf in *existence interval*
  – When leaf $u$ dies
    * At most two other nodes are created
    * At most one block over/underflow one level up (in $parent(u)$)

  \[\downarrow\]
  – During $N$ updates we create:
    * $O(N/B)$ leaves
    * $O(N/B^i)$ nodes $i$ levels up
  \[\Rightarrow \sum_i O(N/B^i) = O(N/B)\text{ blocks}\]
Summary/Conclusion: Persistent B-tree

• Persistent B-tree
  – Update current version
  – Query all versions

• Efficient implementation obtained using existence intervals
  – Standard technique

↓

• During $N$ operations
  – $O(N/B)$ space
  – $O(\log_B N)$ update
  – $O(\log_B N + T/B)$ query
Valid time

• Persistent B-trees critically use that timestamps can only be set to “now”.
• To index *valid time*, we may use a solution to the “interval management” problem:
  – Index $N$ intervals such that a stabbing query at time $x$ and updating the set of intervals is efficient.

• Theoretically optimal solution: [Arge01, sec. 4]
  – Note: Cannot search in stabbed intervals.
Bi-temporal databases

- The two notions of time co-exist.
- Possible to make queries that involve both time dimensions.
- A possible indexing approach is to use multi-dimensional indexes such as R-trees.
Exercise

- Suppose we have access to persistent B-trees and standard B-trees.
- Consider how to make efficient indexes for the following queries: Report the tuples that:
  a) were inserted some time after time $t$.
  b) existed at time $t$.
  c) existed at some point in $[t_1; t_2]$.
  d) existed in the whole time interval $[t_1; t_2]$.
- Extra: Consider the effect of an additional range condition, e.g. $a > 10$. 