Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

EXTENDED ABSTRACT: GENERALIZED DATA STREAM INDEXING AND TEMPORAL QUERY PROCESSING

T.Y. Cliff Leung
Richard R. Muntz

July 1991
CSD-910054

Extended Abstract:
Generalized Data Stream Indexing and Temporal Query Processing

T.Y. Cliff Leung
Richard R. Muntz

Dept. of Computer Science
University of California, Los Angeles
(213) 825-7135 tingyu@cs.ucla.edu
(213) 825-3546 muntz@cs.ucla.edu

July 12, 1991.

Extended Abstract: Generalized Data Stream Indexing and Temporal Query Processing

T.Y. Cliff Leung and Richard R. Muntz
University of California, Los Angeles

1 Introduction

Temporal databases provide several unique characteristics and challenges for query processing. Consider an example. Suppose we have a table which stores university professors — **Faculty**(Name,Rank,TS,TE) where the timestamps TS and TE represents the lifespan [TS,TE) of tuples [Sno85, Seg87]. We further assume that temporal records can be efficiently accessed in increasing TS timestamp order. One may ask who the associate professors were "as of" 2/1/1991. The peculiarity in processing this query is that the "as of time" value may not appear in the table, and we are to find the attribute values as of that time. Similar situations arise in queries asking for faculties who were associate professors for a period "intersecting" the interval [9/1/90,9/1/91). Just as in the "as of" query, the relevant time points may not appear explicitly in any tuples.

There are a number of approaches to processing temporal queries qualified with operators such as "as of", "intersect" and "between". The brute force mechanism is to scan the entire table, which of course should be avoided if possible. An alternative, known as materialized view maintenance [Bla89], is to store all the associate professor records (by selecting qualified tuples from the faculty table) in another table. Processing those queries becomes scanning the new table. This approach may provide a viable solution especially when "associate professor" often appears in the query qualification clauses and the new table is small. The tradeoffs include the overhead associated with updating the new table when there is an update to the faculty relation.

A different approach is to create a temporal index so that tuples "close" to the query-specific time interval can be accessed efficiently; this can be quite attractive when the time interval is relatively small compared with the lifespan of the table. Several indexing methods have been proposed recently [Rot87, Kol89, Lom89, Elm90, Kol91]; generally they are extensions of traditional dense indexing methods such as B+tree or multi-dimensional indexes such as R-tree and grid files, and are based on the timestamp values in tuples. However, these indexes do not necessarily handle well the types of queries mentioned above. Although range search based on timestamp values is generally supported efficiently, processing temporal joins (and perhaps more complex temporal patterns) using these indexes is likely to be inefficient.

In this paper, we propose an alternative indexing strategy. Consider the example "as of" query introduced above. It may be more efficient to search the table in such a way that perhaps only tuples with TS value greater than 1/1/1991 (and less than 2/2/1991) are accessed. The idea is briefly outlined as follows. Suppose we periodically "checkpoint" the faculty table, say at 1/1/91; specifically we store the tuple identifiers (TID's) of all associate professors as of that time and the TID of the first tuple which starts since that time. Checkpoints are in turn indexed on the checkpoint time so that TID's in them can be accessed randomly given a time value. In other words, we are creating a two level index structure. To process the query, we first access the checkpoint at 1/1/1991 via the indexs on checkpoints and use the TID's in the checkpoint to access associate professor tuples that are still "active" as of 1/1/1991 and the tuples which start since 1/1/1991 (and until 2/2/91). The query response is obtained by filtering the accessed tuples. Although the above scheme is a sparse index in the sense that not all temporal tuples can be randomly accessed, it is quite suitable for temporal databases

as older history information may not be frequently accessed. Moreover, temporal queries often ask history information which falls into a time interval specified via the temporal operators "intersect" etc., and therefore, those tuples should be accessed as efficiently as possible.

The approach combines the ideas of materialized views, periodic checkpointing and time indexing into a generalized sparse indexing mechanism. We investigate many optimization issues such as storage requirement of keeping checkpoints, query processing algorithms and their costs, at what time checkpoint should be taken and the frequency of doing so. In this extended abstract, we first present the fundamental concepts. We then discuss query processing algorithms using checkpoints and their indexes. The final section contains the related work, the conclusions, and future work as well as other research issues to be addressed in the fully developed paper.

2 Concepts

Time points are regarded as integers { 0, 1, ..., *now* } where *now* represents the "current" time. A time-interval temporal relation is denoted as **X**(S,V,TS,TE), where S is the surrogate, V is a time-varying attribute, and the interval [TS,TE) denotes the lifespan of a tuple [Sno85, Seg87]. A *data stream* is defined as a time-interval temporal relation **X**(S,V,TS,TE) sorted by TS values in increasing order. That is, tuples can be efficiently accessed in the order of successive TS timestamp values. A *query qualification* consists of a number of comparison predicates and/or join predicates connected via disjunctive ($\vee$) and/or conjunctive ($\wedge$) operators. For a query qualification P and a tuple x, P(x) is a predicate obtained by instantiating P with constants in x. We define Temporal Select-Join (TSJ) as:

$$\sigma_P(\mathbf{R_1} \times \ldots \times \mathbf{R_n}) \qquad \text{or} \qquad \sigma_P(\mathbf{R_1}, \ldots, \mathbf{R_n})$$

where P is the query qualification. In this extended abstract, we consider only the following temporal join operators in TSJ; note that they are really shorthands for the specific query qualification [Leu90]:

meet-join(X,Y) — "X.TE=Y.TS"
contain-join(X,Y) — "X.TS<Y.TS $\wedge$ Y.TE<X.TE"
equal-join(X,Y) — "X.TS=Y.TS $\wedge$ X.TE=Y.TE"
start-join(X,Y) — "X.TS=Y.TS $\wedge$ (X.TE<Y.TE $\vee$ X.TE>Y.TE)"
finish-join(X,Y) — "(X.TS<Y.TS $\vee$ X.TS>Y.TS) $\wedge$ X.TE=Y.TE'
overlap-join(X,Y) — "(Y.TS<X.TS $\wedge$ X.TS<Y.TE) $\vee$ (X.TS<Y.TS $\wedge$ Y.TS<X.TE)".

The semantics of three commonly found temporal operators, which are also of interest here, are:

- **between** — given a time point t and a time interval $[t_s,t_e)$, "t **between** $[t_s,t_e)$" holds if and only if "$t_s \leq t \wedge t < t_e$" holds.
- **intersect** — given a time-interval tuple $x<s,v,t_1,t_2>$ and a time interval $[t_s,t_e)$, "x **intersect** $[t_s,t_e)$" produces:

  $x<s,v,t_1^+,t_2^->$ if the intersection of intervals $[t_1,t_2)$ and $[t_s,t_e)$ produces a non-null interval $[t_1^+,t_2^-)$, or null tuple otherwise.

  Given **X**(S,V,TS,TE), "**X intersect** $[t_s,t_e)$" is defined as $\bigcup_{x \in \mathbf{X}}$ { x **intersect** $[t_s,t_e)$ }.
- **as of** — "**as of** t" is equivalent to "**intersect** [t,t+1)".

The proposed scheme is regarded as a generalized indexing mechanism; we briefly elaborate on this point here. Conceptually, indexes can be specified at two logical levels: *indexing condition* and *implementation.* Indexing conditions are queries that specify the content of index records while the implementation level specifies the file structure that stores the index records. For example, a conventional index to a relation **X** on an attribute A can be viewed as a projection on A and the TID's. That is, the indexing condition can be expressed in the following query, assuming that TID is an attribute that users can reference:

$$\pi_{(A,TID)}\ (\mathbf{X}).$$

Each index record contains the data value and all the TID's of tuples that have the data value. Similarly, a join index [Val87] between two relations **X** and **Y** on an attribute A can be expressed as:

$$\pi_{(X.A,TID_X,TID_Y)}\ \sigma_{(X.A=Y.A)}\ (\mathbf{X}, \mathbf{Y}).$$

Note that both join indexes [Val87] and partial indexes [Sto89] such as a range of attribute values (e.g. 30 < age < 40) fit nicely in this generalized indexing framework. In the proposed scheme, we support the specification of complex queries in TSJ as indexing conditions such as overlap-join:

$$\pi_{(TID_X,TID_Y)}\ \sigma_{((Y.TS<X.TS \wedge X.TS<Y.TE) \vee (X.TS<Y.TS \wedge Y.TS<X.TE))}\ (\mathbf{X}, \mathbf{Y}).$$

That is, each index record contains the TID's of joined tuple pair.

For the implementation specification, the most common file structures for the conventional indexes include B-tree, hashing and heap files [Ull82]. It becomes evident later that existing implementation techniques such as B-tree can be easily adopted in our approach. The main advantage of such generalized indexing scheme is that it forms the basis for users to be able to flexibly specify indexing conditions that are not anticipated by DBMS designers. Other possible advantages are that it unifies views and materialized view maintenance [Bla89,Sto90]. In the following, we focus on the implementation aspect of the proposed scheme tailored for temporal databases.

Consider an indexing condition Q $\in$ TSJ whose query qualification is P on data streams **X** and **Y** as depicted in Figure 1; one can easily generalize the mechanism for more than two data streams. In this paper, we take a different view as opposed to dense indexes. More specifically, we periodically checkpoint the execution of Q on **X** and **Y** along the time axis [1], and checkpoints are in turn indexed on their checkpoint times. For the rest of this section, we discuss in more details on what checkpoints are, but to put it simply, a checkpoint (such as ck_2 in Figure 1) at a time point (say $t(ck_2)$) contains some information of the execution of Q on **X** and **Y** such that the response of Q on tuples that started between time point 0 and t where $t>t(ck_2)$, can be obtained in the following way [2]:

> Access the checkpoint ck_2 using the time index on checkpoints and tuples in operand data streams which started since $t(ck_2)$. "Continue" the execution of Q using the accessed tuples.

Note that not all tuples in operand data streams can be accessed directly, that is, we are indeed creating a sparse index on data streams using Q. We next discuss precisely how checkpointing is performed and what kinds of queries can be processed in this approach.

[1] The frequency of performing checkpointing and the time at which checkpointing is performed involve tradeoffs that can greatly affect the efficiency of the proposed scheme and is a subject to be addressed.

[2] Note that some temporal joins such as Q $\equiv$ before-join(X,Y) whose join condition is "X.TE<Y.TS" does not allow us to process user queries like this. However, we note that the temporal joins listed earlier or simply select queries enable us to do so. The class of user queries and indexing conditions is an issue to be investigated.

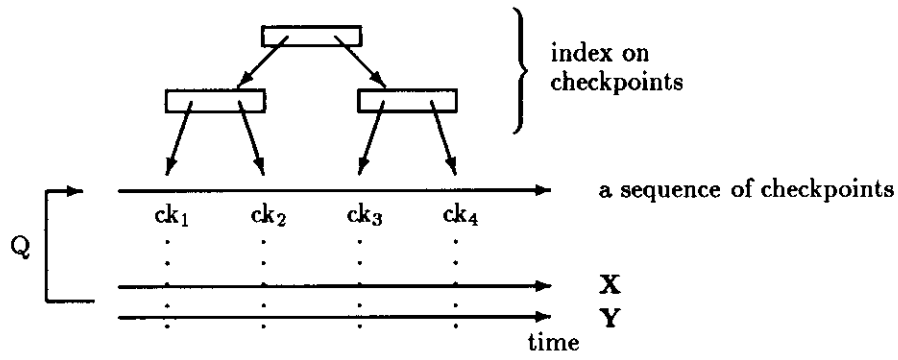

Figure 1: Checkpoints on data streams **X** and **Y** using query Q as the indexing condition

Four kinds of information can be stored in a checkpoint, denoted as ck — *checkpoint time*, *state information*, *incremental results* and *data stream pointers*. For a checkpoint ck, let the checkpoint prior to ck be denoted as ck^- and define [3]:

1. The checkpoint time, denoted as t(ck), is the time at which the checkpoint is performed.
2. The state information, denoted as s(ck), contains the TID's of all tuples $x \in \mathbf{X}$ such that
   - "t(ck) **between** [x.TS,x.TE)" holds, i.e., x spans the checkpoint time, and
   - the instantiated query qualification P(x) is *satisfiable* [4],

   and all tuples $y \in \mathbf{Y}$ such that
   - "t(ck) **between** [y.TS,y.TE)" holds, i.e., y spans the checkpoint time, and
   - the instantiated query qualification P(y) is satisfiable.

   Basically the state information contains tuples which are "active" as of the checkpoint time and partially satisfy the query qualification, that is, they potentially join with "future" tuples. Note that tuples in s(ck) either belong to $s(ck^-)$ or start between the interval $[t(ck^-),t(ck))$.
3. Let T denote the set of tuples that overlap with the interval $[t(ck^-),t(ck))$. The incremental result, denoted as ir(ck), in ck essentially contains the TID's of all tuples that contribute to the response of executing Q on tuples in T. More specifically, ir(ck) contains the TID's of all tuple pairs (x, y) such that
   - $x \in \mathbf{X}$ and $y \in \mathbf{Y}$ and
   - $x \in s(ck^-)$ or "x.TS **between** $[t(ck^-),t(ck))$" holds, and
   - $y \in s(ck^-)$ or "y.TS **between** $[t(ck^-),t(ck))$" holds, and
   - the (x, y) pair satisfies the query qualification P.
4. [*] The data stream pointers, denoted as dsp(ck), contains the TID of tuple $x \in \mathbf{X}$ such that x has the smallest TS value in **X** but greater than t(ck). Similarly, dsp(ck) contains the TID of the qualified tuple $y \in \mathbf{Y}$. Using the data stream pointers, one can access tuples which start after t(ck).

[3] It follows that $t(ck^-) < t(ck)$. If there is no such ck^-, ck^- and $t(ck^-)$ are assumed to be an empty set and 0 respectively.

[4] Testing satisfiability of P(x) is simple. We first obtain a new predicate P_X from P by replacing all terms in P involving Y with "true". That is, P_X contains only comparison predicates involving only X. We then substitute the tuple x into P_X obtaining $P_X(x)$. If $P_X(x)$ is true, then P(x) is satisfiable.

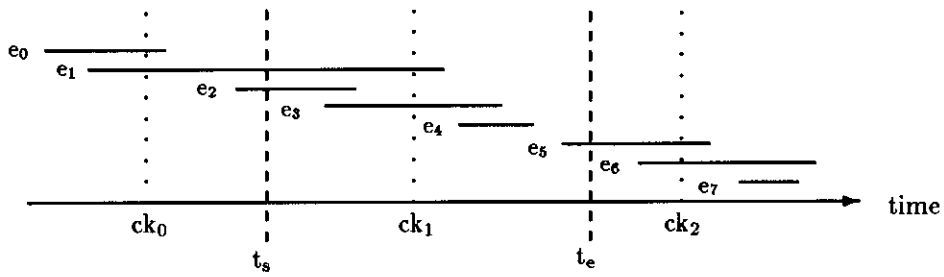

Figure 2: A data stream **X**(S,V,TS,TE) and checkpointing

Note that the state information may still require a lot of storage space when many tuples span the checkpoint time. The required storage space can be reduced as follows. Suppose that there are only a few active tuples at a time point t^- prior to the checkpoint time t(ck). One can store in dsp(ck) the first tuple which starts since t^-, and store in s(ck) the tuples which start prior to t^- and are still active as of t(ck). The optimization seems particularly good when there is a lot of update activity at around the checkpoint time t(ck) and fewer active tuples at t^-. As we will discuss query processing algorithms shortly, processing queries would require access more tuples, i.e., those which start between [t^-,t(ck)), although s(ck) contains less TID's. Let us now illustrate how checkpointing can be performed using several concrete examples. In the next section, we discuss how checkpoints and indexes on checkpoints can be used to process ad-hoc user queries.

Example 1) Suppose the indexing condition on a data stream **X**(S,V,TS,TE) is $Q \equiv \sigma_{V>10}(\mathbf{X})$. The following steps perform a checkpoint ck on **X** using Q:

1. t(ck) contains the checkpoint time.
2. s(ck) contains all tuples $x \in \mathbf{X}$ such that
   - "t(ck) **between** [x.TS,x.TE)" holds, i.e., x is active at the checkpoint time, and
   - "x.V > 10" holds.
3. ir(ck) contains all tuples $x \in \mathbf{X}$ such that
   - $x \in s(ck^-)$ or "x.TS **between** [t(ck^-),t(ck))" holds, and
   - "x.V > 10" holds.
4. dsp(ck) contains the TID of tuple $x \in \mathbf{X}$ which has the smallest TS value in **X** but greater than t(ck).

Consider an example data stream in Figure 2, which shows only the tuples that satisfy the query qualification "V>10". Assuming there is no checkpoint prior to ck_0, the contents of checkpoints ck_0, ck_1 and ck_2 are listed in Table 1.

Example 2) Consider that the indexing condition is overlap-join(**X**,**Y**) on data streams **X** and **Y**, where the query qualification P is the overlap-join condition — "(Y.TS<X.TS $\wedge$ X.TS<Y.TE) $\vee$ (X.TS<Y.TS $\wedge$ Y.TS<X.TE)". For a checkpoint ck:

	ck_0	ck_1	ck_2
t	$t(ck_0)$	$t(ck_1)$	$t(ck_2)$
s	$\{e_0, e_1\}$	$\{e_1, e_3\}$	$\{e_5, e_6\}$
ir	$\{e_0, e_1\}$	$\{e_0, e_1, e_2, e_3\}$	$\{e_1, e_3, e_4, e_5, e_6\}$
dsp	$\{e_2\}$	$\{e_4\}$	$\{e_7\}$

Table 1: Checkpoints on data stream X in Figure 2

- t(ck) contains the checkpoint time.
- s(ck) contains tuple $x \in X$ and tuple $y \in Y$ that are active at the checkpoint time t(ck). Note that P(x) and P(y) are always satisfiable.
- ir(ck) contains (x,y) tuple pairs where
  - tuple $x \in \mathbf{X}$ such that $x \in s(ck^-)$ or "x.TS **between** $[t(ck^-),t(ck))$" holds, and
  - tuple $y \in \mathbf{Y}$ such that $y \in s(ck^-)$ or "y.TS **between** $[t(ck^-),t(ck))$" holds, and
  - the (x,y) tuple pair satisfies overlap-join condition P.
- dsp(ck) contains the TID's of tuples X and Y defined in step [*] earlier in this section.

Given a sequence of checkpoints as illustrated in Figure 1, one can easily build an index on checkpoints based on the checkpoint times. That is, given a time point t, the checkpoint taken at t, or the previous or the next checkpoint can be accessed directly. More importantly, this type of indexing can be implemented using conventional file structures such as B-tree.

3 Query Processing

In this section, we discuss processing algorithms for several types of temporal queries using the proposed checkpointing and indexing scheme. Let us consider the following indexing condition Q and user query Q' to be processed:

- The indexing condition is a query $Q \in$ TSJ whose query qualification is P. Assume there is a sequence of checkpoints on operand data streams based on Q.
- Given an expression $E \in$ TSJ whose query qualification is P'. The query to be processed Q' has the following form:

  $Q' \equiv E$ **intersect** $[t_s,t_e)$, or
  $Q' \equiv E$ **as of** t_s.

For the sake of explanation, we assume that E is subsumed by Q for the moment, that is, the indexing condition implies the user query qualification. Without checkpoints and indexes, processing Q' may be very costly, particularly when the time interval $[t_s,t_e)$ is relatively small compared with the lifespans of data streams. In our scheme, one can process Q' by accessing only a significantly smaller set of tuples that overlap with the time interval $[t_s,t_e)$ using the indexes and checkpoints.

In order that the incremental results in checkpoints can be used for processing Q', the indexing condition has to imply the user query qualification, i.e., P $\Rightarrow$ P'. The query processing algorithm is this:

Algorithm (1)

1. Retrieve tuples using the incremental results in all checkpoints from ck_s to ck_e, where

   ck_s is the earliest checkpoint after t_s, and ck_e is the earliest checkpoint after t_e.

   If there is no ck_e, the last checkpoint prior to t_e, denoted as ck_e^-, will be used. Retrieve

   (a) tuples r using TID's in $s(ck_e^-)$ and
   (b) tuples r such that "r.TS **between** $[t(ck_e^-),t_e)$", i.e. r starts since $t(ck_e^-)$, by following the TID's in $dsp(ck_e^-)$.

2. For each tuple r obtained in (1), select the tuple that satisfies the user query qualification P'; the resulting tuples have valid time interval equal to the intersection of [r.TS,r.TE) and $[t_s,t_e)$.

Example 3) Consider the sequence of checkpoints in Figure 2. Suppose Q' $\equiv$ ($\sigma_{V>10}$ (**X**)) **intersect** $[t_s,t_e)$. Obviously, the indexing condition (V>10) implies the user query qualification (V>10). Using the algorithm (1), checkpoints ck_s and ck_e are ck_1 and ck_2 respectively. The set of tuples obtained using $ir(ck_1)$ and $ir(ck_2)$ in step (1) is $\{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$ and tuples e_0 and e_6 are eliminated in step (2) yielding $\{e_1, e_2, e_3, e_4, e_5\}$. A point to note is that the incremental results prior to ck_1 and after ck_2 are not accessed and therefore this query processing strategy has substantial savings in access times particularly when the interval $[t_s,t_e)$ is relatively small.

Instead of using the incremental results, the state information and data stream pointers in checkpoints can also be used to process Q'. It turns out that the following classes of queries can be processed using the algorithm (2) below; recall that P is the query qualification of the indexing condition Q and P' is the user query qualification:

1. Q' $\equiv$ ($\sigma_{P'}$ (**X, Y**)) **intersect** $[t_s,t_e)$, where P $\Rightarrow$ P'.
2. Q' $\equiv$ ($\sigma_{P'}$ (**X**, y)) **intersect** $[t_s,t_e)$, where P $\Rightarrow$ P' and $y \in$ Y.
3. Q' $\equiv$ ($\sigma_{P'}$ (**X**)) **intersect** $[t_s,t_e)$, where $\pi_X(P) \Rightarrow \pi_X(P')$.
   $\pi_X(P)$ is obtained by replacing the following terms in P with "true":

   - join predicates, and
   - comparison predicates that do not involve attributes in X.

   In other words, $\pi_X(P)$ contains comparison predicates involving only data stream X. Similarly we obtain $\pi_X(P')$. The implication means that the state information in checkpoints obtained using P is a superset of the state information that would have been obtained using P' instead of P, and therefore we can use the TID's in the state information for query processing.

4. Q' $\equiv$ ($\sigma_{P'}$ (**X, Y**)) **intersect** [t_s,t_e), where $\pi_X(P) \Rightarrow \pi_X(P')$ and $\pi_Y(P) \Rightarrow \pi_Y(P')$.

Algorithm (2)

1. Retrieve the tuples using TID's in $s(ck_s)$ where ck_s is the latest checkpoint prior to t_s.
2. Retrieve tuples which start in $[t(ck_s),t_e)$ by following TID's in $dsp(ck_s)$.
3. The set of all tuples from (1) and (2) contains all the tuples in the response. Select tuples that satisfy P'; the resulting tuples have valid time set accordingly.

Example 4) Again, we consider the example in Figure 2. The checkpoint prior to t_s is ck_0, and therefore $s(ck_0) = \{e_0, e_1\}$. Following the TID in $dsp(ck_0) = \{e_2\}$, tuples e_2, e_3, e_4, e_5, and e_6 can be retrieved in step (2). When the tuple e_6 is accessed, step (2) stops and e_6 is discarded from the response as its TS value is greater than t_e. Note that e_0 is filtered out in step (3) as the interval intersection yields null tuple, resulting $\{e_1, e_2, e_3, e_4, e_5\}$ in the response.

Example 5) Consider Example 2 where the indexing condition is overlap-join(X,Y). The checkpoints and the corresponding indexes can actually speed up the processing of a large class of queries, i.e., any combination of (a)–(h) and (1)–(2) queries:

(a) meet-join(X,Y)
(b) meet-join(Y,X)
(c) contain-join(X,Y)
(d) contain-join(Y,X)
(e) equal-join(X,Y)
(f) start-join(X,Y)
(g) finish-join(X,Y)
(h) overlap-join(X,Y)

(1) "**intersect** [t_s,t_e)"
(2) "**as of** t_s"

Readers should note that only query (h) can be processed using incremental results in checkpoints while queries (a)–(g) can be processed using only the state information and data stream pointers.

4 Related Work, Conclusions & Future Work

Several temporal indexes have recently proposed [Rot87, Kol89, Lom89, Elm90, Kol91]. To the best of our knowledge, there has been no proposal on sparse indexing on temporal relations. The proposed scheme fills in the gap between building a full-fledge dense index and no indexing, and therefore provides a good alternative for random accesses. The pros and cons of this scheme can be highlighted as follows. First, it is envisioned that existing software (e.g. B-tree) can be reused in the implementation. Moreover, as complex queries can be indexing conditions, queries such as temporal joins qualified with temporal operators such as **intersect** can be more efficiently processing. This is a great advantage over no indexing at all. On the other hand, query processing generally requires additional computation, and the implementation of query processor becomes more complicated as complex queries can be the indexing conditions.

In this extended abstract, we argue that the proposed scheme is especially suitable in temporal database environment. We present the basic concepts, particularly on how checkpointing using user-defined queries (including temporal joins) are performed. Four types of information can be stored in checkpoints: checkpoint time, state information, incremental results and data stream pointers, which can be exploited in query processing. A number of research directions are now undertaking. Firstly, the class of queries that can be used as indexing conditions appears to be more powerful than the temporal join operators listed earlier. We intend to investigate this. Secondly, we will formalize the idea of generalized indexing mechanism. As we pointed out earlier, it appears that many DBMS functionalities such conventional indexes and materialized view maintenance can be subsumed in this framework. Finally, we will study other implementation issues of the proposed scheme such as tradeoffs involving the frequency of checkpointing, the checkpoint time, the checkpoint storage requirement and the costs of query processing.

References

[Bla89] Blakeley J.A., Coburn N. and Larson P-A., 'Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates,' *ACM Transactions on Database Systems*, Vol. 14, No. 3, September 1989, pp.369–400.

[Elm90] Elmasri R., Wuu G. and Kim Y.J., 'The Time Index: An Access Structure for Temporal Data,' *Proceedings of the 16th Int. Conf. on Very Large Data Bases*, 1990, pp.1–12.

[Kol89] Kolovson C. and Stonebraker M., 'Indexing Techniques for Historical Databases,' *Proceedings of the IEEE Int. Conf. on Data Engineering*, February 1989, pp.127–137.

[Kol91] Kolovson C. and Stonebraker M., 'Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, 1991, pp.138–147.

[Leu90] Leung T.Y. and Muntz R., 'Query Optimization for Temporal Databases,' *Proceedings of the IEEE Int. Conf. on Data Engineering*, 1990, pp.200–207.

[Lom89] Lomet D. and Salzberg B., 'Access Methods for Multiversion Data,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, 1989, pp.315–324.

[Rot87] Rotem D. and Segev A., 'Physical Design of Temporal Databases,' *Proceedings of the IEEE Int. Conf. on Data Engineering*, February 1987, pp.547–553.

[Seg87] Segev A. and Shoshani A., 'Logical Modeling of Temporal Data,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, May 1987, pp.454–466.

[Sno85] Snodgrass R., and Ahn I., 'A Taxomony of Time in Databases,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, May 1985, pp.236–246.

[Sto89] Stonebraker M. 'The Case for Partial Indexes,' *the ACM SIGMOD Record*, Vol.18, no.4, December 1989, pp.4–11.

[Sto90] Stonebraker et. al., 'On Rules, Procedures, Caching and Views in Data Base Systems,' *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, 1990, pp.281–290.

[Ull82] Ullman J.D., 'Principles of Database Systems,' 2nd edition, Computer Science Press, Rockville, Md., 1982.

[Val87] Valduriez P. 'Join Indices,' *ACM Trans. on Database Systems*, Vol.12, No.2, June 1987, pp.218–246.