Efficient Sequenced Temporal Integrity Checking

Wei Li* Richard T. Snodgrass† Shiyan Deng† Vineel K. Gattu‡
Aravindan Kasthurirangan†

*Oracle Corporation
Redwood Shores, CA
weili@us.oracle.com

†Department of Computer Science
University of Arizona
{rts, shiyand, arvindk}@cs.arizona.edu

‡Microsoft Corporation
Redmond, WA

Abstract

Primary key and referential integrity are the most widely used integrity constraints in relational databases. Each has a sequenced analogue in temporal databases, in which the constraint must apply independently at every point in time. In this paper, we assume a stratum approach, which expresses the checking in conventional SQL, as triggers on period-stamped relations. We evaluate several novel approaches that exploit B^+-tree indexes to enable efficient checking of sequenced primary key (SPK) and referential integrity (SRI) constraints. We start out with a brute force SPK algorithm, then adapt the Relational Interval-tree overlap algorithm. After that, we propose a new method, the Straight Traversal algorithm, which utilizes the B^+-tree more directly to identify when multiple key values are present. Our evaluation, on two platforms, shows that Straight Traversal algorithm approaches the performance of built-in nontemporal primary key and referential integrity checking, with constant time per tuple.

1 Introduction

Primary key and referential integrity are the most widely used integrity constraints in relational databases. SQL for years has included syntax to easily specify the attributes that serve as a primary key for the relation, as well as the attributes that refer to other relations.

A temporal database captures time-varying information [Jensen99]. For the purposes of this paper, we assume that each relation is a period-stamped *valid-time state relation* [Jensen98], recording when a fact held in the modeled reality. The algorithms we consider apply equally well to transaction-time relations, recording when a fact was present in the database.

Each non-temporal integrity constraint has a *sequenced* analogue over a temporal relation, in which the constraint must apply independently at every point in time [Snodgrass99]. For a sequenced primary key, at each point in time there are no two tuples that have the same snapshot primary key. Note that there may be two tuples having the same value-equivalent snapshot primary key, but the two tuples cannot be associated with an overlapped validity period.

Sequenced referential integrity has a similar meaning: for every point in the validity period of a tuple in the referencing relation, there is always a corresponding tuple in the referenced relation with the required foreign key value.

There are two basic ways to implement a sequenced integrity constraint, or in general any temporal functionality: either modify the underlying DBMS, or implement that functionality on top of a conventional DBMS, in a *stratum* that translates a temporal expression into conventional SQL [Torp98]. In this paper, we focus on the stratum approach, though our method can also be embedded within a DBMS. We define sequenced integrity checking as conventional SQL triggers; these triggers will make the appropriate checks during any database modification (insertion, deletion, or update).

Conventional primary key and referential constraint checking rely on indexes to locate offending duplicates and missing referents. As we will see, while adding indexes definitely improves the performance of a brute force sequenced trigger, the performance is still woefully inadequate. The underlying problem is that there is no total ordering on periods, so a point-based index such as a B^+-tree is less effective here. That is the reason that so many temporal indexes [Salzberg99] have been proposed. However, given that we are using a stratum, we cannot blithely define a temporal index, because we are viewing the underlying DBMS as a black box that implements SQL. Rather, we have to emit triggers expressed in standard SQL that indirectly utilize the various services of the DBMS, including conventional B^+-tree indexes.

1063-6382/01 $10.00 © 2001 IEEE

The rest of this paper investigates several novel algorithms that exploit B^+-tree indexes on auxiliary relations. For sequenced primary key *(SPK)*, we adapt the *Relational-Interval Tree Overlap algorithm* [Kriegel00], which uses a B^+-tree to effectively encode an interval tree, which it traverses level-by-level in SQL. We then propose a new method, the *Straight Traversal Algorithm*, which utilizes the B^+-tree more directly to identify when multiple key values are present.

For sequenced referential integrity *(SRI)*, the Relational-Interval Tree algorithm doesn't apply, because the relevant predicate is *contains* rather than *overlap* (the period timestamps of the referenced relation for the snapshot key must contain the period of the referencing tuple). We introduce the *Meets Algorithm*, which explicitly identifies gaps in the referenced relation. We compare this algorithm again to a brute force SRI algorithm and to the Straight Traversal algorithm applied to SRI.

2 Previous Work

Previous work in temporal database integrity constraints has focused on two basic problems: how to express complex temporal integrity constraints [Ehrich84, Gal95, Sistla95], and how to implement such constraints with a minimum of stored state [Chomicki95, Lipeck87, Plexousakis93]. (A nice summary of previous work may be found in Wes Cowley's MS thesis [Cowley99].) As such, this previous work is peripheral to the problem we address. Stating SPK and SRI constraints is a straightforward extension of conventional primary key and referential integrity constraints in SQL. And if we assume a temporal database to start with, the history is already available as time-stamped tuples in temporal relations.

Previous research in the area of query evaluation in temporal databases has focused on novel algorithms, e.g., for temporal joins and temporal aggregates, and on novel storage structures, e.g., temporal indexes for valid-time databases or transaction-time databases [Salzberg99]. Such approaches are not relevant when a stratum architecture is imposed.

Recently there has been interest in the stratum approach, which requires no changes to the underlying DBMS. Nascimento and Dunham have proposed a temporal index that uses B^+-trees [Nascimento99]. More recently, Kriegel, Pötke and Seidl proposed a Relational-Interval Tree [Kriegel00], which also exploits conventional B^+-trees. As the latter is specifically designed to support interval intersection, which is the fundamental operation underlying sequenced primary key checking, we will extend that algorithm, in Section 4.

3 Sequenced Integrity Constraint Checking in SQL

Let us start with a conventional relation INCUMBENTS which records which employee holds which job position in the company.

```
INCUMBENTS(SSN, PCN) PRIMARY KEY(SSN)
  FOREIGN KEY(PCN) REFERENCES POSITIONS
```

The primary key constraint says that each employee can have only one position. The foreign key constraint says that the value of the PCN (position control number) will always be found in the POSITIONS relation.

3.1 Sequenced Primary Keys

If we want to keep the history for this relation, then we need to make INCUMBENTS a valid time relation by changing the schema to:

```
INCUMBENTS(SSN,PCN,START_DATE,END_DATE)
```

Alternatively, if we were using a temporal extension of SQL, such as TSQL2 [Snodgrass95], this conversion could be stated as something like

```
ALTER TABLE INCUMBENTS
            ADD VALIDTIME(DAY)
```

When history is added, there may well be several tuples with the same SSN, as illustrated below. (We assume a closed-closed representation for period timestamps, in that the START_DATE and END_DATE days are contained in the period.)

SSN	*PCN*	*START_DATE*	*END_DATE*
111223333	900225	1999-01-01	1999-05-30
111223333	120033	1999-06-01	1999-09-30

The sequenced analog of the non-temporal primary key constraint requires that no two tuples have the same value for the SSN attribute at any point in time. Unfortunately, none of the following approaches correctly specify a sequenced primary key of SSN.

```
ALTER TABLE INCUMBENTS ADD
  PRIMARY KEY (SSN)
ALTER TABLE INCUMBENTS ADD
  PRIMARY KEY (SSN, START_DATE)
ALTER TABLE INCUMBENTS ADD
  PRIMARY KEY (SSN, END_DATE)
ALTER TABLE INCUMBENTS ADD
  PRIMARY KEY (SSN,START_DATE,END_DATE)
```

None of these constraints prevent the following erroneous tuple from being inserted into the INCUMBENTS relation (the resulting relation would have two rows with the same SSN in May and June).

SSN	*PCN*	*START_DATE*	*END_DATE*
111223333	328922	1999-05-01	1999-06-30

The challenge before us is to express the sequenced constraint efficiently in SQL, as an assertion or trigger mentioning the timestamp attributes START_DATE and END_DATE. We can use the following brute-force assertion [Snodgrass99] to check the sequenced primary key constraint on the INCUMBENTS relation.

```
CREATE ASSERTION seq_primary_key
CHECK (NOT EXISTS
(SELECT * FROM INCUMBENTS AS I1 WHERE
   1<(SELECT COUNT(SSN)
      FROM INCUMBENTS AS I2
      WHERE I1.SSN=I2.SSN
        AND I1.START_DATE<=I2.END_DATE
        AND I2.START_DATE<=I1.END_DATE))
```

Unlike the prior attempts, this assertion is correct; it will catch any violations, where two tuples with the same SSN overlap in time. However, this assertion is very slow, for several reasons. It utilizes an aggregate which must be evaluated for each tuple in INCUMBENTS. The assertion will probably read all the tuples in INCUMBENTS(I2) to check whether their snapshot primary keys are the same and whether their validity periods intersect. Due to the inequality predicate, this will probably be evaluated as a nested-loop self-join, which is of time complexity $O(N^2)$ (where N is the cardinality of the INCUMBENTS relation). Additionally, the assertion does not exploit the fact that that before each change, the temporal relation satisfied the assertion. The only violation possible after the operation is the conflict between the new tuple with the original tuples in the relation. Taking this fact into consideration, we can convert the assertion into a trigger that only checks the new tuple(s), possibly yielding linear-time performance. In comparison, conventional primary key checking can be done in constant time by using B^+-tree index (assuming a fixed number of levels). Now the question becomes: can we use a B^+-tree in such a way to devise a sequenced primary key trigger that always runs in constant time per tuple modification?

3.2 Sequenced Referential Integrity

Recall that the INCUMBENTS relation referenced the POSITIONS relation. We now render that latter relation temporal to investigate sequenced referential integrity.

```
POSITIONS(PCN, JOB_TITLE,
          START_DATE, END_DATE)
```

A referential integrity constraint specifies that the value of specified attribute in every tuple of the referencing relation appears as the value of a specified attribute of a tuple of the referenced relation. Sequenced referential integrity requires that at each time point for referencing tuple, there should be corresponding tuple(s) in referenced relation at that time. Effectively, the validity period of the referencing tuple must be *contained* in the combined validity periods of the referenced tuples with the appropriate attribute values.

The key is a *sequenced* foreign key if, for all tuples r in the referencing relation [Snodgrass99],

- there is a tuple with that key value valid in the referenced relation when r started,
- there is a tuple with that key value valid in the referenced relation when r stopped,
- and there are no gaps when there are no tuples in the referenced relation, during r's period of validity, that have that key value.

This brute-force approach (Figure 1) is quite complex. The performance of this assertion is also poor: it involves a self-join (in this case, several, for the nested sub-queries) as well as a whole relation search.

```
CREATE ASSERTION INCUMBENTS_SRI CHECK(
NOT EXISTS(
  SELECT * FROM INCUMBENTS AS I
  WHERE NOT EXISTS(
    SELECT * FROM POSITIONS AS P
    WHERE I.PCN = P.PCN
      AND P.START_DATE<=I.START_DATE
      AND I.START_DATE <= P.END_DATE)
OR NOT EXISTS(
  SELECT * FROM POSITIONS AS P
  WHERE I.PCN = P.PCN
    AND P.START_DATE<=I.END_DATE
    AND I.END_DATE<=P.END_DATE)
OR EXISTS(
   SELECT * FROM POSITIONS AS P
   WHERE I.PCN = P.PCN
     AND I.START_DATE<=P.END_DATE
     AND P.END_DATE<I.END_DATE
     AND NOT EXISTS(
       SELECT * FROM POSITIONS AS P2
       WHERE P2.PCN = P.PCN
       AND P2.START_DATE+1<=P.END_DATE
       AND P.END_DATE < P2.END_DATE))))
```

Figure 1. Brute-force SRI assertion

We first propose more efficient approaches for SPK, then turn to SRI.

4 The Relational-Interval Tree Approach

In a recent paper, Kriegel, Pötke and Seidl [Kriegel00] efficiently implement Edelsbrunner's interval tree [Edelsbrunner80] on top of a relational database system, by utilizing an auxiliary relation with two associated B^+-tree indexes. This Relational-Interval Tree approach yields very fast intersection queries, expressed as single SQL statements on this auxiliary relation; these statements indirectly utilize the indexes.

We extend the intersection query to check for SPK violations: there should be no intersections between the new interval and existing intervals with the same key attribute values [Li01].

The Relational-Interval Tree addresses a more general problem, interval intersection, which raises the possibility that an algorithm customized to SPK may be simpler and more efficient. The next section will present such an approach.

5 The Straight Traversal Approach

The nice thing about the conventional primary key checking with B^+-index is that the index search can be done in constant time, assuming a fixed tree height. When we use a B^+-tree to effect a primary key check, we use only equality predicates, which are well suited for point-based data. For a sequenced PK, the timestamps appear in inequality predicates, which is the source of inefficiency.

The SPK constraint states the following property: for all the tuples with the same snapshot primary key, there are no valid time periods that intersect. In Figure 2, we assume that all the tuples inserted have the same snapshot primary key A. When tuple A^1, A^2 and A^3 are inserted, we see that all the valid time periods hold the SPK constraint. However, when tuple A^4 is inserted, we can see there would be two intersections that violate the constraint. Therefore, A^4 cannot be inserted into the relation.

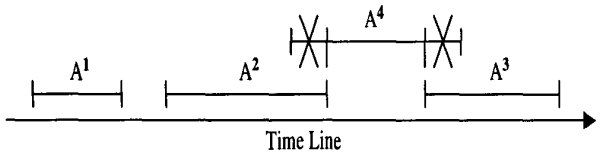


Figure 2. The example of SPK constraint for valid time periods

5.1 Sequenced Primary Key Checking

Consider an auxiliary relation with the schema of the snapshot primary key and a single date value PD, containing two tuples for every tuple in the original relation, recording the validity period's start date and end date.

If we sorted the auxiliary relation on the composite key of snapshot primary key and date value, we would find that any two tuples originating from a valid time tuple will always be consecutive, if the original temporal relation satisfies the SPK constraint(cf., Figure 2). Instead of sorting the auxiliary relation, we declare a B^+-tree index, with an index key of the snapshot key coupled with the date attribute.

There are four cases, shown in Figure 3, that may cause SPK violations. In the figure, *new* denotes the tuple that is being inserted or updated, while *old* denotes the original tuple in the valid time relation.

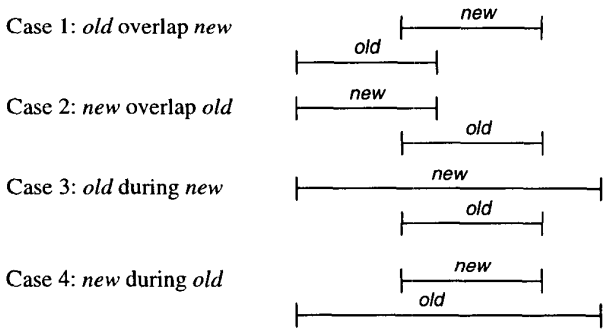


Figure 3. SPK Violation Cases

As a new tuple is being inserted into the original valid-time state relation, we first consider the initial three cases in Figure 3. The inserted *new* tuple must render the relation violate the SPK constraint. Because our B^+ has consecutive pairs of start and end times, it is not difficult to find the violation point with the help of this index. We only need search the tree to find whether there exists index entries with the same snapshot primary key and whose time instant (either the start or end date) is between the *new* tuple's start and end date. If such an index entry exists, it means that there exists a tuple in the valid time relation which would overlap the *new* tuple, thus violating the SPK constraint.

Case 4 in Figure 3 is a little trickier to detect, especially given that we must evaluate the check in constant time. On first thought, we should find a pair of indexes that index to the same tuple. Then we need to check whether the start or end time of the *new* tuple is during the valid time period of the indexed tuple. We can take advantage of two facts to make this check simple:

- In the B^+-tree index, all the two index entries corre-

sponding to a tuple in the original relation must be contiguous in the index: there is no *overlap* or *during* relationships between tuples.

- Before we execute the check of the fourth case, we have finished the check for cases 1, 2 and 3. So we definitely know that cases 1, 2, and 3 have not occurred.

The query can be executed by first searching the B^+-tree to find an index entry that is less than the composite key value of (*new*'s key, *new*'s start date) and is the last one (in time) of all the qualifying index entries. Second, check whether the date value in the found entry is from the start date or the end date (this is indicated by an additional bit in the index entry). If the entry originates from the start date, then the corresponding index entry containing the end date must be greater than the composite key (*new*'s key, *new*'s end date). Consequently this is case 4, since the previous query ensured that case 1 did not occur. Otherwise, if the found entry contains the end date, then we know that the *old* tuple which may cause the during predicate has finished before the *new*'s start date, indicating that the SPK constraint will not be violated.

The query to check the sequenced primary key for insertion is given in Figure 4. It is not difficult to extend the query to deal with update (deletion is not an issue, because it cannot violate a primary key, or SPK, constraint). We term this the *Straight Traversal Approach*, in that it differs from the Relational-Interval Tree approach in that does not simulate a level-by-level traversal of an interval tree, but rather traverses the B^+-tree directly.

Assertion : sequenced primary key checking for insertion
Input *new*(the newly inserted tuples)
Var *e*(index entry)
set $e \leftarrow$ the index entry where $e.pdate >= new.start\_date$
and $e.key = new.key$ **and not exists** f
($f.pdate > new.start\_date$ **and**
$f.key = new.key$ **and** $e.pdate > f.pdate$)
if ($e \neq \perp$) **and** $e.pdate <= new.end\_date$ **then**
assertion failed
endif
set $e \leftarrow$ the index entry where $e.pdate < new.start\_date$
and $e.key = new.key$ **and not exists** f
($f.pdate < new.start\_date$ **and**
$f.key = new.key$ **and** $e.pdate < f.pdate$)
if ($e \neq \perp$) **and** $e.dtype = 0$ **then**
assertion failed
endif
End Assertion

Figure 4. Straight Traversal for SPK Checking

5.2 Sequenced Referential Integrity

Now let us extend this algorithm to perform sequenced referential integrity checking. When we do SRI checking, we need search all the matched tuples in the referenced relation. Given that the referenced relation will satisfy the SPK constraint(we assume that the referenced attributes constitute a sequenced primary key), we know that the several tuples combining into one long-period tuple will be contiguous in the B^+-tree leaf index entries. Accordingly, for one referencing tuple, once the starting referenced index entry is located, all future reading from the index entry will be mostly sequential reads, except for moving to the next index block, which is infrequent. The assertion to check SRI on insertion is listed in Figure 5. This algorithm it works in two steps for one referencing tuple:

1. Find the maximum index entry that is less than the referencing tuple's starting date;
2. Iterate until an index entry matches the referencing tuple's ending date and check for gaps between the two consecutive index entries.

5.3 Implementation on Oracle8i

We elaborate the implementation of the Straight Traversal algorithms on top of the Oracle8i DBMS. No change the underlying DBMS code is needed. Instead, we use Oracle triggers to maintain the structure of auxiliary relation. We assume here the schema of the `INCUMBENTS` and `POSITIONS` relations given earlier. The particular triggers are given elsewhere [Li01].

We create an auxiliary relation containing the snapshot primary key and the `PD` attribute. Every update performed on the original valid time relation invokes a trigger that makes the corresponding changes to the auxiliary relation (e.g., for an insertion, insert two tuples into the auxiliary relation). Then integrity constraints will be checked on the auxiliary relation.

For the `INCUMBENTS` relation, an auxiliary relation `INCUMBENTS_MIRROR` is created.

```
CREATE TABLE INCUMBENTS_MIRROR (
  SSN INT NOT NULL,
  PD  DATE NOT NULL,
  DTYPE NUMBER(1) ) ;
CREATE INDEX MIRROR_IDX ON
  INCUMBENTS_MIRROR ( SSN, PD ) ;
```

In Oracle we can use a so-called *index-organized table* to merge the auxiliary relation and its index: the table will contain both the encoded key value and the associated attribute values for the corresponding table, instead

Assertion : sequenced referential integrity
Input new(the newly inserted tuples)
Var e(index entry)
Var e_first(index entry)
Var e_second(index entry)
set $e \leftarrow$ the index entry where $e.pdate <= new.start\_date$
and $e.key = new.f\_key$ **and not exists** f
($f.pdate <= new.start\_date$ **and**
$f.key = new.f\_key$ **and** $e.pdate < f.pdate$)
if ($e \equiv \perp$) **or** $e.dtype = 1$ **then**
assertion failed
endif
set $e\_first \leftarrow e$
set $e\_second \leftarrow$ the next index entry after e
while (checking is not finished)
if ($e\_second.pdate < new.end\_date$) **then**
set $e \leftarrow e\_second$
set $e\_first \leftarrow$ the first next index entry
set $e\_second \leftarrow$ the second next index entry
if ($e.pdate$ **not meet** $e\_first.pdate$)
assertion failed
endif
else
return successful
endif
end while
End Assertion

Figure 5. Straight Traversal Algorithm for SRI Checking

of ROWID which would be used to retrieve the corresponding tuple in the regular B^+-tree index. One problem is that the index-organized table requires a primary key for the creation of a unique B^+-index. With our time period representation, it is possible that the start time is the same as the end time (if the period of validity is one day long). Consequently, we cannot create a unique index on the attributes (SSN, PD). For this reason, we include the DTYPE attribute. The schema definition for the index-organized table definition is as follows.

```
CREATE TABLE INCUMBENTS_MIRROR (
  SSN INT NOT NULL,
  PD  DATE NOT NULL,
  DTYPE NUMBER(1),
  PRIMARY KEY ( SSN, PD, DTYPE )
) ORGANIZATION INDEX ;
```

For each tuple in the valid time relation, there will be two tuples automatically inserted into in the auxiliary relation by the insertion trigger.

Now, we need to express the two queries in Figure 4 using SQL commands. The following query can detect the first three cases in Figure 3. The query looks simple for we only use SQL to express the condition. The optimizer will choose the access path similar to the one in the Figure 4. (This code is part of the insert trigger.)

```
SELECT I.SSN FROM INCUMBENTS_MIRROR I
WHERE :NEW.SSN=I.SSN
  AND I.PD>=:NEW.START_DATE
  AND I.PD<=:NEW.END_DATE ;
```

If the above query succeeds, it means at least there exists such a tuple whose time instant is between the new's valid time period. If such a new tuple were inserted, it would signal an SPK violation.

The following SQL query deals with the fourth case in Figure 3. The query tries to find whether the greatest tuple that is less than new's start date is of type 0 or type 1. Type 0 means the tuple contains the start date. So if this query returns a tuple, there has been an SPK violation.

```
SELECT I.SSN FROM INCUMBENTS_MIRROR I
WHERE :NEW.SSN=I.SSN
  AND I.PD<:NEW.START_DATE AND I.DTYPE=0
  AND NOT EXISTS
   (SELECT * FROM INCUMBENTS_MIRROR J
    WHERE I.SSN=J.SSN
      AND J.PD<:NEW.START_DATE
      AND J.PD>I.PD) ;
```

6 The Meets Approach

The fundamental problem with SRI checking is ensuring that the collected validity periods of the referenced relation associated with the snapshot primary key value contain the validity period of the inserted tuple of the referencing relation. This involves examining potentially many tuples in the referenced relation looking for gaps. The *Meets Approach* explicitly indicates the gaps via an auxiliary relation that contains the sequenced primary key, the START_DATE and END_DATE attributes (if a ROWID attribute was present in the referenced relation, it could substitute for all these attributes), and a DOES_MEET attribute, which is 0 if another tuple meets (i.e., starts immediately after the current tuple ends), and 1 if there exists a gap immediately following this tuple (we assume that the foreign key is the primary key of the referenced relation).

This auxiliary relation simplifies the third clause, the EXISTS clause to the following.

```
OR EXISTS
(SELECT * FROM POSITIONS_MIRROR AS PM
 WHERE I.PCN = PM.PCN
```

```
AND I.START_DATE<=PM.END_DATE
AND PM.END_DATE<I.END_DATE
AND PM.DOES_MEET=0)
```

A B^+-tree index on SSN and END_DATE works best.

This simplification, which speeds up the assertion, also has a significant cost, in that the auxiliary relation, in particular the DOES_MEET attribute, has to be maintained for the referenced relation.

7 Evaluation

Our experiments examine sequenced primary key and sequenced referential integrity in turn. The experiments were conducted on Oracle 8i system on a Pentium II 266MHz/512KB cache with 128MB of main memory and 4GB hard disks. To demonstrate that this approach is also feasible in other systems, we also tested the performance on Microsoft SQL Server 7.0 on this machine.

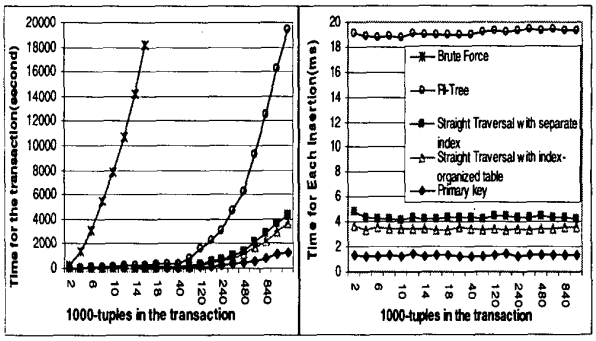
7.1 Sequenced Primary Key Checking

We compared the brute-force approach, the Relational-Interval Tree approach (the specific version used in the performance study reported in [Kriegel00]), and the Straight Traversal approach to SPK (both with a separate index and using an index-organized table). The tests consist of transactions that, starting with an empty relation, insert from 2K to 1M correct tuples into the valid time relation, with each tuple containing 116 bytes. We configured the inserted tuples so that both checks are required for the Straight Traversal approach.

We also measured the performance of maintaining one conventional primary key on the snapshot primary key and the start date. As emphasized in Section 3.1, this constraint is inadequate; we include this only to compare the performance of the (easier) snapshot primary key implemented within the DBMS with the (more difficult) sequenced primary key implemented outside the DBMS in the stratum via a trigger.

We started by inserting data in order by snapshot primary key, which will always add entries to the end of the index. In Figure 6(a), the three SPK and internal conventional primary key algorithms are compared. Note that the x-axis is exponential, while the y-axis is linear. The brute force algorithm is not competitive, and so is not considered further. Although it took a long time for the transaction to finish, the speed is only around three times slower than the transaction with built-in primary keys.

As Figure 6(b) shows, the time for each insertion is constant for all methods, with the Straight Traversal SPK algorithm being 2.7–3.3 times slower than the conventional primary key internal algorithm, and the Relational-Interval Tree algorithm being about 14.6 times slower. We also tried these same tests on Microsoft SQL Server 7, and found that the Straight Traversal SPK algorithm was 16.7 times slower than the conventional primary key. The results were quite similar to that for Oracle 8i. However, we can not report those measurements because the vendors do not allow direct comparisons to be published.(the Relational-Interval Tree algorithm requires object-relational support, and thus is not applicable to SQL Server 7).



a. b.

Figure 6. The performance of SPK checking on Oracle 8i

The next two tests use more realistic data distributions, specifically, randomized data sets and data sets with gaps. In this way, the index entry's insertion into the B^+-tree will be unordered and the expansion of B^+-tree will be more realistic. The gaps were located after every sequence of five consecutive periods. Figure 7 compares the performance with a randomized data set; Figure 8 compares the performance with gaps (again, on a randomized data set). The left side of each figure includes the Relational-Interval Tree algorithm; the right side focuses on the Straight Traversal algorithm; note the smaller range of times on the y-axis. In both tests, the Straight Traversal SPK approach remains constant time across a wide range of relation sizes, while the RI-tree algorithm does not scale well.

The tests to this point all involve insertions. We now consider updates, which are each logically a deletion followed by an insertion. We first insert a number of tuples in the base relation (from 50K tuples to 600K tuples). After that, we randomly choose 10K tuples for update. Figure 9 gives the time for each update of this experiment. From the comparison to the update with traditional primary key, we can see the Straight Traversal SPK algorithm scales very well. In fact, for large relations, using an index-organized auxiliary table is only 18% slower than the internal conventional primary key checking.

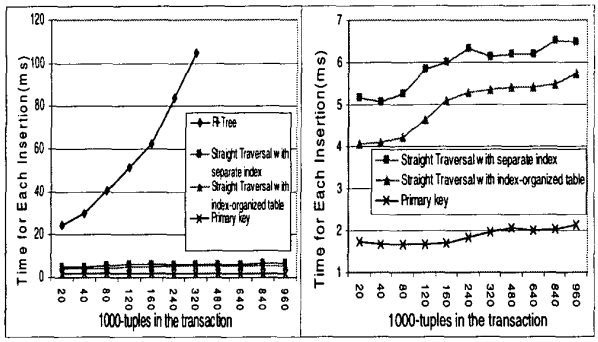


Figure 7. SPK checking randomized data sets on Oracle 8i

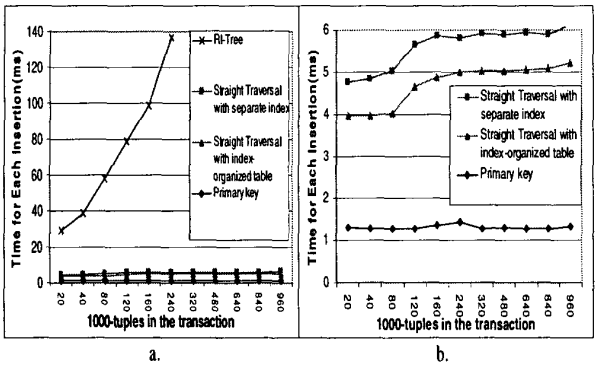


Figure 8. SPK checking randomized data sets with gaps on Oracle 8i

7.2 Sequenced Referential Integrity Checking

For sequenced referential integrity, three algorithms are relevant: the Brute Force SRI algorithm, the Meets algorithm, and the Straight Traversal SRI algorithm. (Recall that since the Relational-Interval Tree approach is built upon an interval intersection algorithm, it is not appropriate for the contains test in SRI.)

In referential integrity checking, we have two relations: the referencing relation and the referenced relation. Because referential integrity checking is done on the referenced relation to see whether there exists corresponding tuple(s) over the entire validity period of the tuple inserted into the referencing relation, the performance of that insertion will depend on how many tuples in the referenced relation need to be checked. The time spent in referential integrity check for a long-lived tuple would be much more

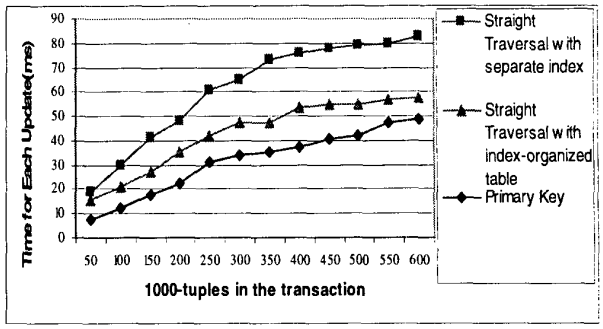


Figure 9. Updates with SPK checking on Oracle 8i

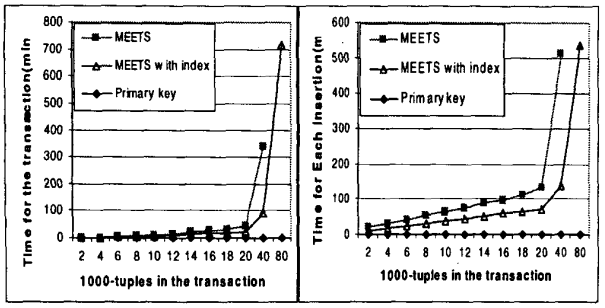
than that of a short-lived tuple.

The experiment begins by inserting a number of tuples into the referenced relation (from 2K tuples to 640K tuples). For each snapshot primary key there are eight tuples with consecutive periods, thus the number of snapshot primary key values in the referenced relation ranges from 250 to 80K. Then, we insert 10K tuples (fewer for less than 80K tuples in the referenced relation) into the referencing relation. For each test, we control how many tuples in the referenced relation overlap with the inserted tuple, from one overlapping tuple (for a short-lived inserted tuple) to eight overlapping tuples (for a relatively long-lived inserted tuple).

We first consider the Meets algorithm, which turns out to perform poorly with regard to the Straight Traversal SRI algorithm. The culprit is not the SRI check; rather, it is maintaining the DOES_MEET attribute in the referenced relation. We tried this portion of the Meets algorithm two ways, one without and one with an index (on PCN and START_DATE) on the referenced relation. In Figure 10(a), we see the total time, in minutes; in Figure 10(b), we see the per-tuple time in milliseconds (the comparison is to conventional primary key checking). Clearly maintaining that attribute simply requires too much time.

The Brute Force algorithm can also be eliminated. For inserting only 1000 tuples into a relation referencing a relation with 80K tuples, this algorithm required 270 *minutes*, which as we'll see shortly, is not even in the ballpark with the other approaches.

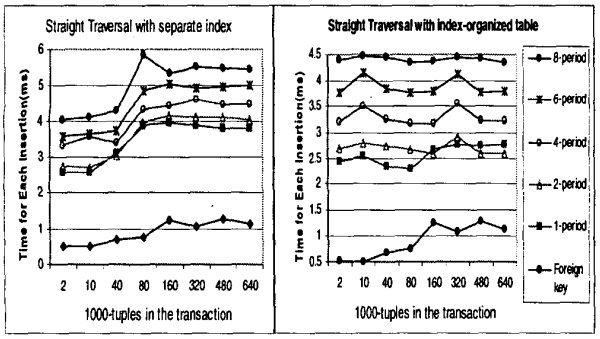
Since the Brute Force and Meets algorithms are so slow, we will now focus just on the Straight Traversal SRI algorithm. The different curves in Figure 11 compare the performance of the stratum-based Straight Traversal SRI algorithm with short-lived (overlaps one tuple) to long-lived (overlaps with eight tuples) with that of the traditional internal referential integrity checking algorithm. It is clear that



a. b.

Figure 10. The performance of the Meets algorithm on Oracle 8i

the performance of Straight Traversal SRI checking is independent to the size of referenced relation, and is only of a factor of 2 or 3 slower than the conventional foreign key checking. (As the SRI check does not consult the referencing relation, the performance would not be affected by the size of referencing relation.)



a. b.

Figure 11. The performance of SRI checking on Oracle 8i

8 Conclusions and Future Work

Sequenced primary key and sequenced referential integrity are important in temporal databases since such constraints will be prevalent, and are expensive when implemented in a brute-force manner.

This paper presents several new approaches that apply B^+-tree indexes to auxiliary relations to check sequenced constraints in a stratum architecture, thereby not requiring any changes to the underlying (conventional) DBMS. Triggers expressed in conventional SQL were used to implement sequenced constraint checking, so that no modifications to legacy code is needed to perform the sequenced checking.

We adapted the Relational-Interval Tree overlap algorithm, and also proposed a new method, the Straight Traversal SPK algorithm. For SRI, we proposed the Meets algorithm, and compared this algorithm again to the Brute Force SRI algorithm and to the Straight Traversal algorithm applied to SRI.

Our performance evaluation, using Oracle 8i and SQL Server 7, shows that both brute force approaches exhibit unacceptable performance. The Straight Traversal algorithm dominates both the Relational-Interval Tree approach (for SPK) and the Meets approach (for SRI). Further, the Straight Traversal algorithm scales nicely, with constant time per tuple, and is a factor of only three to six times slower than built-in nontemporal primary key and referential integrity checking.

In future work we would like to investigate how the Straight Traversal Approach could be applied to sequenced versions of arbitrary nontemporal constraints (e.g., a manager is always paid more than her subordinates), while retaining the excellent performance reported here. In addition, we need to consider how to handle *NOW* [Clifford97] in this approach. Because *NOW* is not constant, but rather always moves forward, the index entries in the B^+-tree may migrate from one place to another, even there is no change in the database. There is a modification of the Relational Interval Tree that accommodates *NOW* nicely [Kriegel00]; we would like to find an analogous extension for the Straight Traversal Approach.

Currently, we only used stratum approach with SQL and trigger to check the constraints. In reality, this approach can be implemented inside the DBMS very easily. The only modification is to insert two B+-tree index entries for each period. Furthermore, we can use some compressing techniques to make the index smaller. In this way, we can make the SPK checking as fast as the built-in primary key checking.

We also would also like to generalize the approaches to bitemporal relations, which support both valid and transaction time, and apply these stratum insights to embedded approaches to sequenced integrity constraint checking.

Acknowledgments

This research was supported in part by NSF grant IIS-9817798. We thank Marco Pötke and Hans-Peter Kriegel for providing us with their Oracle implementation of the Relational-Interval Tree.

References

[Chomicki95] J. Chomicki, "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding," *ACM TODS*, 20(2):149–186, June, 1995.

[Chomicki95b] J. Chomicki and D. Toman, "Implementing Temporal Integrity Constraints Using an Active DBMS," *IEEE Transactions on Knowledge and Data Engineering*, 7(4):566–582, August, 1995.

[Clifford97] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen and R. T. Snodgrass, "On the Semantics of 'Now' in Databases," *ACM Transactions on Database Systems*, 22(2):171–214, June, 1997.

[Cowley99] W. Cowley, Temporal Integrity Constriants with Temporal Indeterminacy, M.S. Thesis, University of South Florida, November, 1999.

[Edelsbrunner80] H. Edelsbrunner, "Dynamic Rectangle Intersection Searching," Institute for Information Processing Report 47, Technical University of Graz, Austria, 1980.

[Ehrich84] H. Ehrich, U. W. Lipeck, and M. Gogolla, "Specification, Semantics and Enforcement of Dynamic Database Constraints," in *Proceedings of the International Conference on Very Large Databases*, pp. 301–308, 1984.

[Gal95] A. Gal, O. Etzion, and A. Segev, "A Language for the Support of Constraints in Temporal Active Databases," in *Proceedings of the ILPS'95 Workshop on Constraints, Databases and Logic Programming*, Portland, Oregon, pp. 42–58, December, 1995.

[Jensen98] C. S. Jensen and C. E. Dyreson (eds), M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, R. Tiberio and G. Wiederhold, "A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[Jensen99] C. S. Jensen and R. T. Snodgrass, "Temporal Data Management," *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, January/February 1999.

[Kriegel00] H.-P. Kriegel, M. Pötke, and T. Seidl, "Managing Intervals Efficiently in Object-Relational Databases," in *Proceedings of the International Conference on Very Large Databases*, Cairo, Egypt, September, 2000.

[Li01] W. Li, R. T. Snodgrass, S. Deng, V.K. Gattu and A. Kasthurirangan, "Efficient Implementation of Sequenced Operations in a Stratum," TIMECENTER Technical Report, 2001.

[Lipeck87] U. W. Lipeck and G. Saake, "Monitoring Dynamic Integrity Constraints Based on Temporal Logic," *Information Systems*, 12(3):255–269, 1987.

[Nascimento99] M. A. Nascimento and M. H. Dunham, "Indexing Valid Time Databases via B+-Trees," *IEEE Transactions on Knowledge and Data Engineering*, 11(6):929–947, 1999.

[Plexousakis93] D. Plexousakis, "Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases," in *Proceedings of the International Conference on Very Large Databases*, Dublin, Ireland, 1993.

[Salzberg99] B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time Evolving Data," in *ACM Computing Surveys*, 31, 2(Jun. 1999), Pages 158 - 221

[Sistla95] A. P. Sistla and O. Wolfson, "Temporal Conditions and Integrity Constraints in Active Database Systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, pp. 269–280, 1995.

[Snodgrass95] R. T. Snodgrass (ed.), I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo and S. M. Sripada, **The TSQL2 Temporal Query Language**, Kluwer Academic Publishers, 1995.

[Snodgrass99] R. T. Snodgrass, **Developing Time-Oriented Database Application in SQL**, Morgan Kaufmann Publishers, 1999.

[Torp98] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Supporting Temporal Data Management Applications via Stratum Approaches," in *Proceedings of the 1998 International Database Engineering and Applications Symposium*, Cardiff, Wales, U.K., July 8–10, 1998.