First publ. in: Proc. of the 3rd Internat. Baltic Workshop on Database and Information Systems, April 1998

Supporting Temporal Query Processing by Hash Filters

Holger Riedel [*]

Abstract

Hash filters are very useful for the optimization of query processing, especially in the case of join queries. In this paper, we investigate the possibilities how hash filters can be utilized in the field of temporal queries. We introduce a flexible kind of such hash filters and discuss different variations. Additionally, we analyze the usefulness of this concept for some well-known types of temporal queries.

1 Introduction

The processing of queries for temporal data significantly differs from well-known techniques in relational or object-oriented databases, because the time dimension has specific properties. For instance, most temporal queries can only inefficiently be handled, if the data is treated like relational data using operators of the relational algebra as query processing primitives.

A major problem is the representation of the history, if the whole temporal information of an object or tuple can consist of a union of different time intervals, like the employment information of part-time workers. This is supported by the TSQL2 proposal [13], but cannot be represented in a 1NF manner in a single tuple using start and stop times. Thus, either a specific "nested" representation or multiple "internal" tuples are necessary to represent a single "conceptual" tuple or object. This complicates the evaluation of queries (like testing predicates for selections or temporal joins) significantly.

Another problem is given by the specific temporal query types which must be supported. For instance, TSQL2 supports five different types of temporal conditions, as described in Figure 1. The TSQL2 data model is based on tuple timestamping. We call a chronon *active*, if it belongs to the history of the tuple or object.

Usually, the temporal data is stored in a similar way as relational data with specific extensions. As an example, Woo and Elmasri [18] propose a specific relational design where a specific attribute is used to mark the specific state of the tuple, which can be used to avoid updates of old tuples. Additionally, specific indices are proposed to support the query processing. An overview can be found in [10]. Often these temporal indices are based on spatial data structures [11] like the R-tree [3]. Although there are some similarities between the temporal and the spatial dimension

[*]adress: Fakultät für Mathematik und Informatik, Universität Konstanz, D-78457 Konstanz, Germany, email: `Holger.Riedel@uni-konstanz.de`

`valid(t1) = valid(t2)`	`t1` and `t2` have the same active chronons.
`valid(t1) OVERLAPS valid(t2)`	`t1` and `t2` have at least one active chronon in common.
`valid(t1) CONTAINS valid(t2)`	All active chronons of `t2` are also active for `t1`.
`valid(t1) PRECEDES valid(t2)`	`t1` is finished, before `t2` begins.
`valid(t1) MEETS valid(t2)`	The begin of `t2` is exactly one chronon after the end of `t1`.

Figure 1: Temporal conditions in TSQL2

(e.g. the mentioned condition predicates of TSQL2 also occur in spatial databases with a slightly different semantics), there is a major divergence that the data of a temporal object may change quite often or has indefinite length which can only inefficiently be incorporated into spatial data structures like the R-tree. A different index concept is the Time-Index [4], where all important instants (i.e., any instant where a "logical" update occurs) are held in a B-tree with references to the relevant database objects.

Another efficient method is hashing, which is widely explored for query processing, mostly for join queries [17, 1, 8]. Up to our knowledge, hashing for temporal query processing has only be discussed as a method for the efficient allocation of blocks within the query processing [2]. Nevertheless, the use of concepts like partitioned hashing [16] or encoding of the history similar to the the z-order encoding in spatial databases [7] seem to be useful to describe the histories of objects. Similar ideas of non-standard hashing are presented in the context of complex object databases in [15], where complex objects are encoded into hash values in order to simplify the equality test.

In our approach, temporal information of an object or a tuple is encoded in a hash value of a fixed length. The hash values can be used to check temporal predicates without accessing the primary data. Thus, these hash values can be used to implement query processing techniques which use hash values as filters to simplify the implementation of selections or joins. More specifically, the temporal hash value can be used in a flexible way:

- The basic granularity of the hash value can be different from the temporal granularity in the database. This is important, because the choice of the granularity in the conceptual model should not be interfered by physical design decisions like the parametrization of an index.
- Often, current data is more important than old data. This can be supported in our approach by using more bits for instants near to the referencing instant. Alternatively, time-intervals of the same size are supported, too.
- Because each bit of the hash value represents a certain time interval I, it has must be clarified whether a `1` in the hash value expresses the fact that the

object is valid at least for one or for all chronons of I. We support both alternatives.

In the next section, we describe the basic structure of the temporal hash values, and in Section 3, we give a detailed analysis how the condition predicates of TSQL2 can be evaluated using these concepts.

2 Temporal hash filters

2.1 The basic idea

We propose a storage structure for temporal data where the temporal information about an object or an attribute is encoded as a hash value. Each hash value is a representation of the complete history of the indexed attribute up to a certain *reference instant* r.

These hash values can be used in a hash-based query processing environment as reference values to check certain conditions. As an alternative, it may also be useful to generate physical addressses out of the hash values.

A temporal hash-index can be seen as a set of triples (o, h, r) where o is the referenced object or attribute value, h is the representation of the history of o, and r is the referencing instant. The hash values are variable in time, i.e. we store hash values with different reference instants in a single file and use them in parallel to process the queries. Therefore, it is necessary that such hash values can be compared using some shifting of a hash value to a different reference instant.

The detailed structure of the hash value h is as follows. Within the hash value, temporal information of a certain interval I is encoded into one bit, thus the temporal interval $(-\infty, r]$ is partitioned into $n = 2^m$ $(m = 1, 2, 3, \ldots)$ parts, each with a representing bit. In our approach, the hash values can be parametrized by the following dimensions:

- *linear* $\leftrightarrow$ *logarithmic:* In many time-oriented applications, the current data is more important than older parts of the database. Thus it may be useful to describe "data near to the referencing instant" by more bits than data of earlier times. We support two alternatives:
  - *linear:* All but the lowest bit represent a time interval of a fixed length, which is called the *granularity* of the index.
  - *logarithmic:* The most current index-granule is represented by $n/2$ bits, the next by $n/4$ bits, the next by $n/8$ bits and so on. The m-th index-granule is represented by one bit and all index-granules before are put together into one.
- *sparse* $\leftrightarrow$ *dense:* Because the granularity of the index is usually not the granularity of the represented data, a bit in the index usually refers to several instants. Thus, it has to be fixed whether a `1` represents that *at least one* or *all* instants are in the lifetime of the refered object. We support both alternatives where the first alternative is called *sparse*, and the second *dense*.

`and`	0	1	?
0	0	0	0
1	0	1	?
?	0	?	?

`or`	0	1	?
0	0	1	?
1	1	1	1
?	?	1	?

Figure 2: Three-valued `and` and `or`

Employees		
Employee	*employment*	*ssno*
Peter	[2/92 - 2/95; 5/95 - 9/96]	55446677
Paul	[2/92 - 12/94; 4/95-10/95]	55448867
Mary	[4/96 - 12/97]	55338967
Doris	[7/89 - 12/89; 7/90-12/90]	55338922

Figure 3: The relation *Employees*

It is important to crasp that the granularity of the database and the granularity of the hash values are two different concepts:

- the *database granularity* is a conceptual decision how temporal information is recorded, e.g., the employment time is recorded month by month.
- The *granularity of the hash values* fixes the distance between reference instants, and may be chosen upon physical database design decisions. Thus, it may be useful to use a representation year by year, where always the refencing instant is the 31st December.

In the following we use the notation *"*`[11110010]`$_{31.12.1996}$ *is an (8-bit,dense-log,1 year) hash value"* to describe that `[11110001]` is a hash value with referencing instant 31st December 1996, as a 8-bit hash value with granularity of one year, based on the dense and logarithmic dimensions. In order to simplify the notations of the examples below, we use `[11110010]`$_{1996}$ as a shorthand for `[11110010]`$_{31.12.1996}$. We use the abbreviations *lin* and *log* to refer to linear or logarithmic hash values. We enumerate the bits of a hash value from 1 to n, where the first bit is the *past-infinity bit*, while the bit n refers to the time interval directly before the referencing instant. Because the bits in the hash value directly correspond to the truth values *true* and *false*, we use the functions `and` and `or` to describe bitwise computations with the hash values. Later on, we need a three-valued approach. So we use `and` and `or` as three-valued operators as given in Figure 2.

Example 1 In order to describe the functionality of temporal hash values, we use a temporal relation *Employees* as given in Figure 3. Because we only use the attribute *employment* for temporal queries in the following examples, we do not add further temporal aspects to the example instance. Thus, the temporal information given through *employment* can be seen as the timestamp for the whole tuple in this example.

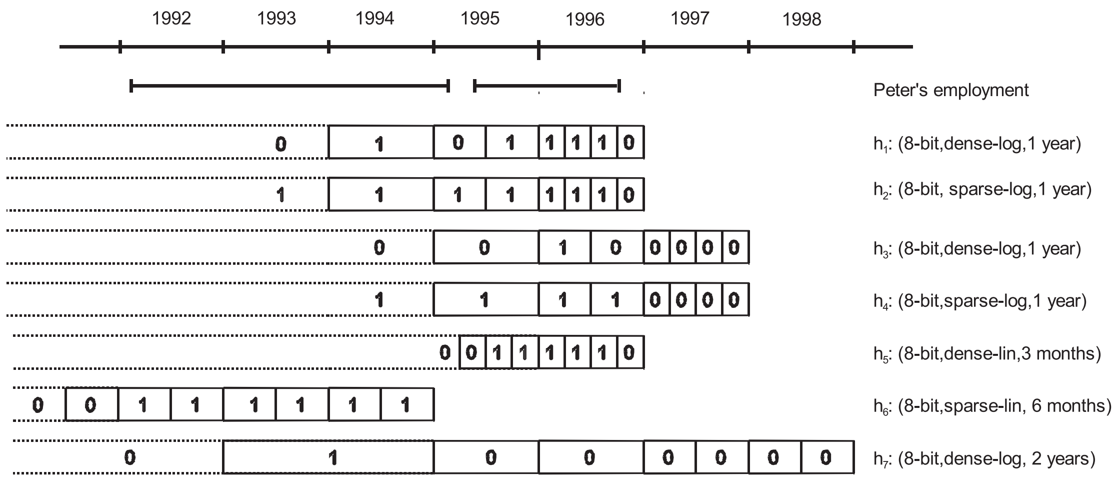
1992
1993
1994
1995
1996
1997
1998
Peter's employment
h1: (8-bit,dense-log,1 year)
h2: (8-bit, sparse-log,1 year)
h3: (8-bit,dense-log,1 year)
h4: (8-bit,sparse-log,1 year)
h5: (8-bit,dense-lin,3 months)
h6: (8-bit,sparse-lin, 6 months)
h7: (8-bit,dense-log, 2 years)

Figure 4: Representation of Peter's employment by 8-bit hash values.

In Figure 4 we use *Peter's* employment times as an example how temporal hash values can be built. In order to simplify the example we use 8 bits, although in practice bigger sizes are very reasonable.

h_1 is a (8-bit, dense-log, 1 year) hash value. Because it is logarithmic, the four highest bits are used for 1996, each for a quarter, 1995 is described in half-year terms and 1994 by a single bit. The complete history up to 1993 is represented by a `0`, because there are instants in the past where Peter was not employed.

The only difference between h_1 and h_2 is that h_2 is sparse, thus the the third bit of h_2 is `1`, because there are instants in the first half of 1995 where Peter was employed. h_3 and h_4 have the same dimensions as h_1 resp. h_2, but the referencing instant is 31st December 1997. Thus, the year 1995 is only represented by one bit in these cases. h_5 is a linear hash value, where each bit represents 3 months. Because the granularity is 3 months, the possible referencing instants are 31st March, 30th June, 30th September and 31st December. h_6 is a (8-bit, sparse-lin, 6 months) hash value referenced at the end of 1995. As the picture shows, h_6 is inappropiate to encode the history of Peter's employment, because its reference instant is not beyond the end of Peter's employment. h_7 is an example of an (8-bit, dense-log, 2 year) hash value. □

2.2 Temporal hash values and query processing

Temporal hash values used can be used to store information about temporal data either in an index or as adresses in hashed based storage environment.

A major reason for the invention of the temporal hash filters was that they can easily be compared with each other even when the referencing instants differ. Before we go into the details, we give an example to show the basic mechanism how queries can be evaluated.

Example 2 We like to check whether Paul was always employed when Peter was employed ($employment(Paul) \supseteq employment(Peter) \equiv Query$ 1) or whether they have at least one month in common ($\equiv Query$ 2). Using (8-bit, dense-log, 1 year) hash values, the hash values are $[\texttt{011110110}]_{1995}$ for Paul and $[\texttt{01011110}]_{1996}$ for

Peter. To use a bitwise comparison, Paul's hash value is shifted to 1996 by setting the four bits of 1996 to `0`, and evaluating the other bits using the given hash value:

1993 and earlier:	`0 and 1`	$\Rightarrow$	`0`
1994:	`1 and 1`	$\Rightarrow$	`1`
1/1995:	`0 and 1`	$\Rightarrow$	`0`
2/1995:	`1 and 0`	$\Rightarrow$	`0`

The resulting hash value is $\texttt{[01000000]}_{1996}$. This can be compared with Peter's hash value $\texttt{[01011110]}_{1996}$ bit-by-bit. Bit 2 shows that both are employed for the whole year 1994 ($\Rightarrow$ *Query* $2 = \texttt{true}$) and Peter is completely employed in the first three quarters of 1996, while Paul is not ($\Rightarrow$ *Query* $1 = \texttt{false}$). $\square$

The important point of the example is that it is necessary to compare index values of different referencing instants. This can be done by shifting one hash value to the other. Because we have a fixed granularity of the referencing instants this works well even in the logarithmic dimension.

The details about shifting hash values are explained in the next section, while the evaluation schema of different query types are discussed in Section 3.

2.3 Shifting hash values

Each hash value $hv_{old} = (o, h_{old}, r_{old})$ can be shifted to the next allowed referencing instant $r_{old} + 1$. All bits refering to instants between r_{old} and $r_{old} + 1$ are set to `0`, and the others are newly calculated.

Definition 1 (Shifting of a hash value) Let $hv_{old} = (o, h_{old}, r_{old})$ be a hash value of length n.

hv_{old} can be shifted to $hv_{new} = (o, h_{new}, r_{old} + 1)$, where h_{new} is given by

- *in the linear case:*

$$h_{new}(j) = \begin{cases} 0 & : \quad j = n \\ h_{old}(j+1) & : \quad j \in \{2, \ldots, n-1\} \\ h_{old}(1)\ \texttt{and}\ h_{old}(2) & : \quad j = 1 \text{ and } h_{old} \text{ is dense} \\ h_{old}(1)\ \texttt{or}\ h_{old}(2) & : \quad j = 1 \text{ and } h_{old} \text{ is sparse} \end{cases}$$

- *in the logarithmic case:*

$$h_{new}(j) = \begin{cases} 0 & : \quad j \in \{n/2+1, \ldots, n\} \\ h_{old}(2j-1)\ \texttt{and}\ h_{old}(2j) & : \quad j \in \{1, \ldots, n/2\} \text{ and } h_{old} \text{ is dense} \\ h_{old}(2j-1)\ \texttt{or}\ h_{old}(2j) & : \quad j \in \{1, \ldots, n/2\} \text{ and } h_{old} \text{ is sparse} \end{cases}$$

$\diamond$

Example 3 As explained in Example 1, h_3 is a shift of h_1 by one index-granule, in this case from 1996 to 1997. $\square$

Shifting is always directed into the future, because shifting into the past is not possible:

- A basic assumption about a hash value is that it encodes the complete history. Thus, moving the reference instant backwards leads to some information loss, because then the new hash value does not describe some parts of the history which was present before, especially when some `1`s get lost.

- It would be necessary to split an interval into two intervals (using linear hash values, this only occurs for the past-infinity bit). It is not clear how the new hash value should be built then. For instance, shifting h_3 from 1997 to 1996, we cannot obtain h_1, because $h_3(1), h_3(2)$, and $h_3(4)$ cannot be split, because a `0` in a dense hash value gives no information whether some of the referenced parts are completely active or not.

3 Query Processing with hash filters

3.1 Temporal queries

In the literature, there are many proposals for temporal query languages [9, 12] with a rather wide range of supported query types. Because we are mainly interested in index support for temporal queries, we focus on the specific temporal conditions which are present in temporal selections and temporal joins. More precisely, we choose the five condition types of TSQL2 as introduced in the Introduction, because they cover a broad range of temporal queries.

These condition predicates can be evaluated using the temporal hash filters by the following steps:

- express the condition on the level of hash values. Shift the hash value to a common referencing instant, if necessary.

- Check the specific condition. The result is usually three-valued:

  `1` : the condition is fulfilled.
  `0` : the condition is not fulfilled.
  `?` : the condition cannot be decided using the hash values.

This can be used to realize usual hash-based query processing strategies with a probe and a test phase as discussed in Example 2. It turns out that `=`, `OVERLAPS`, and `CONTAINS` can be solved by the following strategy:

- compare the hash values bitwise. The bit operation depends on the query type and the index type and is three-valued:
  `1` : this part of the history fulfills the condition.
  `0` : this part of the history does not fulfill the condition.
  `?` : the condition cannot be decided for this part of the history using the hash values. The details about this step are given in Figure 5.

- combine the bitwise results by a three-valued operator (`and` for `=` and `contains`, `or` for `overlaps` as given in Figure 2) and interprete the three-valued result as mentioned before.

dense: =		
	0	1
0	?	0
1	0	1

sparse: =		
	0	1
0	1	0
1	0	?

dense: OVERLAPS		
	0	1
0	?	?
1	?	1

sparse: OVERLAPS		
	0	1
0	0	0
1	0	?

dense: CONTAINS		
	0	1
0	?	0
1	1	1

sparse: CONTAINS		
	0	1
0	1	0
1	1	?

Figure 5: Bitwise operations on the hash values for temporal comparisons

(8-bit,dense-log,1 year)	
object	*hash value*
Peter	`[01011110]`$_{1996}$
Paul	`[01110110]`$_{1995}$
Mary	`[00011111]`$_{1997}$
Doris	`[00010011]`$_{1990}$

(8-bit,sparse-log,1 year)	
object	*hash value*
Peter	`[11111110]`$_{1996}$
Paul	`[11110110]`$_{1995}$
Mary	`[00111111]`$_{1997}$
Doris	`[00010011]`$_{1990}$

Figure 6: The hash values for employment

Example 4 Figure 6 shows an encoding of the employment information as given in Figure 3 into dense and sparse hash values. We are interested in the following query:

```
select e2.* from Employees e1, Employees e2
where e1.name='Peter' and valid(e2) CONTAINS valid(e1)
```

which selects all employees who were at least employed when Peter was employed. Figure 7 shows the steps of the evaluation for each object. At first, the hash values are shifted. Because the referencing instant of o_3 is beyond the referencing instant of o_1, we have to shift o_1 in this case. The fourth column shows the bit-wise comparison results and the fifth column the result. In this example, dense and sparse hash values have the same selectivity. An alternative is a kind of preprocessing where all values are shifted to the highest reference instant which is relevant for the specific query. This scenario is described in Figure 8.

A similar evaluation schema for the query

```
select e2.* from Employee e1, Employee e2
where e1.name='Peter' and valid(e2) OVERLAPS valid(e1)
```

which selects all employees who have at least one common chronon with Peter, is given in Figure 9. Here the selectivity of the dense index is better, because two objects are selected and the other two have to be checked, while the sparse index is no help for this query. □

using dense hash values				
object	*hash value*	*Peter's hash value*	*comparison*	*result* (`and`)
Peter	$[01011110]_{1996}$	$[01011110]_{1996}$	`[?1?1111?]`	?
Paul	$[01000000]_{1996}$	$[01011110]_{1996}$	`[?1?0000?]`	0
Mary	$[00011111]_{1997}$	$[00100000]_{1997}$	`[??011111]`	0
Doris	$[00000000]_{1996}$	$[01011110]_{1996}$	`[?0?0000?]`	0

using sparse hash values				
object	*hash value*	*Peter's hash value*	*comparison*	*result* (`and`)
Peter	$[11111110]_{1996}$	$[11111110]_{1996}$	`[???????1]`	?
Paul	$[11110000]_{1996}$	$[11111110]_{1996}$	`[????0001]`	0
Mary	$[00111111]_{1997}$	$[11110000]_{1997}$	`[00??1111]`	0
Doris	$[10000000]_{1996}$	$[11111110]_{1996}$	`[?000000?]`	0

Figure 7: Evaluation of the `CONTAINS` query

using dense hash values: shifting to 1997				
object	*hash value*	*Peter's hash value*	*comparison*	*result* (`and`)
Peter	$[00100000]_{1997}$	$[00100000]_{1997}$	`[????????]`	?
Paul	$[00000000]_{1997}$	$[00100000]_{1997}$	`[??0?????]`	0
Mary	$[00011111]_{1997}$	$[00100000]_{1997}$	`[??011111]`	0
Doris	$[00000000]_{1997}$	$[00100000]_{1997}$	`[??0?????]`	0

using sparse hash values: shifting to 1997				
object	*hash value*	*Peter's hash value*	*comparison*	*result* (`and`)
Peter	$[11110000]_{1997}$	$[11110000]_{1997}$	`[????1111]`	?
Paul	$[11000000]_{1997}$	$[11110000]_{1997}$	`[??001111]`	0
Mary	$[00111111]_{1997}$	$[11110000]_{1997}$	`[00111111]`	0
Doris	$[10000000]_{1997}$	$[11110000]_{1997}$	`[?0001111]`	0

Figure 8: An alternative evaluation of the `CONTAINS` query

using dense hash values				
object	*hash value*	*Peter's hash value*	*comparison*	*result* (or)
Peter	$[01011110]_{1996}$	$[01011110]_{1996}$	[?1?1111?]	1
Paul	$[01000000]_{1996}$	$[01011110]_{1996}$	[?1??????]	1
Mary	$[00011111]_{1997}$	$[00100000]_{1997}$	[????????]	?
Doris	$[00000000]_{1996}$	$[01011110]_{1996}$	[????????]	?

using sparse hash values				
object	*hash value*	*Peter's hash value*	*comparison*	*result* (or)
Peter	$[11111110]_{1996}$	$[11111110]_{1996}$	[???????0]	?
Paul	$[11110000]_{1996}$	$[11111110]_{1996}$	[????0000]	?
Mary	$[00111111]_{1997}$	$[11110000]_{1997}$	[00??0000]	?
Doris	$[10000000]_{1996}$	$[11111110]_{1996}$	[?0000000]	?

Figure 9: Evaluation of the `OVERLAPS` query

A different evaluation schema is necessary to evaluate `PRECEDES` and `MEETS` predicates, because these operators are based on the order of the instants.

We introduce the following notions for a hash value h:

- $hbit_1(h)$ is the highest bit of h set to `1`.
- $lbit_1(h)$ is the lowest bit of h set to `0`.

Now the predicates `PRECEDES` and `MEETS` can be decided using the hash values as summarized in Figure 10. Depending on the *query type* and the type of the hash value, a *test condition* has to be checked, and if it succeeds, the mentioned *action* has to be performed with the same semantics as before (`1`: the condition is fulfilled; `0`: the condition is not fulfilled; `?`: the condition cannot be decided using the hash values). Usually, the result is two-valued for each query type, only `PRECEDES` queries used with sparse hash values can result in any of the three possible results. The last column shows the result, if `X` or `Y` have no `1`-bits.

Example 5 Figure 11 shows the result of the four conditions

(1) `valid('Peter') MEETS valid(E)`
(2) `valid(E) MEETS valid('Peter')`
(3) `valid('Peter') PRECEDES valid(E)`
(4) `valid(E) PRECEDES valid('Peter')`

according to the given evaluation schema. □

As the examples show, the usefulness of the temporal hash values depends on different circumstances:

- The distribution of data over time: a wide-spread distribution of the data (many "0" in the hash values) yields better results. In our example, the data distribution is rather dense, i.e. there is a high degree of overlapping time intervals in the database. This significantly reduces the selectivity of the hash filters.

query type	*index type*	*test condition*	*action*	**0**?
`X PRECEDES Y`	dense	$hbit_1(\texttt{X}) \geq lbit_1(\texttt{Y})$	0	?
`X PRECEDES Y`	dense	$hbit_1(\texttt{X}) < lbit_1(\texttt{Y})$	?	?
`X PRECEDES Y`	sparse	$hbit_1(\texttt{X}) > lbit_1(\texttt{Y})$	0	0
`X PRECEDES Y`	sparse	$hbit_1(\texttt{X}) = lbit_1(\texttt{Y})$	?	0
`X PRECEDES Y`	sparse	$hbit_1(\texttt{X}) < lbit_1(\texttt{Y})$	1	0
`X MEETS Y`	dense	$hbit_1(\texttt{X}) \geq lbit_1(\texttt{Y})$	0	?
`X MEETS Y`	dense	$hbit_1(\texttt{X}) < lbit_1(\texttt{Y})$	?	?
`X MEETS Y`	sparse	$hbit_1(\texttt{X}) \in \{lbit_1(\texttt{Y}) - 1, lbit_1(\texttt{Y})\}$	?	0
`X MEETS Y`	sparse	$hbit_1(\texttt{X}) \notin \{lbit_1(\texttt{Y}) - 1, lbit_1(\texttt{Y})\}$	0	0

Figure 10: Evaluation schema for `PRECEDES` and `MEETS`

using dense hash values						
`E`	*hash value of* `E`	*Peter's hash value*	(1)	(2)	(3)	(4)
Peter	$\texttt{[01011110]}_{1996}$	$\texttt{[01011110]}_{1996}$	0	0	0	0
Paul	$\texttt{[01000000]}_{1996}$	$\texttt{[01011110]}_{1996}$	0	0	0	0
Mary	$\texttt{[00011111]}_{1997}$	$\texttt{[00100000]}_{1997}$	?	0	?	0
Doris	$\texttt{[00000000]}_{1996}$	$\texttt{[01011110]}_{1996}$	?	?	?	?

using sparse hash values						
`E`	*hash value of* `E`	*Peter's hash value*	(1)	(2)	(3)	(4)
Peter	$\texttt{[11111110]}_{1996}$	$\texttt{[11111110]}_{1996}$	0	0	0	0
Paul	$\texttt{[11110000]}_{1996}$	$\texttt{[11111110]}_{1996}$	0	0	0	0
Mary	$\texttt{[00111111]}_{1997}$	$\texttt{[11110000]}_{1997}$	0	0	0	0
Doris	$\texttt{[10000000]}_{1996}$	$\texttt{[11111110]}_{1996}$	0	?	0	?

Figure 11: Evaluation of the `MEETS` and `PRECEDES` queries

- As usual for hash values, temporal hash filters simplify the exactness of the temporal data, which complicates the query processing of some queries. For instance, there can be "invisible" active chronons in "`0`"-intervals of dense hash values, which restricts the usefulness of dense indices for `CONTAINS` queries. In an sparse index, a "`1`" only signals one active chronon. Thus, finding two objects with the same sparse hash value does not help much for the evaluation of `OVERLAPS` queries, where the bitwise results are combined by `or`.

- The shifting of hash values leads to many `0`s in the index, especially in the logarithmic dimension. This necessarily reduces the usefulness for some query types.

- the past-infinity bit behaves usually different than the others, because in many databases it will be `0` for dense indices and `1` for sparse indices, especially after shifting.

At last, we remark that the benefit of the temporal hash filters is higher than it may seem from some of the given examples because:

- in reality the length of the hash values will be between 32-128 bit, thus, the temporal information is more exactly described by the hash value and the past-infinity bit is not that important.
- in the paper we only discussed the details for the logarithmic dimension. In the linear dimension, usually bigger time intervals are supported and shifting only leads to a linear number of additional `0`s, which extends the expressiveness of the hash values.

4 Conclusion

In this paper we presented a technique how temporal information can be encoded in hash values. We gave several alternatives and analyzed its usefulness for several temporal query types. It turns out that the proposed hash values allow a very flexible description of temporal data, because the representation within the hash value is independent of the granularity of the temporal data in the database. Thus, an index based on such hash values can be optimized w.r.t. the data load and the temporal queries which have to be supported.

The presented framework leaves many ways of future work. At first, it seems interesting to analyze how further kinds of temporal queries can efficiently be supported by the proposed hash values. It has to be worked out how hash values of different granularities can be processed simultaneously. This is necessary when we support optimization of queries with indices of different granularities and dimensions. It seems that this can be tackled by allowing only granularities with fixed fractions of a "global" granularity, similar to the structure of log-structured hash values, but here many details are open.

Another open question bothers how temporal information with null values should be treated. In our approach nulls occured within the scan process of index values and indicated that a certain condition could not be checked by the index, which enforces a lookup to the primary data. Adding nulls to the temporal information and to the temporal hash values may lead to some difficulties, because now nulls are used with two different semantics.

Another part of our future work is the implementation of the proposed index structure in our extensible OODBMS CROQUE [5], where we add temporal support on top of an ODMG-like OO model and queries.

References

[1] J.A. Blakeley and N.L. Martin. Join index, materialized view, and hybrid-hash join: A performance analysis. In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 256–263, 1990.

[2] K. Bratbergsengen and K. Norvag. Improved and optimized partioning techniques in database query processing. In *British National Conference on Databases (BNCOD)*, pages 69–83, 1997.

[3] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 237–246, June 1993.

[4] R. Elmasri, G.T.J. Wuu, and V. Kouramajian. The Time Index and the Monotonic B$^+$-tree. *[14]*, pages 433–456, 1993.

[5] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl. Query evaluation in CROQUE – calculus and algebra coincide. In *British National Conference on Databases (BNCOD)*, pages 84–100, 1997.

[6] W. Kim, editor. *Modern Database Systems.* ACM Press, 1995.

[7] D. Lomet. A review of recent work on multi-attribute access methods. *ACM SIGMOD Record*, 21(4):56–63, 1992.

[8] R. Marek and E. Rahm. Tid hash joins. In *Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 42–49, 1994.

[9] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):410–417, 1995.

[10] B. Salzberg and V. Tsotras. A comparison of access methods for time evolving data. Technical report, Northeastern University, 1994.

[11] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[12] R. Snodgrass. Temporal object-oriented databases: A critical comparison. In *[6]*, pages 386–408, 1995.

[13] R. Snodgrass, editor. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

[14] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[15] V. Turau and H. Duchene. Equality testing for complex objects based on hashing. *Data and Knowledge Engineering*, 10:101–111, 1993.

[16] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, MD, 1988.

[17] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[18] J. Won and R. Elmasri. Representing retroactive and proactive versions in bi-temporal databases (2tdb). In *Proc. of the IEEE Intl. Conf. on Data Engineering*, pages 85–94, 1996.