Evaluating Bestmatch-Joins on Streaming Data

Alfons Kemper Bernhard Stegmaier

Fakultät für Mathematik und Informatik
Universität Passau
D-94030 Passau, Germany

⟨last name⟩@db.fmi.uni-passau.de
http://www.db.fmi.uni-passau.de/

MIP-0204
December, 2002

Abstract

The Internet constitutes a huge distributed information source. Data sources on the Internet are often inherently infinite, e.g., dynamically generated data streams, or very large. New expressive query operators are needed to generate "interesting" data combining these data sources. A common problem is finding best matching pairs of data objects given user-defined multi-dimensional criteria. Traditional techniques do not give satisfying results, because a single "best" pair cannot be determined, since diverse pairs, each being best in different aspects of the comparison, are interesting. We propose the novel class of *bestmatch-join (BMJ) operators* to solve this problem. Unfortunately, the BMJ-operators are inherently blocking (pipeline-breakers), such that, in their basic form, they are not applicable to streaming data or "infinite" data sources. To improve the quality of the result and to overcome this problem we propose the *constrained BMJ-operators*. The constraints in combination with physical properties of the data stream, i.e., being ordered according to a constrained join attribute, enable our new pipelined BMJ-algorithms, which are based on synchronously shifting windows over the data streams. Finally, we present the encouraging results of our experiments, which demonstrate the effectiveness of our approach.

1 Introduction

We are moving fast towards the so-called information society. The Internet has become a huge data source which enables users to globally exchange and query data provided all over the world. Often, complex queries have to be processed to retrieve exactly the information the users are interested in from this huge volume of "raw" data. A common problem is, for instance, finding best matching pairs of data objects provided by different data sources given user-defined criteria. This is done by comparing the data objects on multiple dimensions (attributes), what leads to a partial order on the pairs of data objects. Because partial orders naturally do not have a single minimum, diverse pairs, each being best in different aspects of the comparison, are interesting. Hence, traditional techniques based on determining a single "best" pair fail to produce satisfying results.

We propose the novel class of *BMJ-operators* to solve this problem. Nearest neighbor operators or closest-point algorithms use an overall distance between two data objects, which is defined using all dimensions of the data objects, to determine closest pairs of data objects. In contrast, our new BMJ considers each dimension individually. The quality of the match of a pair of data objects given a special attribute is determined by user-defined functions, e.g., the distance, set inclusion, etc. Hence, the BMJ computes the best matching pairs of data objects having a maximum similarity on each individual join attribute. The result contains those pairs for which no better matching pair exists.

Unfortunately, these basic BMJ-operators are inherently blocking operators (pipeline breakers). Thus, they are not applicable on streaming data, because all input data has to be processed before any results can be delivered. Further, less interesting pairs matching very well in one but worse in all other dimensions may be contained in their result. Therefore, we introduce reasonable restrictions to the basic bestmatch-join operators to improve their results in common application scenarios. These constraints in combination with physical properties of the data streams, i.e., being ordered according to a constrained join attribute, enable a pipelining execution. We propose an algorithm for computing BMJ-operators based on synchronously shifting data windows that allows processing of infinite data streams using limited storage capacity. Because of the pipelined execution, this algorithm is suitable for application in modern stream processing query engines.

1.1 Application Scenarios

The BMJ-operation proposed in this paper for computing best matching pairs of two data sources has a wide possible field of application.

A first application scenario is assembling a composite part out of two base parts *part1* and *part2*. Different suppliers provide different models of these two parts. Since we want to build the "perfect" composite part, we are only interested in those combinations of models of the two parts which, e.g., have the lowest cost, minimum overall tolerance, lowest weight, and are provided by high quality providers. A single best composite part probably cannot be determined, because the cheapest composite part cannot be compared to a more expensive one with a better tolerance and so on.

As another application scenario consider a recruitment agency. The task of such an agency might be to find all the best job seekers for each given project advertised as a post.

Whether a job seeker is suitable for a given project is determined upon attributes such as the grade, professional qualifications, the distance of the job seeker to the company, and the experience of the job seeker. Again, comparing all pairs of open projects and job seekers by means of a single rating function does not deliver satisfying results, because for a given project a job seeker with a better grade might be of interest as well as a job seeker with a worse grade but living closer to the company.

Our BMJ-operator computes in both application scenarios all best and incomparable pairs.

Running Example
In this paper we use as a running example the matching of different sensor data provided as data streams. Consider a weather service which determines the probability of rain at certain locations. Therefore, it shall need the temperature and humidity at these locations. We assume that it is too expensive for the weather service to establish their own meteorological stations and that there are both types of sensors available, which are independently spread over the observed region. Further on, it is not necessary for a specific location to accommodate both types of sensors. The measuring data of all sensors are multiplexed by a suitable infrastructure, e.g., as proposed in [YG02, MF02], into two data sources; one data source contains all temperature data and the other all humidity data. Every data object of these data sources contains the sensor reading, the measurement time, and the (two-dimensional) location of the sensor. To be able to estimate the probability of rain, the weather agency has to find pairs of temperature and humidity measuring data, which are both locally and temporally close together. Additionally, the pairs must satisfy the following requirements to be usable for the estimation:

- The time between the measurement of the temperature and the humidity must not exceed 10 minutes.
- The distance between the sensors for temperature and humidity must not exceed 100 m in each dimension.

Since the individual sensors most likely deliver their measurements with different update rates, it is not possible to make a static correlation of sensors to be used as pairs. In the remainder of the paper we will show how to solve this problem of combining suitable sensor data using BMJ-operators.

1.2 Related Work

Outside the database field, some work was done to solve the problem of finding the best data objects of a single set according to multi-dimensional criteria. In the field of computational geometry [KLP75, PS85] describe the maximum vector problem and introduce several algorithms to solve this problem. [SU00] gives a detailed overview on different aspects of the closest-point and nearest neighbor problems. These algorithms are all main-memory techniques. In the database context, the authors of [BKS01] propose the so-called skyline operator for databases and study different implementation variants. In [TEO01] and [KRR02] algorithms for progressively computing the skyline are proposed.

Also, a lot of research was done in the related field of nearest neighbor queries, e.g., [RKV95, HAK00], and similarity joins, e.g., [BK01]. [Kie02] presents preference queries and a preference algebra using partial orders.

Much research is going on in the field of processing data streams. In [BBD+02] requirements and challenges for stream processing query engines are presented. The Tukwila project for efficiently processing streaming XML data using adaptive query processing techniques is introduced in [ILW+00]. Different architectures for query processing over sensor data streams are proposed in [YG02, MF02]. The authors of [CCC+02] present a system architecture, a stream oriented set of operators, and optimization approaches for monitoring data streams. The focus of [CF02] is to process both continuous and ad-hoc queries on old and new data of data streams by treating data and queries symmetrically. In [RH02] a query processing architecture continually generating partial results of a query is presented.

1.3 Organization of the Paper

The remainder of the paper is outlined as follows. In Section 2 the class of BMJ-operators is formally defined, simple evaluation methods are presented, and the constrained BMJ-operators are introduced. The window-based algorithm, its optimizations, and special application scenarios dealing with time-stamped data streams are shown in Section 3. Experiments and their results are presented in Section 4 and further conclusions are discussed in Section 5.

2 Definition of the Bestmatch-Join Variants

In this section we give a formal definition of the family of BMJ-operators. To simplify the notation we assume that the inputs of the BMJ-operators are two relational tables R and S with the following schema:

$$R : \{[x_1, \ldots, x_n, y_1, \ldots, y_d]\},\ S : \{[y_1, \ldots, y_d, z_1, \ldots, z_m]\}$$

With $r \in R$ and $s \in S$ the BMJ-operators generate pairs of tuples $(r \times s)$. As mentioned in the introduction, these pairs shall match best according to special attributes, which are denoted as *join attributes*. The attributes $y_1, \ldots, y_d$ of R and S are used as join attributes. The attributes $x_1, \ldots, x_n$ and $z_1, \ldots z_m$ contain additional data and do not participate in the computation of best pairs. The terms "join attributes" and "dimensions" are treated as synonyms.

2.1 Comparing Pairs Using Partial Orders

For computing the best matching pairs of tuples all different pairs $(r \times s)$ and $(r' \times s')$ of $R \times S$ have to be compared. Thus, for every join attribute y_i ($i \in \{1, \ldots, d\}$) an order $\leq_{y_i}$—or at least a partial order—has to be defined on the elements of $R \times S$. $(r \times s) \leq_{y_i} (r' \times s')$ denotes the situation that $(r \times s)$ matches better than $(r' \times s')$ according to y_i. These orders represent the user-defined criteria. If the join attributes are of numerical type, an

order could, e.g., be defined using the minimum distance of the join attributes (denoted as *minAttrDist*):

$$(r \times s) \leq_{y_i}^{minAttrDist} (r' \times s') \Leftrightarrow |r.y_i - s.y_i| \leq |r'.y_i - s'.y_i|.$$

An example for an order based on the comparison of set-valued attributes is, for instance, the order *subset*:

$$(r \times s) \leq_{y_i}^{subset} (r' \times s') \Leftrightarrow (r'.y_i \cap s'.y_i) \subseteq (r.y_i \cap s.y_i).$$

Which order shall be used on a specific join attribute depends completely on the intended application. Hence, these orders $\leq_{y_i}$ may be provided as 'black-boxes', e.g., user-defined functions, which get two pairs of tuples and return *true* in case of $\leq_{y_i}$, *false* in case of $>_{y_i}$, and *null* otherwise.

Using the individual orders on each join attribute a single partial order $\prec_{y_1,\ldots,y_d}$ on the elements of $R \times S$ can be constructed as follows:

$$(r \times s) \prec_{y_1,\ldots,y_d} (r' \times s') \Leftrightarrow (r \times s) \leq_{y_1} (r' \times s') \ \wedge \ \ldots \ \wedge (r \times s) \leq_{y_d} (r' \times s') \wedge (r \times s) \neq (r' \times s').$$

The in-equality predicate in the last clause of this partial order only compares the two tuples on the join attributes. $(r \times s) \prec_{y_1,\ldots,y_d} (r' \times s')$ denotes the situation that the tuples r and s match better than r' and s' according to all join attributes; this situation is called $(r \times s)$ *dominates* $(r' \times s')$. Since multiple dimensions are involved in the comparison, two pairs may be incomparable, too.

We distinguish two types of orders on the individual join attributes. The first type of orders—like *minAttrDist*—can be reduced to the order of real numbers using a function $f_i : dom(y_i) \times dom(y_i) \to \mathbb{R}$. With such a function an order $\leq_{y_i}$ can be written as

$$(r \times s) \leq_{y_i} (r' \times s') \Leftrightarrow f_i(r.y_i, s.y_i) \leq f_i(r'.y_i, s'.y_i).$$

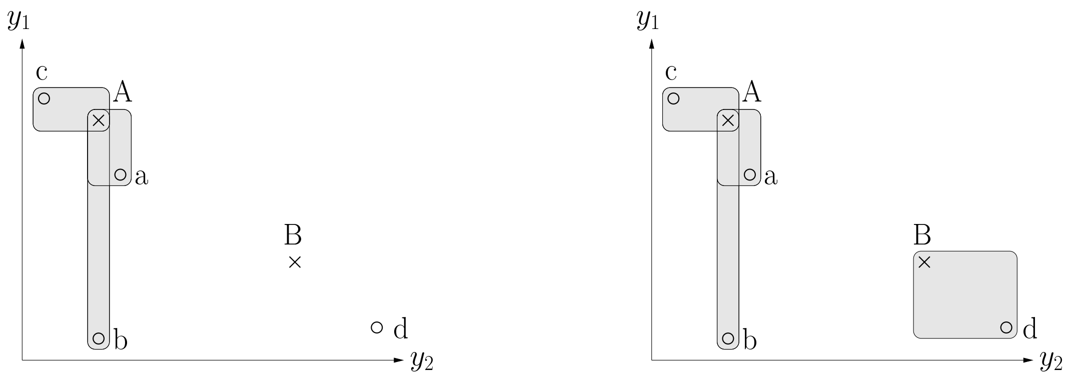
For instance, *minAttrDist* is defined by $f_i(a, b) := |a - b|$. The second type of orders are arbitrary partial orders, e.g., the order *subset*. The methods presented in this paper are applicable on both types. In real-world applications mainly orders of the first type are used and hence we focus on this type in the remainder of the paper.

2.2 The Bestmatch-Join Operators

Using a partial order to compare elements of $R \times S$ the family of BMJ-operators can be defined. The task of the BMJ-operator is to find all best matching elements of $R \times S$ according to a given partial order $\prec_{y_1,\ldots,y_d}$. That is, all elements of $R \times S$ have to be found which are not dominated by any other pair in $R \times S$. With the above notations the BMJ of R and S, symbolized by $\bowtie_{y_1,\ldots,y_d}$, can be defined as

$$R \bowtie_{y_1,\ldots,y_d} S := \left\{ (r \times s) \in (R \times S) \;\middle|\; \neg\exists (r' \times s') \in (R \times S) : (r' \times s') \prec_{y_1,\ldots,y_d} (r \times s) \right\}.$$

Since $\prec_{y_1,\ldots,y_d}$ is a partial order, not only one minimal pair exists in $R \times S$. There may be several minima which are all incomparable against each other. The BMJ computes all of



(a) BMJ (b) left-outer-BMJ

Figure 1: Examples of Bestmatch-Join Variants

these minimal pairs. An alternative notation of the BMJ-operator using a partial order $\prec$ is $\bowtie_{\prec}$. The BMJ-operator is not associative, i.e.

$$(R \bowtie_{\prec} S) \bowtie_{\prec'} T \;\neq\; R \bowtie_{\prec} (S \bowtie_{\prec'} T).$$

It is important to note that the BMJ-operator is different from the matching problem known in the context of graph theory. This matching problem is defined as finding a maximal subset of the edges—representing preferences for pairs of nodes—of a graph such that no edges of this subset share a single node as start resp. end point. In contrast to that, our BMJ-operator computes the overall best matching pairs of objects being incomparable against each other.

Another interesting variant of the BMJ-operator is the left-outer-BMJ. It does not only compute the overall best matching pairs but produces all best matching pairs for each individual tuple of the left input. It can be formally defined as

$$R \mathbin{⟕}_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \;\big|\; \neg\exists (r \times s') \in (R \times S) : (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)\big\}.$$

Of course, the right-outer-BMJ is defined analogously. A last variant of the basic BMJ is the full-outer-BMJ. It computes for each left *and* right input tuple its best pairs. It is formally defined as

$$\begin{aligned} R \mathbin{⟗}_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \;\big|\; & (\neg\exists (r \times s') \in (R \times S) : \; (r \times s') \prec_{y_1,\ldots,y_d} (r \times s)) \vee \\ & (\neg\exists (r' \times s) \in (R \times S) : \; (r' \times s) \prec_{y_1,\ldots,y_d} (r \times s))\big\}. \end{aligned}$$

Figure 1 depicts on abstract examples how the BMJ and the left-outer-BMJ work. It shows two datasets of two-dimensional points, which are symbolized by crosses (representing a relation R) and circles (representing a relation S). The two dimensions are used as join attributes with *minAttrDist* as order on each dimension. The three pairs marked by boxes in Figure 1 (a) are the result of the BMJ $R \bowtie_{y_1,y_2} S$. These pairs are incomparable against each other, because the join pair (A, a) is better than (A, c) with respect to join attribute y_2 and (A, c) is better than (A, a) with respect to join attribute y_1. Similar thoughts hold for (A, b). All other pairs are dominated by one of these three

pairs, e.g., (B, d) is dominated by (A, a), because the distances of the points of both pairs with respect to join attribute y_1 are equal, but (A, a) is better than (B, d) with respect to join attribute y_2. This result can be achieved by, for instance, executing the SQL query

```
select    *
from      R as r, S as s
where     not exists (
  select      *
  from        R as r', S as s'
  where       abs(r'.y1 - s'.y1) <= abs(r.y1 - s.y1) and
              abs(r'.y2 - s'.y2) <= abs(r.y2 - s.y2) and
              r.y1 <> r'.y1 and r.y2 <> r'.y2 and
              s.y1 <> s'.y1 and s.y2 <> s'.y2);
```

Figure 1 (b) shows the result of the left-outer-BMJ $R⟕_{y_1,y_2}S$ using the same inputs. For the first tuple A of R the same conditions hold as before. Hence, the same pairs (A, a), (A, c), and (A, b) are generated. Next, the data point B is examined. The pair (B, d) dominates all other pairs with B as the first point and thus it is contained in the result. There are no more tuples in R, so the result of the left-outer-BMJ contains the marked pairs. To compute this result the definition of the left-outer-BMJ can be analogously transformed into SQL as shown above for the BMJ. In order to specify BMJ-operators in SQL queries, we propose to extend SQL's **from** clause as follows:

```
select    ...
from      R [ ( left | right | full ) outer ] bestmatch join S
            on R.y1 order1 S.y1, R.y2 order2 S.y2, ...
where     ...
```

The join attributes $y_1, y_2, \ldots$ are specified after the **on**-keyword. "order1", "order2", ... denote the names of the user-defined comparison functions to be used as an order on the particular join attribute.

Obviously, there is a close relationship between the BMJ-operator and the skyline operator introduced in [BKS01]. The skyline of a single set of points is defined as those points which are not dominated by any other points of the set. Using the above defined partial order $\prec_{y_1,\ldots,y_d}$ the BMJ can be ascribed to the computation of a skyline of the set $R \times S$ as $R⟕_{y_1,\ldots,y_d}S = skyline_{\prec_{y_1,\ldots,y_d}}(R \times S)$. Therewith, a first approach for computing the BMJ in the materialized case—which is not the scope of this paper—is to apply existing skyline algorithms on $R \times S$. The outer-BMJ variants cannot be reduced to the computation of a single skyline on $R \times S$. Algorithm 1 shows the nested-loops approach for computing the left-outer-BMJ. Similarly, the other outer-BMJ variants can be computed.

2.3 Constrained Bestmatch-Joins

Looking at Figure 1 it can be observed that "extreme pairs" like (A, b) may be contained in the result of best matching pairs. This pair matches very well in one dimension (y_2) but very poorly in the other dimension(s). This behavior is not desirable for many

Algorithm 1 nested-loops left-outer-BMJ

Input: R, S, partial order $\prec_{y_1,\ldots,y_d}$
Output: $R \ltimes_{y_1,\ldots,y_d} S$

```
    for all r ∈ R do
2:    for all s ∈ S do
        dominated = false
4:      for all s' ∈ S do
          if (r × s') ≺_{y1,...,yd} (r × s) then
6:          dominated = true
          end if
8:      end for
        if ¬dominated then
10:       Output (r × s)
        end if
12:   end for
    end for
```

applications, because a single well-matching dimension might not balance the remaining poorly-matching dimensions. Therefore, we propose the approach that only pairs $(r \times s) \in R \times S$ are considered which do not exceed an individual maximum distance in each constrained dimension. This maximum distance on y_i for being considered as a possible pair is denoted as ϵ_i. The distance of two tuples r and s according to y_i is symbolized by $d_i(r, s)$. The situation that $d_i(r, s) \leq \epsilon_i$ for every $i \in \{1, \ldots, d\}$ is denominated as $d(r, s) \leq \epsilon$ (ϵ symbolizes the vector of the individual ϵ_i). So, only those $(r \times s) \in R \times S$ have to be considered as candidates for best matching pairs, for which $d(r, s) \leq \epsilon$ holds. Therewith, the constrained BMJ-operator, denoted as $\ltimes^{\epsilon}_{y_1,\ldots,y_d}$, is formally defined as

$$R \ltimes^{\epsilon}_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \;\big|\; d(r, s) \leq \epsilon \wedge \neg\exists (r' \times s') \in (R \times S) : (d(r', s') \leq \epsilon \wedge (r' \times s') \prec_{y_1,\ldots,y_d} (r \times s))\big\}.$$

Analogously, the constrained left-outer-BMJ operator $\ltimes^{\epsilon}_{y_1,\ldots,y_d}$ is defined as

$$R \ltimes^{\epsilon}_{y_1,\ldots,y_d} S := \big\{(r \times s) \in (R \times S) \;\big|\; d(r, s) \leq \epsilon \wedge \neg\exists (r \times s') \in (R \times S) : (d(r, s') \leq \epsilon \wedge (r \times s') \prec_{y_1,\ldots,y_d} (r \times s))\big\}.$$

Of course, the constrained right-outer-BMJ and the constrained full-outer-BMJ are defined similarly. Constrained BMJ-operators are expressed in SQL using our new notation of the BMJ operators; the maximum distance constraints are simply added to the **where**-clause of the query.

Therewith, the problem of the weather agency introduced in Section 1.1 can be solved on materialized data. The temperature resp. humidity data shall be contained in the relations *Temp* resp. *Hum*. Each relation consists of the attributes t, x, y, and v storing the time of measurement, the x- and y-coordinates of the sensor, and the sensor reading. The attributes t, x, and y are used as join attributes. For comparing pairs of sensor data a partial order $\prec_{t,x,y}$ is constructed using *minAttrDist* on each dimension as an order. The

given restrictions can be employed by setting $\epsilon_t = 10\text{min}$, $\epsilon_x = \epsilon_y = 100\text{m}$. Therewith, the expected pairs of sensor data can be obtained by evaluating the query

$$Temp \; \tilde{\bowtie}^{\epsilon}_{t,x,y} \; Hum.$$

The result contains pairs of sensor data consisting of the best matching humidity measurements for each temperature sensor. If for a certain temperature sensor no humidity measurement satisfying the specified requirements is available, it will not be contained in the result. To compute the results Algorithm 1 has to be extended to compare only those pairs which fulfill the given maximum distance constraints on each dimension. Another possibility is to transform the query into standard SQL as demonstrated in the previous section; the constraints are simply added to the **where**-clauses of the query and the sub-query.

3 Evaluating Bestmatch-Joins on Data Streams

Naturally, Internet data sources provide their data as data streams. These data streams are either infinite or very huge. In both cases it is not feasible to store them locally and process the materialized data. Thus, methods for evaluating the constrained left-outer-BMJ operators on streaming data are needed.

In this paper a data stream R is considered to be an ordered sequence of tuples $< r_1, r_2, \cdots >$. The tuples can only be read one-by-one. That is, the tuples are read in the sequence $r_1, r_2, \ldots$ If r_i has been read, the tuples r_j with $j \leq i$ cannot be accessed anymore. The problems arising from computing blocking operators, like the BMJ operators, on such data streams are discussed in the next section.

3.1 Bestmatch-Joins and Data Streams

To clarify the problems of query evaluation over data streams we first consider the standard join operator. To avoid a blocking execution, utilizing suitable join algorithms, e.g., the double pipelined hash join, enables a pipelined execution. That is, for every read input tuple the join mates can be computed (on the basis of the inputs known up to that time) and delivered. To be able to compute the correct result, i.e., no join mates are missed, all consumed inputs have to be buffered. Hence, working on infinite data streams a pipelined execution is possible, but the size of the state of a join operator grows unboundedly.

In contrast to that, the BMJ-operators are—in their basic form—inherently blocking. This is due to the fact that the best matching pairs of two (finite) data streams might be the last two tuples of the data streams. Therefore, the correct result cannot be delivered before the entire streams have been processed. If the BMJ-operators are forced to deliver any results before, these results are approximative intermediate results. That is, they are correct with respect to the data processed up to that time, but some pairs might get invalidated by new (better matching) pairs computed at a later time. Hence, dealing with infinite data streams only approximative results can be continuously propagated. As for the conventional join, in this case the state of the BMJ-operators has an unlimited size, because all input data has to be buffered.

To avoid the computation of uninteresting pairs we introduced the constrained BMJ-operators. On their own, the maximum distance constraints for pairs being interesting do neither bound the size of the state of the constrained BMJ-operators, nor enable a pipelined execution, because a best matching pair might consist of tuples being delivered at the beginning of the first and at the end of the other data stream. Hence, still the entire streams have to be processed to compute precise results. But, exploiting physical properties of the data stream, i.e., being sorted on one join attribute, the size of the state of the BMJ-operators can be bounded: For any tuple $r \in R$ only those tuples $s \in S$ have to be buffered which are contained in the contiguous interval $r.y_1 - \epsilon_1 \leq s.y_1 \leq r.y_1 + \epsilon_1$ of the data stream S (assuming that the data streams are ordered according to y_1) and vice versa. In case of the outer-BMJ-operators the combination of constraints and physical properties even enables a pipelined execution. For instance, for any tuple r of the left data stream the constrained left-outer-BMJ is able to determine, if all interesting join mates of the right data stream are processed yet. This is the case, if all tuples s of the above mentioned contiguous interval of the right data stream have been read. Afterwards, the precise results for this r can be delivered without processing the remainder of the streams and therefore a pipelined execution is achieved. This applies analogously for all outer-BMJ-operators.

On the whole, in combination with particular physical properties of the (infinite) data streams we are able to compute the constrained outer-BMJ-operators in a pipelined manner utilizing size-limited buffers. The constrained BMJ-operator remains to be inherently blocking, but we are able to compute and deliver approximative intermediate results, which converge to the precise result in case of finite data streams, using a state of limited size. In case of infinite data streams these approximative results are the best we can expect, because a "final" result does not exist.

3.2 The Window-Based Approach

In the previous section we have argued that it is possible to compute the constrained BMJ-operators on data streams being sorted according to a join attribute in a non-blocking fashion. Since the outer-BMJ-operators are more relevant for real-world applications, we focus on the constrained left-outer-BMJ as an example in the remainder of the paper. The restriction that the data streams have to be sorted is not out of touch with reality, because many data streams, e.g., sensor data, stock quotes, etc., will naturally be delivered sorted according to a particular dimension, e.g., the time. Furthermore, we assume that many data sources are somewhat "intelligent", so that they can be told to deliver their data streams sorted.

Utilizing this premise our window-based algorithm MatWin for the constrained left-outer-BMJ works as follows. In the remainder we assume that both data streams are sorted according to y_1, that all distances of pairs $d_i(r, s)$ are normalized to the interval $[0; 1]$, and therewith all ϵ_i are of the interval $]0; 1]$. For the input data streams R and S two data windows W_R and W_S are maintained and synchronously shifted over the data streams. Because each data stream is sorted according to y_1, it can be assured that all tuples belonging to a certain interval with respect to y_1 have been read from the data streams. The upper and lower bounds of this interval are denoted with $maxRS$

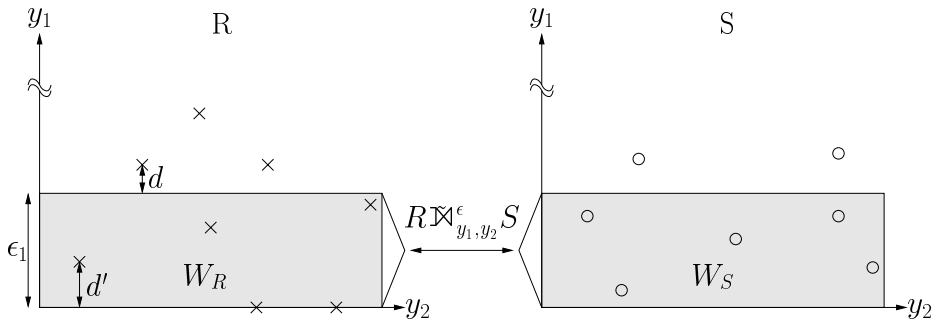


Figure 2: Initial Computation Step

resp. $minRS$. $maxRS - minRS$ is denominated as the height of the data windows. $min_{y_1}(W_R)$ denotes the minimal y_1-value of all tuples contained in W_R. The data windows are maintained such that their height equals ϵ_1. Hence, the contents of the data windows given $minRS$ and $maxRS$ are

$$W_R = \{r_i \in R \mid minRS \leq r_i.y_1 \leq maxRS\},$$
$$W_S = \{s_j \in S \mid minRS \leq s_j.y_1 \leq maxRS\}.$$

During the computation the data windows are moved along the dimension y_1 by increasing $minRS$ and $maxRS$. That is, all tuples $r_i \in W_R$ with $r_i.y_1 < minRS'$ are deleted from W_R and new tuples are read from R and stored in W_R as long as $r_i.y_1 \leq maxRS'$, with $minRS'$ resp. $maxRS'$ being the new lower resp. upper bound of the data windows. Analogously, W_S is maintained. Therewith, it can be assured that for a specific $r_i \in R$ all needed join mates $s_j \in S$ are contained in W_S at the same time: The tuple r_i enters W_R, if $r_i.y_1 = maxRS$. At this time W_S contains all $s_j \in S$, for which

$$r_i.y_1 - s_j.y_1 \leq \epsilon_1$$

holds. Analogously, r_i is deleted from W_R, if $r_i.y_1 < minRS$. Before that, W_S contained all $s_j \in S$ with

$$s_j.y_1 - r_i.y_1 \leq \epsilon_1.$$

Hence, all interesting join mates are contained in W_R and W_S while the data windows are shifted over the data streams. Because only a fixed interval with regard to y_1 of the data streams is contained in the data windows, they have a limited maximum size, which depends on the distribution of the values of the join attribute y_1. However, the maximum size of the data windows might be large, so they are stored on secondary storage in pages of a fixed size. In the next sections, efficient materializing and I/O-scheduling strategies for computing the constrained BMJ-operators on these materialized data windows are presented.

The Figures 2 to 4 show how the MatWin algorithm works. In these figures two join attributes y_1, y_2 are used. The data streams are sorted according to y_1 and therefore the tuples are delivered from the bottom up. Figure 2 shows the initial computation step. The first data windows W_R and W_S are read (with $minRS = 0$, $maxRS = \epsilon_1$), materialized, and the left-outer-BMJ is computed on all pairs of $W_R \times W_S$ using the nested-loops

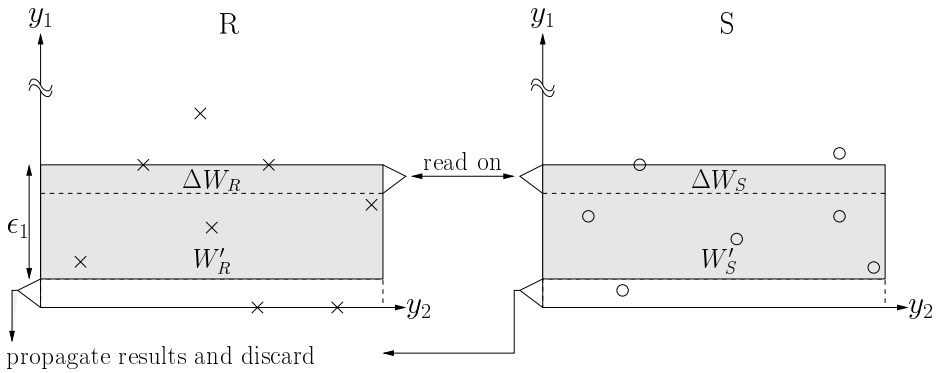


Figure 3: Moving Data Windows

algorithm (Algorithm 1). Not all needed pairs are processed at this time, so the computed best pairs are held in memory as an intermediate result O. Next, the data windows have to be moved. At a first step, all tuples $r_i \in W_R$ are removed, for which $r_i.y_1 = minRS$ holds. To calculate the new bounds of the data windows the two distances d and d' have to be considered. d is the distance with regard to y_1 of the first tuple outside the current data window to $maxRS$. d' is defined as $d' = min_{y_1}(W_R) - minRS$. Therewith, $minRS$ and $maxRS$ are increased by $min(d, d')$ and the data windows are moved as described above. The new situation is depicted in Figure 3. The remaining parts of the data windows are denoted by W'_R resp. W'_S, the newly read parts are denominated as ΔW_R and ΔW_S. Since all discarded tuples of the data windows have been completely processed, all best matching pairs they are involved in can be propagated to the output data stream and deleted from O. The remaining current best pairs have to be updated with the pairs of $(W'_R \cup \Delta W_R) \times (W'_S \cup \Delta W_S)$. This is done by comparing the current best pairs O to pairs of the new data windows using $\prec_{y_1,\ldots,y_d}$. If a pair in O is dominated by a new pair, this new pair is inserted into O and the dominated pair is deleted. $(W'_R \cup \Delta W_R) \times (W'_S \cup \Delta W_S)$ can be rewritten as

$$(W'_R \times W'_S) \cup (W'_R \times \Delta W_S) \cup (\Delta W_R \times (W'_S \cup \Delta W_S)).$$

We can see that not all pairs of the new data windows have to be compared with O, because the pairs of $W'_R \times W'_S$ have already been processed in the previous step. Hence, only the pairs of $W'_R \times \Delta W_S$ and $\Delta W_R \times (W'_S \cup \Delta W_S)$ have to be compared with all pairs contained in O. This situation is depicted in Figure 4. The movement of the data windows, the propagation of finished results, and updating O continues until no more data can be read from the data streams.

The details of the MatWin algorithm are shown in Algorithm 2. The method MaterializeWindows() (line 9) reads tuples from the data streams and materializes the data windows onto secondary storage. Its details are presented in Section 3.3.3. The method Update(...) used in lines 11, 13, and 14 updates the current best pairs of the left-outer-BMJ by the given parts of the data windows using a nested-loops algorithm as described above. Of course, instead of the constrained left-outer-BMJ the other constrained BMJ-operators can be utilized in this method. In case of the constrained BMJ, the propagated pairs are approximative as described in the previous section. Thus, the MatWin algorithm

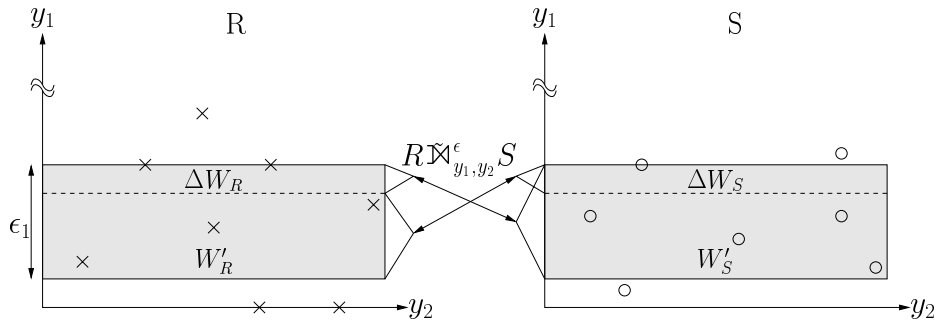


Figure 4: Updating Current Best Pairs

must additionally keep track of propagated pairs for being able to invalidate them at a later time.

3.3 I/O-Scheduling Using the ϵ-Grid-Order

For an efficient computation of the left-outer-BMJ on the materialized data windows using a limited amount of main memory we employ the Epsilon Grid Order developed in [BK01]. It was designed to efficiently compute the similarity join $\{(p \times q) \in (P \times Q) \mid ||p-q|| \leq D\}$ of two data sets P and Q given a maximum distance D of the pairs of points. To suit our needs, we extend this order to consider a maximum distance ϵ_i on each dimension instead of using a single distance D. We denote our extended version of the Epsilon Grid Order as ϵ-Grid-Order. Further, the scheduling algorithms of [BK01] exploiting the Epsilon Grid Order were presented for the similarity self-join; we adapted them to work on two inputs instead of one.

3.3.1 Definition and Properties of the ϵ-Grid-Order

First, we define the ϵ-Grid-Order and afterwards we derive an important property for utilizing it in the context of constrained BMJ computations. To keep it simple, we define the ϵ-Grid-Order and present its properties using d-dimensional vectors p and q. The dimensions of these vectors correspond to the join attributes in the database context. The ϵ-Grid-Order order is symbolized by $<_\epsilon$. The predicate $p <_\epsilon q$ is *true* if there exists a dimension i such that the following condition holds:

$$\left(\left\lfloor \frac{q_i}{\epsilon_i} \right\rfloor < \left\lfloor \frac{p_i}{\epsilon_i} \right\rfloor \right) \quad \wedge \quad \left(\left\lfloor \frac{q_j}{\epsilon_j} \right\rfloor = \left\lfloor \frac{p_j}{\epsilon_j} \right\rfloor \quad \forall j < i \right)$$

p_i resp. q_i denote the i-th dimension of the vectors p and q. Böhm et. al. show that the original Epsilon Grid Order is an irreflexive order. The proofs of the properties of an irreflexive order are similar for the ϵ-Grid-Order and therefore they are not carried out. The ϵ-Grid-Order divides the d-dimensional space into a grid. The edges of every cell of this grid have a length of ϵ_i in the dimension i. The tuples contained in each cell are equal with regard to $<_\epsilon$.

Algorithm 2 MatWin

Input: data streams R and S; $\epsilon_1, \ldots, \epsilon_d$; $\prec_{y_1,\ldots,y_d}$
Output: output data stream of $R \tilde{\bowtie}^{\epsilon}_{y_1,\ldots,y_d} S$

```
    /* initialization */
 2: O ← ∅; oldMaxRS ← 0
    W_R ← ∅; W_S ← ∅
 4: r ← R.next(); s ← S.next()
    minRS ← max(0, r.y_1 − ε_1)
 6: maxRS ← minRS + ε_1
    /* computation */
 8: while (r ≠ null) ∨ (W_R ≠ ∅) do
      MaterializeWindows()
10:   if oldMaxRS = 0 then /* new windows */
        Update(O, W_R, W_S, ≺_{y_1,...,y_d})
12:   else /* moved windows */
        Update(O, W'_R, ΔW_S, ≺_{y_1,...,y_d})
14:     Update(O, ΔW_R, W'_S ∪ ΔW_S, ≺_{y_1,...,y_d})
      end if
16:   /* update window bounds */
      oldMaxRS = maxRS
18:   W_R.delete({r' ∈ W_R | r'.y_1 = minRS})
      if (r = null) ∧ (W_R = ∅) then
20:     break
      else if (r = null) ∧ (W_R ≠ ∅) then
22:     minRS = min_{y_1}(W_R)
      else if (r ≠ null) ∧ (W_R ≠ ∅) then
24:     minRS = min(min_{y_1}(W_R), r.y_1 − ε_1)
      else /* (r ≠ null) ∧ (W_R = ∅) */
26:     minRS = r.y_1 − ε_1
        oldMaxRS = 0
28:   end if
      maxRS ← minRS + ε_1
30:   W_S.delete({s' ∈ W_S | s'.y_1 < minRS})
      output and delete {(r' × s') ∈ O | r'.y_1 < minRS}
32: end while
    output {(r × s) ∈ O}
```

The following properties of the ϵ-Grid-Order enable an efficient computation of constrained BMJ-operators. Let p, p', q be d-dimensional vectors. Given a fixed p, the following condition holds: If $q <_\epsilon p - (\epsilon_1, \ldots, \epsilon_d)$, there exists a dimension $i \in \{1, \ldots, d\}$ with $p_i - q_i > \epsilon_i$ and therewith

$$q \notin [p_1 - \epsilon_1] \times \cdots \times [p_d - \epsilon_d].$$

Additionally, for all p' with $p <_\epsilon p'$ exists a dimension $j \in \{1, \ldots, d\}$ with $p'_j - q_j > \epsilon_j$

and therewith

$$q \notin [p'_1 - \epsilon_1] \times \cdots \times [p'_d - \epsilon_d].$$

Altogether, a point q cannot be a join mate of p if $q <_\epsilon p - (\epsilon_1, \ldots, \epsilon_d)$. Again, the proof of this property is similar as presented for the original Epsilon Grid Order and hence it will not be done in this paper. Analogously, it can be derived that a point q cannot be a join mate of p if $p + (\epsilon_1, \ldots, \epsilon_d) <_\epsilon q$.

These two properties can be utilized for an efficient computation of $R \ltimes^{\epsilon}_{y_1,\ldots,y_d} S$ (and the other variants) as follows. We assume that S is sorted according to the ϵ-Grid-Order on the join attributes. Further, let

$$s - \epsilon := [s.y_1 - \epsilon_1, \ldots, s.y_d - \epsilon_d, s.z_1, \ldots, s.z_m]$$

and analogously $s + \epsilon$. For a specific $r \in R$ only those $s \in S$ are interesting join mates which are contained in the interval

$$[s - \epsilon, s + \epsilon]$$

of the according to $<_\epsilon$ sorted sequence of S. Hence, not all pairs $\{(r \times s) \mid s \in S\}$ have to be examined for computing the best matching pairs.

3.3.2 I/O-Scheduling

As mentioned before, the current data windows W_R and W_S are stored in pages on secondary storage by our MatWin algorithm. The pages of W_R and W_S are denoted as P_R^i resp. P_S^j, where i and j range between 1 and the maximum number of pages of W_R resp. W_S. To be able to exploit the benefits of the ϵ-Grid-Order these data windows have to be stored sorted according to $<_\epsilon$. That is, every page contains a sorted sequence of tuples. Further, the last tuple of a page P_R^i has to be less or equal than the first tuple of another page $P_R^{i'}$ with $i < i'$ (analogously for the pages of W_S). Two approaches for storing the tuples in that manner are presented in the next section. The pages of W_R and W_S have to be loaded into main memory to perform the computation of the constrained BMJ operators. For that purpose a buffer for m pages shall be available. The task of the scheduling algorithm is to load the pages of W_R and W_S into main memory for the computation of the constrained BMJ operators in such a way that the number of disk accesses is minimized.

Figure 5 depicts an exemplary assignment of tuples to pages with respect to the join attributes y_2 and y_3. The join attribute y_1 is omitted, because the height of the data windows is equal to ϵ_1. Hence, with regard to y_1 the tuples of the data windows are all equal or at most divided into two parts—if $minRS$ and $maxRS$ are not a multiple of ϵ_1—according to $<_\epsilon$. Therefore, to be able to present the assignment of tuples to pages in more detail we only show the join dimensions y_2 and y_3. For presentation purposes we assume in the remainder that the layout of pages of both data windows equals that of Figure 5. Of course, in reality the assignment of tuples to pages will not be equal for W_R and W_S.

To perform the update of the current best matching pairs O with new data windows the pages of W_R and W_S have to be loaded into the memory buffers and all interesting

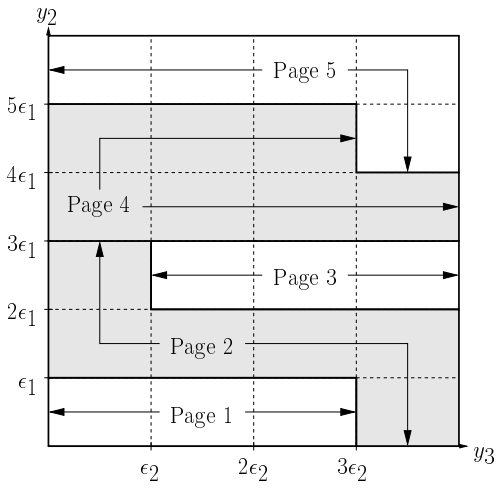


Figure 5: Pages in the Data Space

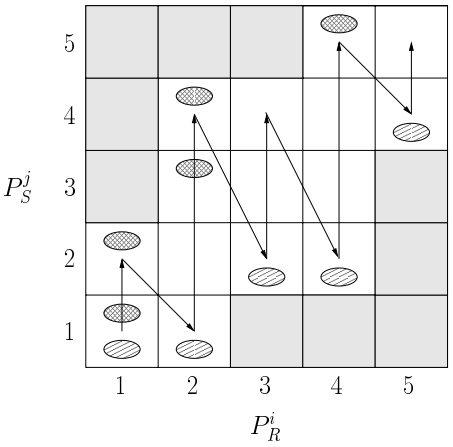


Figure 6: Pairs of Pages to be Processed

pairs have to be examined. Because of the observations of the previous section, not all pairs of pages have to be considered for computing the constrained BMJ. Figure 6 shows which pairs of pages have to be considered at the exemplary page layout of Figure 5. Each cell in the matrix stands for a pair of pages. Obviously, the pair (1,1), i.e. P_R^1 and P_S^1, has to be processed. The pair (1,2) has to be considered, because P_R^1 may contain tuples which have interesting join mates stored in P_S^2. The grey shaded cells need not to be considered. For instance, the pair (1,3) does not have to be processed, because P_S^3 cannot contain any interesting join mate s for any $r \in P_R^1$.

Now, all interesting pairs of pages have to be loaded into the main memory buffers causing a minimal number of disk accesses. Assuming $m = 4$ the naive column-by-column scheduling method is optimal. One buffer is reserved for storing the current page of W_R. The remaining $m - 1$ buffers are available for loading pages of W_S. If $m - 1$ pages of W_S are contained in the buffers and a new page of W_S has to be loaded, a page of W_S is discarded from memory using the LRU strategy. In the figure a disk access of a page of W_R is symbolized by a single hatched oval, a disk access of a page of W_S is symbolized by a double hatched oval. First, P_R^1 and P_S^1 are loaded and O is updated with all pairs of $P_R^1 \times P_S^1$. Next, P_S^2 is loaded and the pairs of $P_R^1 \times P_S^2$ are processed. No other pairs of pages in this column have to be processed. This can be determined by comparing the last and first tuples of the corresponding pages using $<_\epsilon$. The values of the join attributes of the first and last tuple of a page can be held in main memory to avoid additional disk accesses. Proceeding to the next column is done by discarding P_R^1 and loading P_R^2. This scheduling is performed until all pairs of pages have been processed. In the style of [BK01] this scheduling method is called *gallop mode*.

Figure 7 depicts another example using $m = 4$ and the gallop mode. Up to column 2 this method is optimal. Then, thrashing occurs, because the number of pages needed for storing the whole interesting interval of tuples exceeds the number of memory buffers. In this case we switch to the *crabstep mode* (according to [BK01]) as shown in Figure 8. After processing the pair (2,3), P_S^4 has to be loaded, but there is no available buffer. Hence, the loaded pages of W_S are pinned to the buffers. The pages $P_R^4, \ldots, P_R^5$ are

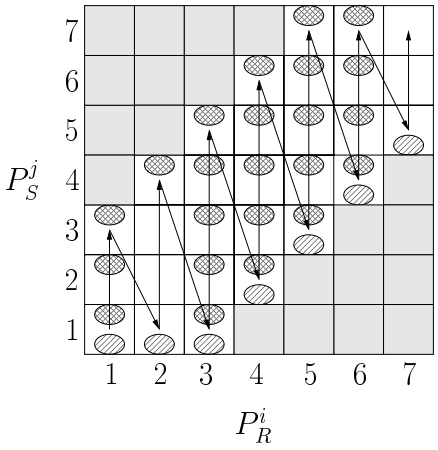


Figure 7: Gallop Mode: Thrashing

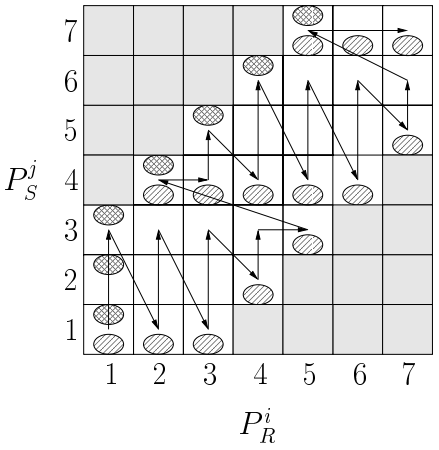


Figure 8: Crabstep Mode

consecutively loaded into the single buffer for pages of W_R and processed. After (5,3) has been computed, all buffers are freed and the scheduling is continued at (2,4) starting with the gallop mode and switching to the crabstep mode if needed. Therewith, the number of disk accesses can be greatly reduced—in this example from 30 to 21.

As explained in Section 3.2, the MatWin algorithm does not have to process all pairs of $W_R \times W_S$ but only parts of the whole data windows, e.g., $\Delta W_R \times W_S$. The tuples of these parts, e.g., ΔW_R, are spread over all pages of the data windows. Of course, while processing a pair of pages the tuples not contained in these interesting parts of the data windows are ignored. To efficiently process only the interesting parts of the pages the following indexing technique can be applied. A page contains an ordered sequence of cells of the ϵ-Grid-Order. In every cell, all tuples are equal according to $<_\epsilon$ and hence the sorting according to y_1 can be preserved in each cell. For every page the start tuples of the contained cells are stored. Therewith, the first tuple of all tuples of a page with a y_1-value greater than a given boundary can be determined by performing a binary search in each cell of the page. Analogously, all tuples of a page with an y_1-value lower than a given boundary are processed by proceeding sequentially through each cell until a tuple with a y_1-value greater than the limit is reached.

3.3.3 Materializing the Data Windows

Finally, we show how to materialize the current data windows W_R and W_S sorted according to $<_\epsilon$.

The common part of both methods is depicted in Algorithm 3. The memory buffers, which are used in the previous section for computing the constrained BMJ operators, are divided between the two inputs. For R resp. S m_R resp. m_S pages can be held in main memory ($m_R + m_S = m$). m_R and m_S may be dynamically adapted to improve the efficiency of the materialization, e.g., if W_R is completely materialized before W_S, the entire buffer is used to materialize W_S. The tuples of the data streams are read and stored in the corresponding buffer. If the buffer of a data window is full or the data window contains all necessary tuples, e.g., W_R is completely filled, if a tuple with $r.y_1 > maxRS$ has been read, the buffer is sorted (in memory) and written to disk using

Algorithm 3 MaterializeWindows

```
   while ((r.y_1 ≤ maxRS) ∧ (r ≠ null)) ∨
         ((s.y_1 ≤ maxRS) ∧ (s ≠ null)) do
2:   if (r.y_1 ≤ maxRS) ∧ (r ≠ null) then
       if |M_R| = m_R − 1 then
4:       Materialize(M_R); M_R ← ∅
       end if
6:     M_R.add(r); r ← R.next()
     end if
8:   if (s.y_1 ≤ maxRS) ∧ (s ≠ null) then
       if s.y_1 ≥ minRS then
10:      if |M_S| = m_S − 1 then
           Materialize(M_S); M_S ← ∅
12:      end if
         M_S.add(s)
14:    end if
       s ← S.next()
16:  end if
   end while
18: Materialize(M_R); Materialize(M_S)
```

the Materialize()-method (lines 4, 11, 18). This is repeated until both data windows are completely materialized. The Materialize()-method implements one of the following two strategies.

The first materializing approach works like external sorting. If the memory buffers of a data stream are full, the pages are sorted and written to secondary storage as a run. This is done until all necessary tuples of a data window have been read. Finally, all created runs and the old data window (W'_R resp. W'_S) of the previous step are merged to create the current sorted data window. This approach is denoted as *standard materialization*.

The second approach, denoted as *grid materialization*, utilizes the fact that the ϵ-Grid-Order divides the data space in cells. This grid is held in memory as an index. Every cell of the grid contains a list of pages. If the memory buffers of a data stream are full, the tuples are sorted in memory. Using the grid structure, all tuples belonging to a single cell are materialized by reading the last page of the list of pages of this cell and writing the tuples to it. If a page is full, a new one is appended to this list.

Both approaches have assets and drawbacks. The standard materialization has to create and merge the runs. Hence, all pages of the old data windows and the current runs are read and written. In contrast to that, the grid materialization has to read and write only the last existing page of each cell for inserting new tuples. Hence, the grid materialization will probably generate less disk accesses. But, the grid structure will grow rapidly with decreasing ϵ_i and an increasing number of dimensions. Additionally, the pages might be populated sparsely, because for each cell a page is reserved.

3.4 Exploiting Fuzzy Orders on Data Streams

The premise of the data streams being ordered is needed for the MatWin algorithm to be able to determine, whether for a given upper bound of the data windows all necessary tuples have been read. However, some data sources may not be able to produce a strictly ordered data stream but a "somewhat ordered" data stream. E.g., a sensor network collecting data from individual sensors might not be able to deliver all sensor readings as a single data stream ordered according to the time of measurement, because the connections of the individual sensors have variable latencies. Hence, on the whole the timestamps of the sensor readings increase, but some tuples may be slightly out of order. But, the sensor network might be able to assure that, e.g., all measurements up to $t - 5s$ are delivered at a time t. We denote data streams satisfying such additional properties as *fuzzy ordered data streams*. Such properties enable the MatWin algorithm to process fuzzy ordered data streams, because therewith it can be determined, if all needed data has been read from a data stream.

We distinguish two types of constraints for fuzzy ordered data streams: static and dynamic constraints. A static constraint is a fixed property of a data stream known in advance. For instance, static constraints are:

value constraint Let r_i be the last read tuple of a data stream R and $max_{y_1} := \max\{r_j.y_1 | 1 \leq j \leq i\}$. Then, this constraint assures that all tuples r' with $r'.y_1 \leq max_{y_1} - c$ have been entirely delivered, with c being a constant depending on the structure of the data stream.

read-ahead constraint The read-ahead constraint defines the number c of tuples which have to be read from the data stream until it can be determined, if all needed tuples have been read yet. E.g., if we want to read all tuples r with $r.y_1 \leq maxRS$, we have to read until c consecutive tuples have y_1-values greater than $maxRS$.

In contrast to that, dynamic constraints are delivered by the data source in the data stream as meta-data interspersed with the data tuples. An example for such a dynamic constraint is the concept of punctuations of data streams proposed by Tucker et. al. [TM02]. A punctuation is a predicate, which is delivered in the data stream like a normal tuple. It means that after a punctuation has been read, no tuples will be contained in the data stream which satisfy that predicate. For instance, after the punctuation "$y_1 \leq c$" has been read, no more tuples with y_1-values less than or equal to c are delivered. Punctuations of a data stream can be arbitrary predicates on any attributes. For our purpose only punctuations concerning the join attribute y_1 in the form of "$y_1 \leq c$" are useful.

Only minor changes to the filling- and materializing-step of the MatWin algorithm (line 9) have to be made to accommodate it to fuzzy ordered data streams. In case of a sorted data stream it is sufficient to read from a data stream given an upper bound $maxRS$ of the data windows until a tuple with $y_1 > maxRS$ has been read. Therewith, the data window contains all necessary tuples and the y_1-value of this "next" tuple outside the current data window, which is needed for the calculation of the new bounds, is known. In case of fuzzy ordered data streams the MatWin algorithm has to continue reading tuples from the stream until the y_1-value of this "next" tuple is definitely known. How many

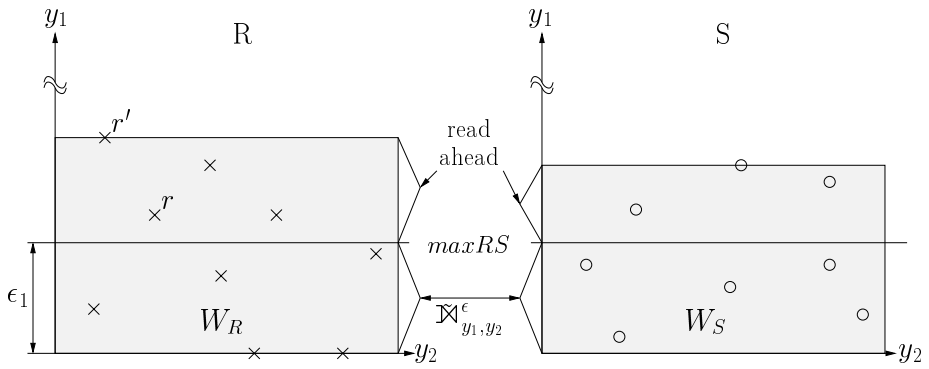


Figure 9: Filling Data Windows on Fuzzy Ordered Data Streams

tuples have to be read ahead is determined by considering the given constraint of the fuzzy ordered stream.

Figure 9 shows this situation for the example of the initial computing step of the MatWin algorithm. R and S shall be fuzzy ordered data streams with the static read-ahead constraint $c = 2$. While filling the data window W_R the tuple r with $r.y_1 > maxRS$ is read. Because of the fuzzy ordered stream, the Materialize()-method continues reading tuples until the read-ahead constraint is satisfied, i.e., until r' is known. Now, it is guaranteed that all necessary tuples for W_R have been delivered and r is the "next" tuple outside the data window. W_S is handled analogously. Dynamic constraints like punctuations can be exploited analogously. For filling a data window with a given upper bound $maxRS$ the data stream is read until a punctuation "$y_1 \leq c$" with $c > maxRS$ is received. Then, all needed tuples are known and the computation can be done as presented in the previous sections.

Obviously, in the presence of fuzzy ordered data streams the data windows may contain tuples lying above the upper bound of the data windows. To avoid multiple processing of these tuples only the tuples within the upper and lower bound of the data windows have to be considered for computing the constrained BMJ operators.

Being applicable on fuzzy ordered data streams the possible field of application of the MatWin algorithm significantly increases. Further, it might be possible that the data sources can be told to deliver the data streams ordered according to the ϵ-Grid-Order. Obviously, such streams are fuzzy ordered data streams and therefore usable by the MatWin algorithm. In this case the sorting steps during the materialization of the data windows can be avoided and the MatWin algorithm becomes even more efficient.

3.5 Dealing with Timestamped Data Streams

In the previous sections we have shown how the constrained BMJ operators can be efficiently processed on (fuzzy ordered) data streams. In this section we propose exemplary application scenarios for computing constrained BMJ-operators on special data streams which we denote as *timestamped data streams*. Timestamped data streams are data streams where each tuple has an associated timestamp t and which are sorted, or at least fuzzy ordered, according to this dimension t. We distinguish two cases with regard

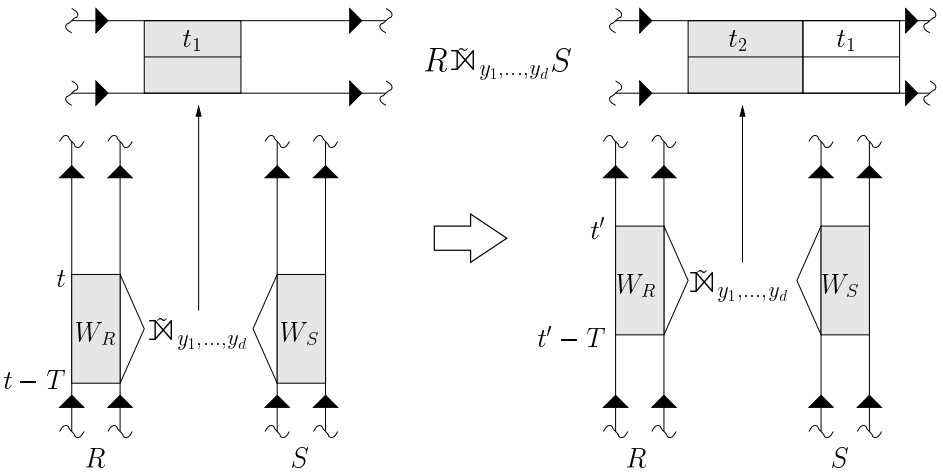


Figure 10: Fixed Time Window Evaluation

to the computation of constrained BMJ-operators on such streams: t is not used as a join attribute and t is a join attribute.

3.5.1 Time as a Non-Join-Attribute

In this case the most common task is to consider all input data given a particular maximum age, but the timestamps of the input tuples do not affect the computation of best matching pairs. For instance, the problem *"Compute* $R \tilde{\bowtie}^{\epsilon}_{y_1,\ldots,y_d} S$ *of the data streams* R *and* S *using the order* $\prec_{y_1,\ldots,y_d}$ *and do not consider any input data older than* T*"* has to be solved.

Therefore, we employ a simple window-based (not the MatWin algorithm) approach as shown in Figure 10. For each data stream a data window is maintained such that for every time all previous tuples of the period T are contained. The upper and lower bounds with regard to the timestamps of the tuples of both data windows are equal. Every time a new tuple with an associated timestamp t is read from a data stream, all tuples with a timestamp older than $t - T$ are discarded from both data windows. The data windows are either materialized, if they are too big for being stored in main memory, or held in main memory. Whenever the data windows change, $W_R \tilde{\bowtie}^{\epsilon}_{y_1,\ldots,y_d} W_S$ is computed, the best matching pairs are annotated with the current time, and propagated to the output data stream. For the computation of the constrained BMJ-operator any algorithm can be used, e.g., the nested-loops algorithm utilizing the ϵ-Grid-Order. We propose this re-evaluation approach, because an algorithm continuously updating the current result of best matching pairs does not produce satisfying results in this situation: Consider a very well matching pair $(r \times s)$ which dominates a pair $(r' \times s')$, whereas r' and s' are younger than both r and s. At a particular point in time, r, s, or both of them are discarded from the data windows, because they are too old. At this time the pair $(r' \times s')$ might be contained in the result, because $(r \times s)$ (which dominated $(r' \times s')$) is not known any longer. An approach which continuously updates the current best pairs will not generate the pair $(r' \times s')$ after $(r \times s)$ has been discarded, because $(r' \times s')$ was already processed and it will not be considered once more.

Of course, fuzzy ordered data streams can be handled analogously as explained in Section 3.4 for the MatWin algorithm.

3.5.2 Time as a Join-Attribute

In the second case the time-dimension is a join attribute and hence influences the computation of best matching pairs. We show how to handle such a situation on the example: *"Compute for each tuple of the data stream R all best matching join mates of the data stream S. Best matching pairs shall be those pairs which match best given a partial order* $\prec_{y_1,\ldots,y_d}$ *(as defined in Section 2.1) not considering the time dimension, as well as pairs being very young, even if they do not match perfectly according to* $\prec_{y_1,\ldots,y_d}$*"*.

To solve this problem we have to define the age of a pair. We consider the age of a pair $(r \times s)$ being $age(r,s) := max(r.t, s.t)$, i.e., a pair is as old as the youngest join mate. In combination with the given partial order $\prec_{y_1,\ldots,y_d}$ a single partial order $\prec$ for comparing pairs of $R \times S$ can be constructed as follows to solve our problem using a left-outer-BMJ-operator (with t_c being the current time):

$$(r \times s) \prec (r \times s') \Leftrightarrow (t_c - age(r,s) \leq t_c - age(r,s')) \ \wedge ((r \times s) \prec_{y_1,\ldots,y_d} (r \times s'))$$

Because of the definition of the age of a pair, a pair consisting of a young and a very old tuple is considered to be young. Again, such pairs are not interesting in real-world applications, because in general we are only interested in young pairs consisting of tuples of roughly the same age. So, we only want to consider pairs $(r \times s)$ which differ not more than a given T in their age, i.e., to be an interesting pair the condition $|r.t - s.t| \leq T$ must hold. Since the data streams are (fuzzy) ordered according to the time-dimension and pairs to be considered have a given maximum distance T with regard to this dimension, the MatWin algorithm is applicable for efficiently computing the constrained left-outer-BMJ. Therewith, our problem is solved and the result contains all pairs for which no younger and better matching pair exists, as well as older pairs which match better than the younger pairs according to $\prec_{y_1,\ldots,y_d}$.

4 Performance Evaluation

In this section we present the results of the benchmarks to show the benefits of our algorithms. All tests were performed using a prototype Java (JDK 1.4) implementation. All I/O operations are based on the `java.nio`-Package. In our experiments we measured the query $R \overset{c}{\bowtie}_{y_1,\ldots,y_d} S$ as an example. As input data we used relations with the schema $\{[data\text{: string}, y_1\text{: double}, \ldots, y_d\text{: double}]\}$, which were delivered as data streams R and S. As before, $y_1, \ldots, y_d$ were the join attributes of the constrained left-outer-BMJ. On each dimension the order *minAttrDist* was utilized for comparison. The values of the join attributes were randomly generated in the range $[0;1]$ using uniform, correlated, and anti-correlated distributions as shown in [BKS01] and a normal distribution. All experiments were carried out on a Sun Enterprise 450 with four 400MHz processors and 4GB of main memory. The implementation is not optimized for parallel processing, so only one processor is utilized. In our tests we varied the size of the inputs, the number

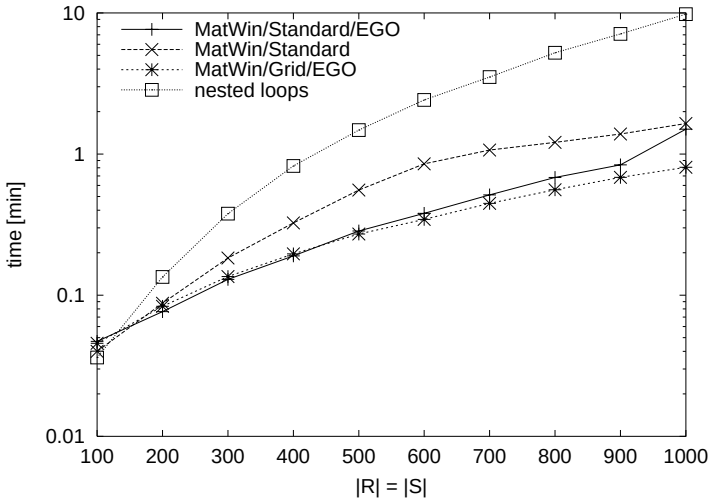


Figure 11: Total Execution Time (2 Dimensions, Uniform Distribution, $\epsilon_i = 0.1$)

of dimensions, the distribution of the join attributes, and the maximum distances ϵ_i for a pair being interesting.

At first, we measured the total execution time of the MatWin algorithm in comparison with the nested-loops algorithm to be able to judge the overall efficiency of the MatWin algorithm. Therefore, we used the 2-dimensional constrained left-outer-BMJ with $\epsilon_i = 0.1$, uniformly distributed values of the join attributes, and varied the size of the input data streams. For the computation of the constrained left-outer-BMJ the MatWin algorithm with/without utilizing the ϵ-Grid-Order using the standard materialization (MatWin/Standard, MatWin/Standard/EGO) and the grid materialization utilizing the ϵ-Grid-Order (MatWin/Grid/EGO) were used. Figure 11 shows the results of this experiment. All measured variants of the MatWin algorithm perform significantly better than the nested-loops algorithm. Utilizing the ϵ-Grid-Order the efficiency can be further increased. The grid materialization performs slightly better than the standard materialization. Figure 12 shows the results of the next experiment. In this test the three MatWin-variants were measured varying the number of dimensions. The sizes of the inputs were fixed to 1000 tuples. The remaining parameters were as before. Obviously, the grid materialization performs better with less dimensions. With an increasing number of dimensions the growth of the grid prevails its benefits and it performs worse. The performance of the standard materialization is not affected by the number of dimensions. Considering the overall efficiency the MatWin/Standard/EGO algorithm outperforms the nested-loops algorithm. Other tests show, that these observations hold varying the distribution of the values of the join attributes and the maximum distances ϵ_i. Of course, the total time of execution can only be determined, if the data streams are finite. But, it has to be kept in mind that the nested-loops algorithm is not applicable to never-ending data streams and thus a comparison of the overall performance is only feasible in this case.

To determine the performance of the MatWin algorithm on infinite data streams the following experiments investigate, how continuous the results are delivered. Therefore, the constrained left-outer-BMJ was computed on data streams of a fixed size of 1000

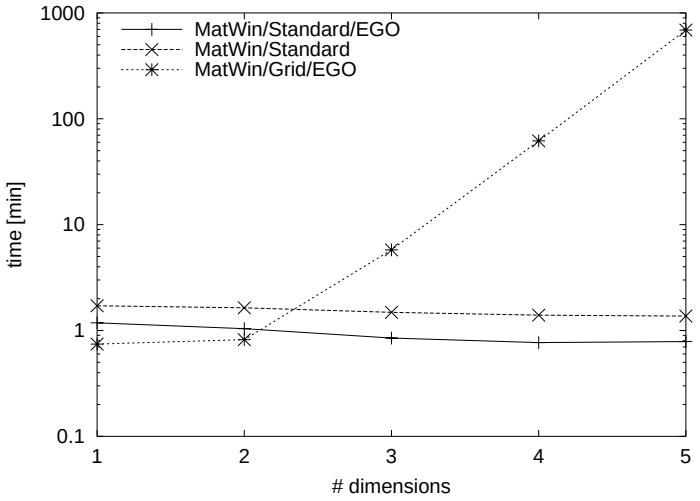


Figure 12: Total Execution Time ($|R| = |S| = 1000$, Uniform Distribution, $\epsilon_i = 0.1$)

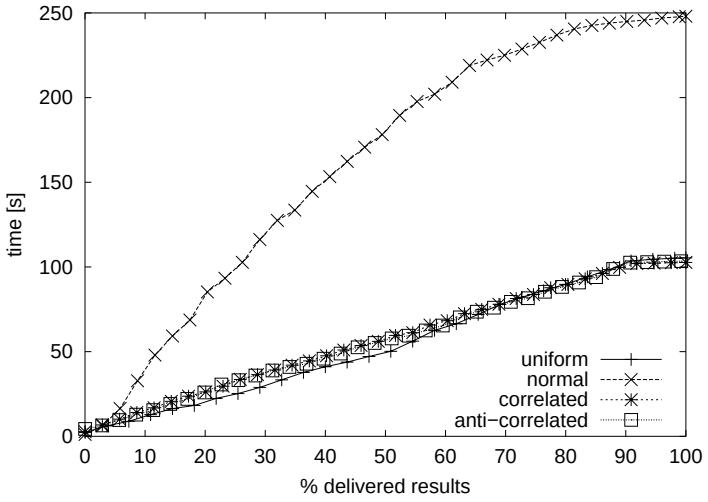


Figure 13: Propagated Results vs. Time ($|R| = |S| = 1000$, 2 Dimensions, $\epsilon_i = 0.1$)

tuples with two join attributes and it was measured what percentage of the result has been propagated at a certain time. The remaining parameters were as mentioned above. Figure 13 depicts the results of this experiment varying the distribution of values of the join attributes. It can be observed that the distribution of the join attributes has no influence on how continuous the results are propagated. The difference in total time is due to the fact that using normally distributed join attributes the result contains more pairs. Thus, more pairs have to be compared during the update of intermediate results and the processing time increases. Figure 14 shows the results of the tests varying the number of distinct values of the join attribute y_1. Therefore, the precision of the values of y_1 was varied in the range of one to five decimal places to investigate, whether the granularity of y_1 influences the propagation of results. It can be observed that the granularity does not influence the continuous propagation of results. An exception to this behavior is the

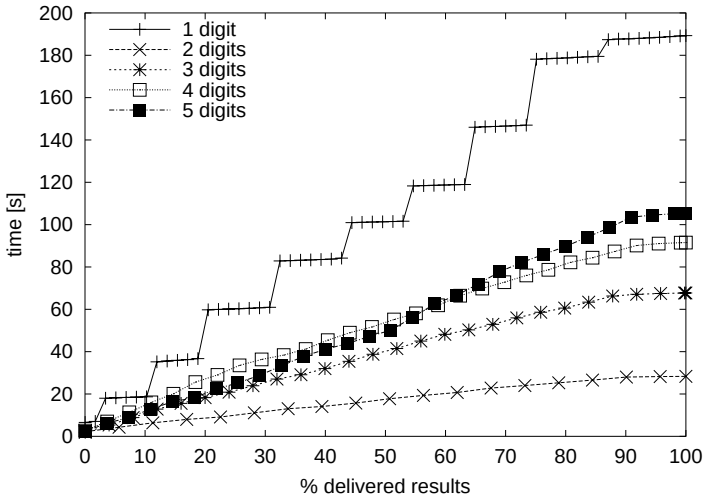


Figure 14: Propagated Results vs. Time ($|R| = |S| = 1000$, 2 Dimensions, Uniform Distribution, $\epsilon_i = 0.1$)

measurement with one decimal place. In this case the precision is equal to $\epsilon_1 = 0.1$. Due to this fact the tuples are always located exactly on the boundaries of the data windows. Whenever the data windows are moved, all tuples of the lower boundary are completely discarded and the results they are involved in are propagated all at once. This explains the steps in the measurement.

On the whole, the MatWin algorithm shows a good overall performance and pipelined behavior in all our experiments.

5 Conclusion

In this paper we introduced the novel class of BMJ-operators and the more relevant family of constrained BMJ-operators to compute best matching pairs of two data sets based on user-defined multi-dimensional criteria. The constraints in combination with physical properties of data streams overcome the blocking nature of the BMJ-operators and enable our new pipelined BMJ-algorithms to process infinite, (fuzzy) ordered data streams. These algorithms are based on synchronously shifting data windows over the data streams and materializing them on secondary storage. We discussed efficient I/O-scheduling methods for processing the materialized data windows based on a particular order of the materialized data. Our experiments showed a good overall performance and pipelining behavior of this approach.

References

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst.*, pages 1–16, Madison, Wisconsin, USA, June 2002.

[BK01] C. Böhm and H.-P. Kriegel. A Cost Model and Index Architecture for the Similarity Join. In *Proc. IEEE Conf. on Data Engineering*, pages 411–420, Heidelberg, Germany, 2001.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proc. IEEE Conf. on Data Engineering*, pages 421–430, Heidelberg, Germany, 2001.

[CCC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebreaker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proc. of the Conf. on Very Large Data Bases*, pages 215–226, Hong Kong, China, August 2002.

[CF02] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *Proc. of the Conf. on Very Large Data Bases*, pages 203–214, Hong Kong, China, August 2002.

[HAK00] A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What Is the Nearest Neighbor in High Dimensional Spaces? In *Proc. of the Conf. on Very Large Data Bases*, pages 506–515, Cairo, Egypt, September 2000.

[ILW+00] Z. G. Ives, A. Y. Levy, D. S. Weld, D. Florescu, and M. Friedman. Adaptive Query Processing for Internet Applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, June 2000.

[Kie02] W. Kießling. Foundations of Preferences in Database Systems. In *Proc. of the Conf. on Very Large Data Bases*, pages 311–322, Hong Kong, China, August 2002.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[KRR02] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Proc. of the Conf. on Very Large Data Bases*, pages 275–286, Hong Kong, China, August 2002.

[MF02] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proc. IEEE Conf. on Data Engineering*, pages 555–566, San Jose, California, 2002.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, Berlin, etc., 1985.

[RH02] V. Raman and J. M. Hellerstein. Partial Results for Online Query Processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Madison, Wisconsin, USA, June 2002.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 71–79, San Jose, CA, USA, May 1995.

[SU00] J.-R. Sack and J. Urrutia, editors. *Handbook of Computational Geometry.* ELSEVIER, Amsterdam, Lausanne, New York, Oxford, Shannon, Signapore, Tokyo, 2000.

[TEO01] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *Proc. of the Conf. on Very Large Data Bases*, pages 301–310, Roma, Italy, September 2001.

[TM02] P. Tucker and D. Maier. Exploiting Punctuation Semantics in Data Streams. In *Proc. IEEE Conf. on Data Engineering*, page 279, San Jose, California, 2002.

[YG02] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM Sigmod Record*, 31(3), September 2002.